ПИШНИК: BECOME HUMAN

Глава 15: Паттерны Заместитель, Декоратор и Адаптер. Потоки Данных часть 2.

Автор презентации – Сагалов Даниил БПИ-196

Паттерн Заместитель

Заместитель (Proxy) – структурный паттерн проектирования, позволяющий подставлять вместо одних объектов их заместителей. Они принимают обращения к оригинальному объекту и при этом позволяют сделать что-либо до или после этих обращений.

Ситуация:

Вы – молодой программист, который наконец-то решил попробовать работать с базами данных. Но вот незадача: база данных – объект ресурсоёмкий, обращаться к нему на постоянной основе – явно не самое оптимальное решение. Конечно же, вместо создания такого объекта сразу Вы можете осуществлять его инициализацию только при первом обращении, однако тогда вероятно придётся прописывать логику такой ленивой инициализации для всех клиентов. А в случае работы со сторонней библиотекой у Вас в принципе может не быть доступа к коду класса объекта.

Решение:

Вы можете создать класс с тем же интерфейсом, что и тот объект, причём данный объект-заместитель может кэшировать запросы к нему и в нужный момент лениво инициализировать исходный объект. Причём особенность заместителя состоит в том, что он может быть передан любому коду, ожидающему замещаемый объект благодаря совместимости интерфейсов.

Паттерн Заместитель

Применение Паттерна:

Данный паттерн будет полезен, если:

- У Вас есть некоторый тяжёлый объект, требующий ленивой инициализации и Вам нужно инициализировать его только тогда, когда он реально понадобится.
- Возникает необходимость хранения лога запросов к объекту. Таким образом, при обращении к заместителю он будет сохранять информацию о полученных запросах.
- Вам нужно обезопасить объект от несанкционированного доступа в программе с различными пользователями. Так, например некоторым пользователям будет отказываться в доступе к объекту без соответствующих прав.
- Если необходимо кэшировать запросы к объекту и управлять ими. Объект-заместитель может подсчитывать количество поступающих запросов, после чего он может разом отправлять их к базе данных. Также при отсутствии запросов можно, например, закрыть поток работы с объектом. В дополнение заместитель может отслеживать информацию о клиенте, не давая ему возможность повторно обратиться к серверу.
- Вам нужно локально запустить сервис, находящийся на удалённом сервере. В таком случае заместитель будет транслировать запросы в понятный удалённому серверу формат.

Паттерн Декоратор

Декоратор (Decorator, он же Wrapper) – структурный паттерн проектирования, позволяющий добавлять объектам новую функциональность благодаря помещению их в некоторую обёртку.

Ситуация:

Представьте себе сценарий, при котором Вам необходимо оповещать всех жителей Валгаллы о набегах через мессенджер Odin™. Для этого определён класс **OdinMailing**, содержащий метод **Notify**, отправляющий указанное сообщение всем жителям. Исходный код не предоставляется Вам в целях безопасности, однако Ваша задача – расширить функционал (добавить возможность ограничить круг лиц в рассылке, добавить дополнительные мессенджеры для рассылки). Вы можете наследоваться от класса OdinMailing, определяя объекты-издателей для различных типов мессенджеров/платформ, что позволяет выбрать одну из платформ рассылки. Однако теперь возникает проблема: если викингам понадобится одновременно рассылать оповещения в несколько мессенджеров сразу, для этого понадобится ещё больше классов-наследников.

Хотя наследование в данном случае первым приходит в голову, оно имеет ряд недостатков: невозможно изменить поведение существующего типа (если такая возможность не предоставлялась разработчиком); нельзя наследовать поведение нескольких классов, из-за чего плодятся подклассы, необходимые для совмещения поведения.

Паттерн Декоратор

Решение:

На помощь в данном случае приходит композиция – вложение объекта внутрь типа (в частном случае, в качестве поля) вместо наследования. Таким образом, Вы сможете запускать базовое поведение объекта при определённых условиях или добавлять к нему что-то дополнительно.

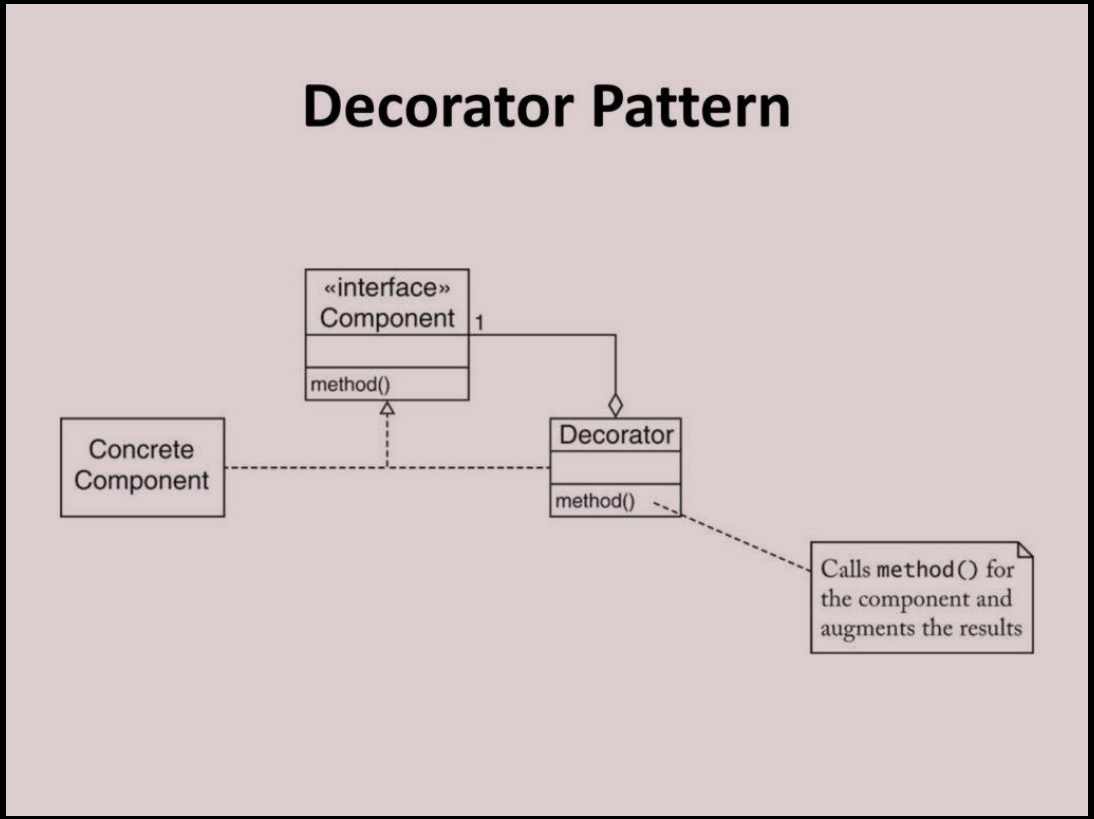
Характерной особенностью при работе с декораторами является возможность добавлять несколько вложенных обёрток. Итоговая обёртка будет совмещать в себе функционал всех предыдущих.

Применение Паттерна:

Данный паттерн будет полезен, если :

- Вы хотите иметь возможность добавлять некий функционал объекту (в виде обёрток) во время выполнения.
- Вы хотите расширить функционал класса, от которого нельзя наследоваться (sealed в C#).
- Наследование неэффективно по причине необходимости создания множества подклассов-комбинаций.

* Недостатками паттерна могут являться необходимость настройки множества обёрток или возникновение обилия мелких классов.



Реализация Паттерна Декоратор

1) Описать интерфейс, содержащий общие методы для данного типа и его обёрток.

2) Разработать конкретный тип, реализующий интерфейс.

3) Создать базовый для декораторов тип, который хранит ссылку на оборачиваемый объект. Методы базового декоратора сводятся к вызову соответствующих методов оборачиваемого объекта. Декоратор реализует тот же интерфейс.

4) Создать конкретные типы декораторов, наследники базового. Конкретные декораторы вызывают как методы обёрнутого объекта, так и вносят свои дополнения.

Реализация Паттерна Декоратор

```
// Интерфейс для типа и его декораторов
Ссылок: 7
public interface IDataSource<T> {
    Ссылок: 3
    public void WriteData(T data) {
        throw new InvalidOperationException("Unable to write data by default.");
    }
    Ссылок: 3
    public T ReadData();
}

// Конкретный оборачиваемый класс, объявлен запечатанным
ссылка: 1
public sealed class File<T> : IDataSource<T> {
    ссылка: 1
    public string FileName { get; private set; }
    Ссылок: 0
    File(string fileName) {
        FileName = fileName;
    }

    Ссылок: 3
    public T ReadData() {
        throw new NotImplementedException("Data reading logic");
    }
    Ссылок: 3
    public void WriteData(T data) { } // запись данных
}
```

```
// Базовый тип декоратора, описывает механизм обёртки
// (не обязательно является абстрактным с виртуальными методами)
Ссылок: 2
public abstract class DataSourceDecorator<T> : IDataSource<T>
    where T : IDataSource<T> {
    protected IDataSource<T> wrappedObj;

    ссылка: 1
    public DataSourceDecorator(IDataSource<T> source) {
        wrappedObj = source;
    }

    Ссылок: 3
    public virtual T ReadData() => wrappedObj.ReadData();
    Ссылок: 3
    public virtual void WriteData(T data) => wrappedObj.WriteData(data);
}

ссылка: 1
public class CompressionDecorator<T> : DataSourceDecorator<T>
    where T : IDataSource<T> {
    // Доп функционал добавляем в классе-обёртке...
    Ссылок: 0
    public CompressionDecorator(IDataSource<T> source) : base(source) { }
}
```

Паттерн Адаптер

Адаптер (Adapter) – структурный паттерн проектирования, позволяющий взаимодействовать объектам с несовместимыми интерфейсами.

Ситуация:

Вы решили стать финансовым аналитиком, собираете и в вашей программе обрабатываете данные из различных XML источников. Внезапно Вам попадается крайне функциональная библиотека для визуализации данных, которая принимает JSON-файлы.

Конечно же, ничего не мешает переписать Вашу библиотеку для работы с JSON, однако представьте сценарий, когда у Вас нет доступа к исходному коду исходной библиотеки для работы с XML.

Решение:

Используйте адаптер – объект, который трансформирует интерфейс таким образом, чтобы он стал совместимым с интерфейсом новой библиотеки. При этом адаптер оборачивает объект таким образом, что другие объекты даже не будут знать об изначальном обёрнутом объекте. При такой логике Вы можете создавать и двусторонние адаптеры.

Заместитель vs. Адаптер vs. Декоратор

На первый взгляд может показаться, что данный паттерны схожи. Действительно, все эти паттерны объединены общей идей создания некоторой обёртки для других классов. Тем не менее, каждый из них имеет уникальные особенности:

- Заместитель предоставляет такой же интерфейс, как и замещаемый им класс и по своей сути не меняет его поведения.
- Декоратор может расширять интерфейс (не обязательно) и при этом расширяет функционал оборачиваемого класса.
- Адаптер предоставляет другой интерфейс для оборачиваемого класса, при этом функционал объекта не меняется.

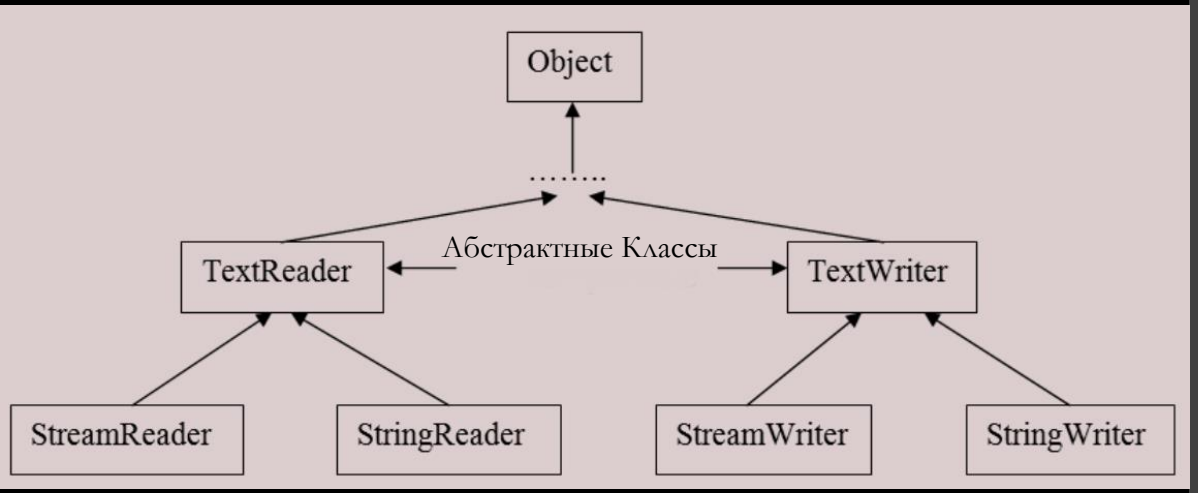
Адаптеры Потоков Данных

Мы разобрались с тем, что такое адаптеры и теперь можем рассмотреть их варианты в System.IO.

TextReader и **TextWriter** – абстрактные классы, реализуют IDisposable, представляют собой средства для считывания наборов символов. От Stream эти классы не наследуются.

StreamWriter, **StreamReader** и **StringWriter, StringReader** – производные классы, предоставляющие средства для чтения и записи файлов (StreamWriter/StreamReader) и строк (StringWriter/StringReader). При этом обратите внимание, что во всех случаях они служат обёрткой, варьирующей поведение потоков.

Важно: зачастую, при очистке ресурсов адаптера сразу же будет очищены все обёрнутые им потоки.



BinaryReader и **BinaryWriter** – адаптеры потоков, предназначенные для чтения и записи в двоичный поток предопределённых типов (byte, int, char и т. д.).

Очистка Ресурсов Адаптеров Потоков

При работе с адаптерами стоит учитывать, что при большинстве сценариев закрытие адаптеров может приводить к закрытию лежащих в их основе потоков. При этом важно помнить, что в случае возникновения исключения в конструкторе адаптера внутри блока using всё равно будет вызван Dispose( ), который очистит и сам поток.

В некоторых случаях возникает необходимость сохранить базовый поток после окончания работы с адаптером, для чего можно поступить так:

- По окончанию работы с адаптером вызвать Flush( ) и установить позицию в потоке на начало.
- Использовать перегрузку адаптера, не сбрасывающую базовый поток, если такая доступна (например, она имеется в наличии у конструкторов типов StreamWriter и StreamReader).

Класс TextReader

TextReader – базовый абстрактный класс, определяющий средства для чтения. Метода Flush( ) нет. Содержит:

protected virtual void Dispose(bool disposing) – очищает ресурсы, вызывая Dispose всех объектов, на которые ссылается. Может быть вызван через Dispose( ) или деструктор в соответствии с паттерном IDisposable.

public virtual int Peek( ) – считывает следующий символ без изменения позиции и возвращает его. Если символов далее нет, вернёт -1. Реализация по умолчанию всегда возвращает -1.

public virtual int Read( ) – считывает следующий символ и возвращает его, позиция сдвигается на 1. Если символов далее нет, вернёт -1.

public virtual int Read(char[] buffer, int index, int count) – считывает максимальное количество символов в буфер, в диапазоне [index ; (index + count - 1)]. ReadBlock – блокирующая версия Read. Будет блокироваться пока не считано count символов или считаны все символы источника.

public virtual string ReadLine( ) – считывает строку до конца строки (или потока) и возвращает её или null, если чтение невозможно. Если количество символов превышает Int32.MaxValue, возникнет **OutOfMemoryException**, позиция в потоке увеличится, а буфер будет утерян. При таком сценарии потребуется заново инициализировать TextReader. Кроме того, если начальная позиция потока неизвестна или поток не поддерживает поиск, также придётся повторно инициализировать и сам поток. Аналогично вы можете попытаться прочитать весь источник до конца с помощью метода **public virtual string ReadToEnd( )**.

Класс TextWriter

TextWriter – базовый абстрактный класс, определяющий средства для записи. Содержит:

public abstract System.Text.Encoding Encoding { get; } – возвращает кодировку, используемую данным TextWriter.

public static readonly TextWriter Null – пустой TextWriter, записанные в него данные никуда не сохраняются.

public virtual string NewLine { get; set; } – задаёт или получает признак конца строки для данного TextWriter. По умолчанию имеет значение **\r\n**. При попытке задать значение null используется значение по умолчанию.

protected virtual void Dispose (bool disposing) – очищает ресурсы, вызывая Dispose всех объектов, на которые ссылается. Может быть вызван через Dispose( ) или деструктор в соответствии с паттерном IDisposable.

public virtual void Flush( ) – загружает данные из буфера в источник данных и очищает буфер.

Также содержит набор виртуальных методов без возвращаемого значения **Write/WriteLine** для записи различных простых типов или для любых объектов с помощью вызова их версии ToString( ). Имеет перегрузку, принимающие объекты типа StringBuilder, а также перегрузку, аналогичную String.Format.

Byte Order Mark (BOM)

При работе с файлами возникает такое понятие, как маркер последовательности байтов (англ. Byte Order Mark). BOM – специальный символ кодировки Unicode, вставляющийся в начало файла или потока, который позволяет обозначить, что используется именно Unicode. Если данный спецсимвол встречается не в начале файла/потока, то он никак не будет отображаться. Однако при этом для некоторых браузеров данный символ может интерпретироваться как символ на целую строку, поэтому для пустого символа существует отдельное значение.

Некоторые кодировки и их маркеры:

Кодировка	Шестнадцатеричный Код	Десятичный Код
UTF-8	EF BB BF	239 187 191
UTF-16 (Little-Endian)	FF FE	255 254
UTF-16 (Big-Endian)	FE FF	254 255
UTF-32 (Little-Endian)	FF FE 00 00	255 254 0 0
UTF-32 (Big-Endian)	00 00 FE FF	0 0 254 255

Дополнительно о BOM:
https://ru.wikipedia.org/wiki/Маркер_последовательности_байтов

Класс StreamReader. Синхронизация с Буфером

StreamReader – адаптер потоков, позволяющий читать байтовый поток как набор символов в определённой кодировке. По умолчанию использует кодировку UTF-8, реализует методы TextReader.

public virtual Stream BaseStream { get; } – возвращает ссылку на поток, оборачиваемый данным адаптером. При позиционировании базового потока стоит понимать, что положение внутреннего буфера и базового потока могут не совпадать. Для сброса буфера используйте метод **public void DiscardBufferedData( )**, однако помните, что данный метод негативно влияет на производительность и вызывать его стоит исключительно при крайней необходимости.

```
public static void Main() {
    // Содержит алфавит: abcdefghijklmnopqrstuvwxyz
    string path = @"c:\temp\alphabet.txt";

    using StreamReader sr = new StreamReader(path);
    // Создаём буфер для символов.
    char[] c = null;
    c = new char[15];
    sr.Read(c, 0, c.Length);
    Console.WriteLine("first 15 characters:");
    Console.WriteLine(c);
    // Напечатает "abcdefghijklmno"

    sr.DiscardBufferedData();
    sr.BaseStream.Seek(2, SeekOrigin.Begin);
    Console.WriteLine("\nBack to offset 2 and read to end: ");
    Console.WriteLine(sr.ReadToEnd());
    // Напечатает "cdefghijklmnopqrstuvwxyz"
    // Без DiscardBufferedData напечатает "pqrstuvwxyzcdefghijklmnopqrstuvwxyz"
}
```

Класс StreamReader

public bool EndOfStream { get; } – свойство, возвращающее true, если был достигнут конец потока.

public StreamReader(Stream stream)

public StreamReader(string path)

public StreamReader(Stream stream, Encoding encoding)

public StreamReader(string path, Encoding encoding)

public StreamReader(Stream stream, Encoding encoding = default, bool detectEncodingFromByteOrderMarks = true, int bufferSize = -1, bool leaveOpen = false) – некоторые из конструкторов StreamReader, позволяют проинициализировать StreamReader, открыв новый поток для работы с файлом по заданному пути или с использованием существующего потока. Если значение detectEncodingFromByteOrderMarks = true, конструктор определит кодировку по первым 4 байтам потока, если возможно, иначе будет использована указанная кодировка encoding. Если bufferSize меньше минимального допустимого значения (128 символов), будет использовано минимальное допустимое значение. Параметр leaveOpen = true позволяет оставить базовый поток открытым после очистки ресурсов экземпляра StreamReader.

Класс StreamWriter

StreamWriter – адаптер потоков, позволяющий записывать символы в байтовый поток в определённой кодировке. По умолчанию использует кодировку UTF-8, реализует методы TextWriter. **Свойства:**

public virtual Stream BaseStream { get; } – ссылка на базовый поток.

public virtual bool AutoFlush { get; set; } – позволяет определить, будет ли сбрасываться буфер в основной поток при каждом вызове Write( ). При AutoFlush = true состояние кодировщика сбрасываться не будет, что позволяет корректно кодировать следующие блоки символов.

public override Encoding Encoding { get; } – возвращает кодировку.

public virtual IFormatProvider FormatProvider { get; } – позволяет получить ссылку на объект, определяющий форматирование символов. Может иметь значение null – в таком случае для форматирования используются язык и региональные параметры текущего потока.

public virtual string NewLine { get; set; } – возвращает или задаёт символ перехода на новую строку.

Класс StreamWriter

public StreamWriter (Stream stream)

public StreamWriter (string path)

public StreamWriter (Stream stream, Encoding encoding)

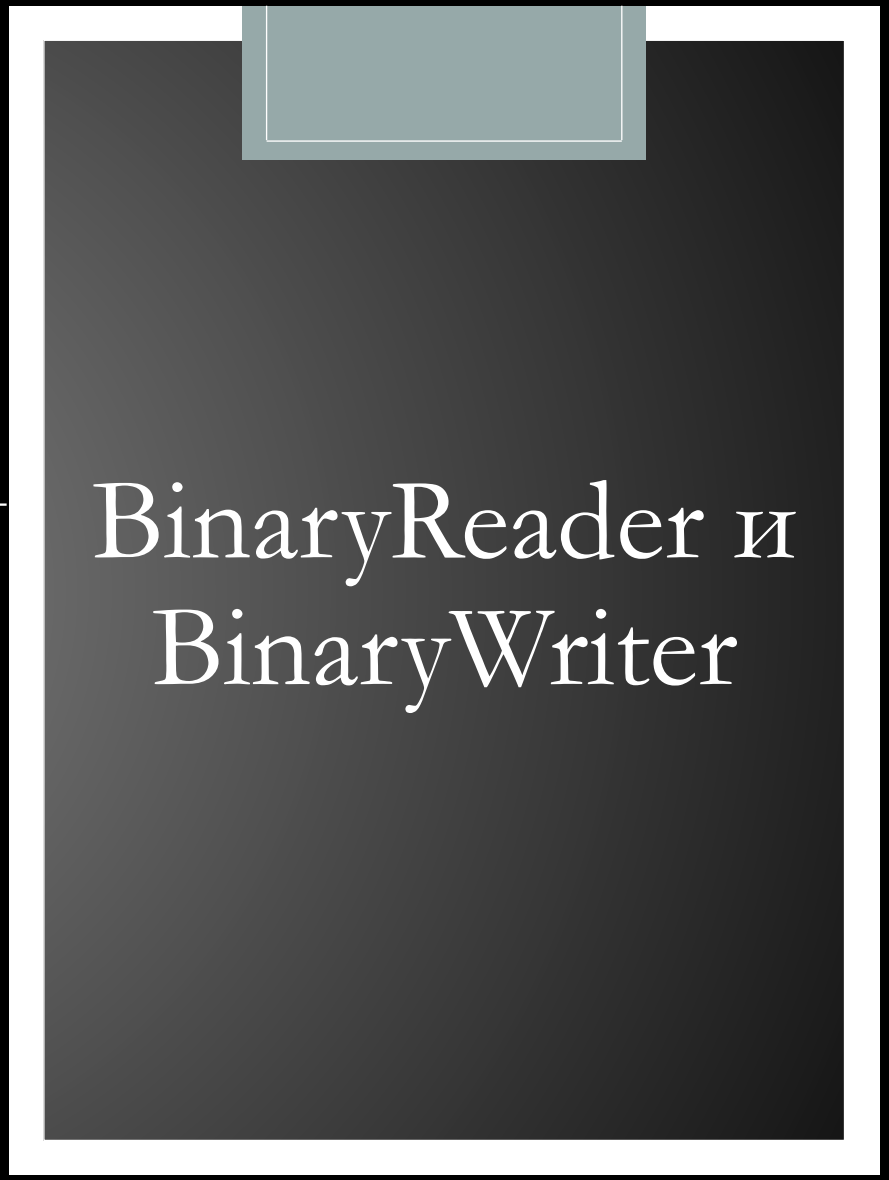
public StreamWriter (Stream stream, Encoding encoding, int bufferSize)

public StreamWriter (string path, bool append, Encoding encoding, int bufferSize)

public StreamWriter (Stream stream, Encoding encoding, int bufferSize, bool leaveOpen) – конструкторы StreamWriter с разными параметрами. Можно использовать как путь-строку, так и ссылку на существующий поток. Вы можете дополнительно указать кодировку encoding, размер буфера int bufferSize. Для некоторых перегрузок, использующих путь используется параметр bool append, который задаёт, будет ли перезаписан файл, если он существует при значении false. Параметр leaveOpen позволяет оставить базовый поток открытым.

```
public static void Main() {
    // Создаём StreamWriter на дополнение доволнение файла temp.txt, если он существует
    // или записи его с нуля в случае отсутствия в кодировке Unicode.
    using (StreamWriter fileWriter = new StreamWriter("temp.txt", true, Encoding.Unicode))
    {
        for (char i = 'a'; i <= 'z'; ++i)
            fileWriter.WriteLine(i);
    }
}
```

.NET предоставляет адаптеры потоков для работы с примитивными типами как с бинарными значениями. Для этого используются классы **BinaryReader** и **BinaryWriter**, реализующие интерфейс IDisposable. Обратите внимение, что данные классе не наследуются ни от Stream, ни от TextReader/TextWriter.



Конструкторы BinaryReader:

public BinaryReader(Stream input)

public BinaryReader(Stream input, Encoding encoding)

public BinaryReader(Stream input, Encoding encoding, bool leaveOpen) – позволяет инициализировать новый экземпляр BinaryReader на основе указанного потока, опционально указав кодировку (по умолчанию – UTF-8). При необходимости можно оставить поток открытым.

Конструкторы BinaryWriter:

protected BinaryWriter( ) – конструктор без параметров для наследников.

public BinaryWriter(Stream output)

public BinaryWriter(Stream output, Encoding encoding)

public BinaryWriter(Stream output, Encoding encoding, bool leaveOpen) – позволяет инициализировать новый экземпляр BinaryWriter на основе указанного потока, опционально указав кодировку (по умолчанию – UTF-8). При необходимости можно оставить поток открытым.

Стандартные Потоки Консоли

На протяжение нашего курса мы часто работаем со статическим классом Console, для которого также актуальны понятия потоков. Дело в том, что при каждом запуске консольного приложения система создаёт для него 3 потока: стандартный поток ввода (TextReader, клавиатура), стандартный поток вывода и стандартный поток вывода ошибок (TextWriter, окно консоли).

Класс консоли предоставляет методы, позволяющие перенаправлять консольные потоки на другие источники данных. При этом все операции ввода-вывода, которые используют данные потоки, будут синхронизированы, т. е. несколько других потоков могут читать-записывать в стандартные консольные. В дополнение, асинхронные методы для потоков консоли выполняются синхронно.

public static TextReader In { get; } – ссылка на стандартный поток ввода.

public static TextWriter Out { get; } – ссылка на стандартный поток вывода.

public static TextWriter Error { get; } – ссылка на стандартный поток вывода ошибок.

public static void SetIn(TextReader newIn) – позволяет задать стандартный поток ввода.

public static void SetOut(TextReader newOut) – позволяет задать стандартный поток вывода.

public static void SetError(TextReader newError) – позволяет задать стандартный поток вывода ошибок.

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);
        using (StreamWriter sw = new StreamWriter(fs)) {
            for (int i = 0; i < 5; i++) {
                sw.Write(i.CompareTo(i * i - i));
            }
            using (StreamReader sr = new StreamReader(fs)) {
                Console.Write(sr.ReadToEnd());
            }
        }
    }
}
```

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);
        using (StreamWriter sw = new StreamWriter(fs)) {
            for (int i = 0; i < 5; i++) {
                sw.Write(i.CompareTo(i * i - i));
            }
        }
        fs = new FileStream("file.txt", FileMode.OpenOrCreate);
        using (StreamReader sr = new StreamReader(fs)) {
            Console.Write(sr.ReadToEnd());
        }
    }
}
```

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

В результате выполнения фрагмента программы:

```
using System;
using System.IO;
class Program {
    static void Main() {
        FileStream fs = new FileStream("int.txt", FileMode.Open);
        using (StreamWriter sw = new StreamWriter(fs)) {
            sw.WriteLine(1);
            sw.WriteLine(2);
            sw.WriteLine(3);
            using (StreamReader sr = new StreamReader(fs)) {
                Console.Write(sr.ReadToEnd());
            }
        }
    }
}
```

могут быть выброшены исключения:

1) UnauthorisedAccessException
2) FileNotFoundException
3) ArgumentException
4) ObjectDisposedException
5) ArgumentNullException

Задание 4

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

1) TextWriter;
2) StringWriter;
3) TextReader;
4) StringReader;
5) Stream;

Задачи

Задание 5

В результате выполнения фрагмента программы:

```
using System;
using System.IO;

class Program {
    static void Main() {
        TextWriter tr = Console.Out;
        Console.SetOut(new StreamWriter("text.txt"));
        for (int i = 0; i < sizeof(long); i += 2) {
            Console.Write(i * i);
            Console.SetOut(tr);
        }
    }
}
```

на экран будет выведено (файл text.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 6

Метод Dispose() реализуют следующие классы (*укажите все верные ответы*)**:**

1) StreamWriter;
2) StringWriter;
3) TextReader;
4) StreamReader;
5) MemoryStream;

Задание 7

В результате выполнения фрагмента программы:

```
using System;
using System.IO;

class Program {
    static void Main() {
        StreamWriter sw = new StreamWriter(new FileStream("tt.txt",
FileMode.OpenOrCreate));
        Console.SetOut(sw);
        for (int i = 0; i < sizeof(bool); i++) {
            Console.WriteLine(i * i);
            Console.SetOut(Console.Out);
        }
    }
}
```

на экран будет выведено (файл tt.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: +++

Ответы

Задание	Ответ
1	**+++** *(ObjectDisposedException)*
2	**010-1-1**
3	**12345**
4	**135**
5	**41636**
6	**12345**
7	**---** *(Консольный вывод перенаправлен в файл)*