# Лабораторная работа №16

# Работа с файлами.

## Часть 1. Работа с файлами

## 1. Цель работы:

1. Получение практических навыков при работe классами.
2. Получение практических навыков при работе с коллекциями.
3. Получение практических навыков при создании диалоговых Windows-приложений.
4. Получение практических навыков при работе с файлами.

## 2. Теоретические сведения

### 2.1.1. Понятие потока

Под **файлом** обычно подразумевается именованная информация на внешнем носителе, например на жестком или гибком магнитном диске. Логически файл можно представить как конечное количество последовательных байтов, поэтому такие устройства, как дисплей, клавиатура и принтер, также можно рассматривать как частные случаи файлов.

Обмен данными реализуется с помощью **потоков**. Поток(stream) — это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику.

Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер).

Обмен с потоком для повышения скорости передачи данных производится, как правило, через специальную область оперативной памяти — **буфер***.* Буфер выделяется для каждого открытого файла. При записи в файл вся информация сначала направляется в буфер и там накапливается до тех пор, пока весь буфер не заполнится. Только после этого или после специальной команды сброса происходит передача данных на внешнее устройство. При чтении из файла данные вначале считываются в буфер, причем не столько, сколько запрашивается, а сколько помещается в буфер.

Для поддержки потоков библиотека .NET содержит иерархию классов.

**Основные классы пространства имен System.IO**

|  |  |
| --- | --- |
| Класс | Описание |
| BinaryReader,  BinaryWriter | Чтение и запись значений простых встроенных типов (целочисленных, логических, строковых и т. п.) во внутренней форме представления |
| FileStream | Произвольный (прямой) доступ к файлу, представленному как поток байтов |
| StreamWriter, StreamReader | Чтение из файла и запись в файл текстовой информации (произвольный доступ не поддерживается) |

Таким образом, выполнять обмен с внешними устройствами можно на уровне:

* двоичного представления данных (BinaryReader, BinaryWriter);
* байтов (FileStream);
* текста, то есть символов (StreamWriter, StreamReader).

В .NET используется кодировка Unicode, в которой каждый символ кодируется двумя байтами. Классы, работающие с **текстом**, являются оболочками классов, использующих байты, и автоматически выполняют перекодирование из байтов в символы и обратно.

**Двоичные и байтовые** потоки хранят данные в том же виде, в котором они представлены в оперативной памяти, то есть при обмене с файлом происходит побитовое копирование информации. Двоичные файлы применяются не для просмотра их человеком, а для использования в программах.

Доступ к файлам может быть **последовательным**, когда очередной элемент можно прочитать (записать) только после аналогичной операции с предыдущим элементом, и **прямым**, при котором выполняется чтение (запись) произвольного элемента по заданному адресу. Текстовые файлы позволяют выполнять только последовательный доступ, в двоичных и байтовых потоках можно использовать оба метода.

Прямой доступ в сочетании с отсутствием преобразований обеспечивает высокую скорость получения нужной информации.

Использование классов файловых потоков в программе предполагает следующие операции:

1. Создание потока и связывание его с физическим файлом.
2. Обмен (ввод-вывод).
3. Закрытие файла.

Каждый класс файловых потоков содержит несколько вариантов конструкторов, с помощью которых можно создавать объекты этих классов различными способами и в различных режимах. Например, файлы можно открывать только для чтения, только для записи или для чтения и записи.

**Режимы доступа к файлу содержатся в перечислении FileAccess, определенном в пространстве имен System.IO.**

|  |  |
| --- | --- |
| Значение | Описание |
| Read | Открыть файл только для чтения |
| ReadWrite | Открыть файл для чтения и записи |
| Write | Открыть файл только для записи |

**Возможные режимы открытия файла определены в перечислении FileMode**

|  |  |
| --- | --- |
| Значение | Описание |
| Append | Открыть файл, если он существует, и установить текущий указатель в конец файла. Если файл не существует, создать новый файл. |
| Create | Создать новый файл. Если в каталоге уже существует файл с таким же именем, он будет стерт. |
| CreateNew | Создать новый файл. Если в каталоге уже существует файл с таким же именем, возникает исключение IOException. |
| Open | Открыть существующий файл. |
| OpenOrCreate | Открыть файл, если он существует. Если нет, создать файл с таким именем |
| Truncate | Открыть существующий файл. После открытия он должен быть обрезан до нулевой длины. |

#### 2.2. Исключительные ситуации при работе с файлами

Операции по открытию файлов могут завершиться неудачно, например, при ошибке в имени существующего файла или при отсутствии свободного места на диске, поэтому рекомендуется всегда контролировать результаты этих операций.

В случае непредвиденных ситуаций среда выполнения генерирует различные исключения, обработку которых следует предусмотреть в программе, например:

* FileNotFoundException, если файла с указанным именем в указанном каталоге не существует;
* DirectoryNotFoundException, если не существует указанный каталог;
* Argument Except ion, если неверно задан режим открытия файла;
* IOException, если файл не открывается из-за ошибок ввода-вывода.

При закрытии файла освобождаются все связанные с ним ресурсы, например, для файла, открытого для записи, в файл выгружается содержимое буфера. Поэтому рекомендуется всегда закрывать файлы после окончания работы, в особенности файлы, открытые для записи. Если буфер требуется выгрузить, не закрывая файл, используется метод Flush.

#### 2.3. FileStream (Потоки байтов)

Ввод-вывод в файл на уровне байтов выполняется с помощью класса FileStream, который является наследником абстрактного класса Stream, определяющего набор стандартных операций с потоками.

**Элементы класса Stream**

|  |  |
| --- | --- |
| Значение | Описание |
| CanRead,  CanSeek,  CanWrite | Свойства, определяющие, какие операции поддерживает поток: чтение, прямой доступ и/или запись |
| Сlose | Закрыть текущий поток и освободить связанные с ним ресурсы (сокеты, указатели на файлы и т. п.) |
| Flush | Записать данные из буфера в связанный с потоком источник данных и очистить буфер. Если для данного потока буфер не используется, то этот метод ничего не делает |
| Length | Возвратить длину потока в байтах |
| Position | Возвратить текущую позицию в потоке |
| Read,  ReadByte | Считать последовательность байтов (или один байт) из текущего потока и переместить указатель в потоке на количество считанных байтов |
| ReadAsync | асинхронная версия метода Read |
| Seek | Установить текущий указатель потока на заданную позицию |
| SetLength | Установить длину текущего потока |
| Write,  WriteByte | Записать последовательность байтов (или один байт) в текущий потоки переместить указатель в потоке на количество записанных байтов |
| WriteAsync | асинхронная версия метода Write |

Текущая позиция в потоке первоначально устанавливается на начало файла (для любого режима открытия, кроме Append) и сдвигается на одну позицию при записи каждого байта.

Для установки желаемой позиции чтения используется метод Seek, имеющий два параметра: первый задает смещение в байтах относительно точки отсчета, задаваемой вторым. Точки отсчета задаются константами перечисления SeekOrigin:

* начало файла — Begin,
* текущая позиция — Current,
* конец файла — End.

#### 2.4. StreamWriter и StreamReader (Потоки символов)

Символьные потоки StreamWriter и StreamReader работают с Unicode-символами, следовательно, ими удобнее всего пользоваться для работы с файлами, предназначенными для восприятия человеком. Эти потоки являются наследниками классов TextWriter и TextReader соответственно, которые обеспечивают их большей частью функциональности.

**Класс StreamWriter**

|  |  |
| --- | --- |
| Значение | Описание |
| Close | Закрыть файл и освободить связанные с ним ресурсы. Если в процессе записи используется буфер, он будет автоматически очищен |
| Flush | Очистить все буферы для текущего файла и записать накопленные в них данные в место их постоянного хранения. Сам файл при этом не закрывается |
| NewLine | Используется для задания последовательности символов, означающих  начало новой строки. |
| Write | Записать фрагмент текста в поток |
| WriteAsync | асинхронная версия метода Write. |
| WriteLinе | Записать строку в поток и перейти на другую строку |
| WriteLineAsync | асинхронная версия метода WriteLine |

**Класс StreamReader**

|  |  |
| --- | --- |
| Значение | Описание |
| Peek | Возвратить следующий символ, не изменяя позицию указателя в файле |
| Read | Считать данные из входного потока |
| ReadAsync | асинхронная версия метода Read |
| ReadBlock | Считать из входного потока указанное пользователем количество символов и записать их в буфер, начиная с заданной позиции |
| ReadLine | Считать строку из текущего потока и возвратить ее как значение типа string. Пустая строка (null) означает конец файла (EOF) |
| ReadLineAsync() | асинхронная версия метода ReadLine |
| ReadToEnd | Считать все символы до конца потока, начиная с текущей позиции, и возвратить считанные данные как одну строку типа string |
| ReadToEndAsync() | асинхронная версия метода ReadToEnd |

#### 2.5. Сериализация

В С# есть возможность сохранять на внешних носителях не только данные примитивных типов, но и объекты. Сохранение объектов называется сериализацией, а восстановление сохраненных объектов — десериализацией. При сериализации объект преобразуется в линейную последовательность байтов. Это сложный процесс, поскольку объект может включать множество унаследованных полей и ссылки на вложенные объекты, которые, в свою очередь, тоже могут состоять из объектов сложной структуры.

Cериализация выполняется автоматически, достаточно просто пометить класс как сериализуемый с помощью атрибута [Serializable]. Атрибуты — это дополнительные

сведения о классе, которые сохраняются в его метаданных. Те поля, которые сохранять не требуется, помечаются атрибутом [NonSerialized].

Существует несколько форматов для сериализации. Для каждого формата предусмотрен свой класс:

* + бинарный – класс BinaryFormatter;
  + xml (eXtensible Markup Language) – класс XmlSerializer;
  + JSON (JavaScript Object Notation) – класс DataContractJsonSerializer**.**

**BinaryFormatter**

Рассмотрим *сохранение объектов в двоичном формате.* Для этого используется класс BinaryFormatter, в котором определены два метода:

* Serialize( поток, объект ) - сохраняет заданный объект в заданном потоке;
* Deserialize (поток ) - восстанавливает объект из заданного потока.

Для *сохранения объекта в двоичном формате необходимо:*

1. Подключить к программе пространство имен System.Runtime.Serialization.

Formatters.Binary.

2. Пометить сохраняемый класс и связанные с ним классы атрибутом [Serializable].

3. Создать поток и связать его с файлом на диске или с областью оперативной памяти.

4. Создать объект класса BinaryFormatter.

5. Сохранить объекты в потоке.

6. Закрыть файл.

**JsonSerializer**

JSON (JavaScript Object Notation) является одним из наиболее популярных форматов для хранения и передачи данных. Класс JsonSerializer позволяет сериализовать объект в json и, наоборот, десериализовать код json в объект C#. Для его использования нужно подключить пространство имен System.Text.Json.

Для сохранения объекта в json в классе JsonSerializer определен статический метод Serialize() и его асинхронный двойник SerializeAsyc(), которые имеют ряд перегруженных версий. Для десериализации кода json в объект C# применяется метод Deserialize() и его асинхронный аналог DeserializeAsync().

Объект, который подвергается десериализации, должен иметь либо конструктор без параметров, либо конструктор, для всех параметров которого в десериализуемом json-объекте есть значения (соответствие между параметрами конструктора и свойствами json-объекта устанавливается на основе названий, причем регистр не играет значения).

**XML – сериализация**

XML-документ должен иметь один единственный корневой элемент, внутрь которого помещаются все остальные элементы.

В данном случае таким элементом является элемент <people>. Внутри корневого элемента <people> задан набор элементов <person>.

Вне корневого элемента мы не можем разместить элементы person. Каждый элемент определяется с помощью открывающего и закрывающего тегов, например, <person> и </person>, внутри которых помещается значение или содержимое элементов. Также элемент может иметь сокращенное объявление: <person /> - в конце элемента помещается слеш.

Элемент может иметь вложенные элементы и атрибуты.

Атрибуты определяются в теле элемента и имеют следующую форму: название="значение".

Для сериализации нужно подключить System.Xml.Serialization.

Класс XmlSerializer упрощает сохранение сложных объектов в формат xml и последующее их извлечение.

Для создания объекта XmlSerializer можно применять различные конструкторы, но почти все они требуют указания типа данных, которые будут сериализоваться и десериализоваться.

Класс, подлежащий сериализации, должен иметь стандартный конструктор без параметров и должен иметь модификатор доступа public. Сериализации подлежат только элементы со спецификатором доступа public.

## 3. Постановка задачи

Написать Windows-приложение для работы с коллекциями. Приложение должно выполнять следующие функции:

1. Сохранять в коллекции заданную информацию.
2. Выполнять просмотр сохраненной информации, если коллекция пустая, то выдавать соответствующее сообщение.
3. Выполнять удаление элементов из коллекции (по ключу).
4. Выполнять корректировку элементов коллекции (по ключу).
5. Выполнять добавление элементов в коллекцию.
6. Сохранять коллекцию в файл (при сохранении пользователь должен иметь возможность выбрать один из 3 способов сериализации: бинарную, JSON или XML).
7. Реализовать не менее 3 запросов, фильтрацию и сортировку по выбранному полю.
8. Загрузить сохраненную коллекцию из файла.

## 4. Методические указания

1. Для выбора действий использовать меню.
2. Для выбора файлов использовать стандартные диалоговые окна.
3. Для добавления и удаления элементов использовать обобщенные коллекции из лаб. работы 11.
4. Предусмотреть заполнение коллекции случайными значениями.
5. Предусмотреть обработку ошибок при чтении из файла, не содержащего информации о требуемых объектах.
6. Рекомендуется разделить функции, выполняющие обработку событий элементов управления и функции, содержащие бизнес логику.

## 5. Содержание отчета

1. Постановка задачи (общая и для конкретного варианта).
2. Диаграмма классов
3. Листинг программы.
4. Тесты для каждой функции. (unit тесты или Excel)
5. Анализ покрытия тестами программы.