МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

НАБЕРЕЖНОЧЕЛНИНСКИЙ ИНСТИТУТ

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Учебно-методическое пособие
по дисциплине
«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»

Набережные Челны
2013

Лабораторный практикум: Учебно-методическое пособие по дисциплине «Объектно-ориентированное программирование» / Зубков Е.В. – Набережные Челны: Изд.-полигр.центр НЧИ К(П)ФУ, 2013. – 36 с.

Рассматриваются проблемы создания программного обеспечения и современные подходы их решения. Представлена технология объектно-ориентированного программирования, позволяющая создавать промышленные программы с большим числом строк кода. Приведены контрольные вопросы.

Для студентов направлений подготовки «Информатика и вычислительная техника», «Программная инженерия» и специальности «Автоматизированные системы обработки информации и управления».

Рецензент: к.т.н., Илюхин Алексей Николаевич

Печатается по решению совета факультета автоматизации и прогрессивных технологий Набережночелнинского института (филиала) ФГАУ ВПО «Казанский (Приволжский) федеральный университет».

Введение

Объектно-ориентированное программирование (ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

ООП возникло в результате развития идеологии процедурного программирования, где данные и подпрограммы (процедуры, функции) их обработки формально не связаны. Для дальнейшего развития объектно-ориентированного программирования часто большое значение имеют понятия события (так называемое событийно-ориентированное программирование) и компонента (компонентное программирование, КОП).

Первым языком программирования, в котором были предложены принципы объектной ориентированности, была Симула. В момент его появления в 1967 году в нём были предложены революционные идеи: объекты, классы, виртуальные методы и др., однако это всё не было воспринято современниками как нечто грандиозное. Тем не менее, большинство концепций были развиты Аланом Кэем и Дэном Ингаллсом в языке Smalltalk. Именно он стал первым широко распространённым объектно-ориентированным языком программирования.

В настоящее время количество прикладных языков программирования (список языков), реализующих объектно-ориентированную парадигму, является наибольшим по отношению к другим парадигмам. В области системного программирования до сих пор применяется парадигма процедурного программирования, и общепринятым языком программирования является Си. При взаимодействии системного и прикладного уровней операционных систем заметное влияние стали оказывать языки объектно-ориентированного программирования, такие как C#, Java и др.

Лабораторная работа № 1

Создание классов

1. Цель работы. Получение навыков в разработке программ с использованием классов.

2. Общие сведения

Классом называется фрагмент кода, которому предшествует зарезервированное слово *class*. Членами класса могут быть данные (они называются полями) и подпрограммы (они называются методами). По своей сути класс — это тип, то есть образец для создания объектов (так называются переменные типа класса). Основная форма синтаксиса объявления класса показана здесь:

```
[модификаторы] class  <имя_класса>
                {
                        <закрытые функции и переменные класса>
                        <открытые функции и переменные класса>
                };
```

Модификаторы предназначены для уточнения объявления класса. С их помощью можно определить область видимости класса, возможность его наследования, готовность класса в целом и отдельных его членов к работе. Модификаторы задаются зарезервированными словами, перечисленными в таблице 1.

Таблица 1.

Модификаторы класса и его членов

Модификатор	**Пояснение**
1	2
Public	Класс или его член доступен из любой точки программы

1	2
internal	Класс (член) доступен в сборке, в которой он определен
pritected	Класс (член) доступен потомкам и только им
private	Члены класса доступны только методам этого же класса
abstract	Абстрактный класс; должен обязательно перекрываться в потомках
sealed	Класс не может иметь наследников
static	Определяет статических член класса. Для доступа к статическому члену можно не создавать объект класса — статические члены принадлежат не отдельному объекту, но классу как таковому. Если таким членом является поле или свойство, оно имеет одинаковое значение для всех экземпляров этого класса

В качестве примера следующий класс определяет тип, который мы назовем queue (очередь):

```
class  queue
 {
   // закрытые элементы класса
       private static int  q=100; //Инициализация переменной
 q и присваивание ей значения
       private static char[] arrchar= new char[4] {'a','b','c','d'};
 //Массив символов arrchar

   // открытые элементы класса
       public static int p = q; //Инициализация переменной p
 и присваивание ей значения q
       public static char ch = arrchar[0]; //Присваивание ch
 значения arrchar[0]
```

}

Данный класс содержит закрытые переменные *q, arrchar*, т.е. функции или переменные, не являющиеся членами класса, не имеют к ним доступа, и открытые переменные *p* и *char*. Именно так достигается инкапсуляция: доступ к определенным частям данных может быть строго контролируемым.

Для того, чтобы сделать части класса доступными из других частей программы, они должны быть объявлены с использованием ключевого слова *public*, которое и служит спецификатором доступа. Все переменные или функции, определенные после ключевого слова *public*, являются открытыми, т.е. доступными как для других членов класса, так и для любой другой части программы, в которой находится этот класс. Хотя можно иметь переменные с открытым доступом, лучше ограничить их использование или вовсе исключить их из употребления. Вместо этого следует сделать все данные закрытыми и контролировать доступ к ним с помощью функций, имеющих спецификатор доступа *public*. Таким образом открытые функции обеспечивают интерфейс к закрытым данным класса. Это позволяет реализовать инкапсуляцию.

Представленная ниже программа демонстрирует все части класса queue:

```
using System;
        namespace Project1
        {
          class  queue
        {
           // закрытые элементы класса
                private static int  q=50;
                private  static  char[]  arrchar=  new  char[4]
        {'a','b','c','d'};
           // открытые элементы класса
                public static int p = q;
                public static char ch = arrchar[0];
        }
        class  Executer
        {
```

```
        [STAThread]
        static void Main()
            {
            Console.WriteLine(queue.p); // Вывод на
экран значения переменной p
            Console.WriteLine(queue.ch); // Вывод на
экран значения переменной ch
            Console.ReadLine(); // Ожидание нажатия
клавиши Enter
            }
        }
}
```

Следует запомнить, что закрытые части объекта доступны только для функций или переменных, которые являются членами класса. Так, например, если написать команду *Console.WriteLine(queue.q)*, то компилятор её не пропустит, выдав сообщение, что переменная *q* имеет защитный уровень.

3. Постановка задачи

Создать класс <имя класса> (имя класса задается в соответствии с выбранным вариантом), содержащий поля, которые можно использовать для хранения данных. Предусмотреть инициализацию переменных (полей) класса, помещение данных в переменные и извлечение данных.

4. Порядок выполнения работы

В главном меню системы программирования выберите команду File□New□Project или щелкните на соответствующей инструментальной кнопке. В окне New Project выберите пункт Console Application и щелкните на кнопке OK. В открывшемся окне в соответствии с вариантом задания напишите программу, использующую созданный вами класс.

5. Содержание отчета

5.1 Тема и цель работы.
5.2 Текст программы.
5.3 Результаты выполнения программы.

Приложение 1

Варианты заданий

№ варианта	Имя класса	Поле 1	Поле 2	Поле 3
1	2	3	4	5
1	Студент	Ф.И.О.	Специальность	№ группы
2	Военнослужащий	Ф.И.О.	Род войск	Звание
3	Процессор	Фирма производитель	Количество ядер	Тактовая частота
4	Жесткий диск	Фирма производитель	Емкость	Интерфейс
5	Корпус системного блока ПК	Фирма производитель	Цвет	Выходная мощность блока питания
6	Монитор	Фирма производитель	Цвет	Размер по диагонали
7	Принтер	Фирма производитель	Тип принтера	Количество печатаемых страниц в минуту
8	Патч-корд	Фирма производитель	Тип кабеля	Длина
9	Печатное издание	Вид издания	Тираж	Формат
10	Легковой автомобиль	Марка	Цвет	Максимальная скорость

1	2	3	4	5
11	Самолет	Марка	Крейсерская скорость	Дальность полета
12	Флеш-карта	Фирма производитель	Стандарт	Емкость
13	Преподаватель	Ф.И.О.	Кафедра	Стаж работы
14	Лампа освещения	Тип лампы	Потребляемая мощность	Выходная световая мощность
15	Сканер	Тип сканера	Фирма производитель	Макс. разрешение
16	Оптический диск	Тип диска	Фирма производитель	Емкость
17	Пишущая ручка	Тип ручки	Фирма производитель	Стоимость
18	Грузовой автомобиль	Марка	Цвет	Макс. грузоподъемность
19	Модем	Тип модема	Фирма производитель	Макс.скорость передачи данных
20	Компьютерная сеть	Топология	Тип физической передающей среды	Макс.скорость передачи данных

1	2	3	4	5
21	Источник бесперебойного питания	Фирма производитель	Марка	Емкость
22	Дискета	Фирма производитель	Размер	Емкость
23	Телефон	Тип телефона	Фирма производитель	Стоимость
24	Материнская плата	Фирма производитель	Сокет	Чипсет
25	Ноутбук	Фирма производитель	Процессор	Стоимость
26	Блок питания ПК	Фирма производитель	Форм-фактор	Мощность
27	Мышь для ПК	Фирма производитель	Тип мыши	Интерфейс

Лабораторная работа № 2

Создание объектов

1. Цель работы. Получение навыков в разработке программ с использованием объектов.

2. Общие сведения

Класс предоставляет компилятору все необходимые сведения для создания создания объекта данного класса. В то же время в реальной программе могут действовать только объекты. Как только определен класс, можно создать объект этого типа, используя имя класса. Фактически имя класса становится спецификатором нового типа данных. Например, следующий код создает объект с именем *intqueue* типа *queue*: *queue intqueue.*

Для создания объекта каждый класс автоматически снабжается одноименным методом без параметров, который называется *конструктором по умолчанию*. Задача конструктора заключается в выделении объекту динамической памяти, необходимой для размещения всех его полей. При обращении к конструктору используется зарезервированное слово *new*. Возможные варианты создания объектов показаны здесь:

```
using System;
        {  class  queue
        {private int  q=50;
          private char[] arrchar= new char[4] {'a','b','c','d'};
          public int p = q;
          public char ch = arrchar[0];
        }
        class  Executer
         {        [STAThread]
                  static void Main()
                  {  //Новый объект можно создать в одной строке:
                  queue a = new queue(); // создание объекта a
        класса queue
                    // ... или двух
                  queue b;
```

```
            b = new queue(); // создание объекта b класса queue
            }
        }
        }
```

Если рассматривается та часть программы, которая не входит в состав класса, то для вызова переменной класса необходимо использовать имя объекта и оператор «точка». Например, следующий фрагмент иллюстрирует вызов переменной *p* для объекта *a* и переменной *ch* для объекта *b*:

```
using System;
{ class queue
...
...
    queue a = new queue(); // создание объекта a класса queue
    queue b;
    b = new queue(); // создание объекта b класса queue
    Console.WriteLine(a.p); // Вызов значения переменной p
    Console.WriteLine(b.ch); // Вызов значения переменной ch
    Console.ReadLine(); // Ожидание нажатия клавиши Enter
            }
    }
}
```

Очень важно ясно себе представлять, что *a* и *b* являются двумя различными объектами. Это означает, что инициализация *a* не означает инициализацию *b*. Единственной связью между объектами *a* и *b* служит то, что они являются объектами одного и того же типа. Более того, копии переменных *q*, *char*, *p* и *ch* объекта *a* совершенно независимы от соответствующих копий переменных объекта *b*.

Использование имени класса и оператора «точка» необходимо только при вызове переменных или функций извне класса. Внутри класса одна переменная или функция может вызывать другую переменную или функцию непосредственно без использования оператора «точка».

Умалчиваемый конструктор, как и любой другой метод класса, можно перегрузить. Перегрузка метода заключается в том, что в классе объявляется одноименный метод. Чтобы

компилятор «понял», о каком методе идет речь, набор входных параметров перегруженного метода (с учетом *полиморфизма*) должен отличаться от параметров оригинального — это обязательное условие. При необходимости можно сколько угодно раз перегружать метод, лишь бы у всех перегруженных методов наборы входных параметров отличались друг от друга и от параметров оригинального метода.

Полиморфизм — это принцип автоматического выбора метода, вызываемого объектом, в соответствии с типом данного объекта и с учетом иерархии наследования.

Программа реализующая перегрузку умалчиваемого конструктора представлена в следующем листинге:

```
using System;
namespace Project2
{ class  queue
{     private int  q=50;
      private char[] arrchar= new char[4] {'a','b','c','d'};
      public int p;
      public char ch;
      public queue() {} // Конструктор по умолчанию
      public queue(int q, int i) // Перегруженный конструктор
      { p = q;
        ch = arrchar[i];
      }
}
class  Executer
{
      [STAThread]
      static void Main()
                  {
// Создание объекта a с помощью перегруженного конструктора
                        queue a = new queue(500,1);
                        Console.WriteLine(a.p);
                        Console.WriteLine(a.ch);
// Создание объекта b с помощью конструктора по умолчанию
```

```
                    queue b = new queue();
                    Console.WriteLine(b.p);
                    Console.WriteLine(b.ch);
                    Console.ReadLine(); // Ожидание нажатия
    клавиши Enter
                }
        }
}
```

3. Постановка задачи

Создать объекты класса <имя класса> (класс и его поля задаются в соответствии с выбранным вариантом в лабораторной работе 1). Объекты должны быть созданы с помощью конструктора по умолчанию и перегруженного конструктора.

4. Порядок выполнения работы

В главном меню системы программирования выберите команду File□New□Project или щелкните на соответствующей инструментальной кнопке. В окне New Project выберите пункт Console Application и щелкните на кнопке OK. В открывшемся окне в соответствии с вариантом задания напишите программу, использующую созданные вами объекты.

5. Содержание отчета

5.1 Тема и цель работы.

5.2 Текст программы.

5.3 Результаты выполнения программы.

Лабораторная работа № 3

Наследование, скрытие полей класса и виртуальные методы

1. Цель работы. Получение навыков в разработке программ с использованием классов образованных от других классов.

2. Общие сведения

Важнейшая характеристика класса — возможность создания на его основе новых классов с наследованием всех его свойств и методов и добавлением собственных. Наследование позволяет создавать новые классы, повторно используя уже готовый исходный код и не тратя времени на его переписывание.

Наследование — это отношение, связывающее классы, один из которых является базовым и называется родительским, а другой создается на его основе и называется наследником. Наследование заключается в том, что класс-наследник приобретает все свойства и методы родительского класс и добавляет к ним собственные.

Например, описан класс queue1, наследующий описание класса queue:

```
using System;
        {  class  queue1 : queue
                  { }
        }
```

В ходе наследования новые классы наследуют поля и методы всех родительских классов, вышестоящих по иерархии наследования. Одни из этих полей и методов могут быть замены на собственные, другие могут представлять собой точные копии родительских. В класс-наследник автоматически включаются все поля наследуемого класса. Поля и другие элементы в классе-наследнике могут переопределяться (скрываться) другими элементами.

Скрытие (переопределение) элементов класса — это явное описание в классе наследнике с новыми характеристиками уже существующих элементов из наследуемого класса.

Следующий пример иллюстрирует переопределение в классе *queue1*, наследуемому от класса *queue*, полей *q* и *arrchar* с невидимых в видимые и присвоению им новых значений, а также введение в класс *queue1* новой переменной *r*.

```
using System;
namespace Project3
{
  class  queue
{          private int  q=50;
           private char[] arrchar= new char[4] {'a','b','c','d'};
           public int p;
           public char ch;
}
 class  queue1 : queue
{           private int r = 50;
            public new int q=50;
            public new char[] arrchar= new char[4] {'e','f','g','h'};
           public queue1() {}
           public queue1(int q, int i)
           { p = q*r;
             ch = arrchar[i]; }
}
class  Executer
{
   static void Main()
                    {
```

//В переменную *a* наследуемого класса записывают экземпляр класса-наследника:

```
           queue a = new queue1(500,1);
// Создание объекта b класса  queue1
           queue1 b = new queue1(50,2);
           Console.WriteLine(a.p);
           Console.WriteLine(a.ch);
           Console.WriteLine(b.p);
           Console.WriteLine(b.ch);
           Console.ReadLine();
                    }
           } }
```

В языке C#, в соответствии с принципами полиморфизма, поддерживается принцип совместимости объектов по цепочке наследуемых классов. Так экземпляр любого из нижестоящих по иерархии наследования классов может быть присвоен переменной, описанной как один из вышестоящих классов. C# также поддерживает и обратную возможность.

Очень интересной возможностью классов, позволяющей родительскому классу обращаться к методам своих наследников, является виртуализация методов. Для этого замещаемый метод родителя объявляется с квалификатором *virtual*. Встретив это слово, компилятор создаст таблицу виртуальных методов, в которую поместит их названия и адреса точек входа. Класс-наследник объявляет замещающий метод с квалификатором *override*. На этапе выполнения программы при обращении любого родительского класса к замещенному в таблицу виртуальных методов помещается ссылка на соответствующий метод наследника, который и работает так, как если бы он изначально был частью родительского класса.

Следующий пример иллюстрирует данную возможность.

```
using System;
namespace Project3a
{ class Roditel
  { public virtual string GetStr() {return "Родитель";}
        public void Test() {Console.WriteLine(GetStr());}
  }
  class Naslednik : Roditel
  { public override string GetStr() {return "Наследник";}
  }
        class Executer
        {       static void Main()
                { Roditel A = new Roditel();
                  Naslednik B = new Naslednik();
                  A.Test(); // Вывод: Родитель
                  B.Test(); // Вывод: Наследник
                  Console.ReadLine(); }
        }
}
```

3. Постановка задачи

Создать класс-наследник от класса <имя класса> (класс и его поля задаются в соответствии с выбранным вариантом в лабораторной работе 1). В классе-наследнике перекрыть некоторые поля класса-родителя и добавить дополнительное поле, связанное со свойствами предмета задания по варианту. При перекрывании полей использовать виртуальные методы. Создать объекты обоих полученных класса и занести в них данные. Организовать обращения родительских методов к замещающим их методам наследника.

4. Порядок выполнения работы

В главном меню системы программирования выберите команду File□New□Project или щелкните на соответствующей инструментальной кнопке. В окне New Project выберите пункт Console Application и щелкните на кнопке OK. В открывшемся окне в соответствии с вариантом задания напишите программу, использующую принципы наследования классов.

5. Содержание отчета

5.1 Тема и цель работы.

5.2 Текст программы.

5.3 Результаты выполнения программы.

Лабораторная работа № 4

Использование классов для работы с массивами данных

1. Цель работы. Получение навыков в разработке программ с использованием классов для создания, хранения и работы с массивами данных.

2. Общие сведения

Массивы представляют собой упорядоченные структуры, содержащие множество данных одного и того же типа. Упорядоченность массива позволяет обращаться к отдельному его элементу с помощью индекса — целочисленного выражения, определяющего положение элемента в массиве. Как и в других С-подобных языках, в C# самый первый элемент массива имеет индекс 0.

Массивы относятся к ссылочным типам, поэтому должны инициализироваться при помощи оператора *new*. Для описания массива нужно указать квадратные скобки за именем типа данных. При этом тип данных определяет тип хранящихся в массиве данных:

```
int[] arrInt; //Целочисленный массив
```

Объявление массива еще не создает объект, который можно использовать в программе. Для инициализации массива следует указать количество его элементов:

```
arrInt = new int[25]; //Массив содержит 25 целых чисел
```

При инициализации массива ему выделяется нужная память, а элементы массива получают значение 0, которое трактуется в зависимости от типа элементов. После инициализации массив готов к работе.

Разумеется, можно объявлять массив и одновременно инициализировать его:

```
int[] arrInt = new int[25];
```

Более того, можно одновременно с созданием массива наполнить его нужными значениями:

int[] *arrInt* = *new int*[3] {1, 2, 3};

Если какой-то член класса оформлен в виде массива, с помощью средств языка можно создать для этого члена *индексатор*, позволяющий обращаться к отдельным элементам члена, как к элементам массива — с помощью квадратных скобок.

Создание индексатора не составляет большого труда: для этого в классе следует создать специальное свойство с помощью конструкции *this*[].

В примере программы создаются 2 класса: класс-контейнер *queues*, предназначенный для хранения множества элементов, и класс *queue*, содержащий параметры одного элемента. Свойство *queues.queue* является индексатором.

```
using System;
namespace Project4
{
        class queue
        {  string str;
           int integer;
           public queue(string str, int integer)
           { this.str=str;
                this.integer=integer;
           }
        class queues
        {
           queue[] Queues = new queue[10];
           public queue this [int pos]
           {  get
                {if (pos >=0 || pos<10) return Queues[pos];
                else throw new
IndexOutOfRangeException("Вне диапазона!");}
                set {Queues[pos] = value;}
           }   }
```

```
        static void Main()
            {       queues A = new queues();
                    A[0] = new queue("Номер в
очереди: ",1);
                    A[1] = new queue("Номер в
очереди: ",2);
                    for (int i = 0; i < 2; i++)
                        {Console.Write(A[i].str);

    Console.WriteLine(A[i].integer);}
                    Console.ReadLine();
                }
            }
    }
```

3. Постановка задачи

Создать объекты класса <имя класса> (класс и его поля задаются в соответствии с выбранным вариантом в лабораторной работе 1), причем объекты класса должны хранить массивы данных о предметной области, связанные со свойствами предмета задания по варианту. Причем количество элементов массива должно задаваться программно пользователем в интерактивном режиме.

4. Порядок выполнения работы

В главном меню системы программирования выберите команду File□New□Project или щелкните на соответствующей инструментальной кнопке. В окне New Project выберите пункт Console Application и щелкните на кнопке OK. В открывшемся окне в соответствии с вариантом задания напишите программу, использующую созданные вами объекты.

5. Содержание отчета

5.1 Тема и цель работы.

5.2 Текст программы.

5.3 Результаты выполнения программы.

Лабораторная работа № 5

Сохранение текущего состояния объектов в файлах

1. Цель работы. Получение навыков в разработке программ с использованием файлов, сохранения информации в них и работы с ними.

2. **Общие сведения**

При разработке приложений часто возникают две в общем случае схожие задачи: сохранить (прочитать) содержимое данных (файла) и сохранить (прочитать) текущее состояние объекта в файле или таблице базы данных. Несмотря на несомненную схожесть указанных задач, для решения каждой из них предусмотрены свои классы.

Техника работы с файлами зависит от типа файла (текстовый или двоичный). При работе с текстовыми файлами сначала создается поток класса *FileStream*. Данные записываются в файл с помощью вспомогательного объекта класса StreamWriter, а читаются объектом класса *StreamReader*. При создании объекта *FileStream* ему передаются имя файла и два параметра: *FileMode*, определяющий способ создания потока, и *FileAccess*, регулирующий доступ потока к данным.

Таблица1.

Значение параметра *FileMode*

Значение	**Описание**
1	2
Append	Добавляет записи в существующий файл или создает новый. Требует, чтобы параметр *FileMode* имел значение *Write*
Create	Создает новый файл или переписывает существующий. Требует, чтобы параметр *FileMode* имел значение *Write*

1	2
CreateNew	Создает новый файл, а если он уже существует, возникает исключение. Требует, чтобы параметр *FileMode* имел значение *Write*
Open	Открывает существующий файл. Если файла нет, возникает исключение
OpenOrCreate	Открывает существующий или создает новый файл, если он еще не создан
Truncate	Открывает существующий файл и делает его размер равным нулю

Таблица2.

Значение параметра *FileAccess*

Значение	**Описание**
Read	Поток может читать данные
ReadWrite	Поток может читать и записывать данные
Write	Поток может записывать данные

Для потока существует понятие текущей записи — в эту запись помещаются данные и из нее они считываются. Положением текущей записи можно управлять с помощью метода *Seek*(), имеющего такую сигнатуру:

```
public virtual long Seek (int Offset, SeekOrigin Origin);
```

Здесь *Offset* — смещение относительно позиции, указанной параметром *Origin*.

Таблица3.

Значение параметра *SeekOrigin*

Значение	**Описание**
Begin	Соответствует началу потока
Current	Соответствует текущей записи потока
End	Соответствует концу потока

Физическая запись данных в файл реализуется в момент закрытия потока методом *Close*() или выталкиванием записей из промежуточного буфера методом *Flush*(). Во втором случае поток не закрывается и готов к продолжению операций.

Обработка двоичных файлов во многом подобна обработке текстовых: сначала создается поток, затем — объекты *BinaryWriter* или *BinaryReader* в зависимости от направления передачи данных (в файл или из файла).

C# поддерживает интересную технологию сохранения текущего состояния объекта на некотором носителе информации. Эта технология называется сериализацией объекта. Сериализованный объект может быть, например, передан по сети на другой компьютер и там восстановлен в первоначальном состоянии — этот процесс называется десериализацией.

Особенностью сериализации является то, что ее можно применять к классам, поддерживающим эту процедуру. Для этого класс должен объявляться с атрибутом [Serializable] или в его объявлении должно быть явное указание на то, что он исполняет интерфейс ISerializable. Лишь относительно небольшое количество (менее 40) стандартных классов реализуют данный интерфейс. Это, в основном, классы, используемые как хранилища данных.

Следующий пример иллюстрирует данную технологию.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace Project5
{
[Serializable]
  class  queue
{
        private int  q=50;
        private char[] arrchar= new char[4] {'a','b','c','d'};
        public int p;
        public char ch;
}
```

```
            [Serializable]
 class  queue1 : queue
{
 private int r = 50;
 public new int q=50;
 public new char[] arrchar= new char[4] {'e','f','g','h'};
            public queue1() {}
            public queue1(int q, int i)
            { p = q*r;
              ch = arrchar[i];
            }
}

class  Executer
{
            [STAThread]
  static void Main()
                        {
            queue1 a = new queue1();
            queue1 b = new queue1(50,2);
            queue1 c = new queue1(500,1);

 FileStream fs;
 fs = new FileStream

("File_queue.bin",FileMode.Create,FileAccess.Write);
 BinaryFormatter sw = new BinaryFormatter();
 sw.Serialize(fs,b);
 sw.Serialize(fs,c);
 fs.Close();

 fs = new FileStream
             ("File_queue.bin",FileMode.Open,FileAccess.Read);
 long i=0,
 j = fs.Length/231; //231 — количество байт занимаемых
одним объектом класса  queue1

 while (i < j)
```

```
{
  a = (queue1) sw.Deserialize(fs);
  Console.WriteLine(a.p);
  Console.WriteLine(a.ch);
  i++;
 }
          fs.Close();
          Console.ReadLine();
                  }
          }
}
```

3. Постановка задачи

Создать объекты класса <имя класса> (класс и его поля задаются в соответствии с выбранным вариантом в лабораторной работе 1), причем объекты класса должны хранить данных о предметной области, связанные со свойствами предмета задания по варианту. Сохранить данные объекты в файле. Причем сохранить необходимо как отдельные объекты, так объект контейнерного типа, полученный в лабораторной работе 4. Считать сохраненные данные из файла и вывести их на экран.

4. Порядок выполнения работы

В главном меню системы программирования выберите команду File□New□Project или щелкните на соответствующей инструментальной кнопке. В окне New Project выберите пункт Console Application и щелкните на кнопке OK. В открывшемся окне в соответствии с вариантом задания напишите программу сохранения в файл и чтения из файла данных, использующую созданные вами объекты.

5. Содержание отчета

5.1 Тема и цель работы.

5.2 Текст программы.

5.3 Результаты выполнения программы.

Лабораторная работа № 6

Сортировка элементов массива

1. Цель работы. Ознакомиться с различными методами сортировки массивов.

2. Общие сведения

Сортировка – это процесс перегруппировки заданного множества объектов в некотором определённом порядке. Целью сортировки является облегчение поиска элементов в массиве.

Существует множество методов сортировки, каждый из которых имеет свои достоинства и недостатки. Выбор алгоритма зависит от структуры обрабатываемых данных.

Методы сортировки можно разбить на два класса – сортировку массивов и сортировку файлов. Иногда их называют внутренней и внешней сортировкой, так как массивы хранятся в быстрой, оперативной, внутренней памяти машины со случайным доступом, а файлы размещаются в более медленной, но и более ёмкой внешней памяти.

Основным условием выбора метода сортировки должно являться экономное использование доступной памяти. Это предполагает, что перестановки, приводящие элементы в порядок, должны выполняться «на том же месте», т.е. методы, в которых элементы из массива «A» передаются в массив «B», представляют существенно меньший интерес. Поэтому будем классифицировать методы по их экономичности, т.е. по времени их работы. Хорошей мерой эффективности может быть C – число необходимых сравнений и M – число перестановок элементов. Эти числа есть функции от n – числа сортируемых элементов.

Сортировка массива методом прямого выбора

Алгоритм сортировки массива по возрастанию методом прямого выбора может быть представлен так:

1. Просматривая массив от первого элемента, найти минимальный элемент и поместить его на место первого элемента, а первый – на место минимального.

2. Просматривая массив от второго элемента, найти минимальный элемент и поместить его на место второго элемента, а второй – на место минимального.

3. И так далее до предпоследнего элемента.

Таблица 1 – Пример сортировки с помощью прямого выбора

Начальное состояние массива	3	5	8	1	2
Шаг 1					
Шаг 2	1				
Шаг 3	1	2			
Шаг 4	1	2	3		
Отсортированный массив	1	2	3	5	8

Сортировка массива методом прямого обмена (пузырьковым методом)

В основе алгоритма метода прямого обмена лежит обмен соседних элементов массива. Каждый элемент массива, начиная с первого, сравнивается со следующим, и если он больше следующего, то элементы меняются местами. Таким образом, элементы с меньшим значением продвигаются к началу массива (всплывают), а элементы с большим значением – к концу массива (тонут), поэтому этот метод иногда называют "пузырьковым". Этот процесс повторяется на единицу меньше раз, чем элементов в массиве.

Сортировка массива методом прямого включения

Сортировка методом прямого включения, так же, как и сортировка методом простого выбора, обычно применяется для массивов, не содержащих повторяющихся элементов.

Сортировка этим методом производится последовательно шаг за шагом. На k–м шаге считается, что часть массива, содержащая первые k–1 элемент, уже упорядочена, то есть $a(1) \le a(2) \le \ldots \le a(k-1)$.

Далее необходимо взять k–й элемент и подобрать для него место в отсортированной части массива такое, чтобы после его вставки упорядоченность не нарушилась, то есть надо найти такое j $(1 \le j \le k-1)$ что $a(j) \le a(k) \le a(j+1)$. Затем надо вставить элемент a(k) на найденное место.

С каждым шагом отсортированная часть массива увеличивается. Для выполнения полной сортировки потребуется выполнить n–1 шаг.

Осталось ответить на вопрос, как осуществить поиск подходящего места для элемента x. Поступим следующим образом: будем просматривать элементы, расположенные левее x (то есть те, которые уже упорядочены), двигаясь к началу массива. Нужно просматривать элементы a(j), j изменяется от k–l до 1. Такой просмотр закончится при выполнении одного из следующих условий:

• найден элемент $a(j) < x$, что говорит о необходимости вставки x между $a(j-1)$ и a(j).

• достигнут левый конец упорядоченной части массива, следовательно, нужно вставить x на первое место.

До тех пор, пока одно из этих условий не выполнится, будем смещать просматриваемые элементы на 1 позицию вправо, в результате чего в отсортированной части будет освобождено место под x.

Методику сортировки иллюстрирует таблица 2.

Первоначально упорядоченная последовательность состоит из 1–го элемента 9. Элемент a(2)=5 – первый из неупорядоченной последовательности и 5 < 9, поэтому ставится на его место, а 9 сдвигается вправо. Теперь упорядоченная последовательность состоит из двух элементов 5, 9. Элемент a(3)=15 неупорядоченной последовательности больше всех элементов упорядоченной последовательности, поэтому остаётся на своём месте и на следующем шаге упорядоченная последовательность состоит из 5, 9, 15, а рассматриваемый

элемент 6. Процесс происходит до тех пор, пока последовательность не станет упорядоченной.

Таблица 2 – Пример сортировки с помощью прямого включения

Начальное состояние массива	9	5	15	6	7	8	13	11
1-й шаг	9	5	15	6	7	8	13	11
2-й шаг	5	9	15	6	7	8	13	11
3-й шаг	5	6	9	15	7	8	13	11
4-й шаг	5	6	7	9	15	8	13	11
5-й шаг	5	6	7	8	9	15	13	11
6-й шаг	5	6	7	8	9	13	15	11
Отсортированный массив	5	6	7	8	9	11	13	15

Шейкерная сортировка

Метод пузырька допускает три простых усовершенствования. Во-первых, если на некотором шаге не было произведено ни одного обмена, то выполнение алгоритма можно прекращать. Во-вторых, можно запоминать наименьшее значение индекса массива, для которого на текущем шаге выполнялись перестановки. Очевидно, что верхняя часть массива до элемента с этим индексом уже отсортирована, и на следующем шаге можно прекращать сравнения значений соседних элементов при достижении такого значения индекса. В-третьих, метод пузырька работает неравноправно для "легких" и "тяжелых" значений. Легкое значение попадает на нужное место за один шаг, а тяжелое на каждом шаге опускается по направлению к нужному месту на одну позицию.

На этих наблюдениях основан метод шейкерной сортировки (ShakerSort). При его применении на каждом следующем шаге

меняется направление последовательного просмотра. В результате на одном шаге "всплывает" очередной наиболее легкий элемент, а на другом "тонет" очередной самый тяжелый. Пример шейкерной сортировки приведен в таблице 3.

Таблица 3 – Пример шейкерной сортировки

Начальное состояние массива	8	23	5	65	44	33	1	6
Шаг 1	8	23	5	65	44	33	1	6
	8	23	5	65	44	1	33	6
	8	23	5	65	1	44	33	6
	8	23	5	1	65	44	33	6
	8	23	1	5	65	44	33	6
	8	1	23	5	65	44	33	6
	1	8	23	5	65	44	33	6
Шаг 2	1	8	23	5	65	44	33	6
	1	8	5	23	65	44	33	6
	1	8	5	23	44	65	33	6
	1	8	5	23	44	33	65	6
	1	8	5	23	44	33	6	65
Шаг 3	1	8	5	23	44	33	6	65
	1	8	5	23	44	6	33	65
	1	8	5	23	6	44	33	65
	1	8	5	6	23	44	33	65
	1	5	8	6	23	44	33	65
Шаг 4	1	5	8	6	23	44	33	65
	1	5	6	8	23	44	33	65
	1	5	6	8	23	33	44	65
Отсортированный массив	1	5	6	8	23	33	44	65

Шейкерную сортировку рекомендуется использовать в тех случаях, когда известно, что массив "почти упорядочен".

Сортировка массива с помощью включений с уменьшающимися расстояниями (метод Шелла)

Д. Шеллом было предложено усовершенствование сортировки с помощью прямого включения.

Идея метода: Сначала отдельно группируются и сортируются элементы, отстоящие друг от друга на расстоянии 4. Такой процесс называется четверной сортировкой. В нашем примере восемь элементов и каждая группа состоит точно из двух элементов. После первого прохода элементы перегруппировываются – теперь каждый элемент группы отстоит от другого на две позиции – и вновь сортируются. Это называется двойной сортировкой. И, наконец, на третьем проходе идет обычная, или одинарная, сортировка.

На первый взгляд можно засомневаться: если необходимо несколько процессов сортировки, причем в каждый включаются все элементы, то не добавят ли они больше работы, чем сэкономят? Однако на каждом этапе либо сортируется относительно мало элементов, либо элементы уже довольно хорошо упорядочены, и требуется сравнительно немного перестановок.

Ясно, что такой метод в результате дает упорядоченный массив, и, конечно же, сразу видно, что каждый проход от предыдущих только выигрывает (так как каждая i-сортировка объединяет две группы, уже отсортированные 2i-сортировкой). Также очевидно, что расстояния в группах можно уменьшать по-разному, лишь бы последнее было единичным, ведь в самом плохом случае последний проход и сделает всю работу. Однако совсем не очевидно, что такой прием "уменьшающихся расстояний" может дать лучшие результаты, если расстояния не будут степенями двойки.

Пример сортировки методом Шелла приведен в таблице 4.

Анализ этого алгоритма поставил несколько весьма трудных математических проблем, многие из вторых так еще и не решены. В частности, неизвестно, какие расстояния дают наилучшие результаты. В работе «Искусство программирования» Кнут показывает, что имеет смысл использовать такую последовательность (она записана в обратном порядке): 1, 4, 13, 40, 121,… или 1, 3, 7, 15, 31,....

Математический анализ показывает, что в последнем случае для сортировки п элементов методом Шелла затраты пропорциональны $n^{1,2}$. Хотя это число значительно лучше п2, тем не менее мы не ориентируемся в дальнейшем на этот метод, поскольку существуют алгоритмы еще лучше.

Таблица 4 – Пример сортировки методом Шелла

Начальное состояние массива	17	2	34	20	11	13	31	29
Шаг 1. ($L=4$)	17	2	34	20	11	13	31	29
Шаг 2. ($L=2$)	11	2	31	20	17	13	34	29
Шаг 3. ($L=1$)	11	2	17	13	31	20	34	29
Отсортированный массив	2	11	13	17	20	29	31	34

3. Постановка задачи

Создать объекты класса <имя класса> (класс и его поля задаются в соответствии с выбранным вариантом в лабораторной работе 1), причем объекты класса должны хранить данных о предметной области, связанные со свойствами предмета задания по варианту. В класс контейнерного типа, полученный в лабораторной работе 4, добавить все рассмотренные сортировки в качестве методов. Осуществите сортировку своих данных по разным полям сравните эффективность методов сортировки.

4. Порядок выполнения работы

В главном меню системы программирования выберите команду File□New□Project или щелкните на соответствующей инструментальной кнопке. В окне New Project выберите пункт Console Application и щелкните на кнопке OK. В открывшемся окне в соответствии с вариантом задания напишите программу для сортировки массива всеми рассмотренными методами.

5. Содержание отчета

5.1 Тема и цель работы.

5.2 Текст программы.

5.3 Результаты выполнения программы.

6. Контрольные вопросы

1. Что называют сортировкой?
2. Какие цели преследует сортировка?
3. Какие два класса сортировки вы знаете?
4. От чего зависит выбор метода сортировки?
5. Какие методы сортировки вы знаете?

В каких случаях применяют тот или иной метод сортиров

Рекомендуемые источники

1. Хорев, П.Б. Технологии объектно-ориентированного программирования : учеб. пособие для студ. вузов по напр. 654600 "Информатика и вычислит. техн" / П. Б. Хорев. - 2-е изд., стер. - М. : Академия, 2008. - 448 с. : ил. - (Высшее проф. образование). - Библиогр.: с. 444-445. - ISBN 978-5-7695-5262-5.
2. Фленов, М.Е. Библия C#. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2011. — 541 с.: ил. + CD-ROM. - ISBN 978-5-9775-0655-7 Режим доступа: http://znanium.com/bookread.php?book=355199

Подписано в печать 02.07.13 г.
Формат 60x84x16 Бумага офсетная Печать ризографическая
Уч.-изд.л. 2,3 Усл.-печ.л. 2,3 Тираж 100 экз.
Заказ 223/1
Издательско-полиграфический центр
Набережночелнинского института
Казанского (Приволжского) федерального университета
_____

423810, г. Набережные Челны, Новый город, проспект Мира, 68/19
Тел./факс (8552) 39-65-99