МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Основы проектирования машин»

СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНЫХ ПРИЛОЖЕНИЙ

Методические рекомендации к лабораторным работам для студентов направления подготовки 15.03.03 «Прикладная механика» очной формы обучения

Могилев 2024

УДК 621.01
ББК 36.4
С87

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Основы проектирования машин» «24» апреля 2024 г., протокол № 10

Составитель канд. техн. наук, доц. А. П. Прудников

Рецензент ст. преподаватель Ю. С. Романович

Изложены цели, содержание и порядок выполнения лабораторных работ.

Учебное издание

СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНЫХ ПРИЛОЖЕНИЙ

Ответственный за выпуск	А. П. Прудников
Корректор	А. Т. Червинская
Компьютерная верстка	Е. В. Ковалевская

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл. печ. л. . Уч.-изд. л. . Тираж 26 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

Содержание

Введение

Методические рекомендации составлены в соответствии с рабочей программой по курсу «Средства разработки программных приложений» для студентов направления подготовки 15.03.03 «Прикладная механика» очной формы обучения.

Информационные технологии играют важную роль при создании новых продуктов и услуг, улучшении коммуникации и доступа к информации. Обладание навыками программирования позволяет решать сложные задачи и проводить научные исследования, автоматизировать и оптимизировать различные процессы, упрощая работу и повышая ее эффективность. С ростом компьютеризированности всех областей человеческой деятельности востребованность навыков создания собственных программ увеличивается и вместе с тем повышается актуальность проблемы обучения и самообучения навыкам программирования.

Целью изучения дисциплины «Средства разработки программных приложений» является формирование у студентов теоретических фундаментальных основ создания программных приложений.

Объектом изучения дисциплины «Средства разработки программных приложений» является интегрированная среда разработки Visual Studio и язык программирования C#.

Язык программирования предназначен для написания компьютерных программ, которые представляют собой набор правил, позволяющих компьютеру выполнить тот или иной вычислительный процесс, организовать управление различными объектами [1–3].

В краткой форме изложены цель, содержание и порядок выполнения лабораторных работ.

Лабораторная работа выполняется в среде Visual Studio, отчет по лабораторной работе представляет собой код приложения по теме лабораторного занятия.

Методические рекомендации предназначены для самостоятельной подготовки студентов к лабораторным работам по дисциплине «Средства разработки программных приложений».

1 Лабораторная работа № 1. Работа в интегрированной среде разработки приложений

Цель работы: ознакомиться с возможностями среды Microsoft Visual Studio.NET.

Теоретические основы

Microsoft Visual Studio.NET – это среда объектно-ориентированного визуального программирования. Работая в ней, даже начинающий программист может создать программу с профессионально выглядящим интерфейсом. Такая программа может выполнять несложные действия, порой выдавать ошибочные ответы, но при этом ее интерфейс не будет отличаться от многих Windows-приложений.

Практически все существующие системы разработки программного обеспечения используют понятие проекта. Проект объединяет все файлы, необходимые для редактирования, трансляции и отладки приложения.

Для создания нового проекта надо выполнить команду File / New / Project, при этом откроется окно New Project. В правой части экрана можно выбрать один из предложенных шаблонов для данного типа проектов:

– Windows Application – шаблон для Windows-приложения;

– Class Library – шаблон для создания библиотеки классов, которая будет использоваться другими приложениями;

– Control Library – шаблон для создания элементов управления, которые будут использоваться в приложениях с графическим пользовательским интерфейсом для платформы Windows (называемых также приложениями Windows Forms);

– ASP.NET Web Application – шаблон для создания Web-приложений ASP.NET;

– ASP.NET Web Service – шаблон для создания Web-сервисов;

– Web Control Library – шаблон для создания элементов управления, которые будут использоваться в Web-приложениях;

– Console Application – шаблон для создания консольных приложений;

– Windows Service – шаблон для создания сервисов операционной системы;

– Empty Project / Empty Web Project – проект, который создается без использования шаблонов;

– New Project In Existing Folder – добавить новый проект в уже существующую папку.

Хотя при создании нового проекта в среде Visual Studio.NET предлагается довольно большой список типов проектов, но на самом деле есть всего три основные разновидности приложений – Windows Application, Console Application и Class Library. Все остальное – это их различные вариации за счет использования тех или иных шаблонов или мастеров (именно поэтому правое окно и называется Templates), обеспечивающих автоматическое выполнение каких-то

начальных действий. Пользователь может подключить и свои собственные варианты шаблонов.

В открывшемся окне создания нового проекта в левом списке нужно выбрать строку Visual C# Project, в правом списке отметить шаблон Windows Application. Затем в поле ввода ввести имя проекта и, используя кнопку Browse, выбрать каталог, в котором будут храниться файлы проекта.

На рабочем экране располагается довольно много вкладок, которые могут закрывать друг друга, кроме того, их можно удалять с экрана. В пункте меню View собраны команды, которые позволяют вывести и активизировать необходимую вкладку. Команда View / Designer открывает вкладку проектирования формы; View / Code – вкладку Редактор кода; View / Toolbox – панель с компонентами; View / Properties Window – вкладку Свойства (Properties).

Среда предоставляет формы, на которых размещаются компоненты. Обычно это оконная форма, хотя могут быть и невидимые формы. На форму с помощью мыши переносятся и размещаются пиктограммы компонентов. С помощью простых манипуляций можно изменить размеры и место положения компонентов. При этом все время в процессе проектирования виден результат – изображение формы и расположенных на ней компонентов.

Во время проектирования формы и размещения на ней компонентов Visual Studio.NET автоматически формирует код программы, включая в него соответствующие фрагменты, описывающие данный компонент. Компоненты могут быть визуальными, видимыми при работе приложения, и невизуальными, выполняющими те или иные служебные функции. Визуальные компоненты сразу видны на экране в процессе проектирования в таком же виде, в каком их увидит пользователь во время выполнения приложения.

Обработчики событий – это функции, которые выполняются при наступлении этих событий. Существует несколько способов создания обработчиков событий. Для создания обработчика события можно, например, сначала на форме выделить компонент, с которым это событие должно быть связано, затем перейти на вкладку Свойства, открыть страничку событий (Events), выбрать имя события из списка и выполнить двойной щелчок справа от него. При этом откроется окно Редактор кода, в котором уже будет сгенерирована средой заготовка нужной функции, разработчику останется только вписать в фигурные скобки операторы, задающие реакцию приложения на выбранное событие.

Одним из способов обнаружить логическую ошибку является выполнение кода программы по одной строке и изучение изменения содержимого одной или нескольких переменных. Для этого можно при работе программы войти в режим останова, а затем просмотреть код во вкладке Редактор кода. Режим останова дает возможность просмотреть программу во время ее исполнения. Точка останова показывает то место программы, где выполнение будет остановлено, и вы сможете использовать инструменты разработки Visual Studio. Простейший способ установить точку останова – щелкнуть на сером поле слева от окна с исходным кодом программы. Можно также переместить курсор на нужную строку и щелкнуть на кнопке Точка останова на панели инструментов. Щелчок на этой кнопке установит точку останова, если ее там не было,

и, наоборот, уберет ее, если она уже была установлена на этой строке. Теперь, если запустить программу в режиме отладки, ее выполнение остановится при достижении точки останова. Желтая стрелка на красном кружке, обозначающем точку останова, указывает, на какой именно точке останова прервано выполнение программы установить точку останова. Переместите указатель мыши к полосе Margin Indicator (указатель поля – серая полоса, расположенная сразу за левым полем окна Редактор кода) рядом с оператором, а затем щелкните на этой полосе, чтобы установить точку останова. Немедленно появится красная точка останова.

Оператор, который вы выбрали для создания точки останова, выделен желтым цветом, а на полосе Margin Indicator (Указатель поля) появилась стрелка. Теперь Visual Studio находится в режиме останова, и вы можете узнать этот режим по слову [break], появившемуся в строке заголовка Visual Studio.

Поместите указатель мыши в Редактор кода над переменной. Visual Studio выведет сообщение, в котором будет указано значение этой переменной.

Щелкните на кнопке Stop Debugging (Остановить отладку) панели инструментов Debug (Отладка).

После остановки программы отладчиком можно продолжить ее выполнение в пошаговом режиме. Есть несколько кнопок, предназначенных для выполнения в пошаговом режиме. Наиболее часто используются следующие из них (в порядке расположения на панели инструментов):

– Step Into (Шаг с заходом);

– Step Over (Шаг с обходом);

– Step Out (Шаг с выходом).

Есть еще одна команда контекстного меню – Run to Cursor (Выполнить до текущей позиции).

Если курсор находится на вызове какой-либо функции, то при щелчке на кнопке Step Into (Шаг с заходом) он перейдет на первую строку этой функции. Если же щелкнуть на кнопке Step Over (Шаг с обходом), произойдет вызов функции и курсор переместится на следующую строку.

Порядок выполнения работы

Одним из самых используемых управляющих компонентов является визуальный компонент Button.

Данный компонент имеет довольно много свойств, но рассмотрим всего шесть из них. Отметим, что этими свойствами обладают почти все визуальные компоненты, а свойством Name вообще, все компоненты. Свойство Name задает имя компонента. По умолчанию имя первой кнопки будет button1, второй кнопки – button2, и т. д. Свойство Name можно изменить. Для этого надо выделить кнопку на форме и открыть вкладку Свойства (Properties), найти название свойства и в поле ввода, расположенном справа от названия, ввести новое имя компонента. Свойство Name – это идентификатор, и строится оно по правилам построения идентификаторов.

Свойство Text – надпись на компоненте, оно имеет тип string. Данное свойство также можно изменить. При задании значения свойства Text можно использовать русские буквы и другие символы, запрещенные при определении идентификаторов.

Свойства Left, Top, задают координаты левого верхнего угла кнопки в системе координат компонента-контейнера. Компонент-контейнер – это компонент, на котором располагается кнопка. Каждый визуальный компонент имеет свою систему координат, в этой системе координат точка (0,0) располагается в верхнем левом углу, ось X направлена слева на право, ось У сверху вниз. Свойства Width, Height задают соответственно ширину и высоту компонента. Свойства Left, Top, Width, Height имеют тип int. Значения всех перечисленных свойств, кроме Name, могут быть изменены во время работы программы. Например, для изменения надписи на кнопке можно использовать следующий код:

```
button1.Text=”Первая кнопка”;
```

Для изменения положения кнопки на форме следующие команды:

```
button1. Left= button1. Left+10;
button1. Top = button1. Top +10;
```

При выполнении последних двух команд, кнопка сдвинется вправо и вниз на 10 пикселей.

При обращении к свойствам формы в обработчиках событий этой формы можно использовать служебное слово this. Например, команды this.Width, this.Height будут обращениями соответственно к ширине и высоте текущей формы. Команда this.Height =450; изменит высоту формы, а следующая строка переместит кнопку button1, если она находится на нижнем краю формы.

```
if (this. Height < button1. Top+25) button1. Top=100;
```

Необходимо создать обработчики событий, выполняющие следующие действия:

1 На форме две кнопки A, B.

По щелчку на кнопке A кнопка B перемещается влево на 10 пикселей, при достижении левого края формы кнопка B перескакивает к правому краю формы. По щелчку на кнопке B кнопка A уменьшается на 2 пикселя в длину, при достижении минимального размера, размер кнопки A восстанавливается.

2 На форме две кнопки A, B.

По щелчку на кнопке A кнопка B движется по диагонали формы (вниз на 2 пикселя и вправо на 4 пикселя), при достижении края формы кнопка B возвращается в начальное положение. По щелчку на кнопке B кнопка A уменьшается на 2 пикселя в длину, при достижении минимального размера, размер кнопки A возвращается в первоначальное состояние.

3 На форме четыре кнопки A, B, C, D. Кнопки A и B расположены одна над другой. По щелчку на кнопке C кнопки A и B сближаются на 16 пикселей, но не перекрывают друг друга. По щелчку на кнопке D кнопки A и B расходятся на 16 пикселей, но останавливаются на краях формы.

4 На форме три кнопки A, B, C.

По щелчку на кнопке A кнопка B движется по диагонали формы (вниз на 2 пикселя и вправо на 4 пикселя), при достижении края формы кнопка B воз-

вращается в начальное положение. По щелчку на кнопке С кнопка В перемещается вниз на 6 пикселей, при достижении края формы возвращается в прежнее положение.

Вопросы для самоконтроля

1 Как создать новый проект? Как открыть проект?
2 Как открыть файл, не принадлежащий проекту?
3 Как создать обработчик события, связанный с определенным компонентом?
4 Как изменить значение свойства Left на этапе проектирования?

2 Лабораторная работа № 2. Классы и объекты. Инкапсуляция

Цель работы: научиться создавать свои классы и работать с ними.

Теоретические основы

Язык C# позволяет разработчику определить новый тип данных – класс. Простейшее описание класса имеет следующий вид:

class имя класса {< список членов класса>}

Имя класса – это любой идентификатор, уникальный в пределах своей сферы действия.

Список членов класса – список, объявляющий члены класса с указанием уровня доступа.

Члены класса – это поля (члены-данные), методы (функции-члены), свойства и события. В данной лабораторной работе будут создаваться классы, имеющие в качестве членов поля и методы. Например:

```
class Myclass {
public int i;  /* слово public  обозначает, что данный член класса является общедоступным*/
public float x;
public float f()
{return i*y;}}
```

Во всех объектно-ориентированных языках четко различаются понятия «класс» и «объект». Класс – это определяемый пользователем тип данных, его можно представить как чертеж, по которому будут строиться сами данные, т. е. объекты (экземпляры) класса.

В C# создание объектов класса происходит с помощью ключевого слова new. Например:

```
Myclass a1=new Myclass();
```

или

```
Myclass a2;
a2=new Myclass();
```

Ключевое слово new означает, что среде выполнения следует вызвать конструктор класса и выделить необходимое количество оперативной памяти под экземпляр класса. Поскольку класс относится к ссылочному типу, то выделение памяти производится из «кучи», находящейся в распоряжении среды выполнения .NET. Команды Myclass a1, Myclass a2, определяют ссылки на объекты класса, а не сами объекты, объекты создаются с помощью ключевого слова new. Но после того, как объекты будут созданы, мы будем их называть объект (экземпляр) a1, и объект (экземпляр) a2. Например:

```
Myclass c=new Myclass();
Myclass d;
d=c;
```

В этом примере создан один объект и две ссылки на него.

Доступ к членам класса осуществляется с помощью операции «точка» «.».

Например:

```
c.x=7; //полю x объекта c присвоено значение 7.
d.i=6; //полю i  объекта d присвоено значение 6.
a1.x=c.f();/*полю x объекта a1 присвоено значение, которое возвращает метод f, вызванный для объекта c, поскольку ссылка d также указывает на этот же объект, то метод f вернет значение 42.*/
```

Инкапсуляция – это механизм объединения данных и кода, манипулирующего этими данными, а также защиты того и другого от внешнего вмешательства, неправильного использования или от несанкционированного доступа. Объект – это то, что поддерживает инкапсуляцию (объединяет в себе данные и код, работающий с ними). Для закрытия данных внутри объекта используются модификаторы доступа, в частности модификатор private, который используется по умолчанию, если не употреблен другой. Данные или методы, объявленные с этим модификатором, будут недоступны вне класса (объекта), т. е. к ним можно будет обратиться через открытые (объявленные с модификатором public) методы или свойства класса. Свойства класса – нечто среднее между полем и методом, представляет собой конструкцию вида:

```
<Модификатор доступа> <Тип свойства> <Имя свойства>
{
get{return <значение>}
set{<поле>=value}
}
```

Обычно свойства связываются с закрытыми полями класса и помогают осуществить доступ к этим полям из внешних (относительно класса) частей программы. Свойства вместе с модификаторами доступа реализуют механизм защиты данных от несанкционированного доступа. Как мы видим, свойство имеет заголовок и тело. В заголовке указывается модификатор доступа (обычно public), тип возвращаемого свойством значения и имя свойства. В теле объявлено два метода get и set. Больше ничего в теле свойства объявлять нельзя. Метод get имеет ключевое слово return и возвращает какое-либо значение (обычно значение какого-либо поля, хотя не обязательно). Метод set имеет ключевое слово value и присваивает (устанавливает) это значение полю объекта.

Пример объявления свойства в классе MyClass:

```
public class MyClass
{
int a; //поле
public int A// свойство
{
get { return a;}
set { a=value;}
}
}
```

Пример использования описанного свойства в программе:

```
MyClass MyObj=new MyClass();
MyObj.A=6; // полю a объекта MyObj присвоится значение 6.
int b=MyObj.A; // переменной b присвоится значение поля a объекта MyObj.
```

Порядок выполнения работы

1 Продумать структуру класса, предназначенного для решения задач, определенных вариантом индивидуального задания. Имена класса и его членов должны быть информативны. В классе не должно быть полей, в которые пользователь мог бы поместить противоречивые данные. Например, в классе Окружность не может быть одновременно полей Радиус и Диаметр, надо выбрать одно из них, а второе заменить методом. Или, если рассматривается класс Равнобокая_трапеция, то набор полей Основание1, Основание2, Боковая_сторона может получить значения, по которым не построишь трапецию (например, 5, 105,10). Но если поле Боковая_сторона заменить на поле Высота, то проблем не будет.

2 Поскольку значения полей класса должен задавать пользователь, то на форму надо поместить столько полей ввода (компонентов *TextBox*), сколько полей в классе.

3 Создание локального объекта, т. е. объекта с которым может работать только один метод. Поместить на форму кнопку. Обработчик события *Click* должен создать объект Вашего класса, считать информацию из полей ввода и поместить ее в поля класса, вызвать методы, выполняющие необходимые вычисления, и вывести на экран возвращенные ими значения.

Задания для самостоятельного решения.

Задача 1. Класс, описывающий прямоугольник. Класс должен иметь методы, выполняющие расчет длины диагонали и расчет площади прямоугольника.

Задача 2. Класс, описывающий квадрат. Класс должен иметь методы, выполняющие расчет длины диагонали и расчет периметра квадрата.

Задача 3. Класс, описывающий равнобокую трапецию. Класс должен иметь методы, выполняющие расчет периметра трапеции и расчет площади трапеции.

Задача 4. Класс, описывающий окружность. Класс должен иметь методы, выполняющие расчет длины окружности и расчет площади окружности.

Задача 5. Класс, описывающий прямоугольный треугольник. Класс должен иметь методы, выполняющие расчет гипотенузы и расчет площади треугольника.

Задача 6. Класс, описывающий круглый прямой цилиндр. Класс должен иметь методы, выполняющие расчет диаметра цилиндра и расчет объема цилиндра.

Задача 7. Класс, описывающий круглый прямой конус. Класс должен иметь методы, выполняющие расчет диаметра конуса и расчет объема конуса.

Вопросы для самоконтроля

1 Что такое класс?
2 Как описывается класс?
3 Что задает слово public?
4 Как происходит доступ к членам класса?
5 Как вызвать метод класса?

3 Лабораторная работа № 3. Конструкторы. Полиморфизм и наследование

Цель работы: научиться создавать конструкторы классов, иерархии классов, организации доступа к элементам базового и производных классов.

Теоретические основы

Конструктор класса – это метод класса, который предназначен для создания объектов класса и инициализации их.

Конструктор обладает следующими свойствами:

– имя конструктора совпадает с именем класса;

– конструктор не имеет возвращаемого значения, даже типа void;

– класс может содержать несколько конструкторов, отличающихся либо количеством, либо типами параметров (отличие в этом случае должно быть существенным), либо и тем и другим;

– конструктор без параметров называется конструктором по умолчанию;

– если ни один конструктор не был написан программистом, то конструктор по умолчанию генерируется компилятором;

– если в классе объявлен хотя бы один конструктор, то компилятор не создает своего;

– в C#, в отличие от многих других языков, любой конструктор, если не указано иное, обнуляет значения полей.

Конструктор не может быть вызван явно, он вызывается автоматически при создании экземпляров класса.

Рассмотрим пример. Определим в классе A два конструктора.

```
class A {  public int x,y;
            public int f() {return x+y;}
    public A() {  x=1; y=2;  }
    public A (int xx, int yy) { x=xx; y=yy;} }
```

При выполнении команды A a=new A(); для создания объекта класса используется конструктор без параметров. Полям объекта a будут присвоены соответственно значения 1 и 2.

Если при создании объекта требуется использовать конструктор с параметрами, то в операторе new после имени класса в круглых скобках указываются значения параметров, например, A b=new A(6,13);

Команда A c=new A(3); вызовет ошибку, потому что класс A не содержит конструктора с одним параметром, которому можно было бы присвоить значение 3.

В языке C# существует упрощенная форма написания конструктора: при описании полей класса можно написать инициализирующие выражения:

```
class B  {
public int i=10;}
```

Все строки, содержащие инициализирующие выражения, включаются в начало всех конструкторов. В данном примере строка i=10; будет включена в конструктор, определяемый компилятором.

Класс в C# может иметь произвольное количество потомков и только одного предка. При описании класса имя его предка записывается в заголовке класса после двоеточия:

[атрибуты][спецификаторы] class <имя_класса> [:предки] <тело класса>

Если имя предка не указано, предком считается базовый класс всей иерархии System. Object.

Элементы базового класса, определенные как private, в производном классе недоступны. Поля, определенные со спецификатором protected, будут доступны методам всех классов, производных от базового. Этот модификатор остается со своим членом независимо от реализуемого количества уровней наследования.

Пример 1. Рассмотрим в качестве базового класс, в котором описывается печатная продукция (журналы, газеты и др.)

```
class Press { //Закрытые поля
string name; //Название
int copies; //Тираж
double price;//Цена
//Конструктор с параметрами
public Press (string name, int copies, double price)
{ this.name = name; this.copies = copies; this.price = price;}
//Свойства
public string Name [get [return name;}} public int Copies (get [return copies;}
set [copies = value;}} public double Price [get {return price;}}
```

```
//Метод вычисления стоимости тиража
public double Cost(){return copies*price;}

//Метод вывода состояния объекта
public void Output(){
Console.WriteLine("Название:  {0} \t Тираж: {1}
\t Цена: {2}",name,copies,price);}}
```

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными далее правилами.

1 Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса без параметров.

2 Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса. Таким образом, каждый конструктор инициализирует свою часть объекта.

3 Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации. Вызов выполняется с помощью ключевого слова base.

Формат расширенного объявления таков:

имя_производного_класса(список_тараметров):base (список_аргументов)
{//тело конструктора}

Здесь с помощью элемента список_аргументов задаются аргументы, необходимые конструктору в базовом классе. При отсутствии ключевого слова base автоматически вызывается конструктор базового класса, действующий по умолчанию.

Пример 2. Дополним базовый класс из примера 1 производным Magazine, в котором есть дополнительное поле – качество бумаги:

```
class Magazine: Press {
string quality; //Качество бумаги
public string Quality //Свойство
{ get { return quality; }     }
//Конструктор
public Magazine(string name, int copies, double pric4, string quality)
: base (name, copies, price) //Вызов базового конструктора
{ this.quality = quality;} }
```

Ключевое слово base всегда отсылает к базовому классу, стоящему в иерархии классов непосредственно над вызывающим классом. В данном примере это класс Press и его конструктор.

Новым членам (полям, методам и свойствам) производного класса можно давать имена, совпадающие с именами членов базового класса. В этом случае перед членом производного класса необходимо поставить ключевое слово new.

При этом, хотя соответствующие члены базового класса наследуются, они становятся скрытыми в производном классе.

Для обращения к скрытому члену применяется ссылка base, которая указывает на базовый класс производного класса. Формат ее записи такой:

base<член базового класса>

Здесь в качестве элемента член базового класса можно указывать либо метод, либо переменную экземпляра.

Объекту базового класса можно присваивать объект производного класса, но вызываются для него только методы и свойства, определенные в базовом классе. Иными словами, возможность доступа к элементам класса определяется типом ссылки, а не типом объекта, на который она указывает. Это и понятно: ведь компилятор еще до выполнения программы определяет, какой метод вызывать, и вставляет в код фрагмент, передающий управление на этот метод. Этот процесс называется ранним связыванием.

Поскольку в производном классе определяется собственный метод с именем Cost, он скрывает одноименный метод, определенный в базовом классе. Следовательно, при вызове этого метода для объекта типа Magazine вычисления выполняются так, как предусмотрено в этом классе, а не в классе Press. Ссылка base позволяет получить доступ к методу в базовом классе. Объект informatica типа Press создается с помощью конструктора класса Magazine. Но поскольку разрешение, какой метод вызвать, в данном случае осуществляется в момент компиляции (раннее связывание), то обращение происходит к методу базового класса.

Запрет наследования: классы, объявленные как sealed. В некоторых случаях требуется запретить создание производных классов.

При работе с объектами производных классов, через ссылки базовых классов, доступны, за одним исключением, только члены, определенные в базовом классе (классе ссылки). Исключение – это виртуальные методы.

Если нужно перейти от ссылки базового класса к ссылке производного класса, то надо выполнить преобразование типов.

Поскольку по ссылке базового класса не всегда ясно на какой объект она указывает, то в C# введены специальные операторы, которые выполняют проверку возможно ли сделать соответствующее преобразование типов.

Оператор is возвращает значение булево типа, true – если объект, на который указывает ссылка, можно привести к типу, стоящему после is и false – если ссылка нулевая или объект нельзя привести к указанному типу. Например:

```
int i=8;
Object o=new Object();
Boolean b=(i is Object); //b будет равно true
Boolean b1=(o is int); //b1 будет равно false
```

Оператор as работает со ссылочными типами, он производит нужное преобразование типов, если оно возможно, и возвращает null, если оно невозможно. Команда A a=o as A; будет выполнена успешно, но в ссылку a будет записан

null, и любое обращение к a как к объекту, например, a.ToString(); вызовет генерацию исключения. Проверку того, не получила ли ссылка нулевого значения, должен выполнить программист. В нашем примере она будет выглядеть так: if(a!=null)a.ToString();

Порядок выполнения работы

Задания для самостоятельной работы.

Задача 1. Базовый класс «Точка». Классы «Окружность» и «Прямоугольный треугольник» являются производными от класса «Точка». Класс «Конус» является производным от класса «Окружность». Во всех классах определить по два метода, вычисляющих площадь фигуры (площадь точки равна нулю, для объемных тел вычисляется площадь полной поверхности). Один метод – виртуальный, другой – обычный. В классе «Прямоугольный треугольник» еще определить метод, вычисляющий периметр. В классе «Конус» определить метод, вычисляющий объем.

Задача 2. Базовый класс «Точка». Классы «Окружность» и «Ромб» являются производными от класса «Точка». Класс «Конус» является производным от класса «Окружность». Во всех классах определить по два метода, вычисляющих площадь фигуры (площадь точки равна нулю, для объемных тел вычисляется площадь полной поверхности). Один метод – виртуальный, другой – обычный. В классе «Ромб» еще определить метод, вычисляющий периметр. В классе «Конус» определить метод, вычисляющий объем.

Задача 3. Базовый класс «Точка». Классы «Правильный пятиугольник», «Прямоугольный треугольник», «Прямоугольник» являются производными от класса «Точка». Во всех классах определить по два метода, вычисляющих площадь фигуры (площадь точки равна нулю, для объемных тел вычисляется площадь полной поверхности). Один метод – виртуальный, другой – обычный. В классе «Прямоугольник» еще определить метод, вычисляющий диагональ. В классах «Треугольник», «Правильный пятиугольник» и «Прямоугольник» определить методы, вычисляющие периметры соответствующих фигур.

Вопросы для самоконтроля

1 Что такое конструктор класса?
2 В какой момент происходит вызов конструктора?
3 Может ли класс содержать несколько конструкторов?
4 В чем состоит принцип наследования?
5 Какие члены класса наследуются?
6 Что представляет собой защищенный доступ?
7 Как происходит вызов конструкторов базового класса?

4 Лабораторная работа № 4. Переопределение операций

Цель работы: получение навыков в разработке программ с использованием перегрузки операторов.

Теоретические основы

Использование нескольких методов с одним и тем же именем, но различными типами параметров и их способами передачи называется перегрузкой методов.

Компилятор определяет, какой именно метод требуется вызвать по типу фактических параметров. Этот процесс называется разрешением (resolution) перегрузки. Тип возвращаемого методом значения в разрешении не участвует. Допустим, имеется четыре варианта метода, определяющего наибольшее значение:

```
// Возвращает наибольшее из двух целых:
int max( int a, int b )
// Возвращает наибольшее из трех целых:
int max( int a, int b, int c )
// Возвращает наибольшее из первого параметра и длины второго:
int max ( int a, string b )
// Возвращает наибольшее из второго параметра и длины первого:
int max ( string b, int a)
Console.WriteLine(max (1,2));
Console.WriteLine(max (1,2,3));
Console.WriteLine(max (1, “2”));
Console.WriteLine(max (“1”,2));
```

При вызове метода max компилятор выбирает вариант метода, соответствующий типу передаваемых в метод аргументов (в приведенном примере будут последовательно вызваны все четыре варианта метода).

Если точного соответствия не найдено, выполняются неявные преобразования типов в соответствии с общими правилами. Если преобразование невозможно, выдается сообщение об ошибке. Если соответствие на одном и том же этапе может быть получено более чем одним способом, выбирается «лучший» из вариантов, т. е. вариант, содержащий меньшие количество и длину преобразований. Если существует несколько вариантов, из которых невозможно выбрать лучший, выдается сообщение об ошибке.

Например, методы, заголовки которых приведены ниже, имеют различные сигнатуры и считаются перегруженными:

```
int max( int a, int b )
int max( int a, ref int b )
```

Перегрузка методов является проявлением полиморфизма, одного из основных свойств объектно-ориентированного программирования.

Язык C# предоставляет возможность создать метод, в который можно передавать разное количество аргументов. Для этого используется ключевое сло-

во params. Параметр, помеченный этим ключевым словом, может быть только один, размещается в списке параметров последним и обозначает массив заданного типа неопределенной длины. Соответствующие ему аргументы должны иметь типы, для которых возможно неявное преобразование к базовому типу массива. Например:

```
public int Calculate( int a, params int[]d)
```

В этот метод можно передать три и более параметров. Внутри метода к параметрам, начиная с третьего, обращаются как к обычным элементам массива. Количество элементов массива получают с помощью его свойства Length.

```
int[] a = { 10, 20,    30 };
Console.WriteLine(Calculate (5,a));
int[] b = { -11, -4, 12, 14, 32, -1, 28}
Console.WriteLine(Calculate (3,b));
short z = 1, e = 12;
byte v = 100;
Console.WriteLine(Calculate z, e, v))
Console.WriteLine(Calculate ());   //ошибка
```

Когда компилятор пытается разрешить вызов метода, он выполняет поиск метода, список аргументов которого соответствует вызываемому методу. Если не удается найти перегрузку метода с соответствующим списком аргументов, но имеется подходящая версия с параметром params нужного типа, выполняется вызов этого метода с добавлением в массив дополнительных аргументов.

C# позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Определение собственных операций класса называют перегрузкой операций.

При перегрузке операции ни одно из его исходных значений не теряется. Перегрузку операции можно расценивать как введение новой операции для класса. Например, можно применять операции таким же образом, как стандартные:

```
MyObject a, b, c;
c = a + b; //используется операция сложения для класса MyObject
```

Операции класса описываются с помощью методов специального вида (функций-операций).

Синтаксис операции:

```
<спецификаторы> <тип_возврата> operator
op(тип_параметра операнд1[,тип_параметра операнд2])
{<тело операции>}
```

В качестве спецификаторов одновременно используются ключевые слова public и static. Элемент op – это оператор (например " + " или " / "), который перегружается. Тело операции определяет действия, которые выполняются при использовании операции в выражении.

При описании операций необходимо соблюдать следующие правила:

– операция должна быть описана как открытый статический метод класса (спецификаторы public static);

– параметры в операцию должны передаваться по значению (т. е. не должны предваряться ключевыми словами ref или out);

– сигнатуры всех операций класса должны различаться;

– типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (т. е. должны быть доступны при использовании операции).

В C# существуют три вида операций класса: унарные, бинарные и операции преобразования типа.

Унарные операции.

Можно определять в классе следующие унарные операции:

+ , – , ! , ~ , ++ , ^, true, false

Примеры заголовков унарных операций:

public static int operator + (MyObject m)

public static MyObject operator --(MyObject m)

public static bool operator true(MyObject m)

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

– для операций +, –, ! и ~ величину любого типа;

– для операций ++ и — величину типа класса, для которого она определяется;

– для операций true и false величину типа bool.

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

Бинарные операции.

Можно определять в классе следующие бинарные операции: + , – , * , / , | , & , | | ,&&,==,=!,>,<,>=, <= Примеры заголовков бинарных операций:

public static MyObj ect operator + (MyObj ect ml, MyObj ect m2)

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

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

Операции == и !=, > и <, >= и <= определяются только парами и обычно возвращают логическое значение.

Пример. Определение операции отношения (сравнение длин векторов) для класса TD:

```
class TD // Класс трехмерных координат
{
double x, y, z; // 3-х-мерные координаты
public TD()
{ x = y = z = 0; }
public TD
(double i, double j, double k)
{ x = i; y = j; z = k; }
static double r(TD op) //Вычисление длины вектора
{
```

```
return
Math.Sqrt(op.x * op, x + op.y * om.y + op.z * op.z);
}
// Перегрузка оператора ">"
public static bool operator > (TD opl, TD op2)
{ return r(opl) > r (op2); }
// Перегрузка оператора "<"
public static bool operator < (TD opl, TD op2)
{ return (opl) <r (op2); }
public void show() II Отображаем координаты X, У, Z
{ Console.WriteLine(x+","+ y + "," + z);}
// Перегрузка унарного оператора"++" public static TD operator ++(TD op)
{op.x++; // Оператор"++" модифицирует аргумент op
op.y++;
op.z++;
return op;}}
class TDDemo
{
public static void Main()
{
TD a new TD(1, 5, 3);
TD c = new TD(3, 2, 1);
Console.Write("Координаты точки a: ");
a.show();
Console.Write("Координаты точки c: ");
c.show();
Console.WriteLine(); if (a > c)
Console.WriteLine("a>c ");
else Console.WriteLine("a<c");
a++; // Инкрементирование a.
Console.Write("Результат a++: ");
a.show();
}
}}
```

Порядок выполнения работы

Разработать класс RationalNumber (дроби) со следующими возможностями:

– создать конструктор, который предотвращает равенство нулю знаменателя дроби, сокращает или упрощает дроби, если они не в сокращенной форме, и исключает отрицательные знаменатели;

– перегрузить операции сложения, вычитания, умножения и деления для этого класса;

– перегрузить операции отношения и проверки на равенство для этого класса.

Написать программу, демонстрирующую работу с этим классом.

Задания для самостоятельной работы.

Задача 1. Описать класс для работы с двумерным массивом строк. Обеспечить перегрузку операции «+» для построчного соединения элементов.

Задача 2. Описать класс для работы с двумерным массивом целых чисел. Реализовать возможность уменьшения количества столбцов массива на заданное число (перегрузка операции «–») с удалением их с начала массива.

Вопросы для самоконтроля

1 Каково назначение перегрузки операторов?

2 Можно ли перегрузкой отменить очередность выполнения операции?

3 Сколько аргументов требуется для определения перегруженной унарной операции?

4 Какие операторы нельзя перегружать?

5 Лабораторная работа № 5. Интерфейсы и абстрактные классы

Цель работы: изучить синтаксис абстрактного класса и интерфейса.

Теоретические основы

Класс называется абстрактным, если он предназначен только для использования в качестве базового при наследовании и нельзя создавать его объекты. На языке C# для описания базового класса используется служебное слово abstract. Абстрактные классы могут содержать точно такие же члены, как и обычные классы. Кроме того, они могут содержать и абстрактные методы – методы, не имеющие реализации. Абстрактные методы описываются специальным образом. Во-первых, при их описании используется служебное слово abstract, а во-вторых вместо описания тела метода ставится точка с запятой. Пример:

```
abstract class M{
public int x;
public int g(){return x;}
abstract public int fun();    }
```

Абстрактный класс M, содержит поле x, не абстрактный метод g() и абстрактный метод fun().

Абстрактные методы не могут содержаться в неабстрактных классах. Все абстрактные методы должны быть определены в производных классах с использованием служебного слова override.

```
class N:M{
override public int fun(){return 3;}}
```

Интерфейсы нужны для стандартизации интерфейсов объектов. Очень важно, чтобы методы, выполняющие сходные действия в разных классах, назывались одинаково и имели одинаковые формальные параметры.

Интерфейс – это именованный набор абстрактных членов. Интерфейсы не могут содержать поля, но могут содержать абстрактные методы, свойства и события.

В языке C# интерфейс – это специальный тип (interface), содержащий только абстрактные члены. Поскольку других членов в интерфейсе быть не может, то при их описании не используется слово abstract. Также интерфейс определяет только общедоступные члены и соответственно при их описании не указывается уровень доступа. Пример:

```
interface I {
int Sq();}
```

Если класс наследует интерфейс, то в нем обязательно должны быть определены все члены, входящие в интерфейс.

```
Class A:I{
public int Sq(){return 6;} }
```

Класс может поддерживать несколько интерфейсов.

Ссылки на интерфейс – это ссылки на объекты классов, поддерживающих данный интерфейс. Если объявлена ссылка на интерфейс I, (I b;), а A – класс, поддерживающий интерфейс I, то можно написать b=new A(); и работать с вновь созданным объектом класса A через ссылку b. Если с объектом класса мы работаем через ссылку на интерфейс, то в этом случае будут доступны только члены, описанные в интерфейсе.

Ссылки на интерфейс могут быть формальными параметрами методов и могут возвращаться методами.

Поскольку поля классов рекомендуется создавать закрытыми, а работа с закрытыми членами класса осуществляется через открытые методы, то данная работа выполняется очень часто. Поэтому для ее облегчения создана специальная синтаксическая конструкция, которая называется свойство.

Свойство – это два метода, предназначенные для работы с закрытым полем и оформленные специальным образом.

Пример описания свойства:

```
private int svv;
public int sv
{get{return svv;}
set{svv=value;} }
```

Если в определении свойства отсутствует раздел get, то свойство предназначено только для записи, а если раздел set, то свойство предназначено только для чтения.

Свойство имеет еще одно преимущество перед обычными методами, обеспечивающими доступ к закрытому члену. Они вызываются специальным образом, имитирующим обращение к полю. Например:

```
f.sv=5;
int i=f.sv;
```

Методы, которые реализуют интерфейс, должны быть объявлены открытыми. Сигнатура типа в реализации метода должна в точности совпадать с сигнатурой типа, заданной в определении интерфейса. В классах, которые реализуют интерфейсы, можно определять дополнительные члены. Пример:

```
interface 1My_A {void meth(int x) ;
// В классе MyClass реализован интерфейс 1My_A class MyClass :
IMy_A {
int h;
public MyClass()
{ h = 5; }
public void meth(ini {h= x + x;
Console.WriteLine("MyClass.meth
}
class Classl:
{ public static void meth(int a)
{Console.WriteLine("Class 1.meth..."+a*a);} public static void Main()
{
MyClass ob = new MyClass();
Console.Write ("Вызов метода ob.meth(): "); ob.meth(3);
meth (3) ; } //реализация интерфейса IMy_A в классе Classl
}
```

Можно объявить ссылочную переменную интерфейсного типа, которая может ссылаться на любой объект, реализующий ее интерфейс. При вызове метода для объекта посредством интерфейсной ссылки будет выполнена та версия указанного метода, которая реализована этим объектом.

Элементы с одинаковыми именами или сигнатурой могут встречаться более чем в одном интерфейсе. В этом случае при множественном наследовании может возникнуть конфликт из-за неоднозначности ситуации, т. к. компилятор не может определить из контекста обращения к элементу, элемент какого именно из реализуемых интерфейсов требуется вызвать. Избежать неоднозначности можно с помощью бинарных операций is или as, которые позволяют убедиться, что объект поддерживает данный интерфейс, или выполнить приведение к данному интерфейсу.

Для этой же цели можно использовать так называемую явную реализацию, когда имя интерфейса явно указывается перед реализуемым элементом через точку. Такая реализация интерфейса является закрытой. Спецификаторы доступа при этом не указываются. К таким элементам можно обращаться в программе только через объект типа интерфейса.

Кроме того, явная реализация позволяет избежать конфликтов при множественном наследовании, если элементы с одинаковыми именами или сигнатурой встречаются более чем в одном интерфейсе.

Пример. Реализовано два интерфейса, причем оба объявляют метод с одним именем:

```
interface My A
{void meth(int x)
```

```
interface My_B
{
void meth(int x)
//В классе MyClass реализованы оба интерфейса
class MyClass IMy_A, IMy_B
{
int h;
public MyClass{}
{h=5; }
II Явным образом реализуем метод meth() интерфейса IMy_A
void IMy_A.meth(int x)
{
h = x;
Console.WriteLine("IMy_A.meth..." + h);
}
public void meth(int x)
{
h = x*x;
Console.WriteLine("meth..." + h);
}
}
class Demo
{
public static void Main()
{
MyClass ob new MyClass();
//Реализация интерфейса IMy_B
Console.Write("Вызов метода ob.meth(): ");
ob.meth(3);
Console.WriteLine ("Реализация интерфейса IMy_A");
(ob as IMy_A).meth(4);
Console.WriteLine("Вызов метода meth инт_ссылкой");
IMy_A a_ob=ob;
//интерфейсной ссылке присваиваем объект класса
Console.WriteLine("Вызов метода IMy_A.meth()");
a_ob.meth(3);
//реализация интерфейса IMy_A
}
}
```

Метод meth() имеет одинаковую сигнатуру в интерфейсах IMy_A и IMy_B. Поскольку единственный способ вызова явно заданного метода состоит в использовании интерфейсной ссылки, метод meth(), объявленный в интерфейсе IMy_A, создает ссылку на интерфейс IMy_A, а метод ob.meth() вызывает метод, объявленный в интерфейсе IMy B.

Интерфейс может не иметь или иметь сколько угодно интерфейсов-предков. В интерфейсе-потомке можно также указать элементы, переопределяющие унаследованные элементы с такой же сигнатурой. В этом случае перед элементом указывается ключевое слово new.

Любой класс, который реализует интерфейс, должен реализовать все методы, определенные этим интерфейсом, включая методы, которые унаследованы от других интерфейсов.

Класс наследует все методы своего предка, в том числе те, которые реализовывали интерфейсы. Он может переопределить эти методы с помощью спецификатора new, но обращаться к ним можно будет только через объект класса. Если использовать для обращения ссылку на интерфейс, вызывается не переопределенная версия.

Однако если интерфейс реализуется с помощью виртуального метода класса, после его переопределения в потомке любой вариант обращения (через класс или через интерфейс) приведет к одному и тому же результату.

Метод интерфейса, реализованный явным указанием имени, объявлять виртуальным запрещается.

Если класс наследует от класса и интерфейса, которые содержат методы

с одинаковыми сигнатурами, унаследованный метод класса воспринимается как реализация интерфейса.

В библиотеке классов .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки. Можно создавать и собственные классы, поддерживающие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами.

Во многих классах необходимо реализовать интерфейс IComparable или IComparer, поскольку они позволяют сравнить два объекта. Интерфейс IComparable состоит только из одного метода: int CompareTo(object v).

Этот метод сравнивает вызывающий объект со значением параметра v. Метод возвращает положительное число, если вызывающий объект больше объекта v, нуль, если два сравниваемых объекта равны, и отрицательное число, если вызывающий объект меньше объекта v. Метод CompareTo может сгенерировать исключение типа ArgumentException, если тип объекта v несовместим с вызывающим объектом.

В интерфейсе IComparer определен метод Compare (), который позволяет сравнивать два объекта:

int Compare(object vl, object v2)

Метод Compared) возвращает положительное число, если значение vl больше значения v2, отрицательное, если vl меньше v2, и нуль, если сравниваемые значения равны. Этот интерфейс можно использовать для задания способа сортировки элементов коллекции.

Порядок выполнения работы

Этап 1.

Размер фигуры определяется размером прямоугольника, в который она помещается, назовем его базовым. Расположение фигуры в этом прямоугольнике, определяется вариантом задания. При выводе фигуры на экран базовый прямоугольник не рисуется.

Создать форму Dialog для ввода размеров базового прямоугольника.

В класс формы Dialog добавить два открытых поля X, У типа int для хранения вводимой информации.

На форму Dialog поместить два текстовых поля и одну кнопку. При нажатии на кнопку информация, записанная в поля ввода, должна преобразовываться в тип int и записываться в поля X и У.

Описать иерархию классов, определяемую вариантом задания. Каждый класс иерархии определить в отдельном файле.

В класс Form1 добавить поле Figure, которое является ссылкой на абстрактный класс для нечетных вариантов и ссылкой на интерфейс для четных вариантов.

На форму Form1 поместить три кнопки и два поля ввода. Первые две из кнопок должны иметь названия, соответствующие именам производных классов, третья – надпись «Рисование».

При нажатии на одну из двух первых кнопок должны выполняться следующие действия:

– создаваться объект класса, имя которого соответствует надписи на кнопке. Адрес созданного объекта должен записываться в ссылку типа Figure;

– выводиться на экран форма Dialog. После ее закрытия, информация из ее полей X и У должна быть переписана в поля созданного объекта;

– при нажатии на кнопку «Рисование» должна выполняться проверка того, что ссылка Figure не является нулевой, и если нет, то на экран должна выводиться фигура. Место положения фигуры должно определяться информацией, находящейся в полях ввода формы Form1.

Задания для самостоятельной работы.

1 Иерархия включает абстрактный базовый класс под названием «Фигура», и два его подкласса. Абстрактный класс содержит абстрактный метод рисования фигуры и два поля или два свойства, определяющие размер фигуры. В подклассах переопределяется абстрактный метод.

2 Иерархия включает интерфейс и два класса, поддерживающих этот интерфейс. Интерфейс содержит метод рисования фигуры и два свойства, определяющие размер фигуры.

Варианты фигур:

– Эллипс, вписанный в базовый прямоугольник;

– Прямоугольник, совпадающий с базовым прямоугольником;

– Шестиугольник (не обязательно правильный), расположение шестиугольника в базовом прямоугольнике студент определяет самостоятельно;

– Пятиугольник (не обязательно правильный), расположение пятиугольника в базовом прямоугольнике студент определяет самостоятельно;

– Ромб, вершины которого лежат на серединах базового прямоугольника;

– Прямоугольный треугольник, катеты которого совпадают со сторонами базового прямоугольника;

– Равнобокая трапеция. Основания трапеции должны лежать на сторонах базового прямоугольника.

Этап 2.

Выполнить задания, используя для хранения экземпляров разработанных классов стандартную коллекцию ArrayList. Во всех классах реализовать интерфейсы IComparable и IComparer. Перегрузить операции отношения для реализации сравнения объектов по указанному полю. Результат вывести на экран. Организовать вывод коллекции.

1 Составить список студентов, включающий ФИО, курс, группу, дату и результат забега. Вывести в новый список информацию о студентах, показавших три лучших результата в забеге. Если окажется, что некоторые студенты получили такие же высокие результаты, то добавить их к списку победителей, отсортировать по результатам.

2 Составить инвентарную ведомость склада, включив следующие данные: вид продукции, стоимость, сорт, количество. Вывести в новый список информацию о той продукции, количество которой меньше заданной величины, отсортировав ее по количеству продукции на складе.

Этап 3.

1 Создать абстрактный базовый класс Figure с виртуальными методами вычисления площади и периметра. Создать производные классы Rectangle (Прямоугольник), Circle (Круг), Trapezium (Трапеция) со своими функциями площади и периметра. Самостоятельно определить, какие поля необходимы, какие из них можно задать в базовом классе, а какие – в производных. Площадь трапеции $S = (a + b) \times h / 2$.

2 Создать абстрактный базовый класс Body (Тело) с виртуальными функциями вычисления площади поверхности и объема. Создать производные классы Parallelepiped (Параллелепипед) и Ball (Шар) со своими функциями площади поверхности и объема.

3 Создать абстрактный класс Currency (Валюта) для работы с денежными суммами. Определить виртуальные функции перевода в рубли и вывода на экран. Реализовать производные классы Dollar (Доллар) и Euro (Евро) со своими функциями перевода и вывода на экран.

Вопросы для самоконтроля

1 Какой класс называется абстрактным?

2 Какие члены может содержать абстрактный класс?

3 Какие члены может содержать интерфейс?

4 Может ли интерфейс содержать поле?

5 Что такое свойство?

6 Лабораторная работа № 6. Делегаты и события

Цель работы: приобрести навыки событийного программирования и его программной реализации.

Теоретические основы

Делегат – это вид класса, предназначенный для хранения ссылок на методы. Делегат представляет собой тип данных. Его базовым классом является класс System.Delegate. Наследовать от делегата нельзя. Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса.

Описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

[атрибуты] [спецификаторы] delegate <тип> <имя_делегата>([параметры])

Спецификаторы делегата имеют тот же смысл, что и для класса, причем допускаются только спецификаторы new, public, protected, internal и private.

Тип описывает возвращаемое значение методов, вызываемых с помощью делегата, а необязательными параметрами делегата являются параметры этих методов.

Делегат может хранить ссылки на несколько методов и вызывать их по очереди. Пример описания делегата:

```
public delegate void D(int i);
```

Делегат может вызывать либо метод экземпляра класса, связанный с объектом, или статический метод, связанный с классом.

При реализации делегата создается ссылка типа делегата и ей присваивается ссылка на метод, который передается делегату в качестве параметра, причем используется только имя метода (параметры не указываются). Затем этот метод вызывается посредством экземпляра делегата.

Таким образом, вызов экземпляра делегата трансформируется в обращение к методу, на который он ссылается при вызове. И, следовательно, решение о вызываемом методе принимается во время выполнения программы, а не в период компиляции.

Многоадресатная передача (multicasting) – это способность создавать список вызовов (или цепочку вызовов) методов, которые должны автоматически вызываться при вызове делегата. Для этого достаточно создать экземпляр делегата, а затем для добавления методов в эту цепочку использовать оператор "+=". Для удаления метода из цепочки используется оператор " –= ".

Делегат с многоадресатной передачей имеет одно ограничение: он должен возвращать тип void.

Пример:

```
using System;
delegate void strMod(ref string str); // Объявляем делегат.
class StringOps {
```

```
// Метод заменяет пробелы дефисами
static void replaceSpaces(ref string a) {
Console.WriteLine("Замена пробелов дефисами.");
a = a.Replace(‘‘,’–‘ ) ;
}
// Метод удаляет пробелы
static void removeSpaces(ref string a) {
string temp = "";
int i;
Console.WriteLine("Удаление пробелов.");
for(i=0; i < a.Length; i++)
if(a[i] != ‘ ’) temp += a[i];
a = temp;
}
// Метод реверсирует строку
static void reverse(ref string a) {
string temp = "";
int i, j;
Console.WriteLine("Реверсирование строки.");
for(j=0, i=a.Length-l; i >= 0; i—, j++)
temp += a[i];
a = temp;
}
public static void Main() {
// Создаем экземпляры делегатов.
strMod strOp;
strMod replaceSp = new strMod(replaceSpaces);
strMod removeSp = new strMod(removeSpaces);
strMod reverseStr = new strMod(reverse);
string str = "Это простой тест.";
// Организация многоадресатной передачи.
strOp = replaceSp;
strOp += reverseStr;
// Вызов делегата с многоадресатной передачей.
strOp(ref str);
Console.WriteLine("Результирующая строка: " + str);
Console.WriteLine();
// Удаляем метод замены пробелов и добавляем метод их удаления.
strOp -= replaceSp;
strOp += removeSp;
str = "Это простой тест."; // Восстановление исходной строки.
// Вызов делегата с многоадресатной передачей.
strOp(ref str);
```

```
Console.WriteLine("Результирующая строка: " + str);
Console.WriteLine();
}}
```

На основе делегатов построено важное средство C#: событие (event). Синтаксис события:

[атрибуты][спецификаторы]event <тип> <имя>

Для событий применяются спецификаторы new, public, protected, internal, private, static, L virtual, sealed, override, abstract и extern. Событие может быть статическим (static), тогда оно связано с классом в целом, или обычным – в этом случае оно связано с экземпляром класса.

Тип события – это тип делегата, на котором основано событие. Пример описания делегата и соответствующего ему события:

```
public delegate void Del (object o); //объявление делегата class A
{public event Del Oops; //объявлениесобытия
}
```

События работают следующим образом. Объект, которому необходима информация о некотором событии, регистрирует обработчик для этого события, сигнатура которых соответствует типу делегата. Когда ожидаемое событие происходит, вызываются все зарегистрированные обработчики.

Если в качестве обработчика используется статический метод, уведомление о событии применяется к классу (и неявно ко всем объектам этого класса). Если же в качестве обработчика событий используется метод экземпляра класса, события посылаются к конкретным экземплярам этого класса.

Добавлять обработчики в список или удалять их можно с помощью операторов "+=" и " –=

Внутри класса, в котором описано событие, с ним можно обращаться, как с обычным полем, имеющим тип делегата: использовать операции отношения, присваивания и т. д. Значение события по умолчанию – null.

Подобно делегатам события могут предназначаться для многоадресатной передачи. В этом случае на одно уведомление о событии может отвечать несколько объектов. Пример:

```
//Демонстрация использования события, предназначенного для много-
адресатной передачи.
using System;
//Объявляем делегат для события
delegate void Del();
//Объявляем класс события
class Et
{
}
public event Del SE;
//Этот метод вызывается для генерирования события,
public void OnSE()
{ if (SE != null) SE(); }
```

```
class X
public void Xh()
{
{ Console.WriteLine ("Событие, полученное объектом X."); }
}
class Y
{
public void Yh()
{ Console.WriteLine ("Событие, полученное объектом У."); }
class EventDemo
{
static void h()
{ Console.WriteLine ("Событие, полученное классом EventDemo."); }
public static void Main()
{
Et evt = new Et();
X xob = new X();
Y yob new Y();
//Добавляем обработчики в список
evt.SE += new Del(h);
evt.SE += new Del(x0b.Xh);
evt.SE += new Del (yob.yff);
//Генерируем событие,
evt.OnSE();
Console.WriteLine();
//Удаляем один обработчик
evt.SE -=
nev Del (xob.Xh);
evt.OnSE();
}
}
```

В библиотеке .NET описано огромное количество стандартных делегатов, предназначенных для реализации механизма обработки событий. Большинство этих классов оформлено по одним и тем же правилам:

– имя делегата заканчивается суффиксом EventHandler;

– делегат получает два параметра: первый параметр задает источник события и имеет тип object; второй параметр задает аргументы события и имеет тип EventArgs или производный от него.

Если обработчикам события требуется специфическая информация о событии, то для этого создают класс, производный от стандартного класса EventArgs, и добавляют в него необходимую информацию. Если делегат не использует такую информацию, можно обойтись стандартным классом делегата System. EventHandler.

Имя обработчика события принято составлять из префикса On и имени события.

Порядок выполнения работы

1 Рассмотрим пример обработки события, связанного с нажатием клавиши на клавиатуре. Событие называется KeyPress, и при каждом нажатии клавиши оно генерируется посредством вызова метода OnKeyPress().

Программа начинается с объявления класса KeyEventArgs, который используется для передачи сообщения о нажатии клавиши обработчику событий. Затем делегат KeyHandler определяет обработчик для событий, связанных с нажатием клавиши на клавиатуре. Эти события инкапсулируются в классе KeyEvent.

Программа для обработки нажатий клавиш создает два класса: ProcessKey и CountKeys. Класс ProcessKey включает обработчик с именем keyhandler(), который отображает сообщение о нажатии клавиши. Класс CountKeys предназначен для хранения текущего количества нажатых клавиш.

В методе Main() создается объект класса KeyEvent. Затем создаются объекты классов ProcessKey и CountKeys, а ссылки на их методы keyhandler() добавляются в список вызовов, реализуемый с помощью событийного объекта kevt.KeyPress. Затем начинает работать цикл, в котором при каждом нажатии клавиши вызывается метод kevt.OnKeyPress(), в результате чего зарегистрированные обработчики уведомляются о событии:

```
using System;
// Выводим собственный класс EventArgs, который будет хранить код клавиши.
class KeyEventArgs : EventArgs
{
public char ch;
}
// Объявляем делегат для события.
delegate void KeyHandler(object source, KeyEventArgs arg);
// Объявляем класс события, вызванного нажатием клавиши на клавиатуре.
class KeyEvent
{
public event KeyHandler KeyPress;
// Этот метод вызывается при нажатии какой-нибудь клавиши.
public void OnKeyPress(char key)
{
KeyEventArgs k = new KeyEventArgs();
if (KeyPress != null)
{
k.ch = key;
KeyPress(this, k);
}
}
```

```
}
// Класс, который принимает уведомления о нажатии клавиши.
class ProcessKey
{
public void keyhandler(object source, KeyEventArgs arg)
{
Console.WriteLine("Получено сообщение о нажатии клавиши: " + arg.ch);
}
}
// Еще один класс, который принимает уведомления о нажатии клавиши.
class CountKeys
{
public int count = 0;
public void keyhandler(object source, KeyEventArgs arg)
{
count++;
}
}
namespace Primer11
{
class Program
{
static void Main()
{
Console. Title = “Демонстрация события о нажатии клавиши";
Console.BackgroundColor = ConsoleColor.White;
Console.Clear();
Console.ForegroundColor = ConsoleColor.Black;
KeyEvent kevt = new KeyEvent();
Processkey pk = new Processkey();
Countkeys ck = new CountKeys();
char ch;
kevt.KeyPress += new KeyHandler(pk.keyhandler);
kevt.KeyPress += new KeyHandler(ck.keyhandler);
Console.WriteLine ("Введите несколько символов. "Для останова введите точку.");
do
{
ch= (char) Console.Read();
kevt.OnKeyPress(ch);
} while (ch != '.');
Console.WriteLine("Было нажато + ck.count + клавиш.");
```

```
Console.Write("Для завершения работы приложения нажмите клавишу <Enter>");
Console.Read();
}
}
}
```

2 Написать делегат, с помощью которого реализуется сортировка книг. Книга представляет собой класс с полями Название, Автор, Издательство и конструктором.

Создать класс, являющийся контейнером для множества книг (массив книг). В этом классе предусмотреть метод сортировки книг. Критерий сортировки определяется экземпляром делегата, который передается методу в качестве параметра. Каждый экземпляр делегата должен сравнивать книги по какому-то одному полю: названию, автору, издательству.

3 Разработать приложение «Название автомобиля», с помощью которого можно отслеживать изменения в названии автомобиля. Обработчик события должен выдавать сообщение “Название изменилось!” при внесении изменения названия автомобиля.

4 Модифицировать приложение «Название автомобиля» таким образом, чтобы пользователю дать возможность принять решение об изменении или отказе от изменения названия автомобиля.

Вопросы для самоконтроля

1 Что такое делегат?
2 Как можно подписаться на событие?
3 Каким образом можно вызвать метод, подписавшийся на событие?

7 Лабораторная работа № 7. Обработка исключений

Цель работы: приобрести навыки обработки исключений в программах.

Теоретические основы

Механизм исключений (Exception), предусмотренный в C#, упрощает обработку ошибок. Вместо того чтобы проверять значение, возвращаемое функциями и методами, вы можете использовать для обнаружения и обработки ошибок структурные операторы, такие как try и catch.

Методы стандартных библиотек классов возбуждают Exception при возникновении ошибочных ситуаций. Заключив «ненадежный» с точки зрения возникновения ошибок код в блок try, вы можете перехватить и обработать Exception в блоке catch.

Заметим, что Exception возникают и в таких распространенных ситуациях, как выход за пределы массива или строки в процессе индексации, деление на

нуль, попытка использования ссылки со значением null, недопустимое преобразование классов и т. д. С помощью Exception все подобные ошибки нетрудно обнаружить и обработать во время выполнения приложения.

К механизму обработки Exception в следующие ключевые слова: try, catch, throw и finally. Схема работы этого механизма следующая. Вы пытаетесь (try) выполнить блок кода, и, если при этом возникает ошибка, система возбуждает (throw) Exception, которое в зависимости от его типа вы можете перехватить (catch) или передать обработчику по-умолчанию (finally).

Общая форма блока обработки Exception ():

```
try {
// блок кода }
catch (ТипException1) {
// обработчик Exception типа ТипException1 }
catch (ТипException2) {
// обработчик Exception типа ТипException2
 throw(e)   // повторное возбуждение Exception }
finally {
}
```

Пример:

```
try {
line = Console.KeadLine();
if (line.Length() == 0)
throw new EmptyLineException("Строка, считанная с консоли, пустая!");
console.printLine("Привет, %s!" % line);
}
catch (EmptyLineException exception) {
  console.printLine("Привет!");
}
catch (Exception exception) {
console.printLine("Ошибка: " + exception.message());
}
else {
  console.printLine("Программа выполнилась без исключительных ситуаций");
}
finally {
console.printLine("Программа завершается");
}
```

Операторы try можно вкладывать друг в друга аналогично тому, как можно создавать вложенные области видимости переменных. Если у оператора try низкого уровня нет раздела catch, соответствующего возбужденному Exception будут проверены разделы catch внешнего оператора try. Пример, в котором два оператора try вложены друг в друга посредством вызова метода:

```
class MultiNest {
static void procedure() {
```

```
try {
int c[] = { 1 };
c[42] = 99;
}
catch(IndexOutOfRangeException e) {
System.out.println("array index oob: " + e);
} }
public static void main(String args[]) {
try {
int a = args.length();
System.out.println("a = " + a);
int b = 42 / a;
procedure();
}
catch (ArithmeticException e) {
System.out.println("div by 0: " + e);
}
} }
```

Хотя встроенные Exception обрабатывают большинство частых ошибок, вероятно, вам потребуется создать ваши собственные типы Exception для обработки ситуаций, специфичных для ваших приложений. Это достаточно просто сделать: просто определите подкласс Exception.

В следующем примере объявляется новый подкласс Exception, который затем используется для сигнализации об ошибочной ситуации в методе. Он переопределяет метод toString (), позволяя отобразить тщательно настроенное описание Exception. Пример:

```
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException!" + detail ();"]";
}
}
class ExceptionDemo {
static void compute(int a)
{
System.out.println("Вызван compute(" + a + ")");
if(a > 10)
throw new MyException(a);
System.out.println("Нормальное завершение");
}
public static void main(String args[]) {
try {
```

```
compute(1);
compute(20);
}
catch (MyException e) {
System.out.println("Перехвачено " + e) ;
}
}
}
```

В этом примере определен подкласс Exception с именем MyException. Этот подкласс достаточно прост: он имеет только конструктор и перегруженный метод toString. Класс ExceptionDemo определяет метод под названием compute (), который возбуждает объект MyException. Это Exception возбуждается, когда целочисленный параметр compute () принимает значение больше 10.

Метод main () устанавливает обработчик Exception MyException, затем вызывает compute () с правильным параметром (меньше 10), и с неправильным, чтобы продемонстрировать оба пути выполнения кода. Ниже показан результат.

```
Вызван compute(1)
Нормальное завершение
Вызван compute(20)
Перехвачено MyException[20]
```

Выше были рассмотрены способы обработки Exception и передача Exception в вызывающий метод. Однако имеется возможность совмещения этих подходов, т. е. Exception может быть обработано и выброшено в вызывающий метод:

```
public String сформироватьОписание() {
try {
if (nazvanie = = null || nazvanie.equals(“”)) throw new MyException (“Не указано название дерева”);
} catch (MyException ex) {
System.out.println(ex.getMessage());
throw ex; // повторный выброс Exception (операция return отсутствует)
}
}
```

Throw можно также использовать в блоке catch для повторного создания Exception, обрабатываемого в блоке catch. В этом случае оператор throw не принимает операнд Exception. Это наиболее полезно, когда метод передает аргумент от вызывающего объекта в другой метод библиотеки, а метод библиотеки создает Exception, которое должно быть передано вызывающему объекту. Например, в следующем примере повторно создается Exception NullReferenceException, возникающее при попытке получить первый символ неинициализированной строки.

Exception, перехваченное в одном блоке catch, может быть повторно сгенерировано в другом блоке, чтобы быть перехваченным во внешнем блоке catch. Наиболее вероятной причиной для повторного генерирования Exception служит предоставление доступа к Exception нескольким обработчикам. Допустим, что

один обработчик оперирует каким-нибудь одним аспектом Exception, а другой обработчик – другим его аспектом. Для повторного генерирования Exception достаточно указать оператор throw без сопутствующего выражения, как в приведенной ниже форме:

```
throw;
```

Не следует, однако, забывать, что когда Exception генерируется повторно, то оно не перехватывается снова тем же самым блоком catch, а передается во внешний блок catch. Пример:

```
using System;
namespace ConsoleApplication1
{
class Program
{
static void Del(int x, int y)
{
try
{
int result = x / y;
}
catch(DivideByZeroException)
{
Console.WriteLine("Деление на ноль!");
throw;
}
}
static void Main()
{
try
{
Del(5, 0);
}
catch (DivideByZeroException)
{
Console.WriteLine("Программная ошибка!");
}
Console.ReadLine();
}
}
}
```

Порядок выполнения работы

1 Смоделировать структуру предприятия, приведенную в таблице 7.1.

1.1 Обработать все Exception с помощью блока try…catch(Exception …) в методе «рассчитать зарплату» классов Штатный сотрудник и Сотрудник

по контракту. При возникновении Exception выводить на экран сообщение об ошибке.

Описать собственный класс Exception PremiyaException. В методе «рассчитать зарплату» класса Штатный сотрудник выбрасывать собственное Exception типа PremiyaException при отрицательном значении свойства Премия. В этом же методе обработать PremiyaException в блоке catch. При возникновении Exception выводить сообщение об ошибке на экран.

В основной программе проверить работу блока обработки Exception метода «рассчитать зарплату».

Таблица 7.1 – Структура предприятия

Класс	Свойство и метод
Фирма	Название (get, set)
Отдел	Название (get, set) Количество сотрудников (get, set)
Сотрудник	ФИО (get, set) Должность (get, set) Оклад (get, set) Рассчитать зарплату()
Штатный сотрудник	Премия (get, set) Рассчитать зарплату()
Сотрудник по контракту	Рассчитать зарплату()

1.2 Описать собственный класс Exception OkladException. В конструкторе реализовать проверку значения оклада, при отрицательном значении выбрасывать собственное Exception типа OkladException. Обработать Exception OkladException с помощью блока try…catch, в блоке обработки Exception вывести на экран сообщение «Невозможно создать сотрудника – указан отрицательный оклад: <оклад> » и повторно создать Exception.

В основной программе (main) обработать вызов конструктора класса Сотрудник и проверить работу обработчика Exception.

2 Смоделировать структуру библиотеки, приведенную в таблице 7.2.

2.1 Обработать все Exception с помощью блока try…catch(Exception …) в методе «сформировать описание» классов Книга и Журнал. При возникновении Exception выводить на экран сообщение об ошибке.

Описать собственный класс Exception BookException. В методе «сформировать описание» классов Книга и Журнал осуществить проверку значений свойств Название и Автор – выбрасывать собственное Exception типа BookException при значении null или пустой строке (“”). В этом же методе обработать BookException в блоке catch. При возникновении Exception выводить сообщение об ошибке на экран.

В основной программе (main) проверить работу блока обработки Exception метода «сформировать описание».

2.2 Описать собственный класс Exception GodIzdaniyaException. В конструкторе реализовать проверку значения года издания, при отрицательном или

нулевом значении выбрасывать собственное Exception типа GodIzdaniyaException. Обработать Exception GodIzdaniyaException с помощью блока try…catch, в блоке обработки Exception вывести на экран сообщение «Невозможно создать издание – указан некорректный год издания: <год издания>» и повторно создать Exception.

В основной программе (main) обработать вызов конструктора класса Издания и проверить работу обработчика Exception.

Таблица 7.2 – Структура библиотеки

Класс	Свойство
Библиотека	Название (get, set)
Отдел (по жанрам)	Название жанра (get, set) Количество изданий (get, set)
Издание	Название (get, set) Автор (get, set) Год издания (get, set) Сформировать описание()
Книга	Резюме (get, set) Сформировать описание()
Журнал	Статьи (get, add) Сформировать описание()

3 Смоделировать структуру автосалона, приведенную в таблице 7.3.

3.1 Обработать все Exception с помощью блока try…catch(Exception …) в методе «рассчитать стоимость заказа» классов Заявка на приобретение со стенда и Заявка на отложенную поставку. При возникновении Exception выводить на экран сообщение об ошибке.

Описать собственный класс Exception KolichestvoAvtomobileyException. В методе «рассчитать стоимость заказа» классов Заявка на приобретение со стенда и Заявка на отложенную поставку выбрасывать собственное Exception типа KolichestvoAvtomobileyException, если количество автомобилей в заявке равно нулю. В этом же методе обработать KolichestvoAvtomobileyException в блоке catch. При возникновении Exception выводить сообщение об ошибке на экран.

В основной программе (main) проверить работу блока обработки Exception метода «рассчитать стоимость заказа».

3.2 Описать собственный класс Exception FIOException. В конструкторе реализовать проверку значения свойства ФИО покупателя, при значении null или пустой строке выбрасывать собственное Exception типа FIOException. Обработать Exception FIOException с помощью блока try…catch, в блоке обработки Exception вывести на экран сообщение «Невозможно создать заявку – не указано ФИО покупателя» и повторно создать Exception.

В основной программе (main) обработать вызов конструктора класса Заявка на покупку и проверить работу обработчика Exception.

Таблица 7.3 – Структура автосалона

Класс	Свойство
Автосалон	Название (get, set)
Автомобиль	Марка (get, set) Макс. количество пассажиров (get, set) Стоимость (get, set) Количество на складе (get, set) Наличие boolean (get, set)
Заявка на покупку	ФИО покупателя (get, set) Номер телефона (get, set) Автомобили (add, remove, get) Рассчитать стоимость заказа ()
Заявка на приобретение со стенда	Рассчитать стоимость заказа ()
Заявка на отложенную поставку	Процент скидки (get, set) Рассчитать стоимость заказа ()

Вопросы для самоконтроля

1 Объясните назначение конструкции try-catch-finally.
2 Как генерировать исключения?
3 Как создать пользовательское исключение?
4 Как обработать несколько исключений?
5 Для чего используются операторы checked и unchecked?

Список литературы

1 **Гагарина, Л. Г.** Технология разработки программного обеспечения : учебное пособие / Л. Г. Гагарина, Е. В. Кокорева, Б. Д. Сидорова-Виснадул ; под ред. Л. Г. Гагариной. – Москва : ФОРУМ; ИНФРА-М, 2019. – 400 с.

2 **Шишкина, В. В.** Лабораторный практикум по основам объектно-ориентированного программирования : методические указания и задания для выполнения лабораторных работ по дисциплине «Основы объектно-ориентированного программирования» / В. В. Шишкина. – Ульяновск : УлГТУ, 2009. – 20 с.

3 **Ефимова, Ю. В.** Методы программирования: методические указания к лабораторным работам для студентов направления подготовки 09.03.01 / Ю. В. Ефимова. – Чистополь: КАИ, 2019. – 52 с.