Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«Владимирский государственный университет имени Александра Григорьевича и Николая Григорьевича Столетовых»
(ВлГУ)

Кафедра информационных систем и программной инженерии

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Методические указания к лабораторным работам

Составители:
Вершинин В.В.

Владимир
2020

ОГЛАВЛЕНИЕ

Лабораторная работа №1

Знакомство с платформой Microsoft .NET Framework

Цель работы

Изучить основные особенности платформы Microsoft .NET Framework, познакомиться с её базовыми компонентами.

Общие сведения

Microsoft .Net Framework является программной платформой, которая позволяет разрабатывать и запускать приложения, под этой платформой. Сделано это для того, чтобы разработчик мог максимально абстрагироваться от программно-аппаратного обеспечения пользователя, которые будет запускать его приложения. Другими словами пользователя (равно как и разработчика) не должно волновать, какая операционная система установлена, какой разрядности процессор, какая у него архитектура и т.д. Для запуска программы достаточно чтобы под данную систему существовала и была установлена реализация .Net Framework. Для операционных систем Windows разработкой платформы занимается компания Microsoft. Кроме этого существуют и независимые реализации, прежде всего это Mono и Portable.NET, позволяющие запускать программы .Net на других операционных системах, например на Linux.

Архитектура .Net Framework

Платформа состоит из двух частей. Основой является исполняющая среда Common Language Runtime (CLR), которая может выполнять как обычные программы, так и серверные приложения. Вторая, не менее важная часть, это библиотека классов Framework Class Library (FCL), содержащая в себе множество компонентов для работы с базами данных, сетью, вводом/выводом, файлами, пользовательским интерфейсом и т.д. Это позволяет разработчику не заниматься низкоуровневым программированием, а использовать уже готовые классы.

Библиотека классов представляет собой огромный набор всевозможных классов, которые в процессе их использования в конечном итоге предоставляют необходимую функциональность для разрабатываемого приложения. Соответственно, все классы строго структурированы и разбиты по группам. Каждый из них инкапсулирует в себе некоторую уникальную функциональность. В отдельные группы попадают те классы, функциональность которых схожа, либо относится к одной теме, либо по еще каким-либо признакам.

Важные части библиотеки классов

Windows Forms — отвечает за разработку графического интерфейса. Фактически является обёрткой над Win32 API.

ADO.NET — предоставляет доступ данным. В основном используется для работы с базами данных.

ASP.NET — технология разработки веб-сайтов, веб-приложений и веб-сервисов.

Language Integrated Query (LINQ) — реализация языка запросов, напоминающего по синтаксису SQL в программах на .Net для работы с базами данных напрямую из кода бизнес-приложений.

Windows Presentation Foundation (WPF) — система создания графических интерфейсов, использующая язык разметки XAML. В отличие от Windows Forms использует графическую технологию DirectX, что обеспечивает более быструю работу за счет аппаратного ускорения графики.

Windows Communication Foundation (WCF) — система обмена данными между приложениями .Net. Используется для создания распределённых приложений.

Центральной частью .NET является CLR (Runtime), которая представляет собой некоторую программную оболочку, управляющая выполнением кода .NET приложения. Это управление проявляется в памяти, в потоках приложения, в удаленном взаимодействии, а также в строгом контроле соответствия исполняемого кода множеству требований. Одним словом, CLR – это набор некоторых служб, которые и выполняют код приложения. Отсюда следует один важный вывод: не каждый код сможет быть выполнен под управлением CLR. Естественно, существует великое множество языков программирования и каждый из них так, или иначе связан с некоторой платформой. В виду этого говорят, что тот код, который разрабатывался для выполнения под управлением CLR называется управляемым (managed). А тот, который не был рассчитан на выполнение под .NET, называется неуправляемым (unmanaged).

Схематическое представление .NET и взаимодействие с ней показано на рис. 1

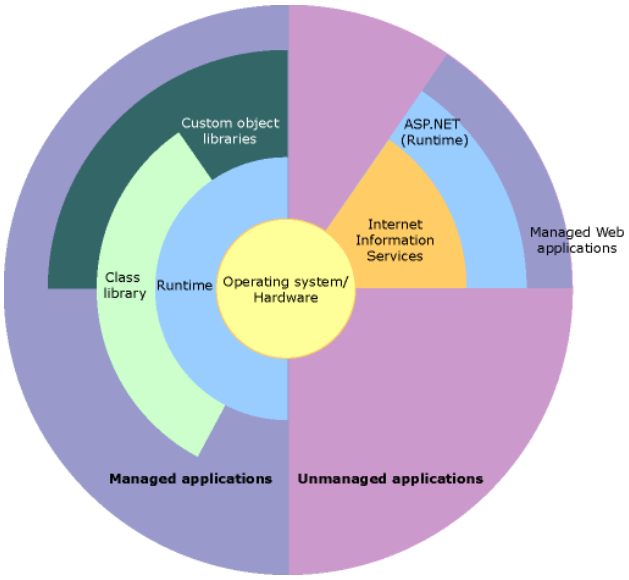

Рисунок 1. Взаимодействие .NET

В итоге, у CLR имеется взаимодействие с операционной системой. Также CLR имеет доступ к библиотеке базовых классов .NET framework и пользовательским классам. Заметьте, что пользовательские классы могут базироваться на базовых классах. Теперь, когда мы познакомились со структурой платформы .NET, можем оговорить главные причины её существования и основные преимущества в её использовании:

1. Для разработки приложения под .NET возможно пользоваться одним из многих .NET-поддерживаемых языков. Код, написанный на разных языках, будет работать одинаково. Это ещё дает то преимущество, что код, написанный на одном языке, может использоваться кодом, написанном на другом;

2. Наличие общей библиотеки базовых классов, использовать которую могут все .NET-совместимые языки;

3. Общий механизм CLR для любого .NET –совместимого языка;

4. Управление и минимизация всевозможных конфликтов версий при развертывании приложения на целевой машине. Кроме того нет необходимости в регистрации компонентов в системном реестре;

5. Процесс разработки любого типа приложений подобен и во многом идентичен, как WindowsForms-приложений, так и веб-приложений;

6. Полная поддержка и контроль соответствия структуры исполняемого кода стандартам объектно-ориентированного программирования.

Языки программирования .Net

Одной из основных идей, заложенной в .Net, является совместимость различных частей приложения, которые могут быть разработаны на разных языках. Например программа, написанная на C# может обратиться к методу из библиотеки, написанной на Visual Basic .NET, или класс на Managed C++ может быть унаследован от класса на Delphi .Net.

Языки, включённые в Visual Studio: C#, Visual Basic .NET, JScript .NET, C++/CLI, F# (Visual Studio 2010). Также существуют независимые проекты, позволяющие разрабатывать программы под .Net Framework на других языках.

Схема трансляции в .NET

Общая схема трансляции в .NET представлена на рисунке 2. Рисунок адаптирован из статьи Дж.Рихтера, опубликованной в сентябрьском выпуске 2000 года журнала MSDN Magazine.

Поскольку опасность замедления учитывалась с самого начала разработки .NET, промежуточное представление было спроектировано таким образом, чтобы облегчить трансляцию в машинные коды на ходу (just-in-time compiling) - в отличие, например, от Java bytecode, который разрабатывался с прицелом на интерпретацию.

Это дало возможность равномерно распределить замедление при запуске, так как обычно компилируется не вся библиотека, а только тот метод, который вызывается, и повторной компиляции одного и того же метода не производится. Таким образом, при первичном запуске приложения нет необходимости дожидаться полной компиляции, достаточно откомпилировать первый запускаемый метод.

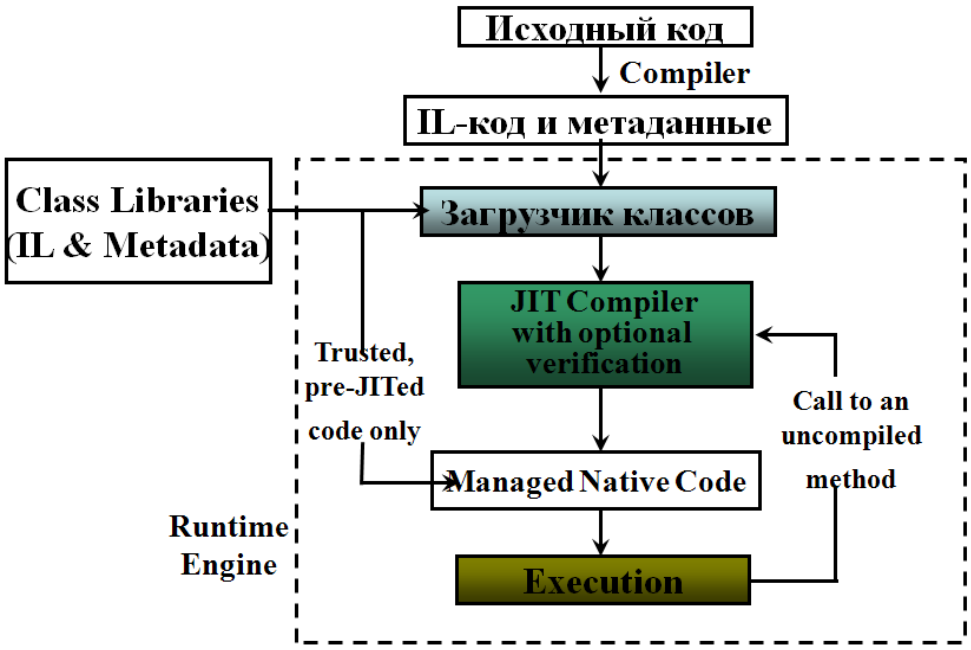

Рисунок 2. Схема трансляции в .NET

Проверка установленных версий .NET Framework

Для пользователей поздних версий Windows-операционных система платформа .NET Framework автоматический предустанавливается. В дальнейшем пользователю предстоит лишь следить за обновлениями, которые ставятся автоматически, либо вручную, скачиванием с сайта компании Microsoft. Путь установки платформы: % SystemRoot%\Microsoft.NET\Framework (или Framework64 для 64-битных систем). Одновременно может быть установлено несколько версий платформы.

В тоже время всегда существует возможность определить какие версии установлены на данном компьютере. Это можно сделать:

1. вручную, просмотрев содержимое реестра (с использованием regedit.exe) по адресу HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP

2. программно с использованием C# кода ( ). Компилируется компилятором (csc.exe) командной строки, входящим в состав .NET Framework.

В обоих случаях параметры обозначают следующее:
Install = 1 версия установлена
SP номер установленного Service Pack
Version полный номер версии

Более полный и точный перечень значений параметров можно посмотреть тут: *https://support.microsoft.com/ru-ru/kb/318785*

3. с использованием внешних утилит, которые делают это красиво. Например утилита ASoft_.NET_Version_Detector_15. Результат ее работы представлен на рис. 3.

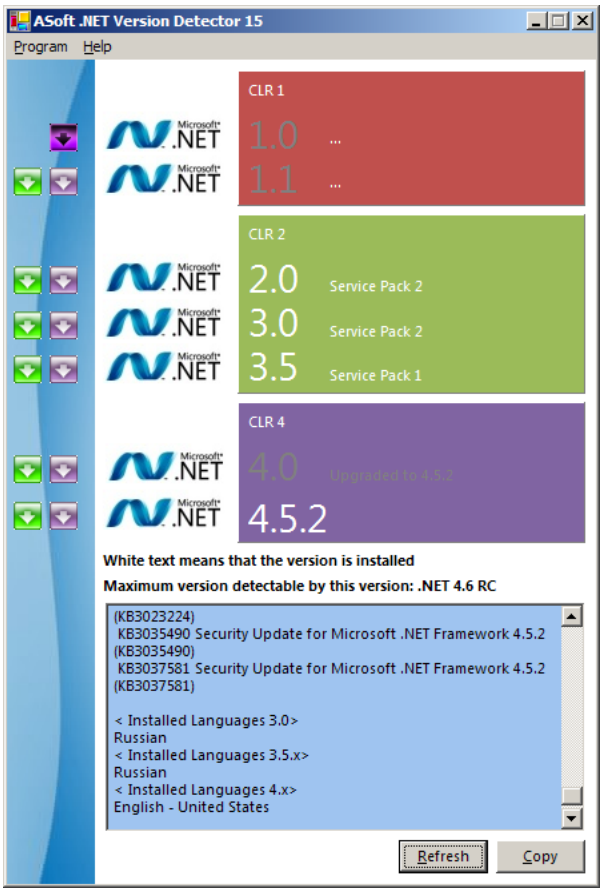

Рисунок 3. Результат определения установленных версий .NET Framework с использованием утилиты ASoft_.NET_Version_Detector_15.

Порядок выполнения работы

1. Изучить принципы организации .NET Framework и теоретические основы её функционирования.

2. На рабочей машине определить версию(-ии) .NET Framework поддерживаемые в операционной системе. Способ, которым это выполнить выбирается самостоятельно.
3. Составить отчет по результатам изучения принципов организации .NET Framework и исследования установленных версий .NET Framework.

Содержание отчета

1. Цель работы;
2. Описание программно-аппаратных характеристик машины, на которой выполнялась лабораторная работа
3. Результаты изучения принципов организации .NET Framework и определенная версия .NET Framework.
4. Выводы по работе.

Контрольные вопросы:

1. Что такое .NET Framework?
2. Какие основные компоненты .NET Framework определяют ее архитектуру?
3. В чем преимущества использования .NET Framework и управляемого кода по сравнению с использованием неуправляемого кода или вообще отказ от использования .NET Framework?
4. Что такое CLR?

Лабораторная работа №2

НАЧАЛЬНОЕ ЗНАКОМСТВО СО СРЕДОЙ РАЗРАБОТКИ MICROSOFT VISUAL STUDIO

Цель работы

Изучить среду разработки программ Microsoft Visual Studio .NET. Разобраться со средствами отладки, включенными в состав среды и реализовать простые алгоритмы на языке C#.

Общие сведения

В данном разделе приведены краткие сведения по функциям ввода-вывода языка C#, наиболее часто используемым в консольных приложениях Microsoft Visual Studio 2005.

Перед использованием функцией ввода/вывода писать Console.<имя_функции>.

Для вывода на экран предусмотрены функции:

- Write(…) – вывод строки на экран, переданной в качестве аргумента, без перевода курсора на следующую строку.
- WriteLine(…) – вывод строки на экран, переданной в качестве аргумента с переводом курсора на следующую строку.
- Для ввода с клавиатуры значений предусмотрены функции:
- Read(…) – ввод значения с клавиатуры и его присвоение переменной без перевода курсора на следующую строку.
- ReadLine(…) – ввод значения с клавиатуры и его присвоение переменной с переводом курсора на следующую строку.

Правила использования функций будут понятны после просмотра примера из приложения к данной лабораторной работе.

Порядок выполнения работы

1. Запустить среду разработчика Microsoft Visual Studio 2005 (Microsoft Visual C# Express).
2. В меню File/New Project выбрать Project. В открывшемся диалоге New Project выбрать: Project Types: Visual C# Projects; Templates: Console Application. Поля Name и Location должны содержать имя и расположение создаваемого проекта. Нажать кнопку Ок, после чего среда создает прототип нового консольного приложения.

3. В теле основного метода Main, сгенерированного по умолчанию класса Class1 написать код из Приложения приведенного в конце лабораторной работы. Опробовать работу программы. Для этого в основном окне среды разработчика в меню Build выбрать пункт меню Build <имя_проекта>. Если в процессе сборки появились ошибки, то их необходимо исправить. Средства отладки и пошагового выполнения программы доступны в меню Debug основного окна среды разработчика.
4. Взять вариант индивидуального задания у преподавателя.
5. Разработать алгоритм решения задачи реализующий вариант индивидуального задания.
6. Реализовать алгоритм на языке высокого уровня C# в среде разработчика. При этом осуществить ввод с клавиатуры всех данных, необходимых для работы программы. Вывод результатов вычисления и работы программы необходимо производить на экран.
7. Осуществить запуск программы и проверить ее выполнение на различных исходных данных.

Содержание отчета

1. Цель работы;
2. Программа на языке C#;
3. Результаты запуска и выполнения программы на контрольном примере;
4. Алгоритм работы программы;
5. Выводы по работе.

Контрольные вопросы

1. Типы данных в языке C#;
2. Массивы и структуры. Объявление и особенности работы;
3. Арифметические операции и оператор присвоения. Постфиксные и префиксные формы записи арифметических операций. Оператор присвоения;
4. Логические операторы;
5. Операторы ветвления. Полная и сокращенная форма операторов ветвления.
6. Оператор цикла с предусловием;
7. Оператор цикла с постусловием;
8. Итеративный цикл.

Варианты индивидуальных заданий

1. Отсортировать массив (использовать любой алгоритм сортировки).
2. Определить количество элементов массива, которым предшествуют элементы с меньшими значениями.
3. Каждому элементу массива, начиная со второго, присвоить значение максимального элемента из числа ему предшествующих и его самого.
4. Транспонировать матрицу.

5. Определить какие два последовательных элемента массива наименее отличаются друг от друга. Найти индекс первого элемента пары.
6. Построить массив, элементы которого суть суммы последовательных пар элементов исходного массива.
7. Определить количество элементов массива, значения которых превышают заданное. Составить новый массив из этих элементов.
8. Массив, элементы которого принадлежат множеству {0,1}, рассматривается как представление целого числа в двоичном коде. Определить значение числа, заданного таким способом. Реализовать проверку, попадает ли число в пределы диапазона, определяемого типом int.
9. Массив, элементы которого принадлежат множеству {0..F}, рассматривается как представление целого числа в шестнадцатиричном коде. Определить значение числа, заданного таким способом. Реализовать проверку, попадает ли число в пределы диапазона, определяемого типом int.
10. Представить целое число, введенное с клавиатуры в виде массива элементов, принадлежащих множеству {0, 1} – т.е. представить целое число в двоичной форме.
11. Представить целое число, введенное с клавиатуры в виде массива элементов, принадлежащих множеству {0, 7} – т.е. представить целое число в восьмеричной форме.
12. Даны два множества, представленные массивами целых чисел. Построить объединение этих множеств (например, для массивов {0, 1, 2} и {1, 3, 5} объединением будет массив {0, 1, 3, 5}).
13. Дан массив целых чисел и целое число N. Найти два элемента массива, сумма которых наиболее близка к данному числу N.
14. Дан массив целых чисел. Найти количество различных чисел среди элементов этого массива.
15. Дан массив целых чисел. Не используя других массивов, переставить элементы массива в обратном порядке.
16. Даны множества A и B, представленные массивами целых чисел. Найти разность A-B этих множеств (например, для A = {1, 2, 3} и B = {2, 1} разность A-B = {3}).

ПРИЛОЖЕНИЕ

Текст программы, реализующей следующее задание: **дан массив, элементы которого принадлежат множеству {0,1}. Определить длину первой последовательности рядом стоящих единиц.**

```
using System;

namespace lab1
{
        /// <summary>
```

```
	/// Summary description for Class1.
	/// </summary>
	class Class1
	{
		/// <summary>
		/// The main entry point for the application.
		/// </summary>
		[STAThread]
		static void Main(string[] args)
		{
			int ArraySize;		//размер будущего вектора (массива)
			int[] BinaryArray;	//массив элементов принадлежащих множеству {0, 1}

			Console.WriteLine("Введите число элементов вектора: ");
			/*Т.к. Функция ReadLine возвращает строку символов введенную с клавиатуры,
			 *то необходимо ее преобразовать к целочисленному типу (к типу int).
			 *Метод Parse выполняет это преобразование. Int32 в данном случае это
			 *целочисленный тип данных, поддерживаемый платформой .NET Framework,
			 *который является аналогом типа int, применяемого в C#*/
			ArraySize = Int32.Parse(Console.ReadLine());
			BinaryArray = new int[ArraySize];

			/* Инициализируем массив значениями 0 или 1!!!
			 * Никаких проверок не выполняем!!!*/
			for(int i=0;i<ArraySize;i++)
			{
				Console.Write("Введите " + i.ToString() + " элемент ");
				BinaryArray[i] = Int32.Parse(Console.ReadLine());
			}

			// Начинаем обрабатывать наш массив в соответствии с заданием.
			int j = 0,		// индекс для обработки массива
			OnesCounter = 0;	// счетчик единиц
			while(j<ArraySize && BinaryArray[j]!=1)
				j++;

			while(j<ArraySize && BinaryArray[j]==1)
			{
				j++;
				OnesCounter++;
			}
			Console.WriteLine("Количество единиц: " + OnesCounter.ToString());
			Console.ReadLine();
		}
	}
}
```

Лабораторная работа №3

ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Цель работы

Изучить основные концепции и особенности объектно-ориентированного программирования (ООП) в языке C#. Создать приложение, иллюстрирующее основные принципы ООП.

Общие сведения

В основе программирования под .NET лежит общая идея того, что по своей структуре любое разрабатываемое приложение является объектно-ориентированным. Это означает, что любой тип данных, сами данные и даже само приложение являются классами или строятся на базе других классов.

Несмотря на то, что с точки зрения ООП C# не привнес много нового, тем не менее, есть некоторые особенности, которые позволяют говорить о том, что C# является более гибким языком позволяющим использовать всю мощь ООП. Объекты – это, по сути, данные. Важно понимать различие между классами, структурами и объектами. В C# объект – это экземпляр структуры или класса. Ряд структур и классов в инфраструктуре .NET представляют основные типы данных в C#: тип bool – синоним класса *System.Boolean*, byte – синоним *System.Byte*, int – синоним *System.Int32* и т.д. Объявляя в программе переменные, например:

```
int iCounter;
System.Boolean bIsWeekend;
```

мы тем самым неявно создаем экземпляры класса. C# оперирует не только базовыми типами (классами) определенными в инфраструктуре .NET, но и имеет языковые средства, позволяющие достаточно гибко описывать и строить свою собственную иерархию классов. В своем базовом понимании средства создания и описания классов в C# во многом схожи с Java и C++. Дабы не повторять уже известные принципы далее рассмотрим основные особенности специфичные для платформы .NET, связанные с ООП.

Прародитель всех классов – базовый класс System.Object

Платформа .NET это строгая иерархия классов, построенная по всем канонам ООП и дающая разработчикам весь необходимый спектр возможностей для построения полноценных приложений. Разработчики .NET придерживались строгой иерархии в построении основных классов. Именно поэтому класс System.Object является базовым классом. Любое объявление (описание) нового класса неявно наследуется от System.Object. Таким образом описанный ниже класс Date по сути является потомком класса System.Object:

```
class Date
{
  public int year;
  public int month;
  public int day;
}
```

что эквивалентно описанию:

```
class Date: System.Object
{
  public int year;
  public int month;
  public int day;
}
```

Но на практике обычно используют первый вариант.

Все объекты классового типа размещаются динамически

Данная особенность заключается в том, что прежде чем использовать экземпляр класса его необходимо создать. Т.е. если в традиционной C++ программе объявляется переменная классового типа то компилятор вызывает конструктор этого класса автоматически при объявлении переменной. В дальнейшем работа с такой переменной будет происходить вполне успешно. В C# данный принцип уже не поддерживается. Так, например, следующий код вызовет ошибку компиляции

```
Date MyBirth;
MyBirth.year = 2008;      // ошибка компилятора здесь!
```

Чтобы этот код успешно работал, необходимо знать, что в .NET все объекты размещаются динамически. Поэтому перед использованием объекта его необходимо создать, путем вызова соответствующего конструктора, например, так:

```
Date MyBirth;
MyBirth = new Date();     // вызов конструктора
MyBirth.year = 2008;      // теперь все ОК
```

или так:

```
Date MyBirth = new Date();      // объявление объекта и его создание
MyBirth.year = 2008;            // теперь все ОК
```

Идея динамического создания и размещения всех объектов понятна и исходит из общей концепции построения .NET платформы, в которой есть компонент, отвечающий за автоматическое освобождение памяти, называемый уборщик мусора (garbage collector).

Доступ к полям объекта через свойства

Одним из дополнительных новшеств языка C# является введение понятия свойства класса. Свойства – специальные методы класса, позволяющие организовать чтение (запись) атрибутов класса. Например, организация непосредственного доступа к атрибутам класса Date (смотри модификатор доступа public) является потенциально не безопасным, потому что программист может установить отрицательное значение любому из атрибутов, или же значение выходящее за границы допустимого значения (для месяца больше 12, для дня больше 31). Введение свойств позволяет элегантно решить эту проблему, повысив тем самым надежность кода.

```
class Date
{
  private int year;
  private int month;
  private int day;
```

```
    public int Year
    {
      set
      {
         if(value < 1)
           throw new ArgumentOutOfRangeException("Year<1");
           // возбуждаем исключительную ситуацию (ошибку)
         year = value;
      }
      get
      {
         return year;
      }
    }

    public int Month
    {
      set
      {
         if(value < 1 || value > 12)
           throw new ArgumenOutOfRangeException(
             "Month должен быть от 1 до 12");
         month = value;
      }
      get
      {
         return month;
      }
    }
  }

  class UsingOfDate
  {
     public void Main()
     {
        Date MyBirth = new Date();
        MyBirth.month = 10;      // ошибка! доступ к закрытому полю класса
        MyBirth.Month = 10;      // все ОК! доступ к открытому свойству класса

        if (MyBirth.Month >= 3 && MyBirth.Month <= 5)
            Concole.WriteLine("Рожден(а) весной!");
     }
  }
```

Структуры

В отличие от классов создавать экземпляр структуры явным образом не требуется. Экземпляр структуры создается автоматически при объявлении переменной соответствующего типа. Однако существует ограничение, связанное с использованием структур. Оно заключается в том, что нельзя использовать экземпляр структуры, не инициализировав его.

Например, требуется описать структуру, хранящую в себе дату и описание праздника. Это можно сделать так:

```
    struct Holiday
    {
      private int day, month;    //день и месяц праздника (целочисленный тип)
      public String Title;       //название праздника (строковый)
    }
```

В программе описанную структуру мы можем использовать следующим образом:

```
    Holiday NY;              // NY - переменная типа Holiday
    NY.Title = "Новый Год"; // Ошибки нет. Переменная NY имеет структурный тип
```

Принципы объектно-ориентированного программирования

Ниже будут перечислены основные принципы объектно-ориентированного программирования и особенности их реализации в языке C#.

Наследование

Наследование – важнейший механизм ООП, позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса наследуются новым классом. Другими словами, класс-наследник реализует спецификацию уже существующего класса (базового, родительского класс). Это позволяет обращаться с объектами класса-наследника точно так же, как с объектами базового класса.

В языке C# не разрешено множественное наследование, т.е. класс-наследник может иметь только одного предка.

В некоторых языках используются абстрактные классы. *Абстрактный класс* — это класс, содержащий хотя бы один абстрактный метод[1]; класс описан в программе, имеет поля, методы, но не может использоваться для непосредственного создания объекта. От абстрактного класса можно только наследовать. Объекты создаются только на основе производных классов, наследованных от абстрактного. Например, абстрактным классом может быть базовый класс «сотрудник ВУЗа», от которого наследуются классы «аспирант», «профессор» и т.д. Т.к. производные классы имеют общие поля и функции (например, поле «год рождения»), то эти члены класса могут быть описаны в базовом классе. В программе создаются объекты на основе классов «аспирант», «профессор», но нет смысла создавать объект на основе класса «сотрудник вуза».

Пример наследования:

```
// базовый класс
class Employee
{
   public string Name;
   public DateTime BirthDate;
}

// наследники – аспирант...
class Aspirant : Employee
{
   public string CertificateNumber;
}

// ... и профессор
class Professor : Employee
{
   public string DiplomaNumber;
   public string[] OwningNIRs;
}
```

[1] Абстрактный метод – метод, имеющий *сигнатуру* (имя и список параметров и возвращаемых значений), но не имеющий реализации в данном классе.

Инкапсуляция

Инкапсуляция – это свойство языка программирования, позволяющее объединить данные и код в объект и скрыть реализацию объекта от пользователя. При этом пользователю предоставляется только спецификация (интерфейс) объекта. Пользователь может взаимодействовать с объектом только через этот интерфейс.

Одна из наиболее распространенных ошибок делать сокрытие реализации только ради сокрытия. Целями, достойными усилий, являются:

- достижение предельной локализации изменений при необходимости таких изменений,
- прогнозируемость изменений (какие изменения в коде надо сделать для заданного изменения функциональности) и прогнозируемость последствий изменений.

Часто инкапсуляция может быть достигнута простейшими организационными мерами: знание того, что «вот так-то делать нельзя» иногда является самым эффективным средством инкапсуляции.

Пример инкапсуляции:

```
class A
{
   // реализация класса A, скрытая от "пользователя"
   private int hiddenVariable;
   private int anotherHiddenVariable;

   private int someHiddenMethod()
   { ... }

   // интерфейс класса A, доступный "пользователю"
   public int InterfaceMethod()
   { ... }
}

// класс - "пользователь" класса A, отсюда нам виден только интерфейс
// класса A – метод InterfaceMethod
class Program
{
   public static void Main(string[] params)
   {
      A objectOfA = new A();
      int someValue = objectOfA.InterfaceMethod();
   }
}
```

Полиморфизм

Полиморфизм – это взаимозаменяемость объектов с одинаковым интерфейсом. Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию — например, реализация класса может быть изменена в процессе наследования. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество методов».

Полиморфизм позволяет писать более абстрактные программы и повысить коэффициент повторного использования кода. Общие свойства объектов объединяются в систему, которую могут называть по-разному — интерфейс, класс. Общность имеет внешнее и внутреннее выражение. Внешне общность проявляется как одинаковый набор методов с одинаковыми именами и *сигнатурами* (типами аргументов и результатов). Внутренняя общность есть одинаковая функциональность методов. Её можно описать интуитивно или выразить в виде строгих законов, правил, которым должны подчиняться методы. Возможность приписывать разную функциональность одному методу (функции, операции) называется *перегрузкой* метода (функций, операций).

В объектно-ориентированных языках класс является типом данных. Полиморфизм реализуется с помощью наследования классов. Класс-потомок наследует сигнатуры методов класса-родителя, но реализация этих методов может быть другой, соответствующей специфике класса-потомка. Это называется *переопределением метода*. Другие функции могут работать с объектом класса-родителя, при этом вместо него во время исполнения будет подставляться один из классов-потомков. Это называется *поздним связыванием*. Класс-потомок сам может быть родителем. Это позволяет строить сложные схемы наследования — древовидные или сетевидные.

Абстрактные методы не имеют реализации вообще. Они специально предназначены для наследования. Их реализация должна быть определена в классах-потомках. Очевидно, что не может существовать объект класса, в котором хотя бы один метод абстрактный, так как неизвестно, как он должен работать.

Пример полиморфизма.

```
// базовый класс
abstract class Employee
{
   public string Name;
   public DateTime BirthDate;

   // абстрактный метод - не имеющий реализации
   public abstract string WhoAmI();
}

// наследники - аспирант...
class Aspirant : Employee
{
   public string CertificateNumber;

   // переопределение метода у аспиранта - сигнатура метода осталась та же
   public override string WhoAmI()
   {
      return "Я аспирант!";
   }
}

// ... и профессор
class Professor : Employee
{
   public string DiplomaNumber;
   public string[] OwningNIRs;

   // переопределение метода у профессора - сигнатура метода осталась та же
   public override string WhoAmI()
   {
      return "Я профессор!";
   }

}
```

Использование определенных выше классов:

```
class Program
{
   static void Main(string[] params)
   {
      Employee[] staff = new Employee[10];

      staff[0] = new Aspirant();
      staff[1] = new Aspirant();
      staff[2] = new Professor();
      staff[3] = new Aspirant();
      staff[4] = new Professor();

      for (int i = 0, i<5, i++)
      {
         Console.WriteLine("{0}, ты кто? – {1}", i, staff[i].WhoAmI());
      }
   }
}
```

Порядок выполнения работы

1. Проанализировать предметную область, выделив ее сущности (например, предметная область – «Университет», а сущности – студент, аспирант, аудитория и т.п.), которые будут реализованы в виде классов.
2. Выделить атрибуты структур и классов, организовать к ним доступ через методы и/или свойства.
3. Разработать основные методы классов, иллюстрирующие работу с ними.
4. Проиллюстрировать использование принципов ООП. Привести иной пример полиморфизма, в отличие от представленного в примере.
5. Реализовать программу, иллюстрирующую взаимодействие классов предметной области. Реализовать ввод всех необходимых данных с клавиатуры.
6. Опробовать работу программы.

Содержание отчета

1. Цель работы;
2. Вариант индивидуального задания;
3. Результаты анализа предметной области с указанием всех особенностей последующей реализации (описание структур и классов с характеристикой их атрибутов и методов);
4. Программа на языке C#, реализующая задание к работе;
5. Результаты запуска и выполнения программы;
6. Выводы по работе.

Варианты индивидуальных заданий

В каждом варианте индивидуального задания указана некоторая предметная область. На основании задания необходимо разработать архитектуру приложения таким образом, чтобы была явно выделена структура, описывающая какой либо объект предметной области и новый класс, также связанный с предметной областью и использующий структуру.

Доступ к полям класса (структуры) ограничить модификаторами private или protected. Доступ к полям класса организовать через открытые свойства. Доступ к полям структуры организовать через открытые методы.

В головной программе создать экземпляр класса и структуры и проанализировать их работу путем вызова соответствующих методов.

1. предметная область «ГИБДД» (примерные возможные объекты: сотрудники, подразделения, документы и т.д.);
2. предметная область «Зоопарк» (примерные возможные объекты: животные и их разновидности, сотрудники, клетки и т.д.);
3. предметная область «Автосервис» (примерные возможные объекты: различные сотрудники, автомобили, клиенты, заказы и т.д.);
4. предметная область «Развлекательный центр» (примерные возможные объекты: клиенты, заказы, развлечения, концерты, билеты и т.д.);
5. предметная область «Туристическая фирма» (примерные возможные объекты: клиенты, отели, заказы, туры, страны, билеты и т.д.);
6. предметная область «Морской грузовой порт» (примерные возможные объекты: порты, судна грузовые, пассажирские, речные объекты, грузы сотрудники, заказы на перевозку и т.д.);
7. предметная область «Библиотека» (примерные возможные объекты: книги, журналы, читатели, читательские билеты, книжная полка, разделы заказ книг и журналов и т.д.);
8. предметная область «Авиакомпания» (примерные возможные объекты: рейс, летательные аппараты, пассажиры, билеты и т.д.);
9. предметная область «Поликлиника» (примерные возможные объекты: пациент, врач, отделение, заявка, диагноз, лечение и т.д.);
10. предметная область «Магазин» (примерные возможные объекты: товар, накладная, заказа товара, продавец покупатель и т.д.);
11. предметная область «Кадровое агентство» (примерные возможные объекты: вакансия, резюме, соискатель, работодатель, каталог и т.д.);
12. предметная область «Страховое агентство» (примерные возможные объекты: страхователь, страховщик, договор страхования, объекты страхования и т.д.);
13. предметная область «Университет» (примерные возможные объекты: студенты, преподаватели, дисциплины, экзамены, зачеты ,ведомости и т.д.);,
14. предметная область «Склад» (примерные возможные объекты: склад, работники склада: кладовщик, учетчик, грузчик; товар, принятие товара на хранение, отгрузка товара, накладная, табель товара и т.д.);
15. предметная область «Промышленное производство» (примерные возможные объекты: цех, работники цеха (начальник, бригадир, наладчик, оператор), станки, продукция и ее разновидности и т.д.);

Лабораторная работа №4

ОСНОВЫ СОЗДАНИЯ WINDOWS-ПРИЛОЖЕНИЙ ДЛЯ ПЛАТФОРМЫ .NET

Цель работы

Изучить основные концепции и особенности построения Windows-приложений для платформы Microsoft .NET.

Общие сведения

См. C# для профессионалов, том 1. Глава 9, стр. 335 – 358.

Порядок выполнения работы

1. Создать простейшее Windows-приложение, содержащее одну форму с единственным элементом управления – кнопкой. В обработчик события, возникающего при нажатии на кнопку, поместить следующий код: MessageBox.Show("Hello world");
2. Программу, разработанную в предыдущей лабораторной работе, изменить таким образом, чтобы ввод и вывод данных для иллюстрации ее работы осуществлялся с использованием форм и элементов управления Windows.

Содержание отчета

1. Цель работы.
2. Вариант индивидуального задания.
3. Программа на языке C#, реализующая задание к работе.
4. Результаты запуска и выполнения программы.
5. Выводы по работе.

Контрольные вопросы

1. Понятия форм и элементов управления Windows.
2. Виды элементов управления.
3. Понятие события
4. Понятие обработчика событий
5. Что необходимо сделать для "привязки" события к функции-обработчику этого события.

Варианты индивидуальных заданий

В качестве варианта индивидуального задания следует использовать вариант индивидуального задания из лабораторной работы №3.

Лабораторная работа №5

РАСШИРЕННЫЕ ВОЗМОЖНОСТИ ЯЗЫКА ПРОГРАММИРОВАНИЯ C#

Цель работы

Познакомиться с возможностями языка программирования C# при описании интерфейсов

Общие сведения

Интерфейсы

В C# для полного отделения структуры класса от его реализации используется механизм интерфейсов. Интерфейс является расширением идеи абстрактных классов и методов. Синтаксис интерфейсов подобен синтаксису абстрактных классов. Объявление интерфейсов выполняется с помощью ключевого слова interface. При этом методы в интерфейсе не поддерживают реализацию.

Членами интерфейса могут быть методы, свойства, индексаторы и события. Интерфейс может реализовываться произвольным количеством классов. Один класс, в свою очередь, может реализовывать любое число интерфейсов. Каждый класс, включающий интерфейс, должен реализовывать его методы. В интерфейсе для методов неявным образом задается тип public. В этом случае также не допускается явный спецификатор доступа.

Синтаксис:

```
[атрибуты] [модификаторы] interface
Имя_интерфейса[:список_родительских_интерфейсов] {
обьявление_свойств_и_методов}
```

Пример:

```
using System;

class Composition {

  interface ISecret {
    void Encrypt (byte []inbuf,
                  out byte[] outbuf,
                  int key);
    void Decrypt (byte []inbuf,
                  out byte[] outbuf,
                  int key);
    string MethodName {get; set;}
  }

  class Mod2 : ISecret {
    private string FMethodName = "The cryptograhy method based on XOR
operation";
    public void Encrypt (byte []inbuf,
                  out byte[] outbuf,
                  int key) {
      outbuf = new byte[2] {1, 2};
    }

    public void Decrypt (byte []inbuf,
                  out byte[] outbuf,
                  int key) {
      outbuf = new byte[2] {3, 4};
    }
    public string MethodName {
      set {FMethodName = value; }
```

```
        get {return FMethodName; }
      }
    }
  static void Main() {
  }
}
```

Можно объявлять ссылочную переменную, имеющую интерфейсный тип. Подобная переменная может ссылаться на любой объект, который реализует ее интерфейс. При вызове метода объекта с помощью интерфейсной ссылки вызывается версия метода, реализуемого данным объектом.

Возможно наследование интерфейсов. В этом случае используется синтаксис, аналогичный наследованию классов. Если класс реализует интерфейс, который наследует другой интерфейс, должна обеспечиваться реализация для всех членов, определенных в составе цепи наследования интерфейсов.

Порядок выполнения работы

1. Реализовать программу на C# в соответствии с вариантом задания.
2. При реализации программы дополнить ее работу выводом служебной информации: время вызова; имя вызываемого метода; дополнительное описание;
3. Опробовать работу программы.
4. Сформировать отчет о проделанной работе с выводами по работе.

Содержание отчета

1. Цель работы;
2. Вариант индивидуального задания;
3. Результаты анализа предметной области с указанием всех особенностей последующей реализации (что реализовано через интерфейс и почему, что сделано с использованием делегатов и почему);
4. Программа на языке C#, реализующая задание к работе;
5. Результаты запуска и выполнения программы;
6. Выводы по работе.

Варианты индивидуальных заданий

В качестве варианта индивидуального задания следует использовать вариант индивидуального задания из лабораторной работы №3.

Контрольные вопросы:

1. Дать понятие термина «интерфейс»?
2. Чем отличается интерфейс от абстрактного класса?
3. Поддерживают ли реализацию методы интерфейса?
4. Какие объекты языка C# могут быть членами интерфейсов?
5. Каким количеством классов может быть реализован интерфейс?
6. Может ли класс реализовывать множественные интерфейсы?
7. Необходима ли реализация методов интерфейса в классе, включающе
8. этот интерфейс?

9. Возможно ли наследование интерфейсов?

Лабораторная работа №6

РАСШИРЕННЫЕ ВОЗМОЖНОСТИ C#. ДЕЛЕГАТЫ

Цель работы

Познакомиться с возможностями языка программирования C# для организации вызова методов с использованием делегатов

Общие сведения

Делегаты

Делегат — это объект, имеющий ссылку на метод. Делегат позволяет выбрать вызываемый метод во время выполнения программы. Фактически значение делегата – это адрес области памяти, где находится точка входа метода.

Важным свойством делегата является то, что он позволяет указать в коде программы вызов метода, но фактически вызываемый метод определяется во время работы программы, а не во время компилирования.

Делегат объявляется с помощью ключевого слова delegate, за которым указывается тип возвращаемого значения, имя делегата и список параметров вызываемых методов.

Синтаксис:

```
delegate тип_возвращаемого_значения имя_делегата (список_параметров);
```

Характерной особенностью делегата является возможность его использования для вызова любого метода, который соответствует подписи делегата. Это дает возможность определить во время выполнения программы, какой из методов должен быть вызван. Вызываемый метод может быть методом экземпляра, ассоциированным с объектом, либо статическим методом, ассоциированным с классом. Метод можно вызвать только тогда, когда его подпись соответствует подписи делегата.

Многоадресность делегатов

Многоадресность — это способность делегата хранить несколько ссылок на различные методы, что позволяет при вызове делегата инициировать эту цепочку методов.

Для создания цепочки методов необходимо создать экземпляр делегата, и пользуясь операторами + или += добавлять методы к цепочке. Для удаления метода из цепочки используется оператор - или -=. Делегаты, хранящие несколько ссылок, должны иметь тип возвращаемого значения void.

Демонстрационный пример. Предметная область «*Студенческие рейтинги*»

```
using System;

namespace DelegateGropuAddressDemo {
  public delegate void StudentRatingProcessor(Group group);

  enum RatingStatus { Passed, Notpassed };

  class StudentGroupActions {

    //печать рейтинговой ведомости
    public static void PrintRatingVedomost(Group group) {
      //...
      }
  }
    //расчет рейтинга студентов
    public static void CalculateRatings(Group group) {
        //...

        if (summBalls >= (10 * 4 / 2)) group[i].Rating = (int)RatingStatus.Passed;
        else group[i].Rating = (int)RatingStatus.Notpassed;
```

```
      }
    }
    //печать зачетной ведомости
    public static void PrintRankingList(Group group) {
      //...
        Console.WriteLine((group[i].Rating == (int)RatingStatus.Passed) ?
"ЗАЧТЕНО" : "НЕЗАЧТЕНО");
      }
    }
  }

  public class Student {
    String fio;
    public String FIO { set { fio = value; } get { return fio; } }

    Byte[] balls;
    public Byte this[int i] { get { return balls[i]; } set { balls[i] = value; } }

    int rating;
    public int Rating { get { return rating; } set { rating = value; } }

    public Student(String FIO) {
      rating = 0;
      balls = new Byte[4];
      fio = FIO;
    }
  }

  public class Group {
    int groupSize;
    string groupTitle = "";

    public string GroupTitle { get { return groupTitle; } }
    public int GroupSize { get { return groupSize; } }

    Student[] students;
    public Student this[int i] { get { return students[i]; } }

    public Group(string Title, int Size) {
      groupTitle = Title;
      groupSize = Size;
      students = new Student[Size];
    }
    public void InitRandom() {
      try {
        var rnd = new Random();
        for (int i = 0; i < GroupSize; i++) {
          String randomString = "";
          for (int j = 0; j < 5; j++)
            randomString += (rnd.Next(255).ToString());
          students[i] = new Student(randomString);

          for (int j = 0; j < 4; j++) students[i][j] = (byte)rnd.Next(10);
        }
      }
      catch (System.ArgumentOutOfRangeException e) {
        Console.WriteLine(e.Message);
      }
    }
  }

  class Program {
    static void Main() {
      Group ist110 = new Group("IST110", 10);
```

```
            ist110.InitRandom();

            StudentRatingProcessor studentRating;

            StudentRatingProcessor Print = StudentGroupActions.PrintRatingVedomost;
            StudentRatingProcessor Calculate = StudentGroupActions.CalculateRatings;
            StudentRatingProcessor PrintList = StudentGroupActions.PrintRankingList;

            studentRating = Print;
            studentRating += Calculate;
            studentRating += PrintList;

            studentRating(ist110);

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

Порядок выполнения работы

1. Запустить демонстрационный пример, показывающий работу с делегатами. Перед конечным запуском дописать код примера (в тексте помечен как //…) таким образом, чтобы был полный вывод ведомости рейтинговой и зачетной, а так же расчет рейтинга студента;
2. Реализовать программу на C# в соответствии с вариантом задания с использованием делегатов.
3. При реализации программы дополнить ее работу выводом служебной информации: время вызова; имя вызываемого метода; дополнительное описание;
4. Опробовать работу программы.
5. Дополнить разработанную программу
6. Сформировать отчет о проделанной работе с выводами по работе.

Содержание отчета

1. Цель работы;
2. Вариант индивидуального задания;
3. Результаты анализа предметной области с указанием всех особенностей последующей реализации (какие делегаты использованы и почему, какие методы были делегированы);
4. Программа на языке C#, реализующая задание к работе;
5. Результаты запуска и выполнения программы;
6. Выводы по работе.

Варианты индивидуальных заданий

В качестве варианта индивидуального задания следует использовать вариант индивидуального задания из лабораторной работы №3.

Контрольные вопросы:

1. Дать понятие «делегата»

2. В чем основные преимущества и особенности использования делегатов?
3. Когда осуществляется выбор вызываемого метода при использовании делегатов?
4. Возможно ли использование делегата для вызова метода соответствующего подписи делегата?
5. Возможен ли вызов метода в том случае, если его сигнатура не соответствует сигнатуре делегата?
6. Как осуществляется создание цепочки методов для многоадресных делегатов?
7. Какие операторы языка C# используются для создания цепочки методов для многоадресных делегатов?
8. Каким образом осуществляется удаление цепочки методов для многоадресных делегатов?
9. Какие операторы языка C# используются для удаления цепочки методов для многоадресных делегатов?

Лабораторная работа №7

РАСШИРЕННЫЕ ВОЗМОЖНОСТИ C#
НА ПРИМЕРЕ СОБЫТИЙНЫХ МЕХАНИЗМОВ

Цель работы

Познакомиться с возможностями языка программирования C# для организации событийно-ориентированного программирования на основе делегатов и событий

Общие сведения

События

События представляет собой автоматическое уведомление о том, что произошло некоторое действие. События действуют по следующему принципу: объект, проявляющий интерес к событию, регистрирует обработчик этого события. Когда же событие происходит, вызываются все зарегистрированные обработчики этого события. Обработчики событий обычно представлены делегатами.

События являются членами класса и объявляются с помощью ключевого слова event. Механизм событий основан на использовании делегатов.

Синтаксис:

```
event имя_делегата имя_обьекта;
```

Широковещательные события

События могут активизировать несколько обработчиков, в том числе те, что определены в других объектах. Такие события называются широковещательными. Широковещательные события создаются на основе многоадресных делегатов.

Пример использования событий. За основу взят код из предыдущей лабораторной работы. Здесь приводится только добавляемый код, для избегания дублирования.

1. Добавим вспомогательный класс , который выполняет дополнительные сервисные функции для группы, например, рассылку почтовых уведомлений:

```
public class Emailer {
  public static void SendSessionNotify(Group group) {
    //отправить e-mail уведомление группе с рейтинговой / зачетной ведомостью
    Console.WriteLine("На email группы " + group.GroupTitle + " отправлена
рейтинговая ведомость...");
  }
}
```

2. Добавим класс, объявляющий событие и метод активизирующий событие:

```
public class SessionEvent {
  public event RatingHandler activate;
  public void FixSemesterResults(Group group) {
    if (activate != null) activate(group);
    }
}
```

3. Изменим головной класс с точкой входа в программу переписав его под использование событий:

```
class Program {
  static void Main() {
    Group ist110 = new Group("IST110", 10);
    ist110.InitRandom();

    SessionEvent sessionEvent = new SessionEvent();
    //пописываем классы на получение события об окончании семестра
    sessionEvent.activate += new
RatingHandler(StudentGroupActions.PrintRatingVedomost);
```

```
    sessionEvent.activate += new
RatingHandler(StudentGroupActions.CalculateRatings);
    sessionEvent.activate += new
RatingHandler(StudentGroupActions.PrintRankingList);
    sessionEvent.activate += new RatingHandler(Emailer.SendSessionNotify);

    //фиксируем итоги семестра для группы ИСТ-110
    sessionEvent.FixSemesterResults(ist110);

    //изменим порядок обработки события
    sessionEvent.activate -= new
RatingHandler(StudentGroupActions.PrintRatingVedomost);

    Console.ReadLine();
}
```

Порядок выполнения работы

1. Доработать демонстрационный пример из предыдущей лабораторной работы с учетом добавляемой событийности. Запустить доработанный демонстрационный пример, показывающий работу с событиями. При необходимости дополнить код;
2. Реализовать программу на C# в соответствии с вариантом задания с использованием делегатов и событий.
3. При реализации программы дополнить ее работу выводом служебной информации: время вызова; имя вызываемого метода; дополнительное описание;
4. Опробовать работу программы.
5. Дополнить разработанную программу
6. Сформировать отчет о проделанной работе с выводами по работе.

Содержание отчета

1. Цель работы;
2. Вариант индивидуального задания;
3. Результаты анализа предметной области с указанием всех особенностей последующей реализации (какие события были добавлены и какие обработчики были реализованы, пояснить);
4. Программа на языке C#, реализующая задание к работе;
5. Результаты запуска и выполнения программы;
6. Выводы по работе.

Варианты индивидуальных заданий

В качестве варианта индивидуального задания следует использовать вариант индивидуального задания из лабораторной работы №3.

Контрольные вопросы:

1. Что понимается под термином «событие»?
2. Являются ли события членами классов?

3. Как выполняется описание событий? Проиллюстрируйте его фрагментом программы на языке C#.
4. Каковы механизмы языка C# для поддержки событий?
5. Что понимается под термином «широковещательное событие» и на основе какого механизма строятся широковещательные события?

Лабораторная работа №8

РАСШИРЕННЫЕ ВОЗМОЖНОСТИ C#. ЛЯМБДА ВЫРАЖЕНИЯ.

Цель работы

Познакомиться с возможностями языка программирования C# с использованием лямбда выражений.

Общие сведения

Лямбда-выражение это анонимная функция, которую можно использовать для создания делегатов или произвольных типов дерева выражений. Такие конструкции позволяют определить функции, которые в дальнейшем можно передавать в качестве аргументов или возвращать в качестве значений при вызове методов. Если рассматривать лямбда выражения в более широком смысле, то их так же можно использовать в LINQ запросах при работе с базами данных.

При создании лямбда-выражения можно определить произвольные входные параметры. Типичным признаком того, что используется лямбда-выражение будет использование специальной конструкции “=>”. В общем виде описание произвольного лямбда выражения может быть представлено так:

```
(входной параметр [, входной параметр …])  =>  {оператор;};
```

входной параметр определяет значения, которые передаются внутрь функции, тело которой определяется оператором или группой операторов. Если входной параметр один, то круглые скобки можно не ставить. Если оператор, реализующий тело лямбда-выражения один, то его можно не заключать в фигурные скобки. Таким образом лямбда-выражение может быть представлено как анонимная функция, которая объявляется и используется по месту объявления.

Как правило лямбда-выражения, выражающие какую-либо функцию используются совместно с делегатами. Простейшее лямбда-вырвжение можно описать, например, так:

```
delegate System.Int32 simpleDelegate(System.Int32 i);
class SimpleLambda {
    static void Main(string[] args) {
        simpleDelegate Square = x => x * x;
        int squareRes = Square(7);
        Console.WriteLine("Результат: {0}", squareRes);
    }
}
```

Результатом выполнения этого кода представлен на рисунке далее.

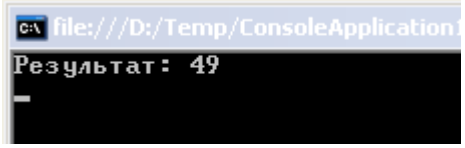

Рисунок 4. Результат выполнения кода лямбда-выражения Square.

Следует обратить внимание, что в данном примере сигнатура делегата имеет один неявный входной параметр типа int и возвращает значение типа int. Лямбда-выражение можно преобразовать в делегат соответствующего типа, так как он также имеет один параметр ввода (i) и возвращает значение, которое компилятор может неявно преобразовать в тип int. Делегат, вызываемый посредством параметра ввода 7, возвращает результат 49.

С выходом C# версии 3.0 в язык пришли лямбда-выражения и LINQ. Более того в некоторые классы библиотеки .NET Framework в качестве в качестве аргументов передаются предикаты, которые могут быть построены на базе лямбда-выражений. Благодаря этому код становится более компактным и легче воспринимается. Рассмотрим пример поиска элементов в списке по определенному критерию.

Пример. Имеется список, элементы которого содержат информацию о людях (имя и возраст). Задача сформировать новый список из исходного, в котором присутствуют люди старше 16 лет. Очевидным решением будет код, который представлен ниже:

```
namespace LambdaExpressionsDemo {
    public class Man {
        public int Age{ get; set; }
        public string Name{ get; set; }
        public Man(int age, string name) {
            Age = age;
            Name = name;
        }
    }

    class LamdaTest {
        static void Main(string[] args) {
            List<Man> mansCollection = new List<Man>() {
                new Man(15, "Петя"),
                new Man(45, "Вася"),
                new Man(13, "Оля"),
                new Man(29, "Катя"),
                new Man(5, "Ваня"),
                new Man(54, "Боб"),
                new Man(22, "Алиса")
            };
            List<Man> filteredMans = new List<Man>();
            for (int i = 0; i < mansCollection.Count; i++)
                if (mansCollection[i].Age > 16)
                    filteredMans.Add(mansCollection[i]);

            foreach (var man in filteredMans)
            Console.WriteLine(man.Name + " " + man.Age);
        }
    }

}
```

Однако, использование механизма лямбда-выражений позволит сократить объем кода, а так же придаст универсальность коду, а в случае если коллекция большая и/или способ отбора данных будет более сложным, то еще и упростит общее представление.

Поэтому код, который будет отбирать записи из коллекции можно записать так:

```
var filteredMans=mansCollection.FindAll(manscollection=>manscollection.Age>16);
```

Тип var на этапе компиляции просто превратится в List<Man> (так как метод FindAll имеет именно такой возвращаемый тип). Правая же часть метод содержит вызов метода FindAll. Если посмотреть на сигнатуру этого метода, то она будет следующей:

```
public List<T> FindAll(Predicate<T> match);
```

т.е. на вход он требует Predicate<T> match, который возвращает значение типа bool (в нашем случае - true, если Age больше 16). Подобных методов в библиотеке классов достаточно много. Например, все для того же списочного класса List это будут:

```
public int FindIndex(Predicate<T> match);
public int FindIndex(int startIndex, Predicate<T> match);
public int FindIndex(int startIndex, int count, Predicate<T> match);
public T FindLast(Predicate<T> match);
public int FindLastIndex(Predicate<T> match);
public int FindLastIndex(int startIndex, Predicate<T> match);
public int FindLastIndex(int startIndex, int count, Predicate<T> match);
```

и т.д.

Порядок выполнения работы

1. Выполнить демонстрационные примеры, показывающий работу с лямбда-выражениями.
2. Реализовать программу на C# в соответствии с вариантом задания из предыдущих работ с использованием лямбда-выражений.
3. Опробовать работу программы.
4. Сформировать отчет о проделанной работе с выводами по работе.

Содержание отчета

1. Цель работы;
2. Вариант индивидуального задания;
3. Результаты анализа предметной области с указанием всех особенностей реализации с учетом использованных лямбда-выражений;
4. Исходный код программы на C#;
5. Результаты запуска и выполнения программы;
6. Выводы по работе.

Варианты индивидуальных заданий

В качестве варианта индивидуального задания следует использовать вариант индивидуального задания из лабораторной работы №3.

Контрольные вопросы:

Дать понятие анонимной функции и лямбда-выражения

В чем основные преимущества и особенности использования лямбда-выражений?

В чем особенность объявления и использования метода с сигнатурой, объявленной как: `Predicate<T> match`

Возможно ли использование лямбда-выражений без делегатов?

Лабораторная работа №9

РАЗРАБОТКА МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЙ

Цель работы

Познакомиться с возможностями языка программирования C# при разработке многопоточных приложений.

Общие сведения

Многопотоочность — свойство платформы или приложения, состоящее в том, что процесс, который был запущен в ОС, может состоять запускать внутри себя нескольких потоков. Эти потоки будут выполняться параллельно или «параллельно». Порядок выполнения потоков ни коим образом не может быть предопределен.

Для применения многопоточности существует несколько причин:

- долгие вычисления, которые можно делать в фоне (класс **BackgroundWorker**);
- для приложений без пользовательского интерфейса (обработчики, сервсисы, службы, сервера и т.д.);
- приложения, выполняющие множественные вычисления (на многопроцессорных системах они гарантированно будут выполняться быстрее, см. свойство **Environment.ProcessorCount**);

Однако в ряде случаев многопоточность может давать и отрицательныерезультаты (но не всегда из того, то перечисляется далее):

- когда приложение простое;
- когда происходит усложнение приложения без надобности на то (причем усложнение происходит при организации их взаимодействия);
- нужно понимание, что использование многопоточности это ресурсы времени на переключение CPU. Например, работа с диском может оказаться быстрее, если все это организовать в одном (максимум двух потоках), чем дробить на множество потоков.

Класс Thread является самым элементарным из всех типов пространства имен System.Threading для определения потока. Этот класс представляет объектно-ориентированную оболочку вокруг заданного пути выполнения внутри определенного AppDomain. Этот тип также определяет набор методов (как статических, так и уровня экземпляра), которые позволяют создавать новые потоки внутри текущего AppDomain, а также приостанавливать, останавливать и уничтожать определенный поток. Список основных статических членов приведен ниже:

CurrentContext

Это свойство только для чтения возвращает контекст, в котором в данный момент выполняется поток

CurrentThread

Это свойство только для чтения возвращает ссылку на текущий выполняемый поток

GetDomain(), GetDomainID()

Этот метод возвращает ссылку на текущий AppDomain или идентификатор этого домена, в котором выполняется текущии поток

Sleep()

Общие свойства потоков и методы, которые доступны и с пользованию которых можно управлять потоками представлены в таблице 1.

Таблица 1. Свойства и методы доступные для управления потоками

Член уровня экземпляра	Назначение

IsAlive	Возвращает булевское значение, указывающее на то, запущен ли поток (и еще не прерван и не отменен)
IsBackground	Получает или устанавливает значение, указывающее, является ли данный поток "фоновым" (подробнее объясняется далее)
Name	Позволяет вам установить дружественное текстовое имя потока
Priority	Получает или устанавливает приоритет потока, который может принимать значение из перечисления ThreadPriority
ThreadState	Получает состояние данного потока, которому может быть присвоено значение из перечисления ThreadState
Abort()	Инструктирует CLR прервать поток, как только это будет возможно
Interrupt()	Прерывает (т.е. приостанавливает) текущий поток на заданный период ожидания
Join()	Блокирует вызывающий поток до тех пор, пока указанный поток (тот, в котором вызван Join()) не завершится
Resume()	Возобновляет ранее приостановленный поток
Start()	Инструктирует CLR запустить поток как можно скорее
Suspend()	Приостанавливает поток. Если поток уже приостановлен, вызов Suspend() не дает эффекта

Как правило, потоки используют разделяемые данные, поэтому часто необходимо говорить о потоковой безопасности. Например, предположим, мы хотим обрабатывать элементы массива в двух потоках. Следующий пример демонстрирует это. Под обработкой в данном простейшем случае подразумевается, что внутри потока выводится имя потока и обрабатываемый индекс. Переменная i разделяемая и определяет текущий обработанный индекс массива. По сути же то, что будет выводиться на экран, при каждом запуске будет разным, т.к. переменная i является разделяемой (используется в основном и порождаемом потоке). Этот код (исходный код приведен далее) демонстрирует потоковую небезопасность:

```
using System;
using System.Threading;


class ThreadSafe
{
    static int maxi = 30;
    static int i;

    static void Main()
    {
        new Thread(GoA).Start();
        GoB();
        Console.ReadLine();
    }

    static void GoA()
    {
        for (; i < maxi; i++)
            Console.Write("A" + i + " ");

    }

    static void GoB()
    {
        for (; i < maxi; i++)
            Console.Write("B" + i + " ");
```

```
        }
    }
```

В целом, при работе с потоками

При программном создании потоков и управления ими необходимо придерживаться следующему порядку:

- создать метод, который будет точкой входа для нового потока;
- создать новый делегат ParametrizedThreadStart (или ThreadStart), передав конструктору адрес метода, определенного на предыдущем шаге;
- создать объект Thread, передав в качестве аргумента конструктора ParametrizedThreadStart/ThreadStart;
- установить начальные характеристики потока (имя, приоритет и т.п.).
- вызвать метод Thread.Start(). Это запустит поток на методе, который указан делегатом, созданным на втором шаге, как только это будет возможно.

При этом следует понимать, что поток может находиться в определенных состояниях. Обобщенная диаграмма состояния потока приведена на рисунке 1.

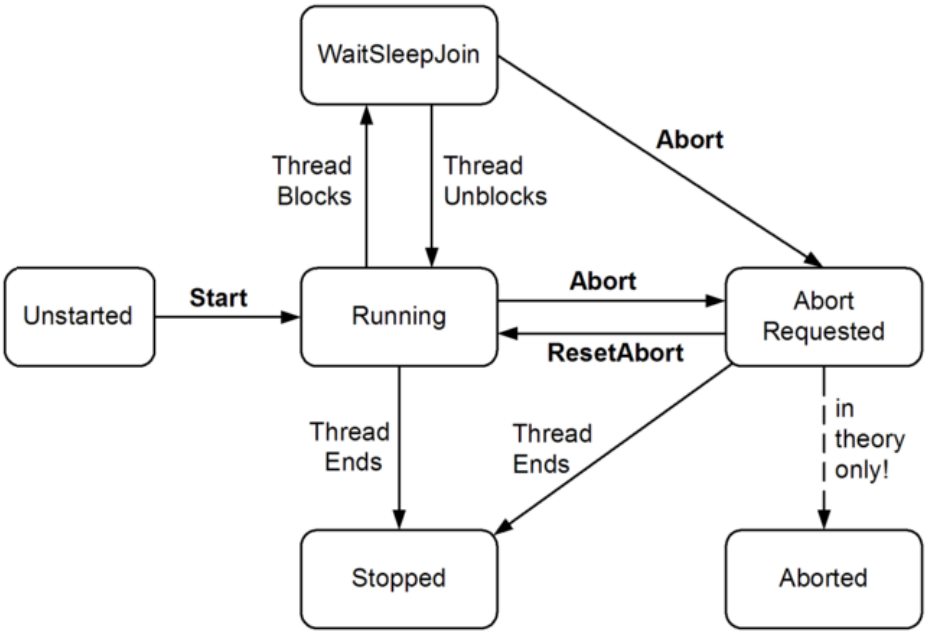

Рисунок 5. Диаграмма состояний потока.

При построении многопоточного приложения необходимо обеспечивать защиту разделяемых данных от возможных изменений их значений множеством других работающих потоков. Учитывая, что все потоки имеют параллельный доступ к разделяемым данным приложения, эффект от неуправляемого совместного использования таких данных может быть непредсказуемым. Для ранее приведенного примера как раз часто возникают случаи, что один и тот же индекс выводится для разных потоков один и тот же. ***Таким образом, необходимо выполнять синхронизацию потоков.***

В основе синхронизации лежит базовое понятие блокировки процесса, через которую обеспечивается доступ к блоку кода внутри объекта. Когда объект заблокирован одним потоком, остальные потоки не могут получить доступ к заблокированному кодовому блоку. Когда же блокировка снимается одним потоком, объект становится доступным для использования в другом потоке.

Средство блокировки встроено в язык C#. Благодаря этому все объекты могут быть синхронизированы. Синхронизация организуется с помощью ключевого слова **lock**. Общая форма блокировки выглядит так:

```
lock(lockObj) {
// синхронизируемые операторы
}
```

где *lockObj* содержит ссылку на синхронизируемый объект.

Предыдущий пример с использованием блокировок может быть переписан например так:

```
using System;
using System.Threading;


class ThreadSafe
{
    static int maxi = 30;
    static int i;
    static object locker = new object();

    static void Main()
    {
        new Thread(GoA).Start();
        GoB();
        Console.ReadLine();
    }

    static void GoA()
    {
        for (; i < maxi; )
        {
            lock (locker)
            {
                Console.Write("A" + i + " ");
                i++;
            }
        }

    }

    static void GoB()
    {
        for (; i < maxi; )
        {
            lock (locker)
            {
                Console.Write("B" + i + " ");

                i++;
            }

        }
    }

}
```

Существуют и иные способы синхронизации потоков с использованием классов Mutex и Semaphore.

Порядок выполнения работы

1. Выполнить демонстрационные примеры, показывающий работу с лямбда потоками.

2. Реализовать программу на C# в соответствии с вариантом задания из предыдущих работ с использованием многопоточности. Это будет например какая то обработка данных в фоне, когда пользователь вводит данные.
3. Опробовать работу программы.
4. Сформировать отчет о проделанной работе с выводами по работе.
5. Дополнительным заданием для продвинутого понимания сути многопоточности предлагается реализовать процедуру сортировки большого массива без использования потоков и сделать тоже самое с использованием потоков. Времена выполнения сортировки для каждого из случаев фиксировать и построить график зависимости скорости сортировки от, например, числа потоков, объема данный и т.д.

Содержание отчета

1. Цель работы;
2. Вариант индивидуального задания;
3. Результаты анализа предметной области с указанием всех особенностей реализации с учетом используемой многопоточночти;
4. Исходный код программы на C#;
5. Результаты запуска и выполнения программы;
6. Выводы по работе.

Контрольные вопросы:

1. Дать понятие потока
2. Какие делегаты используются для того, чтобы организовать ссылку на метод, который будет использоваться при запуске потока?
3. В чем причина потоковой небезопасности?
4. Продемонстрировать пример рассинхронизации потоков?
5. Методы синхронизации потоков и организации потоковой безопасности.