IT-Academy

ОБРАЗОВАТЕЛЬНЫЙ ЦЕНТР
ПРОГРАММИРОВАНИЯ И ВЫСОКИХ ТЕХНОЛОГИЙ

2020г.

Работа с файловой системой

- ✓ Понятие потока данных
- ✓ Стандартные типы потоков
- ✓ Тип File
- ✓ Основы XML
- ✓ Основы JSON
- ✓ Сериализация и десериализация данных: бинарная, XML&JSON

Поток данных

Поток данных – это абстрактное представление данных в виде последовательности байт.

Поток данных – это последовательность байтов, которую можно использовать для записи или чтения из вспомогательного запоминающего устройства, являющегося одним из устройств хранения информации (например, дисков или памяти). Есть несколько видов запоминающих устройств, отличных от дисков, и существует несколько видов потоков, помимо файловых потоков, например: сетевые потоки, потоки памяти и потоки каналов.

Стандартные типы потоков

1. Абстрактный класс **System.IO.Stream** – базовый класс для других классов, представляющих потоки.

2. Классы для работы с потоками, связанными с хранилищами:

FileStream – класс для работы с файлами, как с потоками (пространство имён **System.IO**).

MemoryStream – класс для представления потока в памяти (пространство имён **System.IO**).

NetworkStream – работа с сокетами, как с потокам (пространство имён **System.Net.Sockets**).

PipeStream – абстрактный класс из пространства имён **System.IO.Pipes**, базовый для классов-потоков, которые позволяют передавать данные между процессами системы.

Стандартные типы потоков

3. Декораторы потоков.

Декорируют другой поток, преобразуя данные тем или иным образом:

DeflateStream и **GZipStream** – классы (пространство имён **System.IO.Compression**) для потоков со сжатием данных.

CryptoStream – поток зашифрованных данных (пространство имён **System.Security.Cryptography**).

BufferedStream – поток с поддержкой буферизации данных (пространство имён **System.IO**).

4. Адаптеры потоков.

Преобразуют информацию определённого формата в набор байт:

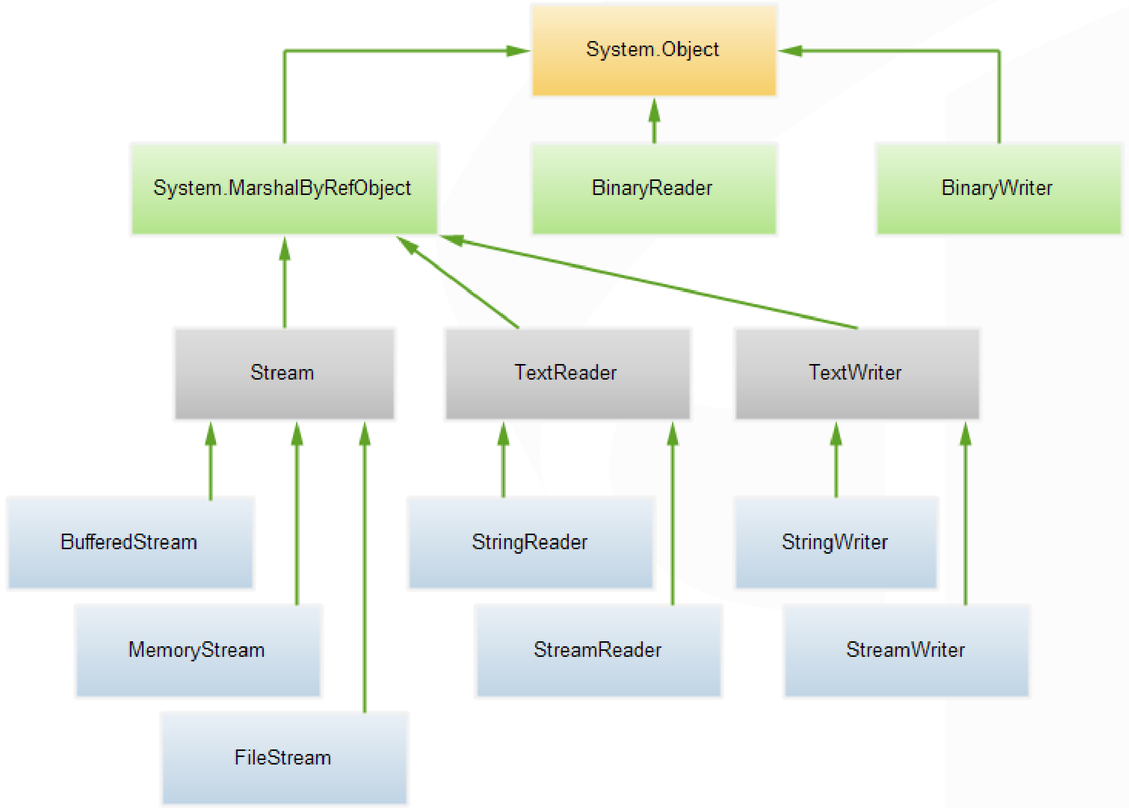
BinaryReader и **BinaryWriter** – классы для ввода/вывода примитивных типов в двоичном формате.

StreamReader и **StreamWriter** – классы для ввода/вывода информации в строковом представлении.

XmlReader и **XmlWriter** – абстрактные классы для ввода/вывода **XML**.

Стандартные типы потоков

Абстрактные классы **TextReader** и **TextWriter** дают возможность читать и записывать данные в поток в строковом представлении. От этих классов наследуются классы **StreamReader** и **StreamWriter**.



System.IO.Stream

В абстрактном классе **System.IO.Stream** определен набор членов, которые обеспечивают поддержку синхронного и асинхронного взаимодействия с хранилищем (например, файлом или областью памяти).

Метод, свойство	Описание
CanRead, CanWrite, CanSeek	Определяют, поддерживает ли текущий поток чтение, поиск и/или запись.
Close()	Закрывает текущий поток и освобождает все ресурсы (такие как сокеты и файловые дескрипторы), ассоциированные с текущим потоком.
Flush()	Обновляет лежащий в основе источник данных или репозиторий текущим состоянием буфера с последующей очисткой буфера. Если поток не реализует буфер, метод не делает ничего.

System.IO.Stream

Метод, свойство	Описание
Length	Возвращает длину потока в байтах.
Position	Определяет текущую позицию в потоке.
Read(), ReadByte()	Читает последовательность байт (или одиночный байт) из текущего потока и перемещает текущую позицию потока на количество прочитанных байтов.
Seek()	Устанавливает позицию в текущем потоке.
SetLength()	Устанавливает длину текущего потока.
Write(), WriteByte()	Пишет последовательность байт (или одиночный байт) в текущий поток и перемещает текущую позицию на количество записанных байтов.

Объекты файловой системы (1. Directory)

Классы для работы с объектами файловой системы – дисками, каталогами, файлами содержатся в пространстве имен **System.IO**

DriveInfo – предоставляет информацию о диске.

Directory, **File**, **DirectoryInfo** и **FileInfo** – предназначены для работы с каталогами и файлами.

```
string path = @"C:\SomeDir";
DirectoryInfo dirInfo = new DirectoryInfo(path);
if (!dirInfo.Exists){
    dirInfo.Create();
}

var file = new FileInfo($"{path}\note.txt");
FileStream fs = file.Open(FileMode.OpenOrCreate,
   FileAccess.ReadWrite, FileShare.None);
```

Тип FileInfo

Класс **FileInfo** позволяет получать подробности относительно существующих файлов на жестком диске (т.е. время создания, размер и атрибуты) и предназначен для создания, копирования, перемещения и удаления файлов. Вдобавок к набору функциональности, унаследованной от **FileSystemInfo**, есть некоторые члены, уникальные для класса **FileInfo**, которые описаны ниже:

Метод, свойство	Описание
AppendText()	Создает объект **StreamWriter** и добавляет текст в файл.
CopyTo()	Копирует существующий файл в новый файл.
Create()	Создает новый файл и возвращает объект **FileStream** для взаимодействия с вновь созданным файлом.
CreateText()	Создает объект **StreamWriter**, записывающий новый текстовый файл.
Delete()	Удаляет файл, к которому привязан экземпляр **FileInfo**.

Тип FileInfo (2. FileInfo)

Метод, свойство	Описание
Directory	Получает экземпляр родительского каталога.
DirectoryName	Получает полный путь к родительскому каталогу.
Length	Получает размер текущего файла или каталога.
MoveTo()	Перемещает указанный файл в новое местоположение, предоставляя возможность указать новое имя файла.
Open()	Открывает файл с различными привилегиями чтения/записи и совместного доступа.
OpenRead()	Создает доступный только для чтения объект **FileStream**.
OpenText()	Создает объект **StreamReader** и читает из существующего файла.
OpenWrite()	Создает доступный только для записи объект **FileStream**.

Тип File (3. File)

Тип **File** предоставляет функциональность, почти идентичную типу FileInfo, с помощью статических методов. Тип **File** также поддерживает несколько уникальных методов:

Метод	Описание
ReadAllBytes()	Открывает файл, возвращает данные в виде массива байт и закрывает файл.
ReadAllLines()	Открывает файл, возвращает данные в виде массива строк и закрывает файл.
ReadAllText()	Открывает файл, возвращает данные в виде строки и закрывает файл.
WriteAllBytes()	Открывает файл, записывает в него массив байт и закрывает файл.
WriteAllLines()	Открывает файл, записывает в него массив строк и закрывает файл.
WriteAllText()	Открывает файл, записывает в него строку и закрывает файл.

Основы XML

XML (англ. **eXtensible Markup Language** — расширяемый язык разметки) — рекомендованный Консорциумом Всемирной паутины (W3C) язык разметки. Спецификация **XML** описывает **XML-документы** и частично описывает поведение **XML-процессоров** (программ, читающих **XML-документы** и обеспечивающих доступ к их содержимому). **XML** разрабатывался как язык с простым формальным синтаксисом, удобный для создания и обработки документов программами и одновременно удобный для чтения и создания документов человеком, с подчёркиванием нацеленности на использование в Интернете.

Язык называется расширяемым, поскольку он не фиксирует разметку, используемую в документах: разработчик волен создать разметку в соответствии с потребностями к конкретной области, будучи ограниченным лишь синтаксическими правилами языка.

До популяризации **JSON**, **XML** являлся одним из самых распространенных стандартов документов, который позволял в удобной форме сохранять сложные по структуре данные. Поэтому разработчики платформы **.NET** включили в фреймворк широкие возможности для работы с **XML**.

Основы XML

Чтобы начать работу с документом xml, нам надо создать объект **XmlDocument** и затем загрузить в него **xml-файл**: **xDoc.Load("users.xml");**

При разборе **XML** для начала мы получаем корневой элемент документа с помощью свойства **xDoc.DocumentElement**. Далее уже происходит собственно разбор узлов документа.

Следовательно у нас есь несколько подходов для работы с XML:

- Чтение документа в память целиком и представление документа в виде иерархии объектов определенных типов.
- Последовательное чтение документа, и анализ каждого прочитанного элемента при помощи классов и их методов из пространстра имен **System.Xml**.

XML-документ объявляет строка **<?xml version="1.0" encoding="utf-8" ?>.** Она задает версию (1.0) и кодировку (utf-8) xml. Далее идет собственно содержимое документа.

Основные классы для работы с XML

Класс	Описание
XmlNode	Представляет узел **xml**. В качестве узла может использоваться весь документ, так и отдельный элемент.
XmlDocument	Представляет весь **xml-**документ.
XmlElement	Представляет отдельный элемент. Наследуется от класса **XmlNode.**
XmlAttribute	Представляет атрибут элемента.
XmlText	Представляет значение элемента в виде текста, то есть тот текст, который находится в элементе между его открывающим и закрывающим тегами.
XmlComment	Представляет комментарий в **xml.**
XmlNodeList	Используется для работы со списком узлов.

Методы и свойства XmlNode

Метод, свойство	Описание
AppendChild()	Добавляет в конец текущего узла новый дочерний узел.
InsertAfter()	Добавляет новый узел после определенного узла.
InsertBefore()	Добавляет новый узел до определенного узла.
RemoveAll()	Удаляет все дочерние узлы текущего узла.
Name	Возвращает название узла.
InnerText	Возвращает текстовое значение узла.
HasChildNodes	Возвращает true, если текущий узел имеет дочерние узлы.
ChildNodes	Возвращает коллекцию дочерних узлов для данного узла.
ParentNode	Возвращает родительский узел у текущего узла.

Основы JSON

JSON — это формат, который хранит структурированную информацию и в основном используется для передачи данных между сервером и клиентом.

Файл **JSON** представляет собой более простую и лёгкую альтернативу расширению с аналогичными функциями **XML**.

Есть два основных элемента объекта **JSON**: ключи и значения.

Ключи должны быть строками. Они содержат последовательность символов, которые заключены в кавычки.

Значения являются допустимым типом данных **JSON**. Они могут быть в форме массива, объекта, строки, логического значения, числа или значения **null**.

Объект **JSON** начинается и заканчивается фигурными скобками **{}**. Внутри может быть одна или больше пар ключей/значений с запятой для их разделения. Между тем за каждым ключом следует двоеточие, чтобы отличить его от значения.

Основы JSON

Основная функциональность по работе с **JSON** сосредоточена в пространстве имен **System.Text.Json.**

Ключевым типом является класс **JsonSerializer**, который и позволяет сериализовать объект в **json** и, наоборот, десериализовать код **json** в объект **C#**.

Для сохранения объекта в **json** в классе **JsonSerializer** определен статический метод **Serialize()**, который имеет ряд перегруженных версий.

Поскольку методы **SerializeAsyc**/**DeserializeAsync** могут принимать поток типа **Stream**, то соответственно мы можем использовать файловый поток для сохранения и последующего извлечения данных.

По умолчанию сериализации подлежат все публичные свойства. Кроме того, в выходном объекте **json** все названия свойств соответствуют названиям свойств объекта **C#**. Однако атрибут **JsonIgnore** позволяет исключить из сериализации определенное свойство. А **JsonPropertyName** позволяет замещать оригинальное название свойства.

Сериализация и десериализация данных

Сериализация – действие, при котором данные объекта в памяти переносятся в поток данных для сохранения или передачи.

Десериализация – обратное действие, заключающееся в восстановлении состояния объекта из потока.

Хотя сериализация представляет собой преобразование объекта в некоторый набор байтов, но в действительности только бинарным форматом она не ограничивается.

Форматы сериализации :

- **Бинарный;**
- **XML;**
- **JSON.**

Для каждого формата предусмотрен свой класс: для сериализации в бинарный формат – класс **BinaryFormatter**, для **xml** - **XmlSerializer**, для **json** - **JsonSerializer**.

Сериализация и десериализация данных

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом **Serializable.**

При сериализации с помощью **JsonSerializer**, можно не использовать арибут **Serializable**, так как **JsonSerializer** использует свой механизм сериализации.

Сериализация применяется к свойствам и полям класса. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом **NonSerialized.**

Объект, который подвергается десериализации, должен иметь конструктор без параметров.

Сериализации подлежат только публичные свойства и поля объекта.

JsonSerializer:

По умолчанию сериализации подлежат все публичные свойства. Кроме того, в выходном объекте **json** все названия свойств соответствуют названиям свойств объекта **C#**. Атрибут **JsonIgnore** позволяет исключить из сериализации определенное свойство. А **JsonPropertyName** позволяет замещать оригинальное название свойства.

Задание

1. Создайте в своей папке 20 дочерних директорий с именами MyTestFolder0 MyTestFolder19. Удалите их программо.
2. Создайте любой файл, запишите в него текст

“Привет с первой строки

Привет с 3й строки”,
закройте файл. Прочитайте файл и покажите на консоль.

3. Создайте класс MyItem с 2мя public свойствами: Age и Name. Создайте объект данного типа. Сериализуйте этот обект в JSON используя Newtonsoft.Json так, чтобы поле Name не попадало в JSON, а значение Age было в поле JSON с именем MyAge.
4. Создать строковую переменную "HI, my name is tist file”. Сохранить ее в файл. Изменить tist на test не перезаписывая сам файл (просто изменить).