МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Юго-Западный государственный университет»
(ЮЗГУ)

Кафедра программной инженерии

УТВЕРЖДАЮ
Проректор по учебной работе
_____ О.Г. Локтионова
«15» 12 2017 г.

РАБОТА С КЛАССАМИ И ОБЪЕКТАМИ В C#

Методические указания
к лабораторным работам по дисциплине
«Языки объектно-ориентированного программирования»
для студентов направления подготовки
09.03.04 «Программная инженерия»

Курск 2017

УДК 681.3

Составитель Е.И.Аникина

Рецензент
Кандидат технических наук, доцент кафедры программной инженерии *Н.Н. Бочанова*

Работа с классами и объектами в C#: методические указания к лабораторным работам по дисциплине «Языки объектно-ориентированного программирования» для студентов направления подготовки 09.03.04 «Программная инженерия»/Юго-Зап. гос. ун-т; сост. Е.И. Аникина. Курск, 2017. 54 с.

Содержит теоретические сведения, примеры решения задач и задания к лабораторным работам по темам дисциплины, связанным с технологией разработки приложений в парадигме объектно-ориентированного подхода.

Предназначено для студентов направления подготовки бакалавров 09.03.04 «Программная инженерия».

Текст печатается в авторской редакции.

Подписано в печать . Формат 60x84 1/16.
Усл. печ. л. . Уч.-изд. л. . Тираж 100 экз. Заказ .
Бесплатно.

Юго-Западный государственный университет
305040, Курск, ул.50 лет Октября, 94.

СОДЕРЖАНИЕ

Лабораторная работа №1

КЛАССЫ И ОБЪЕКТЫ

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке - наличие двух рук, двух ног, головы, пищеварительной, нервной системы, головного мозга и т.д. Есть некоторый шаблон - этот шаблон можно назвать классом. Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса.

Класс определяется с помощью ключевого слова **class**:

```
class Book
{

}
```

Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями. В прошлой главе мы создали структуру Book. Теперь переделаем ее в класс Book:

```
class Book
{
      public string name;
    public string author;
    public int year;

    public void Info()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) была издана
в {2} году", name, author, year);
    }
}
```

Кроме обычных методов в классах используются также и специальные методы, которые называются **конструкторами**. Конструкторы вызываются при создании нового объекта данного

класса. Отличительной чертой конструктора является то, что его название должно совпадать с названием класса:

```
class Book
{
      public string name;
    public string author;
    public int year;

      public Book()
    { }

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

    public void Info()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) была издана
в {2} году", name, author, year);
    }
}
```

Одно из назначений конструктора - начальная инициализация членов класса. В данном случае мы использовали два конструктора. Один пустой. Второй конструктор наполняет поля класса начальными значениями, которые передаются через его параметры.

Поскольку имена параметров и имена полей (name, author, year) в данном случае совпадают, то мы используем ключевое слово **this**. Это ключевое слово представляет ссылку на текущий экземпляр класса. Поэтому в выражении `this.name = name;` первая часть `this.name` означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово `this` было бы необязательно.

Теперь используем класс в программе. Создадим новый проект. Затем нажмем правой кнопкой мыши на название проекта в окне **Solution Explorer (Обозреватель решений)** и в появившемся меню выберем пункт **Class**.

В появившемся диалоговом окне дадим новому классу имя **Book** и нажмем кнопку **Add (Добавить)**. В проект будет добавлен новый файл *Book.cs*, содержащий класс Book.

Изменим в этом файле код класса Book на следующий:

```
class Book
{
      public string name;
    public string author;
    public int year;

      public Book()
    {
            name = "неизвестно";
        author = "неизвестно";
        year = 0;
      }

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

    public void GetInformation()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) была издана
в {2} году", name, author, year);
    }
}
```

Теперь перейдем к коду файла *Program.cs* и изменим метод Main класса Program следующим образом:

```
class Program
{
    static void Main(string[] args)
    {
        Book b1 = new Book("Война и мир", "Л. Н. Толстой",
1869);
        b1.GetInformation();
```

```
        Book b2 = new Book();
        b2.GetInformation();

        Console.ReadLine();
    }
}
```

Если мы запустим код на выполнение, то консоль выведет нам информацию о книгах b1 и b2. Обратите внимание, что чтобы создать новый объект с использованием конструктора, нам надо использовать ключевое слово **new**. Оператор `new` создает объект класса и выделяет для него область в памяти.

Инициализаторы объектов

Для нашего класса Book мы могли бы установить последовательно значения для всех трех полей класса:

```
Book b1 = new Book();
b1.name = "Война и мир";
b1.author = "Л. Н. Толстой";
b1.year = 1869;

b1.GetInformation();
```

Но можно также использовать **инициализатор объектов**:

```
Book b2 = new Book { name = "Отцы и дети", author = "И. С. Тургенев", year = 1862 };
b2.GetInformation();
```

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

ЗАДАНИЕ

Разработать класс, инкапсулирующий двумерный массив. Класс должен содержать поля и методы, необходимые для реализации приведенного ниже задания к ЛР. Номер варианта задания соответствует порядковому номеру студента с списке группы.

Варианты заданий

Вариант 1

Дана вещественная матрица размером 4 строки, 5 столбцов. Переставляя ее строки и столбцы, добейтесь того, чтобы наибольший элемент (один из них) оказался в верхнем левом углу.

Вариант 2

Определите, является ли заданная целочисленная квадратная матрица порядка 5 симметричной относительно главной диагонали.

Вариант 3

Дана вещественная матрица размером 4 строки, 5 столбцов. Поменяйте местами максимальный и минимальный элементы матрицы.

Вариант 4

Дана целочисленная квадратная матрица порядка 5. Найдите максимальный элемент среди элементов, лежащих ниже главной диагонали, и максимальный элемент среди элементов, лежащих выше главной диагонали, поменяйте их местами.

Вариант 5

Дана целочисленная квадратная матрица порядка 5. Найдите максимальный элемент среди элементов, лежащих левее вспомогательной диагонали, и максимальный элемент среди элементов, лежащих правее вспомогательной диагонали, поменяйте их местами.

Вариант 6

Среди строк целочисленной квадратной матрицы порядка 5 найдите строку с минимальной суммой элементов.

Вариант 7

Дана целочисленная квадратная матрица порядка 5. Найдите минимальный элемент среди элементов, лежащих ниже главной диагонали, и максимальный элемент среди элементов, лежащих выше главной диагонали, вычислите их среднее арифметическое.

Вариант 8

Дана целочисленная квадратная матрица порядка 5. Найдите минимальный элемент среди элементов, лежащих левее вспомогательной диагонали, и максимальный элемент среди элементов, лежащих правее вспомогательной диагонали, и вычислить их среднее геометрическое.

Вариант 9

Среди столбцов целочисленной квадратной матрицы порядка 5 найдите столбец с максимальной суммой элементов.

Вариант 10

Среди тех столбцов целочисленной матрицы размером 3 строки, 5 столбцов, которые содержат только такие элементы, значения которых по модулю не превышают 10, найдите столбец с минимальным произведением элементов.

Вариант 11

Даны целые числа $a_1,...,a_{10}$, целочисленная квадратная матрица порядка 5. Замените нулями в матрице те элементы, для которых имеются равные числа среди $a_1,...,a_{10}$.

Вариант 12

В двумерном целочисленном массиве размером 4 строки, 5 столбцов поменяйте местами строки, симметричные относительно середины массива (горизонтальной линии).

Вариант 13

В двумерном целочисленном массиве размером 4 строки, 5 столбцов поменяйте местами столбцы, симметричные относительно середины массива (вертикальной линии).

Вариант 14

Даны две вещественные квадратные матрицы порядка 4. Получите новую матрицу прибавлением к элементам каждого столбца первой матрицы минимального элемента соответствующего столбца второй матрицы.

Вариант 15

В целочисленной квадратной матрице порядка 4 найдите наибольший по модулю элемент. Получите квадратную матрицу порядка 3 путем выбрасывания из исходной матрицы строки и столбца, на пересечении которых расположен элемент с найденным значением.

Вариант 16

В данной вещественной квадратной матрице порядка 5 найдите наименьший элемент. Получите квадратную матрицу порядка 4 путем выбрасывания из исходной матрицы строки и столбца, на пересечении которых расположен элемент с найденным значением.

Лабораторная работа №2

НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Наследование (inheritance) является одним из ключевых моментов ООП. Благодаря наследованию один класс может унаследовать функциональность другого класса.

Пусть у нас есть следующий класс Person, описывающий отдельного человека:

```
1
2
3
4
5
6class Person
7{
8    private string _firstName;
9    private string _lastName;
1
0    public string FirstName
1    {
1        get { return _firstName; }
1        set { _firstName = value; }
2    }
1    public string LastName
3    {
1        get { return _lastName; }
4        set { _lastName = value; }
1    }
5    public void Display()
1    {
6        Console.WriteLine($"{FirstName}
1{LastName}");
7    }
1}
8
1
9
2
0
```

Но вдруг нам потребовался класс, описывающий сотрудника предприятия - класс Employee. Поскольку этот класс будет реализовывать тот же функционал, что и класс Person, так как сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (или наследником, или подклассом) от класса Person, который, в свою очередь, называется базовым классом или родителем (или суперклассом):

```
class Employee
: Person
{

}
```

После двоеточия мы указываем базовый класс для данного класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же свойства, методы, поля, которые есть в классе Person. Единственное, что не передается при наследовании, это конструкторы базового класса.

Таким образом, наследование реализует отношение **is-a** (является), объект класса Employee также является объектом класса Person:

```
static void Main(string[] args)
{
    Person p = new Person { FirstName = "Bill",
LastName = "Gates" };
    p.Display();
    p = new Employee { FirstName = "Denis",
LastName = "Ritchi" };
    p.Display();
    Console.Read();
}
```

И поскольку объект Employee является также и объектом Person, то мы можем так определить переменную: Person p = new Employee().

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

- Не поддерживается множественное наследование, класс может наследоваться только от одного класса. Хотя проблема множественного наследования

реализуется с помощью концепции интерфейсов, о которых мы поговорим позже.

•При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**.

•Если класс объявлен с модификатором **sealed**, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

```
sealed class
Admin
{
}
```

Доступ к членам базового класса из класса-наследника

Вернемся к нашим классам Person и Employee. Хотя Employee наследует весь функционал от класса Person, посмотрим, что будет в следующем случае:

```
class Employee : Person
{
    public void Display()
    {
        Console.WriteLine(_f
irstName);
    }
}
```

Этот код не сработает и выдаст ошибку, так как переменная _firstName объявлена с модификатором privateи поэтому к ней доступ имеет только класс Person. Но зато в классе Person определено общедоступное свойство FirstName, которое мы можем использовать, поэтому следующий код у нас будет работать нормально:

```
class Employee : Person
{
    public void Display()
    {
```

```
5         Console.WriteLine(Fi
6rstName);
7     }
 }
```

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами **public**, **internal**, **protected** и **protected internal**.

Ключевое слово base

Теперь добавим в наши классы конструкторы:

```
1class Person
2{
3     public string FirstName { get;  set; }
4     public string LastName {get; set; }
5
6     public Person(string firstName, string
7lastName)
8     {
9          FirstName = firstName;
1          LastName = lastName;
0     }
1
1     public void Display()
1     {
2          Console.WriteLine($"{FirstName}
1
3{LastName}");
1     }
4}
1
5class Employee : Person
1{
6     public string Company { get; set; }
1
7     public Employee(string firstName, string
1lastName, string comp)
8          : base(firstName, lastName)
1     {
9          Company = comp;
2     }
0}
```

Класс Person имеет стандартный конструктор, который устанавливает два свойства. Поскольку класс Employee наследует и устанавливает те же свойства, что и класс Person, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса Person. К тому же свойств, которые надо установить, и параметров может быть гораздо больше.

С помощью ключевого слова **base** мы можем обратиться к базовому классу. В нашем случае в конструкторе класса Employee нам надо установить имя, фамилию и компанию. Но имя и фамилию мы передаем на установку в конструктор базового класса, то есть в конструктор класса Person, с помощью выражения base(fName, lName).

```
static void Main(string[] args)
{
    Person p = new Person("Bill", "Gates");
    p.Display();
    Employee emp = new Employee ("Tom",
"Simpson", "Microsoft");
    emp.Display();
    Console.Read();
}
```

ЗАДАНИЕ

1. Разработать объектно-ориентированную программу-редактор, позволяющую выводить заданный текст в графической рамке. Рамка отрисовывается символьными примитивами. Предусмотреть минимум 3 типа рамок, реализовать родительский класс для абстрактной рамки, конкретные рамки унаследовать от родительского класса.

2. Разработать объектно-ориентированную программу для конвертации чисел в двоичной системе счисления в десятичную и обратно. Преобразование систем счисления реализовать самостоятельно. Реализовать механизм наследования классов.

3. Разработать объектно-ориентированную программу для конвертации чисел в шестнадцатиричной системе счисления в десятичную и обратно. Преобразование систем счисления

реализовать самостоятельно. Реализовать механизм наследования классов.

4. Разработать объектно-ориентированную программу для конвертации чисел в двоичной системе счисления в шестнадцатиричную и обратно. Преобразование систем счисления реализовать самостоятельно. Реализовать механизм наследования классов.

5. Разработать объектно-ориентированную программу для базовой расстановки кораблей на поле для игры «Морской бой». Предусмотреть 4 типа кораблей: однопалубный, двухпалубный, трехпалубный и четырехпалубный. Корабли не должны соприкасаться между собой даже углами. Для каждого корабля реализовать метод, определяющий, касается ли он другого (заданного) корабля на поле. Реализовать следующие методы проверки: потоплен корабли или нет; подбит или нет. Реализовать механизм наследования классов.

6. Реализовать объектно-ориентированную программу для формирования расстановки мин на поле для игры «Сапер». Реализовать механизм наследования классов.

7. Реализовать объектно-ориентированную программу — редактор информации о сотрудниках предприятия. Сотрудники делятся на рядовых сотрудников и руководителей. Для каждого руководителя определен список подчиненных. Программа должна работать в двух режимах: отображение данных о сотруднике без возможности редактирования и редактирование данных. Для каждого режима работы создать отдельные классы. Реализовать механизм наследования классов.

8. Реализовать объектно-ориентированную программу — редактор информации о студентах вуза. Студенты организованы в группы. У каждой группы имеется список студентов и староста. Программа должна работать в двух режимах: отображение данных о студенте /группе без возможности редактирования и редактирование данных. Для каждого режима работы создать отдельные классы. Реализовать механизм наследования классов.

9. Реализовать объектно-ориентированную программу для отображения на двумерном графике различных классов точек.

Каждый класс точек отображать разными графическими примитивами разного цвета (например, красные круги, желтые треугольники и т. д.). Реализовать механизм наследования классов.

10. Создать класс Point — точка. На его основе создать классы ColoredPoint и Line. На основе класса Line создать класс ColoredLine и класс PolyLine — многоугольник. Все классы должны иметь методы для установки и получения значений всех координат, а также изменения цвета и получения текущего цвета. Написать демонстрационную программу, в которой будет использоваться список объектов этих классов в динамической памяти.

11. Создать класс Vehicle. На его основе реализовать классы Plane, Car и Ship. Классы должны иметь возможность задавать и получать координаты, параметры средств передвижения (цена, скорость, год выпуска). Для самолета должна быть определена высота, для самолета и корабля — количество пассажиров. Для корабля — порт приписки. Написать программу, создающую список объектов этих классов в динамической памяти и демонстрирующую функционал разработанных классов.

12. Реализовать объектно-ориентированный справочник планет Солнечной системы. Справочник должен включать само Солнце, планеты и их спутники. Для каждой планеты указать наименование, фото, массу и радиус, удаленность от Солнца (от родительской планеты — в случае спутников). Для спутников указать родительскую планету. Реализовать механизм наследования классов.

13. Разработать объектно-ориентированную программу для представления дробей: десятичных, обыкновенных и периодических. Предусмотреть конвертацию десятичных и периодических в обыкновенные дроби; обыкновенных в десятичные или периодические. Реализовать механизм наследования классов.

14. Разработать объектно-ориентированную программу для представления и вывода на экран комплексных и экспоненциальных чисел. Реализовать механизм наследования классов.

15. Разработать объектно-ориентированную программу для представления и вывода на экран верхних и нижних индексов символа. Реализовать механизм наследования классов.

16. Разработать объектно-ориентированную программу для базовой расстановки фигур на шахматной доске. Для каждой фигуры запрограммировать правила выполнения ходов. Реализовать механизм наследования классов.

Лабораторная работа №3

АБСТРАКТНЫЕ КЛАССЫ И ИНТЕРФЕЙСЫ

Кроме обычных классов в C# есть **абстрактные классы**. Абстрактный класс похож на обычный класс. Он также может иметь переменные, методы, конструкторы, свойства. Единственное, что при определении абстрактных классов используется ключевое слово **abstract**:

```
abstract class Human
{
    public int Length { get; set; }
    public double Weight { get; set; }
}
```

Но главное отличие состоит в том, что мы **не можем** использовать конструктор абстрактного класса для создания его объекта. Например, следующим образом:

```
Human h = new Human();
```

Зачем нужны абстрактные классы? Допустим, в нашей программе для банковского сектора мы можем определить две основных сущности: клиента банка и сотрудника банка. Каждая из этих сущностей будет отличаться, например, для сотрудника надо определить его должность, а для клиента - сумму на счете. Соответственно клиент и сотрудник будут составлять отдельные классы Client и Employee. В то же время обе этих сущности могут иметь что-то общее, например, имя и фамилию, какую-то другую общую функциональность. И эту общую функциональность лучше вынести в какой-то отдельный класс, например, Person, который описывает человека. То есть классы Employee (сотрудник) и Client (клиент банка) будут производными от класса Person. И так как все объекты в нашей системе будут представлять либо сотрудника банка, либо клиента, то напрямую мы от класса Person создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным:

```
abstract class Person
{
```

```
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Person(string name, string surname)
    {
        FirstName = name;
        LastName = surname;
    }

    public void Display()
    {
        Console.WriteLine(FirstName + " " +
LastName);
    }
}

class Client : Person
{
    public int Sum { get; set; }    // сумма
на счету

    public Client(string name, string surname,
int sum)
        : base(name, surname)
    {
        Sum = sum;
    }
}

class Employee : Person
{
    public string Position { get; set; } //
должность

    public Employee(string name, string
surname, string position)
        : base(name, surname)
    {
            Position = position;
    }
}
```

```
2
8
2
9
3
0
3
1
3
2
3
3
3
4
3
5
3
6
3
7
3
8
```

Затем мы сможем использовать эти классы:

```
Client client = new Client("Tom", "Smith",
1500);
2Employee employee = new Employee ("Bob",
3"Tompson", "Apple");
4client.Display();
employee.Display();
```

Или даже так:

```
Person client = new Client("Tom", "Smith",
1500);
2Person employee = new Employee ("Bob",
"Tompson", "Операционист");
```

Но мы НЕ можем создать объект Person, используя конструктор класса Person:

```
1Person person = new Person
("Bill", "Gates");
```

Однако несмотря на то, что напрямую мы не можем вызвать конструктор класса Person для создания объекта, тем не менее конструктор в абстрактных классах то же может играть важную роль. В частности, в классе Person

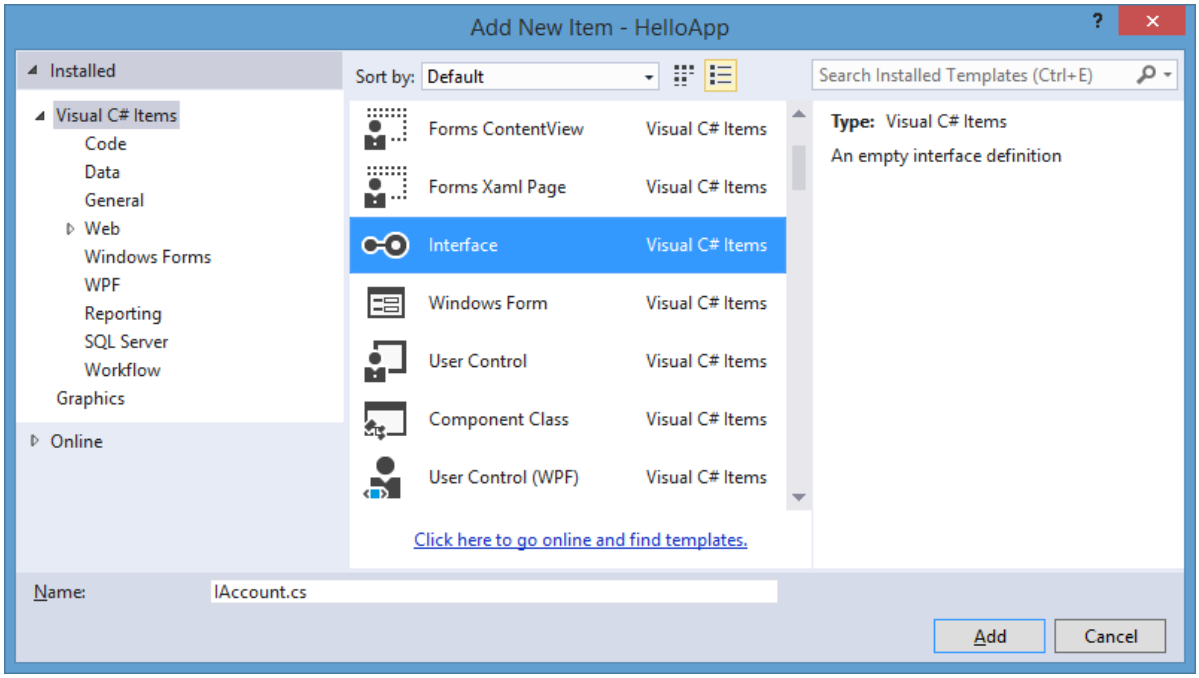
конструктор инициализирует свойства FirstName и LastName. И хотя напрямую он не вызывается, тем не менее производные классы Client и Employee могут обращаться к нему.

Используя механизм наследования, мы можем дополнять и переопределять общий функционал базовых классах в классах-наследниках. Однако напрямую мы можем наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.

В языке C# подобную проблему позволяют решить интерфейсы. Они играют важную роль в системе ООП. Интерфейсы позволяют определить некоторый функционал, не имеющий конкретной реализации. Затем этот функционал реализуют классы, применяющие данные интерфейсы.

Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы **I**, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования. Интерфейсы также, как и классы, могут содержать свойства, методы и события, только без конкретной реализации.

Определим следующий интерфейс IAccount, который будет содержать методы и свойства для работы со счетом клиента. Для добавления интерфейса в проект можно нажать правой кнопкой мыши на проект и в появившемся контекстном меню выбрать **Add**-> **New Item** и в диалоговом окне добавления нового компонента выбрать Interface:



Изменим пустой код интерфейса IAccount на следующий:

```
interface IAccount
{
    // Текущая сумма на счету
    int CurrentSum { get; }
    // Положить деньги на счет
    void Put(int sum);
    // Взять со счета
    void Withdraw(int sum);
}
```

У интерфейса методы и свойства не имеют реализации, в этом они сближаются с абстрактными методами абстрактных классов. Сущность данного интерфейса проста: он определяет свойство для текущей суммы денег на счете и два метода для добавления денег на счет и изъятия денег.

Еще один момент в объявлении интерфейса: все его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

ЗАДАНИЕ

Общее задание для всех вариантов: в программу из ЛР№4 добавить абстрактные классы и/или интерфейсы, и абстрактные методы.

Лабораторная работа №4

СТАТИЧЕСКИЕ МЕТОДЫ И КЛАССЫ

Ранее, чтобы использовать какой-нибудь класс, устанавливать и получать его поля, использовать его методы, мы создавали его объект. Однако если данный класс имеет статические методы, то, чтобы получить к ним доступ, необязательно создавать объект этого класса. Например, создадим новый класс Account, который будет представлять счет в банке:

```
class Account
{
	public Account(decimal sum, decimal rate)
    {
		if (sum < MinSum) throw new Exception("Недопустимая сумма!");
	Sum = sum; Rate = rate;
    }
	private static decimal minSum = 100; // минимальная допустимая сумма для всех счетов
    public static decimal MinSum
    {
		get { return minSum; }
	set { if(value>0) minSum = value; }
    }

	public decimal Sum { get; private set; }    // сумма на счете
	public decimal Rate { get; private set; }   // процентная ставка

	// подсчет суммы на счете через определенный период по определенной ставке
	public static decimal GetSum(decimal sum, decimal rate, int period)
	{
		decimal result = sum;
	for (int i = 1; i <= period; i++)
			result = result + result * rate / 100;
	return result;
	}
}
```

Переменная minSum, свойство MinSum, а также метод GetSum здесь определены с ключевым словом **static**, то есть они являются статическими.

Переменная minSum и свойство MinSum представляют минимальную сумму, которая допустима для создания

счета. Этот оказатель не относится к какому-то конкретному счету, а относится ко всем счетам в целом. Если мы изменим этот показатель для одного счета, то он также должен измениться и для другого счета. То есть в отличии от свойств Sum и Rate, которые хранят состояние объекта, переменная minSum хранит состояние для всех объектов данного класса.

То же самое с методом GetSum - он вычисляет сумму на счете через определенный период по определенной процентной ставке для определенной начальной суммы. Вызов и результат этого метода не зависит от конкретного объекта или его состояния.

Таким образом, переменные и свойства, которые хранят состояние, общее для всех объектов класса, следует определять как статические. И также методы, которые определяют общее для всех объектов поведение, также следует объявлять как статические.

Статические члены класса являются общими для всех объектов этого класса, поэтому к ним надо обращаться по имени класса:

```
Account.MinSum = 560;
decimal result = Account.GetSum(1000, 10, 5);
```

При использовании статических членов класса нам необязательно создавать экземпляр класса, а мы можем обратиться к ним напрямую.

На уровне памяти для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса.

Нередко статические поля применяются для хранения счетчиков. Например, пусть у нас есть класс User, и мы хотим иметь счетчик, который позволял бы узнать, сколько объектов User создано:

```
class User
{
    private static int counter = 0;
    public User()
    {
        counter++;
    }
```

```
        public static void DisplayCounter()
        {
                Console.WriteLine($"Создано {counter} объектов User");
        }
}
class Program
{
        static void Main(string[] args)
        {
                User user1 = new User();
                                User user2 = new User();
                                User user3 = new User();
                                User user4 = new User();
                                User user5 = new User();

                User.DisplayCounter(); // 5

                Console.Read();
        }
}
```

Статический конструктор

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы выполняются при самом первом создании объекта данного класса или первом обращении к его статическим членам (если таковые имеются):

```
class User
{
        static User()
        {
                Console.WriteLine("Создан первый пользователь");
        }
}
class Program
{
        static void Main(string[] args)
        {
                User user1 = new User(); // здесь сработает статический конструктор
                User user2 = new User();

                Console.Read();
        }
}
```

Статические классы

Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы. Например, если бы класс Account имел бы только статические переменные, свойства и методы, то его можно было бы объявить как статический:

```
static class Account
{
        private static decimal minSum = 100; // минимальная допустимая сумма для всех счетов
    public static decimal MinSum
    {
                get { return minSum; }
        set { if(value>0) minSum = value; }
        }

        // подсчет суммы на счете через определенный период по определенной ставке
    public static decimal GetSum(decimal sum, decimal rate, int period)
    {
                decimal result = sum;
        for (int i = 1; i <= period; i++)
                        result = result + result * rate / 100;
        return result;
        }
}
```

В C# показательным примером статического класса является класс Math, который применяется для различных математических операций.

ЗАДАНИЕ

1. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Notepad, содержащий список (использовать контейнер List) объектов класса Note. Количество объектов Note в списке не ограничено. Классы Note и Notepad разработать самостоятельно.
   Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0" ?>
<notepad>
 <note id="1" date="12/04/99" time="13:40">
 <subject>Важная деловая встреча</subject>
 <importance/>
```

```
 <text>
 Надо встретиться с Иваном Ивановичем, предварительно позвонив ему
по телефону <tel>123-12-12</tel>
 </text>
 </note>
 ...
 <note id="2" date="12/04/99" time="13:58">
 <subject>Позвонить домой</subject>
 <text>
 <tel>124-13-13</tel>
 </text>
 </note>
</notepad>
```

2. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Log, содержащий список (использовать контейнер List) объектов класса Event. Количество объектов Event в списке не ограничено. Классы Event и Log разработать самостоятельно.
   Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0"  ?>
<log>
 <event date=" 27/May/1999:02:32:46 " result="success">
 <ip-from> 195.151.62.18 </ip-from>
 <method>GET</method>
 <url-to> /misc/</url-to>
 <response>200</response>
 </event>
 <event date=" 27/May/1999:02:41:47 " result="success">
 <ip-from> 195.209.248.12 </ip-from>
 <method>GET</method>
 <url-to> /soft.htm</url-to>
 <response>200</response>
 </event>
 </log>
```

3. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса PurchaseOrder, содержащий список (использовать контейнер List) объектов класса Item. Количество объектов Item в списке не ограничено. Классы Item и PurchaseOrder разработать самостоятельно.

Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0"?>
<PurchaseOrder PurchaseOrderNumber="99503" OrderDate="1999-10-20">
  <Address Type="Shipping">
    <Name>Ellen Adams</Name>
    <Street>123 Maple Street</Street>
    <City>Mill Valley</City>
    <State>CA</State>
    <Zip>10999</Zip>
    <Country>USA</Country>
  </Address>
  <Address Type="Billing">
    <Name>Tai Yee</Name>
    <Street>8 Oak Avenue</Street>
    <City>Old Town</City>
    <State>PA</State>
    <Zip>95819</Zip>
    <Country>USA</Country>
  </Address>
  <DeliveryNotes>Please leave packages in shed by
driveway.</DeliveryNotes>
  <Items>
    <Item PartNumber="872-AA">
      <ProductName>Lawnmower</ProductName>
      <Quantity>1</Quantity>
      <USPrice>148.95</USPrice>
      <Comment>Confirm this is electric</Comment>
    </Item>
    <Item PartNumber="926-AA">
      <ProductName>Baby Monitor</ProductName>
      <Quantity>2</Quantity>
      <USPrice>39.98</USPrice>
      <ShipDate>1999-05-21</ShipDate>
    </Item>
  </Items>
</PurchaseOrder>
```

4. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Dogs, содержащий список (использовать контейнер List) объектов класса Dog. Количество

объектов Dog в списке не ограничено. Классы Dog и Dogs разработать самостоятельно.
Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0" ?>
 <dogs>
 <dog>
  <dogName>Шарик</dogName>
  <dogWeight caption="кг">18</dogWeight>
  <dogColor>рыжий с черными подпалинами</dogColor>
 </dog>
 <dog>
  <dogName>Тузик</dogName>
  <dogWeight caption="кг">10</dogWeight>
  <dogColor>белый с черными пятнами</dogColor>
 </dog>
 <dog>
  <dogName>Бобик</dogName>
  <dogWeight caption="кг">2</dogWeight>
  <dogColor>бело-серый</dogColor>
 </dog>
 <dog>
  <dogName>Трезор</dogName>
  <dogWeight caption="кг">25</dogWeight>
  <dogColor>черный</dogColor>
 </dog>
 </dogs>
```

5. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Catalog, содержащий список (использовать контейнер List) объектов класса CD. Количество объектов CD в списке не ограничено. Классы CD и Catalog разработать самостоятельно.
   Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0" ?>
<CATALOG>
<CD>
<TITLE>Ниссовый Мед</TITLE>
<ARTIST>Ван Моррисон</ARTIST>
<COUNTRY>Соединенное Королевство</COUNTRY>
```

<COMPANY>Полидор</COMPANY>
<PRICE>8.20</PRICE>
<YEAR>1971</YEAR>
</CD>
<CD>
<TITLE>Midt om natten</TITLE>
<ARTIST>Ким Ларсен</ARTIST>
<COUNTRY>EC</COUNTRY>
<COMPANY>Смесь</COMPANY>
<PRICE>7.80</PRICE>
<YEAR>1983</YEAR>
</CD>
<CD>
<TITLE>Паваротти Гала Концерт</TITLE>
<ARTIST>Лучиано Паваротти</ARTIST>
<COUNTRY>Соединенное Королевство</COUNTRY>
<COMPANY>ДЕККА</COMPANY>
<PRICE>9.90</PRICE>
<YEAR>1991</YEAR>
</CD>
</CATALOG>

6. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Catalog – каталог магазина растений, содержащий список (использовать контейнер List) объектов класса Plant. Количество объектов Plant в списке не ограничено. Классы Plant и Catalog разработать самостоятельно. Написать программу, иллюстрирующую работу созданных классов.

<?xml version="1.0" ?>
<CATALOG>
<PLANT>
<COMMON>Кровавый корень</COMMON>
<BOTANICAL>Sanguinaria canadensis</BOTANICAL>
<ZONE>4</ZONE>
<LIGHT>По большей части тень</LIGHT>
<PRICE>$2.44</PRICE>
<AVAILABILITY>031599</AVAILABILITY>
</PLANT>
<PLANT>
<COMMON>Аквилегия</COMMON>
<BOTANICAL>Aquilegia canadensis</BOTANICAL>

<ZONE>3</ZONE>
<LIGHT>По большей части тень</LIGHT>
<PRICE>$9.37</PRICE>
<AVAILABILITY>030699</AVAILABILITY>
</PLANT>
<PLANT>
<COMMON>Болотная Календула</COMMON>
<BOTANICAL>Caltha palustris</BOTANICAL>
<ZONE>4</ZONE>
<LIGHT>По большей части свет</LIGHT>
<PRICE>$6.81</PRICE>
<AVAILABILITY>051799</AVAILABILITY>
</PLANT>
</CATALOG>

7. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса CaffeeMenu, содержащий список (использовать контейнер List) объектов класса Food. Количество объектов Food в списке не ограничено. Классы Food и CaffeMenu разработать самостоятельно.
   Написать программу, иллюстрирующую работу созданных классов.

<?xml version="1.0" ?>
<breakfast_menu>
<food>
<name>Бельгийские Вафли</name>
<price>$5.95</price>
<description>
две известных Бельгийских Вафли с обилием настоящего кленового сиропа
</description>
<calories>650</calories>
</food>
<food>
<name>Французский Тост</name>
<price>$4.50</price>
<description>
толстые куски, сделанные из кусочков домашнего хлеба из опары
</description>
<calories>600</calories>
</food>
<food>

<name>Домашний Завтрак</name>
<price>$6.95</price>
<description>
пара яиц, бекон или колбаса, тост, и наши всегда популярные картофельные оладьи
</description>
<calories>950</calories>
</food>
</breakfast_menu>

8. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Order, содержащий список (использовать контейнер List) объектов класса Item. Количество объектов Item в списке не ограничено. Классы Item и Order разработать самостоятельно.
   Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0" ?>
<shipOrder>
  <shipTo>
    <name>Tove Svendson</name>
    <street>Ragnhildvei 2</street>
    <address>4000 Stavanger</address>
    <country>Norway</country>
  </shipTo>
  <items>
    <item>
      <title>Empire Burlesque</title>
      <quantity>1</quantity>
      <price>10.90</price>
    </item>
    <item>
      <title>Hide your heart</title>
      <quantity>1</quantity>
      <price>9.90</price>
    </item>
  </items>
</shipOrder>
```

9. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Resume – резюме, содержащий список (использовать контейнер List) объектов класса

ExperienceItem (обязанности в организации). Количество объектов ExperienceItem в списке не ограничено. Классы ExperienceItem и Resume разработать самостоятельно.
Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0"?>
    <resume name="Ilya Voronin" date-of-birth="23.08.1985" email="ivoronin@ivoronin.pp.ru">
      <education>
        <item name="Student of Moscow State University of Applied Biotechnogy"/>
      </education>
      <experience>
        <organisation name="Amatis Media" time="April 2003 - March 2004" job ="System Administrator">
          <item name="System Adminitrator (Debian GNU/Linux, Windows 2000 Server)"/>
      ...
          <item name="Users support"/>
          <item name="Python programming (for Zope WAS)"/>
        </organisation>
      </experience>
    </resume>
```

10. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Languages, содержащий список (использовать контейнер List) объектов класса ProgrammingLanguage. Количество объектов ProgrammingLanguage в списке не ограничено. Классы ProgrammingLanguage и Languages разработать самостоятельно.
    Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0" ?>
<languages>
 <lang name="C">
  <appeared>1972</appeared>
  <creator>Dennis Ritchie</creator>
 </lang>
 <lang name="PHP">
  <appeared>1995</appeared>
  <creator>Rasmus Lerdorf</creator>
 </lang>
```

```
 <lang name="Java">
  <appeared>1995</appeared>
  <creator>James Gosling</creator>
 </lang>
</languages>
```

11. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Cars, содержащий список (использовать контейнер List) объектов класса Car – подержанный автомобиль, выставленный на продажу. Классы Cars и Car разработать самостоятельно.
    Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0"?>
<cars>
<shop id="0">
<model>Volvo</model>
<year>2001</year>
<price>6000</price>
</shop>
<shop id="1">
<model>BMB</model>
<year>2009</year>
<price>17000</price>
</shop>
</cars>
```

12. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Company, содержащий список (использовать контейнер List) объектов класса Staff. Количество объектов Staff в списке не ограничено. Классы Staff и Company разработать самостоятельно.
    Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0"?>
<company>
   <staff id="1001">
      <firstname>Иван</firstname>
      <lastname>Иванов</lastname>
      <nickname>ivanov</nickname>
      <salary>100000</salary>
   </staff>
```

```
   <staff id="2001">
      <firstname>Петр</firstname>
      <lastname>Петров</lastname>
      <nickname>petrov</nickname>
      <salary>200000</salary>
   </staff>
</company>
```

13. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Customers, содержащий список (использовать контейнер List) объектов класса Customer. Количество объектов Customer в списке не ограничено. Классы Customer и Customers разработать самостоятельно.
    Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0"?>
<customers>
  <customer fullname="George Washington" address="The President's House"
            city="New York" state="NY" phone="212-555-1212"/>
  <customer fullname="John Adams" address="The White House"
            city="Washington" state="DC" phone="202-555-1212"/>
  <customer fullname="Thomas Jefferson" address="The White House"
            city="Washington" state="DC" phone="202-555-1212"/>
</customers>
```

14. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса AddressBook, содержащий список (использовать контейнер List) объектов класса Address. Количество объектов Address в списке не ограничено. Классы Address и AddressBook разработать самостоятельно.
    Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0"?>
<addressbook>
   <address>
     <addressee>John Smith</addressee>
     <streetaddress>250 18th Ave SE</streetaddress>
     <city>Rochester</city>
     <state>MN</state>
     <postalCode>55902</postalCode>
```

```
  </address>
  <address>
    <addressee>Bill Morris</addressee>
    <streetaddress>1234 Center Lane NW</streetaddress>
    <city>St. Paul</city>
    <state>MN</state>
    <postalCode>55123</postalCode>
</address>
</addressbook>
```

15. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Realty, содержащий список (использовать контейнер List) объектов класса Location. Количество объектов Location в списке не ограничено. Классы Location и Realty разработать самостоятельно.
    Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0"?>
<realty>
<location>
        <country>Россия</country>
        <region>Московская область</region>
        <district>Одинцовский район</district>
        <locality-name>Одинцово</locality-name>
       <sub-locality-name>Центральный</sub-locality-name>
       <non-admin-sub-locality>Центр</non-admin-sub-locality>
        <address>Пушкинская ул., д. 12</address>
        <direction>Минское шоссе</direction>
</location>
<location>
        <country>Россия</country>
        <region>Московская область</region>
        <district>Истринский район</district>
        <locality-name>деревня Подушкино</locality-name>
        <direction>Рублево-Успенское шоссе</direction>
        <distance>10</distance>
</location>
</realty>
```

16. Разработать статический класс для преобразования приведенного ниже xml документа в объект класса Books, содержащий список (использовать контейнер List) объектов класса Book. Количество

объектов Book в списке не ограничено. Классы Book и Books разработать самостоятельно.

Написать программу, иллюстрирующую работу созданных классов.

```
<?xml version="1.0"?>
<books >
<book ref="www.publisher1.com">
<author>Author1</author>
<name>Name1</name>
</book>
<book ref="www.publisher2.com">
<author>Author2</author>
<name>Name2</name>
<cover>Hard</cover>
</book>
</books>
```

Лабораторная работа №5

ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Иногда при выполнении программы возникают ошибки, которые трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Например, при передачи файла по сети может неожиданно оборваться сетевое подключение. такие ситуации называются **исключениями**. Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция **try...catch...finally**. При возникновении исключения среда CLR ищет блок catch, который может обработать данное исключение. Если такого блока не найдено, то пользователю отображается сообщение о необработанном исключении, а дальнейшее выполнение программы останавливается. И чтобы подобной остановки не произошло, и надо использовать блок try..catch. Например:

```
static void Main(string[] args)
{
        int[] a = new int[4];
        try
        {
                a[5] = 4; // тут возникнет исключение, так как у нас в массиве только
4 элемента
                Console.WriteLine("Завершение блока try");
        }
        catch (Exception ex)
        {
                Console.WriteLine("Ошибка: " + ex.Message);
        }
        finally
        {
                Console.WriteLine("Блок finally");
        }
        Console.ReadLine();
}
```

При использовании блока **try...catch..finally** вначале выполняются все инструкции между операторами try и catch. Если между этими операторами вдруг возникает исключение, то обычный порядок выполнения останавливается и переходит к инструкции catch. В данном случае у нас явно возникнет

исключение в блоке try, так как мы пытаемся присвоить значение шестому элементу массива в то время, как в массиве всего 4 элемента. И дойдя до строки a[5] = 4;, выполнение программы остановится и перейдет к блоку catch

Инструкция **catch** имеет следующий синтаксис: catch (тип_исключения имя_переменной). В нашем случае объявляется переменная ex, которая имеет тип Exception. Но если возникшее исключение не является исключением типа, указанного в инструкции catch, то оно не обрабатывается, а программа просто зависает или выбрасывает сообщение об ошибке.

Однако так как тип Exception является базовым классом для всех исключений, то выражение catch (Exception ex) будет обрабатывать практически все исключения. Вся обработка исключения в нашем случае сводится к выводу на консоль сообщении об исключении, которое в свойстве message класса Exception.

Далее в любом случае выполняется блок finally. Однако этот блок необязательный, и его можно при обработке исключений опускать. Если же в ходе программы исключений не возникнет, то программа не будет выполнять блок catch, сразу перейдет к блоку finally, если он имеется.

Обработка нескольких исключений

При необходимости мы можем разграничить обработку различных типов исключений, включив дополнительные блоки catch:

```
static void Main(string[] args)
{
	try
	{

	}
		catch (FileNotFoundException e)
		{
			// Обработка исключения, возникшего при отсутствии файла
		}
		catch (IOException e)
		{
			// Обработка исключений ввода-вывода
		}
	Console.ReadLine();
}
```

Если у нас возникает исключение определенного типа, то оно переходит к соответствующему блоку catch.

При этом более частные исключения следует помещать в начале, и только потом более общие классы исключений. Например, сначала обрабатывается исключение IOException, и только потом Exception (так как IOException наследуется от класса Exception).

Оператор throw

Чтобы сообщить о выполнении исключительных ситуаций в программе, можно использовать оператор **throw**. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения. Например, в нашей программе происходит ввод строки, и мы хотим, чтобы, если длина строки будет больше 6 символов, возникало исключение:

```
static void Main(string[] args)
{
	try
	{
			string message = Console.ReadLine();
		if (message.Length > 6)
		{
			throw new Exception("Длина строки больше 6 символов");
		}
	}
		catch (Exception e)
		{
			Console.WriteLine("Ошибка: " + e.Message);
		}
	Console.ReadLine();
}
```

Обработка исключений и условные конструкции

Ряд исключительных ситуаций может быть предвиден разработчиком. Например, пусть программа предусматривает ввод числа и вывод его квадрата:

```
static void Main(string[] args)
{
	Console.WriteLine("Введите число");
	int x = Int32.Parse(Console.ReadLine());

	x *= x;
	Console.WriteLine("Квадрат числа: " + x);
```

```
            Console.Read();
}
```

Если пользователь введет не число, а строку, какие-то другие символы, то программа выпадет в ошибку. С одной стороны, здесь как раз та ситуация, когда можно применить блок try..catch, чтобы обработать возможную ошибку. Однако гораздо оптимальнее было бы проверить допустимость преобразования:

```
static void Main(string[] args)
{
        Console.WriteLine("Введите число");
        int x;
        string input = Console.ReadLine();
        if (Int32.TryParse(input, out x))
        {
                x *= x;
                Console.WriteLine("Квадрат числа: " + x);
        }
        else
        {
                Console.WriteLine("Некорректный ввод");
        }
                Console.Read();
}
```

Метод Int32.TryParse() возвращает true, если преобразование можно осуществить, и false - если нельзя. При допустимости преобразования переменная x будет содержать введенное число. Так, не используя try...catch можно обработать возможную исключительную ситуацию.

С точки зрения производительности использование блоков try..catch более накладно, чем применение условных конструкций. Поэтому по возможности вместо try..catch лучше использовать условные конструкции на проверку исключительных ситуаций.

Фильтры исключений

В C# 6.0 (Visual Studio 2015) была добавлена такая функциональность, как фильтры исключений. Они позволяют обрабатывать исключения в зависимости от определенных условий:

```
int x = 1;
int y = 0;
```

```
try
{
        int result = x / y;
}
catch(Exception ex) when (y==0)
{
        Console.WriteLine("y не должен быть равен 0");
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message);
}
```

В данном случае будет выброшено исключение, так как y=0. Здесь два блока catch, но поскольку для первого блока указано условие с помощью ключевого слова **when**, то сработает первый блок catch. Если бы y не было бы равно 0, то сработал бы второй блок catch.

ЗАДАНИЕ

Общее задание для всех вариантов: в программы из ЛР№4 и ЛР №6 добавить обработку исключительных ситуаций.

Лабораторная работа №6

ПЕРЕГРУЗКА ОПЕРАТОРОВ И МЕТОДОВ

Иногда возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определенную версию метода. Такая возможность еше называется **перегрузкой методов** (method overloading).

И в языке C# мы можем создавать в классе несколько методов с одним и тем же именем. Но при этом мы должны учитывать, что методы с одним и тем же именем должны иметь либо разное количество параметров, либо параметры разных типов.

Например, пусть у нас есть следующий класс Calculator:

```
class Calculator
{
    public void Add(int a, int b)
    {
        int result = a + b;
        Console.WriteLine($"Result is
{result}");
    }
    public void Add(int a, int b, int c)
    {
        int result = a + b + c;
        Console.WriteLine($"Result is
{result}");
    }
    public int Add(int a, int b, int c,
int d)
    {
        int result = a + b + c + d;
        Console.WriteLine($"Result is
{result}");
        return result;
    }
    public void Add(double a, double b)
    {
        double result = a + b;
        Console.WriteLine($"Result is
{result}");
```

```
    }
}
```

Здесь представлены четыре разных версии метода Add, то есть определены четыре перегрузки данного метода.

Первые три версии метода отличаются по количеству параметров. Четвертая версия совпадает с первой по количеству параметров, но отличается по их типу. При этом достаточно, чтобы хотя бы один параметр отличался по типу. Поэтому это тоже допустимая перегрузка метода Add.

Стоит отметить, что разные версии метода могут иметь разные возвращаемые значения, как в данном случае третья версия возвращает объект типа int. Однако простое изменение возвращаемого типа у метода еще не является основанием для создания новой версии метода. Опять же новая версия должна отличаться от других по количеству параметров или по их типу.

После определения перегруженных версий мы можем использовать их в программе:

```
class Program
{
    static void Main(string[]
args)
    {
        Calculator calc = new
Calculator();
        calc.Add(1, 2); // 3
        calc.Add(1, 2, 3); // 6
        calc.Add(1, 2, 3, 4); //
10
        calc.Add(1.4, 2.5); // 3.9

        Console.ReadKey();
    }
}
```

Консольный вывод:

```
Result is 3
Result is 6
Result is 10
Result is 3.9
```

Наряду с методами мы можем также перегружать операторы. Например, пусть у нас есть следующий класс Counter:

```
class Counter
{
    public int Value {
get; set; }
}
```

Данный класс представляет некоторый счетчик, значение которого хранится в свойстве Value.

И допустим, у нас есть два объекта класса Counter - два счетчика, которые мы хотим сравнивать или складывать на основании их свойства Value, используя стандартные операции сравнения и сложения:

```
Counter c1 = new Counter {
Value = 23 };
Counter c2 = new Counter {
Value = 45 };

bool result = c1 > c2;
Counter c3 = c1 + c2;
```

Но на данный момент ни операция сравнения, ни операция сложения для объектов Counter не доступны. Эти операции могут использоваться для ряда примитивных типов. Например, по умолчанию мы можем складывать числовые значения, но как складывать объекты комплексных типов - классов и структур компилятор не знает. И для этого нам надо выполнить перегрузку нужных нам операторов.

Перегрузка операторов заключается в определении в классе, для объектов которого мы хотим определить оператор, специального метода:

```
public static возвращаемый_тип operator
оператор(параметры)
{  }
```

Этот метод должен иметь модификаторы **public static**, так как перегружаемый оператор будет использоваться для всех объектов данного класса. Далее идет название возвращаемого типа. Возвращаемый тип представляет тот тип, объекты которого мы хотим получить. К примеру, в результате сложения двух объектов Counter мы ожидаем получить новый объект Counter. А в результате сравнения двух мы хотим получить объект типа bool, который указывает истинно ли условное выражение или ложно. Но в зависимости от задачи возвращаемые типы могут быть любыми.

Затем вместо названия метода идет ключевое слово **operator** и собственно сам оператор. И далее в скобках перечисляются параметры. Бинарные операторы принимают два параметра, унарные - один параметр. И в любом случае один из параметров должен представлять тот тип - класс или структуру, в котором определяется оператор.

Например, перегрузим ряд операторов для класса Counter:

```
class Counter
{
    public int Value { get; set; }

    public static Counter operator +(Counter c1, Counter c2)
    {
        return new Counter { Value = c1.Value + c2.Value };
    }
    public static bool operator >(Counter c1, Counter c2)
    {
        if (c1.Value > c2.Value)
            return true;
        else
            return false;
    }
    public static bool operator <(Counter c1, Counter c2)
    {
        if (c1.Value < c2.Value)
            return true;
        else
            return false;
    }
}
```

Поскольку все перегруженные операторы - бинарные - то есть проводятся над двумя объектами, то для каждой перегрузки предусмотрено по два параметра.

Так как в случае с операцией сложения мы хотим сложить два объекта класса Counter, то оператор принимает два объекта этого класса. И так как мы хотим в результате сложения получить новый объект Counter, то данный класс также используется в качестве возвращаемого типа. Все действия этого оператора сводятся к созданию, нового объекта, свойство Value которого объединяет значения свойства Value обоих параметров:

```
public static Counter operator +(Counter
1c1, Counter c2)
2{
3    return new Counter { Value = c1.Value
4+ c2.Value };
}
```

Также переопределены две операции сравнения. Если мы переопределяем одну из этих операций сравнения, то мы также должны переопределить вторую из этих операций. Сами операторы сравнения сравнивают значения свойств Value и в зависимости от результата сравнения возвращают либо true, либо false.

Теперь используем перегруженные операторы в программе:

```
static void Main(string[] args)
1{
2    Counter c1 = new Counter { Value
3= 23 };
4    Counter c2 = new Counter { Value
5= 45 };
6    bool result = c1 > c2;
7    Console.WriteLine(result); //
8false
9
1    Counter c3 = c1 + c2;
0    Console.WriteLine(c3.Value);  //
123 + 45 = 68
1
1    Console.ReadKey();
2}
```

Стоит отметить, что так как по сути определение оператора представляет собой метод, то этот метод мы также можем перегрузить, то есть создать для него еще одну версию. Например, добавим в класс Counter еще один оператор:

```
public static int operator +(Counter
c1, int val)
{
            return c1.Value + val;
}
```

Данный метод складывает значение свойства Value и некоторое число, возвращая их сумму. И также мы можем применить этот оператор:

```
Counter c1 = new Counter {
Value = 23 };
int d = c1 + 27; // 50
Console.WriteLine(d);
```

Следует учитывать, что при перегрузке не должны изменяться те объекты, которые передаются в оператор через параметры. Например, мы можем определить для класса Counter оператор инкремента:

```
public static Counter operator
++(Counter c1)
{
    c1.Value += 10;
    return c1;
}
```

Поскольку оператор унарный, он принимает только один параметр - объект того класса, в котором данный оператор определен. Но это **неправильное** определение инкремента, так как оператор не должен менять значения своих параметров.

И более корректная перегрузка оператора инкремента будет выглядеть так:

```
public static Counter operator
++(Counter c1)
{
    return new Counter { Value =
c1.Value + 10 };
}
```

То есть возвращается новый объект, который содержит в свойстве Value инкрементированное значение.

При этом нам не надо определять отдельно операторы для префиксного и для постфиксного инкремента (а также декремента), так как одна реализация будет работать в обоих случаях.

Например, используем операцию префиксного инкремента:

```
Counter counter = new Counter() {
Value = 10 };
Console.WriteLine($"{counter.Value}");      // 10
Console.WriteLine($"{(++counter).Value}");  // 20
Console.WriteLine($"{counter.Value}");      // 20
```

Консольный вывод:

```
10
20
20
```

Теперь используем постфиксный инкремент:

```
Counter counter = new Counter() {
Value = 10 };
Console.WriteLine($"{counter.Value}");      // 10
Console.WriteLine($"{(counter++).Value}");  // 10
Console.WriteLine($"{counter.Value}");      // 20
```

Консольный вывод:

```
10
10
20
```

Также стоит отметить, что мы можем переопределить операторы **true** и **false**. Например, определим их в классе Counter:

```
class Counter
{
    public int Value { get; set; }

    public static bool operator true(Counter c1)
    {
        return c1.Value != 0;
    }
    public static bool operator false(Counter c1)
    {
        return c1.Value == 0;
    }

    // остальное содержимое класса
}
```

Эти операторы перегружаются, когда мы хотим использовать объект типа в качестве условия. Например:

```
Counter counter = new Counter() { Value = 0 };
if (counter)
    Console.WriteLine(true);
else
    Console.WriteLine(false);
```

При перегрузке операторов надо учитывать, что не все операторы можно перегрузить. В частности, мы можем перегрузить следующие операторы:

- унарные операторы +, -, !, ~, ++, --
- бинарные операторы +, -, *, /, %
- операции сравнения ==, !=, <, >, <=, >=
- логические операторы &&, ||
- операторы присваивания +=, -=, *=, /=, %=

И есть ряд операторов, которые нельзя перегрузить, например, операцию равенства = или тернарный оператор ?:, а также ряд других.

Полный список перегружаемых операторов можно найти в документации msdn

ЗАДАНИЕ

1. Разработать класс, инкапсулирующий дату (день, месяц, год). Перегрузить операторы + (количество дней), - (количество дней), == (объект даты), != (объект даты). Реализовать методы вывода даты в разных форматах.

2. Разработать класс, инкапсулирующий время (часы, минуты, секунды). Перегрузить операторы + (количество секунд), + (объект время), - (количество дней), - (объект время), == (объект время), != (объект время). Реализовать методы вывода времени в разных форматах.

3. Разработать класс, инкапсулирующий комплексные числа. Перегрузить операторы +, -, *, /, ==, !=.

4. Разработать класс, инкапсулирующий простые дроби. Перегрузить операторы +, -, *, /, ==, !=. Реализовать метод упрощения дроби.

5. Разработать классы Point (точка на плоскости) и Points (множество точек на плоскости). Перегрузить операции + (точка) — добавляет точку в множество, - (точка) — убирает точку из множества, если она в нем присутствует, + (множество точек) — объединение множеств, - (множество точек) — вычитание множеств, == - для точки и для множества, != - для точки и для множества.

6. Разработать классы Point (точка в пространстве) и Points (множество точек в пространстве). Перегрузить операции + (точка) — добавляет точку в множество, - (точка) — убирает точку из множества, если она в нем присутствует, + (множество точек) — объединение множеств, - (множество точек) — вычитание множеств, == - для точки и для множества, != - для точки и для множества.

7. Разработать классы Point (точка на плоскости) и Points (множество точек на плоскости). Перегрузить операции + (точка), - (точка) – смещение множества на заданный вектор, * (скаляр) — для точки и для множества, == - для точки и для множества, != - для точки и для множества.

8. Разработать классы Point (точка в пространстве) и Points (множество точек в пространстве). Перегрузить операции + (точка), - (точка) – смещение множества на заданный вектор, * (скаляр) — для точки и для множества, == - для точки и для множества, != - для точки и для множества.

9. Разработать класс, инкапсулирующий матрицу произвольной размерности. Перегрузить операторы +, * (скаляр), * (матрица), ==, !=.

10. Разработать класс строк на основе массива символов. Перегрузить операторы +, ==, !=.

11. Разработать класс, инкапсулирующий двумерный вектор. Перегрузить операторы +, -, * (скаляр), ==, !=.

12. Разработать класс, инкапсулирующий трехмерный вектор. Перегрузить операторы +, -, * (скаляр), ==, !=.

13. Разработать класс, инкапсулирующий вектор произвольной размерности. Перегрузить операторы +, * (скаляр), * (вектор), ==, !=.

14. Разработать класс, инкапсулирующий количество (вещественное число) с единицей измерения. Перегрузить операторы +, -, ==, !=. Разработать статический класс для конвертации единиц измерения.

15. Разработать классы Student (ФИО, группы) и Students (список студентов). В Students перегрузить операторы + (Student), - (Student). В Student перегрузить операторы == и
!=.

16. Разработать классы Book (автор, название, isbn) и Books (список книг). В Books перегрузить операторы + (Book), - (Book). В Book перегрузить операторы == и
!=.