ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

УПРАВЛЕНИЕ ДИСТАНЦИОННОГО ОБУЧЕНИЯ И ПОВЫШЕНИЯ КВАЛИФИКАЦИИ

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

Учебно-методическое пособие
по дисциплине

«ПРОГРАММИРОВАНИЕ ПОД ПЛАТФОРМУ .NET»

Автор
Долгов В.В.

Ростов-на-Дону, 2018

Аннотация

Учебно-методическое пособие предназначено для студентов очной формы обучения направления 09.03.04 «Программная инженерия».

Авторы

доцент, к.т.н.,
зав.каф. ПОВТиАС
Долгов В.В.

Оглавление

1. Лабораторная работа №1: СОЗДАНИЕ КОНСОЛЬНЫХ ПРИЛОЖЕНИЙ.........5

1.1. Работа с исключениями 6
1.2. Задание к лабораторной работе 8
1.3. Контрольные вопросы........ 9

2. Лабораторная работа №2: ОПИСАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ КЛАССОВ 10

2.1. Задание к лабораторной работе 12
2.2. Контрольные вопросы........13

3. Лабораторная работа №3: ОПИСАНИЕ ИНТЕРФЕЙСОВ 13

3.1. Задание к лабораторной работе 14

4. Лабораторная работа №4: ДЕЛЕГАТЫ И РАБОТА С СОБЫТИЯМИ 17

4.1. Задание к лабораторной работе 18

5. Лабораторная работа №5: ИНТЕРФЕЙСЫ IEnumerable/IEnumerator........20

5.1. Задание к лабораторной работе 23
5.2. Контрольные вопросы........24

6. Лабораторная работа №6: ВВОД/ВЫВОД В СРЕДЕ .NET 25

6.1. Задание к лабораторной работе 30
6.2. Контрольные вопросы........31

7. Лабораторная работа №7: НАЗНАЧЕНИЕ РЕФЛЕКСИИ И ПОЗДНЕЕ СВЯЗЫВАНИЕ........32

7.1. Цель работы 32
7.2. Задание к лабораторной работе 35

8. Лабораторная работа №8: СЕРИАЛИЗАЦИЯ В СРЕДЕ .NET........37

8.1. Цель работы 37
8.2. Задание к лабораторной работе 40

8.3. Контрольные вопросы......................................41

9. Лабораторная работа №9: ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА С ИСПОЛЬЗОВАНИЕМ ПОТОКОВ 42

9.1. Задание к лабораторной работе 43

9.2. Контрольные вопросы......................................45

10. Лабораторная работа №10: ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ TPL (TASK PARALLEL LIBRARY) 46

10.1. Задание к лабораторной работе 47

Список литературы 49

1. ЛАБОРАТОРНАЯ РАБОТА №1: СОЗДАНИЕ КОНСОЛЬНЫХ ПРИЛОЖЕНИЙ

Для создания консольного приложения в среде Visual Studio 2015 необходимо указать специальный тип проекта – «Console Application». В этом случае при запуске проекта система будет создавать текстовое консольное окно вместо графического окна. Для ввода/вывода в таком окне используются функции, описанные в специальном классе Console пространства имен System. Основные методы и свойства этого класса приведены в табл. 1. Стоит обратить особое внимание, что в среде отсутствуют средства непосредственного ввода чисел, а весь ввод рассматривается как строковая информация. Для преобразования строковых данных в тип, необходимый программе обычно используют либо методы семейства Parse (описанные у многих типов данных), либо возможностями специального класса Convert.

Таблица 1. Основные элементы класса System.Console

Метод/Свойство	*Описание*
void Clear()	Очищает область консольного окна (заполняя его пробелами) и восстанавливает значения цветов вывода по умолчанию.
int Read()	Считывает один символ из очереди ввода и возвращает его в виде числа (кода в таблице символом текущей кодировки). Если обнаружен конец текстового потока данных, возвращает -1. При работе с консолью блокирует работу программы до появления готовых к вводу символов после нажатия клавиши Enter. Для преобразования к символьному виду возвращаемого значения необходимо воспользоваться возможностями класса Convert.
string ReadLine()	Возвращает текстовую строку, считываемую из стандартного потока ввода. При работе с консолью блокирует работу программы до нажатия клавиши Enter, после чего возвращает всю введенную строку.
void WriteLine(object o);	Выводит строку на консоль, преобразуя объект к строковому представлению.

void WriteLine(string format, Object[] args)	Выводит строку на консоль, формируя её на основе форматной строки и массива объектов.

1.1. Работа с исключениями

Исключение представляет собой ошибку или непредвиденную ситуацию, происходящую в процессе выполнения программы. На сегодняшний день исключения являются основным способом обработки ошибок, возникающих в программах, и в языке C# их обработка основывается на четырех ключевых словах: try, catch, throw и finally.

```
try
{
  блок команд, в котором может возникнуть ошибка
}
 [catch [(тип_исключения имя_исключения)]
{
  блок обработки исключения
}]
 [finally
{
  команды, которые выполняются в любом случае
}]
```

Такая конструкция выполняет команды, расположенные в блоке try, и, если внутри происходит исключение, передает управление на блок catch, которому возникшее исключение передается в качестве параметра. В любом случае (произошло исключение или нет) после выполнения блока try вызывается блок команд finally, в котором расположены команды освобождения ресурсов.

Основные стандартные типы исключений приведены в табл. 2. Кроме стандартных типов программист может создавать собственные исключения, специфичные для его программы, которые должны быть унаследованы от специального класса Exception,

являющегося базовым для всех исключений в среде .NET.

Таблица 2. Основные стандартные исключения среды .NET

Класс исключения	Описание исключительной ситуации
ArgumentException	Значение параметра, переданное в функцию неверно.
ArgumentNullException	Значение параметра, переданного в функцию, является пустой ссылкой, что недопустимо по смыслу функции или операции.
InvalidCastException	Некорректное преобразование типов данных.
StackOverflowException	Переполнение стека. Возникает, как правило, при неверной рекурсии.
OverflowException	Арифметическое переполнение.
DivideByZeroException	В ходе вычислений возникло деление на ноль
IndexOutOfRangeException	Индекс массива выходит за пределы диапазона. Используется в том же смысле для любых коллекций, допускающих индексное обращение.
ArrayTypeMismatchException	Тип сохраняемого в массив значения несовместим с типом элементов массива.
OutOfMemoryException	Недостаточно памяти. Обычно возникает в процессе создания нового объекта оператором new.

.

1.2. Задание к лабораторной работе

Создать в среде программирования Microsoft Visual Studio проект консольного приложения на языке C#. Используя класс System.Console для ввода/вывода информации, реализовать программу в соответствии с вариантом задания (табл. 3). В процессе выполнения работы запрещено использовать стандартные функции сортировки массивов, содержащиеся в библиотеки классов среды .NET.

Код программы должен содержать обработку исключительных ситуаций, которые могут возникнуть в ходе выполнения программы. В случае возникновения исключения, организовать информативный вывод данных о возникшем исключении на экран с предложением продолжить выполнение, проигнорировав ошибку, или завершить выполнение программы.

При защите работы студент должен уметь: создавать консольные проекты, устанавливать точки останова для отладки программы, выполнять программу пошагово в режиме отладки, просматривать значения переменных при отладке, знать назначение основных служебных окон среды Visual Studio.

Таблица 3. Варианты заданий

№ варианта	*Задание к лабораторной работе*
1	Ввести с консоли массив целых чисел и отсортировать его методом прямого включения.
2	Ввести с консоли массив целых чисел и отсортировать его методом прямого выбора.
3	Ввести с консоли массив целых чисел и отсортировать его методом пузырька.
4	Используя массивы, ввести с клавиатуры две прямоугольных матрицы и вывести на экран результат суммирования первой из них с транспонированной второй матрицей.
5	Ввести с клавиатуры массив строк, отсортировать полученный массив по длине строки и вывести результат на экран.
6	Ввести с консоли массив вещественных чисел, вычислить среднегеометрическое и среднеарифметическое значения и вывести их на экран.

7	Ввести с консоли массив вещественных чисел, нормализовать его относительно наибольшего элемента и вывести результаты на экран.
8	Рассматривая два массива чисел как координаты векторов комплексной плоскости, найти пару векторов, образующих при взаимном перемножении вектор наибольшей длины.

1.3. Контрольные вопросы

1. В чем отличие консольных приложений от оконных? Может ли оконное приложение осуществлять вывод на консоль?
2. Какой класс в языке C# отвечает за ввод/вывод данных на консоль?
3. Можно ли в языке C# работать с элементами массива сразу после его объявления?
4. К какому типу данных относятся массивы, к ссылочным типам или к типам-значениям? Какие особенности это накладывает при работе с массивами? Привести примеры.
5. С какого индекса начинается нумерация элементов массива в языке C#?
6. Можно ли изменить количество элементов в массиве после его создания?
7. В каком служебном окне среды Visual Studio можно просмотреть файловую структуру проекта?
8. В чем отличие блока catch от блока finally?
9. Что является параметром оператора throw?
10. Что происходит, если исключение возникло, а в функции, выполняющейся в этот момент, не описаны блоки обработки исключений?
11. Какие типы данных могут быть параметрами оператора throw?
12. Какую возможность дает указание после слова catch типа-исключения?
13. Если после оператора try стоят и блок catch, и блок finally, какой из них будет выполнен в случае исполнения кода без ошибок? А в случае исключения?
14. Могут ли блоки try/catch вкладываться друг в друга в рамках одной функции?

2. ЛАБОРАТОРНАЯ РАБОТА №2: ОПИСАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ КЛАССОВ

Платформа Microsoft .NET Framework представляет собой полностью объектно-ориентированную среду создания и выполнения программ. Все типы данных (включая базовые типы) являются классами иерархии, на вершине которой находится класс System.Object. Такой подход позволяет унифицировать работу с данными и создавать алгоритмы общего назначения.

В языке C# различают четыре основных элемента, из которых состоит большинство классов (табл. 1), каждый из которых относится к одной из областей видимости (табл. 2). В среде .NET видимость как членов классов, так и самих классов может быть ограничена модификаторами видимости.

Таблица 1. Элементы классов

Наименование	*Описание*
Поля	Используются для хранения внутренних данных класса, его состояния и других данных, используемых в алгоритмах методов и общих для класса в целом. Поля относятся к внутренней структуре класса и, основываясь на принципах инкапсуляции, не должны быть доступны для внешнего (по отношению к классу) кода.
Свойства	Элементы, определяющие состояние класса для внешнего наблюдателя (внешнего кода). С точки зрения внешнего кода, свойство класса представляется разновидностью переменной, значение которой можно считать и/или изменить. С точки зрения внутреннего устройства, свойства определяются двумя специальными методами «get» и «set», отвечающими соответственно за получение (чтение) и изменение (присвоение) значения. Значения свойств могут быть основаны как на данных, хранимых в полях класса, так и вычисляться на основе алгоритмов.

Методы	Методы класса определяют действия, которые могут производиться над классом или классом над другими классами. Алгоритмы методов зависят от внутреннего состояния класса и параметров метода.
События	Специального вида элементы класса, описывающие точки подключения к событиям, возникающим в классе. События могут сообщать внешнему коду об изменениях состояния класса, происходить по времени или любой другой причине, которую архитектор класса сочтет важной. Методы-обработчики событий могут получать параметры, через которые могут тем или иным образом влиять на алгоритмы работы класса.

Таблица 2. Модификаторы областей видимости

Идентификатор	Описание
private	Члены класса, помеченные модификатором видимости private, доступны только внутри класса, в котором они описаны. Любой внешний по отношению к классу код не имеет доступа к таким членам. Данный модификатор является модификатором видимости по умолчанию.
protected	Защищенные члены класса доступны классу, в котором они описаны, и всем потомкам класса.
internal	Члены класса с таким модификатором видимости доступны любому коду, находящемуся в той же сборке, что и класс.
public	Общедоступные элементы класса. Доступ к таким элементам не ограничивается и работать с ними может любой код, имеющий доступ к самому классу.

Среда .NET допускает только одиночное наследование, когда у класса может быть указан только один непосредственный класс-предок. Однако допускается реализация множества программных интерфейсов.

Для обеспечения полиморфного поведения классов, так же как во многих языках программирования, используется механизм

виртуальных методов. Однако, в отличие от языка C++, синтаксические конструкции описания и перекрытия таких методов отличны друг от друга. Для описания нового виртуального метода используется ключевое слово *virtual*, а для перекрытия уже существующего – *override*.

2.1. Задание к лабораторной работе

Реализовать иерархию классов согласно варианту задания (табл. 3). Классы должны содержать поля, свойства, обычные и виртуальные методы. Виртуальные методы должны иметь разную реализацию в базовых и производных классах. Создать программу, демонстрирующую работу классов. Выполненная работа должна включать исходный текст работоспособной программы и отчет о выполнении лабораторной работы.

Таблица 3. Варианты заданий

№ варианта	*Иерархия классов*
1	Животное, млекопитающее, лошадь, рыбы, насекомые, пауки, собаки, крокодилы.
2	Средства передвижения, пассажир, автомобиль, поезд, ребенок, самолет, ракета, перевозимый груз.
3	Строение, комната, мебель, стул, холодильник, многоэтажное здание, кухня, лампа, окно, дверь.
4	Домашняя утварь, электрическая техника, холодильник, лампа, утюг, механические приборы, ложка, вилка, пылесос, розетка.
5	Принтер, компьютер, компьютерная техника, монитор, клавиатура, запоминающее устройство, материнская плата, процессор, жесткий диск, съемный диск, аудиоколонки.
6	Человек, сотрудник, рабочий, студент, библиотекарь, директор, сотрудники, охранник, кассир, рабочее место, кабинет.
7	Бумага, газета, книга, журнал, учебник, плакат, картина, библиотека, газетный киоск.
8	Ядро атома, протон, нейтрон, электрон, фотон, атом, химический элемент, ион, химическая реакция.

2.2. Контрольные вопросы

1. В чем отличие свойств класса от полей?
2. В чем отличие свойств класса от методов класса?
3. Почему поля класса рекомендуется в обязательном порядке объявлять с модификатором видимости private?
4. Что означает модификатор видимости internal?
5. Что понимается под словом «класс»?
6. Какие модификаторы видимости Вам известны?
7. Для чего используется конструктор класса?
8. Что такое конструктор по умолчанию?
9. Какая область видимости присваивается членам класса по умолчанию?
10. Для чего необходимы виртуальные методы класса? Приведите свой собственный пример, где необходимы виртуальные методы.
11. Могут ли свойства класса быть виртуальными?
12. С помощью какого ключевого слова в языке C# осуществляется переопределение виртуального метода в дочерних классах?
13. Что означает ключевое слово abstract?

3. ЛАБОРАТОРНАЯ РАБОТА №3: ОПИСАНИЕ ИНТЕРФЕЙСОВ

Под интерфейсом мы можем назвать совокупность соглашений в соответствие с которыми ведется взаимодействие двух или более частей системы. В случае с графическими интерфейсами – это соглашение между программистами (компьютером) и пользователями. Например, о том, что, нажав на крестик в верхнем правом углу экрана, соответствующее окно будет закрыто, а не, например, загружено в графический редактор для редактирования или там отправлено другу по почте. В интерфейсе COM1 – это соглашения о распайке кабелей и о правилах обмена сигналами по ним. В случае розеток на стенах (тех, что 220В), что там именно 220В и что для ее использования нужна вилка с двумя торчащими контактами. И так далее.

Интерфейс – это оговоренный стандарт на взаимодействие.

Применительно к языкам программирования, интерфейс – это оговоренный заранее (читай описанный) набор методов и свойств элемента программы.

При этом программистов интересует в случае интерфейсов

не то, чем является объект, а то, поддерживает ли объект нужный интерфейс. Другими словами если в комнате есть розетка с 220В, то я могу сказать, что комната поддерживает интерфейс «Силовая сеть».

Описание интерфейса в языке C# осуществляется в соответствие со следующим синтаксисом

```
  [<модификаторы>] interface <имя-интерфейса>
[: <список-родительских-интерфейсов>]
  {
         <обьявление-свойств-и-методов>
  }
```

При этом надо помнить, что все элементы интерфейса всегда имеют область видимость public и поэтому никакие модификаторы видимости при определении элементов интерфейса не допускаются.

3.1. Задание к лабораторной работе

В соответствие с вариантом задания (Таблица №1) описать указанные интерфейсы и реализовать их в перечисленных классах. Реализовать события для всех основных действий, допустимых по отношению к конкретному классу. Создать программу на языке C#, демонстрирующую работу реализованных классов, использование интерфейсов, возникновение и обработку событий.

Выполненная работа должна включать исходный текст работоспособной программы и отчет о выполнении работы.

Таблица №1. Варианты заданий

№ варианта	*Задание к лабораторной работе*

1	Реализовать системы электрических источников и приборов, соединенных между собой через шнуры. В интерфейсах должны быть предусмотрена возможность получения информации о напряжении и максимальной мощности, которую поддерживает элемент. Прибор должен иметь наименование, потребляемую мощность, а источник и провод – списки подключенных приборов. Интерфейсы: IElectricSource (источник тока) IElectricAppliance (электрический прибор) IElectricWire (электрический шнур) Классы: SolarBattery (солнечная батарея) DieselGenerator (дизельный генератор) NuclearPowerPlant (атомная электростанция) Kettle (чайник) Lathe (токарный станок) Refrigerator (холодильник) ElectricPowerStrip (электрический удлинитель) HighLine (высоковольтная линия) StepDownTransformer (понижающий трансформатор, должен реализовывать интерфейсы и потребителя и источника тока)

2	Реализовать компоненты компьютерной системы, связанные между собой через определенные интерфейсы. Обеспечить возможность стыковки элементов системы между собой в случае совпадения интерфейсов взаимодействия. Интерфейсы в обязательном порядке поддерживать информацию о максимальной скорости передачи данных и возможность передавать как минимум побайтовые данные. Интерфейсы: IUsbBus (шина USB) ISata (шина SATA) INetwork (сеть) IInnerBus (внутренняя шина компьютера) Классы: MotherBoard (материнская плата с процессором) RamMemory (оперативная память) HardDisk (жесткий диск) Printer (принтер) Scanner (сканер изображений) NetworkCard (сетевая карта) Keyboard (клавиатура)
3	Реализовать набор коллекций, реализующих стандартные интерфейсы по работе с коллекциями из пространства имен System.Collections. Интерфейсы: IEnumerable (последовательность элементов) ICollection (коллекция) IList (список) IDictinary (словарь) Классы: List (список) Queue (очередь) Dictinary (словарь)

4. ЛАБОРАТОРНАЯ РАБОТА №4: ДЕЛЕГАТЫ И РАБОТА С СОБЫТИЯМИ

В простейшем случае можно сказать, что делегат (delegate) – это указатель на функцию. По крайней мере, именно указатели на функции выполняли в «старых» языках программирования те задачи, которые в .NET возложены на делегаты.

Формально, делегат определяет прототип функции. То есть набор, типы и последовательности параметров и тип возвращаемого результата. Однако делегаты могут указывать на несколько функций одновременно, что является существенным отличие делегатов от указателей в «старых» языках.

Очень часто делегатов используют там, где необходимо получить возможность изменять поведение алгоритма. Например, при сортировке, обработке полей базы данных и т.п. Общее описание делегата выглядит как:

[<модификатор>] delegate <тип-результата> <имя_делегата>(<список_параметров>);

Например,

public delegate int CompareCats(Cat cat1, Cat cat2);

При этом вызов функций, инкапсулированных внутрь делегата, ничем не отличается от вызова функций через указатели.

События обычно относят к модели программирования с управлением по событиям (event-driven model). Такой подход чаще всего встречается при написании графического пользовательского интерфейса, когда какому-либо классу необходимо сообщить о чем-то для него важном.

Класс как бы публикует (описывает) те события, которые он может генерировать, а другие «подписываются» на них. Функция, которая при этом должна вызываться описывается при помощи делегата. При этом используется правило, согласно которому делегат события должен иметь два параметра: первый – объект инициатор события и второй – данные, передаваемые вместе с событием (являются наследником класса EventArgs). Например,

public delegate void ArraySorted(object sender, ArraySortedEventArgs args);

public event ArraySorted OnArraySorted;

За пределами класса описания, событие может быть использовано только слева от операторов += (подписка) и -= (отказ от подписки). Другое использование событий вне класса не

допускается.

4.1. Задание к лабораторной работе

В соответствие с вариантом задания (Таблица №1) описать указанные интерфейсы и реализовать их в перечисленных классах. Реализовать события для всех основных действий, допустимых по отношению к конкретному классу. Создать программу на языке C#, демонстрирующую работу реализованных классов, использование интерфейсов, возникновение и обработку событий.

Выполненная работа должна включать исходный текст работоспособной программы и отчет о выполнении работы.

Таблица №1. Варианты заданий

№ варианта	*Задание к лабораторной работе*
1	Реализовать системы электрических источников и приборов, соединенных между собой через шнуры. В интерфейсах должны быть предусмотрена возможность получения информации о напряжении и максимальной мощности, которую поддерживает элемент. Прибор должен иметь наименование, потребляемую мощность, а источник и провод – списки подключенных приборов. Интерфейсы: IElectricSource (источник тока) IElectricAppliance (электрический прибор) IElectricWire (электрический шнур) Классы: SolarBattery (солнечная батарея) DieselGenerator (дизельный генератор) NuclearPowerPlant (атомная электростанция) Kettle (чайник) Lathe (токарный станок) Refrigerator (холодильник) ElectricPowerStrip (электрический удлинитель) HighLine (высоковольтная линия) StepDownTransformer (понижающий трансформатор, должен реализовывать интерфейсы и потребителя и источника тока)

2	Реализовать компоненты компьютерной системы, связанные между собой через определенные интерфейсы. Обеспечить возможность стыковки элементов системы между собой в случае совпадения интерфейсов взаимодействия. Интерфейсы в обязательном порядке поддерживать информацию о максимальной скорости передачи данных и возможность передавать как минимум побайтовые данные. Интерфейсы: IUsbBus (шина USB) ISata (шина SATA) INetwork (сеть) IInnerBus (внутренняя шина компьютера) Классы: MotherBoard (материнская плата с процессором) RamMemory (оперативная память) HardDisk (жесткий диск) Printer (принтер) Scanner (сканер изображений) NetworkCard (сетевая карта) Keyboard (клавиатура)
3	Реализовать набор коллекций, реализующих стандартные интерфейсы по работе с коллекциями из пространства имен System.Collections. Интерфейсы: IEnumerable (последовательность элементов) ICollection (коллекция) IList (список) IDictinary (словарь) Классы: List (список) Queue (очередь) Dictinary (словарь)

5. ЛАБОРАТОРНАЯ РАБОТА №5: ИНТЕРФЕЙСЫ IENUMERABLE/IENUMERATOR

Одной из самых часто используемых операций при обработке данных является перебор элементов в последовательностях данных (коллекциях) самого разного вида. Такими последовательностями могут быть массивы, списки, очереди, деревья, стеки, текстовые строки, наборы строк БД и многое другое. В более старых языках программирования для их перебора обычно использовали либо цикл *for*, либо *while*. Условие завершения цикла зависело от типа коллекции, что не позволяло создавать универсальный код обработки данных. Для унификации такой обработки в язык C# был введен цикл *foreach* и два интерфейса: IEnumerable и IEnumerator.

```
public interface IEnumerable {
  IEnumerator GetEnumerator();
}
public interface IEnumerator {
  object Current { get; }
  bool MoveNext();
  void Reset();
}
```

Как можно видеть из их описания, интерфейс *IEnumerable* содержит всего один метод, возвращающий реализацию интерфейса *IEnumerator*. Тот же самый метод необходим циклу *foreach* для перебора коллекции, однако вся основная работа ложится на интерфейс *IEnumerator*. Такое разделение труда было сделано в расчете на возможную обработку одной и той же коллекции в параллельных потоках кода. При параллельной обработке коллекции каждый из операторов *foreach* должен иметь свою собственную информацию о том, какие элементы уже были просмотрены а какие нет. Именно эта роль, – хранителя состояния, – и была возложена на интерфейс *IEnumerator*. Интерфейс *IEnumerable* же должен каждый раз создавать экземпляр класса, реализующего *IEnumerator* и возвращать ссылку на него.

До появления версии 2.0 языка C# реализация классов-итераторов полностью ложилась на плечи программистов и из-за этого была не сильно распространена. Во второй версии языка описание итераторов было кардинально упрощено, а работа по созданию кода классов-итераторов возложена на компилятор.

Решением стали функции-итераторы.

В качестве функции-итератора может выступать любой метод класса (включая виртуальные и статические), описанный с типом результата *IEnumerable*, либо *IEnumerator,* и возвращающий данные с использованием специального оператора «*yield return*». Рассмотрим пример, эмулирующий работу обычного оператора *for*:

```
IEnumerable Range(int init, int upTo)
{
  Console.WriteLine("Начало выполнения итератора...");
  for( int i = init; i < upTo; i++ )
          yield return i;
}
...
Console.WriteLine("Начало программы...");
foreach(int j in Range(0, 10)) Console.Write("{0} ", j);
```

Каждый раз, когда выполнение кода доходит до оператора *yield return*, программа сохраняет полное состояние функции-итератора, а параметр оператора передается осуществляющему перебор циклу *foreach*. После выполнения тела цикла, оператор *foreach* запрашивает следующий элемент последовательности. Это приводит в восстановлению сохраненного состояния функции-итератора и управление передается на следующий за *yield return* оператор. Таким образом, происходит итеративный перебор данных, когда функция-итератор и тело цикла *foreach* выполняются попеременно, передавая управление друг другу. Для рассмотренного выше примера на консоль будет выведен текст «1 2 3 4 5 6 7 8 9».

Функция-итератор может иметь несколько операторов yield return в своем коде, каждый из которых будет выполняться в зависимости от некоторых условий или же безусловно:

```
IEnumerable Range(int init, int upTo)
{
  yield return -1;          // просто вернем какое-то число
  if( upTo >= init ) {
          for( int i = init; i < upTo; i++ )
                  yield return i;
  }
  else {
```

```
            for( int i = init-1; i >= upTo; i-- )
                    yield return i;
    }
}
```

Если же в коде функции-итератора необходимо полностью завершить перебор, можно воспользоваться оператором yield break.

Из-за особенностей реализации функций-итераторов выполнение их кода начинается не в момент их непосредственного вызова, а в момент, когда оператор foreach начинает перебирать коллекцию, формируемую итератором.

Изменим рассмотренный ранее пример:

```
static IEnumerable Range(int init, int upTo)
{
  Console.WriteLine("Начало выполнения итератора...");
  for( int i = init; i < upTo; i++ )
          yield return i;
  Console.WriteLine("\nКонец выполнения итератора...");
}
...
Console.WriteLine("Начало программы...");
var iter = Range(0, 5);
Console.WriteLine("Начало цикла...");
foreach(int j in iter)
  Console.Write("{0} ", j);
Console.WriteLine("Завершение цикла...");
```

Результат работы данного примера приведен ниже на рисунке 1. Обратите внимание, что фраза «Начало выполнения итератора…» была выведена после фразы «Начало цикла…», несмотря на то, что вызов метода Range формально был произведен до цикла. Такое поведение, когда некоторый код выполняются только тогда, когда возникает необходимость в результатах его вычислений, называется ленивыми вычислениями. Эту особенность необходимо учитывать при разработке функций-итераторов, а их код строить так, чтобы его работа зависела только и исключительно от передаваемых параметров.

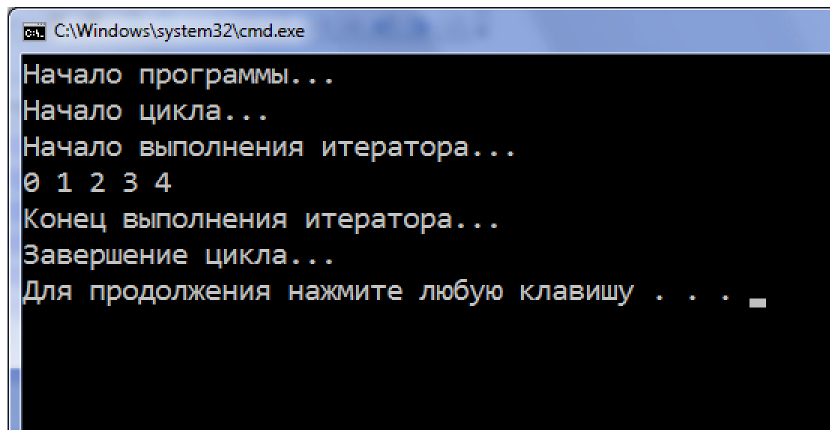

Рисунок 1. Пример отложенного выполнения функции-итератора

.

5.1. Задание к лабораторной работе

В соответствие с вариантом задания (Таблица №1) написать два варианта программы. Первый – с реализацией интерфейсов *IEnumerable*/*IEnumerator*, а второй – с использованием функций возвращающих итераторы и операторы *yield return*. При реализации программ запрещено использовать стандартные классы коллекций библиотеки .NET.

Показать работоспособность программы, произвести ее трассировку и объяснить последовательность выполнения участков кода при трассировке.

Таблица №1. Варианты заданий

№ варианта	*Задание к лабораторной работе*
1	Реализовать расчет математического ожидания и дисперсии для ряда псевдослучайных чисел. Ряд чисел должен генерироваться функцией, возвращающей IEnumerable<double> и принимающей в качестве параметра требуемое количество чисел в ряде.
2	Реализовать в виде последовательности генерацию первых N чисел Фибоначи (N0=1, N1=1, Nn=Nn-1+Nn-2).

3	Реализовать простой вариант бинарного дерева целых чисел с возможность добавления новых элементов. Создать код для обхода дерева в порядке Лево-Корень-Право (ЛКП).
4	Реализовать в виде последовательности генерацию первых N простых чисел.
5	Из входной строки организовать последовательность слов, входящих в строку, отсортированных по длине.
6	Реализовать генерацию последовательности $x_n = \frac{1}{y^n}$. Элементы последовательности генерируются до тех пор, пока не будет выполнено условие $\lvert x_n \rvert \le e$, где e – предварительно заданная точность. При реализации алгоритмов предусмотреть проверку, что $y > 1$.
7	Из заданной строки сформировать последовательность уникальных чисел содержащихся в строке отсортировав их в порядке убывания.

5.2. Контрольные вопросы

1. Почему для перебора элементов в среде .NET используется два интерфейса, а не один?
2. Что содержит свойство Current интерфейса IEnumerator до вызова метода MoveNext()?
3. Как определить, что в коллекции больше нет элементов и перебор завершен?
4. В каких классах может быть реализован интерфейс IEnumerable?
5. Какие стандартные классы, реализующие интерфейс IEnumerable, из библиотеки .NET Вы знаете?
6. Приведите общий шаблон функции, возвращающей итератор.
7. В чем отличие в выполнении оператора «yield return» от оператора «return»?
8. Может ли в функции быть более одного оператора «yield return»?
9. Что такое отложенное выполнение и как оно проявляется

при выполнении программ? Как могут использоваться полезные возможности отложенного выполнения, и какие опасности оно может иметь? Приведите примеры.

6. ЛАБОРАТОРНАЯ РАБОТА №6: ВВОД/ВЫВОД В СРЕДЕ .NET

Платформа Microsoft .Net Framework поддерживает широкий выбор средств по работе с файловой системой (ФС) и другими способами хранения и передачи данных. Данные средства разделяются на две группы: средства работы с двоичными потоками данных (в том числе с файлами), средства чтения и записи текстовой информации. Такое разделение было заложено из-за возможности представления и работы с текстовой информацией во множестве кодировок (кодовых страниц), которые определяются экземплярами класса Encoding. Помимо этих двух групп платформа предоставляет программисту ряд утилитарных классов, облегчающих работу с ФС. Большинство типов данных, используемых при вводе/выводе, расположены в пространстве имен System.IO.

Работа с двоичными данными

Центральное место при работе с двоичными данными в среде .Net занимает абстракция потока данных, реализуемого классом Stream. Под потоком принято понимать некоторую последовательность байт, передаваемую от источника приемнику. При этом особенности реализации ни источника, ни приемника, ни среды передачи не конкретизируются, что позволяет использовать данную абстракцию при описании обмена данными с файлами на дисках (класс FileStream), между процессами в рамках компьютера и по сети (классы PipeStream, NetworkStream и др.), передачу данных с их одновременной обработкой (классы GZipStream, CryptoStream и др.) и т.д. Класс Stream описывает большое количество методов и свойств для работы с двоичными потоками данных, среди которых ключевыми являются методы Read, ReadByte, Write, WriteByte, Seek, CopyTo, Close, Flush (см. табл. 1) и свойства Length и Position. Надо помнить, что не все потоки допускают выполнение операций чтения, записи или изменения текущей позиции и их допустимость может быть проверена через значения свойств CanRead, CanWrite и CanSeek соответственно.

Таблица 1. Наиболее часто используемые методы класса Stream

Метод	*Описание*
int Read(byte[] buffer, int offset, int count)	Считывает последовательность длиной «*count*» байт из потока и помещает их в массив «*buffer*» начиная с индекса «*offset*». Возвращает количество считанных байт данных.
int ReadByte()	Считывает из потока и возвращает один байт. В случае достижения конца потока, возвращает -1.
void Write(byte[] buffer, int offset, int count)	Записывает в поток последовательность байт длиной «count», взятых из массива «buffer» начиная с индекса «offset».
void WriteByte(byte value)	Записывает в поток один байт данных.
long Seek(long offset, SeekOrigin origin)	Изменяет текущее положение указателя в потоке, смещая его на «offset» байт, начиная с позиции «origin».
void CopyTo(Stream destination)	Копирует данные потока с текущей позиции и до конца в поток «destination».
void Close()	Закрывает поток данных, освобождая системные ресурсы.
void Flush()	Очищает кэш в памяти, сохраняя все измененные данные на носитель.

Наиболее часто используемым классом при работе с потоками все же остается класс FileStream – класс, предоставляющий функционал работы с двоичными файлами данных. Его экземпляр может быть получен в программе либо с использованием одного из его многочисленных конструкторов, либо с использованием утилитарных функций класса File, описанных в таблице 5.

Работа с текстовыми данными

В основе библиотеки ввода/вывода текстовой информации в среде .Net лежат два абстрактных класса: TextReader и TextWriter. Каждый из них определяет ряд функциональных примитивов, обеспечивающих соответственно чтение и запись текста, и общих для всех потомков данных классов. Основные методы класса TextReader приведены в таблице 2, а класса TextWriter – в таблице 3.

Таблица 2. Ключевые методы класса TextReader

Метод/Свойство	*Описание*
int Read()	Считывает из текстового потока один символ в виде его кода. В случае достижения конца потока или других ошибок возвращает -1.
int ReadBlock(char[] buffer, int index, int count)	Считывает последовательность символов длиной «count» в текстовый массив «buffer», размещая их, начиная с позиции «index» в массиве.
string ReadLine()	Считывает и возвращает текст с текущей позиции до конца строки.
string ReadToEnd()	Считывает и возвращает текст с текущей позиции до конца текстового потока.

Таблица 3. Ключевые методы класса TextWriter

Метод/Свойство	*Описание*
void Flush()	Сбрасывает незаписанные изменения на носитель, очищая внутренние буферы потока вывода.
string NewLine { get; set; }	Определяет последовательность символом, определяющую конец текстовой строки и перевод на новую строчку. Значение данного свойства специфично для различных ОС.
void Write(…)	Сохраняет переданные данные в текстовый поток. Формат и поведение данных методов аналогично функции Write класса Console.
void WriteLine(…)	Сохраняет переданные данные в текстовый поток вставляя после записанных данных признак конца строки. Формат и поведение данных методов аналогично функции WriteLine класса Console.

Реализацией описанной функциональности занимаются классы StreamReader/Writer, StringReader/Writer. Первый из них организует текстовый ввод/вывод, используя двоичные потоки данных в качестве основы, второй – осуществляет надстройку над текстовой строкой в памяти (типов System.String), позволяя при-

менять к ней идеологию доступа на основе текстовых потоков. Получить конкретные экземпляры данных классов можно либо через их конструкторы, либо (для классов StreamReader/Writer) через утилитарные функции класса File, описанные ниже.

Утилитарные классы

Наиболее часто используемыми вспомогательными классами при работе с ФС являются классы Path, File и Directory. Первый из них (Path) ориентирован на манипулирование строковыми представлениями путей к файлам и каталогам. Наиболее важные из его элементов описаны в табл. 4.

Таблица 4. Наиболее важные элементы класса Path

Метод/Свойство	*Описание*
string Combine(string path1, string path2)	Комбинирует пути, добавляя второй к первому с учетом правильности расстановки разделителей.
string GetDirectoryName(string path)	Для указанного имени файла возвращает путь к нему.
string GetExtension(string path)	Возвращает только расширение файла вместе с ведущим символом «.».
string GetFileName(string path)	Возвращает имя файла вместе с расширением.
string GetTempPath()	Возвращает путь к временному каталогу.

Вспомогательный класс File предоставляет удобный функционал по обработки файлов, их копированию, созданию, удалению, переименованию и т.д, а также ряд методов облегчающих атомарные операции записи/чтения всего содержимого файла. Основные методы класса File приведены в табл. 5.

Таблица 5. Наиболее важные элементы класса File

Метод/Свойство	*Описание*
void Copy(...)	Производит копирование файла по указанному пути с возможным замещением.
FileStream Create(...)	Создает или замещает файл по указанному пути, возвращая файловый двоичный поток.

StreamWriter CreateText(...)	Создает или замещает файл, возвращая текстовый поток в кодировке UTF8 ориентированный на запись.
void Delete(string path)	Удаляет файл по указанному пути.
bool Exists(string path)	Проверяет наличие указанного файла на диске.
StreamReader OpenText(string path)	Открывает существующий файл, возвращая поток в кодировке UTF8 ориентированный на чтение.
byte[] ReadAllBytes(string path)	Считывает все содержимое существующего файла и возвращает его в виде двоичного массива байт.
string[] ReadAllLines(string path)	Возвращает все содержимое текстового файла в кодировке UTF8, считанное построчно.
string ReadAllText(string path)	Считывает все содержимое текстового файла в кодировке UTF8 и возвращает его в виде строки.
void WriteAllBytes(string path, byte[] bytes)	Сохраняет двоичный массив байт в файл, перезаписывая его если необходимо.
void WriteAllLines(string path,string[] contents)	Записывает текстовый файл, сохраняя в нем построчно переданный массив.
void WriteAllText(string path, string contents)	Сохраняет содержимое строки в файл, перезаписывая его если необходимо.

Вспомогательный класс Directory включает основные функции для работы с каталогами ФС, такие как создание, удаление, переименование и получение списка элементов каталогов. Наиболее важные функции представлены в табл. 6.

Таблица 6. Наиболее важные элементы класса Directory

Метод/Свойство	*Описание*
DirectoryInfo CreateDirectory(string path)	Создает все необходимые каталоги и подкаталоги, необходимые для существования указанного пути.
void Delete(string path)	Производит удаление указанного пустого каталога.
bool Exists(string path)	Проверяет наличие указанного каталога.

string GetCurrentDirectory()	Возвращает строковый абсолютный путь к каталогу, считающемуся для программы текущим.
string[] GetDirectories(string path)	Возвращает в виде массива пути к подкаталогам, найденным по указанному пути.
string[] GetFiles(string path)	Возвращает пути доступа к файлам, найденным по указанному пути.
string[] GetLogicalDrives()	Определяет список логических дисков, возвращая пути к их корневым каталогам (например, «D:\»).

6.1. Задание к лабораторной работе

Реализовать программу согласно варианту задания (табл. 7). Для сдачи лабораторной работы представить работоспособный программный проект, и пояснительную записку.

Таблица №7. Варианты заданий к лабораторной работе

№ вар-та	*Задание к лабораторной работе*
1	Написать простой файловый менеджер, имеющий текстовый интерфейс. Менеджер должен выполнять такие функции как копирование, перемещение, удаление файлов, создание каталога, изменение текущего каталога, просмотр содержимого каталога. Для всех функций менеджера предусмотреть обработку исключений с выводом осмысленных сообщений об ошибках на русском языке.
2	Описать библиотеку для посимвольного и построчного копирования одного потока в другой с возможностью фильтрации содержимого. Для фильтрации использовать делегат.
3	Написать программу преобразования текстовых файлов из одной кодировки в другую. Программа должна поддерживать несколько кодировок и осуществлять перекодирование из любой в любую. Параметры для перекодирования должны задаваться через командную строку.

<table>
<tr><td>4</td><td>Реализовать программу монитор, отслеживающую состояние заданного при запуске каталога и выполняющего над новыми файлами некие действия. Список действий должен определяться списком, каждый элемент которого реализует интерфейс вида

interface IFileOperation {
    //надо ли обрабатывать файл
    bool Accept(string fileName);
    //функция-обработчик файла
    void Process(string fileName);
}
Реализовать несколько альтернативных обработчиков и показать работоспособность программного комплекса.</td></tr>
<tr><td>5</td><td>Создать программу анализа лог-файла прокси-сервера, содержащего в каждой строке информацию о клиенте прокси-сервера, запрашиваемом адресе, дате и времени поступления запроса, размере переданных данных и пользователе, запросившем страницу. После анализа предоставить возможность построения текстовых отчетов по суммарным оборотам пользователя, запрошенных доменов за заданный промежуток времени.</td></tr>
</table>

6.2. Контрольные вопросы

1. В чем заключается концепция потока данных?
2. Перечислите стандартные классы, реализующие концепцию потока данных.
3. В чем заключается роль классов File и Directory?
4. Перечислите известные Вам способы открыть файл с двоичными данными.
5. Для чего предназначен класс BufferedStream?
6. Для чего предназначен класс FileSystemWatcher? Приведите примеры использования.
7. В каких случаях удобно использовать класс MemoryStream?
8. По каким причинам классы для работы с текстовыми данными выделены в отдельную иерархию?
9. Какая текстовая кодировка используется по умолчанию при чтении/записи текстовых данных?

7. ЛАБОРАТОРНАЯ РАБОТА №7: НАЗНАЧЕНИЕ РЕФЛЕКСИИ И ПОЗДНЕЕ СВЯЗЫВАНИЕ

7.1. Цель работы

В среде .NET рефлексия – это процесс нахождения (обнаружения) и исследования внутренней структуры сборок, структур и классов непосредственно в процессе работы программы. То есть другими словами непосредственно в программе мы можем получить, например, список всех типов, объявленных в модуле, список всех методов, интерфейсов, полей и свойств этих типов, а также списки атрибутов, примененных к различным элементам. Вся эта информация носит название *метаданных* – данных о данных и описывается рядом специальных типов данных (табл. 1).

Основные возможности, предоставляемые механизмом рефлексии, сосредоточены в классах System.Type, System.Activator и System.Assembly и нескольких вспомогательных типах расположенных в пространстве имен System.Reflection.

Таблица 1. Некоторые типы метаданных

Имя класса	Назначение и описание
MethodInfo	Предоставляет информацию о методе класса или структуры данных. О количестве параметров, их типах, типе возвращаемого значения и т.д.
ParameterInfo	Описывает параметр, передаваемый в метод класса (его тип, имя, порядковый номер в списке параметров).
PropertyInfo	Описывает характеристики свойства класса (тип, имя, возможность чтения/записи и т.д.).

Пример программы, в которой используются средства рефлексии для получения методов, описанных в классе MyClass. Для каждого метода выводится его имя и тип:

```
using System;
using System.Reflection;
class MyClass {
int x;
int y;
public MyClass(int i, int j) { this.x = i; this.y = j; }
public  int  sum()  {  return   x+y; }
```

```
public void set(double a, double b) { x = (int) a; y = (int) b; }
}
class ReflectDemo {
public static void Main() {
// Получаем Type-объект, описывающий MyClass
Type t = typeof(MyClass);
Console.WriteLine("Поддерживаемые методы: " ) ;
foreach(Methodinfo m in t.GetMethods()) {
            // Отображаем тип результата и имя метода
Console.Write(" {0} {1}(...)", m.ReturnType.Name,
                    m.Name) ;
            }
} //Main
} //ReflectDemo
```

Зная, какие методы поддерживает тип, можно вызвать любой из них. Для этого используется метод, определенный в классе MethodInfo как *object Invoke(object ob, object[] args)*, где *obj* соответствует объекту, для которого вызывается метод (аналог параметра *this*), а *args* – набору аргументов, передаваемых в метод. Причем, если метод не имеет аргументов, параметр *args* должен быть представлен массивом нулевой длины (а не значением *null*).

АТРИБУТЫ КАК ЭЛЕМЕНТЫ ДЕКЛАРАТИВНОГО ПРОГРАММИРОВАНИЯ

Любой компилятор может поместить информацию обо всех элементах языка в создаваемый исполняемый файл (делает он это или нет, другой вопрос). Однако добавить что-либо к этой информации программист не может. Другими словами, при желании можно узнать, как называется то или иное поле класса и данные какого типа оно хранит, какие методы есть у класса и т.д. Но узнать из программы кто и когда последний раз редактировал метод или получить описание на русском языке назначения поля класса уже нельзя. Эта информация чаще всего относилась к категории комментариев в коде программы, которые компилятор просто пропускает. Именно нишу дополнительной, определяемой программистом, информации о коде программы в среде .NET заняли атрибуты. Атрибуты позволяют программистам добавлять к декларации основных структурных элементов языка (сборки, классы, поля, методы, параметры методов и др.) своего рода

«комментарии», – произвольно структурированные данные, определяемые в момент написания программы или программной библиотеки и сохраняемые компилятором совместно с метаданными в готовом коде. Таким образом, атрибуты существенно увеличивают возможности по использованию анализа метаданных средствами рефлексии и фактически выводят пользу от такого анализа на принципиально новый уровень. Сама среда программирования Visual Studio и компиляторы языков в среде .NET в большом объеме пользуются атрибутами для описания правил обращения с элементами программ. Например, стандартный атрибут *ConditionalAttribute* указывает компилятору C#, что метод помеченный этим атрибутом (а также все его вызовы) должны присутствовать в программе только в том случае, если определен специальный идентификатор, имя которого указывается в конструкторе атрибута. Если же идентификатор с таким именем на момент компиляции программы не определен, то сам метод и все его вызовы «изымаются» из кода программы, что достаточно широко используется при отладке.

Для создания собственного атрибута необходимо описать новый класс наследник класса *Attribute*, определить в нем необходимые поля и свойства для хранения информации и набор конструкторов для его инициализации:

```
public class TrustMethodAttribute : Attribute {
  private int _trustLevel;
  public int TrustLevel { get { return _trustLevel; } }
  public TrustMethodAttribute(int trustLevel) {
        this._trustLevel = trustLevel;
  }
}
```

Имея готовый атрибут, можно применять его для различных элементов языка:

```
[TrustMethod(4)]
public void FooMethod(int param1) { … }
```

Для контроля использования самого атрибута, в том числе для каких элементов языка может применяться созданный атрибут и будет ли он автоматически наследоваться при его применении для пользовательских классов, необходимо использовать атрибут *AttributeUsageAttribute* с соответствующими параметрами.

Для анализа того, какие атрибуты применены к программной единице, нужно воспользоваться методами *GetCustomAttribute(s)* базового абстрактного класса *MemberInfo*. На основании полученных данных можно принимать решения о дальнейшем поведении алгоритма. В том числе и с использованием данных, хранимых в атрибуте.

7.2. Задание к лабораторной работе

ЗАДАНИЕ 1.

1) Описать класс MyClass, который будет содержать:

- поля различных типов и различным уровнем доступа;
- методы, с различным набором аргументов и различным типом возвращаемого значения.

2) Объявить класс MyTestClass, который будет содержать методы выполняющие следующие действия:

- выводить по имени класса имена методов, которые содержат строковые параметры (имя класса передается в качестве аргумента);
- вызывать некоторый метод класса, при этом значения для его параметров необходимо прочитать из текстового файла (имя класса и имя метода передаются в качестве аргументов).

ЗАДАНИЕ 2

1) Расположить класс MyClass в отдельном .cs-файле и дополнить его следующими членами:

- перегрузить конструктор: один конструктор без параметров, другой с параметрами;
- объявить два интерфейса (IInterface1 и IInterface2) как минимум с двумя методами каждый и реализовать их

- одно из полей объявить как static

2) В классе MyTestClass реализовать метод (принимающий в качестве параметра имя класса), который выводит всё содержимое класса в текстовый файл;

3) Реализовать метод (принимающий в качестве параметра имя класса), который записывает все члены класса в файл *.cs, который должен правильно компилироваться в среде .NET.

ЗАДАНИЕ 3.

С использованием механизма рефлексии и пользовательских атрибутов выполнить один из следующих вариантов:

1) Реализовать атрибут *CommandLineAttribute* с параметром *CommandSwitch* указывающим имя параметра командной строки программы. Атрибут должен применяться к полям и свойствам класса. Написать алгоритм разбора командной строки вида «-<имя-параметра1>[=<значение1>] …» присваивающий соответствующим полям и свойствам объекта значения параметра из командной строки. Должны поддерживаться поля и свойства логического, целочисленного и строкового типов.

2) Реализовать алгоритм отладочной печати для объектов произвольного типа. На экран должны выдаваться строки вида <имя>=<значение>, где *имя* – имя общедоступного поля или свойства, помеченного атрибутом *DebugPrintAttribute* и хранящего значение форматной строки для *значения* (форматная строка по умолчанию – “{0}”).

3) Реализовать возможность сохранения и считывания однотипных объектов из файлов данных на диске (вариант базы данных). Имя файла для каждого класса определяется атрибутом класса *TableNameAttribute*, поля и свойства подлежащие сохранению и считыванию должны помечаться атрибутом *FieldName* с указанием имени поля в файле данных. Реализовать возможность считывания как всех объектов из одного файла сразу (например в массив или список) так и обращение по порядковому номеру в файле.

8. ЛАБОРАТОРНАЯ РАБОТА №8: СЕРИАЛИЗАЦИЯ В СРЕДЕ .NET

8.1. Цель работы

В терминах .NET сериализация – это процесс преобразования объекта в форму в которой он может быть сохранен на внешнем носителе или передан по сети передачи данных между различными вычислительными системами. Обратный процесс восстановления объектов из данных называется десериализацией. В стандартной реализации библиотеки .NET содержится два класса сериализации объектов – BinaryFormatter и SoapFormatter. BinaryFormatter удобен в использовании, когда необходимо получить небольшой по размеру образ объектов, в совокупности с быстрым и простым способом программирования. Кроме того, BinaryFormatter в отличие от XML/SOAP форматирования сохраняет полную структуру объекта, как он представлен в памяти компьютера, сохраняя значения, в том числе private полей.

ДВОИЧНАЯ СЕРИАЛИЗАЦИЯ

Для того чтобы некий объект мог быть сериализован, его класс (в самом простейшем варианте) достаточно пометить атрибутом *Serializable*. Это необходимое и достаточное условие для сохранения данных объекта заданного класса.

В некоторых случаях классы содержат поля, хранящие либо постоянные значения, либо значения, которые после сериализации/десериализации могут не иметь смысла, или не могут сохраняться по соображениям безопасности. Такие поля необходимо пометить атрибутом *NonSerialized*. Например,

```
[Serializable]
public class CarEngine
{
  private readonly int engineId;
  [NonSerialized] public int temperature;
  public bool TurnOn();
  public bool TurnOff();
}
```

Атрибут *Serializable* НЕ наследуется. Таким образом, если возникает необходимость сериализации классов наследников,

каждый из них должен быть помечен соответствующим атрибутом. В противном случае любая попытка обращения к подсистеме сериализации .NET для данного класса будет приводить к исключению *SerializationException*.

Для того чтобы сериализовать приведенный выше класс *CarEngine* в двоичном виде достаточно написать следующий код:

```
public void SerializeCarTo(string fileName, CarEngine care)
{
Stream st = new FileStream(fileName, FileMode.Create);
IFormatter f = new BinaryFormatter();
f.Serialize(st, care);
st.Close();
}
```

а для чтения сохраненного объекта из файла подойдет функция

```
public CarEngine DeserializeCarFrom(string fileName)
{
Stream st = new FileStream(fileName, FileMode.Open, FileAccess.Read);
IFormatter f = new BinaryFormatter();
CarEngine car = f.Deserialize(st);
st.Close();
return car;
}
```

XML/SOAP-СЕРИАЛИЗАЦИЯ

Сериализация программных объектов в формат XML связанно с использованием вместо класса *BinaryFormatter* другого специального объекта *XmlSerializer* описанного в пространстве имен *System.Xml.Serialization*. XML-сериализация производит сохранение только полей и свойств имеющих область видимости *public*. Сохраняемые свойства должны быть доступны как для чтения, так и для записи, и исключение делается только для свойств поддерживающих интерфейс *IList*. По умолчанию, сериализатор при записи объекта создает XML следующего вида

```
<тип-объекта>
  <имя-поля>значение-поля</имя-поля>
  ...
</тип-объекта>
```

записывая все *public* поля, доступные на чтение/запись. Для

игнорирования некоторых полей и исключения их из процесса записи можно использовать атрибут *XmlIgnoreAttribute* выполняющий ту же функцию что и атрибут *NoSerialized* при двоичной сериализации.

В случаях, когда сериализация выполняется в целях обмена информацией с другими системами, или в случаях, когда на XML наложена строгая схема, определяющая имена элементов и атрибутов, имена элементов в XML могут не совпадать с именами полей, объявленных в программных объектах. Для реализации такой возможности предназначен атрибут *XmlElementAttribute*, изменяющий имя элемента XML-файла. Ещё одной возможностью является запись полей объекта в виде атрибутов вместо полей.

```
public class Book {
  [XmlAttribute("Name-Author")]
  public string Author = "неизвестен";
  public string ISBN;
}
```

Результатом будет получен xml-строка следующего вида:

```
<?xml version="1.0" encoding="utf-16"?>
<Book Name-Author="Неизвестен">
  <ISBN>11234</ISBN>
</Book>
```

Сериализация коллекций производиться по общему формату

```
<имя-поля-коллекции>
  <тип-элемента-коллекции>
          поля объекта в коллекции
  </тип-элемента-коллекции>
</имя-поля-коллекции>
```

Применяя к полю атрибут *XmlArrayAttribute* можно заменить имя элемента для поля-коллекции.

```
public class Book {
  [XmlAttribute("Name-Author")]
  public string Author = "неизвестен";
  public string ISBN;
  [XmlArray("par-list"), XmlArrayItem("par")]
  public string[] Paragraphs;
}
```

Атрибут *XmlArrayItemAttribute* изменяет соответственно имя элементов, вложенных в основной элемент коллекции.

```
<?xml version="1.0" encoding="utf-16"?>
<Book Name-Author="Неизвестен">
  <ISBN>11234</ISBN>
  <par-list>
    <par>1</par>
    <par>2</par>
  </par-list>
</Book>
```

Надо учитывать, что если коллекция может содержать различные классы, то тип каждого из них необходимо указать с помощью параметра *type* атрибута *XmlArrayItem*.

8.2. Задание к лабораторной работе

Используя подходящие классы-коллекции из пространств имен System.Collections и System.Collections.Generic выполнить соответствующий вариант задания (Таблица №1), дополнив его возможностями сохранения и чтения данных в/из файлов. Предусмотреть возможность сериализации данных, как в двоичном формате, так и в формате SOAP/XML.

Предоставить работоспособную программу и объяснить выбор того или иного класса, использованного при решении задачи.

Таблица №1. Варианты заданий

№ ва-та	Задание к лабораторной работе
1	Написать двоичное дерево поиска произвольных объектов, обладающее возможность добавления, удаления и поиска элементов. Для сравнения объектов между собой использовать делегат, передаваемый при создании дерева.

2	Сымитировать работу автозаправочной станции. Количество заправочных мест задается при создании объекта «станция». На каждое заправочное место через случайное время (от n до N минут) подъезжают машины. На заправку каждой из них уходит от k до K минут. Если очереди к каждому из заправочных мест более 3 машин очередная машина проезжает мимо стоянки. Посчитать, какое количество машин проехало мимо заправки за S часов ее работы.
3	Организовать БД студентов (фамилия, имя, отчество, дата рождения, адрес, серия паспорта). Для каждого студента сохранять список экзаменов с датой сдачи, наименованием предмета и полученной оценкой. Реализовать функции добавления, удаления, поиска студентов, а также составления текстовых отчетов о среднем бале успеваемости для каждого студента.
4	Создать класс неориентированного графа. Количество вершин и дуг которого может изменяться в процессе работы. Реализовать функции, создания и редактирования вершин и дуг графа, а также процедуру проверки наличия в графе циклов.
5	Создать класс, представляющий неориентированный граф с взвешенными связями. Реализовать функции ввода графа и вычисления пути с наименьшим весом между двумя произвольными вершинами.
6	Напишите простой англо-русский словарь с возможностью хранения нескольких вариантов перевода для одного и того же слова. Напишите программу для работы с таким словарем.
7	Создайте программу для продажи билетов в кинотеатре. Для каждого сеанса необходимо хранить время его начала, наименование кинофильма, длительность сеанса и карта свободных/занятых мест в кинозале (все места считать равноценными).

8.3. Контрольные вопросы

1. Что называют сериализацией в среде .NET?
2. Приведите названия и опишите основные атрибуты, используемые для управления сериализацией в XML.
3. Для чего используется сериализация?

4. Каким условиям должен удовлетворять класс, чтобы его можно было сериализовать в двоичном виде?

5. В каких целях используется интерфейс IDeserializationCallback?

9. ЛАБОРАТОРНАЯ РАБОТА №9: ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА С ИСПОЛЬЗОВАНИЕМ ПОТОКОВ

ПОТОКИ В СРЕДЕ MICROSOFT .NET FRAMEWORK

В среде Microsoft .NET Framework основные классы для организации многопоточных вычислений и их синхронизации сосредоточены в пространстве имен *System.Threading* (см. таблица 1).

Таблица 1 – Основные классы пространства имен *System.Threading*

Тип данных	Описание
Interlocked	Предоставляет набор атомарных операций (сложение, вычитание и др.) над переменными для многопоточных приложений.
Semaphore	Элемент синхронизации, позволяющий ограничить количество потоков имеющих одновременный доступ к некоторому ресурсу.
Thread	Основной класс, представляющий параллельный поток исполнения (под управлением среды CLR).
ThreadPool	Пул потоков, позволяющий отказаться от «ручного» запуска потоков и отдать управление ими среде .NET. Использование пула потоков не гарантирует моментальное начало выполнения запланированного кода.
Timer	Вспомогательный класс, позволяющий выполнять код по некоторому заданному расписанию.

Ключевым классом, конечно же, является класс *Thread*. Он представляет объектно-ориентированную обертку над параллельно выполняемым кодом (и соответствующим объектом ядра операционной системы) в домене .NET-приложения. Его ключевые свойства и методы представлены в таблице 2.

Таблица 2 – Ключевые свойства и методы класса *Thread*

Элемент класса	Описание
Abort()	Отдает приказ среде исполнения CLR прервать исполнение потока как можно быстрее. При этом внутри кода потока будет сгенерировано исключение ThreadAbortException.
IsAlive	Логическое свойство, возвращающее true (истина), если поток был запущен и к моменту чтения свойства не завершен (состояние не Terminated и не Aborted).
Join()	Блокирует вызывающий поток до момента пока поток, связанный с методом, не будет завершен (ожидание завершения потока).
Priority	Свойство, позволяющее считать или изменить приоритет выполнения потока. Доступные приоритеты перечислены в перечислении ThreadPriority.
Resume()	Возобновляет исполнение ранее приостановленного потока (см. метод Suspend).
Start()	Запускает поток на исполнение, ставя его в очередь планирования многозадачной среды.
Suspend()	Приостанавливает выполнение потока. Если потока к моменту вызова уже был приостановлен, вызов не приводит к изменению состояния потока.
ThreadState	Возвращает текущее состояние потока.

9.1. Задание к лабораторной работе

Используя потоки (класс *Thread*) операционной системы или пулы потоков, а также необходимые средства синхронизации параллельных процессов из пространства имен *System.Threading*, реализовать программу согласно варианту задания (см. таблица 3).

При выполнении задания учесть, что количество активно работающих потоков не должно превышать количества установленных в компьютере логических процессоров.

Таблица 3 – Варианты заданий

№ ва-та	Задание к лабораторной работе
1	Предполагая, что в некотором каталоге на диске сохранено большое количество файлов с логами (журналами работы) прокси-сервера, написать программу вычисляющую статистику потребления трафика сети Интернет. На выходе программа должна создавать три текстовых документа: статистика по пользователям, статистика по доменам, статистика по датам. В качестве статистики использовать общий объем потребленного трафика, соответственно пользователем за все дни, при обращении к домену, в указанный день. При разработке программы считать, что каждый файл должен обрабатываться отдельно параллельно выполняющимся участком кода. После обработки всех файлов, полученные результаты для каждого из них должны суммироваться в общую сводку.
2	Реализовать параллельный алгоритм умножения матрицы на вектор. Количество параллельно работающих участков кода должно соответствовать количеству процессоров на компьютере. Значения элементов исходной матрицы и вектора должны считываться из текстовых файлов. Результирующий вектор записывается в новый файл.
3	Разработать программу для вычисления определенного интеграла от заданной функции на заданном отрезке методом прямоугольников. Общий интервал интегрирования должен разбиваться на подинтервалы согласно количеству параллельно выполняемых участков. По окончании параллельных вычислений, их результаты должны складываться и общее значение интеграла выводиться на экран. Интервал шаг интегрирования должен задаваться пользователем после запуска программы. В качестве проверочных функций можно использовать: a) sin(x) * cos(x2) b) ln(x) + 10*cos(x) c) xx – 10*sin(5*x)

4	Написать программу, параллельно вычисляющую значения числа Пи методом Монте-Карло. Общее количество случайных точек для всех параллельных потоков должно задаваться пользователем после запуска программы. Результаты вычислений параллельных участков после их завершения должны усредняться. Полученный результат вывести на экран.
5	Разработать программу для сравнения эффективности двух алгоритмов сортировки путем их одновременного запуска на случайном массиве из 500 000 целых чисел (случайные числа генерировать в диапазоне от 0 до 1000 000). Для повышения точности сравнения проводить сортировку каждым из алгоритмов не менее 10 раз. Результаты усреднять. Алгоритмы сортировки должны быть выбраны из следующего списка: a) быстрая сортировка b) сортировка пузырьком c) сортировка прямым включением d) сортировка выбором e) сортировка Шелла

9.2. Контрольные вопросы

1. В чем сходство и отличие потоков от процессов? В каких случаях использование потоков имеет преимущество?

2. Какие ограничения для ускорения работы параллельных алгоритмов вы можете назвать?

3. Приведите примеры необходимости использования средств синхронизации при распараллеливании алгоритмов.

4. В чем отличие между созданием потока и использованием пула потоков? Какие преимущества и недостатки у каждого способа?

5. Перечислите ключевые классы библиотеки FCL .NET для работы с потоками.

6. Какими особенностями должен обладать алгоритм, поддающийся эффективному распараллеливанию?

7. Назовите причины, по которым время работы параллельных алгоритмов может разли- чаться от раза к разу?

10. ЛАБОРАТОРНАЯ РАБОТА №10: ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ TPL (TASK PARALLEL LIBRARY)

ЭЛЕМЕНТЫ БИБЛИОТЕКИ TPL

Основные классы библиотеки (см. таблицу 1) Task Parallel Library (TPL) нацелены на упрощение работы программиста при разработке параллельных и асинхронных алгоритмов. Основной единицей, с которой придется иметь дело при работе с TPL, является Задача (Task), инкапсулирующая некоторый асинхронно выполняемый алгоритм. В отличие от потока (Thread) программист не может прямо управлять планированием кода задачи. Будучи созданной и запущенной, задача будет поставлена в очередь на выполнение, которое начнется, как только появится свободный поток, способный обслужить код задачи. Кроме того, поскольку задача является также и асинхронным примитивом, несколько задач могут одновременно выполняться, используя один и тот же поток, что повышается общую производительность вычислений.

Таблица 1 – Основные классы пространства имен *System.Threading.Tasks*

Тип данных	Описание
Parallel	Вспомогательный утилитарный класс, содержащий параллельные реализации циклов for и foreach, а также метод Invoke, позволяющий запускать в синхронно-параллельном режиме множество указанных вычислений.
Task	Описывает одну асинхронно выполняемую единицу, не имеющую результирующего значения.
Task<TResult>	Описывает одну асинхронно выполняемую единицу, возвращающую по окончанию значение типа TResult.
TaskFactory и TaskFactory<T>	Вспомогательные классы, облегчающие создание и планирование работы различного рода задач.

10.1. Задание к лабораторной работе

Используя задачи (объекты классов Task и Task<TResult>), реализовать программу согласно варианту задания (см. таблица 2). Спроектировать задачу так, чтобы, несмотря на количество созданных в процессе выполнения задач, завершение выполнения всего расчета можно было отследить по одной, объединяющей остальные, задаче.

Плюсом будет считаться использование продолжений (continuations) и технологии LINQ там, где это представляется возможным.

Таблица 3 – Варианты заданий

№ ва-та	Задание к лабораторной работе
1	Написать программу, параллельно вычисляющую значения числа Пи методом Монте-Карло. Общее количество случайных точек для всех параллельных потоков должно задаваться пользователем после запуска программы. Результаты вычислений параллельных участков после их завершения должны усредняться. Полученный результат вывести на экран.
2	Имея на входе каталог с текстовыми файлами (литературные произведения) составьте частотный словарь всех слов, букв и двухсловных сочетаний, встреченных в текстах. Полученные словари запишите в файлы формата CSV.
3	Число 197 является довольно любопытным числом, так как все возможные числа, полученные из него путем циклического сдвига цифр, являются простыми (197, 971, 719). Для чисел меньше ста таким свойством обладают 13 чисел: 2, 3, 5, 7, 11, 13, 17, 31, 37, 71, 73, 79 и 97. Найдите список всех подобных чисел меньших некоторой указанной в виде параметра границы.

4	Предполагая, что в некотором каталоге на диске сохранено большое количество файлов с логами (журналами работы) прокси-сервера, написать программу вычисляющую статистику потребления трафика сети Интернет. Результатом обработки файлов должны стать три словаря (объекты типа Dictionary), содержащие суммарный переданный объем трафика для пользователей, доменов и дней недели. Например, для пользователей словарь должен быть типа Dictionary<string, long> в котором ключом является имя пользователя, а значением – потребленный трафик.
5	Разработать программу для вычисления определенного интеграла от заданной функции на заданном отрезке методом прямоугольников. Интервал шаг интегрирования должен задаваться пользователем после запуска программы. В качестве проверочных функций можно использовать: a) sin(x) * cos(x2) b) ln(x) + 10*cos(x) c) xx – 10*sin(5*x)

СПИСОК ЛИТЕРАТУРЫ

1. Павловская Т.А. C#. Программирование на языке высокого уровня: учебник для ВУЗов. – СПб.: Питер, 2013.
2. А. Хейлсберг и др. Язык программирования C#. – СПб.: Питер, 2012.
3. Туральчук К.А. Параллельное программирование с помощью языка C#. – Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. – URL: http://www.iprbookshop.ru/39560.html