Урок №7

Содержание

Сборка мусора

Жизненный цикл объектов

Практически любая программа для своей работы использует различные объекты, работа с которыми приводит к расходованию оперативной памяти. При объектно-ориентированном подходе программирования, каждый тип описывает некий объект, с которым после создания может работать программа.

Жизненный цикл любого объекта можно представить следующим образом:

1) выделение памяти для типа;
2) инициализация выделенной памяти (установка объекта в начальное значение, вызов конструктора);
3) использование объекта в программе;
4) разрушение состояния объекта;
5) освобождение занятой памяти.

Управление жизненным циклом объектов представляет собой утомительную задачу и является постоянным источником ошибок. Поэтому был разработан автоматический механизм управления памятью – сбор мусора. Программирование в среде, поддерживающей автоматическую сборку мусора, значительно упрощает разработку приложений. Программисты, использующие C++ знают, что если забыть вручную удалить размещенные в динамической памяти объекты, то это может привести к «утечке памяти». Недопущение этого явления является одним из самых трудоемких и нудных аспектов программирования на неуправляемых языках. Поручив системе автоматического сбора мусора уничтожение объектов, вы снимаете с себя ответственность за управление памятью и перекладываете его на среду – CLR.

Понятие сборщика мусора

Сбор мусора освобождает программиста от необходимости постоянно следить за использованием и своевременным освобождением памяти. Специальный код, осуществляющий сбор мусора называется «сборщик мусора» (Garbage Collector).

Сборщик мусора отслеживает и уничтожает объекты, находящиеся в управляемой памяти. Периодически сборщик выполняет сборку мусора, чтобы высвободить память, отведенную под объекты, на которые нет действительных ссылок. Сборка мусора автоматически запускается в случае, если требуемый объем памяти больше доступного объема свободной памяти. Кроме того, приложение может принудительно запустить сборку мусора с помощью метода `Collect` класса `System.GC`.

Сборка мусора состоит из следующих шагов:

1. Сборщик мусора осуществляет поиск управляемых объектов, на которые есть ссылки в управляемом коде.
2. Сборщик мусора пытается завершить объекты, на которые нет ссылок.
3. Сборщик мусора освобождает объекты, на которые нет ссылок, и высвобождает выделенную им память.

Во время сборки мусора сборщик не освобождает объект, если он еще используется приложением, т.е. достижим. Объект считается достижимым, если сборщик находит хотя бы одну ссылку на него в управляемом коде.

Деструктор и метод Finalize

Исходя из жизненного цикла объекта, после использования он должен быть разрушен. Сборщик мусора умеет автоматически разрушать простые объекты (такие, как `String`, `Attribute`, `Exception`, `Delegate` и др.) и освобождать занимаемую ими память. Что же касается сложных типов, то их сборщик мусора правильно разрушать не умеет, и поэтому

необходимо писать специальный экземплярный код для правильного уничтожения объекта. К таким типам относятся, в первую очередь, типы, использующие не только оперативную память, а и другие машинные ресурсы, а также типы, которые представляют собой «оболочку» для так называемого «неуправляемого кода» (`System.IO.FileStream`, `System.Net.Socket`, `System.Threading.Mutex` и др.). Для таких типов CLR поддерживает специальный механизм, называемый финализацией объектов (finalization). Для выполнения этого механизма объект должен реализовать одноименный метод (`Finalize`), который будет вызван сборщиком мусора непосредственно перед его уничтожением. Однако непосредственно данный метод перегрузить нельзя – для реализации данной функциональности (завершителя) используется специальный синтаксис. В C# для определения завершителя используется тильда (~) и имя класса.

```
using System;

namespace FinalizeExample
{
    class Program
    {
        static void Main(string[] args)
        {
            MyClass mc = new MyClass();
        }
    }


    class MyClass
    {
        public MyClass()
        {
            Console.WriteLine("Создание объекта");
        }

        ~MyClass()
        {
            // Здесь уничтожаем ресурсы, которые занял данный объект.
            // Данный метод вызывается непосредственно
            // перед уничтожением объекта
            Console.WriteLine("Уничтожение объекта");
        }
    }
}
```

Скомпилировав данный тип в сборку и применив Reflector или IL-DASM, мы обнаружим в метаданных типа `MyClass` описание метода `Finalize()` (см. рис. 1).

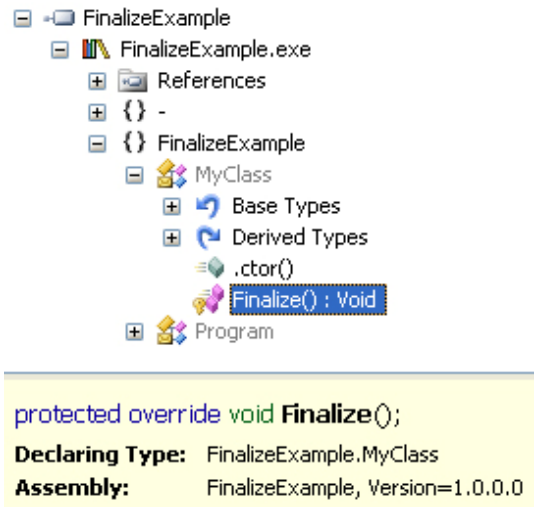

Рис. 1. Отражение метода Finalize() типа MyClass.

Перегружать финализатор и использовать в нем параметры нельзя. Несмотря на то, что данный синтаксис очень похож на деструктор в C++, задачи, которые решает финализация в .NET, отличаются от таковых для детерминированного уничтожения объектов типа посредством деструктора в C++.

Метод Dispose и интерфейс IDisposable

Метод `Finalize()` напрямую вызвать нельзя – его вызывает Garbage Collector при разрушении объекта во время сборки мусора. В некоторых случаях возникает необходимость предоставить возможность явно освободить внешние ресурсы, захваченные объектом, перед тем как сборщик мусора уничтожит объект. Если внешние ресурсы дефицитны или дороги, наилучшей производительности можно достичь при явном освобождении ресурсов, когда они уже не используются. Такое поведение реализуется при помощи интерфейса `IDisposable`, который определяет наличие метода `Dispose()` у своих наследников. Пользователь

объекта должен вызывать этот метод при завершении использования объекта.

Обратите внимание, что даже при наличии явного управления с помощью метода `Dispose` следует предоставить и неявное освобождение ресурсов с использованием метода `Finalize`. Метод `Finalize` предоставляет дополнительный способ предотвращения потери ресурсов в случае, если программист забыл вызывать `Dispose`.

Ниже показан базовый шаблон, рекомендуемый Microsoft для реализации `IDisposable`.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace IDisposableExample
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // Инициализация потока для передачи в
                // DisposableResource
                Console.Write("Введите путь к файлу: ");
                string fileSpec = Console.ReadLine();
                FileStream fs = File.OpenRead(fileSpec);
                DisposableResource testObj = new DisposableResource(fs);

                // Использование ресурса.
                testObj.DoSomethingWithResource();

                // Освобождение ресурса
                testObj.Dispose();

            }
            catch (FileNotFoundException e)
            {
                Console.WriteLine(e.Message);
            }

        }
    }


    /// <summary>
    /// Демонстрационный класс, наследующий
    /// и реализующий интерфейс IDisposable.
    /// </summary>
    public class DisposableResource : IDisposable
    {
```

```
        #region Fields

        private Stream _resource;
        private bool _disposed;

        #endregion

        /// <summary>
        /// Поток, передаваемый в конструктор
        /// должен быть читаем и не null
        /// </summary>
        /// <param name="stream"></param>
        public DisposableResource(Stream stream)
        {
            if (stream == null)
                throw new ArgumentNullException("Поток null.");
            if (!stream.CanRead)
                throw new ArgumentException("Поток должен быть"+
                    "доступен для чтения.");

            this._resource = stream;
            this._disposed = false;
        }

        /// <summary>
        /// Метод демонстрирует использование ресурса.
        /// Ресурс не должен быть освобожден к моменту вызова метода
        /// </summary>
        public void DoSomethingWithResource()
        {
            if (_disposed)
                throw new ObjectDisposedException("Ресурс был освобожден.");
            //Показать количество байт
            int numBytes = (int)_resource.Length;
            Console.WriteLine("Количество байт: {0}", numBytes.ToString());
        }

        #region IDisposable Members
        public void Dispose()
        {
            this.Dispose(true);
            // Необходимо использовать SupressFinalize,
            // чтобы не выполнять финализацию после
            // явного освобождения ресурсов
            GC.SuppressFinalize(this);
        }
        #endregion


        /// <summary>
        /// Финализатор
        /// </summary>
        ~DisposableResource()
        {
            this.Dispose(false);
        }


        protected virtual void Dispose(bool disposing)
        {
            if (!_disposed)
            {
                if (disposing)
```

```
                {
                    if (_resource != null)
                        _resource.Dispose();
                    Console.WriteLine("Ресурс освобожден.");
                }

                // Индикация, что ресурс уже был освобожден
                this._resource = null;
                this._disposed = true;
            }
        }
    }
}
```

Использование данного шаблона позволяет правильно очищать машинные ресурсы как явно, так и посредством автоматического сбора мусора. Код, вызывающий метод `Dispose()` должен быть защищен блоком `try`/`finally`, что будет гарантировать завершение объекта и освобождение ресурсов.

```
namespace IDisposableExample
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // Инициализация потока для передачи в
                // DisposableResource
                Console.Write("Введите путь к файлу: ");
                string fileSpec = Console.ReadLine();
                FileStream fs = File.OpenRead(fileSpec);
                DisposableResource testObj = new DisposableResource(fs);
                try
                {
                    // Использование ресурса.
                    testObj.DoSomethingWithResource();
                }
                finally
                {
                    // Освобождение ресурса
                    if(testObj != null)
                        testObj.Dispose();
                }
            }
            catch (FileNotFoundException e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Учитывая то, что блок try/finally во всех подобных случаях будет практически одинаков, разработчики C# для наследников интерфейса IDisposable предлагают упрощенный синтаксис (с использованием оператора using), который указывает компилятору сгенерировать идентичный код. Таким образом, использование оператора using позволяет упростить синтаксис и одновременно гарантировать правильную очистку ресурсов: как только выполнение кода выйдет за пределы операторных скобок using'а, будут разрушены (вызваны методы Dispose()) все объекты, созданные в его пределах. Следует обратить внимание, что оператор using «умеет» работать только с наследниками интерфейса IDisposable, поэтому применение его для объектов, не реализующих данный интерфейс, бессмысленно.

```
namespace IDisposableExample
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // Инициализация потока для передачи в
                // DisposableResource
                Console.Write("Введите путь к файлу: ");
                string fileSpec = Console.ReadLine();
                FileStream fs = File.OpenRead(fileSpec);
                DisposableResource testObj = new DisposableResource(fs);
                using (DisposableResource testObj = new DisposableResource(fs))
                {
                    testObj.DoSomethingWithResource();
                }
            }
            catch (FileNotFoundException e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Класс System.GC

Microsoft настоятельно не рекомендует вмешиваться в работу сборщика мусора, но, несмотря на это, в .NET Framework такая возможность есть. Класс, представляющий сборщик мусора, называется `System.GC`. Методы этого класса оказывают влияние на то, когда выполняется сборка мусора и когда высвобождаются ресурсы, выделенные объектом. Свойства этого класса предоставляют сведения об общем объеме доступной памяти системы и о том, к какому поколению относится память, выделенная объекту.

Как уже было сказано выше, во время сборки мусора сборщик не освобождает объект, если находит хотя бы одну ссылку на него в управляемом коде. Однако сборщик мусора не распознает ссылки на объект из неуправляемого кода и потому может уничтожать объекты, которые используются исключительно в неуправляемом коде, если явно не предотвратить такое действие. Метод `KeepAlive` предоставляет механизм, который предотвращает уничтожение сборщиком мусора объектов, все еще используемых в неуправляемом коде.

Пример использования основных методов класса `System.GC` показан ниже.

```
using System;

namespace GCExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Демонстрация System.GC");
            Console.WriteLine("Максимальное поколение: {0}", GC.MaxGeneration);

            GarbageHelper hlp = new GarbageHelper();
            // Узнаем поколение, в котором находится объект
            Console.WriteLine("Поколение объекта: {0}", GC.GetGeneration(hlp));
            // Количество занятой памяти
            Console.WriteLine("Занято памяти (байт): {0}",
                GC.GetTotalMemory(false));
            // Создаем мусор
            hlp.MakeGarbage();
            Console.WriteLine("Занято памяти (байт): {0}",
                GC.GetTotalMemory(false));
```

```
            // Вызываем явный сбор мусора в первом поколении
            GC.Collect(0);
            // Количество занятой памяти
            Console.WriteLine("Занято памяти (байт): {0}",
                GC.GetTotalMemory(false));
            // Узнаем поколение, в котором находится объект
            Console.WriteLine("Поколение объекта: {0}",
                GC.GetGeneration(hlp));
            // Вызываем явный сбор мусора во всех поколениях
            GC.Collect();
            // Количество занятой памяти
            Console.WriteLine("Занято памяти (байт): {0}",
                GC.GetTotalMemory(false));
            // Узнаем поколение, в котором находится объект
            Console.WriteLine("Поколение объекта: {0}", GC.GetGeneration(hlp));
            Console.Read();
        }
    }


    /// <summary>
    /// Вспомогательный класс для создания мусора
    /// </summary>
    class GarbageHelper
    {
        /// <summary>
        /// Метод, создающий мусор
        /// </summary>
        public void MakeGarbage()
        {
            for (int i = 0; i < 1000; i++)
            {
                var p = new Person();
            }
        }

        class Person
        {
            private string _name;
            private string _surname;
            private byte _age;

            public Person(string name, string surname, byte age)
            {
                this._age = age;
                this._name = name;
                this._surname = surname;
            }

            public Person()
                : this("", "", 0)
            {
            }
        }
    }
}
```

Понятие поколений при сборке мусора

Процесс сборки мусора сам по себе замедляет работу системы и, чтобы его ускорить, было сделано следующее допущение: изначально считается, что память бесконечна. Это позволяет выделять память для объектов очень быстро, т.к. в этом случае нет необходимости в их фрагментации – память выделяется последовательно от конца предыдущего объекта единым блоком. Но, естественно, данное допущение изначально ложно, поэтому необходим сбор мусора.

Сборщик мусора после очистки памяти, чтобы не допустить фрагментации, «сжимает» ее, другими словами, он «сдвигает» все объекты так, чтобы они располагались последовательно (см. рис. 2). Естественно, что при этом меняется значение ссылок на оставшиеся объекты, поэтому сборщику во всем приложении необходимо обновить ссылки. Во время этого процесса все потоки приложения ожидают обновления ссылок. Другими словами, во время сбора мусора приложение как бы «зависает». Поэтому сбор мусора должен происходить как можно быстрее.

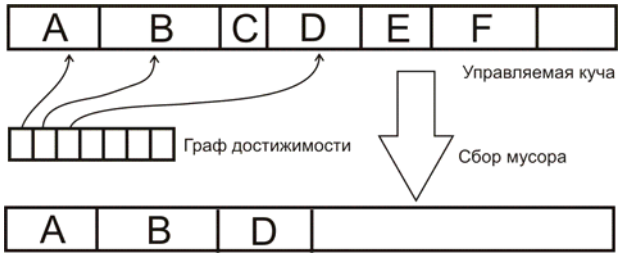

Рис. 2. Схема сжатия памяти при сборке мусора.

Как показали исследования, большая часть объектов, которые созданы недавно, быстро становятся недостижимыми. Поэтому управляемая куча была разделена на 3 поколения: 0, 1 и 2. Размер памяти под каждое из поколений разный – он увеличивается с ростом номера поколения. Все вновь созданные объекты попадают в первое (0) поколение. Сбор мусора автоматически вызывается по окончанию памяти в одном из поколений (при этом сбор мусора происходит не только в этом поколе-

нии, а и во всех младших, но не старших). Объекты, которые пережили сбор мусора, переходят в следующее поколение.

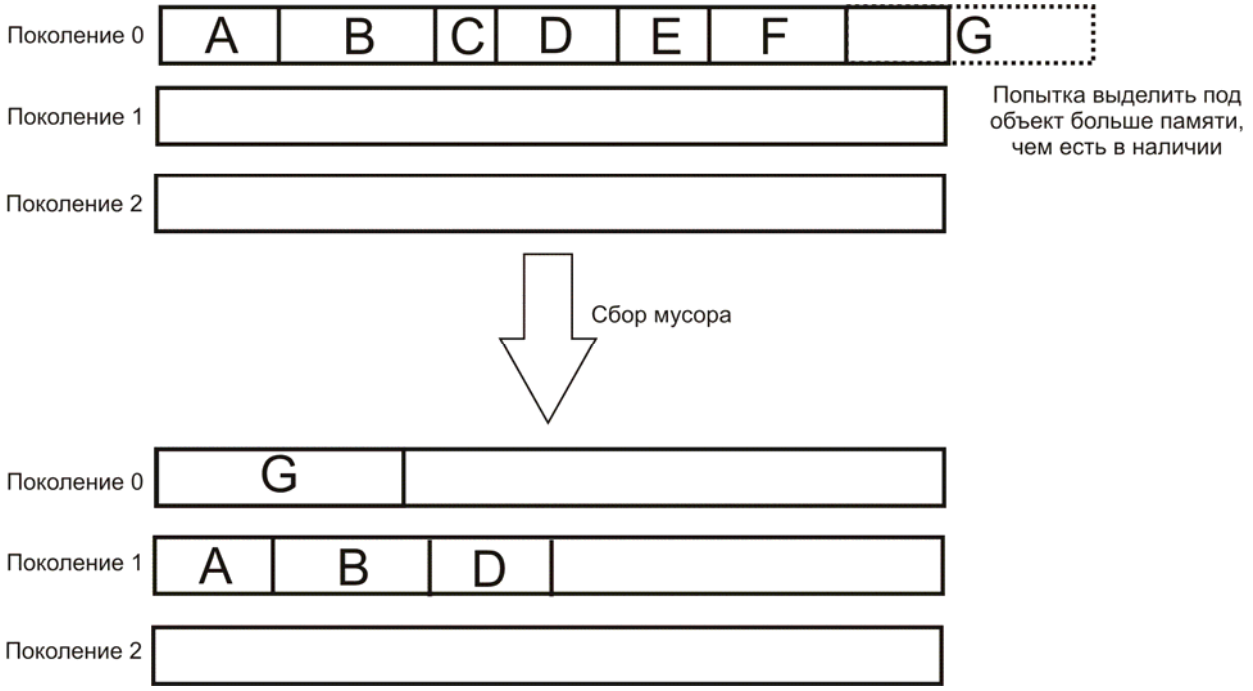

Рис. 3. Организация кучи в виде поколений.

Данный подход позволяет очень быстро очищать память: чем меньше размер поколения, тем быстрее проходят операции по дефрагментации памяти. С другой стороны, чем выше номер поколения, в котором находится объект, тем меньше вероятность его уничтожения сборщиком мусора, даже если на него уже давно нет ссылок в приложении. Автоматический сбор мусора не происходит, пока есть еще свободная память в данном поколении, даже если множество объектов в нем уже недостижимы.

Ниже показан пример, демонстрирующий переход объектов в более высокие поколения, при сборе мусора.

```
using System;

namespace GCGenerationsExample
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            // Создадим несколько объектов класса Garbage
            Garbage g1 = new Garbage();
            Garbage g2 = new Garbage();
            Garbage g3 = new Garbage();
            // Выведем номер поколения,
            // в котором находится каждый из них.
            // После создания все объекты попадают
            // в поколение №0
            Console.WriteLine("Номер поколения g1: {0}",
                GC.GetGeneration(g1));
            Console.WriteLine("Номер поколения g2: {0}",
                GC.GetGeneration(g2));
            Console.WriteLine("Номер поколения g3: {0}",
                GC.GetGeneration(g3));
            // Проведем сбор мусора и больше не будем в коде
            // ссылаться на объект g1, чтобы его уничтожил
            // сборщик мусора
            Console.WriteLine("Количество занятой памяти " +
                "до сбора мусора: {0} байт",
                GC.GetTotalMemory(false));
            // Проводим сбор мусора в поколении 0.
            // Объекты, на которые
            // уже нет ссылок (g1) уничтожаются, а те, на
            // которые еще есть - переходят в
            // более старшее поколение (1)
            GC.Collect(0);
            Console.WriteLine("Количество занятой памяти "+
                "после сбора мусора: {0} байт",
                GC.GetTotalMemory(false));
            // Узнаем номера поколений, в котором находтся
            // объекты g2 и g3
            Console.WriteLine("Номер поколения g2: {0}",
                GC.GetGeneration(g2));
            Console.WriteLine("Номер поколения g3: {0}",
                GC.GetGeneration(g3));
            // Проведем сбор мусора и больше не будем в коде
            // ссылаться на объект g2, чтобы его уничтожил
            // сборщик мусора
            Console.WriteLine("Количество занятой памяти "+
                "до сбора мусора: {0} байт",
                GC.GetTotalMemory(false));
            // Проводим сбор мусора в поколении 0.
            // Все объекты в нем должны быть уничтожены
            GC.Collect(0);
            // Но объект g2 не был уничтожен,
            // ведь он находится в поколении 1. Кроме того,
            // объкт g3 не перешел в старшее поколение:
            Console.WriteLine("Номер поколения g3: {0}",
                GC.GetGeneration(g3));
            // Проведем сбор мусора в поколении 1, сравнивая
            // объем занятой памяти до и после сбора мусора:
            Console.WriteLine("Количество занятой памяти " +
                "до сбора мусора: {0} байт",
                GC.GetTotalMemory(false));
            GC.Collect(1);
            Console.WriteLine("Количество занятой памяти " +
                "после сбора мусора: {0} байт",
                GC.GetTotalMemory(false));
            // Память очистилась - был уничтожен объект g2,
            // а объект g3 перешел в старшее поколение:
```

```
            Console.WriteLine("Номер поколения g3: {0}",
                GC.GetGeneration(g3));
        }
    }


    /// <summary>
    /// Класс, создающий мусор
    /// </summary>
    class Garbage
    {
        private int[] _garbage = null;

        public Garbage()
        {
            // Создаем мусор
            Random rnd = new Random();
            this._garbage = new int[rnd.Next(1000, 10000)];
            for (int i = 0; i < this._garbage.Length; i++)
            {
                this._garbage[i] = rnd.Next();
            }
        }
    }
}
```

Модель потоков в C#. Пространство System.IO

Класс Stream

Практически любые операции, связанные с вводом/выводом любой последовательности байт (файл, устройство ввода-вывода, канал взаимодействия процессов, поток шифрования и др.) в .NET осуществляются с помощью потоков. Базовый класс для всех потоков – `System.IO.Stream`. Класс `Stream` и его производные классы предоставляют универсальное представление различных типов ввода и вывода, изолируя программиста от отдельных сведений операционной системы и базовых устройств.

В зависимости от базового источника или хранилища данных потоки могут поддерживать только некоторые из этих возможностей. Приложение может запрашивать возможности потока с помощью свойств `CanRead`, `CanWrite` и `CanSeek`.

Методы `Read` и `Write` позволяют выполнять чтение и запись данных в различных форматах. Если поток поддерживает поиск, для отправки запросов и изменения текущего положения и длины потока рекомендуется использовать методы `Seek`, `SetLength`, а также свойства `Position`, `Length`.

Некоторые реализации потоков выполняют локальную буферизацию основных данных для улучшения производительности. В таких потоках для удаления внутренних буферов и обеспечения записи всех данных в основной источник данных или хранилище объектов можно использовать метод `Flush`.

При вызове метода `Close` для класса `Stream` все данные буфера будут очищены, по сути выполняется вызов метода `Flush`. Метод `Close` также освобождает такие ресурсы операционной системы, как дескрипторы файлов, сетевые подключения или память, используемую для внутренней буферизации.

Анализ байтовых, символьных и двоичных классов потоков

Многие потоки, которые работают непосредственно с устройствами ввода/вывода, умеют писать/читать только последовательности байт. Естественно, что это не совсем удобно. Поэтому в .Net существуют классы-наследники `Stream`, которые принимают в свой конструктор базовый поток и умеют работать с более сложными типами данных. Другими словами, эти классы способны выступать в виде некой обертки над байтовым потоком, способной преобразовывать записываемые или читаемые данные в массив байт и, наоборот. Такими классами-обертками, например, являются классы `StreamReader`, `StreamWriter`, `BinaryReader`, `BinaryWriter` и др. Работу с ними мы рассмотрим позднее.

Использование класса FileStream для файловых операций

Класс `FileStream` представляет собой поток файла. Он позволяет производить операции чтения и записи из файлов (как синхронные, так и асинхронные). Данный класс поддерживает метод `Seek`, что позволяет перемещать текущее положение курсора в файле. Данный класс позволяет работать с файлом как с последовательностью байт и, следовательно, не поддерживает напрямую чтение и запись других типов данных.

Пример использования данного класса представлен ниже.

```
using System;
using System.IO;
using System.Text;

namespace FileStreamExample
{
    class Program
    {
        // Демонстрационный пример
        // работы с классом FileStream
        static void Main(string[] args)
        {
            // Получаем путь к файлу
            Console.WriteLine("Введите путь к файлу:");
            string filePath = Console.ReadLine();
            // Создаем объект FileStream -
```

```
            // коментарии к конструктору см. ниже в уроке
            // Использование блока using позволяет правильно
            // разрушить FileStream
            using (FileStream fs = new FileStream(filePath,
                FileMode.Create, FileAccess.ReadWrite, FileShare.ReadWrite))
            {
                // Получаем данные для записи в файл
                Console.WriteLine("Введите строку для записи в файл:");
                string writeText = Console.ReadLine();
                // Преобразуем строку в массив байт,
                // т.к. FileStream умеет работать только с байтами
                // Как вам известно, строку можно по-разному преобразовать
                // в массив байт - все зависит от выбранной кодировки.
                // В данном случае мы используем кодировку UTF-8 (Unicode),
                // следовательно каждый символ будет преобразован
                // в 2 байта согласно правилам UTF-8.
                byte[] writeBytes = Encoding.UTF8.GetBytes(writeText);
                // Запишем данные в файл
                fs.Write(writeBytes, 0, writeBytes.Length);
                // Сохраним данные из буфера на диск
                fs.Flush();
                // Теперь прочитаем данные из файла
                // для этого нужно сначала установить курсор
                // на начало файла
                fs.Seek(0, SeekOrigin.Begin);
                // Теперь прочитаем данные в другой массив байт
                byte[] readBytes = new byte[(int)fs.Length];
                fs.Read(readBytes, 0, readBytes.Length);
                // Преобразуем байты в строку
                // Для этого используем ту же кодировку, что и для записи,
                // иначе нет гарантии, что мы получим правильную строку
                string readText = Encoding.UTF8.GetString(readBytes);
                // Выведем ее на консоль
                Console.WriteLine("Данные, прочитанные из файла: {0}",
                    readText);
            }
            Console.Read();
        }
    }
}
```

Особого внимания заслуживает конструктор FileStream:

```
FileStream fs = new FileStream(filePath, FileMode.Create,
     FileAccess.ReadWrite, FileShare.ReadWrite);
```

Первым параметром передается путь к файлу (с именем и расширением) в виде переменной или литерала `String`.

Второй параметр данного конструктора – переменная типа `FileMode` (перечисление). Он описывает, каким образом файл должен

быть открыт операционной системой. Принимаемые им значения приведены в табл. 1.

Табл. 1. Значения, принимаемые типом FileMode.

FileMode.Append	Открывает файл, если он существует, и перемещает курсор в конец файл. Если файл не существует – создает новый файл. FileMode.Append можно использовать только вместе с FileAccess.Write.
FileMode.Create	Создает новый файл, если файл уже существует, то он будет переписан. Курсор устанавливается на начало файла.
FileMode.CreateNew	Создает новый файл, если файл уже существует, то будет вызвано исключение IOException. Курсор устанавливается на начало файла.
FileMode.Open	Открывает существующий файл, если файла не существует, то будет вызвано исключение FileNotFoundException. Курсор устанавливается на начало файла.
FileMode.OpenOrCreate	Открывает существующий файл, если файла не существует, то будет создан новый файл. Курсор устанавливается на начало файла.
FileMode.Truncate	Открывает существующий файл и усекает его размер до нуля.

Третий параметр – переменная типа FileAccess (перечисление, имеющее атрибут FlagsAttribute, т.е. поддерживающее побитовое соединение составляющих его значений с помощью оператора |). Он описывает, каким образом осуществляется доступ к файлу: запись (FileAccess.Write), чтение (FileAccess.Read) или запись и чтение (FileAccess.ReadWrite).

Последний параметр – переменная типа FileShare (тоже перечисление, имеющее атрибут FlagsAttribute). Этот параметр позволяет управлять доступом, который другие объекты FileStream могут осуществлять к этому файлу (см. табл. 2).

Табл. 2. Значения, принимаемые типом FileAccess.

FileShare.Delete	Разрешает удаление файла.
FileShare.Inheritable	Разрешает наследование дескриптора файла дочерними процессами. В Win32 непосредственная поддержка этого свойства не обеспечена.
FileShare.None	Отклоняет совместное использование текущего файла. Любой запрос на открытие файла (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт.
FileShare.Read	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.
FileShare.ReadWrite	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.
FileShare.Write	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.

Использование класса StreamWriter и StreamReader для файловых операций

Как уже упоминалось выше, работать с байтовыми массивами непосредственно не всегда удобно. И для решения этой проблемы существуют классы-обертки, упрощающие решение задачи. Такими классами обертками для работы с символами и строками могут служить классы System.IO.StreamReader и System.IO.StreamWriter. Данные классы могут самостоятельно работать с текстовыми файлами – для этого необходимо использовать соответствующие конструкторы.

Необходимо помнить, что вызов метода Dispose() или Close() объекта StreamReader или StreamWriter завершает и базовый поток, поэтому повторный вызов этих методов для базового потока не требуется.

Продемонстрируем на примере работу с данными классами.

```
using System;
using System.IO;
using System.Text;
```

```
namespace StreamWriterReaderExample
{
    class Program
    {
        // Демонстрационный пример
        // работы с классами StreamWriter
        // и StreamReader
        static void Main(string[] args)
        {
            // Получаем путь к файлу
            Console.WriteLine("Введите путь к файлу:");
            string filePath = Console.ReadLine();
            // Создаем объект StreamWriter для
            // записи строк в различных кодировках в файл.
            // Создадим его как обертку для
            // другого файлового потока
            FileStream fs =
                new FileStream(filePath, FileMode.Create,
                    FileAccess.ReadWrite, FileShare.ReadWrite);
            StreamWriter sw = new StreamWriter(fs, Encoding.UTF8);
            // Получаем данные для записи в файл
            Console.WriteLine("Введите строку для записи в файл:");
            string writeText = Console.ReadLine();
            // Запишем данные в поток
            sw.Write(writeText);
            // Сохраним данные из буфера на диск и закроем поток
            sw.Dispose();
            // Теперь прочитаем данные из файла
            // для этого используем StreamReader.
            // На этот раз воспользуемся другим конструктором
            StreamReader sr = new StreamReader(filePath, Encoding.UTF8);
            // Прочитаем данные
            string readText = sr.ReadToEnd();
            sr.Dispose();
            // Выведем их на консоль
            Console.WriteLine("Данные, прочитанные из файла: {0}", readText);
            Console.Read();
        }
    }
}
```

При работе с этими классами стоит обратить внимание на следующие моменты. У объектов типа `StreamWriter` есть булевое свойство `AutoFlush`, установив которое в `true` можно не вызывать каждый раз метод `Flush()` после записи очередного блока данных в поток. Методы `Write()` и `WriteLine()` принимают объекты различных типов, вызывая методы `ToString()` каждого из них. Также стоит обратить внимание на то, что эти методы поддерживают форматирование строк, используя для этого `String.Format()`.

У объектов типа `StreamReader` стоит обратить внимание на булевое свойство `EndOfStream`, которое показывает, не достигли ли мы конца потока. Также у них определено несколько методов для чтения:

1) `Read` – читает следующий символ, либо массив символов;
2) `ReadBlock` – читает массив символов;
3) `ReadLine` – читает строку от текущей позиции до символа перехода на новую строчку.
4) `ReadToEnd` – читает поток от текущей позиции до конца.

Использование класса BinaryWriter и BinaryReader для файловых операций

Данные классы предназначены для записи простых типов данных в поток в виде двоичных значений, а также строк в определенной кодировке. Они выступают в виде обертки для других, базовых потоков. Оба эти класса имеют перегруженные конструкторы, в которых можно задать кодировку для работы со строками.

Пример работы с ними представлен ниже.

```
using System;
using System.IO;

namespace BinaryWriterReaderExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Получаем путь к файлу
            Console.WriteLine("Введите путь к файлу:");
            string filePath = Console.ReadLine();
            using (FileStream fs =
                new FileStream(filePath, FileMode.Create,
                    FileAccess.ReadWrite, FileShare.ReadWrite))
            {
                // Создаем объект BinaryWriter для
                // записи простых типов данных в поток.
                BinaryWriter bw = new BinaryWriter(fs);
                // Получаем данные для записи в файл
                Console.WriteLine("Введите строку для записи в файл:");
                string writeText = Console.ReadLine();
                int writeNum = -1;
                while (true)
```

```
                {
                    try
                    {
                        Console.WriteLine("Введите целое число " +
                            "для записи в файл:");
                        writeNum = Convert.ToInt32(Console.ReadLine());
                        break;
                    }
                    catch
                    {
                        Console.WriteLine("Ошибка! " +
                            "Не удалось преобразовать строку в целое число!");
                    }
                }
                // Запишем данные в поток
                bw.Write(writeText);
                bw.Write(writeNum);
                // Сохраним данные из буфера на диск
                bw.Flush();
                // Установим курсор на начало файла
                fs.Seek(0, SeekOrigin.Begin);
                // Теперь прочитаем данные из файла
                // для этого используем BinaryReader.
                BinaryReader br = new BinaryReader(fs);
                // Данные читать нужно в том же порядке,
                // в котором они и записывались в поток
                Console.WriteLine("Строка, прочитанная из файла:");
                Console.WriteLine(br.ReadString());
                Console.WriteLine("Целое число, прочитанное из файла:");
                Console.WriteLine(br.ReadInt32());
            }
            Console.Read();
        }
    }
}
```

Объекты класса `BinaryWriter` имеют всего один метод для записи (`Write`), который имеет несколько перегруженных вариантов, принимающих объекты различных типов данных. Объекты же класса `BinaryReader` имеют несколько методов, которые предназначены для чтения различных типов данных.

При их использовании необходимо обратить внимание на то, что данные с помощью `BinaryReader`'а необходимо читать в том же порядке, в котором они были записаны `BinaryWriter`'ом.

Использование классов Directory, DirectoryInfo, File, FileInfo для файловых операций

Данные классы предназначены для работы с папками и файлами: `Directory` и `DirectoryInfo` – для работы с папками, а `File` и `FileInfo` – с файлами. Отличие этих классов в том, что классы `Directory` и `File` - статические, а `DirectoryInfo` и `FileInfo` позволяют создавать объекты. Назначение же и тех и других одинаковое.

Легче всего понять разницу между ними, если рассмотреть пример создания папки обоими способами.

С помощью `Directory`:

```
using System;
using System.IO;

namespace DirectoryFileExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Введите полный путь для создания новой папки: ");
            Directory.CreateDirectory(Console.ReadLine());
        }
    }
}
```

И с помощью `DirectoryInfo`:

```
using System;
using System.IO;

namespace DirectoryFileExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Введите полный путь для создания новой папки: ");
            DirectoryInfo di = new DirectoryInfo(Console.ReadLine());
            di.Create();
        }
    }
}
```

Аналогично различаются и классы `FileInfo` и `File`.

Рассмотрим назначение основных методов этих классов.

Табл. 3. Назначение основных методов и свойств классов Directory и DirectoryInfo.

Directory	DirectoryInfo	File	FileInfo	Описание
CreateDirectory	Create	Create	Create	Создает папку или файл
Delete	Delete	Delete	Delete	Удаляет папку или файл. Для папок есть перегруженный вариант, который принимает дополнительный булевый параметр, установка которого в true приводит к удалению папки со всем содержимым.
Move	MoveTo	Move	MoveTo	Перемещает файл или папку в новое место. Если переместить в то же самое место, но с другим именем – переименовывает.
-	-	Copy	CopyTo	Копирует файлы в новое место. У папок данный метод отсутствует.
Exists	Exists	Exists	Exists	Существует ли файл (папка).
GetParent	Parent	-	Directory	Возвращает `DirectoryInfo` более высокого уровня.
GetLogicalDrives	-	-	-	Возвращает логические диски компьютера. Аналогичен методу `Environment.GetLogicalDrives()` и `DriveInfo.GetDrives()`.
SetCurrentDirectory	-	-	-	Устанавливает текущую папку для всего приложения в целом.
GetDirectories	GetDirectories	-	-	Возвращает массив имен или `DirectoryInfo` подкаталогов. Имеется перегруженный вариант, который принимает поисковый шаблон в формате DOS (?, *), позволяющий выбирать только те папки, которые соответствуют заданному шаблону.
GetFiles	GetFiles	-	-	Возвращает массив имен или `FileInfo` принадлежащих папке файлов. Имеется перегруженный вариант, который принимает поисковый шаблон в формате DOS (?, *), позволяющий выбирать только те файлы, которые соответствуют заданному шаблону.
GetFileSystemEntries	GetFileSystemInfos	-	-	Возвращает массив имен или `FileSystemInfo`, т.е. принадлежащих каталогу файлов и подкаталогов. Имеется перегруженный вариант, который принимает поисковый шаблон в формате DOS (?, *), позволяющий выбирать только те папки и файлы, которые соответствуют заданному шаблону.
CreateDirectory	CreateSubdirectory	-	-	Создает папку в текущей папке (поддиректорию).
GetCreationTime, GetCreationTimeUtc, SetCreationTime, SetCreationTimeUtc	CreationTime, CreationTimeUtc	GetCreationTime, GetCreationTimeUtc, SetCreationTime, SetCreationTimeUtc	CreationTime, CreationTimeUtc	Возвращает или устанавливает дату и время создания файла или папки в обычном и международном (UTC) формате.
GetLastAccessTime, GetLastAccessTimeUtc, SetLastAccessTime, SetLastAccessTimeUtc	LastAccessTime, LastAccessTimeUtc	GetLastAccessTime, GetLastAccessTimeUtc, SetLastAccessTime, SetLastAccessTimeUtc	LastAccessTime, LastAccessTimeUtc	Возвращает или устанавливает дату и время последнего доступа к файлу или папке в обычном и

				международном (UTC) формате.
GetLastWriteTime, GetLastWriteTimeUtc, SetLastWriteTime, SetLastWriteTimeUtc	LastWriteTime	GetLastWriteTime, GetLastWriteTimeUtc, SetLastWriteTime, SetLastWriteTimeUtc	LastWriteTime	Возвращает или устанавливает дату и время последнего изменения (записи) файла или папки в обычном и международном (UTC) формате.

Пример работы с данными классами вы можете найти в папке с исходными кодами.

Кроме того, хотелось бы обратить внимание на статический класс .NET, облегчающий работу с путями – `Path`. Рассмотрим назначение основных методов данного класса (табл. 4).

Табл. 4. Назначение методов класса `Path`.

Название метода	Назначение
Path.ChangeExtension	Позволяет изменить расширение файла в пути.
Path.Combine	Позволяет объединить две части пути. Например, путь к папке и имя файла, при этом автоматически определяет наличие или отсутствие разделителя / в конце первой части пути и в начале второй части. Другими словами, метод корректно объединяет две части пути в один вида: @"C:\temp\" и @"\filename.dat".
Path.GetDirectoryName	Вычленяет из полного пути путь к папке (без оконечного разделителя \).
Path.GetExtension	Вычленяет из полного пути расширение файла (с лидирующей точкой).
Path.GetFileName	Вычленяет из полного пути имя файла с расширением.
Path.GetFileNameWithoutExtension	Вычленяет из полного пути имя файла без расширения.
Path.GetFullPath	Возвращает полный путь по относительному.
Path.GetInvalidFileNameChars	Возвращает массив символов, которые не могут быть использованы в имени файла.
Path.GetInvalidPathChars	Возвращает массив символов, которые не могут быть использованы в пути.
Path.GetInvalidFileNameChars	Возвращает массив символов, которые не могут быть использованы в имени файла.
Path.GetPathRoot	Возвращает корень (диск, с оконечным разделителем /) данного пути.
Path.GetRandomFileName	Генерирует корректное случайное имя файла (с расширением).
Path.GetTempFileName	Генерирует уникальное имя файла во временной папке и возвращает полный путь к нему.
Path.GetTempPath	Возвращает полный путь к временной папке.
Path.HasExtension	Возвращает булевое значение – есть ли расширение у файла (`true`) или нет (`false`).
Path.IsPathRooted	Возвращает булевое значение – абсолютный ли путь (`true`) или нет (`false`).

Регулярные выражения

.NET Framework имеет встроенную поддержку регулярных выражений. Классы, осуществляющие работу с регулярными выражениями в расположены в пространстве имен `System.Text.RegularExpressions`. Их назначение описано в таблице.

Табл. 5. Классы пространства имен System.Text.RegularExpressions

`Capture`	Представляет результаты из отдельной записи части выражения. `Capture` представляет одну подстроку для отдельной успешной записи.
`CaptureCollection`	Представляет последовательность подстрок записи. `CaptureCollection` возвращает набор записей, сделанных одной группой записи.
`Group`	`Group` представляет результаты отдельной группы записи. Группа записи может записать ноль, одну или более строк в одно совпадение из-за кванторов, таким образом `Group` предоставляет коллекцию объектов `Capture`.
`GroupCollection`	Представляет коллекцию групп записи. `GroupCollection` возвращает набор групп записи в одном совпадении.
`Match`	Представляет результаты из отдельного совпадения регулярного выражения.
`MatchCollection`	Представляет набор успешных совпадений, обнаруженных путем итеративного применения шаблона регулярного выражения к входной строке.
`Regex`	Представляет постоянное регулярное выражение.
`RegexCompilationInfo`	Представляет сведения о регулярном выражении, используемые для компиляции регулярного выражения в отдельную сборку.
`RegexRunner`	Класс `RegexRunner` является базовым классом для скомпилированных регулярных выражений.
`RegexRunnerFactory`	Создает класс `RegexRunner` для скомпилированного регулярного выражения.

Ключевой класс, который позволяет создавать регулярные выражения – `Regex` – имеет несколько конструкторов.

1) `new Regex(string pattern)`
2) `new Regex(string pattern, RegexOptions options)`

Рассмотрим назначение параетров конструкторов:

1) `pattern` – это маска регулярного выражения, созданная согласно синтаксису. С помощью нее будет производиться поиск в тексте.

2) `options` – дополнительные параметры, перечисление типа `RegexOptions`, которые имеют атрибут `FlagsAttribute` и, следовательно, могут быть объединены с помощью оператора |. Данное перечисление имеет следующие значения:
   - `RegexOptions.Compiled` – указывает, что регулярное выражение будет скомпилировано в сборку, что приводит к более быстрой работе, однако увеличивает время запуска.
   - `RegexOptions.CultureInvariant` – указывает на игнорирование региональных языковых различий.
   - `RegexOptions.ECMAScript` – включает ECMAScript-совместимое поведение для регулярного выражения. Это значение может быть использовано только вместе со значениями `IgnoreCase`, `Multiline` и `Compiled`. Использование этого значения вместе с любыми другими параметрами приводит к исключению.
   - `RegexOptions.ExplicitCapture` – указывает, что единственные допустимые записи являются явно поименованными или пронумерованными группами в форме (?<name>…).
   - `RegexOptions.IgnoreCase` – указывает, что регулярное выражение не будет учитывать регистр.
   - `RegexOptions.IgnorePatternWhitespace` – устраняет из шаблона неизбежные пробелы и включает комментарии, помеченные "#".
   - `RegexOptions.Multiline` – многострочный режим. Изменяет значение символов "^" и "$" так, что они совпадают, соответственно, в начале и конце любой строки, а не только в начале и конце целой строки.
   - `RegexOptions.None` – указывает на отсутствие дополнительных параметров.
   - `RegexOptions.RightToLeft` – указывает на направление поиска – справа налево.

- RegexOptions.Singleline – указывает на применение однострочного режима.

Рассмотрим синтаксис и примеры использования регулярных выражений для поиска и замены подстрок в тексте. Для этого создадим небольшое демонстрационное консольное приложение. Данное приложение в предложенном тексте будет находить и заменять участки, соответствующие маске регулярного выражения. Листинг кода приведен ниже.

```
using System;
using System.Text.RegularExpressions;

namespace RegexConsoleExample
{
    class Program
    {
        static void Main(string[] args)
        {
            while (true)
            {
                Console.WriteLine("Введите анализируемый текст:");
                string inputText = Console.ReadLine();
                Console.WriteLine("Введите регулярное выражение для поиска:");
                string pattern = Console.ReadLine();
                //Создаем объект регулярного выражения
                Regex regex = new Regex(pattern,
                    RegexOptions.Multiline | RegexOptions.Compiled |
                    RegexOptions.ExplicitCapture);
                //Получаем коллекцию совпадений
                MatchCollection matchCollection =
                    regex.Matches(inputText);
                //Формируем сообщение о найденных совпадениях
                Console.WriteLine("Найдено совпадений: {0}",
                    matchCollection.Count);
                //Выводим информацию о каждом найденном совпадении
                for (int i = 0; i < matchCollection.Count; i++)
                {
                    Console.WriteLine("{0}:\t{1}", i+1,
                        matchCollection[i].Value);
                }
                //Произведем замену всех найденных совпадений
                Console.WriteLine("Вы хотите заменить найденные" +
                    "совпадения новым текстом? y(es) / n(o)");
                string answ = Console.ReadLine();
                if (!String.IsNullOrEmpty(answ) &&
                    answ.ToLower().StartsWith("y"))
                {
                    //Заменяем все найденные совпадения в тексте
                    Console.WriteLine("Введите текст для замены:");
                    string replacementText = Console.ReadLine();
                    Console.WriteLine("Результат обработки:");
                    Console.WriteLine(regex.Replace(inputText,
                        replacementText));
                }
                Console.WriteLine("Продолжить? y(es) / n(o)");
                answ = Console.ReadLine();
```

```
                if (String.IsNullOrEmpty(answ) ||
                    answ.ToLower().StartsWith("n"))
                    break;
            }
        }
    }
}
```

Рассмотрим назначение и использование специальных символов, применяемых в масках регулярных выражений. Колонки «inputText» и «pattern» содержат значения одноименных переменных в листинге, приведенном выше.

Символ	Значение	inputText	pattern
.	Любой одиночный символ, кроме символа новой строки \n. Если использовано RegexOptions.Singleline, то точка представляет также и символ \n	x1zx2zx3zx4z	x.z
****	Определение метасимвола (escape). Например, символ "." соответствует любому символу, а "\." – только символу точки	x1zx2zx3zx.z	x\.z
|	Разделение шаблона (или)	xyzxyzxyzxyz	x|z
[множество символов]	Соответствует любому символу из данного множества (одному и только одному)	xyzwxyzwxyzwxyzw	[xyz]
[^множество символов]	Отрицание множества символов	xyzwxyzwxyzwxyzw	[^xyz]
^	Соответствует началу строки Отрицание – если применен при определении множества символов	xyzwvuxyzwvu	^xyz
$	Соответствует концу строки. Это конец строки или позиция перед символом начала новой строки	xyzwvuxyzwvu	wvu$
(элементы)	Группировка элементов	xyzwvuxyzwvu	(xy)
?	Повторение 0 или 1 раз регулярного выражения, стоящего перед данным символом	xxyyzzwwvvuu	x?y?
*****	Повторение 0 или более раз регулярного выражения, стоящего перед данным символом	xxyyzzwwvvuu	x*y*
+	Повторение 1 или более раз регулярного выражения, стоящего перед данным символом	xxyyzzwwvvuu	x+y+
{n}	Повторение n раз регулярного	xxyyyyzzwww	xy{2}

	выражения, стоящего перед данным выражением		
{n, m}	Повторение от n до m раз регулярного выражения, стоящего перед данным выражением	xxyyyyzwxxyyyzzw	xy{1,3}
{n,}	Повторение n и более раз регулярного выражения, стоящего перед данным выражением	xxyyyyzwxxyyyzzw	xy{1,}
***?**	Повторение 0 или более раз регулярного выражения, стоящего перед данным выражением, минимально возможное количество	xyyyzwvxyyyyyzwv	xy*?z
+?	Повторение 1 или более раз регулярного выражения, стоящего перед данным выражением, минимально возможное количество	xyyyzwvxyyyyyzwv	xy+?z
??	Повторение 0 или 1 раз регулярного выражения, стоящего перед данным выражением, минимально возможное количество	xyyyzwvxyyyyyzwv	xy??z
\w	Слово (**w**ord) – цифра или буква	xyz_~`!@#$%^&*()-=+|:;\",.<>?/12345	\w
\W	Не слово (не цифра и не буква)	xyz_~`!@#$%^&*()-=+|:;\",.<>?/12345	\W
\d	Десятичная (**d**ecimal) цифра	abcd12345	\d
\D	Не десятичная цифра	abcd12345	\D
\s	Пустое место (**s**pace) – пробел, \f, \n, \r, \t, \v	xy vw	\s
\S	Не пустое место (не пробел, не \f, не \n, не \r, не \t, не \v)	xy z\nw	\S
\b	Граница слова (**b**oundary). Символы для поиска пишутся для начала слова – справа, для конца – слева	xyzwvu qwerty qwerty	\bqw
\B	Не граница слова (символы для поиска пишутся для начала слова – справа, для конца – слева.	xyzwvu qwerty qwerty	\Ber
\A	"Истинное" начало строки	xyzwvu qwerty qwerty	xy\A
\Z	Конец строки, позиция перед символом начала новой строки	xyzw qwerty qwerty\n	ty\Z
\z	"Истинный" конец строки, позиция перед символом начала новой строки	xyzw qwerty qwerty\n	ty\n\z
\G	Непрерывное совпадение от конца предыдущего (**g**roup). Работает только с методом NextMatch() при использовании класса Match.	123456789a123456789	\G\d{3}
(?=…)	Позитивная опережающая проверка. Здесь: ? определяет, что в скобках действие над группи-	qwertyxyzwvu	.*(?=)

	ровками, а не группировка для поиска; = позитивная опережающая проверка или то, что должно обязательно идти после подстроки, которую мы ищем.		
(?!…)	Негативная опережающая проверка. Здесь: ? определяет, что в скобках действие над группировками, а не группировка для поиска; ! негативная опережающая проверка или то, что ни в коем случае не должно стоять после искомой строки.	qwertyxyzwvu	.*(?!)
(?<=…)	Позитивная ретроспективная проверка Здесь: ? определяет, что в скобках действие над группировками, а не группировка для поиска; <= позитивная ретроспективная проверка или то, что должно обязательно идти перед подстрокой, которую мы ищем.	qwertyxyzwvu	.*(?<=)
(?<!…)	Негативная ретроспективная проверка Здесь: ? определяет, что в скобках действие над группировками, а не группировка для поиска; <! – негативная ретроспективная проверка или то, что ни в коем случае не должно идти перед подстрокой, которую мы ищем.	qwertyxyzwvu	.*(?<!<b>)

Домашнее задание

1. Написать консольное приложение, реализующее работу с файлами. Приложение должно удовлетворять следующим требованиям:
   1) наличие меню;
   2) возможность осуществлять поиск файлов и папок по имени, размеру, датам создания, доступа и модификации;
   3) осуществлять поиск текстовых файлов по содержимому;
   4) работа с найденным списком – удаление, перемещение, копирование, замена в выбранных текстовых файлах одной подстроки на другую.
2. Написать консольное приложение, принимающее на вход путь к файлу, содержащему xhtml разметку. Приложение с помощью регулярных выражений должно осуществлять очистку от тегов содержимого переданного файла и предлагать сохранять результат в другой текстовый файл.
3. Создать класс `ApplicationSettingsHelper`, который бы позволял считывать и записывать с помощью `BinaryReader`'а и `BinaryWritter`'а настройки приложения (цвет консоли, размер окна, заголовок). Создать приложение, демонстрирующее работу этого класса.