**Использование класса Stream**

Класс Stream является абстрактным базовым классом для всех потоков данных в .NET. Потоки представляют собой последовательности байтов, через которые можно читать и записывать данные. Потоки могут быть связаны с файлами, памятью, сетевыми соединениями и другими источниками данных.

Stream предоставляет основные методы для работы с потоками, такие как:

Чтение данных: Read();

Запись данных: Write();

Перемещение по потоку: Seek();

Закрытие потока: Close();

**Основные свойства:**

CanRead: Возвращает true, если поток поддерживает чтение.

CanWrite: Возвращает true, если поток поддерживает запись.

CanSeek: Возвращает true, если поток поддерживает операцию поиска.

Length: Возвращает длину потока.

Position: Позволяет получить или установить текущее положение в потоке.

**Основные методы:**

Read(byte[] buffer, int offset, int count): Читает данные из потока в указанный массив байтов.

Write(byte[] buffer, int offset, int count): Записывает данные в поток из указанного массива байтов.

Seek(long offset, SeekOrigin origin): Устанавливает текущее положение в потоке.

Flush(): Очищает все буферы потока и записывает все данные на устройство.

**Пример использования класса Stream**

Предположим, что нужно прочитать данные из файла и вывести их на экран, а также записать новые данные в файл. **(пример 1)**

**Описание примера 1:**

Запись данных:

В первой части программы создается поток FileStream для записи данных в файл example.txt. Данные "Hello, Stream!" записываются в файл как массив байтов с помощью метода Write().

Чтение данных:

Во второй части программы создается поток FileStream для чтения данных из того же файла. Данные читаются в буфер, затем преобразуются в строку и выводятся на экран.

**Использование класса FileStream для файловых операций**

Класс FileStream в C# предоставляет возможность работы с файлами на уровне байтов. Это производный класс от Stream и используется для чтения и записи данных в файл. FileStream поддерживает работу с файлами синхронно и асинхронно, что делает его универсальным инструментом для работы с файловыми операциями.

Основные операции, которые поддерживает FileStream:

Чтение данных из файла.

Запись данных в файл.

Перемещение указателя чтения/записи в файле.

Очистка буфера (запись на диск).

Основные конструкторы:

FileStream(string path, FileMode mode): Открывает файл по пути path в режиме mode.

FileStream(string path, FileMode mode, FileAccess access): Открывает файл в определённом режиме с указанием доступа (чтение, запись или оба).

FileStream(string path, FileMode mode, FileAccess access, FileShare share): Открывает файл, указывая режим совместного использования файла.

Пример использования класса FileStream **(примеры 2-3).**

Описание примеров:

**Запись данных в файл:**

В примере с записью создаётся поток FileStream, который работает в режиме FileMode.Create. Это значит, что файл будет создан, если его нет, или перезаписан, если уже существует. С помощью метода Write() в файл записывается строка, преобразованная в массив байтов.

**Чтение данных из файла:**

Во втором примере поток FileStream создаётся в режиме FileMode.Open для чтения данных из существующего файла. Прочитанные байты сохраняются в массив и преобразуются обратно в строку с помощью метода Encoding.UTF8.GetString().

Основные параметры FileMode:

FileMode.Create: Создаёт новый файл. Если файл уже существует, то он будет перезаписан.

FileMode.Open: Открывает существующий файл. Если файл не существует, будет вызвано исключение.

FileMode.Append: Открывает файл и добавляет данные в конец файла. Если файл не существует, он будет создан.

FileMode.Truncate: Открывает существующий файл и обрезает его до нулевой длины.

Основные параметры FileAccess:

FileAccess.Read: Открытие файла только для чтения.

FileAccess.Write: Открытие файла только для записи.

FileAccess.ReadWrite: Открытие файла для чтения и записи.

**Пример асинхронных операций с FileStream**

C# также поддерживает асинхронные операции для работы с файлами, что полезно при работе с большими файлами, чтобы не блокировать основной поток. **(пример 4).**

Описание асинхронного примера:

В этом примере используются методы WriteAsync() и ReadAsync() для асинхронной записи и чтения данных из файла. Эти методы не блокируют основной поток во время выполнения операции и позволяют продолжить выполнение других задач.

**Использование класса StreamWriter для файловых операций**

Класс StreamWriter в C# используется для записи текстовых данных в файл или поток. В отличие от FileStream, который работает с байтами, StreamWriter работает с текстом (строками), автоматически преобразуя его в байты с использованием указанной кодировки (по умолчанию — UTF-8). Этот класс удобен для записи текстовой информации, такой как строки, символы или массивы символов.

Основные методы класса StreamWriter:

Write(string value): Записывает строку в поток.

WriteLine(string value): Записывает строку в поток и добавляет символ новой строки (перенос строки).

Flush(): Очищает буфер и записывает данные на диск.

Close(): Закрывает StreamWriter и освобождает ресурсы.

Основные свойства:

AutoFlush: Если установлено в true, буфер автоматически очищается при записи данных, например при использовании метода Write().

Encoding: Получает кодировку, которая используется для записи в поток.

Пример использования класса StreamWriter (пример 5)

Описание примера:

В этом примере создается объект StreamWriter для записи в файл example.txt. Используя метод WriteLine(), записываются строки с добавлением символа новой строки после каждой записи. После завершения работы с файлом объект StreamWriter автоматически закрывается благодаря конструкции using.

**Пример Запись в файл с указанием кодировки (пример 6)**

Описание примера:

Здесь создается объект StreamWriter, но с использованием кодировки UTF-16 (указанной через Encoding.Unicode). Записывается строка в файл с этой кодировкой.

**Асинхронная запись с использованием StreamWriter**

C# также поддерживает асинхронные операции с StreamWriter, что полезно для больших файлов и операций, не блокирующих основной поток.

**Пример асинхронной записи (пример 7)**

Описание примера:

В этом примере используется метод WriteLineAsync() для асинхронной записи строки в файл. Этот метод выполняет запись в файл в асинхронном режиме, что позволяет выполнять другие задачи во время записи данных.

Работа с буфером и автоочисткой

Иногда полезно явно управлять тем, когда данные записываются на диск, используя свойства буферизации и автоочистки:

**Пример с буфером и автоочисткой (пример 8)**

Описание примера:

В этом примере используется свойство AutoFlush, которое автоматически очищает буфер после каждой записи. Это гарантирует, что данные записываются на диск сразу, а не остаются в памяти. Однако можно использовать метод Flush() вручную для того, чтобы явно сбросить буфер и записать данные на диск.

**Использование класса StreamReader для файловых операций**

Класс StreamReader в C# используется для чтения текстовых данных из файлов или потоков. В отличие от FileStream, который работает на уровне байтов, StreamReader предназначен для работы с текстом (строками), используя определённую кодировку (по умолчанию — UTF-8). Это удобный инструмент для чтения текстовой информации построчно или по символам.

Основные методы класса StreamReader:

Read(): Читает один символ из потока и возвращает его как целое число.

ReadLine(): Читает одну строку текста из потока.

ReadToEnd(): Читает весь текст из текущего положения до конца потока.

Peek(): Возвращает следующий доступный символ, не извлекая его из потока.

Close(): Закрывает объект StreamReader и освобождает все системные ресурсы, связанные с ним.

**Пример использования класса StreamReader - Чтение текста из файла построчно (пример 9)**

Описание примера:

В этом примере создается объект StreamReader для чтения данных из файла example.txt. Метод ReadLine() используется для построчного чтения файла. Если файл существует, каждая строка выводится на экран, пока не будут прочитаны все строки.

**Пример - Чтение всего содержимого файла (пример 10)**

Описание примера:

В этом примере используется метод ReadToEnd(), который читает весь текстовый файл целиком и выводит его содержимое. Это полезно, если нужно сразу прочитать все данные, а не построчно.

**Пример Чтение с указанием кодировки – (пример 11)**

Описание примера:

Здесь используется кодировка UTF-16 при чтении файла. Это может быть необходимо, если файл был записан с использованием нестандартной кодировки. Метод ReadToEnd() читает весь файл, который закодирован с использованием указанной кодировки.

**Асинхронное чтение с использованием StreamReader**

Асинхронные методы позволяют выполнять чтение файлов без блокировки основного потока программы, что полезно при работе с большими файлами или в сценариях, где необходимо продолжать выполнение других задач**. Пример асинхронного чтения файла (пример 12)**

Описание примера:

В этом примере используется метод ReadToEndAsync(), который читает файл асинхронно. Асинхронные методы полезны, когда работа с большими файлами не должна блокировать основной поток выполнения программы.

**Пример использования метода Peek (пример 14)**

Метод Peek() позволяет заглянуть в следующий символ, не извлекая его из потока. Это полезно в сценариях, где нужно принять решение о дальнейших действиях в зависимости от следующего символа.

Описание примера:

Здесь используется метод Peek(), чтобы проверить, есть ли следующий символ в потоке. Если символ существует, он читается с помощью метода Read() и выводится на экран. Этот пример читает файл посимвольно.

Преимущества использования StreamReader:

Простота работы с текстом: StreamReader специально создан для работы с текстовыми файлами, что делает его использование удобным.

Поддержка кодировок: StreamReader позволяет работать с разными кодировками (например, UTF-8, UTF-16).

Асинхронные операции: Асинхронные методы позволяют работать с файлами без блокировки основного потока программы.

Гибкость: Можно читать как весь файл целиком, так и построчно или посимвольно, в зависимости от задач.