

Приднестровский государственный университет им. Т.Г. Шевченко

физико-математический факультет

кафедра прикладной математики и информатики

ЛАБОРАТОРНАЯ РАБОТА № 5

по дисциплине:
«Системы программирования»

Тема:
«Структуры и классы»

РАЗРАБОТАЛИ:

ст. преподаватель кафедры ПМИИ
Великодный В.И.

ст. преподаватель кафедры ПМИИ
Калинкова Е.В.

Цель работы:

Изучение способов реализации структур и классов в языке C#, получение практических навыков решения задач с использованием структур и классов.

Теоретическая часть

Структуры

Структуры объявляются при помощи ключевого слова `struct`. Общая форма объявления структуры:

```
struct имя
{
    объявления членов
}
```

Например:

```
struct Book    // объявление структуры
{
    // поля структуры
    public string name;
    public string author;
    public int year;
}
```

Объект структуры может быть создан с помощью оператора `new`. В этом случае вызывается конструктор, используемый по умолчанию. Например:

```
Book b = new Book();    // создание объекта с инициализацией полей значениями по
                        // умолчанию (например, для числовых данных - это число 0)
```

Для доступа к членам структуры используется оператор “точка” (`.`). Общий формат этого оператора имеет вид:

объект.член

Например:

```
b.year = 1998;
```

В общем случае оператор “точка” можно использовать для доступа как к переменным экземпляров, так и методам.

Так как структуры – значимый тип данных, их экземпляры можно создавать без ключевого слова `new`. Например:

```
Book b;                // создание объекта
b.name = "BookName";   // инициализация поля name
```

Структуры используют большую часть того же синтаксиса, что и классы, однако они более ограничены по сравнению с ними.

- В объявлении структуры поля не могут быть инициализированы за исключением случаев, когда они объявлены как константы или статические.
- Структура не может объявлять используемый по умолчанию конструктор (конструктор без параметров), но может объявлять конструкторы, имеющие параметры.
- В отличие от классов, структуры можно создавать без использования оператора `new`.
- Структуры являются типами значений, а классы – ссылочными типами.

- Структуры копируются при присваивании. При присваивании структуры новой переменной выполняется копирование всех данных, а любое изменение новой копии не влияет на данные в исходной копии.
- Структура не может быть потомком другой структуры или класса и не может быть основой для других классов. Все структуры – потомки `System.ValueType`, который, в свою очередь, является потомком `System.Object`.

Структуру можно объявить как внутри пространства имен, так и внутри класса, но не внутри метода.

Пример. В приведенном ниже примере программы демонстрируется применение структуры для хранения информации о книге.

```
using System;

namespace Structures
{
    struct Book    // объявление структуры
    {
        // поля структуры
        public string name;
        public string author;
        public int year;

        // метод для вывода информации о книге
        public void PrintInfo()
        {
            Console.WriteLine("Книга '{0}' (автор {1}) была издана в {2} году",
name, author, year);
        }
    }

    class Program
    {
        static void Main()
        {
            Book book;    // создание объекта

            // инициализация полей объекта book
            book.name = "Война и мир";
            book.author = "Л. Н. Толстой";
            book.year = 1983;

            // вывод информации о книге book на экран (вызов метода PrintInfo)
            book.PrintInfo();

            Console.ReadLine();
        }
    }
}
```

Кроме обычных методов структура может содержать специальный метод – конструктор, который выполняет некую начальную инициализацию объекта, т.е. присваивает всем полям некоторые значения.

Вызов конструктора происходит при создании объекта операцией `new`. При этом конструктору можно передать параметры:

```
new название_структуры ([список_параметров])
```

Для конструктора не указывается тип возвращаемого значения, и название конструктора всегда совпадает с именем структуры.

Пример. Применение структуры с конструктором.

```
using System;
```

```
namespace Structures
```

```
{
```

```
    struct Book
```

```
    {
```

```
        public string name;
```

```
        public string author;
```

```
        public int year;
```

```
        // конструктор с параметрами
```

```
        public Book(string n, string a, int y)
```

```
        {
```

```
            name = n;
```

```
            author = a;
```

```
            year = y;
```

```
        }
```

```
        // метод для вывода информации о книге
```

```
        public void PrintInfo()
```

```
        {
```

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

```
        }
```

```
    }
```

```
class Program
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        // вызов явно заданного конструктора
```

```
        Book book = new Book("Война и мир", "Л. Н. Толстой", 1983);
```

```
        // вызов метода PrintInfo
```

```
        book.PrintInfo();
```

```
        Console.ReadLine();
```

```
    }
```

```
}
```

```
}
```

Структура обладает более скромными возможностями, чем класс. Структуры относятся к значимым типам, и поэтому ими можно оперировать непосредственно, а не по ссылке. Следовательно, для работы со структурой вообще не требуется переменная ссылочного типа. Работа со структурой не приводит к ухудшению производительности, столь характерному для обращения к объекту класса.

Классы

Классы и объекты

Класс представляет собой шаблон, описывающий некоторую группу схожих объектов. В нем указываются данные, характеризующие объект, и код, который будет оперировать этими данными. Объекты, строящиеся на основе этого шаблона, называются *экземплярами* класса.

Важно подчеркнуть, что класс является логической абстракцией. Описание класса само по себе не создает объект.

При определении класса описываются данные и код. Если самые простые классы могут содержать только код или только данные, то большинство настоящих классов содержит и то и другое.

Данные содержатся в *членах данных*, определяемых классом, а код – в *функциях-членах*.

Данные-члены – это те члены, которые содержат данные класса – поля, константы, события. Данные-члены могут быть статическими (`static`). Член класса является членом экземпляра, если только он не объявлен явно как `static`.

Функции-члены – это члены, которые обеспечивают некоторую функциональность для манипулирования данными класса. Они включают методы, свойства, конструкторы, финализаторы, операции и индексаторы.

Класс создается с помощью ключевого слова `class`. Ниже приведена общая форма определения простого класса, содержащая только переменные экземпляра и методы:

```
class имя_класса {
    // Объявление переменных экземпляра
    доступ тип переменная1;
    доступ тип переменная2;
    //...
    доступ тип переменнаяN;

    // Объявление методов
    доступ возвращаемый_тип метод1 (параметры) {
        // тело метода
    }
    доступ возвращаемый_тип метод2 (параметры) {
        // тело метода
    }
    //...
    доступ возвращаемый_тип методN (параметры) {
        // тело метода
    }
}
```

Для создания экземпляра некоторого класса используется следующая конструкция:

```
имя_класса имя_переменной = new имя_класса();
```

Пример.

```
using System;

class Book
{
    // поля класса
    public string name;
    public string author;
    public int year;

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

```

class Program
{
    static void Main()
    {
        // создаем объект типа Book
        Book b1 = new Book();

        // инициализируем поля объекта b1
        b1.name = "Война и мир";
        b1.author = "Л.Н. Толстой";
        b1.year = 1983;

        // выводим информацию в консоль
        b1.PrintInfo();
        Console.ReadLine();
    }
}

```

В данном примере определяется пользовательский класс `Book`, который содержит 3 поля и 1 метод, которые являются открытыми (т.е. содержат модификатор доступа `public`). Открытые члены отличаются тем, что доступны из других классов. В методе `Main()` создается один экземпляр этого класса: `b1`. Затем инициализируются поля данных экземпляра и вызывается метод `PrintInfo()`.

Конструкторы

Конструктор инициализирует объект при его создании. У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу. Но у конструкторов нет возвращаемого типа, указываемого явно. Ниже приведена общая форма конструктора:

```

доступ имя_класса (список_параметров)
{
    тело конструктора
}

```

Как правило, конструктор используется для задания первоначальных значений переменных экземпляра, определенных в классе, или же для выполнения любых других установочных процедур, которые требуются для создания полностью сформированного объекта. Кроме того, доступ обычно представляет собой модификатор доступа типа `public`, поскольку конструкторы зачастую вызываются в другом классе. А `список_параметров` может быть как пустым, так и состоящим из одного или более указываемых параметров.

У всех классов имеются конструкторы, независимо от того, определены они явно или нет. Если конструктор не объявлен, в C# автоматически создается конструктор этого класса, используемый по умолчанию и инициализирующий все переменные экземпляра их значениями по умолчанию. Для большинства типов данных значением по умолчанию является нулевое, для типа `bool` – значение `false`, а для ссылочных типов – пустое значение (`null`). Но как только вы определите свой собственный конструктор, конструктор по умолчанию больше не используется.

Пример.

```

using System;

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

```

```

public Book(string Name, string Author, int Year) // конструктор с параметрами
{
    name = Name;
    author = Author;
    year = Year;
}

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

class Program
{
    static void Main()
    {
        Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1983);
        Book b2 = new Book("Горе от ума", "А. С. Грибоедов", 1995);
        b1.PrintInfo();
        b2.PrintInfo();
        Console.ReadLine();
    }
}

```

Результат выполнения программы:

Книга 'Война и мир' (автор Л. Н. Толстой) была издана в 1983 году
Книга 'Горе от ума' (автор А. С. Грибоедов) была издана в 1995 году

Ключевое слово this

В языке C# имеется ключевое слово `this`, которое обеспечивает доступ к текущему экземпляру класса. Одно из возможных применений ключевого слова `this` состоит в том, чтобы разрешать неоднозначность контекста, которая может возникнуть, когда входящий параметр назван так же, как поле данных данного типа.

Пример.

```

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

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

```

Несколько конструкторов

В классе можно указывать несколько конструкторов, главное чтобы они отличались сигнатурами. **Сигнатура**, в случае конструкторов, – это порядок типов аргументов. Например, нельзя создать два конструктора, которые принимают два аргумента типа `int`. В противном случае возникнет неоднозначность при вызове.

Пример использования нескольких конструкторов:

```
using System;

class Car
{
    public double mileage; // пробег
    public double fuel;    // количество топлива

    public Car() // конструктор без параметров
    {
        mileage = 0;
        fuel = 0;
    }

    public Car(double mileage, double fuel) // конструктор с параметрами
    {
        this.mileage = mileage;
        this.fuel = fuel;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Car car1 = new Car(); //создаем автомобиль с параметрами по умолчанию, 0 и 0
        Car car2 = new Car(100, 50); // создаем автомобиль с указанными параметрами
    }
}
```

Инициализаторы объектов

Инициализаторы объектов предоставляют способ создания объекта и инициализации его полей и свойств. Если используются инициализаторы объектов, то вместо обычного вызова конструктора класса указываются имена полей или свойств, инициализируемых первоначально задаваемым значением. Следовательно, синтаксис инициализатора объекта предоставляет альтернативу явному вызову конструктора класса.

Общая форма синтаксиса инициализации объектов:

```
new имя_класса {имя = выражение, имя = выражение, ...}
```

где *имя* обозначает имя поля или свойства, т.е. доступного члена класса, на который указывает *имя_класса*. А *выражение* обозначает инициализирующее выражение, тип которого, конечно, должен соответствовать типу поля или свойства.

Инициализаторы объектов обычно не используются в именованных классах, хотя это вполне допустимо. Вообще, при обращении с именованными классами используется синтаксис вызова обычного конструктора.

Пример использования инициализаторов объекта:

```
using System;

class autoCar
{
    public string marka;
    public short year;
}
```



```

class Program
{
    static void Main()
    {
        // используем инициализаторы
        autoCar myCar = new autoCar { marka = "Renault", year = 2004 };
        Console.ReadLine();
    }
}

```

Модификаторы доступа, инкапсуляция и свойства

Модификаторы доступа

Управление доступом в языке C# организуется с помощью четырех модификаторов доступа: `public`, `private`, `protected` и `internal`.

Когда член класса обозначается спецификатором `public`, он становится доступным из любого другого кода в программе, включая и методы, определенные в других классах. Когда же член класса обозначается спецификатором `private`, он может быть доступен только другим членам этого класса. Следовательно, методы из других классов не имеют доступа к закрытому члену (`private`) данного класса. Если ни один из спецификаторов доступа не указан, член класса считается закрытым для своего класса по умолчанию. Поэтому при создании закрытых членов класса спецификатор `private` указывать для них необязательно.

С помощью модификатора доступа `protected` обеспечивается создание защищенного члена класса, доступ к которому открыт в пределах иерархии классов. А модификатор `internal` служит в основном для ограничения видимости членов текущей сборки (в частности, проектом).

Инкапсуляция

Инкапсуляция – это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных.

Основной единицей инкапсуляции в C# является класс, который определяет форму объекта.

Свойства инкапсуляции:

- совместное хранение данных и функций (методов);
- сокрытие внутренней информации от пользователя;
- изоляция пользователя от особенностей реализации.

Для реализации инкапсуляции в C# используются модификаторы доступа (`private`, `public`...). Применение модификаторов доступа типа `private` защищает переменную от внешнего доступа.

Для управления доступом во многих языках программирования используются специальные методы – геттеры и сеттеры. В C# их роль, как правило, выполняют свойства.

Свойства

Свойство в C# – это член класса, который предоставляет удобный механизм доступа к полю класса (чтение поля и запись). Свойство представляет собой что-то среднее между полем и методом класса. При использовании свойства, мы обращаемся к нему, как к полю класса, но на самом деле компилятор преобразовывает это обращение к вызову соответствующего неявного метода. Такой метод называется **аксессор** (accessor). Существует два таких метода: **get** (для получения данных) и **set** (для записи). Объявление простого свойства имеет следующую структуру:

```

[модификатор доступа] возвращаемый_тип имя_свойства
{
    get
    {
        // тело аксесора для чтения из поля
    }

    set
    {
        // тело аксесора для записи в поле
    }
}

```

Например:

```

class Person
{
    private string name;           //объявление закрытого поля

    public string Name             //объявление свойства
    {
        get                       // аксесор чтения поля
        {
            return name;
        }

        set                       // аксесор записи в поле
        {
            name = value;
        }
    }
}

```

Здесь у нас есть закрытое поле `name` и есть общедоступное свойство `Name`. Хотя они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия у них могут быть произвольные и не обязательно должны совпадать.

Через это свойство мы можем управлять доступом к переменной `name`. Стандартное определение свойства содержит блоки **get** и **set**. В блоке **get** мы возвращаем значение поля, а в блоке **set** устанавливаем. Параметр `value` представляет передаваемое значение.

Свойства позволяют вложить дополнительную логику, которая может быть необходима, например, при присвоении переменной класса какого-либо значения. Например, нам надо установить проверку по возрасту:

```

class Person
{
    private int age;

    public int Age
    {
        get
        {
            return age;
        }
        set
        {
            if (value < 18)
                Console.WriteLine("Возраст должен быть больше 18");
            else
                age = value;
        }
    }
}

```

```

    }
}

class Program
{
    static void Main()
    {
        Person p = new Person();
        p.Age = 20;      // записываем в поле, используя аксессор set
        Console.WriteLine(p.Age); // читаем поле, используя аксессор get
        Console.ReadLine();
    }
}

```

Блоки `set` и `get` не обязательно одновременно должны присутствовать в свойстве. Например, мы можем закрыть свойство от установки, чтобы только можно было получать значение. Для этого опускаем блок `set`. И, наоборот, можно удалить блок `get`, тогда можно будет только установить значение, но нельзя получить:

```

class Person
{
    private string name;
    // свойство только для чтения
    public string Name
    {
        get
        {
            return name;
        }
    }

    private int age;
    // свойство только для записи
    public int Age
    {
        set
        {
            age = value;
        }
    }
}

```

Мы можем применять модификаторы доступа не только ко всему свойству, но и к отдельным блокам – либо `get`, либо `set`. При этом если мы применяем модификатор к одному из блоков, то к другому мы уже не можем применить модификатор:

```

class Person
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }

        private set
        {

```

```

        name = value;
    }
}

public Person(string name)
{
    Name = name;
}
}

```

Теперь закрытый блок `set` мы сможем использовать только в данном классе – в его методах, свойствах, конструкторе, но никак не в другом классе:

```

Person p = new Person("Tom");
// Ошибка - set объявлен с модификатором private
// p.Name = "John";
Console.WriteLine(p.Name);

```

Автоматические свойства

Свойства управляют доступом к полям класса. Однако, если у нас с десяток и более полей, то определять каждое поле и писать для него одноименное свойство было бы утомительно. Поэтому в C# 3.0 были добавлены автоматические свойства. Они имеют сокращенное объявление:

```

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

```

На самом деле тут также создаются поля для свойств, только их создает не программист в коде, а компилятор автоматически генерирует при компиляции.

С одной стороны, автоматические свойства довольно удобны. С другой стороны, стандартные свойства имеют ряд преимуществ: например, они могут инкапсулировать дополнительную логику проверки значения.

Может возникнуть вопрос, в чем разница между простыми открытыми полями и автоматическими свойствами. У таких свойств остается возможность делать их только на чтение или только на запись. Для этого уже используется модификатор доступа `private` перед именем аксессуара:

```

public string Name { get; private set; } // свойство только на чтение
public string Name { private get; set; } // свойство только на запись

```

Перегрузка методов и операторов

В C# допускается совместное использование одного и того же имени двумя или более методами одного и того же класса, при условии, что их параметры объявляются по-разному. В этом случае говорят, что методы перегружаются, а сам процесс называется **перегрузкой методов**. Перегрузка методов относится к одному из способов реализации полиморфизма в C#.

В общем, для перегрузки метода достаточно объявить разные его варианты, а об остальном позаботится компилятор. Но при этом необходимо соблюсти следующее важное условие: *тип или число параметров у каждого метода должны быть разными*.

Совершенно недостаточно, чтобы два метода отличались только типами возвращаемых значений. Они должны также отличаться типами или числом своих параметров. (Во всяком случае, типы возвращаемых значений дают недостаточно сведений компилятору C#, чтобы решить, какой именно метод следует использовать.) Разумеется, перегружаемые методы могут отличаться и типами возвращаемых значений. Когда вызывается перегружаемый метод, то выполняется тот его вариант, параметры которого соответствуют (по типу и числу) передаваемым аргументам.

Пример того, как может быть перегружен метод:

```
public void SomeMethod()
{
    // тело метода
}
public void SomeMethod(int a) // от первого отличается наличием параметра
{
    // тело метода
}
public void SomeMethod(string s) // от второго отличается типом параметра
{
    // тело метода
}
public int SomeMethod(int a, int b) // от предыдущих отличается количеством
                                   // параметров (плюс изменен тип возврата)
{
    // тело метода
    return 0;
}
```

Рассмотрим пример использования перегрузки методов:

```
using System;

class UserInfo
{
    // Перегружаем метод ui
    public void ui()
    {
        Console.WriteLine("Пустой метод\n");
    }

    public void ui(string Name, string Family)
    {
        Console.WriteLine("Имя пользователя: {0}\nФамилия пользователя: {1}",
Name, Family);
    }

    public void ui(string Name, string Family, byte Age)
    {
        Console.WriteLine("Имя пользователя: {0}\nФамилия пользователя:
{1}\nВозраст: {2}", Name, Family, Age);
    }
}

class Program
{
```

```

static void Main(string[] args)
{
    UserInfo user1 = new UserInfo();
    // Разные реализации вызова перегружаемого метода
    user1.ui();
    user1.ui("Ерохин", "Александр", 26);
    Console.ReadLine();
}
}

```

Наряду с методами мы можем также перегружать операторы.

При перегрузке оператора мы указываем модификаторы `public static`, так как перегружаемый оператор будет использоваться для всех объектов данного класса, далее идет название возвращаемого типа, и после него ключевое слово `operator`. Затем название оператора и параметры:

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

Посмотрим на примере. У нас есть класс `State` (государство). Но вдруг нам понадобится в задаче объединять государства, тогда эту задачу нам может облегчить перегрузка оператора `+`. Кроме того, предусмотрим операторы сравнения `>` и `<`, с помощью которых мы будем сравнивать два государства по площади:

```

class State
{
    public string Name;           // название
    public int Population;        // население
    public double Area;           // площадь

    public State(string n, int p, double a) // конструктор с параметрами
    {
        Name = n;
        Population = p;
        Area = a;
    }

    public static State operator +(State s1, State s2)
    {
        string name = s1.Name;
        int people = s1.Population + s2.Population;
        double area = s1.Area + s2.Area;
        // возвращаем новое объединенное государство
        return new State (name, people, area);
    }

    public static bool operator <(State s1, State s2)
    {
        if (s1.Area < s2.Area)
            return true;
        else
            return false;
    }

    public static bool operator >(State s1, State s2)
    {
        if (s1.Area > s2.Area)
            return true;
        else
            return false;
    }
}

```

```
}
```

Поскольку все перегруженные операторы бинарные, то есть проводятся над двумя объектами, то для каждой перегрузки предусмотрено по два параметра. Теперь используем перегруженные операторы в программе:

```
class Program
{
    static void Main(string[] args)
    {
        State s1 = new State("State1", 100, 300);
        State s2 = new State("State2", 70, 200);
        if (s1 > s2)
            Console.WriteLine("Государство s1 больше государства s2");
        else if (s1 < s2)
            Console.WriteLine("Государство s1 меньше государства s2");
        else
            Console.WriteLine("Государства s1 и s2 равны");
        State s3 = s1 + s2;
        Console.WriteLine("Название государства : {0}", s3.Name);
        Console.WriteLine("Площадь государства : {0}", s3.Area);
        Console.WriteLine("Население государства : {0}", s3.Population);
        Console.ReadLine();
    }
}
```

При перегрузке операторов надо учитывать, что не все операторы можно перегрузить. Например, нельзя перегрузить присваивание.

Обработка исключений

Перехват исключений

В предыдущих лабораторных работах в некоторых программах мы не учитывали непредвиденные ситуации, которые могут приводить к ошибкам. Например, когда нам необходимо было ввести число. Если вместо числа мы ввели бы строку, то при преобразовании этой строки в числовой тип программа бы аварийно завершила работу, и мы получили бы ошибку.

Такие ошибки и другие непредвиденные ситуации в C# называются исключениями.

Обработка исключений – это описание реакции программы на подобные события (исключения) во время выполнения программы. Реакцией программы может быть корректное завершение работы программы, вывод информации об ошибке и запрос повторения действия (при вводе данных).

Примерами исключений может быть:

- деление на ноль;
- конвертация некорректных данных из одного типа в другой;
- попытка открыть файл, которого не существует;
- доступ к элементу вне рамок массива;
- исчерпывание памяти программы;
- другое.

Для обработки исключений в C# используется оператор try-catch. Он имеет следующую структуру:

```
try
```

```

{
    //блок кода, в котором возможно исключение
}
catch ([тип_исключения] [имя])
{
    //блок кода – обработка исключения
}

```

Работает это следующим образом. Выполняется код в блоке try, и, если в нем происходит исключение типа, соответствующего типу, указанному в catch, то управление передается блоку catch. При этом весь оставшийся код от момента выбрасывания исключения до конца блока try не будет выполнен. После выполнения блока catch оператор try-catch завершает работу.

Указывать имя исключения не обязательно. Исключение представляет собою объект, и к нему мы имеем доступ через это имя. С этого объекта мы можем получить, например, стандартное сообщение об ошибке (Message), или трассировку стека (StackTrace), которая поможет узнать место возникновения ошибки. В этом объекте хранится детальная информации об исключении.

Если тип выброшенного исключения не будет соответствовать типу, указанному в catch – исключение не обработается, и программа завершит работу аварийно.

Ниже приведен пример программы, в которой используется обработка исключения некорректного формата данных:

```

using System;
class Program
{
    static void Main()
    {
        Console.WriteLine("Введите число:");
        try
        {
            //вводим данные и конвертируем в целое число
            int a = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Вы ввели число " + a);
        }
        catch (FormatException)
        {
            Console.WriteLine("Ошибка. Вы ввели не число.");
        }
        Console.ReadLine();
    }
}

```

Типы исключений

Ниже приведены некоторые из часто встречаемых типов исключений.

Exception – базовый тип всех исключений. Блок catch, в котором указан тип *Exception* будет «ловить» все исключения.

FormatException – некорректный формат операнда или аргумента (при передаче в метод).

NullReferenceException – в экземпляре объекта не задана ссылка на объект, объект не создан.

IndexOutOfRangeException – индекс вне рамок коллекции.

FileNotFoundException – файл не найден.

DivideByZeroException – деление на нуль.

Обработка нескольких исключений

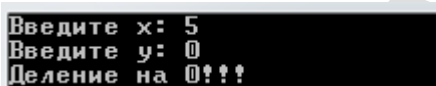
Одному блоку `try` может соответствовать несколько блоков `catch`, которые проверяются последовательно. Если у нас возникает исключение определенного типа, то оно переходит к соответствующему блоку `catch`. Например:

```
using System;
class Program
{
    static void Main()
    {
        try
        {
            Console.Write("Введите x: ");
            int x = int.Parse(Console.ReadLine());
            Console.Write("Введите y: ");
            int y = int.Parse(Console.ReadLine());
            int result = x / y;
            Console.WriteLine("Результат: " + result);
        }

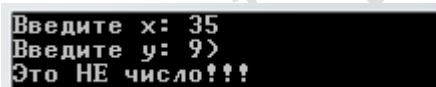
        // Обрабатываем исключение, возникающее при делении на ноль
        catch (DivideByZeroException)
        {
            Console.WriteLine("Деление на 0!!!");
        }

        // Обрабатываем исключение при некорректном вводе числа в консоль
        catch (FormatException)
        {
            Console.WriteLine("Это НЕ число!!!");
        }
        Console.ReadLine();
    }
}
```

Результат работы программы:



```
Введите x: 5
Введите y: 0
Деление на 0!!!
```



```
Введите x: 35
Введите y: 9>
Это НЕ число!!!
```

Данный простой пример наглядно иллюстрирует обработку исключительной ситуации при делении на 0 (`DivideByZeroException`), а также пользовательскую ошибку при вводе не числа (`FormatException`).

Операторы `try-catch` также могут быть вложенными. Внутри блока `try` либо `catch` может быть еще один `try-catch`.

Перехват всех исключений

Время от времени возникает потребность в перехвате всех исключений независимо от их типа. Модифицируем предыдущий пример и добавим вместо двух операторов `catch` следующий код:

```
using System;
class Program
{
```

```

static void Main()
{
    try
    {
        Console.Write("Введите x: ");
        int x = int.Parse(Console.ReadLine());
        Console.Write("Введите y: ");
        int y = int.Parse(Console.ReadLine());
        int result = x / y;
        Console.WriteLine("Результат: " + result);
    }

    // Обрабатываем все исключения
    catch (Exception ex)
    {
        Console.WriteLine("Ошибка: " + ex.Message);
    }
    Console.ReadLine();
}
}

```

Результат работы программы:

```

Введите x: 5
Введите y: 0
Ошибка: Попытка деления на ноль.

```

```

Введите x: 6.3
Ошибка: Входная строка имела неверный формат.

```

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 сообщения об исключении, которое находится в свойстве `Message` класса `Exception`.

Применяя «универсальный» перехват, следует иметь в виду, что его блок должен располагаться последним по порядку среди всех блоков `catch`.

В подавляющем большинстве случаев «универсальный» обработчик исключений не применяется. Как правило, исключения, которые могут быть сгенерированы в коде, обрабатываются по отдельности. Неправильное использование «универсального» обработчика может привести к тому, что ошибки, перехватывавшиеся при тестировании программы, маскируются. Кроме того, организовать надлежащую обработку всех исключительных ситуаций в одном обработчике не так-то просто. Иными словами, «универсальный» обработчик исключений может оказаться пригодным лишь в особых случаях, например в инструментальном средстве анализа кода во время выполнения.

Блок `finally`

Оператор `try-catch` также может содержать блок `finally`. Особенность блока `finally` в том, что код внутри этого блока выполнится в любом случае, в независимости от того, было ли исключение или нет.

```

try
{
    // Блок кода, предназначенный для обработки ошибок
}
catch (ExceptionType1 exObj)
{
    // Обработчик исключения типа ExceptionType1
}
catch (ExceptionType2 exObj)

```

```

        // Обработчик исключения типа Exception
    }
    finally
    {
        // Код завершения обработки исключений
    }

```

Выполнение кода программы в блоке `finally` происходит в последнюю очередь. Сначала `try`, затем `finally` или `catch-finally` (если было исключение).

Обычно он используется для освобождения ресурсов. Классическим примером использования блока `finally` является закрытие файла. `Finally` гарантирует выполнение кода, несмотря ни на что. Даже если в блоках `try` или `catch` будет происходить выход из метода с помощью оператора `return` – `finally` выполнится.

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 следующей программе происходит ввод строки, и мы хотим, чтобы, если длина строки будет больше 6 символов, возникало исключение:

```

using System;
class Program
{
    static void Main()
    {
        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();
    }
}

```

Практическая часть

Пример 1. Описать структуру `Point`, соответствующую точкам на плоскости. Определить в ней метод, вычисляющий расстояние между двумя точками, и метод, вычисляющий расстояние от точки до начала координат. Написать программу, использующую эту структуру.

```

using System;

struct Point // объявление структуры
{
    // поля
    public double x;
    public double y;

    // статический метод вычисления расстояния между двумя точками
}

```

```

static public double Length(Point A, Point B)
{
    return Math.Sqrt(Math.Pow(A.x - B.x, 2) + Math.Pow(A.y - B.y, 2));
}

// экземплярный метод вычисления расстояния от точки до начала координат
public double LengthTo0()
{
    return Math.Sqrt(x * x + y * y);
}
}

class Program
{
    static void Main()
    {
        Point A = new Point();
        Point B = new Point();
        Console.WriteLine("Введите координаты 1-й точки ");
        Console.Write("x= ");
        A.x = Convert.ToDouble(Console.ReadLine());
        Console.Write("y= ");
        A.y = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("Введите координаты 2-й точки ");
        Console.Write("x= ");
        B.x = Convert.ToDouble(Console.ReadLine());
        Console.Write("y= ");
        B.y = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("Расстояние между точками равно {0}", Point.Length(A, B));
        Console.WriteLine("Расстояние от первой точки до начала координат равно {0}",
A.LengthTo0());
        Console.WriteLine("Расстояние от второй точки до начала координат равно {0}",
B.LengthTo0());
        Console.ReadLine();
    }
}

```

Ввод данных организуем через метод.

```

using System;

struct Point
{
    public double x;
    public double y;

    static public double Length(Point A, Point B)
    {
        return Math.Sqrt(Math.Pow(A.x - B.x, 2) + Math.Pow(A.y - B.y, 2));
    }

    public double LengthTo0()
    {
        return Math.Sqrt(x * x + y * y);
    }
}

class Program
{
    static void Main()
    {
        Console.WriteLine("Введите координаты 1-й точки ");
        Point A = InputPoint();
        Console.WriteLine("Введите координаты 2-й точки ");
        Point B = InputPoint();
        Console.WriteLine("Расстояние между точками равно {0}", Point.Length(A, B));
    }
}

```

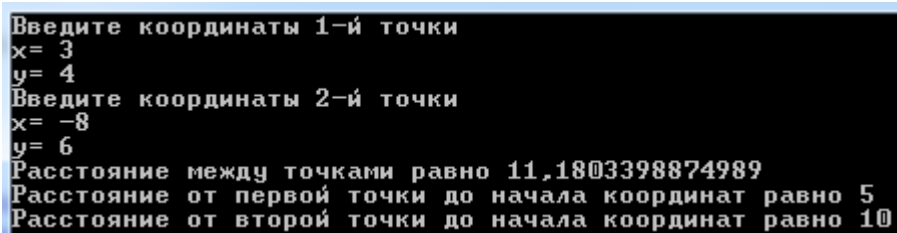
```

        Console.WriteLine("Расстояние от первой точки до начала координат равно {0}",
A.LengthToO());
        Console.WriteLine("Расстояние от второй точки до начала координат равно {0}",
B.LengthToO());
        Console.ReadLine();
    }

    // статический метод для ввода данных
    static Point InputPoint()
    {
        Point P = new Point();
        Console.Write("x= ");
        P.x = Convert.ToDouble(Console.ReadLine());
        Console.Write("y= ");
        P.y = Convert.ToDouble(Console.ReadLine());
        return P;
    }
}

```

Результат выполнения программы:



```

Введите координаты 1-й точки
x= 3
y= 4
Введите координаты 2-й точки
x= -8
y= 6
Расстояние между точками равно 11.1803398874989
Расстояние от первой точки до начала координат равно 5
Расстояние от второй точки до начала координат равно 10

```

Пример 2. Описать класс Rectangle, соответствующий прямоугольникам. Определить в нем:

- конструктор, принимающий длину и ширину прямоугольника;
- метод Square, возвращающий площадь прямоугольника;
- метод Perimeter, возвращающий периметр прямоугольника;
- переопределенный метод ToString.

Предусмотреть возможные исключительные ситуации, если это необходимо. Написать программу, использующую этот класс.

```

using System;

class Rectangle
{
    // объявление закрытых полей
    private double length;    // длина
    private double width;     // ширина

    // объявление свойств
    public double Length      // свойство для поля length
    {
        get { return length; }
        set
        {
            if (value == 0)
                throw new Exception("Нулевое значение стороны");
            if (value < 0)
                throw new Exception("Отрицательное значение стороны");
            length = value;
        }
    }

    public double Width      // свойство для поля width
    {
        get { return width; }
    }
}

```

```

        set
        {
            if (value == 0)
                throw new Exception("Нулевое значение стороны");
            if (value < 0)
                throw new Exception("Отрицательное значение стороны");
            width = value;
        }
    }

    public Rectangle(double l, double w) // конструктор с параметрами
    {
        Length = l;
        Width = w;
    }

    public double Square() // метод вычисления площади прямоугольника
    {
        return Length * Width;
    }

    public double Perimeter() // метод вычисления периметра прямоугольника
    {
        return 2 * (Length + Width);
    }

    public override string ToString() // переопределенный метод ToString
    {
        return string.Format("длина: {0}, ширина: {1}", Length, Width);
    }
}

class Program
{
    static void Main()
    {
        try
        {
            double l, h;
            Console.WriteLine("Введите длину и ширину прямоугольника: ");
            l = Convert.ToDouble(Console.ReadLine());
            h = Convert.ToDouble(Console.ReadLine());
            Rectangle rect = new Rectangle(l, h);
            Console.WriteLine("Размеры прямоугольника: {0}", rect.ToString());
            Console.WriteLine("Площадь прямоугольника: {0}", rect.Square());
            Console.WriteLine("Периметр прямоугольника: {0}", rect.Perimeter());
        }
        catch (FormatException)
        {
            Console.WriteLine("Неверный формат данных");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Ошибка: " + ex.Message);
        }
        Console.ReadLine();
    }
}

```

Результат выполнения программы:

```
Введите длину и ширину прямоугольника:  
6  
5  
Размеры прямоугольника: длина: 6, ширина: 5  
Площадь прямоугольника: 30  
Периметр прямоугольника: 22
```

Задания для самостоятельной работы

Разработайте приложения для решения задач из сборника задач по программированию согласно вашему варианту.

ШГУ им. Т.Г.Шевченко