ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В рамках дисциплины «Основы объектно-ориентированного программирования» предлагается выполнить 5 лабораторных работ, направленных на приобретение навыков объектно-ориентированного программирования (ООП). Лабораторные работы выполняются в среде Microsoft Visual Studio 2008 в рамках консольного приложения.

Первые две работы посвящены освоению основных конструкций, используемых в ООП. Третья лабораторная работа посвящена работе с массивами. В рамках четвертой лабораторной работы происходит знакомство со специальной конструкцией класса C# – индексаторами, а так же со статическими полями и параметризованными классами. Пятая лабораторная работа посвящена переопределению операций.

Для создания консольного приложения в Microsoft Visual Studio 2008 необходимо сделать следующее.

В меню «File»(«Файл») выбрать пункт «New»(«Новый») и в нем выбрать подпункт «Project»(«Проект»), как показано на рисунке 1.

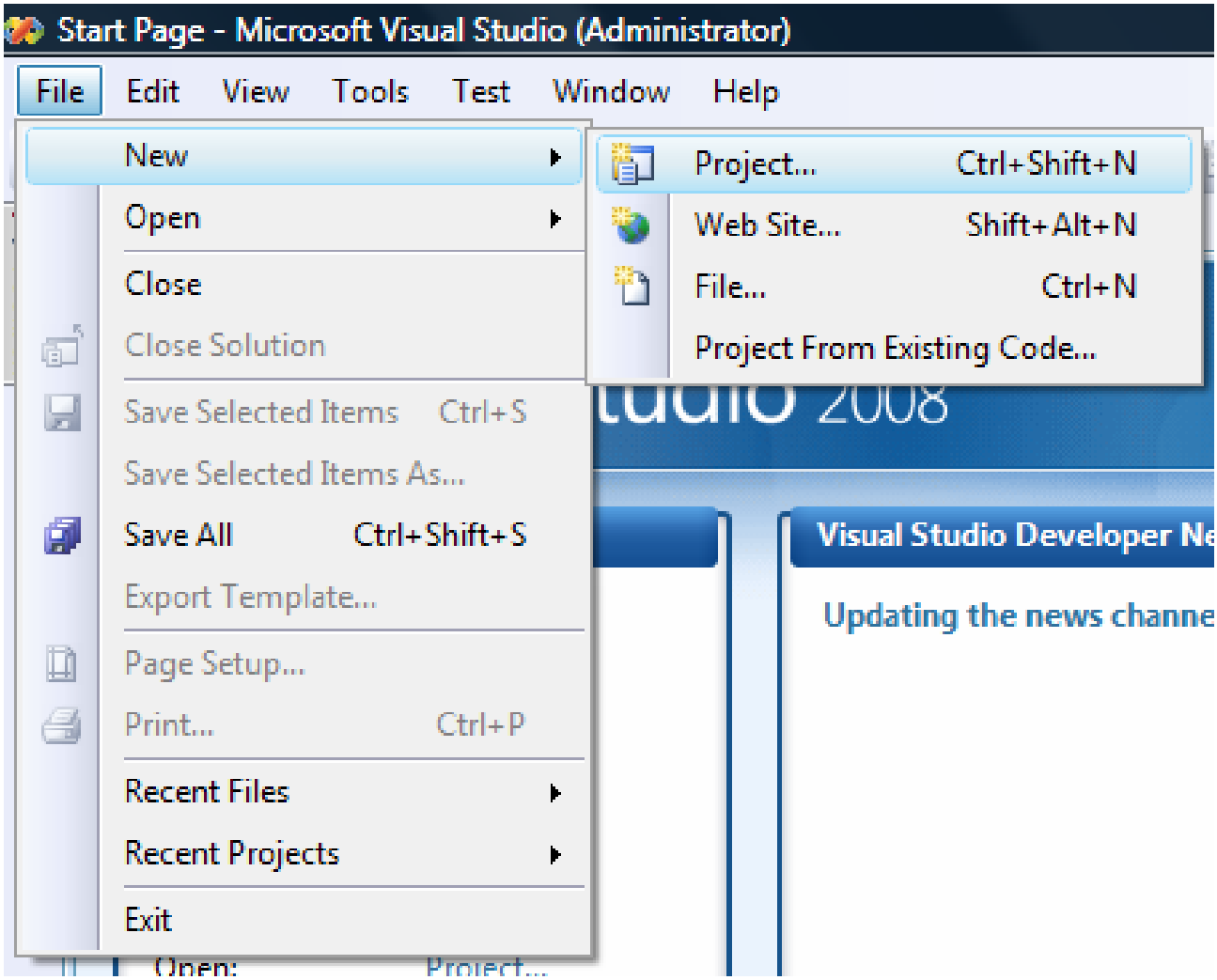

Рис.1

Далее в диалоговом окне, показанном на рисунке 2 в разделе Project types необходимо выбрать пункт Visual C# и подпункт Windows. В разделе Templates выбрать Console Application. В нижней части окна необходимо задать имя проекта и папку, где он будет сохраняться. При этом необходимо следить, чтобы папка, в которую сохраняется проект, была доступна для записи.

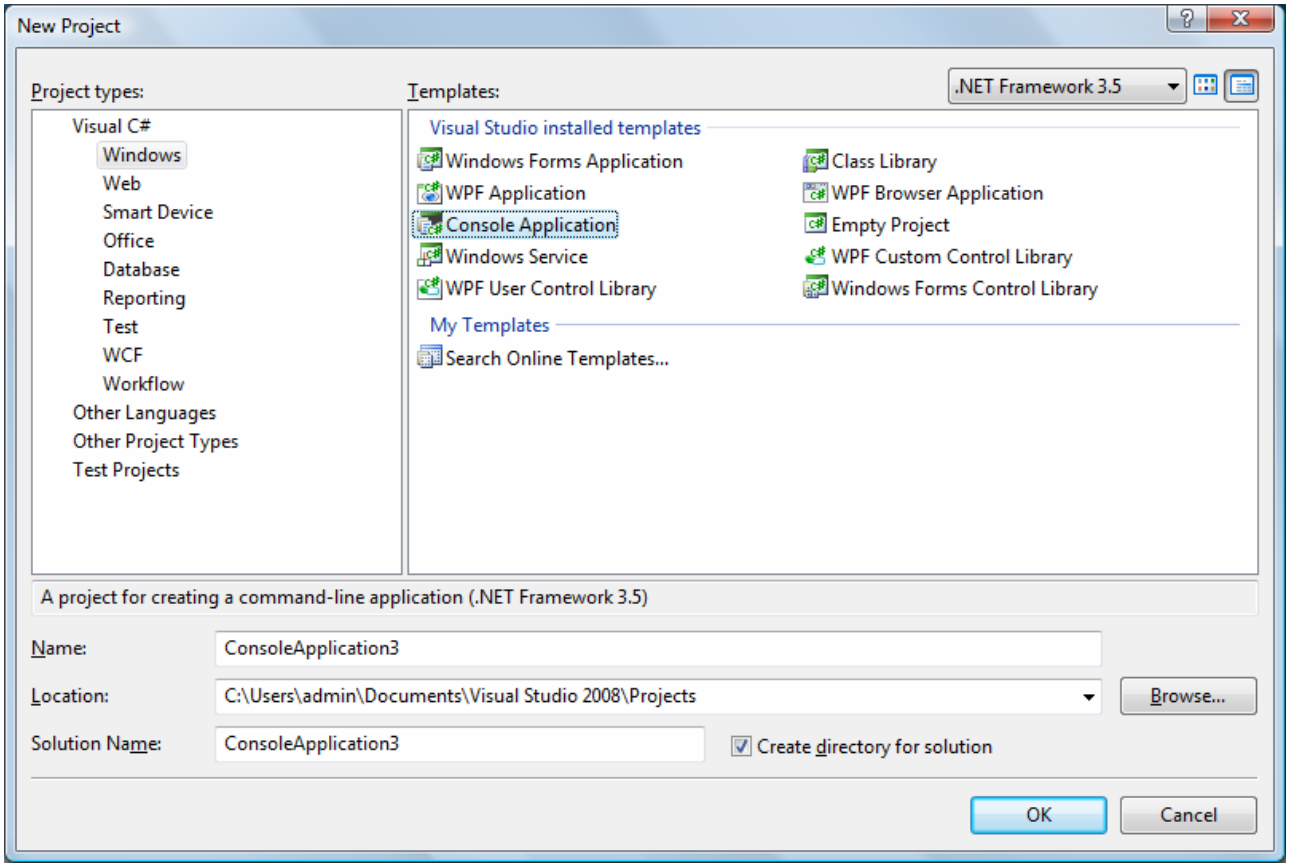

Рис. 2

Если все сделано правильно, то в итоге на экране появится окно, изображенное на рисунке 3.

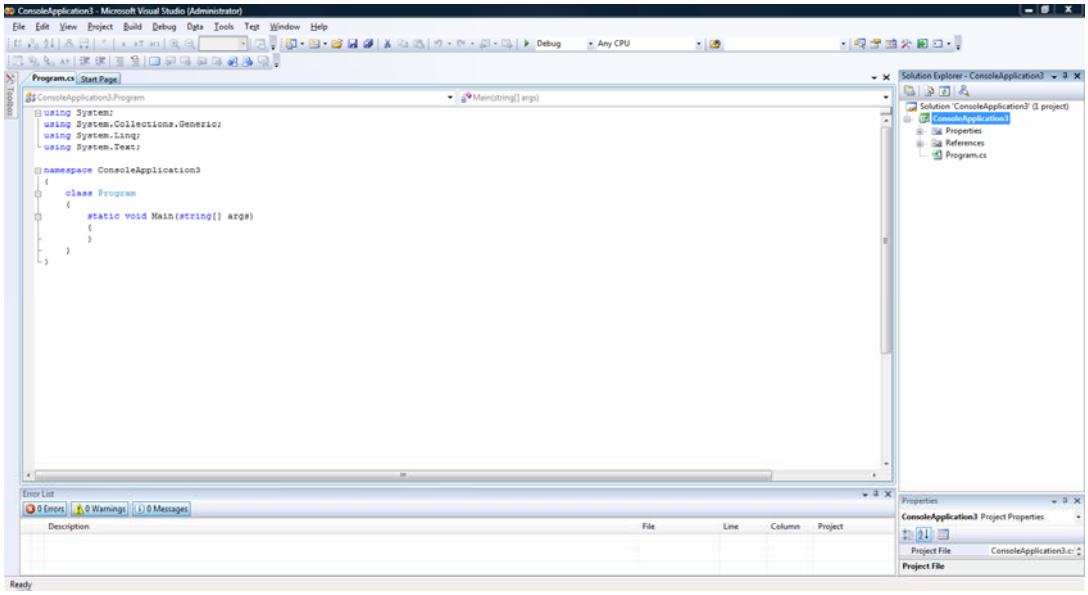

Рис. 3

Ввод данных с клавиатуры осуществляется посредством метода ReadLine() класса Console. Данный метод возвращает строку, которую ввел с клавиатуры пользователь.

Вывод данных на экран осуществляется посредством метода WriteLine() класса Console.

Пример консольного приложения:

```
using System;
public class Program
{
public static void Main(string[] args)
{
Console.WriteLine(«Введите свое имя»);
string st=Console.ReadLine();
Console.WriteLine(«Привет {0}»,st);
Console.ReadLine();
}
}
```

ЛАБОРАТОРНАЯ РАБОТА № 1. КЛАССЫ И ОБЪЕКТЫ. ИНКАПСУЛЯЦИЯ

Цель лабораторной работы: познакомиться с созданием классов и объектов на языке C#. Изучить основные конструкции, используемые при построении классов и объектов. Разобраться с понятием инкапсуляции.

Теоретические основы

С теоретической точки зрения:

Класс – это тип, описывающий устройство объектов.

Поля – это переменные, принадлежащие классу.

Методы – это функции (процедуры), принадлежащие классу.

Объект – это экземпляр класса, сущность в адресном пространстве компьютера.

Можно сказать, что класс является шаблоном для объекта, описывающим его структуру и поведение. Поля класса определяют структуру объекта, методы класса – поведение объекта.

С точки зрения практической реализации (в самом тексте программы) класс является типом данных, а объект – переменной этого типа.

Объявление класса состоит из двух частей: объявление заголовка класса и объявление тела класса. Заголовок класса состоит из модификатора доступа, ключевого слова class и имени самого класса. Тело класса – есть конструкция, заключенная в фигурные скобки и содержащая объявление полей и методов, принадлежащих классу.

Пример объявления класса:

```
public class MyClass { int a; }
```

Объявление объекта (создание объекта как экземпляра класса) состоит из двух частей: создание переменной-ссылки на область памяти, в которой будет располагаться объект, выделение памяти для объекта и заполнение этой памяти начальными значениями, иначе говоря инициализация данной переменной-ссылки. Объявление переменной-ссылки, а иными словами объекта, подчиняется общему правилу объявления переменных в C#. Переменные могут объявляться в любом месте в теле методов, за исключением тел условных операторов. Переменная, объявленная вне тела метода, но внутри тела класса, становится полем.

<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 объявление свойства в классе MyClass:

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

Пример использование 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.
```

Так как программа на языке C# может иметь множество классов со множеством методов, то необходимо каким-то образом определять точку, откуда начнется выполнятся программа. Эта точка называется точкой входа и представляет собой метод любого класса, объявленный с заголовком static void Main(string[] args). Точка входа может принадлежать любому классу, из описанных в программе, единственно, что она должна быть в программе одна.

Задание к лабораторной работе

В рамках консольного приложения создать класс A с полями a и b и свойством c. Свойство – значение выражения над полями a и b (выражение и типы полей – см. вариант в таблице 1). Поля инициализировать при объявлении класса. Конструктор оставить по умолчанию. Проследить, чтобы поля a и b

напрямую в других классах были недоступны. Создать класс Programm с одним методом – точкой входа. В теле метода создать объект класса A, вывести на экран значение свойства c.

Таблица 1. Варианты заданий к лабораторной работе №1

Вариант	Тип полей a b,	Выражения
1	float	/,-
2	float	/=, +
3	int	*,-
4	int	*=,+
5	int	(постфиксный)++,*=,-
6	int	%=,+
7	float	/=,(постфиксный)--,*
8	float	*=,++(префиксный),/
9	decimal	+=,-
10	decimal	(постфиксный)++,-
11	decimal	++(префиксный),-
12	decimal	--(префиксный),+
13	int	*=,(постфиксный)++,-
14	int	+=,--(префиксный),-
15	int	-=,(постфиксный)++,-
16	float	/=,*
17	float	*=,/
18	decimal	(постфиксный)--,+
19	decimal	-=,+
20	decimal	*=,-
21	decimal	+=,-
22	float	%=,/
23	float	%=,*
24	int	--(префиксный),++(префиксный),+
25	byte	(постфиксный)--,(постфиксный)++,-
26	byte	*=,+
27	byte	*=,-

Контрольные вопросы

1. Что такое класс?
2. Что такое объект?
3. Как связаны между собой классы и объекты в программе?
4. Что такое инкапсуляция?
5. За счет чего реализуется защита от несанкционированного доступа к данным?
6. Чем отличаются поля от переменных?

7. Что такое свойство?
8. Какие методы есть у свойства?
9. Что делают эти методы?
10. Что такое точка входа?

ЛАБОРАТОРНАЯ РАБОТА № 2. КОНСТРУКТОРЫ, ПОЛИМОРФИЗМ И НАСЛЕДОВАНИЕ

Цель лабораторной работы: познакомиться с созданием конструкторов для классов. Изучить механизмы наследования и полиморфизма. Познакомиться с управляющими операторами языка C#.

Теоретические основы

Конструктор – специальный метода объекта, решающий задачу начальной инициализации полей объекта и объявленный следующим образом: для этого метода всегда используется модификатор доступа public, нет типа возвращаемого значения (нет даже void), имя метода совпадает с именем класса.

Пример объявления конструктора в классе MyClass:

```
public class MyClass
{
int a;
public MyClass(){a=0;} //Конструктор без параметров. Инициализирует поле a нулем
public MyClass(int a){this.a=a;} //Конструктор с параметром типа int.
public MyClass(char a){this.a=a;} //Конструктор с параметром типа char.
}
```

Реализация для одного класса нескольких конструкторов является примером полиморфизма. Полиморфизм – механизм, позволяющий использовать одно имя для реализации схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. В более общем смысле, в основе полиморфизма лежит идея «использовать один интерфейс для множества методов». Для компилятора полиморфные функции должны различаться принимаемыми параметрами. Это различие может быть по их количеству или по их типам.

Наследование – это процесс, посредством которого один объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию иерархии классов (hierarchical classification). Применение иерархии классов делает управляемыми большие потоки информации. Без использования иерархии классов, для каждого объекта пришлось бы задать все характеристики, которые бы исчерпывающи его определяли. Однако при использовании наследования можно описать объект

путем определения того общего класса (или классов), к которому он относится, но со специальными чертами, делающие объект уникальным.

Синтаксис наследования следующий: при описании класса-потомка его класс-предок указывается через двоеточие.

Пример определения класса-предка Dad и класса-потомка Son:

```
public class Dad {}
public class Son: Dad {}
```

При инициализации полей объектов класса-наследника необходимо также инициализировать и поля базового класса. Инициализация полей, как было сказано выше, обычно осуществляется с использованием конструктора. Передача управления конструктору базового класса при создании объекта – представителя производного класса осуществляется посредством конструкции ...(...):base(...){...}, которая располагается в объявлении конструктора класса-наследника между заголовком конструктора и телом. После ключевого слова base в скобках располагается список значений параметров конструктора базового класса. Очевидно, что выбор соответствующего конструктора определяется типом значений в списке (возможно, пустом) параметров.

Пример:

```
public class Dad
{
int a;
public Dad(int s);
}
public class Son: Dad
{
public Son(int k):base(k) {}
}
```

Если же у базового класса не объявлено ни одного конструктора (оставлен конструктор по умолчанию) или объявлен конструктор без параметров, тогда конструкцию base можно не использовать: при ее отсутствии управление передается конструктору без параметров.

Однако при вызове конструктора можно передавать управление не только конструктору базового класса, но и другому конструктору данного класса. Это удобно в тех случаях, когда необходимо создать множество объектов, различающихся между собой каким-либо образом, но и имеющим некую общую часть. Тогда для реализации общей части можно написать какой-то общий конструктор, а уже в других конструкторах, выполняющих более детальную настройку объекта, вызывать общий. Передача управления собственному конструктору аналогична описанной выше, только вместо ключевого слова base используется ключевое слово this.

Пример:

```
public class Dad
{
int a;
```

```
public Dad(int s);
}
public class Son: Dad
{
public Son(int k):base(k) {}
public Son():this(10) {}
}
```

В рамках данной лабораторной работы согласно варианту предложено познакомиться с одним из управляющих операторов языка C#. Управляющие операторы применяются в рамках методов. Они определяют последовательность выполнения операторов в программе и являются основным средством реализации алгоритмов.

В данной лабораторной работе необходимо познакомиться со следующими категориями управляющих операторов:

1. Условные операторы. Вводятся ключевыми словами if, if ... else ..., switch.
2. Циклы. Вводятся ключевыми словами while, do ... while, for, foreach.

Условный оператор if имеет следующие правила использования. После ключевого слова if располагается взятое в круглые скобки условное выражение (булево выражение), следом за которым располагается оператор (блок операторов) произвольной сложности. Далее в операторе if ... else ... после ключевого слова else размещается еще один оператор.

Невозможно построить оператор if ... else ... на основе одиночного оператора объявления:

```
if (true) int X = 12;
if (true) int X = 12; else int Z = 1;
```

Такие конструкции ошибочные, так как одиночный оператор в C# – это не блок, и ставить в зависимость от условия (пусть даже всегда истинного) создание объекта нельзя.

Но в блоках операций определена своя область видимости, и создаваемые в них объекты, никому не мешая, существуют по своим собственным правилам.

```
if (true) {int X = 12;}
if (true) {int X = 12;} else {int Z = 0;}
```

Это правило действует во всех случаях, где какой-то оператор выполняется в зависимости от условия.

Оператор switch имеет вид:

```
switch(<проверяемая_переменная>)
{
case <констаната_1_значение_переменной>: <оператор>; break;
….
case <констаната_n_значение_переменной>: <оператор>; break;
default: < оператор>break;
}
```

Пример:

```
int val;
```

```
switch (val)
{
case 0: Console.WriteLine(0);break;
case 1: Console.WriteLine(1);break;
default: Console.WriteLine(«Число неизвестно»);break;
}
```

Так как case-блоки строятся на основе одиночного оператора, то объявлять переменные в них нельзя. Ключевое слово break управляет выходом из switch. В следующем примере, если значение переменной val будет равно 0, то на экран выведется два сообщения: 0 и 1.

```
int val;
switch (val)
{
case 0: Console.WriteLine(0); // нет break;
case 1: Console.WriteLine(1);break;
default: Console.WriteLine(«Число неизвестно»);break;
}
```

Цикл while – это цикл с предусловием. Имеет следующий синтаксис:

```
while (УсловиеПродолжения) Оператор
```

Пример

```
int i=0;
while(i<=10) i++;
```

Работает по следующему правилу: сначала проверяется условие продолжения оператора и в случае, если значение условного выражения равно true, соответствующий оператор (блок операторов) выполняется.

Цикл do ... while – цикл с постусловием. Синтаксис:

```
do Оператор while (УсловиеПродолжения)
```

Пример:

```
int i=0;
do
i++;
while(i<=10) ;
```

Разница с ранее рассмотренным оператором цикла состоит в том, что здесь сначала выполняется оператор (блок операторов), а затем проверяется условие продолжения оператора.

Цикл for – пошаговый цикл. Имеет синтаксис:

```
for([Выражение_Инициализации];[Условие_Продолжения];[Выражение_Шага])
```

Выражение_Инициализации, Условие_Продолжения, Выражение_Шага в могут быть пустыми, но наличие пары символов ';' в заголовке цикла обязательно.

Пример:

```
for(int i=0;i<10;i++) Console.WriteLine(i);
```

Переменные, объявленные в операторе инициализации данного цикла, не могут быть использованы непосредственно после оператора до конца блока, содержащего этот оператор.

Цикл forech является универсальным перечислителем для элементов какой-либо коллекции. С ним предлагается познакомиться в следующей лабораторной работе.

Задание к лабораторной работе

В рамках консольного приложения разработать класс B-наследник класса A (из лабораторной работы №1) с полем d и свойством c2. Свойство c2 – результат вычисления выражения над полями a, b, d. В теле свойства использовать управляющий оператор (см. вариант в таблице 2). У класса A создать конструктор, инициализирующий его поля. Для класса B определить 2 конструктора: один – наследуется от конструктора класса A, второй – собственный. В теле программы создать объекты классов A и B, продемонстрировав работу всех конструкторов. Вывести значения свойства на экран.

Таблица 2. Варианты заданий к лабораторной работе №2

Вариант	Управляющий оператор	Вариант	Управляющий оператор
1	if	15	do while
2	switch	16	if
3	for	17	switch
4	while	18	for
5	do while	19	while
6	if	20	do while
7	switch	21	if
8	for	22	switch
9	while	23	for
10	do while	24	while
11	if	25	do while
12	switch	26	if
13	for	27	switch
14	while		

Контрольные вопросы

1. Что такое конструктор?
2. Зачем нужен конструктор?
3. Что такое полиморфизм?
4. Что такое наследование?

5. Как при вызове конструктора класса-наследника передать управление конструктору базового класса?
6. Как при вызове одного конструктора класса передать управление другому конструктору класса?
7. Какие циклы есть в C#?
8. В чем особенность каждого из циклов, используемых в C#?
9. Что запрещено делать в теле условного оператора?
10. Какое назначение ключевого слова break в операторе switch?

ЛАБОРАТОРНАЯ РАБОТА №3. МАССИВЫ

Цель лабораторной работы: Научиться работать с массивами и циклом foreach.

Теоретические основы

Массив – упорядоченное множество однотипных элементов. Одной из характеристик массива является ранг или размерность массива. Массив размерности (или ранга) N (N определяет число измерений массива) – это массив массивов (или составляющих массива) ранга N–1. Составляющие массива – это массивы меньшей размерности, являющиеся элементами данного массива. При этом составляющая массива сама может быть либо массивом, либо элементом массива.

В C# определены две различных категории массивов:

1. простые (прямоугольные) массивы,
2. jagged (вложенные) массивы.

Примеры ссылок на массив:

```
int[,] arr0; // прямоугольный двумерный массив
int[][] arr5;// одномерный массив одномерных массивов
```

В C# массив является ссылочным типом, то есть в программе переменная-массив – ссылка на область памяти, где этот массив фактически хранится. Поэтому при работе с массивами обязательно использовать оператор new для выделения памяти при создании нового массива.

Примеры создания массива:

```
int[,] arr0 = new int[2,2];
```

Цикл foreach – универсальный перечислитель для коллекций. Синтаксис:

foreach(<переменная_элемент_коллекции> in <коллекция>)

Цикл имеет следующую семантику «Для каждого элемента из коллекции делать». Так как массив можно определить как коллекцию, то этот цикл может использоваться для перебора элементов массива.

Пример:

```
int[] ar=new int[9];
foreach(int i in ar) Console.WriteLine(i);
```

Задание к лабораторной работе

В класс B добавить поле-массив. Разработать конструктор для инициализации массива, который при своем вызове передает управление собственному конструктору класса B. Размер массива – поле a, инициализация элементов массива: свойство c2 (см. лабораторную работу №2), умноженное на индекс элемента массива. В программе вывести на экран элементы массива. Для вывода использовать цикл foreach.

Контрольные вопросы

1. Что такое массив?
2. Как массив представляется в C#?
3. Какие виды массивов определяются в C#?
4. Какие назначение и логика работы цикла foreach?
5. Какое значение индекса первого элемента в массиве?

ЛАБОРАТОРНАЯ РАБОТА №4. ИНДЕКСАТОРЫ. СТАТИЧЕСКИЕ ПОЛЯ. ПАРАМЕТРИЗОВАННЫЕ КЛАССЫ

Цель лабораторной работы: Научиться работать с индексаторами, статическими полями и параметризованными классами.

Теоретические основы

Индексаторы являются синтаксическим удобством, позволяющим создавать класс, структуру или интерфейс, доступ к которому клиентские приложения получают, как к массиву. Чаще всего индексаторы реализуются для доступа к закрытой внутренней коллекции или закрытому массиву. Вместе с модификаторами доступа индексаторы реализуют механизм инкапсуляции для полей-массивов и являются аналогами свойств, определяемых для обычных полей.

Пример объявления индексатора

```
public class AClass1
   {
      private int[] imyArray = new int[20];
      public int  this[int ind1] //индексатор
      {
         get
         { return imyArray[ind1]; }
         set
         { imyArray[ind1] = value; }
      }
}
```

Статические поля – поля, принадлежащие классу. Они объявляются с ключевым словом static. Основное отличие от обычных полей – для обращения к статическим полям не требуется создание объекта. Доступ осуществляется напрямую через имя класса. Более того, через объекты к статическим полям обратиться нельзя.

Пример объявления статического поля:

public static int I;

Параметризованные классы – классы, позволяющие определить тип своих аргументов при непосредственном создании объектов.

Пример параметризованного класса:

```
public class AClass1<T>
{
   private T[] imyArray = new T[20];
 }
public class M
{
   static void Main(string[] args)
   {
      AClass1<string> K = new AClass1<string>();
      AClass1<int> K2 = new AClass1<int>();
   }
}
```

Основное ограничение, налагаемое на параметризованные классы при их создании: необходимо следить, чтобы операции, используемые для типа-параметра, были определены для всех типов или же использовать механизмы преобразования типов.

Задание к лабораторной работе

В классе B определить индексатор для исходного массива. Вывести в программе на экран элементы массива через индексатор. Добавить в B еще один массив и определить индексатор и для него. Вывести на экран значения элементов второго массива через индексатор. Второй массив инициализировать при описании (то есть НЕ в конструкторе) . Создать параметризованный класс C со статическим полем. В программе продемонстрировать умение работы со статическим полем и параметризацией класса. В качестве параметров взять строковый тип и числовой тип (то есть создать 2 объекта с разными параметрами). Статическое поле – тип строка.

Контрольные вопросы

1. Что такое индексатор?
2. Сколько индексаторов может быть у класса?
3. В чем отличие статических полей от обычных?
4. Что такое параметризованные классы?

5. Что необходимо учитывать при проектировании параметризованных классов?

ЛАБОРАТОРНАЯ РАБОТА №5. ПЕРЕОПРЕДЕЛЕНИЕ ОПЕРАЦИЙ

Цель лабораторной работы: научиться переопределять операции для классов. В частности разобраться с определением критериев истинности для объектов классов и перегрузкой логических операций.

Теоретические основы

Перегрузка операций в C# позволяет определять смысл стандартных операций C# (+, - и т. д.) для классов, определяемых пользователем. Например, что значит, сложить два объекта класса A. Перегрузка операций строится на основе открытых статических функций-членов, объявленных с использованием ключевого слова operator.

Не все операции могут быть перегружены. Существуют определенные правила и ограничения на перегрузку операций:

+, –, !, ~, ++, —, true, false	Унарные символы операций, допускающие перегрузку. true и false также являются операциями
+, –, *, /, %, &, |, ^, <<, >>	Бинарные символы операций, допускающие перегрузку
==, !=, <, >, <=, >=	Операции сравнения перегружаются
&&, ||	Условные логические операции моделируются с использованием ранее переопределенных операций & и |
[]	Операции доступа к элементам массивов моделируются за счет индексаторов
+=, –=, *=, /=, %=, &=, =, ^=, <<=, >>=	Операции не перегружаются по причине невозможности перегрузки операции присвоения
=, ., ?:, –>, new, is, sizeof, typeof	Операции, не подлежащие перегрузке

Правила:

1. Префиксные операции ++ и — перегружаются парами;
2. Операции сравнения перегружаются парами: если перегружается операция ==, также должна перегружаться операция !=, < и >, <= и >=.
3. Операции true и false также перегружаются парами. В этом случае для объекта класса определяются критерии истинности. Необходимо следить, чтобы критерии истинности, определенные в операции true и в операции false, не противоречили друг другу.

Синтаксис:

public static <тип возвращаемого значения> operator <операция>(<параметры>)

Пример

```
class Program
  {
    public static Program operator ++(Program par1)
    {
      par1.x++;
      return par1;
    }
  }
```

Задание к лабораторной работе

Для класса B переопределить операции согласно варианту (см. вариант в таблице 3). В основной программе продемонстрировать использование переопределенных операций.

Таблица 3. Варианты заданий для лабораторной работы №5

Вариант	Операции	Вариант	Операции	Вариант	Операции
1	true, false, &	10	true, false, &	19	true, false, &
2	true, false, |	11	true, false, |	20	true, false, |
3	true, false, !	12	true, false, !	21	true, false, !
4	true, false, &	13	true, false, &	22	true, false, &
5	true, false, |	14	true, false, |	23	true, false, |
6	true, false, !	15	true, false, !	24	true, false, !
7	true, false, &	16	true, false, &	25	true, false, &
8	true, false, |	17	true, false, |	26	true, false, |
9	true, false, !	18	true, false, !	27	true, false, !

Контрольные вопросы

1. Какой базовый принцип ООП лежит в основе переопределения операций?
2. В чем особенность переопределения логических операторов?
3. Какие принципы следует учитывать при переопределении операций?
4. Какие операции не подлежат переопределению?
5. Какие операции моделируются за счет других?