ПИШНИК: BECOME HUMAN

Глава 15: Потоки данных часть 1. Паттерн IDisposable. BitConverter и Encoding.

Автор презентации – Сагалов Даниил БПИ-196

Предисловие

От автора:

Пространство имён System.IO затрагивается повторно неспроста: на разных этапах различные идеи и принципы работы вещей легче осознаются. Это второй раз, когда мы сталкиваемся с System.IO, впервые мы разбирали классы File, FileInfo, Directory, DirectoryInfo и Path.

На данной лекции будет рассмотрена концепция потоков данных в языке C#, однако у нас ещё будет повод вернуться к System.IO: материалу по использованию асинхронных методов для работы с файлами, а также таким понятиям, как файловый дескриптор будут посвящены отдельные лекции.

Концепция Потока Данных

Поток данных (англ. Stream) – абстракция, представляющая собой последовательность байтов (по сути, как некоторый источник данных). Она позволяет описать общие принципы взаимодействия программ с источниками данных различной природы (память компьютера, интернет, базы данных и т. д.).

Работа с потоками включает 3 основные операциях:

- **Чтении из потока** – передаче данных из потока в структуру данных.
- **Записи в поток** – передаче данных из структуры данных в поток.
- **Поиске (позиционировании) в потоке**. Данная операция опциональна, она доступна не для всех типов потоков. Возможность позиционирования зависит от типа источника данных, например, Вы не сможете изменить позицию в сетевом потоке.

Потоки в .NET представлены абстрактным базовым классом **System.IO.Stream** и его наследниками. Данный класс реализует интерфейс **IDisposable** и **IAsyncDisposable**, начиная с C# 8.0.

Буфер при Работе с Потоками. Stream.Null

При работе с потоками используется понятие **буфера**. Дело в том, что источник данных может быть крайне объёмным, по этой причине чтение его сразу и полностью может быть крайне объёмным. Аналогичную вещь можно сказать и о записи крупного массива данных в источник сразу. Буфер позволяет считывать или загружать данные порциями, по частям, решаю проблему работы с большим количеством данных.

В классе Stream есть одно единственное специальное открытое поле: **public static readonly System.IO.Stream Null**, которое представляет поток без резервного хранилища. Вы можете использовать его для того, чтобы перенаправить вывод в поток, не потребляющий ресурсы операционной системы. Чтение из Null всегда возвращает ноль без каких-либо дополнительных действий.

Интерфейс IDisposable

IDisposable – крайне важный интерфейс, предназначенный для типов, поддерживающий механизм очистки ресурсов (в основном, неуправляемых).

Как мы знаем, очистка управляемых ресурсов в языке C# осуществляется сборщиком мусора автоматически, если объект больше не используется. При этом возникает пара проблем:

- Невозможно точно предсказать, когда произойдёт сборка мусора.
- Сборщик мусора в принципе не имеет представления об используемых неуправляемых ресурсах (таких как файлы, дескрипторы или потоки).

IDisposable требует реализовать единственный метод – **public void Dispose( )**, предназначенный для ручной активации механизма очистки ресурсов.

Тем не менее, одной простой реализации Dispose( ) зачастую недостаточно, т. к. очистка ресурсов также должна корректно осуществляться при наследовании. Оператор using позволяет осуществлять корректную очистку ресурсов автоматически, подробнее о нём на последующих слайдах.

Паттерн IDisposable

При реализации IDisposable следует придерживаться следующей логики:

1) Добавьте bool disposed = false. Данное поле указывает, была ли произведена очистка.

2) Добавьте открытый метод Dispose( ), вызывающий ручную очистку Dispose(true) (см. шаг 3 и пример на следующем слайде) и сообщающий сборщику мусора, что выполнять автоматическую очистку данного объекта не нужно – вызовите GC.SuppressFinalize(this).

3) Добавьте Защищённый виртуальный метод без возвращаемого значения: Dispose(bool disposing). Если disposed = true, сразу завершите выполнение метода (с помощью return). Далее, если аргумент disposing = true, в ручную очистите управляемые ресурсы, после чего вне зависимости от значения disposing очистите имеющиеся неуправляемые ресурсы. В самом конце установите значение disposed = true.

4) Деструктор типа, вызывающий Dispose(false). False используется т. к. сборщик мусора и так корректно очистит все управляемые ресурсы.

Наследникам стоит переопределять метод Dispose(bool disposing), а также по цепочке очищать ресурсы родителей.

Реализация Паттерна IDisposable

```
class BaseClass : IDisposable {
        // Выполнялась ли очистка ресурсов?
        bool disposed = false;

        // Открытая реализация очистки ресурсов
        Ссылок: 0
        public void Dispose() {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        // Защищённая переопределяемая реализация очистки
        Ссылок: 2
        protected virtual void Dispose(bool disposing) {
            // Сразу завершаем очистку, если она уже производилась
            if (disposed)
                return;

            if (disposing) {
                // Здесь происходит очистка управляемых ресурсов
            }

            // Здесь происходит очистка неуправляемых ресурсов
            disposed = true;
        }

        // Очистка ресурсов сборщиком мусора
        Ссылок: 0
        ~BaseClass() {
            Dispose(false);
        }
    }
```

```
class DerivedClass : BaseClass {
    // Выполнялась ли очистка ресурсов?
    // Значение disposed для наследника своё.
    bool disposed = false;

    // Переопределение защищённой реализации очистки
    Ссылок: 5
    protected override void Dispose(bool disposing) {
        // Сразу завершаем очистку, если она уже производилась
        if (disposed)
            return;

        if (disposing) {
            // Здесь происходит очистка управляемых ресурсов
        }

        // Здесь происходит очистка неуправляемых ресурсов
        disposed = true;

        // Крайне важно - наследник по цепочке вызывает
        // реализации очистки базовых классов.
        base.Dispose(disposing);
    }

    // Очистка ресурсов сборщиком мусора
    Ссылок: 0
    ~DerivedClass() {
        Dispose(false);
    }
}
```

Члены Абстрактного Класса Stream

Свойства:

public abstract bool CanRead/CanWrite/CanSeek { get; } – при переопределении в производном классе должны возвращать true, если поток допускает **чтение**, **запись** и **позиционирование** соответственно.

public abstract long Length { get; } – при переопределении в производном классе должно возвращать длину потока в байтах, или выбрасывать исключения, если этот функционал не предполагается.

public abstract long Position { get; set; } – при переопределении в производном классе должно возвращать или задавать текущую позицию в потоке, если в потоке разрешено позиционирование.

public virtual bool CanTimeout { get; } – при переопределении в производном классе должно возвращать true, если для потока существует понятие истечения времени ожидания и false в противном случае. По умолчанию без переопределения всегда возвращает false.

public virtual int ReadTimeout/WriteTimeout { get; set; } – при переопределении в производном классе должны возвращать/задавать время в миллисекундах, которое проходит до истечения времени ожидания при чтении из потока или при записи в него. По умолчанию без переопределения всегда выбрасывает **InvalidOperationException**. Следует использовать с потоками, имеющими CanTimeout = true.

Члены Абстрактного Класса Stream

Основные Методы:

public abstract void Flush( ) – при переопределении в производном классе должен очищать все буферы потока и записывать их содержимое в источник. В целях совместимости, для не поддерживающих запись потоков не рекомендуется выбрасывать исключение данным методом, вместо этого реализуйте его пустым.

public virtual void Close( ) – закрывает поток, отключает и очищает все связанные с ним ресурсы. По умолчанию вызывает Dispose(true), поэтому вместо переопределения данного метода рекомендуется переопределять Dispose(bool disposing). Попытка обращения к закрытому потоку приводит к **ObjectDisposedException**.

public abstract long Seek (long offset, SeekOrigin origin) – при переопределении в производном классе должен перемещать текущую позицию в потока со смещением относительно параметра перечисления **SeekOrigin**: Begin – 0 (начало потока), Current – 1 (текущая позиция) или End (конец потока) на offset и вернуть значение установленной позиции. offset может и положительным (сдвиг вправо), и отрицательным (сдвиг влево) и равным нулю (без сдвига), допускается позиционирование за границы потока. Для проверки возможности позиционирования используйте свойство CanSeek.

public abstract void SetLength (long value) – при переопределении в производном классе должен задавать длину потока. SetLength должен использоваться только с потоками, которые поддерживают и запись (CanWrite) и поиск (CanSeek). Поток может быть как расширен, так и усечён в зависимости от переданного значения value.

Члены Абстрактного Класса Stream

public abstract int Read(byte[] buffer, int offset, int count) – при переопределении в производном классе должен считать последовательность байтов из текущего потока и переместить позицию в потоке на указанное число байтов. buffer после выполнения метода содержит массив байтов, в котором содержатся считанные из потока байты в диапазоне [offset; (offset + count – 1)]. offset – смещение индекса в buffer (с 0), с которого начинается сохранение данных из потока в массив, count – максимальное количество байтов, которое должно быть считано из потока. Возвращаемое значение – количество считанных в буфер байтов, оно может отличаться от запрошенного количества, если часть байтов недоступна или равняться 0, если достигнут конец потока. При возникновении исключения позиция в потоке не изменится.

public virtual int ReadByte( ) – считывает байт из потока, перемещает позицию в потоке на 1 и возвращает байт без знака, приведённый к int или -1, если достигнут конец потока. Для проверки возможности чтения используйте свойство CanRead. Имейте в виду, что реализация по умолчанию создаёт новый однобайтовый массив и вызывает метод **Read**, что не является эффективным решением. При переопределении потоки с буфером должны предоставлять более эффективную реализацию, избегающую дополнительное создание массива.

Члены Абстрактного Класса Stream

public abstract void Write(byte[] buffer, int offset, int count) – при переопределении в производном классе должен записать данные из буфера в поток и переместить позицию в нём вперёд на количество записанных байтов. offset – отступ индекса относительно 0 в буфере, count – количество байтов, которое нужно записать в поток. При возникновении исключения позиция в потоке не изменится.

public virtual void WriteByte(byte value) – записывает переданное значение в поток на текущую позицию и увеличивает позицию в потоке на 1. Имейте в виду, что реализация по умолчанию создаёт новый однобайтовый массив и вызывает метод **Write**, что не является эффективным решением. При переопределении потоки с буфером должны предоставлять более эффективную реализацию, избегающую дополнительное создание массива.

public void CopyTo (Stream destination)

public void CopyTo (Stream destination, Int32 bufferSize) – считывает байты с текущей позиции данного потока и копирует их в целевой поток. Можно указать размер буфера (по умолчанию равен 81920) в качестве 2 параметра. Позиция целевого потока не сбрасывается после копирования.

*Методы **Dispose( )** и **Dispose(bool disposing)**, описаны ранее.

Класс FileStream

FileStream – класс-наследник Stream для работы с файлами, позволяющий осуществлять операции файлового ввода-вывода: открытие, закрытие, чтение, запись и т. д. Файлы могут поддерживать позиционирование.

Зачастую при работе с FileStream будут полезны средства классов File, Directory или Path. В классе File содержится набор статических методов, в результате работы возвращающих ссылки на создаваемые объекты типа FileStream.

При работе с файлами допускается доступ к одному файлу со стороны нескольких потоков/процессов. По этой причине, .NET проверяет возможные изменения, в случае с Read заново считывает поток из файла, предварительно очистив буфер, а Write очистит буфер и выбросит **IOException**.

В конструкторе FileStream вы можете использовать перечисления **FileMode**, **FileAccess**, **FileShare**, **FileOptions**, **FileSystemRights** и класс **FileSecurity** для определения свойств файла и параметров доступа к нему.

FileSystemRights: https://docs.microsoft.com/ru-ru/dotnet/api/system.security.accesscontrol.filesystemrights?view=net-5.0

FileSecurity: https://docs.microsoft.com/ru-ru/dotnet/api/system.security.accesscontrol.filesecurity?view=net-5.0

Перечисление FileMode

Константа	Описание
CreateNew = 1	Указывает, что операционная система должна создавать новый файл. Для этого требуется разрешение Write. Если файл уже существует, создается исключение **IOException**.
Create = 2	Указывает, что операционная система должна создавать новый файл. Если файл уже существует, он будет перезаписан. Для этого требуется разрешение Write. Значение FileMode.Create эквивалентно требованию использовать значение CreateNew, если файл не существует, и значение Truncate в противном случае. Если файл уже существует, но является скрытым, создается исключение **UnauthorizedAccessException**.
Open = 3	Указывает, что операционная система должна открыть существующий файл. Возможность открыть данный файл зависит от значения, задаваемого перечислением FileAccess. Исключение **FileNotFoundException** создается, если файл не существует.
OpenOrCreate = 4	Указывает, что операционная система должна открыть файл, если он существует, в противном случае должен быть создан новый файл. Если файл открыт с помощью FileAccess.Read, требуется разрешение Read. Если доступ к файлу является FileAccess.Write, требуется разрешение Write. Если файл открыт с помощью FileAccess.ReadWrite, требуются разрешения Read и Write.
Truncate = 5	Указывает, что операционная система должна открыть существующий файл. Если файл открыт, он должен быть усечен таким образом, чтобы его размер стал равен нулю байтов. Для этого требуется разрешение Write. Попытки выполнить чтение из файла, открытого с помощью FileMode.Truncate, вызывают исключение **ArgumentException**.
Append = 6	Открывает файл, если он существует, и находит конец файла; либо создает новый файл. Для этого требуется разрешение Append. FileMode.Append можно использовать только вместе с FileAccess.Write. Попытка поиска положения перед концом файла вызывает исключение **IOException**, и любая попытка чтения заканчивается неудачей, и создает исключение **NotSupportedException**.

* Данное перечисление задаёт, каким образом операционная система должна открыть файл.

Перечисление FileAccess

Константа	Описание
Read = 1	Доступ для чтения файла. Данные можно прочитать из файла. Для получения доступа для чтения и записи необходимо объединить с Write.
Write = 2	Доступ для записи в файл. Данные можно записать в файл. Для получения доступа для чтения и записи необходимо объединить с Read.
ReadWrite = 3	Доступ для чтения и записи файла. Данные можно записать в файл и прочитать из файла.

* Обеспечивает доступ на чтение-запись файла. Помечено атрибутом [Flags].

Перечисление FileShare

Константа	Описание
None = 0	Отклоняет общий доступ к текущему файлу. Любой запрос на открытие файла (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт.
Read = 1	Разрешает последующее открытие файла для чтения. Если этот флаг не задан, любой запрос на открытие файла для чтения (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт. Однако, даже если этот флаг задан, для доступа к данному файлу могут потребоваться дополнительные разрешения.
Write = 2	Разрешает последующее открытие файла для записи. Если этот флаг не задан, любой запрос на открытие файла для записи (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт. Однако, даже если этот флаг задан, для доступа к данному файлу могут потребоваться дополнительные разрешения.
ReadWrite = 3	Разрешает последующее открытие файла для чтения или записи. Если этот флаг не задан, любой запрос на открытие файла для записи или чтения (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт. Однако, даже если этот флаг задан, для доступа к данному файлу могут потребоваться дополнительные разрешения.
Delete = 4	Разрешает последующее удаление файла.
Inheritable = 16	Разрешает наследование дескриптора файла дочерними процессами. В Win32 непосредственная поддержка этого свойства не обеспечена.

* Позволяет задать права доступа другим объектам FileStream к этому же файлу. Помечено атрибутом [Flags].

Перечисление FileOptions

Константа	Описание
None = 0	Указывает, что при создании объекта FileStream не должны использоваться дополнительные параметры.
Encrypted = 16384	Указывает, что файл является зашифрованным и может быть расшифрован только с использованием той же учетной записи пользователя, которая применялась для шифрования.
SequentialScan = 134217728	Указывает, что доступ к файлу осуществляется последовательно от начала к концу. Система может использовать это в качестве указания для оптимизации кэширования файлов. Если в приложении указатель позиции в файле перемещается для произвольного доступа, оптимального кэширования можно не достигнуть, однако правильная работа гарантируется. Указание этого флага в некоторых случаях может повысить производительность.
RandomAccess = 268435456	Указывается, что доступ к файлу осуществляется произвольно. Система может использовать это в качестве указания для оптимизации кэширования файлов.
DeleteOnClose = 67108864	Указывает, что файл автоматически удаляется, если он не будет больше использоваться.
Asynchronous = 1073741824	Указывает, что файл может использоваться для асинхронного чтения и записи.
WriteThrough = -2147483648	Указывает, что запись в системе должна выполняться через любой промежуточный кэш и переходить напрямую на диск.

* Данное перечисление позволяет определять специальные файлы. Помечено атрибутом [Flags].

Методы File, Возвращающие FileStream

Как уже известно, **статический класс File** содержит методы, открывающие потоки:
public static FileStream Create (string path)
public static FileStream Create (string path, int bufferSize)
public static FileStream Create (string path, int bufferSize, FileOptions options) – создаёт или перезаписывает файл по указанному пути, если это возможно, в противном случае выбрасывает различные исключения. bufferSize позволяет указать число байт, буферизируемых при чтении или записи в файл.

public static FileStream Open (string path, FileMode mode)
public static FileStream Open (string path, FileMode mode, FileAccess access)
public static FileStream Open (string path, FileMode mode, FileAccess access, FileShare share) – открывает файл для чтения, если это возможно, в противном случае выбрасывает различные исключения. FileMode задаёт режим доступа для возвращаемого потока, FileAccess даёт доступ на чтение/запись, FileShare задаёт параметры совместного доступа для других потоков/процессов.

public static FileStream OpenRead (string path) – эквивалент вызова конструктора FileStream(String, FileMode, FileAccess, FileShare) с FileMode.Open, FileAccess.Read и FileShare.Read, т. е. с доступом только для чтения при совместном доступе.

public static FileStream OpenWrite (string path) – эквивалент вызова конструктора FileStream(String, FileMode, FileAccess, FileShare) с FileMode.OpenOrCreate, FileAccess.Write, FileShare.None, т. е. с доступом только на запись только данным потоком.

public FileStream (string path, FileMode mode)

public FileStream (string path, FileMode mode, FileAccess access)

public FileStream (string path, FileMode mode, FileAccess access, FileShare share)

public FileStream (string path, FileMode mode, FileAccess access, FileShare share, int bufferSize)

public FileStream (string path, FileMode mode, FileAccess access, FileShare share, int bufferSize, FileOptions options)

Конструкторы FileStream

Конструктор позволяет инициализировать новый экземпляр FileStream по абсолютному или относительному пути с указанными режимом создания файла. Перегрузки дают возможность дополнительно указать режим на чтение/запись, разрешение на совместное использование, положительные размер буфера (буфер по умолчанию – 4096), а также дополнительные параметры файла.

Важно понимать, что для корректного завершения работы необходимо в начале вызвать **Flush( )** (при наличии буфера, который нужно освободить), затем **Close( )**, после чего вы сможете обнулить ссылку. В противном случае Вы рискуете потерять данные буфера, которые могут сброситься.

Прочие Члены Класса FileStream

Кроме реализации методов Stream, класс FileStream вводит ещё несколько дополнительных членов:

string Name { get; } – возвращает полный путь файла, использующим FileStream.

public virtual void Lock (long position, long length) – запрещает другим процессам доступ к фрагменту файла, начиная с position до length, даже если у них есть соответствующее разрешение на чтение и запись файла.

public virtual void Unlock (long position, long length) – разблокирует ранее заблокированный файл или его часть.

Пример Работы с FileStream

```
static void Main() {
    // Создадим файл output.txt, размер буфера - 4096,
    // получим ссылку на поток через статический метод File.
    FileStream newFileStream = File.Create("output.txt", 4096);
    string input = Console.ReadLine();
    // Посимвольно запишем введённую строку в файл.
    foreach (char symbol in input)
        newFileStream.WriteByte((byte)symbol);

    // Переместимся в начало файла для корректного чтения
    newFileStream.Seek(0, SeekOrigin.Begin);
    int output;
    long outputSymbolCount = 0;
    // Считываем побайтово символы из файла, пока метод не
    // вернёт -1 (т. е. пока не достигнут конец файла).
    while ((output = newFileStream.ReadByte()) != -1)
    {
        ++outputSymbolCount;
        // Приводим тип к char при выводе.
        Console.WriteLine($"Symbol {outputSymbolCount} is {(char)output}");
    }
    Console.WriteLine($"Количество символов в файле: {outputSymbolCount}");
    // Очистка буфера
    newFileStream.Flush();
    // Закрытие потока
    newFileStream.Close();
    // Обнуление ссылки
    newFileStream = null;
}
```

**без обработки исключений*

Как уже известно, для корректного завершения работы с потоком данных следует вызвать Close( ) (Flush( ) опционален, может быть использован отнюдь не во всех ситуациях). Тем не менее, в процессе работы могут возникнуть исключения. В примере на прошлом слайде при таком сценарии поток не будет корректно закрыт.

На помощь в данной ситуации приходит конструкция try-finally, которая позволяет в блоке finally закрыть поток даже в случае возникновения исключения.

Оператор using позволяет укоротить запись, избавляя программиста от постоянной необходимости писать try-finally (по сути, компилятор переводит инструкцию using именно как try-finally). Вы можете использовать using с любым типом, реализующим интерфейс IDisposable или IAsyncDisposable (начиная с C# 8.0).

Вы можете использовать оператор using так:

```
using (FileStream fs = new FileStream("example.txt", FileMode.Create)) {
    // Код с использованием потока
} // fs очищена и уже недоступна с этого места
```

Оператор using

C# также допускает объявление нескольких локальных переменных внутри одного блока using через запятую, вложенные блоки using, а также передачу уже созданного IDisposable объекта внутрь блока using. Делать так не рекомендуется, потому что после Dispose у Вас всё равно будет возможность обратиться к данному объекту, что чревато исключением.

Начиная с версии C# 8.0, Вы можете использовать using-объявления локальных переменных. Такие объявления работают аналогично обычным блокам с using, однако предоставляют ещё более короткий синтаксис. При объявлении using-переменной Вы обязаны сразу проинициализировать её, в противном случае возникнет ошибка компиляции. Вы также можете объявить несколько using-переменных через запятую.

```
static void Main() {
    // using-объявление двух файловых потоков
    using FileStream fileStream1 = new FileStream("file1.txt", FileMode.Create),
        fileStream2 = new FileStream("file2.txt", FileMode.Create);
    // Заполнение первого потока символами от 0 до 9
    for (int i = '0'; i <= '9'; i++)
        fileStream1.WriteByte((byte)i);

    // Сброс позиции в потоке на начало
    fileStream1.Seek(0, SeekOrigin.Begin);
    // Чтение 10 символов из потока в буфер
    byte[] result = new byte[10];
    fileStream1.Read(result, 0, 10);
    // Запись во второй файл только чётных
    for (int i = 0; i < result.Length; i++)
        if (result[i] % 2 == 0)
            fileStream2.WriteByte(result[i]);
    // 1 файл: 0123456789
    // 2 файл: 02468
}
```

Класс MemoryStream

MemoryStream – поток данных, хранилищем для которого является память, создающийся на основе массива байтов.

public MemoryStream( )

public MemoryStream(byte[] buffer)

public MemoryStream(byte[] buffer, bool writable)

public MemoryStream(byte[] buffer, int index, int count, bool writable, bool publiclyVisible) – инициализирует экземпляр на основе указанного массива байтов, начиная с индекса index длиной count. Если writable = true, поток доступен для записи, publiclyVisible = true позволяет пользоваться методом GetBuffer( ).

public virtual int Capacity { get; set; } – свойство, определяющее ёмкость массива байтов потока. Данному свойству не может быть присвоено значение меньше текущей длины потока, это приведёт к **ArgumentOutOfRangeException**.

public virtual byte[] GetBuffer( ) – возвращает массив байтов, использовавшийся для создания потока. Если MemoryStream создавался с закрытым буфером, попытка вызова этого метода приведёт к **UnauthorizedAccessException**.

Класс BitConverter

BitConverter – специальный статический класс, предназначенный для преобразования простых типов в массивы байтов и наоборот. Является полезным инструментом для получения массивов байтов для записи в потоки.

public static readonly bool IsLittleEndian позволяет определить порядок следования байтов в архитектуре данного компьютера. Если порядок идёт от младшего байта к старшему, значение будет равным true, иначе false.

public static byte[] GetBytes(<Встроенный Тип C#> value) – метод, получающий массив байтов из значения простого типа. Например, передача значения типа int массив из 4 байтов. Порядок байтов зависит от того, имеет ли архитектура прямой или обратный порядок.

static double ToDouble(byte[] value, int startIndex), ToSingle( ), ToBoolean( ), ToChar( ), ToInt16( ), ToInt32( ), ToInt64( ), ToUint16( ), ToUint32( ), ToUint64( ) – набор методов для преобразования массива байтов к простым типам, начиная с указанной позиции в массиве.

public static string ToString (byte[] value) – переводит указанный массив байтов в строку из шестнадцатеричных чисел с разделителем «-». В перегрузке можно указать начальный индекс и количество элементов.

Работа с Бинарным Файлом. Пример

Вы можете пользоваться классом BitConverter для чтения и записи данных в бинарные файлы. Так как метод Read читает данные из потока в буфер типа byte[ ], после чтения Вы можете сразу привести полученный массив к нужному типу:

```
static void Main()
{
    // Создаём бинарный файл
    using FileStream binFileStream = new FileStream("binaryexample.bin", FileMode.Create);

    int example = 97;
    byte[] byteArr = BitConverter.GetBytes(example);
    // Записываем значения в бинарный файл
    for (int i = 0; i < 3; i++)
    {
        byteArr = BitConverter.GetBytes(example + i);
        binFileStream.Write(byteArr, 0, sizeof(int));
    }

    binFileStream.Seek(0, SeekOrigin.Begin);
    int result;
    // Считываем из файла, пока он не пуст и конвертируем буфер в int и char.
    while ((result = binFileStream.Read(byteArr, 0, sizeof(int))) != 0)
    {
        Console.WriteLine(BitConverter.ToInt32(byteArr));
        Console.WriteLine(BitConverter.ToChar(byteArr));
    }
}
```

Наследники Класса Encoding

При чтении бинарных файлов крайне важная задача – сохранить изначальную кодировку при восстановлении файла. Для этой задачи могут пригодиться наследники абстрактного класса **Encoding**.

public virtual byte[] GetBytes (char[] chars)

public virtual byte[] GetBytes (char[] chars, int index, int count)

public virtual byte[] GetBytes (string s) – некоторые из перегрузок, позволяющие закодировать массив символов или строку в массив байтов.

public virtual char[] GetChars (byte[] bytes)

public virtual char[] GetChars (byte[] bytes, int index, int count) – некоторые из перегрузок, позволяющие декодировать массив байтов в массив символов.

public virtual string GetString (byte[] bytes)

public virtual string GetString (byte[] bytes, int index, int count) – некоторые из перегрузок, позволяющие декодировать массив байтов как строку.

Подробнее об Encoding: https://docs.microsoft.com/ru-ru/dotnet/api/system.text.encoding?view=net-5.0

Задачи

Задание 1

В результате выполнения фрагмента программы:

```
using System;
using System.IO;

class Program {
    static void Main() {
        FileStream fs = new FileStream("file.txt",
FileMode.OpenOrCreate);
        fs.Write(new byte[] { 2, 5, 7, 10, 3 }, 2, 2);
        fs.Position = 0;
        Console.Write(fs.ReadByte());
    }
}
```

на экран будет выведено (файл file.txt до запуска программы не существовал):

Примечание:

*Если возникнет ошибка компиляции, введите: ****

Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---

Если возникнет ошибка исполнения или исключение, введите: +++

Задание 2

В результате выполнения фрагмента программы:

```
using System;
using System.IO;

class Program {
    static void Main() {
        FileStream fs = new FileStream("file.txt",
FileMode.OpenOrCreate);
        fs.Write(new byte[] { 3, 4, 7, 12, 18 }, 1, 4);
        fs.Position = 0;
        Console.Write(fs.ReadByte());
        Console.Write(fs.ReadByte());
        for (byte i = 0; i < 5; i++) {
            fs.Position--;
            fs.WriteByte((byte)(i * 2));
            Console.Write(fs.ReadByte());
        }
    }
}
```

на экран будет выведено (файл file.txt до запуска программы не существовал):

Примечание:

*Если возникнет ошибка компиляции, введите: ****

Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---

Если возникнет ошибка исполнения или исключение, введите: +++

Задачи

Задание 3

Выберите верные утверждения (*укажите все верные ответы*):

1) Использование блока `using` для файловых потоков позволяет не использовать явно метод `Dispose()`.
2) Для подключения определённого класса в директиве `using` требуется указывать ключевое слово `static`.
3) В объявлении блока `using` можно использовать любой тип.
4) В случае, если два подключённых пространства имён содержат в себе одноимённый класс, компилятор будет использовать класс из ранее объявленного пространства имён.
5) Возможно делать вложенные блоки `using` .

Задание 4

Статическими классами являются (*укажите все верные ответы*):

1) `Stream;`
2) `FileInfo;`
3) `FileStream;`
4) `File;`
5) `MemoryStream;`

Задание 5

Членами перечисления FileAccess являются (*укажите все верные ответы*):

1) `Read;`
2) `Write;`
3) `Execute;`
4) `WriteRead;`
5) `ReadWrite;`

Ответы

Ответ 1: **7**

Ответ 2: **471218-1-1-1**

Ответ 3: **125**

Ответ 4: **4**

Ответ 5: **125**