

Системное программирование

Системное программирование (или программирование систем) — это вид программирования, который заключается в работе с системным программным обеспечением. Главным отличием системного программирования по сравнению с прикладным программированием является то, что прикладное программное обеспечение предназначено выпускать (создавать и обновлять) программы для пользователей, тогда как системное программирование предназначено выпускать программы, обслуживающие аппаратное обеспечение, что обуславливает значительную зависимость такого типа ПО от аппаратной части.

Кратко про классы и интерфейсы

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о машине, у которой есть такие характеристики как производитель, модель, год выпуска и т.д. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, у одних машин одно название, других - другое. И реально существующая машина (фактически экземпляр данного класса) будет представлять объект этого класса.

По сути, класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова `class`:

```
class Car
{
}
```

Класс можно определять внутри пространства имен, вне пространства имен, внутри другого класса. Как правило, классы помещаются в отдельные файлы.

Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями.

```
class Car
{
    public string brand; // производитель
    public string model; // модель

    //метод получения информации
    public void GetInfo()
    {
        Console.WriteLine($"Производитель: {brand} Модель: {model}");
    }
}
```

В данном случае класс `Car` представляет машину. Поле `brand` хранит производителя, а поле `model` - модель. А метод `GetInfo` выводит все данные на консоль. Чтобы все данные были доступны вне класса `Car` переменные и метод определены с модификатором `public`.

Конструкторы

Кроме обычных методов в классах используются также и специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор по умолчанию. Такой конструктор не имеет параметров и не имеет тела.

Так как класс Car не имеет никаких конструкторов, для него автоматически создается конструктор по умолчанию. И мы можем использовать этот конструктор. В частности, создадим один объект класса Car:

```
class Program
{
    static void Main(string[] args)
    {
        Car bmw = new Car();

        bmw.GetInfo();

        bmw.brand = "BMW";
        bmw.model = "X5";
        bmw.GetInfo();

        Console.Read();
    }
}
```

Для создания объекта Car используется выражение new Car(). Оператор new выделяет память для объекта Car. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Car. А переменная bmw получит ссылку на созданный объект.

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения).

После создания объекта мы можем обратиться к переменным объекта Car через переменную bmw и установить или получить их значения, например, bmw.brand = "BMW";.

Инициализаторы объектов

Для инициализации объектов классов можно применять инициализаторы. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

```
class Program
{
```

```
static void Main(string[] args)
{
    Car bmw = new Car { brand = "BMW", model = "X5" };

    bmw.GetInfo();

    Console.Read();
}
}
```

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

Интерфейсы

Интерфейс представляет ссылочный тип, который определяет набор методов и свойств, но не реализует их. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

Для определения интерфейса используется ключевое слово `interface`. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, `IComparable`, `IEnumerable` (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования. Например, интерфейс `IMovable`:

```
interface IMovable
{
    void Move();
}
```

Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами абстрактных классов. В данном случае интерфейс определяет метод `Move`, который будет представлять некоторое передвижение. Он не принимает никаких параметров и ничего не возвращает.

Еще один момент в объявлении интерфейса: все его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ `public`, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

В целом интерфейсы могут определять следующие сущности:

- Методы
- Свойства
- Индексаторы
- События

Однако интерфейсы не могут определять статические члены, переменные, константы.

Затем какой-нибудь класс или структура могут применить данный интерфейс:

```
class Car : IMovable
{
    public string brand; // производитель
    public string model; // модель

    //метод получения информации
    public void GetInfo()
    {
        Console.WriteLine($"Производитель: {brand} Модель: {model}");
    }

    public void Move()
    {
        Console.WriteLine("Машина едет");
    }
}

class Person : IMovable
{
    public void Move()
    {
        Console.WriteLine("Человек идет");
    }
}
```

При применении интерфейса, как и при наследовании после имени класса или структуры указывается двоеточие и затем идут названия применяемых интерфейсов. При этом класс должен реализовать все методы и свойства применяемых интерфейсов. При этом поскольку все методы и свойства интерфейса являются публичными, при реализации этих методов и свойств в классе к ним можно применять только модификатор public. Поэтому если класс должен иметь метод с каким-то другим модификатором, например, protected, то интерфейс не подходит для определения подобного метода.

Если класс или структура не реализуют какие-либо свойства или методы интерфейса, то мы столкнемся с ошибкой на этапе компиляции.

Применение интерфейса в программе:

```
class Program
{
    static void Action(IMovable movable)
    {
        movable.Move();
    }
    static void Main(string[] args)
    {
        Person person = new Person();
        Car car = new Car();
        Action(person);
        Action(car);
        Console.Read();
    }
}
```

В данной программе определен метод Action(), который в качестве параметра принимает объект интерфейса IMovable. На момент написания кода

мы можем не знать, что это будет за объект - какой-то класс или структура. Единственное, в чем мы можем быть уверены, что этот объект обязательно реализует метод Move и мы можем вызвать этот метод.

Иными словами, интерфейс — это контракт, что какой-то определенный тип обязательно реализует некоторый функционал.

Коллекции

Хотя в языке C# есть массивы, которые хранят в себе наборы однотипных объектов, но работать с ними не всегда удобно. Например, массив хранит фиксированное количество объектов, однако, что, если мы заранее не знаем, сколько нам потребуется объектов. И в этом случае намного удобнее применять коллекции. Еще один плюс коллекций состоит в том, что некоторые из них реализуют стандартные структуры данных, например, стек, очередь, словарь, которые могут пригодиться для решения различных специальных задач.

Большая часть классов коллекций содержится в пространствах имен System.Collections (простые необобщенные классы коллекций), System.Collections.Generic (обобщенные или типизированные классы коллекций) и System.Collections.Specialized (специальные классы коллекций). Также для обеспечения параллельного выполнения задач и многопоточного доступа применяются классы коллекций из пространства имен System.Collections.Concurrent

Основой для создания всех коллекций является реализация интерфейсов IEnumerator и IEnumerable (и их обобщенных двойников IEnumerator<T> и IEnumerable<T>). Интерфейс IEnumerator представляет перечислитель, с помощью которого становится возможен последовательный перебор коллекции, например, в цикле foreach. А интерфейс IEnumerable через свой метод GetEnumerator предоставляет перечислитель всем классам, реализующим данный интерфейс. Поэтому интерфейс IEnumerable (IEnumerable<T>) является базовым для всех коллекций.

Рассмотрим создание и применение двух коллекций:

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            // необобщенная коллекция ArrayList
            ArrayList objectList = new ArrayList() { 1, 2, "строка", 'c' };

            object obj = 45.8;

            objectList.Add(obj); // Добавление объекта в коллекцию objectList

            objectList.Add("строка 2"); // Добавление строки в коллекцию objectList

            objectList.RemoveAt(0); // удаление первого элемента
        }
    }
}
```

```

        foreach (object o in objectList) // Вывод в консоль каждого элемента из
objectList
        {
            Console.WriteLine(o);
        }

        Console.WriteLine("Общее число элементов коллекции: {0}", objectList.Count);

        // обобщенная коллекция List
        List<string> countries = new List<string>() { "Россия", "США",
"Великобритания", "Китай" };

        countries.Add("Франция");

        countries.RemoveAt(1); // удаление второго элемента

        foreach (string s in countries)
        {
            Console.WriteLine(s);
        }

        Console.ReadLine();
    }
}

```

Здесь используются две коллекции: необобщенная - `ArrayList` и обобщенная - `List`. Большинство коллекций поддерживают добавление элементов. Например, в данном случае добавление производится методом `Add`, но для других коллекций название метода может отличаться. Также большинство коллекций реализуют удаление (в данном примере производится с помощью метода `RemoveAt`).

С помощью свойства `Count` у коллекций можно посмотреть количество элементов.

И так как коллекции реализуют интерфейс `IEnumerable/IEnumerable<T>`, то все они поддерживают перебор в цикле `foreach`.

Конкретные методы и способы использования могут различаться от одного класса коллекции к другому, но общие принципы будут одни и те же для всех классов коллекций.

Необобщенные коллекции

Необобщенные или простые коллекции определены в пространстве имен `System.Collections`. Их особенность состоит в том, что их функциональность, функциональные возможности описываются в интерфейсах, которые также находятся в этом пространстве имен.

Рассмотрим основные интерфейсы:

IEnumerator: определяет метод `GetEnumerator`. Данный метод возвращает перечислитель - то есть некоторый объект, реализующий интерфейс `IEnumerator`.

IEnumerator: реализация данного интерфейса позволяет перебирать элементы коллекции с помощью цикла `foreach`

ICollection: является основой для всех необобщенных коллекций, определяет основные методы и свойства для всех необобщенных коллекций (например, метод `CopyTo` и свойство `Count`). Данный интерфейс унаследован от интерфейса `IEnumerable`, благодаря чему базовый интерфейс также реализуется всеми классами необобщенных коллекций

IList: позволяет получать элементы коллекции по порядку. Также определяет ряд методов для манипуляции элементами: `Add` (добавление элементов), `Remove/RemoveAt` (удаление элемента) и ряд других.

IComparer: определяет метод `int Compare(object x, object y)` для сравнения двух объектов

IDictionary: определяет поведение коллекции, при котором она должна хранить объекты в виде пар ключ-значение: для каждого объекта определяется уникальный ключ, и этому ключу соответствует определенное значение

IDictionaryEnumerator: определяет методы и свойства для перечислителя словаря

IEqualityComparer: определяет два метода `Equals` и `GetHashCode`, с помощью которых два объекта сравниваются на предмет равенства

IStructuralComparer: определяет метод `Compare` для структурного сравнения двух объектов: при таком сравнении сравниваются не ссылки на объекты, а непосредственное содержимое объектов

IStructuralEquatable: позволяет провести структурное равенство двух объектов. Как и в случае с интерфейсом `IStructuralComparer` сравнивается содержимое двух объектов

Эти интерфейсы реализуются следующими классами коллекций в пространстве имен `System.Collections`:

ArrayList: класс простой коллекции объектов. Реализует интерфейсы `IList`, `ICollection`, `IEnumerable`

BitArray: класс коллекции, содержащей массив битовых значений. Реализует интерфейсы `ICollection`, `IEnumerable`

Hashtable: класс коллекции, представляющей хэш-таблицу и хранящий набор пар "ключ-значение"

Queue: класс очереди объектов, работающей по алгоритму FIFO ("первый вошел - первый вышел"). Реализует интерфейсы `ICollection`, `IEnumerable`

SortedList: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы `ICollection`, `IDictionary`, `IEnumerable`

Stack: класс стека

Рассмотрим некоторые из этих классов коллекций.

ArrayList

Итак, класс ArrayList представляет коллекцию объектов. И если надо сохранить вместе разнотипные объекты - строки, числа и т.д., то данный класс как раз для этого подходит. Посмотрим на примере.

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList list = new ArrayList();
            list.Add(2.3); // заносим в список объект типа double
            list.Add(55); // заносим в список объект типа int
            list.AddRange(new string[] { "Hello", "world" }); // заносим в список
// строковый массив

            // перебор значений
            foreach (object o in list)
            {
                Console.WriteLine(o);
            }

            // удаляем первый элемент
            list.RemoveAt(0);

            // переворачиваем список
            list.Reverse();

            // получение элемента по индексу
            Console.WriteLine(list[0]);

            // перебор значений
            for (int i = 0; i < list.Count; i++)
            {
                Console.WriteLine(list[i]);
            }

            Console.ReadLine();
        }
    }
}
```

Во-первых, так как класс ArrayList находится в пространстве имен System.Collections, то подключаем его (using System.Collections;).

Вначале создаем объект коллекции через конструктор как объект любого другого класса: ArrayList list = new ArrayList();.

Далее последовательно добавляем разные значения. Данный класс коллекции, как и большинство других коллекций, имеет два способа добавления: одиночного объекта через метод Add и набора объектов, например, массива или другой коллекции через метод AddRange

Через цикл `foreach` мы можем пройти по всем объектам списка. И поскольку данная коллекция хранит разнородные объекты, а не только числа или строки, то в качестве типа перебираемых объектов выбран тип `object`: `foreach (object o in list)`

Многие коллекции, в том числе и `ArrayList`, реализуют удаление с помощью методов `Remove/RemoveAt`. В данном случае мы удаляем первый элемент, передавая в метод `RemoveAt` индекс удаляемого элемента.

В завершении мы опять же выводим элементы коллекции на экран только уже через цикл `for`. В данном случае с перебором коллекций дело обстоит также, как и с массивами. А число элементов коллекции мы можем получить через свойство `Count`

С помощью индексатора мы можем получить по индексу элемент коллекции так же, как и в массивах.

Обобщенные коллекции

Классы обобщенных коллекций находятся в пространстве имен `System.Collections.Generic`. Функционал коллекций также по большей части описывается в обобщенных интерфейсах.

Только интерфейсы обобщенных коллекций отличаются от необобщенных двойников не только наличием универсального параметра `T`, но и самой функциональностью. Рассмотрим основные интерфейсы обобщенных коллекций:

- `IEnumerable<T>`: определяет метод `GetEnumerator`, с помощью которого можно получать элементы любой коллекции
- . Реализация данного интерфейса позволяет перебирать элементы коллекции с помощью цикла `foreach`
- `IEnumerator<T>`: определяет методы, с помощью которых потом можно получить содержимое коллекции по очереди
- `ICollection<T>`: представляет ряд общих свойств и методов для всех обобщенных коллекций (например, методы `CopyTo`, `Add`, `Remove`, `Contains`, свойство `Count`)
- `IList<T>`: предоставляет функционал для создания последовательных списков
- `IComparer<T>`: определяет метод `Compare` для сравнения двух однотипных объектов
- `IDictionary<TKey, TValue>`: определяет поведение коллекции, при котором она должна хранить объекты в виде пар ключ-значение: для каждого объекта определяется уникальный ключ типа, указанного в параметре `TKey`, и этому ключу соответствует определенное значение, имеющее тип, указанный в параметре `TValue`

- `IEqualityComparer<T>`: определяет методы, с помощью которых два однотипных объекта сравниваются на предмет равенства

Эти интерфейсы реализуются следующими классами коллекций в пространстве имен `System.Collections.Generic`:

`List<T>`: класс, представляющий последовательный список. Реализует интерфейсы `ICollection<T>`, `IEnumerable<T>`

`Dictionary<TKey, TValue>`: класс коллекции, хранящей наборы пар "ключ-значение". Реализует интерфейсы `ICollection<T>`, `IEnumerable<T>`, `IDictionary<TKey, TValue>`

`LinkedList<T>`: класс двухсвязанного списка. Реализует интерфейсы `ICollection<T>` и `IEnumerable<T>`

`Queue<T>`: класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел"). Реализует интерфейсы `ICollection`, `IEnumerable<T>`

`SortedSet<T>`: класс отсортированной коллекции однотипных объектов. Реализует интерфейсы `ICollection<T>`, `ISet<T>`, `IEnumerable<T>`

`SortedList<TKey, TValue>`: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы `ICollection<T>`, `IEnumerable<T>`, `IDictionary<TKey, TValue>`

`SortedDictionary<TKey, TValue>`: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. В общем похож на класс `SortedList<TKey, TValue>`, основные отличия состоят лишь в использовании памяти и в скорости вставки и удаления

`Stack<T>`: класс стека однотипных объектов. Реализует интерфейсы `ICollection<T>` и `IEnumerable<T>`

Большинство обобщенных классов коллекций дублируют необобщенные классы коллекций. Но если вам не надо хранить объекты разных типов, то предпочтительнее использовать обобщенные коллекции.

Рассмотрим основные обобщенные коллекции.

Список `List<T>`

Класс `List<T>` представляет простейший список однотипных объектов.

Среди его методов можно выделить следующие:

- `void Add(T item)`: добавление нового элемента в список
- `void AddRange(ICollection collection)`: добавление в список коллекции или массива

- `int BinarySearch(T item)`: бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.
- `int IndexOf(T item)`: возвращает индекс первого вхождения элемента в списке
- `void Insert(int index, T item)`: вставляет элемент `item` в списке на позицию `index`
- `bool Remove(T item)`: удаляет элемент `item` из списка, и если удаление прошло успешно, то возвращает `true`
- `void RemoveAt(int index)`: удаление элемента по указанному индексу `index`
- `void Sort()`: сортировка списка

Посмотрим реализацию списка на примере:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> numbers = new List<int>() { 1, 2, 3, 45 };
            numbers.Add(6); // добавление элемента

            numbers.AddRange(new int[] { 7, 8, 9 });

            numbers.Insert(0, 666); // вставляем на первое место в списке число 666

            numbers.RemoveAt(1); // удаляем второй элемент

            foreach (int i in numbers)
            {
                Console.WriteLine(i);
            }

            Console.WriteLine();

            List<Car> cars = new List<Car>(3);

            cars.Add(new Car() { Brand = "BMW" });
            cars.Add(new Car() { Brand = "Ford" });

            foreach (Car c in cars)
            {
                Console.WriteLine(c.Brand);
            }

            Console.ReadLine();
        }
    }
}

class Car
```

```
{  
    public string Brand { get; set; }  
}
```

Здесь у нас создаются два списка: один для объектов типа `int`, а другой - для объектов `Car`. В первом случае мы выполняем начальную инициализацию списка: `List<int> numbers = new List<int>() { 1, 2, 3, 45 };`

Во втором случае мы используем другой конструктор, в который передаем начальную емкость списка: `List<Car> cars = new List<Car>(3);`. Указание начальной емкости списка (`capacity`) позволяет в будущем увеличить производительность и уменьшить издержки на выделение памяти при добавлении элементов. Также начальную емкость можно установить с помощью свойства `Capacity`, которое имеется у класса `List`.

Двухсвязный список `LinkedList<T>`

Класс `LinkedList<T>` представляет двухсвязный список, в котором каждый элемент хранит ссылку одновременно на следующий и на предыдущий элемент.

Если в простом списке `List<T>` каждый элемент представляет объект типа `T`, то в `LinkedList<T>` каждый узел представляет объект класса `LinkedListNode<T>`. Этот класс имеет следующие свойства:

Value: само значение узла, представленное типом `T`

Next: ссылка на следующий элемент типа `LinkedListNode<T>` в списке. Если следующий элемент отсутствует, то имеет значение `null`

Previous: ссылка на предыдущий элемент типа `LinkedListNode<T>` в списке. Если предыдущий элемент отсутствует, то имеет значение `null`

Используя методы класса `LinkedList<T>`, можно обращаться к различным элементам, как в конце, так и в начале списка:

- `AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode):` вставляет узел `newNode` в список после узла `node`.
- `AddAfter(LinkedListNode<T> node, T value):` вставляет в список новый узел со значением `value` после узла `node`.
- `AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode):` вставляет в список узел `newNode` перед узлом `node`.
- `AddBefore(LinkedListNode<T> node, T value):` вставляет в список новый узел со значением `value` перед узлом `node`.
- `AddFirst(LinkedListNode<T> node):` вставляет новый узел в начало списка

- AddFirst(T value): вставляет новый узел со значением value в начало списка
- AddLast(LinkedListNode<T> node): вставляет новый узел в конец списка
- AddLast(T value): вставляет новый узел со значением value в конец списка
- RemoveFirst(): удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным
- RemoveLast(): удаляет последний узел из списка

Посмотрим на двухсвязный список в действии:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            LinkedList<int> numbers = new LinkedList<int>();

            numbers.AddLast(1); // вставляем узел со значением 1 на последнее место
                               // так как в списке нет узлов, то последнее будет также и первым

            numbers.AddFirst(2); // вставляем узел со значением 2 на первое место

            numbers.AddAfter(numbers.Last, 3); // вставляем после последнего узла новый
            узел со значением 3

            foreach (int i in numbers)
            {
                Console.WriteLine(i);
            }

            LinkedList<Car> cars = new LinkedList<Car>();

            // добавляем car в список и получим объект LinkedListNode<Car>, в котором
            хранится BMW
            LinkedListNode<Car> bmw = cars.AddLast(new Car() { Brand = "BMW" });

            cars.AddLast(new Car() { Brand = "Ford" });
            cars.AddFirst(new Car() { Brand = "Dodge" });

            Console.WriteLine(bmw.Previous.Value.Brand); // получаем узел перед bmw и его
            значение
            Console.WriteLine(bmw.Next.Value.Brand); // получаем узел после тома и его
            значение

            Console.ReadLine();
        }
    }

    class Car
    {
        public string Brand { get; set; }
    }
}
```

Здесь создаются и используются два списка: для чисел и для объектов класса Car. Разберем работу с классом Car.

Методы вставки (AddLast, AddFirst) при добавлении в список возвращают ссылку на добавленный элемент LinkedListNode<T> (в нашем случае LinkedListNode<Car>). Затем управляя свойствами Previous и Next, мы можем получить ссылки на предыдущий и следующий узлы в списке.

Очередь Queue<T>

Класс Queue<T> представляет обычную очередь, работающую по алгоритму FIFO ("первый вошел - первый вышел").

У класса Queue<T> можно отметить следующие методы:

- Dequeue: извлекает и возвращает первый элемент очереди
- Enqueue: добавляет элемент в конец очереди
- Peek: просто возвращает первый элемент из начала очереди без его удаления

Посмотрим применение очереди на практике:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Queue<int> numbers = new Queue<int>();

            numbers.Enqueue(3); // очередь 3
            numbers.Enqueue(5); // очередь 3, 5
            numbers.Enqueue(8); // очередь 3, 5, 8

            // получаем первый элемент очереди
            int e1 = numbers.Dequeue();

            Console.WriteLine(e1);

            // получаем первый элемент очереди
            int e2 = numbers.Peek();

            Console.WriteLine(e2);

            Console.ReadLine();
        }
    }
}
```

Коллекция Stack<T>

Класс `Stack<T>` представляет коллекцию, которая использует алгоритм LIFO ("последний вошел - первый вышел"). При такой организации каждый следующий добавленный элемент помещается поверх предыдущего. Извлечение из коллекции происходит в обратном порядке - извлекается тот элемент, который находится выше всех в стеке.

В классе `Stack` можно выделить два основных метода, которые позволяют управлять элементами:

- `Push`: добавляет элемент в стек на первое место
- `Pop`: извлекает и возвращает первый элемент из стека
- `Peek`: просто возвращает первый элемент из стека без его удаления

Посмотрим на примере:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Stack<int> numbers = new Stack<int>();

            numbers.Push(3); // в стеке 3
            numbers.Push(5); // в стеке 5, 3
            numbers.Push(8); // в стеке 8, 5, 3

            int stackElement = numbers.Pop();

            Console.WriteLine(stackElement);
        }
    }
}
```

Коллекция `Dictionary<T, V>`

Еще один распространенный тип коллекции представляют словари. Словарь хранит объекты, которые представляют пару ключ-значение. Каждый такой объект является объектом структуры `KeyValuePair<TKey, TValue>`. Благодаря свойствам `Key` и `Value`, которые есть у данной структуры, мы можем получить ключ и значение элемента в словаре.

Рассмотрим на примере использование словарей:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
```

```

{
    static void Main(string[] args)
    {
        Dictionary<int, string> countries = new Dictionary<int, string>(5);
        countries.Add(1, "Россия");
        countries.Add(3, "Англия");
        countries.Add(2, "США");
        countries.Add(4, "Франция");
        countries.Add(5, "Китай");

        foreach (KeyValuePair<int, string> keyValue in countries)
        {
            Console.WriteLine(keyValue.Key + " - " + keyValue.Value);
        }

        Console.WriteLine();

        // получение элемента по ключу
        string country = countries[4];
        // изменение объекта
        countries[4] = "Япония";
        // удаление по ключу
        countries.Remove(2);

        foreach (KeyValuePair<int, string> keyValue in countries)
        {
            Console.WriteLine(keyValue.Key + " - " + keyValue.Value);
        }

        Console.ReadLine();
    }
}

```

Класс словарей также, как и другие коллекции, предоставляет методы Add и Remove для добавления и удаления элементов. Только в случае словарей в метод Add передаются два параметра: ключ и значение. А метод Remove удаляет не по индексу, а по ключу.

Так как в нашем примере ключами являются объекты типа int, а значениями - объекты типа string, то словарь в нашем случае будет хранить объекты KeyValuePair<int, string>. В цикле foreach мы их можем получить и извлечь из них ключ и значение.

Кроме того, мы можем получить отдельно коллекции ключей и значений словаря.

Интерфейсы IEnumerable и IEnumerator

Как мы увидели, основной для большинства коллекций является реализация интерфейсов IEnumerable и IEnumerator. Благодаря такой реализации мы можем перебирать объекты в цикле foreach:

```

foreach (var item in перечислимый_объект)
{
    // ...
}

```

Перебираемая коллекция должна реализовать интерфейс IEnumerable.

Интерфейс IEnumerable имеет метод, возвращающий ссылку на другой интерфейс - перечислитель:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

А интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере:

```
public interface IEnumerator
{
    bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов
    object Current { get; } // текущий элемент в контейнере
    void Reset(); // перемещение в начало контейнера
}
```

Метод MoveNext() перемещает указатель на текущий элемент на следующую позицию в последовательности. Если последовательность еще не закончилась, то возвращает true. Если же последовательность закончилась, то возвращается false.

Свойство Current возвращает объект в последовательности, на который указывает указатель.

Метод Reset() сбрасывает указатель позиции в начальное положение.

Каким именно образом будет осуществляться перемещение указателя и получение элементов зависит от реализации интерфейса. В различных реализациях логика может быть построена различным образом.

Например, без использования цикла foreach переберем коллекцию с помощью интерфейса IEnumerator:

```
using System;
using System.Collections;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = { 0, 2, 4, 6, 8, 10 };

            IEnumerator ie = numbers.GetEnumerator(); // получаем IEnumerator

            while (ie.MoveNext()) // пока не будет возвращено false
            {
                int item = (int)ie.Current; // берем элемент на текущей позиции
                Console.WriteLine(item);
            }
            ie.Reset(); // сбрасываем указатель в начало массива

            Console.Read();
        }
    }
}
```

Реализация IEnumerable и IEnumerator

Рассмотрим простешую реализацию IEnumerable на примере:

```
using System;
using System.Collections;

namespace ConsoleApp
{
    class Week : IEnumerable
    {
        string[] days = { "Понедельник", "Вторник", "Среда", "Четверг", "Пятница",
"Суббота", "Воскресенье" };

        public IEnumerator GetEnumerator()
        {
            return days.GetEnumerator();
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Week week = new Week();
            foreach (var day in week)
            {
                Console.WriteLine(day);
            }
            Console.Read();
        }
    }
}
```

В данном случае класс Week, который представляет неделю и хранит все дни недели, реализует интерфейс IEnumerable. Однако в данном случае мы поступили очень просто - вместо реализации IEnumerator мы просто возвращаем в методе GetEnumerator объект IEnumerator для массива.

```
public IEnumerator GetEnumerator()
{
    return days.GetEnumerator();
}
```

Благодаря этому мы можем перебрать все дни недели в цикле foreach.

Однако это было довольно просто - мы просто используем уже готовый перечислитель массива. Однако, возможно, потребуется задать свою собственную логику перебора объектов. Для этого реализуем интерфейс IEnumerator:

```
using System;
using System.Collections;

namespace ConsoleApp
{
    class WeekEnumerator : IEnumerator
    {
        string[] days;
        int position = -1;

        public WeekEnumerator(string[] days)
        {
            this.days = days;
        }
    }
}
```

```

        public object Current
        {
            get
            {
                if (position == -1 || position >= days.Length)
                    throw new InvalidOperationException();
                return days[position];
            }
        }

        public bool MoveNext()
        {
            if (position < days.Length - 1)
            {
                position++;
                return true;
            }
            else
                return false;
        }

        public void Reset()
        {
            position = -1;
        }
    }

    class Week
    {
        string[] days = { "Понедельник", "Вторник", "Среда", "Четверг", "Пятница",
"Суббота", "Воскресенье" };

        public IEnumerator GetEnumerator()
        {
            return new WeekEnumerator(days);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Week week = new Week();
            foreach (var day in week)
            {
                Console.WriteLine(day);
            }
            Console.Read();
        }
    }
}

```

Здесь теперь класс Week использует не встроенный перечислитель, а WeekEnumerator, который реализует IEnumerator.

Ключевой момент при реализации перечислителя - перемещения указателя на элемент. В классе WeekEnumerator для хранения текущей позиции определена переменная position. Следует учитывать, что в самом начале (в исходном состоянии) указатель должен указывать на позицию условно перед первым элементом. Когда будет производиться цикл foreach, то данный цикл вначале вызывает метод MoveNext и фактически перемещает указатель на одну позицию в

перед и только затем обращается к свойству Current для получения элемента в текущей позиции.

Задание 1. Имеется 3 стека, один из них пустой, другие два заполнены десятью случайными числами от 1 до 100. Написать программу, которая перенесет числа из двух заполненных стеков так, чтобы сначала в стеке шли четные числа, а после нечетные. Числа должны быть отсортированы в порядке возрастания. Ответ нужно вывести в консоль.

Работа со строками

Строки и класс System.String

Довольно большое количество задач, которые могут встретиться при разработке приложений, так или иначе связано с обработкой строк - парсинг веб-страниц, поиск в тексте, какие-то аналитические задачи, связанные с извлечением нужной информации из текста и т.д. Поэтому в этом плане работе со строками уделяется особое внимание.

В языке C# строковые значения представляет тип string, а вся функциональность работы с данным типом сосредоточена в классе System.String. Собственно string является псевдонимом для класса System.String. Объекты этого класса представляют текст как последовательность символов Unicode. Максимальный размер объекта String может составлять в памяти 2 ГБ, или около 1 миллиарда символов.

Создание строк

Создавать строки можно, как используя переменную типа string и присваивая ей значение, так и применяя один из конструкторов класса String:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "hello";
            string s2 = null;

            string s3 = new String('a', 6); // результатом будет строка "aaaaaa"
            string s4 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' });

            Console.WriteLine($"s1: {s1} s2: {s2} s3: {s3} s4: {s4}");

            Console.Read();
        }
    }
}
```

Конструктор String имеет различное число версий. Так, вызов конструктора `new String('a', 6)` создаст строку "aaaaaa". И так как строка представляет ссылочный тип, то может хранить значение `null`.

Строка как набор символов

Так как строка хранит коллекцию символов, в ней определен индексатор для доступа к этим символам:

```
public char this[int index] { get; }
```

Применяя, индексатор, мы можем обратиться к строке как к массиву символов и получить по индексу любой из ее символов:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "hello";
            char ch1 = s1[1]; // символ 'e'
            Console.WriteLine(ch1);
            Console.WriteLine(s1.Length);

            Console.Read();
        }
    }
}
```

Используя свойство `Length`, как и в обычном массиве, можно получить длину строки.

Основные методы строк

Основная функциональность класса String раскрывается через его методы, среди которых можно выделить следующие:

- `Compare`: сравнивает две строки с учетом текущей культуры (локали) пользователя
- `CompareOrdinal`: сравнивает две строки без учета локали
- `Contains`: определяет, содержится ли подстрока в строке
- `Concat`: соединяет строки
- `CopyTo`: копирует часть строки или всю строку в другую строку
- `EndsWith`: определяет, совпадает ли конец строки с подстрокой
- `Format`: форматирует строку
- `IndexOf`: находит индекс первого вхождения символа или подстроки в строке
- `Insert`: вставляет в строку подстроку
- `Join`: соединяет элементы массива строк
- `LastIndexOf`: находит индекс последнего вхождения символа или подстроки в строке

- Replace: замещает в строке символ или подстроку другим символом или подстрокой
- Split: разделяет одну строку на массив строк
- Substring: извлекает из строки подстроку, начиная с указанной позиции
- ToLower: переводит все символы строки в нижний регистр
- ToUpper: переводит все символы строки в верхний регистр
- Trim: удаляет начальные и конечные пробелы из строки

Разберем работу этих методов.

Конкатенация

Конкатенация строк или объединение может производиться как с помощью операции +, так и с помощью метода Concat:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "hello";
            string s2 = "world";
            string s3 = s1 + " " + s2; // результат: строка "hello world"
            string s4 = String.Concat(s3, "!!!"); // результат: строка "hello world!!!"

            Console.WriteLine(s3 + " " + s4);

            Console.Read();
        }
    }
}
```

Метод Concat является статическим методом класса String, принимающим в качестве параметров две строки. Также имеются другие версии метода, принимающие другое количество параметров.

Для объединения строк также может использоваться метод Join:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "Плохое";
            string s2 = "ПО";
            string s3 = "одного";
            string s4 = "человека";
            string s5 = "-";
            string s6 = "постоянная";
            string s7 = "работа";
            string s8 = "другого";
```

```

        string[] values = new string[] { s1, s2, s3, s4, s5, s6, s7, s8 };

        string s9 = String.Join(" ", values);

        Console.WriteLine(s9);

        Console.Read();
    }
}

```

Метод Join также является статическим. Используемая выше версия метода получает два параметра: строку-разделитель (в данном случае пробел) и массив строк, которые будут соединяться и разделяться разделителем.

Сравнение строк

Для сравнения строк применяется статический метод Compare:

```

using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "Привет";
            string s2 = "мир!!!";

            int result = String.Compare(s1, s2);
            if (result < 0)
            {
                Console.WriteLine("Строка s1 перед строкой s2");
            }
            else if (result > 0)
            {
                Console.WriteLine("Строка s1 стоит после строки s2");
            }
            else
            {
                Console.WriteLine("Строки s1 и s2 идентичны");
            }

            Console.ReadLine();
        }
    }
}

```

Данная версия метода Compare принимает две строки и возвращает число. Если первая строка по алфавиту стоит выше второй, то возвращается число меньше нуля. В противном случае возвращается число больше нуля. И третий случай - если строки равны, то возвращается число 0.

В данном случае так как символ П по алфавиту стоит ниже символа М, то и первая строка будет стоять после.

Поиск в строке

С помощью метода IndexOf мы можем определить индекс первого вхождения отдельного символа или подстроки в строке:

```

using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "Привет мир";
            char ch = 'и';
            int indexOfChar = s1.IndexOf(ch); // равно 2
            Console.WriteLine(indexOfChar);

            string subString = "вет";
            int indexOfSubString = s1.IndexOf(subString); // равно 3
            Console.WriteLine(indexOfSubString);

            Console.ReadLine();
        }
    }
}

```

Подобным образом действует метод `LastIndexOf`, только находит индекс последнего вхождения символа или подстроки в строку.

```

using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "Привет мир";
            char ch = 'и';
            int indexOfChar = s1.LastIndexOf(ch); // равно 2
            Console.WriteLine(indexOfChar);

            string subString = "вет";
            int indexOfSubString = s1.LastIndexOf(subString); // равно 3
            Console.WriteLine(indexOfSubString);

            Console.ReadLine();
        }
    }
}

```

Еще одна группа методов позволяет узнать начинается ли строка на определенную подстроку. Для этого предназначены методы `StartsWith` и `EndsWith`.

```

using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "Привет мир";
            string s2 = "Пр";

```



```

        string s3 = "мип";

        if (s1.StartsWith(s2) && !s1.EndsWith(s3))
        {
            Console.WriteLine("Строка s1 начинается с s2");
        }
        else if (s1.EndsWith(s3) && !s1.StartsWith(s2)) {
            Console.WriteLine("Строка s1 заканчивается на s3");
        }
        else
        {
            Console.WriteLine("Строка s1 начинается с s2 и заканчивается на s3");
        }

        Console.ReadLine();
    }
}

```

Разделение строк

С помощью функции Split мы можем разделить строку на массив подстрок. В качестве параметра функция Split принимает массив символов или строк, которые и будут служить разделителями. Например, подсчитаем количество слов в строке, разделив ее по пробельным символам:

```

using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string text = "Большинство из вас знают достоинства программистов. Их, конечно же, три: лень, нетерпеливость и высокомерие.";

            string[] words = text.Split(new char[] { ' ' });

            foreach (string s in words)
            {
                Console.WriteLine(s);
            }

            Console.ReadLine();
        }
    }
}

```

Это не лучший способ разделения по пробелам, так как во входной строке у нас могло бы быть несколько подряд идущих пробелов и в итоговый массив также бы попадали пробелы, поэтому лучше использовать другую версию метода:

```

string[] words = text.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

```

Второй параметр StringSplitOptions.RemoveEmptyEntries говорит, что надо удалить все пустые подстроки.

Обрезка строки

Для обрезки начальных или конечных символов используется функция Trim:

```

using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string text = " hello world ";

            text = text.Trim(); // результат "hello world"
            text = text.Trim(new char[] { 'd', 'h' }); // результат "ello worl"

            Console.WriteLine(text);

            Console.ReadLine();
        }
    }
}

```

Функция Trim без параметров обрезает начальные и конечные пробелы и возвращает обрезанную строку. Чтобы явным образом указать, какие начальные и конечные символы следует обрезать, мы можем передать в функцию массив этих символов.

Эта функция имеет частичные аналоги: функция TrimStart обрезает начальные символы, а функция TrimEnd обрезает конечные символы.

Обрезать определенную часть строки позволяет функция Substring:

```

using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string text = "Хороший день";
            // обрезаем начиная с третьего символа
            text = text.Substring(2);
            // результат "роший день"
            Console.WriteLine(text);
            // обрезаем сначала до последних двух символов
            text = text.Substring(0, text.Length - 2);
            // результат "роший де"
            Console.WriteLine(text);

            Console.ReadLine();
        }
    }
}

```

Функция Substring также возвращает обрезанную строку. В качестве параметра первая использованная версия применяет индекс, начиная с которого надо обрезать строку. Вторая версия применяет два параметра - индекс начала обрезки и длину вырезаемой части строки.

Вставка

Для вставки одной строки в другую применяется функция Insert:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string text = "Хороший день";
            string subString = "замечательный ";

            text = text.Insert(8, subString);
            Console.WriteLine(text);

            Console.ReadLine();
        }
    }
}
```

Первым параметром в функции Insert является индекс, по которому надо вставлять подстроку, а второй параметр - собственно подстрока.

Удаление строк

Удалить часть строки помогает метод Remove:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string text = "Хороший день";
            // индекс последнего символа
            int i = text.Length - 1;
            // вырезаем последний символ
            text = text.Remove(i);
            Console.WriteLine(text);

            // вырезаем первые два символа
            text = text.Remove(0, 2);

            Console.WriteLine(text);

            Console.ReadLine();
        }
    }
}
```

Первая версия метода Remove принимает индекс в строке, начиная с которого надо удалить все символы. Вторая версия принимает еще один параметр - сколько символов надо удалить.

Замена

Чтобы заменить один символ или подстроку на другую, применяется метод Replace:

```

using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string text = "хороший день";

            text = text.Replace("хороший", "плохой");
            Console.WriteLine(text);

            text = text.Replace("о", "");
            Console.WriteLine(text);

            Console.ReadLine();
        }
    }
}

```

Во втором случае применения функции Replace строка из одного символа "о" заменяется на пустую строку, то есть фактически удаляется из текста. Подобным способом легко удалять какой-то определенный текст в строках.

Смена регистра

Для приведения строки к верхнему и нижнему регистру используются соответственно функции ToUpper() и ToLower():

```

using System;
using System.Collections.Generic;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "Привет мир!";

            Console.WriteLine(s1.ToLower());
            Console.WriteLine(s1.ToUpper());

            Console.ReadLine();
        }
    }
}

```

Форматирование строк

При выводе строк в консоли с помощью метода Console.WriteLine мы можем применять форматирование вместо конкатенации:

```

using System;

namespace ConsoleApp
{
    class Car
    {
        public string Brand { get; set; }
        public string Model { get; set; }
    }
}

```

```

    }

    class Program
    {
        static void Main(string[] args)
        {
            Car car = new Car { Brand = "BMW", Model = "X5" };

            Console.WriteLine("Производитель: {0} Модель: {1}", car.Brand, car.Model);

            Console.ReadLine();
        }
    }
}

```

В строке " Производитель: {0} Модель: {1}" на место {0} и {1} затем будут вставляться в порядке следования car.Brand и car.Model.

То же самое мы можем сделать с помощью метода String.Format:

```

using System;

namespace ConsoleApp
{
    class Car
    {
        public string Brand { get; set; }
        public string Model { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Car car = new Car { Brand = "BMW", Model = "X5" };

            string s1 = String.Format("Производитель: {0} Модель: {1}", car.Brand,
car.Model);

            Console.WriteLine(s1);

            Console.ReadLine();
        }
    }
}

```

Метод Format принимает строку с плейсхолдерами типа {0}, {1} и т.д., а также набор аргументов, которые вставляются на место данных плейсхолдеров. В итоге генерируется новая строка.

В методе Format могут использоваться различные спецификаторы и описатели, которые позволяют настроить вывод данных. Рассмотрим основные описатели.

Все используемые форматы:

| | |
|--------------|--|
| C / c | Задаёт формат денежной единицы, указывает количество десятичных разрядов после запятой |
| D / d | Целочисленный формат, указывает минимальное количество цифр |
| E / e | Экспоненциальное представление числа, указывает количество десятичных разрядов после запятой |
| F / f | Формат дробных чисел с фиксированной точкой, указывает количество |

| | |
|--------------|---|
| | десятичных разрядов после запятой |
| G / g | Задаёт более короткий из двух форматов: F или E |
| N / n | Также задаёт формат дробных чисел с фиксированной точкой, определяет количество разрядов после запятой |
| P / p | Задаёт отображения знака процентов рядом с числом, указывает количество десятичных разрядов после запятой |
| X / x | Шестнадцатеричный формат числа |

Форматирование валюты

Для форматирования валюты используется описатель "C":

```
using System;
using System.Globalization;
using System.Text;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            //Console.OutputEncoding = Encoding.Unicode;
            double number = 23.7;
            string result = string.Format("{0:C2}", number);

            Console.WriteLine(result);

            Console.ReadLine();
        }
    }
}
```

Число после описателя указывает, сколько чисел будет использоваться после разделителя между целой и дробной частью. При выводе также добавляется обозначение денежного знака для текущей культуры компьютера.

Форматирование целых чисел

Для форматирования целочисленных значений применяется описатель "d":

```
using System;
using System.Globalization;
using System.Text;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            int number = 23;
            string result = String.Format("{0:d}", number);
            Console.WriteLine(result); // 23
            string result2 = String.Format("{0:d4}", number);
            Console.WriteLine(result2); // 0023

            Console.ReadLine();
        }
    }
}
```

```
}  
}
```

Число после описателя указывает, сколько цифр будет в числовом значении. Если в исходном числе цифр меньше, то к нему добавляются нули.

Форматирование дробных чисел

Для форматирования дробных чисел используется описатель F, число после которого указывает, сколько знаков будет использоваться после разделителя между целой и дробной частью. Если исходное число - целое, то к нему добавляются разделитель и нули.

```
using System;  
using System.Globalization;  
using System.Text;  
  
namespace ConsoleApp  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int number = 23;  
            string result = String.Format("{0:f}", number);  
            Console.WriteLine(result); // 23,00  
  
            double number2 = 45.08;  
            string result2 = String.Format("{0:f4}", number2);  
            Console.WriteLine(result2); // 45,0800  
  
            double number3 = 25.07;  
            string result3 = String.Format("{0:f1}", number3);  
            Console.WriteLine(result3); // 25,1  
  
            Console.ReadLine();  
        }  
    }  
}
```

Формат процентов

Описатель "P" задает отображение процентов. Используемый с ним числовой спецификатор указывает, сколько знаков будет после запятой:

```
using System;  
using System.Globalization;  
using System.Text;  
  
namespace ConsoleApp  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            double number = 0.15345;  
            Console.WriteLine("{0:P1}", number);  
  
            Console.ReadLine();  
        }  
    }  
}
```

Настраиваемые форматы

Используя знак #, можно настроить формат вывода. Например, нам надо вывести некоторое число в формате телефона +x (xxx)xxx-xx-xx:

```
using System;
using System.Globalization;
using System.Text;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            long number = 79255553344;
            string result = String.Format("{0:+# (###) ###-##-##}", number);
            Console.WriteLine(result);

            Console.ReadLine();
        }
    }
}
```

Метод ToString

Метод ToString() не только получает строковое описание объекта, но и может осуществлять форматирование. Он поддерживает те же описатели, что используются в методе Format:

```
using System;
using System.Globalization;
using System.Text;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            //Console.OutputEncoding = Encoding.Unicode;

            long number = 79255553344;
            Console.WriteLine(number.ToString("+# (###) ###-##-##"));

            double money = 24.8;
            Console.WriteLine(money.ToString("C2"));

            Console.ReadLine();
        }
    }
}
```

Интерполяция строк

Начиная с версии языка C# 6.0 (Visual Studio 2015) была добавлена такая функциональность, как интерполяция строк. Эта функциональность призвана заменить форматирование строк.

```
using System;

namespace ConsoleApp
{
```



```

class Car
{
    public string Brand { get; set; }
    public string Model { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Car car = new Car { Brand = "BMW", Model = "X5" };

        Console.WriteLine($"Производитель: {car.Brand}    Модель: {car.Model}");

        Console.ReadLine();
    }
}

```

Знак доллара перед строкой указывает, что будет осуществляться интерполяция строк. Внутри строки опять же используются плейсхолдеры {...}, только внутри фигурных скобок уже можно напрямую писать те выражения, которые мы хотим вывести.

Интерполяция, по сути, представляет более лаконичное форматирование. При этом внутри фигурных скобок мы можем указывать не только свойства, но и различные выражения языка C#:

```

using System;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 8;
            int y = 7;

            string result = $"{x} + {y} = {x + y}";

            Console.WriteLine(result);

            Console.ReadLine();
        }
    }
}

```

Уже внутри строки можно применять форматирование. В этом случае мы можем применять все те же описатели, что и в методе Format. Например, выведем номер телефона в формате +x xxx xxx xx xx:

```

using System;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            long number = 7925553344;

```

```

        Console.WriteLine($"{number:+# ### ## ##}");

        Console.ReadLine();
    }
}

```

Класс StringBuilder

Хотя класс `System.String` предоставляет нам широкую функциональность по работе со строками, все-таки он имеет свои недостатки. Прежде всего, объект `String` представляет собой неизменяемую строку. Когда мы выполняем какой-нибудь метод класса `String`, система создает новый объект в памяти с выделением ему достаточного места. Удаление первого символа - не самая затратная операция. Однако, когда подобных операций множество, а объем текста, для которого надо выполнить данные операции, также не самый маленький, то издержки при потере производительности становятся более существенными.

Чтобы выйти из этой ситуации во фреймворк `.NET` был добавлен новый класс `StringBuilder`, который находится в пространстве имен `System.Text`. Этот класс представляет динамическую строку.

При создании строки `StringBuilder` выделяет памяти больше, чем необходимо этой строке:

```

using System;
using System.Text;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            StringBuilder sb = new StringBuilder("Привет мир");

            Console.WriteLine("Длина строки: {0}", sb.Length);
            Console.WriteLine("Емкость строки: {0}", sb.Capacity);

            Console.ReadLine();
        }
    }
}

```

Теперь переменная `sb` представляет начальную строку "Привет мир". Эта строка имеет длину в 10 символов. Для хранения длины в классе `StringBuilder` есть свойство `Length`. Однако есть и вторая величина - емкость выделенной памяти. Это значение хранится в свойстве `Capacity`. Емкость — это выделенная память под объект. Хотя в данном случае длина равна 10 символов, но реально емкость будет составлять по умолчанию 16 символов.

Класс `StringBuilder` имеет еще ряд конструкторов, которые позволяют разными способами выполнить начальную инициализацию объекта. Так, мы можем задать пустой объект, но установить начальную емкость:

```

StringBuilder sb = new StringBuilder(20);

```

Если у нас заранее известен максимальный размер объекта, то мы можем таким образом сразу задать емкость и избежать последующих издержек при дополнительном выделении памяти.

Теперь посмотрим на примере использование и преимущества класса `StringBuilder`:

При создании объекта `StringBuilder` выделяется память по умолчанию для 16 символов, так как длина начальной строки меньше 16.

Дальше применяется метод `Append` - этот метод добавляет к строке подстроку. Так как при объединении строк их общая длина - 22 символа - превышает начальную емкость в 16 символов, то начальная емкость удваивается - до 32 символов.

Если бы итоговая длина строки была бы больше 32 символов, то емкость расширилась бы до размера длины строки.

Далее опять применяется метод `Append`, однако финальная длина уже будет 28 символов, что меньше 32 символов, и дополнительная память не будет выделяться.

Кроме метода `Append` класс `StringBuilder` предлагает еще ряд методов для операций над строками:

- `Insert`: вставляет подстроку в объект `StringBuilder`, начиная с определенного индекса
- `Remove`: удаляет определенное количество символов, начиная с определенного индекса
- `Replace`: заменяет все вхождения определенного символа или подстроки на другой символ или подстроку
- `AppendFormat`: добавляет подстроку в конец объекта `StringBuilder`

```
using System;
using System.Text;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            StringBuilder sb = new StringBuilder("Привет мир");
            sb.Append("!");
            sb.Insert(7, "компьютерный ");
            Console.WriteLine(sb);

            // заменяем слово
            sb.Replace("мир", "world");
            Console.WriteLine(sb);

            // удаляем 13 символов, начиная с 7-го
            sb.Remove(7, 13);
            Console.WriteLine(sb);

            // получаем строку из объекта StringBuilder
        }
    }
}
```

```

        string s = sb.ToString();
        Console.WriteLine(s);

        Console.ReadLine();
    }
}

```

Когда надо использовать класс String, а когда StringBulder?

Microsoft рекомендует использовать класс String в следующих случаях:

- При небольшом количестве операций и изменений над строками
- При выполнении фиксированного количества операций объединения. В этом случае компилятор может объединить все операции объединения в одну
- Когда надо выполнять масштабные операции поиска при построении строки, например IndexOf или StartsWith. Класс StringBuilder не имеет подобных методов.

Класс StringBuilder рекомендуется использовать в следующих случаях:

- При неизвестном количестве операций и изменений над строками во время выполнения программы
- Когда предполагается, что приложению придется сделать множество подобных операций

Регулярные выражения

Классы StringBuilder и String предоставляют достаточную функциональность для работы со строками. Однако .NET предлагает еще один мощный инструмент - регулярные выражения. Регулярные выражения представляют эффективный и гибкий метод по обработке больших текстов, позволяя в то же время существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками.

Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен System.Text.RegularExpressions. А центральным классом при работе с регулярными выражениями является класс Regex. Например, у нас есть некоторый текст и нам надо найти в нем все словоформы какого-нибудь слова. С классом Regex это сделать очень просто:

```

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

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";
            Regex regex = new Regex(@"туп(\w*)");
            MatchCollection matches = regex.Matches(s);
            if (matches.Count > 0)

```

```

        {
            foreach (Match match in matches)
                Console.WriteLine(match.Value);
        }
        else
        {
            Console.WriteLine("Совпадений не найдено");
        }

        Console.ReadLine();
    }
}

```

Здесь мы находим в искомой строке все словоформы слова "туп". В конструктор объекта `Regex` передается регулярное выражение для поиска.

Чуть позже мы разберем некоторые элементы синтаксиса регулярных выражений, а пока достаточно знать, что выражение `туп(\w*)` обозначает, найти все слова, которые имеют корень "туп" и после которого может стоять различное количество символов. Выражение `\w` означает алфавитно-цифровой символ, а звездочка после выражения указывает на неопределенное их количество - их может быть один, два, три или вообще не быть.

Метод `Matches` класса `Regex` принимает строку, к которой надо применить регулярные выражения, и возвращает коллекцию найденных совпадений.

Каждый элемент такой коллекции представляет объект `Match`. Его свойство `Value` возвращает найденное совпадение.

Параметр `RegexOptions`

Класс `Regex` имеет ряд конструкторов, позволяющих выполнить начальную инициализацию объекта. Две версии конструкторов в качестве одного из параметров принимают перечисление `RegexOptions`. Некоторые из значений, принимаемых данным перечислением:

- `Compiled`: при установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение
- `CultureInvariant`: при установке этого значения будут игнорироваться региональные различия
- `IgnoreCase`: при установке этого значения будет игнорироваться регистр
- `IgnorePatternWhitespace`: удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака `#`
- `Multiline`: указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы `"^"` и `"$"` совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста
- `RightToLeft`: приписывает читать строку справа налево
- `Singleline`: устанавливает однострочный режим, а весь текст рассматривается как одна строка

Например:

```
Regex regex = new Regex(@"туп(\w*)", RegexOptions.IgnoreCase);
```

При необходимости можно установить несколько параметров:

```
Regex regex = new Regex(@"тип(\w*)", RegexOptions.Compiled | RegexOptions.IgnoreCase);
```

Синтаксис регулярных выражений

Рассмотрим вкратце некоторые элементы синтаксиса регулярных выражений:

- `^`: соответствие должно начинаться в начале строки (например, выражение `@'^пр\w*` соответствует слову "привет" в строке "привет мир")
- `$`: конец строки (например, выражение `@'\w*ир$` соответствует слову "мир" в строке "привет мир", так как часть "ир" находится в самом конце)
- `.`: знак точки определяет любой одиночный символ (например, выражение `"м.р"` соответствует слову "мир" или "мор")
- `*`: предыдущий символ повторяется 0 и более раз
- `+`: предыдущий символ повторяется 1 и более раз
- `?`: предыдущий символ повторяется 0 или 1 раз
- `\s`: соответствует любому пробельному символу
- `\S`: соответствует любому символу, не являющемуся пробелом
- `\w`: соответствует любому алфавитно-цифровому символу
- `\W`: соответствует любому не алфавитно-цифровому символу
- `\d`: соответствует любой десятичной цифре
- `\D`: соответствует любому символу, не являющемуся десятичной цифрой

Это только небольшая часть элементов. Более подробное описание синтаксиса регулярных выражений можно найти на msdn в статье Элементы языка регулярных выражений — краткий справочник.

Теперь посмотрим на некоторые примеры использования. Возьмем первый пример с скороговоркой "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа" и найдем в ней все слова, где встречается символы "губ":

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

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = "Бык Тупогуб, тупогубенький бычок, у быка губа бела была тупа";
            Regex regex = new Regex(@"\w*губ\w*");
            MatchCollection matches = regex.Matches(s);
            if (matches.Count > 0)
            {
                foreach (Match match in matches)
                {
                    Console.WriteLine(match.Value);
                }
            }
            else
            {
                Console.WriteLine("Совпадений не найдено");
            }

            Console.ReadLine();
        }
    }
}
```

```
}
```

Так как выражение `\w*` соответствует любой последовательности алфавитно-цифровых символов любой длины, то данное выражение найдет все слова, содержащие корень "губ".

Второй простенький пример - нахождение телефонного номера в формате +7 111-111-11-11:

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

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = "+7 999-555-44-33";
            Regex regex = new Regex(@"\+7 \d{3}-\d{3}-\d{2}-\d{2}");

            MatchCollection matches = regex.Matches(s);
            if (matches.Count > 0)
            {
                foreach (Match match in matches)
                    Console.WriteLine(match.Value);
            }
            else
            {
                Console.WriteLine("Совпадений не найдено");
            }

            Console.ReadLine();
        }
    }
}
```

Если мы точно знаем, сколько определенных символов должно быть, то мы можем явным образом указать их количество в фигурных скобках: `\d{3}` - то есть в данном случае три цифры.

Мы можем не только задать поиск по определенным типам символов - пробелы, цифры, но и задать конкретные символы, которые должны входить в регулярное выражение. Например, перепишем пример с номером телефона и явно укажем, какие символы там должны быть:

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

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = "+7 999-555-44-33";
            Regex regex = new Regex(@"\+7 [0-9]{3}-[0-9]{3}-[0-9]{2}-[0-9]{2}");

            MatchCollection matches = regex.Matches(s);
            if (matches.Count > 0)
            {

```

```

        foreach (Match match in matches)
            Console.WriteLine(match.Value);
    }
    else
    {
        Console.WriteLine("Совпадений не найдено");
    }

    Console.ReadLine();
}
}
}

```

В квадратных скобках задается диапазон символов, которые должны в данном месте встречаться. В итоге данный и предыдущий шаблоны телефонного номера будут эквивалентны.

Также можно задать диапазон для алфавитных символов: `Regex regex = new Regex("[a-v]{5}");` - данное выражение будет соответствовать любому сочетанию пяти символов, в котором все символы находятся в диапазоне от а до v.

Можно также указать отдельные значения: `Regex regex = new Regex(@"[2]*-[0-9]{3}-\d{4}");`.

С помощью операции `|` можно задать альтернативные символы: `Regex regex = new Regex(@"[2|3]{3}-[0-9]{3}-\d{4}");`. То есть первые три цифры могут содержать только двойки или тройки. Такой шаблон будет соответствовать, например, строкам "222-222-2222" и "323-435-2318". А вот строка "235-435-2318" уже не подпадает под шаблон, так как одной из трех первых цифр является цифра 5.

Итак, у нас такие символы, как `*`, `+` и ряд других используются в качестве специальных символов. И возникает вопрос, а что делать, если у нас надо найти, строки, где содержится точка, звездочка или какой-то другой специальный символ? В этом случае нам надо просто экранировать эти символы слешем.

Проверка на соответствие строки формату

Нередко возникает задача проверить корректность данных, введенных пользователем. Это может быть проверка электронного адреса, номера телефона, Класс `Regex` предоставляет статический метод `IsMatch`, который позволяет проверить входную строку с шаблоном на соответствие

```

using System;
using System.Text.RegularExpressions;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string pattern = @"^(?("")(""[^"]*"")@)|(((?!-)[0-9a-z]|\.(?!\.))|[-!#$%&'*/+=\^_\{\}|\~\w])*)(?<=[0-9a-z]@))?(\[ \]|\[ \d{1,3}\. \]{3}\d{1,3}\]|(((?!-)[0-9a-z]|\.(?!\.))|[-!#$%&'*/+=\^_\{\}|\~\w])*(?!-)[0-9a-z]@))";
            while (true)
            {
                Console.WriteLine("Введите адрес электронной почты");
                string email = Console.ReadLine();
            }
        }
    }
}

```



```

        if (Regex.IsMatch(email, pattern, RegexOptions.IgnoreCase))
        {
            Console.WriteLine("Email подтвержден");
        }
        else
        {
            Console.WriteLine("Некорректный email");
        }
    }
}
}
}
}

```

Переменная `pattern` задает регулярное выражение для проверки адреса электронной почты. Данное выражение предлагает нам Microsoft на страницах [msdn](https://msdn.microsoft.com/ru-ru/library/aa264737(v=vs.103).aspx).

Для проверки соответствия строки шаблону используется метод `IsMatch`: `Regex.IsMatch(email, pattern, RegexOptions.IgnoreCase)`. Последний параметр указывает, что регистр можно игнорировать. И если введенная строка соответствует шаблону, то метод возвращает `true`.

Замена и метод `Replace`

Класс `Regex` имеет метод `Replace`, который позволяет заменить строку, соответствующую регулярному выражению, другой строкой:

```

using System;
using System.Text.RegularExpressions;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = "Мама мыла раму. ";
            string pattern = @"\s+";
            string target = " ";
            Regex regex = new Regex(pattern);
            string result = regex.Replace(s, target);

            Console.WriteLine(result);

            Console.ReadLine();
        }
    }
}

```

Данная версия метода `Replace` принимает два параметра: строку с текстом, где надо выполнить замену, и сама строка замены. Так как в качестве шаблона выбрано выражение `"\s+"` (то есть наличие одного и более пробелов), метод `Replace` проходит по всему тексту и заменяет несколько подряд идущих пробелов одинарными.

Задание 2. Найти самое длинное слово в введенной строке и вывести его на консоль. Предполагается, что все слова разделены пробелами.

```

using System;

```

```

using System.Text.RegularExpressions;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Введите строку: ");
            string input = Console.ReadLine();
            string[] words = input.Split(new char[] { ' ' });

            int maxLength = 0,
                index = 0;

            for(int i = 0; i < words.Length; i++)
            {
                if (words[i].Length > maxLength)
                {
                    maxLength = words[i].Length;
                    index = i;
                }
            }

            Console.WriteLine(words[index]);

            Console.ReadLine();
        }
    }
}

```

Работа с потоками и файловой системой

Большинство задач в программировании так или иначе связаны с работой с файлами и каталогами. Нам может потребоваться прочитать текст из файла или наоборот произвести запись, удалить файл или целый каталог, не говоря уже о более комплексных задачах, как например, создание текстового редактора и других подобных задачах.

Фреймворк .NET предоставляет большие возможности по управлению и манипуляции файлами и каталогами, которые по большей части сосредоточены в пространстве имен System.IO. Классы, расположенные в этом пространстве имен (такие как Stream, StreamWriter, FileStream и др.), позволяют управлять файловым вводом-выводом.

Работа с дисками

Работу с файловой системой начнем с самого верхнего уровня - дисков. Для представления диска в пространстве имен System.IO имеется класс DriveInfo.

Этот класс имеет статический метод GetDrives, который возвращает имена всех логических дисков компьютера. Также он предоставляет ряд полезных свойств:

- AvailableFreeSpace: указывает на объем доступного свободного места на диске в байтах
- DriveFormat: получает имя файловой системы
- DriveType: представляет тип диска

- IsReady: готов ли диск (например, DVD-диск может быть не вставлен в дисковод)
- Name: получает имя диска
- TotalFreeSpace: получает общий объем свободного места на диске в байтах
- TotalSize: общий размер диска в байтах
- VolumeLabel: получает или устанавливает метку тома

Получим имена и свойства всех дисков на компьютере:

```
using System;
using System.IO;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            DriveInfo[] drives = DriveInfo.GetDrives();

            foreach (DriveInfo drive in drives)
            {
                Console.WriteLine("Название: {0}", drive.Name);
                Console.WriteLine("Тип: {0}", drive.DriveType);
                if (drive.IsReady)
                {
                    Console.WriteLine("Объем диска: {0} Мб", drive.TotalSize /
(1024*1024));
                    Console.WriteLine("Свободное пространство: {0} Мб",
drive.TotalFreeSpace / (1024 * 1024));
                    Console.WriteLine("Метка: {0}", drive.VolumeLabel);
                }
                Console.WriteLine();
            }

            Console.ReadLine();
        }
    }
}
```

Работа с каталогами

Для работы с каталогами в пространстве имен System.IO предназначены сразу два класса: Directory и DirectoryInfo.

Класс Directory

Класс Directory предоставляет ряд статических методов для управления каталогами. Некоторые из этих методов:

- CreateDirectory(path): создает каталог по указанному пути path
- Delete(path): удаляет каталог по указанному пути path
- Exists(path): определяет, существует ли каталог по указанному пути path. Если существует, возвращается true, если не существует, то false
- GetDirectories(path): получает список каталогов в каталоге path
- GetFiles(path): получает список файлов в каталоге path
- Move(sourceDirName, destDirName): перемещает катало
- GetParent(path): получение родительского каталога

Класс DirectoryInfo

Данный класс предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Во многом он похож на Directory. Некоторые из его свойств и методов:

- Create(): создает каталог
- CreateSubdirectory(path): создает подкаталог по указанному пути path
- Delete(): удаляет каталог
- Свойство Exists: определяет, существует ли каталог
- GetDirectories(): получает список каталогов
- GetFiles(): получает список файлов
- MoveTo(destDirName): перемещает каталог
- Свойство Parent: получение родительского каталога
- Свойство Root: получение корневого каталога

Посмотрим на примерах применение этих классов.

Получение списка файлов и подкаталогов

```
using System;
using System.IO;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Введите букву диска:");
            string dirName = Console.ReadLine();//"E:\\";

            dirName = dirName.ToUpper() + ":\\";

            if (Directory.Exists(dirName))
            {
                Console.WriteLine("Каталоги:");
                Console.WriteLine();
                string[] dirs = Directory.GetDirectories(dirName);
                foreach (string s in dirs)
                {
                    Console.WriteLine("  {0}", s);
                }
                Console.WriteLine();
                Console.WriteLine("Файлы:");
                Console.WriteLine();
                string[] files = Directory.GetFiles(dirName);
                foreach (string s in files)
                {
                    Console.WriteLine("  {0}", s);
                }
            }
            else
            {
                Console.WriteLine("Диска {0} не существует!", dirName);
            }

            Console.ReadLine();
        }
    }
}
```

```
}
```

Обратите внимание на использование слешей в именах файлов. Либо мы используем двойной слеш: "C:\\", либо одинарный, но тогда перед всем путем ставим знак @: @"C:\Program Files"

Создание каталога

```
using System;
using System.IO;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"E:\SomeDir";
            string subpath = @"new folder";
            DirectoryInfo dirInfo = new DirectoryInfo(path);
            if (!dirInfo.Exists)
            {
                dirInfo.Create();
            }
            dirInfo.CreateSubdirectory(subpath);

            Console.ReadLine();
        }
    }
}
```

Вначале проверяем, а нету ли такой директории, так как если она существует, то ее создать будет нельзя, и приложение выбросит ошибку. В итоге у нас получится следующий путь: "C:\SomeDir\new folder "

Получение информации о каталоге

```
using System;
using System.IO;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string dirName = "E:\\Program Files";

            DirectoryInfo dirInfo = new DirectoryInfo(dirName);

            Console.WriteLine("Название каталога: {0}", dirInfo.Name);
            Console.WriteLine("Полное название каталога: {0}", dirInfo.FullName);
            Console.WriteLine("Время создания каталога: {0}", dirInfo.CreationTime);
            Console.WriteLine("Корневой каталог: {0}", dirInfo.Root);

            Console.ReadLine();
        }
    }
}
```

Удаление каталога

Если мы просто применим метод Delete к непустой папке, в которой есть какие-нибудь файлы или подкаталоги, то приложение нам выбросит ошибку. Поэтому нам надо передать в метод Delete дополнительный параметр булевого типа, который укажет, что папку надо удалять со всем содержимым:

```
using System;
using System.IO;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string dirName = @"E:\SomeFolder";

            try
            {
                DirectoryInfo dirInfo = new DirectoryInfo(dirName);
                dirInfo.Delete(true);
                Console.WriteLine("Удаление каталога прошло успешно");
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }

            Console.ReadLine();
        }
    }
}
```

Либо если делать это через Directory:

```
using System;
using System.IO;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string dirName = @"E:\SomeFolder";

            Directory.Delete(dirName, true);

            Console.ReadLine();
        }
    }
}
```

Перемещение каталога

```
using System;
using System.IO;

namespace ConsoleApp
{
    class Program
```

```

{
    static void Main(string[] args)
    {
        string oldPath = @"E:\SomeFolder";
        string newPath = @"E:\SomeDir";
        DirectoryInfo dirInfo = new DirectoryInfo(oldPath);
        if (dirInfo.Exists && Directory.Exists(newPath) == false)
        {
            dirInfo.MoveTo(newPath);
            Console.WriteLine("Каталог успешно перемещен!");
        }
        else
        {
            Console.WriteLine("Каталога {0} не существует", oldPath);
        }

        Console.ReadLine();
    }
}

```

При перемещении надо учитывать, что новый каталог, в который мы хотим переместить все содержимое старого каталога, не должен существовать.

Работа с файлами. Классы File и FileInfo

Подобно паре Directory/DirectoryInfo для работы с файлами предназначена пара классов File и FileInfo. С их помощью мы можем создавать, удалять, перемещать файлы, получать их свойства и многое другое.

Некоторые полезные методы и свойства класса FileInfo:

- CopyTo(path): копирует файл в новое место по указанному пути path
- Create(): создает файл
- Delete(): удаляет файл
- MoveTo(destFileName): перемещает файл в новое место
- Свойство Directory: получает родительский каталог в виде объекта DirectoryInfo
- Свойство DirectoryName: получает полный путь к родительскому каталогу
- Свойство Exists: указывает, существует ли файл
- Свойство Length: получает размер файла
- Свойство Extension: получает расширение файла
- Свойство Name: получает имя файла
- Свойство FullName: получает полное имя файла

Класс File реализует похожую функциональность с помощью статических методов:

- Copy(): копирует файл в новое место
- Create(): создает файл
- Delete(): удаляет файл
- Move: перемещает файл в новое место
- Exists(file): определяет, существует ли файл

Получение информации о файле

```

using System;
using System.IO;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"E:\example\1.txt";
            FileInfo fileInf = new FileInfo(path);
            if (fileInf.Exists)
            {
                Console.WriteLine("Имя файла: {0}", fileInf.Name);
                Console.WriteLine("Время создания: {0}", fileInf.CreationTime);
                Console.WriteLine("Размер: {0}", fileInf.Length);
            }
            else
            {
                Console.WriteLine("Файл не существует");
            }

            Console.ReadLine();
        }
    }
}

```

Удаление файла

```

using System;
using System.IO;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"E:\example\1.txt";
            FileInfo fileInf = new FileInfo(path);
            if (fileInf.Exists)
            {
                fileInf.Delete();
                // альтернатива с помощью класса File
                //File.Delete(path);
            }
            else
            {
                Console.WriteLine("Файл не существует");
            }

            Console.ReadLine();
        }
    }
}

```

Перемещение файла

```

using System;
using System.IO;

namespace ConsoleApp

```



```

{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"E:\example\1.txt";
            string newPath = @"E:\example2\1.txt"; // папка должна существовать
            FileInfo fileInf = new FileInfo(path);
            if (fileInf.Exists)
            {
                fileInf.MoveTo(newPath);
                // альтернатива с помощью класса File
                // File.Move(path, newPath);
            }
            else
            {
                Console.WriteLine("Файл не существует");
            }

            Console.ReadLine();
        }
    }
}

```

Директория нового пути должна существовать.

Копирование файла

```

using System;
using System.IO;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"E:\example\1.txt";
            string newPath = @"E:\example2\1.txt"; // папка должна существовать
            FileInfo fileInf = new FileInfo(path);
            if (fileInf.Exists)
            {
                fileInf.CopyTo(newPath, true);
                // альтернатива с помощью класса File
                // File.Copy(path, newPath, true);
            }
            else
            {
                Console.WriteLine("Файл не существует");
            }

            Console.ReadLine();
        }
    }
}

```

Метод `CopyTo` класса `FileInfo` принимает два параметра: путь, по которому файл будет копироваться, и булево значение, которое указывает, надо ли при копировании перезаписывать файл (если `true`, как в случае выше, файл при копировании перезаписывается). Если же в качестве последнего параметра

передать значение false, то если такой файл уже существует, приложение выдаст ошибку.

Метод Copy класса File принимает три параметра: путь к исходному файлу, путь, по которому файл будет копироваться, и булево значение, указывающее, будет ли файл перезаписываться.

Чтение и запись файла. Класс FileStream

Класс FileStream представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Рассмотрим наиболее важные его свойства и методы:

- Свойство Length: возвращает длину потока в байтах
- Свойство Position: возвращает текущую позицию в потоке
- Метод Read: считывает данные из файла в массив байтов. Принимает три параметра: `int Read(byte[] array, int offset, int count)` и возвращает количество успешно считанных байтов. Здесь используются следующие параметры:
 - `array` - массив байтов, куда будут помещены считываемые из файла данные
 - `offset` представляет смещение в байтах в массиве `array`, в который считанные байты будут помещены
 - `count` - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.
- Метод `long Seek(long offset, SeekOrigin origin)`: устанавливает позицию в потоке со смещением на количество байт, указанных в параметре `offset`.
- Метод Write: записывает в файл данные из массива байтов. Принимает три параметра: `Write(byte[] array, int offset, int count)`
 - `array` - массив байтов, откуда данные будут записываться в файл
 - `offset` - смещение в байтах в массиве `array`, откуда начинается запись байтов в поток
 - `count` - максимальное число байтов, предназначенных для записи

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если вам надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Поэтому для работы с текстовыми файлами применяются другие классы.

В то же время при работе с различными бинарными файлами, имеющими определенную структуру FileStream может быть очень даже полезен для извлечения определенных порций информации и ее обработки.

Посмотрим на примере считывания-записи в текстовый файл:

```
using System;
using System.IO;

namespace ConsoleApp
{
    class Program
    {
```

```

static void Main(string[] args)
{
    string filePath = @"E:\example\1.txt";
    Console.WriteLine("Введите строку для записи в файл:");
    string text = Console.ReadLine();

    // запись в файл
    using (FileStream fstream = new FileStream(filePath, FileMode.OpenOrCreate))
    {
        // преобразуем строку в байты
        byte[] array = System.Text.Encoding.Default.GetBytes(text);
        // запись массива байтов в файл
        fstream.Write(array, 0, array.Length);
        Console.WriteLine("Текст успешно записан в файл");
    }

    // чтение из файла
    using (FileStream fstream = File.OpenRead(filePath))
    {
        // преобразуем строку в байты
        byte[] array = new byte[fstream.Length];
        // считываем данные
        fstream.Read(array, 0, array.Length);
        // декодируем байты в строку
        string textFromFile = System.Text.Encoding.Default.GetString(array);
        Console.WriteLine("Текст из файла: {0}", textFromFile);
    }

    Console.ReadLine();
}
}
}

```

Разберем этот пример. И при чтении, и при записи используется оператор `using`. Не надо путать данный оператор с директивой `using`, которая подключает пространства имен в начале файла кода. Оператор `using` позволяет создавать объект в блоке кода, по завершению которого вызывается метод `Dispose` у этого объекта, и, таким образом, объект уничтожается. В данном случае в качестве такого объекта служит переменная `fstream`.

Объект `fstream` создается двумя разными способами: через конструктор и через один из статических методов класса `File`.

Здесь в конструктор передается два параметра: путь к файлу и перечисление `FileMode`. Данное перечисление указывает на режим доступа к файлу и может принимать следующие значения:

- `Append`: если файл существует, то текст добавляется в конец файл. Если файла нет, то он создается. Файл открывается только для записи.
- `Create`: создается новый файл. Если такой файл уже существует, то он перезаписывается
- `CreateNew`: создается новый файл. Если такой файл уже существует, то он приложение выбрасывает ошибку
- `Open`: открывает файл. Если файл не существует, выбрасывается исключение
- `OpenOrCreate`: если файл существует, он открывается, если нет - создается новый

- Truncate: если файл существует, то он перезаписывается. Файл открывается только для записи.

Статический метод `OpenRead` класса `File` открывает файл для чтения и возвращает объект `FileStream`.

Конструктор класса `FileStream` также имеет ряд перегруженных версий, позволяющий более точно настроить создаваемый объект. Все эти версии можно посмотреть на `msdn`.

И при записи, и при чтении применяется объект кодировки `Encoding.Default` из пространства имен `System.Text`. В данном случае мы используем два его метода: `GetBytes` для получения массива байтов из строки и `GetString` для получения строки из массива байтов.

В итоге введенная нами строка записывается в файл `note.txt`. По сути это бинарный файл (не текстовый), хотя если мы в него запишем только строку, то сможем посмотреть в удобочитаемом виде этот файл, открыв его в текстовом редакторе. Однако если мы в него запишем случайные байты, например:

```
fstream.WriteByte(13);
```

То у нас могут возникнуть проблемы с его пониманием. Поэтому для работы непосредственно с текстовыми файлами предназначены отдельные классы - `StreamReader` и `StreamWriter`.

Произвольный доступ к файлам

Нередко бинарные файлы представляют определенную структуру. И, зная эту структуру, мы можем взять из файла нужную порцию информации или наоборот записать в определенном месте файла определенный набор байтов. Например, в `wav`-файлах непосредственно звуковые данные начинаются с 44 байта, а до 44 байта идут различные метаданные - количество каналов аудио, частота дискретизации и т.д.

С помощью метода `Seek()` мы можем управлять положением курсора потока, начиная с которого производится считывание или запись в файл. Этот метод принимает два параметра: `offset` (смещение) и позиция в файле. Позиция в файле описывается тремя значениями:

- `SeekOrigin.Begin`: начало файла
- `SeekOrigin.End`: конец файла
- `SeekOrigin.Current`: текущая позиция в файле

Курсор потока, с которого начинается чтение или запись, смещается вперед на значение `offset` относительно позиции, указанной в качестве второго параметра. Смещение может отрицательным, тогда курсор сдвигается назад, если положительное - то вперед.

Рассмотрим на примере:

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

```

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string filePath = @"E:\example\note.dat";
            /*Console.WriteLine("Введите строку:");
            string text = Console.ReadLine();*/
            string text = "hello world";

            // запись в файл
            using (FileStream fstream = new FileStream(filePath, FileMode.OpenOrCreate))
            {
                // преобразуем строку в байты
                byte[] input = Encoding.Default.GetBytes(text);
                // запись массива байтов в файл
                fstream.Write(input, 0, input.Length);
                Console.WriteLine("Текст записан в файл");

                // перемещаем указатель в конец файла, до конца файла- пять байт
                fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца потока

                // считываем четыре символа с текущей позиции
                byte[] output = new byte[4];
                fstream.Read(output, 0, output.Length);
                // декодируем байты в строку
                string textFromFile = Encoding.Default.GetString(output);
                Console.WriteLine("Текст из файла: {0}", textFromFile); // worl

                // заменим в файле слово world на слово house
                string replaceText = "house";
                fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца потока
                input = Encoding.Default.GetBytes(replaceText);
                fstream.Write(input, 0, input.Length);

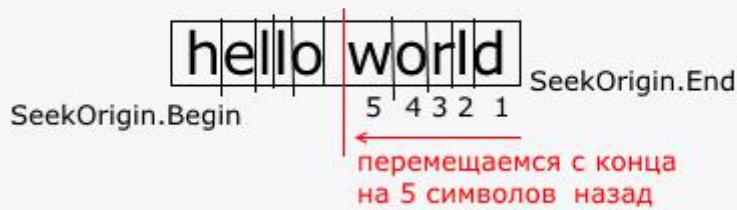
                // считываем весь файл
                // возвращаем указатель в начало файла
                fstream.Seek(0, SeekOrigin.Begin);
                output = new byte[fstream.Length];
                fstream.Read(output, 0, output.Length);
                // декодируем байты в строку
                textFromFile = Encoding.Default.GetString(output);
                Console.WriteLine("Текст из файла: {0}", textFromFile); // hello house
            }

            Console.ReadLine();
        }
    }
}

```

Вызов `fstream.Seek(-5, SeekOrigin.End)` перемещает курсор потока в конец файлов назад на пять символов:

`fstream.Seek(-5, SeekOrigin.End)`



То есть после записи в новый файл строки "hello world" курсор будет стоять на позиции символа "w".

После этого считываем четыре байта начиная с символа "w". В данной кодировке 1 символ будет представлять 1 байт. Поэтому чтение 4 байтов будет эквивалентно чтению четырех символов: "worl".

Затем опять же перемещаемся в конец файла, не доходя до конца пять символов (то есть опять же с позиции символа "w"), и осуществляем запись строки "house". Таким образом, строка "house" заменяет строку "world".

Заккрытие потока

В примерах выше для закрытия потока применяется конструкция `using`. После того как все операторы и выражения в блоке `using` отработают, объект `FileStream` уничтожается. Однако мы можем выбрать и другой способ:

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

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream fstream = null;
            try
            {
                fstream = new FileStream(@"E:\example\note.dat", FileMode.OpenOrCreate);
                // операции с потоком
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                if (fstream != null)
                    fstream.Close();
            }

            Console.ReadLine();
        }
    }
}
```

Чтение и запись текстовых файлов. StreamReader и StreamWriter

Класс FileStream не очень удобно применять для работы с текстовыми файлами. К тому же для этого в пространстве System.IO определены специальные классы: StreamReader и StreamWriter.

Чтение из файла и StreamReader

Класс StreamReader позволяет нам легко считывать весь текст или отдельные строки из текстового файла. Среди его методов можно выделить следующие:

- Close: закрывает считываемый файл и освобождает все ресурсы
- Peek: возвращает следующий доступный символ, если символов больше нет, то возвращает -1
- Read: считывает и возвращает следующий символ в численном представлении. Имеет перегруженную версию: Read(char[] array, int index, int count), где array - массив, куда считываются символы, index - индекс в массиве array, начиная с которого записываются считываемые символы, и count - максимальное количество считываемых символов
- ReadLine: считывает одну строку в файле
- ReadToEnd: считывает весь текст из файла

Считаем текст из файла различными способами:

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

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"E:\example\1.txt";

            try
            {
                Console.WriteLine("*****считываем весь файл*****");
                using (StreamReader sr = new StreamReader(path))
                {
                    Console.WriteLine(sr.ReadToEnd());
                }

                Console.WriteLine();
                Console.WriteLine("*****считываем построчно*****");
                using (StreamReader sr = new StreamReader(path,
                    System.Text.Encoding.Default))
                {
                    string line;
                    while ((line = sr.ReadLine()) != null)
                    {
                        Console.WriteLine(line);
                    }
                }

                Console.WriteLine();
                Console.WriteLine("*****считываем блоками*****");
```

```

        using (StreamReader sr = new StreamReader(path,
System.Text.Encoding.Default))
        {
            char[] array = new char[4];
            // считываем 4 символа
            sr.Read(array, 0, 4);

            Console.WriteLine(array);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }

    Console.ReadLine();
}
}
}

```

Как и в случае с классом `FileStream` здесь используется конструкция `using`.

В первом случае мы разом считываем весь текст с помощью метода `ReadToEnd()`.

Во втором случае считываем построчно через цикл `while`: `while ((line = sr.ReadLine()) != null)` - сначала присваиваем переменной `line` результат функции `sr.ReadLine()`, а затем проверяем, не равна ли она `null`. Когда объект `sr` дойдет до конца файла и больше строк не останется, то метод `sr.ReadLine()` будет возвращать `null`.

В третьем случае считываем в массив четыре символа.

Обратите внимание, что в последних двух случаях в конструкторе `StreamReader` указывалась кодировка `System.Text.Encoding.Default`. Свойство `Default` класса `Encoding` получает кодировку для текущей кодовой страницы ANSI. Также через другие свойства мы можем указать другие кодировки. Если кодировка не указана, то при чтении используется UTF8. Иногда важно указывать кодировку, так как она может отличаться от UTF8, и тогда мы получим некорректный вывод.

Запись в файл и `StreamWriter`

Для записи в текстовый файл используется класс `StreamWriter`. Свою функциональность он реализует через следующие методы:

- `Close`: закрывает записываемый файл и освобождает все ресурсы
- `Flush`: записывает в файл оставшиеся в буфере данные и очищает буфер.
- `Write`: записывает в файл данные простейших типов, как `int`, `double`, `char`, `string` и т.д.
- `WriteLine`: также записывает данные, только после записи добавляет в файл символ окончания строки

Рассмотрим запись в файл на примере:

```

using System;
using System.IO;

namespace ConsoleApp

```



```

{
    class Program
    {
        static void Main(string[] args)
        {
            string readPath = @"E:\example\1.txt";
            string writePath = @"E:\example\2.txt";

            string text = "";
            try
            {
                using (StreamReader sr = new StreamReader(readPath,
System.Text.Encoding.Default))
                {
                    text = sr.ReadToEnd();
                }
                using (StreamWriter sw = new StreamWriter(writePath, false,
System.Text.Encoding.Default))
                {
                    sw.WriteLine(text);
                }

                using (StreamWriter sw = new StreamWriter(writePath, true,
System.Text.Encoding.Default))
                {
                    sw.WriteLine("Дозапись");
                    sw.Write(4.5);
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }

            Console.ReadLine();
        }
    }
}

```

Здесь сначала мы считываем файл в переменную text, а затем записываем эту переменную в файл, а затем через объект StreamWriter записываем в новый файл.

Класс StreamWriter имеет несколько конструкторов. Здесь мы использовали один из них: new StreamWriter(writePath, false, System.Text.Encoding.Default). В качестве первого параметра передается путь к записываемому файлу. Второй параметр представляет булеву переменную, которая определяет, будет файл дозаписываться или перезаписываться. Если этот параметр равен true, то новые данные добавляются в конце к уже имеющимся данным. Если false, то файл перезаписывается. И если в первом случае файл перезаписывается, то во втором делается дозапись в конец файла.

Третий параметр указывает кодировку, в которой записывается файл.

Работа с бинарными файлами. BinaryWriter и BinaryReader

Для работы с бинарными файлами предназначена пара классов `BinaryWriter` и `BinaryReader`. Эти классы позволяют читать и записывать данные в двоичном формате.

Основные методы класса `BinaryWriter`:

- `Close()`: закрывает поток и освобождает ресурсы
- `Flush()`: очищает буфер, дописывая из него оставшиеся данные в файл
- `Seek()`: устанавливает позицию в потоке
- `Write()`: записывает данные в поток

Основные метода класса `BinaryReader`:

- `Close()`: закрывает поток и освобождает ресурсы
- `ReadBoolean()`: считывает значение `bool` и перемещает указатель на один байт
- `ReadByte()`: считывает один байт и перемещает указатель на один байт
- `ReadChar()`: считывает значение `char`, то есть один символ, и перемещает указатель на столько байтов, сколько занимает символ в текущей кодировке
- `ReadDecimal()`: считывает значение `decimal` и перемещает указатель на 16 байт
- `ReadDouble()`: считывает значение `double` и перемещает указатель на 8 байт
- `ReadInt16()`: считывает значение `short` и перемещает указатель на 2 байта
- `ReadInt32()`: считывает значение `int` и перемещает указатель на 4 байта
- `ReadInt64()`: считывает значение `long` и перемещает указатель на 8 байт
- `ReadSingle()`: считывает значение `float` и перемещает указатель на 4 байта
- `ReadString()`: считывает значение `string`. Каждая строка предваряется значением длины строки, которое представляет 7-битное целое число

С чтением бинарных данных все просто: соответствующий метод считывает данные определенного типа и перемещает указатель на размер этого типа в байтах, например, значение типа `int` занимает 4 байта, поэтому `BinaryReader` считывает 4 байта и переместит указатель на эти 4 байта.

Посмотрим на реальной задаче применение этих классов. Попробуем с их помощью записывать и считывать из файла массив структур:

```
using System;
using System.IO;

namespace ConsoleApp
{
    struct State
    {
        public string name;
        public string capital;
        public int area;
        public double people;

        public State(string n, string c, int a, double p)
        {
            name = n;
            capital = c;
            people = p;
            area = a;
        }
    }
}
```

```

    }

    class Program
    {
        static void Main(string[] args)
        {
            State[] states = new State[2];
            states[0] = new State("Германия", "Берлин", 357168, 80.8);
            states[1] = new State("Франция", "Париж", 640679, 64.7);

            string path = @"E:\example\states.dat";

            try
            {
                // создаем объект BinaryWriter
                using (BinaryWriter writer = new BinaryWriter(File.Open(path,
                FileMode.OpenOrCreate)))
                {
                    // записываем в файл значение каждого поля структуры
                    foreach (State s in states)
                    {
                        writer.Write(s.name);
                        writer.Write(s.capital);
                        writer.Write(s.area);
                        writer.Write(s.people);
                    }
                }
                // создаем объект BinaryReader
                using (BinaryReader reader = new BinaryReader(File.Open(path,
                FileMode.Open)))
                {
                    // пока не достигнут конец файла
                    // считываем каждое значение из файла
                    while (reader.PeekChar() > -1)
                    {
                        string name = reader.ReadString();
                        string capital = reader.ReadString();
                        int area = reader.ReadInt32();
                        double population = reader.ReadDouble();

                        Console.WriteLine("Страна: {0} столица: {1} площадь {2} кв. км
                        численность населения: {3} млн. чел.",
                        name, capital, area, population);
                    }
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }

            Console.ReadLine();
        }
    }
}

```

Итак, у нас есть структура State с некоторым набором полей. В основной программе создаем массив структур и записываем с помощью BinaryWriter. Этот класс в качестве параметра в конструкторе принимает объект Stream, который создается вызовом File.Open(path, FileMode.OpenOrCreate).

Затем в цикле пробегаемся по массиву структур и записываем каждое поле структуры в поток. В том порядке, в каком эти значения полей записываются, в том порядке они и будут размещаться в файле.

Затем считываем из записанного файла. Конструктор класса `BinaryReader` также в качестве параметра принимает объект потока, только в данном случае устанавливаем в качестве режима `FileMode.Open`: `new BinaryReader(File.Open(path, FileMode.Open))`

В цикле `while` считываем данные. Чтобы узнать окончание потока, вызываем метод `PeekChar()`. Этот метод считывает следующий символ и возвращает его числовое представление. Если символ отсутствует, то метод возвращает `-1`, что будет означать, что мы достигли конца файла.

В цикле последовательно считываем значения поле структур в том же порядке, в каком они записывались.

Таким образом, классы `BinaryWriter` и `BinaryReader` очень удобны для работы с бинарными файлами, особенно когда нам известна структура этих файлов. В то же время для хранения и считывания более сложных объектов, например, объектов классов, лучше подходит другое решение - сериализация.

Создание и чтение сжатых файлов. `GZipStream` и `DeflateStream`

Кроме классов чтения-записи .NET предоставляет классы, которые позволяют сжимать файлы, а также затем восстанавливать их в исходное состояние.

Это классы `DeflateStream` и `GZipStream`, которые находятся в пространстве имен `System.IO.Compression` и представляют реализацию одного из алгоритмов сжатия `Deflate` или `GZip`.

Рассмотрим применение класса `GZipStream` на примере:

```
using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApp
{
    class Program
    {
        public static void Compress(string sourceFile, string compressedFile)
        {
            // поток для чтения исходного файла
            using (FileStream sourceStream = new FileStream(sourceFile,
                FileMode.OpenOrCreate))
            {
                // поток для записи сжатого файла
                using (FileStream targetStream = File.Create(compressedFile))
                {
                    // поток архивации
                    using (GZipStream compressionStream = new GZipStream(targetStream,
                        CompressionMode.Compress))
                    {
                        sourceStream.CopyTo(compressionStream); // копируем байты из
                        // одного потока в другой
                        Console.WriteLine("Сжатие файла {0} завершено. Исходный размер:
                        {1} сжатый размер: {2}.",
```

```

        sourceFile, sourceStream.Length.ToString(),
        targetStream.Length.ToString());
    }
}

public static void Decompress(string compressedFile, string targetFile)
{
    // поток для чтения из сжатого файла
    using (FileStream sourceStream = new FileStream(compressedFile,
        FileMode.OpenOrCreate))
    {
        // поток для записи восстановленного файла
        using (FileStream targetStream = File.Create(targetFile))
        {
            // поток разархивации
            using (GZipStream decompressionStream = new GZipStream(sourceStream,
                CompressionMode.Decompress))
            {
                decompressionStream.CopyTo(targetStream);
                Console.WriteLine("Восстановлен файл: {0}", targetFile);
            }
        }
    }
}

static void Main(string[] args)
{
    string sourceFile = "E://example/book.pdf"; // исходный файл
    string compressedFile = "E://example/book.gz"; // сжатый файл
    string targetFile = "E://example/book_new.pdf"; // восстановленный файл

    // создание сжатого файла
    Compress(sourceFile, compressedFile);
    // чтение из сжатого файла
    Decompress(compressedFile, targetFile);

    Console.ReadLine();
}
}

```

Метод Compress получает название исходного файла, который надо архивировать, и название будущего сжатого файла.

Сначала создается поток для чтения из исходного файла - FileStream sourceStream. Затем создается поток для записи в сжатый файл - FileStream targetStream. Поток архивации GZipStream compressionStream инициализируется потоком targetStream и с помощью метода CopyTo() получает данные от потока sourceStream.

Метод Decompress производит обратную операцию по восстановлению сжатого файла в исходное состояние. Он принимает в качестве параметров пути к сжатому файлу и будущему восстановленному файлу.

Здесь в начале создается поток для чтения из сжатого файла FileStream sourceStream, затем поток для записи в восстанавливаемый файл FileStream targetStream. В конце создается поток GZipStream decompressionStream, который с помощью метода CopyTo() копирует восстановленные данные в поток targetStream.

Чтобы указать потоку GZipStream, для чего именно он предназначен - сжатия или восстановления - ему в конструктор передается параметр CompressionMode, принимающий два значения: Compress и Decompress.

Если бы захотели бы использовать другой класс сжатия - DeflateStream, то мы могли бы просто заменить в коде упоминания GZipStream на DeflateStream, без изменения остального кода. Их использование идентично.

В то же время при использовании этих классов есть некоторые ограничения, в частности, мы можем сжимать только один файл. Для архивации группы файлы лучше выбрать другие инструменты.

Задание 3. Написать программу для удаления комментариев и тегов из html-файла и обеспечить правильную подстановку для знаков <, >, &, “. Результат должен быть записан в новый файл.

Сериализация

Введение в сериализацию объектов

Ранее мы рассмотрели, как сохранять информацию в текстовые файлы, а также затронули сохранение несложных структур в бинарные данные. Но нередко подобных механизмов оказывается недостаточно особенно для сохранения сложных объектов. С этой проблемой призван справиться механизм сериализации. Сериализация представляет процесс преобразования какого-либо объекта в поток байтов. После преобразования мы можем этот поток байтов или записать на диск или сохранить его временно в памяти. А при необходимости можно выполнить обратный процесс - десериализацию, то есть получить из потока байтов ранее сохраненный объект.

Атрибут Serializable

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

```
[Serializable]
class Person
{
    public string Name { get; set; }
    public int Year { get; set; }

    public Person(string name, int year)
    {
        Name = name;
        Year = year;
    }
}
```

При отсутствии данного атрибута объект Person не сможет быть сериализован, и при попытке сериализации будет выброшено исключение `SerializationException`.

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

```
[NonSerialized]
public string accNumber;
```

При наследовании подобного класса, следует учитывать, что атрибут `Serializable` автоматически не наследуется. И если мы хотим, чтобы производный класс также мог бы быть сериализован, то опять же мы применяем к нему атрибут:

```
[Serializable]
class Worker : Person
```

Формат сериализации

Хотя сериализация представляет собой преобразование объекта в некоторый набор байтов, но в действительности только бинарным форматом она не ограничивается. Итак, в .NET можно использовать следующие форматы:

- бинарный
- SOAP
- xml
- JSON

Для каждого формата предусмотрен свой класс: для сериализации в бинарный формат - класс `BinaryFormatter`, для формата SOAP - класс `SoapFormatter`, для xml - `XmlSerializer`, для json - `DataContractJsonSerializer`.

Для классов `BinaryFormatter` и `SoapFormatter` сам функционал сериализации определен в интерфейсе `IFormatter`:

```
public interface IFormatter
{
    SerializationBinder Binder { get; set; }
    StreamingContext Context { get; set; }
    ISurrogateSelector SurrogateSelector { get; set; }
    object Deserialize(Stream serializationStream);
    void Serialize(Stream serializationStream, object graph);
}
```

Хотя классы `BinaryFormatter` и `SoapFormatter` по-разному реализуют данный интерфейс, но общий функционал будет тот же: для сериализации будет использоваться метод `Serialize`, который в качестве параметров принимает поток, куда помещает сериализованные данные (например, бинарный файл), и объект, который надо сериализовать. А для десериализации будет применяться метод `Deserialize`, который в качестве параметра принимает поток с сериализованными данными.

Класс XmlSerializer не реализует интерфейс IFormatter и по функциональности в целом несколько отличается от BinaryFormatter и SoapFormatter, но и он также предоставляет для сериализации метод Serialize, а для десериализации Deserialize. И в этом плане очень легко при необходимости перейти от одного способа сериализации к другому.

Бинарная сериализация. BinaryFormatter

Для бинарной сериализации применяется класс BinaryFormatter:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace ConsoleApp
{
    [Serializable]
    class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public Person(string name, int age)
        {
            Name = name;
            Age = age;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // объект для сериализации
            Person person = new Person("Том", 29);
            Console.WriteLine("Объект создан");

            // создаем объект BinaryFormatter
            BinaryFormatter formatter = new BinaryFormatter();
            // получаем поток, куда будем записывать сериализованный объект
            using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))
            {
                formatter.Serialize(fs, person);

                Console.WriteLine("Объект сериализован");
            }

            // десериализация из файла people.dat
            using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))
            {
                Person newPerson = (Person)formatter.Deserialize(fs);

                Console.WriteLine("Объект десериализован");
                Console.WriteLine("Имя: {0} --- Возраст: {1}", newPerson.Name,
newPerson.Age);
            }

            Console.ReadLine();
        }
    }
}
```


Так как класс `BinaryFormatter` определен в пространстве имен `System.Runtime.Serialization.Formatters.Binary`, то в самом начале подключаем его.

У нас есть простенький класс `Person`, который объявлен с атрибутом `Serializable`. Благодаря этому его объекты будут доступны для сериализации.

Далее создаем объект `BinaryFormatter`: `BinaryFormatter formatter = new BinaryFormatter();`

Затем последовательно выполняем сериализацию и десериализацию. Для обеих операций нам нужен поток, в который либо сохранять, либо из которого считывать данные. Данный поток представляет объект `FileStream`, который записывает нужный нам объект `Person` в файл `people.dat`.

Сериализация одним методом `formatter.Serialize(fs, person)` добавляет все данные об объекте `Person` в файл `people.dat`.

При десериализации нам нужно еще преобразовать объект, возвращаемый функцией `Deserialize`, к типу `Person`: `(Person)formatter.Deserialize(fs)`.

Как вы видите, сериализация значительно упрощает процесс сохранения объектов в бинарную форму по сравнению, например, с использованием связки классов `BinaryWriter/BinaryReader`.

Хотя мы взяли лишь один объект `Person`, но равным образом мы можем использовать и массив подобных объектов, список или иную коллекцию, к которой применяется атрибут `Serializable`. Посмотрим на примере массива:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace ConsoleApp
{
    [Serializable]
    class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public Person(string name, int age)
        {
            Name = name;
            Age = age;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Массив для сериализации
            Person person1 = new Person("Tom", 29);
            Person person2 = new Person("Bill", 25);

            Person[] people = new Person[] { person1, person2 };
            Console.WriteLine("Массив создан");

            // создаем объект BinaryFormatter
            BinaryFormatter formatter = new BinaryFormatter();
```

```

        // получаем поток, куда будем записывать сериализованный объект
        using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))
        {
            formatter.Serialize(fs, people);

            Console.WriteLine("Массив сериализован");
        }

        // десериализация из файла people.dat
        using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))
        {
            Person[] deserilizePeople = (Person[])formatter.Deserialize(fs);

            Console.WriteLine("Массив десериализован");

            foreach (Person p in deserilizePeople)
            {
                Console.WriteLine("Имя: {0} --- Возраст: {1}", p.Name, p.Age);
            }
        }

        Console.ReadLine();
    }
}

```

Протокол SOAP (Simple Object Access Protocol) представляет простой протокол для обмена данными между различными платформами. При такой сериализации данные упакуются в конверт SOAP, данные в котором имеют вид xml-подобного документа. Посмотрим на примере.

Прежде чем использовать класс SoapFormatter, нам надо добавить в проект сборку System.Runtime.Serialization.Formatters.Soap.dll. После этого нам станет доступным функциональность SoapFormatter:

```

using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;

namespace ConsoleApp
{
    [Serializable]
    class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public Person(string name, int age)
        {
            Name = name;
            Age = age;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Массив для сериализации
            Person person1 = new Person("Tom", 29);
            Person person2 = new Person("Bill", 25);
        }
    }
}

```

```

Person[] people = new Person[] { person1, person2 };
Console.WriteLine("Массив создан");

// создаем объект SoapFormatter
SoapFormatter formatter = new SoapFormatter();

// получаем поток, куда будем записывать сериализованный объект
using (FileStream fs = new FileStream("people.soap", FileMode.OpenOrCreate))
{
    formatter.Serialize(fs, people);

    Console.WriteLine("Массив сериализован");
}

// десериализация из файла people.dat
using (FileStream fs = new FileStream("people.soap", FileMode.OpenOrCreate))
{
    Person[] deserilizePeople = (Person[])formatter.Deserialize(fs);

    Console.WriteLine("Массив десериализован");

    foreach (Person p in deserilizePeople)
    {
        Console.WriteLine("Имя: {0} --- Возраст: {1}", p.Name, p.Age);
    }
}

Console.ReadLine();
}
}
}

```

Принцип использования SoapFormatter похож на рассмотренную в прошлой теме бинарную сериализацию. Здесь также создается поток, записывающий данные в файл people.soap. Для сериализации используется метод `formatter.Serialize(fs, people)`, использующий поток и объект для сериализации.

При десериализации считываем ранее сохраненные объекты и преобразуем их к нужному нам объекту в методе `Deserialize`: `Person[] newPeople = (Person[])formatter.Deserialize(fs)`

После сериализации все данные будут сохранены в файл people.soap.

Сериализация в XML. XmlSerializer

Для сериализации объектов в файлы xml используется класс `XmlSerializer`. Он стоит несколько особняком от других ранее рассмотренных классов сериализаций, поэтому работа с ним будет немного отличаться.

Во-первых, `XmlSerializer` предполагает некоторые ограничения. Например, класс, подлежащий сериализации, должен иметь стандартный конструктор без параметров. Также сериализации подлежат только открытые члены. Если в классе есть поля или свойства с модификатором `private`, то при сериализации они будут игнорироваться.

Во-вторых, `XmlSerializer` требует указания типа:

```

using System;
using System.IO;
using System.Xml.Serialization;

```

```

namespace ConsoleApp
{
    [Serializable]
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }

        // стандартный конструктор без параметров
        public Person()
        { }

        public Person(string name, int age)
        {
            Name = name;
            Age = age;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // объект для сериализации
            Person person = new Person("Том", 29);
            Console.WriteLine("Объект создан");

            // передаем в конструктор тип класса
            XmlSerializer formatter = new XmlSerializer(typeof(Person));

            // получаем поток, куда будем записывать сериализованный объект
            using (FileStream fs = new FileStream("persons.xml", FileMode.OpenOrCreate))
            {
                formatter.Serialize(fs, person);

                Console.WriteLine("Объект сериализован");
            }

            // десериализация
            using (FileStream fs = new FileStream("persons.xml", FileMode.OpenOrCreate))
            {
                Person newPerson = (Person)formatter.Deserialize(fs);

                Console.WriteLine("Объект десериализован");
                Console.WriteLine("Имя: {0} --- Возраст: {1}", newPerson.Name,
newPerson.Age);
            }

            Console.ReadLine();
        }
    }
}

```

Итак, класс Person общедоступный и имеет общедоступные свойства, поэтому он может сериализоваться. При создании объекта XmlSerializer передаем в конструктор тип класса.

И, как и с другими классами-сериализаторами, метод Serialize добавляет данные в файл persons.xml. А метод Deserialize извлекает их оттуда.

Равным образом мы можем сериализовать массив или коллекцию объектов, но главное требование состоит в том, чтобы в них был определен стандартный конструктор:

```
using System;
using System.IO;
using System.Xml.Serialization;

namespace ConsoleApp
{
    [Serializable]
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }

        // стандартный конструктор без параметров
        public Person()
        { }

        public Person(string name, int age)
        {
            Name = name;
            Age = age;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Person("Tom", 29);
            Person person2 = new Person("Bill", 25);
            Person[] people = new Person[] { person1, person2 };

            XmlSerializer formatter = new XmlSerializer(typeof(Person[]));

            using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))
            {
                formatter.Serialize(fs, people);
            }

            using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))
            {
                Person[] newpeople = (Person[])formatter.Deserialize(fs);

                foreach (Person p in newpeople)
                {
                    Console.WriteLine("Имя: {0} --- Возраст: {1}", p.Name, p.Age);
                }
            }

            Console.ReadLine();
        }
    }
}
```

Но это был простой объект. Однако с более сложными по составу объектами работать так же просто. Например:

```
using System;
using System.IO;
using System.Xml.Serialization;
```

```

namespace ConsoleApp
{
    [Serializable]
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
        public Company Company { get; set; }

        public Person()
        { }

        public Person(string name, int age, Company comp)
        {
            Name = name;
            Age = age;
            Company = comp;
        }
    }

    [Serializable]
    public class Company
    {
        public string Name { get; set; }

        // стандартный конструктор без параметров
        public Company() { }

        public Company(string name)
        {
            Name = name;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Person("Tom", 29, new Company("Microsoft"));
            Person person2 = new Person("Bill", 25, new Company("Apple"));
            Person[] people = new Person[] { person1, person2 };

            XmlSerializer formatter = new XmlSerializer(typeof(Person[]));

            using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))
            {
                formatter.Serialize(fs, people);
            }

            using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))
            {
                Person[] newpeople = (Person[])formatter.Deserialize(fs);

                foreach (Person p in newpeople)
                {
                    Console.WriteLine("Имя: {0} --- Возраст: {1} --- Компания: {2}",
p.Name, p.Age, p.Company.Name);
                }

                Console.ReadLine();
            }
        }
    }
}

```

Класс Person содержит свойство Company, которое будет хранить объект класса Company. Члены класса Company объявляются с модификатором public, кроме того также присутствует стандартный конструктор без параметров.

Сериализация в JSON. DataContractJsonSerializer

Для сериализации объектов в формат JSON в пространстве System.Runtime.Serialization.Json определен класс DataContractJsonSerializer. Чтобы задействовать этот класс, в проект надо добавить сборку System.Runtime.Serialization.dll. Для записи объектов в json-файл в этом классе имеется метод WriteObject(), а для чтения ранее сериализованных объектов - метод ReadObject(). Рассмотрим их применение.

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Json;

namespace ConsoleApp
{
    [DataContract]
    public class Person
    {
        [DataMember]
        public string Name { get; set; }
        [DataMember]
        public int Age { get; set; }

        public Person(string name, int year)
        {
            Name = name;
            Age = year;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // объект для сериализации
            Person person1 = new Person("Tom", 29);
            Person person2 = new Person("Bill", 25);
            Person[] people = new Person[] { person1, person2 };

            DataContractJsonSerializer jsonFormatter = new
            DataContractJsonSerializer(typeof(Person[]));

            using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))
            {
                jsonFormatter.WriteObject(fs, people);
            }

            using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))
            {
                Person[] newpeople = (Person[])jsonFormatter.ReadObject(fs);

                foreach (Person p in newpeople)
                {
                    Console.WriteLine("Имя: {0} --- Возраст: {1}", p.Name, p.Age);
                }
            }
        }
    }
}
```

```

        }

        Console.ReadLine();
    }
}

```

Чтобы пометить класс как сериализуемый, к нему применяется атрибут `DataContract`, а ко всем его сериализуемым свойствам - атрибут `DataMember`.

Метод `WriteObject()` принимает два параметра: файловый поток `FileStream` и объект, который надо сериализовать - в данном случае массив объектов `Person`. А метод `ReadObject()` принимает в качестве параметра файловый поток, который представляет файл в формате `json`.

Подобным образом можно сериализовать/десериализовать более сложные объекты:

```

using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Json;

namespace ConsoleApp
{
    [DataContract]
    public class Person
    {
        [DataMember]
        public string Name { get; set; }
        [DataMember]
        public int Age { get; set; }
        [DataMember]
        public Company Company { get; set; }

        public Person()
        { }

        public Person(string name, int age, Company comp)
        {
            Name = name;
            Age = age;
            Company = comp;
        }
    }

    public class Company
    {
        public string Name { get; set; }

        public Company() { }

        public Company(string name)
        {
            Name = name;
        }
    }

    class Program
    {

```



```

static void Main(string[] args)
{
    Person person1 = new Person("Tom", 29, new Company("Microsoft"));
    Person person2 = new Person("Bill", 25, new Company("Apple"));
    Person[] people = new Person[] { person1, person2 };

    DataContractJsonSerializer jsonFormatter = new
DataContractJsonSerializer(typeof(Person[]));

    using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))
    {
        jsonFormatter.WriteObject(fs, people);
    }

    using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))
    {
        Person[] newpeople = (Person[])jsonFormatter.ReadObject(fs);

        foreach (Person p in newpeople)
        {
            Console.WriteLine("Имя: {0} --- Возраст: {1} --- Компания: {2}",
p.Name, p.Age, p.Company.Name);
        }

        Console.ReadLine();
    }
}

```

Задание 4. Написать программу, которая преобразует данные о студенте из XML в JSON и выведет из на экран. Структура состоит из: студенты, группы, курсы, дисциплины (зависят от курса студента). Данные заполнить самостоятельно.

Сборка мусора, управление памятью и указатели

Сборщик мусора в C#

При использовании переменных типов значений в методе, все значения этих переменных попадают в стек. После завершения работы метода стек очищается.

При использовании же ссылочных типов, например, объектов классов, для них также будет отводиться место в стеке, только там будет храниться не значение, а адрес на участок памяти в хипе или куче, в котором и будут находиться сами значения данного объекта. И если объект класса перестает использоваться, то при очистке стека ссылка на участок памяти также очищается, однако это не приводит к немедленной очистке самого участка памяти в куче. Впоследствии сборщик мусора (garbage collector) увидит, что на данный участок памяти больше нет ссылок, и очистит его.

```

using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Json;

namespace ConsoleApp
{

```

```

class Country
{
    public int x;
    public int y;
}

class Program
{
    static void Main(string[] args)
    {
        Test();
    }
    private static void Test()
    {
        Country country = new Country();
        country.x = 10;
        country.y = 15;
    }
}

```

В методе Test создается объект Country. С помощью оператора new в куче для хранения объекта CLR выделяет участок памяти. А в стек добавляет адрес на этот участок памяти. В главном методе Main мы вызываем метод Test. И после того, как Test отработает, место в стеке очищается, а сборщик мусора очищает ранее выделенный под хранение объекта country участок памяти.

Сборщик мусора не запускается сразу после удаления из стека ссылки на объект, размещенный в куче. Он запускается в то время, когда среда CLR обнаружит в этом потребность, например, когда программе требуется дополнительная память.

Как правило, объекты в куче располагаются неупорядоченно, между ними могут иметься пустоты. Куча довольно сильно фрагментирована. Поэтому после очистки памяти в результате очередной сборки мусора оставшиеся объекты перемещаются в один непрерывный блок памяти. Вместе с этим происходит обновление ссылок, чтобы они правильно указывали на новые адреса объектов.

Так же надо отметить, что для крупных объектов существует своя куча - Large Object Heap. В эту кучу помещаются объекты, размер которых больше 85 000 байт. Особенность этой кучи состоит в том, что при сборке мусора сжатие памяти не проводится по причине больших издержек, связанных с размером объектов.

Несмотря на то, что, на сжатие занятого пространства требуется время, да и приложение не сможет продолжать работу, пока не отработает сборщик мусора, однако благодаря подобному подходу также происходит оптимизация приложения. Теперь чтобы найти свободное место в куче среде CLR не надо искать островки пустого пространства среди занятых блоков. Ей достаточно обратиться к указателю кучи, который указывает на свободный участок памяти, что уменьшает количество обращений к памяти.

Кроме того, чтобы снизить издержки от работы сборщика мусора, все объекты в куче разделяются по поколениям. Всего существует три поколения объектов: 0, 1 и 2-е.

К поколению 0 относятся новые объекты, которые еще ни разу не подвергались сборке мусора. К поколению 1 относятся объекты, которые пережили одну сборку, а к поколению 2 - объекты, прошедшие более одной сборки мусора.

Когда сборщик мусора приступает к работе, он сначала анализирует объекты из поколения 0. Те объекты, которые остаются актуальными после очистки, повышаются до поколения 1.

Если после обработки объектов поколения 0 все еще необходима дополнительная память, то сборщик мусора приступает к объектам из поколения 1. Те объекты, на которые уже нет ссылок, уничтожаются, а те, которые по-прежнему актуальны, повышаются до поколения 2.

Поскольку объекты из поколения 0 являются более молодыми и нередко находятся в адресном пространстве памяти рядом друг с другом, то их удаление проходит с наименьшими издержками.

Класс System.GC

Функционал сборщика мусора в библиотеке классов .NET представляет класс System.GC. Через статические методы данный класс позволяет обращаться к сборщику мусора. Как правило, надобность в применении этого класса отсутствует. Наиболее распространенным случаем его использования является сборка мусора при работе с неуправляемыми ресурсами, при интенсивном выделении больших объемов памяти, при которых необходимо такое же быстрое их освобождение.

Рассмотрим некоторые методы и свойства класса System.GC:

- Метод AddMemoryPressure информирует среду CLR о выделении большого объема неуправляемой памяти, которую надо учесть при планировании сборки мусора. В связке с этим методом используется метод RemoveMemoryPressure, который указывает CLR, что ранее выделенная память освобождена, и ее не надо учитывать при сборке мусора.
- Метод Collect приводит в действие механизм сборки мусора. Перегруженные версии метода позволяют указать поколение объектов, вплоть до которого надо произвести сборку мусора
- Метод GetGeneration(Object) позволяет определить номер поколения, к которому относится переданный в качестве параметра объект
- Метод GetTotalMemory возвращает объем памяти в байтах, которое занято в управляемой куче
- Метод WaitForPendingFinalizers приостанавливает работу текущего потока до освобождения всех объектов, для которых производится сборка мусора

```
long totalMemory = GC.GetTotalMemory(false);

GC.Collect();
GC.WaitForPendingFinalizers();
```

С помощью перегруженных версий метода GC.Collect можно выполнить более точную настройку сборки мусора. Так, его перегруженная версия принимает в качестве параметра число - номер поколения, вплоть до которого надо выполнить очистку. Например, GC.Collect(0) - удаляются только объекты поколения 0.

Еще одна перегруженная версия принимает еще и второй параметр - перечисление GCCollectionMode. Это перечисление может принимать три значения:

- Default: значение по умолчанию для данного перечисления (Forced)
- Forced: вызывает немедленное выполнение сборки мусора
- Optimized: позволяет сборщику мусора определить, является ли текущий момент оптимальным для сборки мусора

Например, немедленная сборка мусора вплоть до первого поколения объектов:
GC.Collect(1, GCCollectionMode.Forced);

Финализируемые объекты

Большинство используемых объектов, используемых в программах на C#, относятся к управляемым или managed-коду, и легко очищаются сборщиком мусора. Однако вместе с тем встречаются также и такие объекты, которые задействуют неуправляемые объекты (низкоуровневые файловые дескрипторы, сетевые подключения и т.д.). Такие неуправляемые объекты обращаются к API операционной системы через службы PInvoke. Сборщик мусора может справиться с управляемыми объектами, однако он не знает, как удалять неуправляемые объекты. В этом случае разработчик должен сам реализовывать механизмы очистки на уровне программного кода.

Освобождение неуправляемых ресурсов подразумевает реализацию одного из двух механизмов:

- Создание деструктора
- Реализация классом интерфейса System.IDisposable

Создание деструкторов

Метод деструктора носит имя класса (как и конструктор), перед которым стоит знак тильды (~). Например, создадим деструктор класса Person:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Json;

namespace ConsoleApp
{
    public class Person
    {
        ~Person()
        {
            Console.Beep();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Test();

            Console.ReadKey();
        }

        private static void Test()
        {
            Person p = new Person();
        }
    }
}
```

```
    }  
  }  
}
```

Используя в программе класс `Person`, после ее завершения можно будет услышать голосовой сигнал.

Деструктор в отличие от конструктора не может иметь модификаторов доступа. В данном случае в деструкторе в целях демонстрации просто вызывается звуковой сигнал, но в реальных программах в деструктор вкладывается логика освобождения неуправляемых ресурсов.

Однако на деле при очистке сборщик мусора вызывает не деструктор, а метод `Finalize` класса `Person`. Все потому, что компилятор C# компилирует деструктор в конструкцию, которая эквивалентна следующей:

```
protected override void Finalize()  
{  
    try  
    {  
        // здесь идут инструкции деструктора  
    }  
    finally  
    {  
        base.Finalize();  
    }  
}
```

Метод `Finalize` уже определен в базовом для всех типов классе `Object`, однако данный метод нельзя так просто переопределить. И фактическая его реализация происходит через создание деструктора.

Обратите внимание, что даже после завершения метода `Test` и соответственно удаления из стека ссылки на объект `Person` в куче, может не последовать немедленного вызова деструктора. Лишь при завершении всей программы гарантировано произойдет очистка памяти и вызов деструктора.

На уровне памяти это выглядит так: сборщик мусора при размещении объекта в куче определяет, поддерживает ли данный объект метод `Finalize`. И если объект имеет метод `Finalize`, то указатель на него сохраняется в специальной таблице, которая называется очередь финализации. Когда наступает момент сборки мусора, сборщик видит, что данный объект должен быть уничтожен, и если он имеет метод `Finalize`, то он копируется в еще одну таблицу и окончательно уничтожается лишь при следующем проходе сборщика мусора.

И здесь мы можем столкнуться со следующей проблемой: а что, если нам немедленно надо вызвать деструктор и освободить все связанные с объектом неуправляемые ресурсы? В этом случае мы можем использовать второй подход - реализацию интерфейса `IDisposable`.

Интерфейс `IDisposable`

Интерфейс `IDisposable` объявляет один единственный метод `Dispose`, в котором при реализации интерфейса в классе должно происходить освобождение неуправляемых ресурсов. Например:

```
using System;
```

```

namespace ConsoleApp
{
    public class Person : IDisposable
    {
        public void Dispose()
        {
            Console.Beep();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Test();

            Console.ReadKey();
        }
        private static void Test()
        {
            Person p = null;
            try
            {
                p = new Person();
            }
            finally
            {
                if (p != null)
                {
                    p.Dispose();
                }
            }
        }
    }
}

```

В данном коде используется конструкция try...finally. По сути эта конструкция по функционалу в общем эквивалентна следующим двум строкам кода:

```

private static void Test()
{
    Person p = new Person();
    p.Dispose();
}

```

Но конструкцию try...finally предпочтительнее использовать при вызове метода Dispose, так как она гарантирует, что даже в случае возникновения исключения произойдет освобождение ресурсов в методе Dispose.

Синтаксис C# также предлагает синонимичную конструкцию для автоматического вызова метод Dispose - конструкцию using:

```

private static void Test()
{
    using (Person p = new Person())
    {
    }
}

```

Комбинирование подходов

Мы рассмотрели два подхода. Какой же из них лучше? С одной стороны, метод Dispose позволяет в любой момент времени вызвать освобождение связанных ресурсов, а с другой - программист, использующий наш класс, может забыть поставить в коде вызов метода Dispose. В общем бывают различные ситуации. И чтобы сочетать плюсы обоих подходов мы можем использовать комбинированный подход. Microsoft предлагает нам использовать следующий формализованный шаблон:

```
public class SomeClass : IDisposable
{
    private bool disposed = false;

    // реализация интерфейса IDisposable.
    public void Dispose()
    {
        Dispose(true);
        // подавляем финализацию
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Освобождаем управляемые ресурсы
            }
            // освобождаем неуправляемые объекты
            disposed = true;
        }
    }

    // Деструктор
    ~SomeClass()
    {
        Dispose(false);
    }
}
```

Логика очистки реализуется перегруженной версией метода Dispose(bool disposing). При вызове деструктора в качестве параметра disposing передается значение false, чтобы избежать очистки управляемых ресурсов, так как мы не можем быть уверенными в их состоянии, что они до сих пор находятся в памяти. И в этом случае остается полагаться на деструкторы этих ресурсов. Ну и в обоих случаях освобождаются неуправляемые ресурсы.

Еще один важный момент - вызов в методе Dispose метода GC.SuppressFinalize(this). GC.SuppressFinalize не позволяет системе выполнить вызов метода Finalize для данного объекта.

Таким образом, даже если разработчик не использует в программе метод Dispose, все равно произойдет очистка и освобождение ресурсов.

Общие рекомендации по использованию Finalize и Dispose

- Деструктор следует реализовывать только у тех объектов, которым он действительно необходим, так как метод `Finalize` оказывает сильное влияние на производительность
- После вызова метода `Dispose` необходимо блокировать у объекта вызов метода `Finalize` с помощью `GC.SuppressFinalize`
- При создании производных классов от базовых, которые реализуют интерфейс `IDisposable`, следует также вызывать метод `Dispose` базового класса:

```
public class Derived : Base
{
    private bool IsDisposed = false;

    protected override void Dispose(bool disposing)
    {
        if (IsDisposed) return;
        if (disposing)
        {
            // Освобождение управляемых ресурсов
        }
        IsDisposed = true;
        // Обращение к методу Dispose базового класса
        base.Dispose(disposing);
    }
}
```

- Отдавайте предпочтение комбинированному шаблону, реализующему как метод `Dispose`, так и деструктор

При изучении C/C++, вы сталкивались с таким понятием как указатели. Указатели позволяют получить доступ к определенной ячейке памяти и произвести определенные манипуляции со значением, хранящимся в этой ячейке.

В языке C# указатели очень редко используются, однако в некоторых случаях можно прибегать к ним для оптимизации приложений. Код, применяющий указатели, еще называют небезопасным кодом. Однако это не значит, что он представляет какую-то опасность. Просто при работе с ним все действия по использованию памяти, в том числе по ее очистке, ложится целиком на нас, а не на среду CLR. И с точки зрения CLR такой код не безопасен, так как среда не может проверить данный код, поэтому повышается вероятность различного рода ошибок.

Чтобы использовать небезопасный код в C#, надо первым делом указать проекту, что он будет работать с небезопасным кодом. Для этого надо установить в настройках проекта соответствующий флаг - в меню Project (Проект) найти Свойства проекта. Затем в меню Build установить флажок Allow unsafe code (Разрешить небезопасный код).

Ключевое слово `unsafe`

Блок кода или метод, в котором используются указатели, помечается ключевым словом `unsafe`:

```
unsafe
{
}
```


Метод, использующий указатели:

```
unsafe private static void PointerMethod()
{
}
}
```

Также с помощью unsafe можно объявлять структуры:

```
unsafe struct State
{
}
}
```

Операции * и &

Ключевой при работе с указателями является операция *, которую еще называют операцией разыменовывания. Операция разыменовывания позволяет получить или установить значение по адресу, на который указывает указатель. Для получения адреса переменной применяется операция &:

```
using System;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            unsafe
            {
                int* x; // определение указателя
                int y = 10; // определяем переменную

                x = &y; // указатель x теперь указывает на адрес переменной y
                Console.WriteLine(*x); // 10

                y = y + 20;
                Console.WriteLine(*x); // 30

                *x = 50;
                Console.WriteLine(y); // переменная y=50
            }

            Console.ReadLine();
        }
    }
}
```

При объявлении указателя указываем тип `int*` `x`; - в данном случае объявляется указатель на целое число. Но кроме типа `int` можно использовать и другие: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` или `bool`. Также можно объявлять указатели на типы `enum`, структуры и другие указатели.

Выражение `x = &y`; позволяет нам получить адрес переменной `y` и установить на него указатель `x`. До этого указатель `x` не на что не указывал.

После этого все операции с `y` будут влиять на значение, получаемое через указатель `x` и наоборот, так как они указывают на одну и ту же область в памяти.

Для получения значения, которое хранится в области памяти, на которую указывает указатель `x`, используется выражение `*x`.

Получение адреса

```
using System;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            unsafe
            {
                int* x; // определение указателя
                int y = 10; // определяем переменную

                x = &y; // указатель x теперь указывает на адрес переменной y

                // получим адрес переменной y
                ulong addr = (ulong)x;
                Console.WriteLine("Адрес переменной y: {0}", addr);
            }

            Console.ReadLine();
        }
    }
}
```

Так как значение адреса - это целое число, а на 32-разрядных системах диапазон адресов 0 до 4 000 000 000, то для получения адреса используется преобразование в тип `uint`, `long` или `ulong`. Соответственно на 64-разрядных системах диапазон доступных адресов гораздо больше, поэтому в данном случае лучше использовать `ulong`, чтобы избежать ошибки переполнения.

Операции с указателями

Кроме операции разыменовывания к указателям применимы еще и некоторые арифметические операции (`+`, `++`, `-`, `--`, `+=`, `-=`) и преобразования. Например, мы можем преобразовать число в указатель:

```
using System;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            unsafe
            {
                int* x; // определение указателя
                int y = 10; // определяем переменную
                x = &y; // указатель x теперь указывает на адрес переменной y

                // получим адрес переменной y
                uint addr = (uint)x;
                Console.WriteLine("Адрес переменной y: {0}", addr);
            }
        }
    }
}
```

```

        byte* bytePointer = (byte*)(addr + 4); // получить указатель на следующий
байт после addr
        Console.WriteLine("Значение byte по адресу {0}: {1}", addr + 4,
*bytePointer);

        // обратная операция
        uint oldAddr = (uint)bytePointer - 4; // вычитаем четыре байта, так как
bytePointer - указатель на байт
        int* intPointer = (int*)oldAddr;
        Console.WriteLine("Значение int по адресу {0}: {1}", oldAddr,
*intPointer);

        // преобразование в тип double
        double* doublePointer = (double*)(addr + 4);
        Console.WriteLine("Значение double по адресу {0}: {1}", addr + 4,
*doublePointer);
    }

    Console.ReadLine();

}
}
}

```

Так как у нас `x` - указатель на объект `int`, который занимает 4 байта, то мы можем получить следующий за ним байт с помощью выражения `byte* bytePointer = (byte*)addr+4;`. Теперь указатель `bytePointer` указывает на следующий байт. Равным образом мы можем создать и другой указатель `double* doublePointer = (double*)addr + 4;`, только этот указатель уже будет указывать на следующие 8 байт, так как тип `double` занимает 8 байт.

Чтобы обратно получить исходный адрес, вызываем выражение `bytePointer - 4`. Здесь `bytePointer` - это указатель, а не число, и операции вычитания и сложения будут происходить в соответствии с правилами арифметики указателей.

Хотя мы к указателю прибавляем число 4, но итоговый адрес увеличится на 8, так как размер объекта `char` - 2 байта, а $2 * 4 = 8$. Подобным образом действует сложение с другими типами указателей.

Аналогично работает вычитание.

Указатель на другой указатель

Объявление и использование указателя на указатель:

```

using System;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            unsafe
            {
                int* x; // определение указателя
                int y = 10; // определяем переменную

                x = &y; // указатель x теперь указывает на адрес переменной y
                int** z = &x; // указатель z теперь указывает на адрес, который указывает
и указатель x
                **z = **z + 40; // изменение указателя z повлечет изменение переменной y
            }
        }
    }
}

```

```

        Console.WriteLine(y); // переменная y=50
        Console.WriteLine(**z); // переменная **z=50
    }

    Console.ReadLine();
}
}
}

```

Указатели на структуры, члены классов и массивы

Указатели на типы и операция ->

Кроме указателей на простые типы можно использовать указатели на структуры. А для доступа к полям структуры, на которую указывает указатель, используется операция ->:

```

using System;

namespace ConsoleApp
{
    public struct Person
    {
        public int age;
        public int height;
    }

    class Program
    {
        static void Main(string[] args)
        {
            unsafe
            {
                Person person;
                person.age = 29;
                person.height = 176;
                Person* p = &person;
                p->age = 30;
                Console.WriteLine(p->age);

                // разыменовывание указателя
                (*p).height = 180;
                Console.WriteLine((*p).height);
            }

            Console.ReadLine();
        }
    }
}

```

Обращаясь к указателю `p->age = 30`; мы можем получить или установить значение свойства структуры, на которую указывает указатель. Обратите внимание, что просто написать `p.age=30` мы не можем, так как `p` - это не структура `Person`, а указатель на структуру.

Альтернативой служит операция разыменования: `(*p).height = 180`;

Указатели на массивы и `stackalloc`

С помощью ключевого слова `stackalloc` можно выделить память под массив в стеке. Смысл выделения памяти в стеке в повышении быстродействия кода. Посмотрим на примере вычисления факториала:

```
using System;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            unsafe
            {
                const int size = 7;
                int* factorial = stackalloc int[size]; // выделяем память в стеке под
                семь объектов int
                int* p = factorial;

                *(p++) = 1; // присваиваем первой ячейке значение 1 и
                           // увеличиваем указатель на 1
                for (int i = 2; i <= size; i++, p++)
                {
                    // считаем факториал числа
                    *p = p[-1] * i;
                }
                for (int i = 1; i <= size; ++i)
                {
                    Console.WriteLine(factorial[i - 1]);
                }
            }

            Console.ReadLine();
        }
    }
}
```

Оператор `stackalloc` принимает после себя массив, на который будет указывать указатель. `int* factorial = stackalloc int[size];`.

Для манипуляций с массивом создаем указатель `p`: `int* p = factorial;`, который указывает на первый элемент массива, в котором всего 7 элементов

Далее начинаются уже сами операции с указателем и подсчет факториала. Так как факториал 1 равен 1, то присваиваем первому элементу, на который указывает указатель `p`, единицу с помощью операции разыменования: `*(p++) = 1;`

Из прошлой темы мы узнали, что чтобы вложить некоторое значение по адресу, который хранит указатель, надо использовать выражение: `*p=1`. Но, кроме этого, тут происходит также инкремент указателя `p++`. То есть сначала первому элементу массива присваивается единица, потом указатель `p` смещается и начинает указывать уже на второй элемент.

Чтобы получить предыдущий элемент и сместиться назад, можно использовать операцию декремента: `Console.WriteLine(*(--p));`. Обратите внимание, что операции `*(--p)` и `*(p--)` различаются, так как в первом случае сначала идет смещение указателя, а затем его разыменовывание. А во втором случае - наоборот.

Затем вычисляем факториал всех остальных шести чисел: `*p = p[-1] * i;`. Обращение к указателям как к массивам представляет альтернативу операции разыменовывания для получения значения. В данном случае мы получаем значение предыдущего элемента.

И в заключении, используя указатель `factorial`, выводим факториалы всех семи чисел.

Оператор `fixed` и закрепление указателей

Ранее мы рассмотрели, как создавать указатели на типы значений, например, `int` или структуры. Однако кроме структур в C# есть еще и классы, которые в отличие от типов значений, помещают все связанные значения в куче. И в работу данных классов может в любой момент вмешаться сборщик мусора, периодически очищающий кучу. Чтобы фиксировать на все время работы указатели на объекты классов используется оператор `fixed`.

Допустим, у нас есть класс `Person`:

```
public class Person
{
    public int age;
    public int height;
}
```

Зафиксируем указатель с помощью оператора `fixed`:

```
static void Main(string[] args)
{
    unsafe
    {
        Person person = new Person();
        person.age = 28;
        person.height = 178;
        // блок фиксации указателя
        fixed (int* p = &person.age)
        {
            if (*p < 30)
            {
                *p = 30;
            }
        }
        Console.WriteLine(person.age); // 30
    }

    Console.ReadLine();
}
```

Оператор `fixed` создает блок, в котором фиксируется указатель на поле объекта `person`. После завершения блока `fixed` закрепление с переменных снимается, и они могут быть подвержены сборке мусора.

Кроме адреса переменной можно также инициализировать указатель, используя массив, строку или буфер фиксированного размера:

```
unsafe
{
    int[] nums = { 0, 1, 2, 3, 7, 88 };
    string str = "Привет мир";
    fixed (int* p = nums)
```

```
    {  
    }  
    fixed (char* p = str)  
    {  
    }  
}
```

При инициализации указателей на строку следует учитывать, что указатель должен иметь тип `char*`.

Задание 5. Написать программу, считающую Фибоначчи с использованием указателей. Последовательность начинается с 0.

Многопоточность

Введение в многопоточность. Класс Thread

Одним из ключевых аспектов в современном программировании является многопоточность. Ключевым понятием при работе с многопоточностью является поток. Поток представляет некоторую часть кода программы. При выполнении программы каждому потоку выделяется определенный квант времени. И при помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы заблокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому, к примеру, клиент-серверные приложения (и не только они) практически не мыслимы без многопоточности.

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен `System.Threading`. В нем определен класс, представляющий отдельный поток - класс `Thread`.

Класс `Thread` определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства класса:

- Статическое свойство `CurrentContext` позволяет получить контекст, в котором выполняется поток
- Статическое свойство `CurrentThread` возвращает ссылку на выполняемый поток
- Свойство `IsAlive` указывает, работает ли поток в текущий момент
- Свойство `IsBackground` указывает, является ли поток фоновым
- Свойство `Name` содержит имя потока
- Свойство `Priority` хранит приоритет потока - значение перечисления `ThreadPriority`
- Свойство `ThreadState` возвращает состояние потока - одно из значений перечисления `ThreadState`
- Некоторые методы класса `Thread`:
- Статический метод `GetDomain` возвращает ссылку на домен приложения

- Статический метод `GetDomainId` возвращает id домена приложения, в котором выполняется текущий поток
- Статический метод `Sleep` останавливает поток на определенное количество миллисекунд
- Метод `Abort` уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно. Для проверки завершенности потока следует опрашивать его свойство `ThreadState`
- Метод `Interrupt` прерывает поток на некоторое время
- Метод `Join` блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
- Метод `Resume` возобновляет работу ранее приостановленного потока
- Метод `Start` запускает поток
- Метод `Suspend` приостанавливает поток

Получение информации о потоке

Используем вышеописанные свойства и методы для получения информации о потоке:

```
using System;
using System.Threading;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            // получаем текущий поток
            Thread t = Thread.CurrentThread;

            //получаем имя потока
            Console.WriteLine("Имя потока: {0}", t.Name);
            t.Name = "Метод Main";
            Console.WriteLine("Имя потока: {0}", t.Name);

            Console.WriteLine("Запущен ли поток: {0}", t.IsAlive);
            Console.WriteLine("Приоритет потока: {0}", t.Priority);
            Console.WriteLine("Статус потока: {0}", t.ThreadState);

            // получаем домен приложения
            Console.WriteLine("Домен приложения: {0}", Thread.GetDomain().FriendlyName);

            Console.ReadLine();
        }
    }
}
```

Так как по умолчанию свойство `Name` у объектов `Thread` не установлено, то в первом случае мы получаем в качестве значения этого свойства пустую строку.

Статус потока

Статусы потока содержатся в перечислении `ThreadState`:

- Aborted: поток остановлен, но пока еще окончательно не завершен
- AbortRequested: для потока вызван метод Abort, но остановка потока еще не произошла
- Background: поток выполняется в фоновом режиме
- Running: поток запущен и работает (не приостановлен)
- Stopped: поток завершен
- StopRequested: поток получил запрос на остановку
- Suspended: поток приостановлен
- SuspendRequested: поток получил запрос на приостановку
- Unstarted: поток еще не был запущен
- WaitSleepJoin: поток заблокирован в результате действия методов Sleep или Join

В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение Unstarted. Запустив поток, мы изменим его статус на Running. Вызвав метод Sleep, статус изменится на WaitSleepJoin. А применяя метод Abort, мы тем самым переведем поток в состояние AbortRequested, а затем Aborted, после чего поток окончательно завершится.

Приоритеты потоков

Приоритеты потоков располагаются в перечислении ThreadPriority:

- Lowest
- BelowNormal
- Normal
- AboveNormal
- Highest

По умолчанию потоку задается значение Normal. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

Создание потоков. Делегат ThreadStart

Используя класс Thread, мы можем выделить в приложении несколько потоков, которые будут выполняться одновременно.

Во-первых, для запуска нового потока нам надо определить задачу в приложении, которую будет выполнять данный поток. Для этого мы можем добавить новый метод, производящий какие-либо действия.

Для создания нового потока используется делегат ThreadStart, который получает в качестве параметра метод, который мы определили выше.

И чтобы запустить поток, вызывается метод Start. Рассмотрим на примере:

```
using System;
using System.Threading;
```

```

namespace ConsoleApp
{
    class Program
    {
        public static void Count()
        {
            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine("Второй поток:");
                Console.WriteLine(i * i);
                Thread.Sleep(400);
            }
        }
        static void Main(string[] args)
        {
            // создаем новый поток
            Thread myThread = new Thread(new ThreadStart(Count));
            myThread.Start(); // запускаем поток

            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine("Главный поток:");
                Console.WriteLine(i * i);
                Thread.Sleep(300);
            }
            Console.ReadKey();
        }
    }
}

```

Здесь новый поток будет производить действия, определенные в методе Count. В данном случае это возведение в квадрат числа и вывод его на экран. И после каждого умножения с помощью метода Thread.Sleep мы усыпляем поток на 400 миллисекунд.

Чтобы запустить этот метод в качестве второго потока, мы сначала создаем объект потока: `Thread myThread = new Thread(new ThreadStart(Count));`. В конструктор передается делегат ThreadStart, который в качестве параметра принимает метод Count. И следующей строкой `myThread.Start()` мы запускаем поток. После этого управление передается главному потоку, и выполняются все остальные действия, определенные в методе Main.

Таким образом, в нашей программе будут работать одновременно главный поток, представленный методом Main, и второй поток. Кроме действий по созданию второго потока, в главном потоке также производятся некоторые вычисления. Как только все потоки отработают, программа завершит свое выполнение.

Подобным образом мы можем создать и три, и четыре, и целый набор новых потоков, которые смогут решать те или иные задачи.

Существует еще одна форма создания потока: `Thread myThread = new Thread(Count);`

Хотя в данном случае явным образом мы не используем делегат ThreadStart, но неявно он создается. Компилятор C# выводит делегат из сигнатуры метода Count и вызывает соответствующий конструктор.

Потоки с параметрами и ParameterizedThreadStart

Для передачи параметров в поток используется делегат `ParameterizedThreadStart`. Его действие похоже на функциональность делегата `ThreadStart`. Рассмотрим на примере:

```
using System;
using System.Threading;

namespace ConsoleApp
{
    class Program
    {
        public static void Count(object x)
        {
            for (int i = 1; i < 9; i++)
            {
                int n = (int)x;

                Console.WriteLine("Второй поток:");
                Console.WriteLine(i * n);
                Thread.Sleep(400);
            }
        }
        static void Main(string[] args)
        {
            int number = 4;
            // создаем новый поток
            Thread myThread = new Thread(new ParameterizedThreadStart(Count));
            myThread.Start(number);

            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine("Главный поток:");
                Console.WriteLine(i * i);
                Thread.Sleep(300);
            }

            Console.ReadKey();
        }
    }
}
```

После создания потока мы передаем метод `myThread.Start(number)`; переменную, значение которой хотим передать в поток.

При использовании `ParameterizedThreadStart` мы сталкиваемся с ограничением: мы можем запускать во втором потоке только такой метод, который в качестве единственного параметра принимает объект типа `object`. Поэтому в данном случае нам надо дополнительно привести переданное значение к типу `int`, чтобы его использовать в вычислениях.

Но что делать, если нам надо передать не один, а несколько параметров различного типа? В этом случае на помощь приходит классовый подход:

```
using System;
using System.Threading;

namespace ConsoleApp
{
    public class Counter
    {
        public int x;
```

```

        public int y;
    }

    class Program
    {
        public static void Count(object obj)
        {
            for (int i = 1; i < 9; i++)
            {
                Counter c = (Counter)obj;

                Console.WriteLine("Второй поток:");
                Console.WriteLine(i * c.x * c.y);
            }
        }
        static void Main(string[] args)
        {
            Counter counter = new Counter();
            counter.x = 4;
            counter.y = 5;

            Thread myThread = new Thread(new ParameterizedThreadStart(Count));
            myThread.Start(counter);

            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine("Главный поток:");
                Console.WriteLine(i * i);
                Thread.Sleep(300);
            }

            Console.ReadKey();
        }
    }
}

```

Сначала определяем специальный класс Counter, объект которого будет передаваться во второй поток, а в методе Main передаем его во второй поток.

Но тут опять же есть одно ограничение: метод Thread.Start не является типобезопасным, то есть мы можем передать в него любой тип, и потом нам придется приводить переданный объект к нужному нам типу. Для решения данной проблемы рекомендуется объявлять все используемые методы и переменные в специальном классе, а в основной программе запускать поток через ThreadStart. Например:

```

using System;
using System.Threading;

namespace ConsoleApp
{
    public class Counter
    {
        private int x;
        private int y;

        public Counter(int x, int y)
        {
            this.x = x;
            this.y = y;
        }
    }
}

```

```

        public void Count()
        {
            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine("Второй поток:");
                Console.WriteLine(i * x * y);
                Thread.Sleep(400);
            }
        }
    }

    class Program {

        static void Main(string[] args)
        {
            Counter counter = new Counter(5, 4);

            Thread myThread = new Thread(new ThreadStart(counter.Count));
            myThread.Start();

            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine("Главный поток:");
                Console.WriteLine(i * i);
                Thread.Sleep(300);
            }

            Console.ReadKey();
        }
    }
}

```

Синхронизация потоков

Нередко в потоках используются некоторые разделяемые ресурсы, общие для всей программы. Это могут быть общие переменные, файлы, другие ресурсы. Например:

```

using System;
using System.Threading;

namespace ConsoleApp
{
    class Program {

        static int x = 0;
        static void Main(string[] args)
        {
            for (int i = 0; i < 5; i++)
            {
                Thread myThread = new Thread(Count);
                myThread.Name = "Поток " + i.ToString();
                myThread.Start();
            }

            Console.ReadLine();
        }
        public static void Count()
        {
            x = 1;
            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
            }
        }
    }
}

```

```

        x++;
        Thread.Sleep(100);
    }
}
}
}
}

```

Здесь у нас запускаются пять потоков, которые работают с общей переменной `x`. И мы предполагаем, что метод выведет все значения `x` от 1 до 8. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной `x` становится непредсказуемым.

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово `lock`. Оператор `lock` определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. И мы можем переделать предыдущий пример следующим образом:

```

using System;
using System.Threading;

namespace ConsoleApp
{
    class Program {

        static int x = 0;
        static object locker = new object();
        static void Main(string[] args)
        {
            for (int i = 0; i < 5; i++)
            {
                Thread myThread = new Thread(Count);
                myThread.Name = "Поток " + i.ToString();
                myThread.Start();
            }

            Console.ReadLine();
        }
        public static void Count()
        {
            lock (locker)
            {
                x = 1;
                for (int i = 1; i < 9; i++)
                {
                    Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
                    x++;
                    Thread.Sleep(100);
                }
            }
        }
    }
}

```

Для блокировки с ключевым словом `lock` используется объект-заглушка, в данном случае это переменная `locker`. Когда выполнение доходит до оператора `lock`, объект `locker` блокируется, и на время его блокировки монопольный доступ к

блоку кода имеет только один поток. После окончания работы блока кода, объект locker освобождается и становится доступным для других потоков.

Мониторы

Наряду с оператором lock для синхронизации потоков мы можем использовать мониторы, представленные классом System.Threading.Monitor. Фактически конструкция оператора lock из прошлой темы инкапсулирует в себе синтаксис использования мониторов. А рассмотренный в прошлой теме пример будет эквивалентен следующему коду:

```
using System;
using System.Threading;

namespace ConsoleApp
{
    class Program
    {
        static int x = 0;
        static object locker = new object();
        static void Main(string[] args)
        {
            for (int i = 0; i < 5; i++)
            {
                Thread myThread = new Thread(Count);
                myThread.Name = "Поток " + i.ToString();
                myThread.Start();
            }

            Console.ReadLine();
        }
        public static void Count()
        {
            try
            {
                Monitor.Enter(locker);
                x = 1;
                for (int i = 1; i < 9; i++)
                {
                    Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
                    x++;
                    Thread.Sleep(100);
                }
            }
            finally
            {
                Monitor.Exit(locker);
            }
        }
    }
}
```

Метод Monitor.Enter блокирует объект locker так же, как это делает оператор lock. А в блоке try...finally с помощью метода Monitor.Exit происходит освобождение объекта locker, и он становится доступным для других потоков.

Кроме блокировки и разблокировки объекта класс Monitor имеет еще ряд методов, которые позволяют управлять синхронизацией потоков. Так, метод Monitor.Wait освобождает блокировку объекта и переводит поток в очередь ожидания объекта. Следующий поток в очереди готовности объекта блокирует данный объект. А все

потоки, которые вызвали метод Wait, остаются в очереди ожидания, пока не получат сигнала от метода Monitor.Pulse или Monitor.PulseAll, посланного владельцем блокировки. Если метод Monitor.Pulse отправлен, поток, находящийся во главе очереди ожидания, получает сигнал и блокирует освободившийся объект. Если же метод Monitor.PulseAll отправлен, то все потоки, находящиеся в очереди ожидания, получают сигнал и переходят в очередь готовности, где им снова разрешается получать блокировку объекта.

Класс AutoResetEvent

Класс AutoResetEvent также служит целям синхронизации потоков. Этот класс является оберткой над объектом ОС Windows "событие" и позволяет переключить данный объект-событие из сигнального в несигнальное состояние. Так, пример из предыдущей темы мы можем переписать с использованием AutoResetEvent следующим образом:

```
using System;
using System.Threading;

namespace ConsoleApp
{
    class Program
    {
        static AutoResetEvent waitHandler = new AutoResetEvent(true);
        static int x = 0;

        static void Main(string[] args)
        {
            for (int i = 0; i < 5; i++)
            {
                Thread myThread = new Thread(Count);
                myThread.Name = "Поток " + i.ToString();
                myThread.Start();
            }

            Console.ReadLine();
        }

        public static void Count()
        {
            waitHandler.WaitOne();
            x = 1;
            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
                x++;
                Thread.Sleep(100);
            }
            waitHandler.Set();
        }
    }
}
```

Во-первых, создаем переменную типа AutoResetEvent. Передавая в конструктор значение true, мы тем самым указываем, что создаваемый объект изначально будет в сигнальном состоянии.

Когда начинает работать поток, то первым делом срабатывает определенный в методе Count вызов waitHandler.WaitOne(). Метод WaitOne указывает, что текущий поток переводится в состояние ожидания, пока объект waitHandler не

будет переведен в сигнальное состояние. И так все потоки у нас переводятся в состояние ожидания.

После завершения работы вызывается метод `waitHandler.Set`, который уведомляет все ожидающие потоки, что объект `waitHandler` снова находится в сигнальном состоянии, и один из потоков "захватывает" данный объект, переводит в несигнальное состояние и выполняет свой код. А остальные потоки снова ожидают.

Так как в конструкторе `AutoResetEvent` мы указываем, что объект изначально находится в сигнальном состоянии, то первый из очереди потоков захватывает данный объект и начинает выполнять свой код.

Но если бы мы написали `AutoResetEvent waitHandler = new AutoResetEvent(false)`, тогда объект изначально был бы в несигнальном состоянии, а поскольку все потоки блокируются методом `waitHandler.WaitOne()` до ожидания сигнала, то у нас попросту случилась бы блокировка программы, и программа не выполняла бы никаких действий.

Если у нас в программе используются несколько объектов `AutoResetEvent`, то мы можем использовать для отслеживания состояния этих объектов методы `WaitAll` и `WaitAny`, которые в качестве параметра принимают массив объектов класса `WaitHandle` - базового класса для `AutoResetEvent`.

Так, мы тоже можем использовать `WaitAll` в вышеприведенном примере. Для этого надо строку `waitHandler.WaitOne()`; заменить на следующую:

```
AutoResetEvent.WaitAll(new WaitHandle[] { waitHandler });
```

Мьютексы

Еще один инструмент управления синхронизацией потоков представляет класс `Mutex`, также находящийся в пространстве имен `System.Threading`. Данный класс является классом-оболочкой над соответствующим объектом ОС Windows "мьютекс". Перепишем пример из прошлой темы, используя мьютексы:

```
using System;
using System.Threading;

namespace ConsoleApp
{
    class Program
    {
        static Mutex mutexObj = new Mutex();
        static int x = 0;

        static void Main(string[] args)
        {
            for (int i = 0; i < 5; i++)
            {
                Thread myThread = new Thread(Count);
                myThread.Name = "Поток " + i.ToString();
                myThread.Start();
            }

            Console.ReadLine();
        }
        public static void Count()
        {

```

```

        mutexObj.WaitOne();
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
            x++;
            Thread.Sleep(100);
        }
        mutexObj.ReleaseMutex();
    }
}
}

```

Сначала создаем объект мьютекса: `Mutex mutexObj = new Mutex()`.

Основную работу по синхронизации выполняют методы `WaitOne()` и `ReleaseMutex()`. Метод `mutexObj.WaitOne()` приостанавливает выполнение потока до тех пор, пока не будет получен мьютекс `mutexObj`.

После выполнения всех действий, когда мьютекс больше не нужен, поток освобождает его с помощью метода `mutexObj.ReleaseMutex()`

Таким образом, когда выполнение дойдет до вызова `mutexObj.WaitOne()`, поток будет ожидать, пока не освободится мьютекс. И после его получения продолжит выполнять свою работу.

Мы использовали мьютекс для синхронизации потоков. Однако замечательная черта мьютексов состоит также в том, что они могут также применяться не только внутри одного процесса, но и между процессами. Типичный пример - создание приложения, которое можно запустить только один раз. Создадим подобное приложение:

```

using System;
using System.Reflection;
using System.Runtime.InteropServices;
using System.Threading;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            bool existed;
            // получаем GUID приложения
            string guid =
Marshal.GetTypeLibGuidForAssembly(Assembly.GetExecutingAssembly()).ToString();

            Mutex mutexObj = new Mutex(true, guid, out existed);

            if (existed)
            {
                Console.WriteLine("Приложение работает");
            }
            else
            {
                Console.WriteLine("Приложение уже было запущено. И сейчас оно будет
закрыто.");
                Thread.Sleep(3000);
                return;
            }
        }
    }
}

```

```

        Console.ReadLine();
    }
}

```

В данном случае для создания мьютекса мы используем другую перегрузку конструктора. Значение `true`, которое передается в качестве первого параметра конструктора, указывает, что приложение будет запрашивать владение мьютексом. Второй параметр указывает на уникальное имя мьютекса. В данном случае в качестве имени выбран `guid` приложения, то есть глобальный уникальный идентификатор.

Третий параметр возвращает значение из конструктора. Если он равен `true`, то это означает, что мьютекс запрошен и получен. А если `false` - то запрос на владение мьютексом отклонен.

И после создания мьютекса, если мы запустим вторую копию приложения, то она будет закрыта. И в один момент времени сможет работать только одна копия программы.

Семафоры

Еще один инструмент, который предлагает нам платформа .NET для управления синхронизацией, представляют семафоры. Семафоры позволяют ограничить доступ определенным количеством объектов.

Например, у нас такая задача: есть некоторое число читателей, которые приходят в библиотеку три раза в день и что-то там читают. И пусть у нас будет ограничение, что одновременно в библиотеке не может находиться больше трех читателей. Данную задачу очень легко решить с помощью семафоров:

```

using System;
using System.Threading;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 1; i < 6; i++)
            {
                Reader reader = new Reader(i);

                Console.ReadLine();
            }
        }

        class Reader
        {
            // создаем семафор
            static Semaphore sem = new Semaphore(3, 3);
            Thread myThread;
            int count = 3; // счетчик чтения

            public Reader(int i)
            {
                myThread = new Thread(Read);
            }
        }
    }
}

```

```

        myThread.Name = "Читатель " + i.ToString();
        myThread.Start();
    }

    public void Read()
    {
        while (count > 0)
        {
            sem.WaitOne();

            Console.WriteLine("{0} входит в библиотеку", Thread.CurrentThread.Name);

            Console.WriteLine("{0} читает", Thread.CurrentThread.Name);
            Thread.Sleep(1000);

            Console.WriteLine("{0} покидает библиотеку", Thread.CurrentThread.Name);

            sem.Release();

            count--;
            Thread.Sleep(1000);
        }
    }
}

```

В данной программе читатель представлен классом Reader. Он инкапсулирует всю функциональность, связанную с потоками, через переменную Thread myThread.

Для создания семафора используется класс Semaphore: static Semaphore sem = new Semaphore(3, 3);. Его конструктор принимает два параметра: первый указывает, какому числу объектов изначально будет доступен семафор, а второй параметр указывает, какой максимальное число объектов будет использовать данный семафор. В данном случае у нас только три читателя могут одновременно находиться в библиотеке, поэтому максимальное число равно 3.

Основной функционал сосредоточен в методе Read, который и выполняется в потоке. В начале для ожидания получения семафора используется метод sem.WaitOne(). После того, как в семафоре освободится место, данный поток заполняет свободное место и начинает выполнять все дальнейшие действия. После окончания чтения мы высвобождаем семафор с помощью метода sem.Release(). После этого в семафоре освобождается одно место, которое заполняет другой поток.

А в методе Main нам остается только создать читателей, которые запускают соответствующие потоки.

Использование таймеров

Одним из важнейших классов, находящихся в пространстве имени System.Threading, является класс Timer. Данный класс позволяет запускать определенные действия по истечению некоторого периода времени.

Например, нам надо запускать какой-нибудь метод через каждые 2000 миллисекунд, то есть раз в две секунды:

```

using System;
using System.Threading;

```

```

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            int num = 0;
            // устанавливаем метод обратного вызова
            TimerCallback tm = new TimerCallback(Count);
            // создаем таймер
            Timer timer = new Timer(tm, num, 0, 2000);

            Console.ReadLine();
        }
        public static void Count(object obj)
        {
            int x = (int)obj;
            for (int i = 1; i < 9; i++, x++)
            {
                Console.WriteLine("{0}", x * i);
            }
            Console.WriteLine();
        }
    }
}

```

Первым делом создается объект делегата `TimerCallback`, который в качестве параметра принимает метод. Причем данный метод должен в качестве параметра принимать объект типа `object`.

И затем создается таймер. Данная перегрузка конструктора таймера принимает четыре параметра:

- объект делегата `TimerCallback`
- объект, передаваемый в качестве параметра в метод `Count`
- количество миллисекунд, через которое таймер будет запускаться. В данном случае таймер будет запускать немедленно после создания, так как в качестве значения используется 0
- интервал между вызовами метода `Count`

И, таким образом, после запуска программы каждые две секунды будет срабатывать метод `Count`.

Если бы нам не надо было бы использовать параметр `obj` у метода `Count`, то при создании таймера мы могли бы указывать в качестве соответствующего параметра значение `null`: `Timer timer = new Timer(tm, null, 0, 2000);`

Задание 6. Задача об обедающих философах. Пять безмолвных философов сидят вокруг круглого стола, перед каждым философом стоит тарелка спагетти. Вилки лежат на столе между каждой парой ближайших философов.

Каждый философ может либо есть, либо размышлять. Приём пищи не ограничен количеством оставшихся спагетти — подразумевается бесконечный запас. Тем не менее, философ может есть только тогда, когда держит две вилки — взятую справа и слева (альтернативная формулировка проблемы подразумевает миски с рисом и палочки для еды вместо тарелок со спагетти и вилок).

Каждый философ может взять ближайшую вилку (если она доступна) или положить — если он уже держит её. Взятие каждой вилки и возвращение её на стол являются отдельными действиями, которые должны выполняться одно за другим.

Вопрос задачи заключается в том, чтобы разработать модель поведения (параллельный алгоритм), при котором ни один из философов не будет голодать, то есть будет вечно чередовать приём пищи и размышления.

Разобраться в данной программе.

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace ConsoleApp
{
    class Program
    {
        public static Semaphore Semaphore = new Semaphore(3, 3);
        static readonly List<Fork> Fork = new List<Fork>();
        static readonly List<Philosopher> Ph = new List<Philosopher>();

        static void Main()
        {
            Console.WriteLine("Введите кол-во философов: ");
            var count = Convert.ToInt32(Console.ReadLine());

            for (var i = 0; i < count; i++)
            {
                Fork.Add(new Fork());
                Ph.Add(new Philosopher((i + 1).ToString(), i));
                new Thread(Ph[i].Start).Start(Fork);
            }
        }

        public class Philosopher
        {
            bool _isHunger;
            readonly string _philosopherName;
            readonly int _number;
            int _time;

            public Philosopher(string name, int number)
            {
                _philosopherName = name;
                _number = number;
            }

            void GetFork(IList<Fork> fork)
            {
                _time = new Random().Next(System.DateTime.Now.Millisecond);

                Console.WriteLine("Философ " + _philosopherName + " ждёт вилку" + "\t" +
                    ({0}мс)", _time);

                var first = _number;
                var second = (_number + 1) % (fork.Count - 1);

                if (fork[first].IsUsing || fork[second].IsUsing) return;

                fork[first].IsUsing = true;
            }
        }
    }
}
```

```

        fork[second].IsUsing = true;

        Console.WriteLine("Философ " + _philosopherName + " обедает" + "\t ({0}мс)",
            _time);
        Console.WriteLine("Вилки " + (first + 1) + " и " + (second + 1) + " заняты "
            + "\t ({0}мс)", _time);
        Console.WriteLine();
        Thread.Sleep(_time);

        fork[first].IsUsing = false;
        fork[second].IsUsing = false;
    }

    public void Start(object obj)
    {
        while (true)
        {
            Thread.Sleep(_time);
            ChangeStatus();
            if (_isHunger)
                GetFork((List<Fork>)obj);
        }
    }

    void ChangeStatus()
    {
        _isHunger = !_isHunger;
        if (!_isHunger)
            Console.WriteLine("Философ " + _philosopherName + " думает." + "\t
            ({0}мс)", _time);
    }
}

public class Fork
{
    public bool IsUsing { get; set; }
}

```

Параллельное программирование и библиотека TPL

В эпоху многоядерных машин, которые позволяют параллельно выполнять сразу несколько процессов, стандартных средств работы с потоками в .NET уже оказалось недостаточно. Поэтому во фреймворк .NET была добавлена библиотека параллельных задач TPL (Task Parallel Library), основной функционал которой располагается в пространстве имен System.Threading.Tasks. Данная библиотека позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах, если на целевом компьютере имеется несколько ядер. Кроме того, упрощается сама работа по созданию новых потоков. Поэтому начиная с .NET 4.0. рекомендуется использовать именно TPL и ее классы для создания многопоточных приложений, хотя стандартные средства и класс Thread по-прежнему находят широкое применение.

Задачи и класс Task

В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов .NET задача

представлена специальным классом - классом Task, который находится в пространстве имен System.Threading.Tasks. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

Для определения и запуска задачи можно использовать различные способы. Первый способ создание объекта Task и вызов у него метода Start:

```
Task task = new Task(() => Console.WriteLine("Hello Task!"));
task.Start();
```

В качестве параметра объект Task принимает делегат Action, то есть мы можем передать любое действие, которое соответствует данному делегату, например, лямбда-выражение, как в данном случае, или ссылку на какой-либо метод. То есть в данном случае при выполнении задачи на консоль будет выводиться строка "Hello Task!".

А метод Start() собственно запускает задачу.

Второй способ заключается в использовании статического метода Task.Factory.StartNew(). Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться. При этом этот метод сразу же запускает задачу:

```
Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!"));
```

В качестве результата метод возвращает запущенную задачу.

Третий способ определения и запуска задач представляет использование статического метода Task.Run():

```
Task task = Task.Run(() => Console.WriteLine("Hello Task!"));
```

Метод Task.Run() также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.

Ожидание задачи

Важно понимать, что задачи не выполняются последовательно. Первая запущенная задача может завершить свое выполнение после последней задачи.

Или рассмотрим еще один пример:

```
using System;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task task = new Task(Display);
            task.Start();

            Console.WriteLine("Завершение метода Main");

            Console.ReadLine();
        }
    }
}
```



```

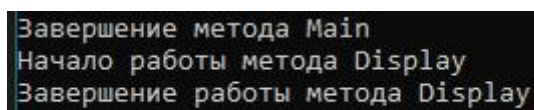
static void Display()
{
    Console.WriteLine("Начало работы метода Display");

    Console.WriteLine("Завершение работы метода Display");
}
}

```

Класс Task в качестве параметра принимает метод Display, который соответствует делегату Action. Далее чтобы запустить задачу, вызываем метод Start: task.Start(), и после этого метод Display начнет выполняться во вторичном потоке. В конце метода Main выводит некоторый маркер-строку, что метод Main завершился.

Однако в данном случае консольный вывод может выглядеть следующим образом:



```

Завершение метода Main
Начало работы метода Display
Завершение работы метода Display

```

То есть мы видим, что даже когда основной код в методе Main уже отработал, запущенная ранее задача еще не завершилась.

Чтобы указать, что метод Main должен подождать до конца выполнения задачи, нам надо использовать метод Wait:

```

static void Main(string[] args)
{
    Task task = new Task(Display);
    task.Start();
    task.Wait();

    Console.WriteLine("Завершение метода Main");

    Console.ReadLine();
}

```

Свойства класса Task

Класс Task имеет ряд свойств, с помощью которых мы можем получить информацию об объекте. Некоторые из них:

- AsyncState: возвращает объект состояния задачи
- CurrentId: возвращает идентификатор текущей задачи
- Exception: возвращает объект исключения, возникшего при выполнении задачи
- Status: возвращает статус задачи

Работа с классом Task

Вложенные задачи

Одна задача может запускать другую - вложенную задачу. При этом эти задачи выполняются независимо друг от друга. Например:

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            var outer = Task.Factory.StartNew(() =>          // внешняя задача
            {
                Console.WriteLine("Outer task starting...");
                var inner = Task.Factory.StartNew(() =>      // вложенная задача
                {
                    Console.WriteLine("Inner task starting...");
                    Thread.Sleep(2000);
                    Console.WriteLine("Inner task finished.");
                });
            });
            outer.Wait(); // ожидаем выполнения внешней задачи
            Console.WriteLine("End of Main");

            Console.ReadLine();
        }
    }
}

```

Несмотря на то, что здесь мы ожидаем выполнения внешней задачи, но вложенная задача может завершить выполнение даже после завершения метода Main:

```

Outer task starting...
Inner task starting...
End of Main
Inner task finished.

```

Если необходимо, чтобы вложенная задача выполнялась вместе с внешней, необходимо использовать значение `TaskCreationOptions.AttachedToParent`:

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            var outer = Task.Factory.StartNew(() =>          // внешняя задача
            {
                Console.WriteLine("Outer task starting...");
                var inner = Task.Factory.StartNew(() =>      // вложенная задача
                {
                    Console.WriteLine("Inner task starting...");
                    Thread.Sleep(2000);
                    Console.WriteLine("Inner task finished.");
                }, TaskCreationOptions.AttachedToParent);
            });
            outer.Wait(); // ожидаем выполнения внешней задачи
        }
    }
}

```

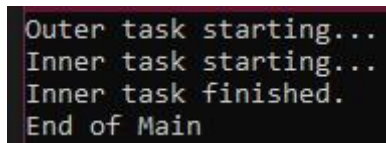
```

        Console.WriteLine("End of Main");

        Console.ReadLine();
    }
}

```

Консольный вывод:



```

Outer task starting...
Inner task starting...
Inner task finished.
End of Main

```

Массив задач

Также, как и с потоками, мы можем создать и запустить массив задач. Можно определить все задачи в массиве непосредственно через объект Task:

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task[] tasks1 = new Task[3]
            {
                new Task(() => Console.WriteLine("First Task")),
                new Task(() => Console.WriteLine("Second Task")),
                new Task(() => Console.WriteLine("Third Task"))
            };
            // запуск задач в массиве
            foreach (var t in tasks1)
                t.Start();

            Console.ReadLine();
        }
    }
}

```

Либо также можно использовать методы Task.Factory.StartNew или Task.Run и сразу запускать все задачи:

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task[] tasks2 = new Task[3];
            int j = 1;
            for (int i = 0; i < tasks2.Length; i++)
                tasks2[i] = Task.Factory.StartNew(() => Console.WriteLine($"Task {j++}"));
        }
    }
}

```

```

        Console.ReadLine();
    }
}

```

Но в любом случае мы опять же можем столкнуться, что все задачи из массива могут завершиться после того, как отработает метод Main, в котором запускаются эти задачи:

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task[] tasks1 = new Task[3]
            {
                new Task(() => Console.WriteLine("First Task")),
                new Task(() => Console.WriteLine("Second Task")),
                new Task(() => Console.WriteLine("Third Task"))
            };
            foreach (var t in tasks1)
                t.Start();

            Task[] tasks2 = new Task[3];
            int j = 1;
            for (int i = 0; i < tasks2.Length; i++)
                tasks2[i] = Task.Factory.StartNew(() => Console.WriteLine($"Task {j++}"));

            Console.WriteLine("Завершение метода Main");

            Console.ReadLine();
        }
    }
}

```

Один из возможных консольных выводов программы:

```

Завершение метода Main
Task 1
Task 2
First Task
Task 3
Third Task
Second Task

```

Если необходимо выполнять некоторый код лишь после того, как все задачи из массива завершатся, то применяется метод Task.WaitAll(tasks):

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {

```

```

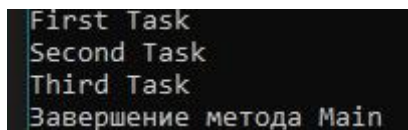
{
    static void Main(string[] args)
    {
        Task[] tasks1 = new Task[3]
        {
            new Task(() => Console.WriteLine("First Task")),
            new Task(() => Console.WriteLine("Second Task")),
            new Task(() => Console.WriteLine("Third Task"))
        };
        foreach (var t in tasks1)
            t.Start();
        Task.WaitAll(tasks1); // ожидаем завершения задач

        Console.WriteLine("Завершение метода Main");

        Console.ReadLine();
    }
}

```

В этом случае сначала завершатся все задачи, и лишь только потом будет выполняться последующий код из метода Main:



```

First Task
Second Task
Third Task
Завершение метода Main

```

В то же время порядок выполнения самих задач в массиве также недетерминирован.

Также мы можем применять метод Task.WaitAny(tasks). Он ждет, пока завершится хотя бы одна из массива задач.

Возвращение результатов из задач

Задачи могут не только выполняться как процедуры, но и возвращать определенные результаты:

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task<int> task1 = new Task<int>(() => Factorial(5));
            task1.Start();

            Console.WriteLine($"Факториал числа 5 равен {task1.Result}");

            Task<Book> task2 = new Task<Book>(() =>
            {
                return new Book { Title = "Война и мир", Author = "Л. Толстой" };
            });
            task2.Start();

            Book b = task2.Result; // ожидаем получение результата
            Console.WriteLine($"Название книги: {b.Title}, автор: {b.Author}");
        }
    }
}

```

```

        Console.ReadLine();
    }

    static int Factorial(int x)
    {
        int result = 1;

        for (int i = 1; i <= x; i++)
        {
            result *= i;
        }

        return result;
    }

    public class Book
    {
        public string Title { get; set; }
        public string Author { get; set; }
    }
}

```

Во-первых, чтобы задать возвращаемый из задачи тип объекта, мы должны типизировать Task. Например, Task<int> - в данном случае задача будет возвращать объект int.

И, во-вторых, в качестве задачи должен выполняться метод, возвращающий данный тип объекта. Например, в первом случае у нас в качестве задачи выполняется функция Factorial, которая принимает числовой параметр и также на выходе возвращает число.

Возвращаемое число будет храниться в свойстве Result: task1.Result. Нам не надо его приводить к типу int, оно уже само по себе будет представлять число.

То же самое и со второй задачей task2. В этом случае в лямбда-выражении возвращается объект Book. И также мы его получаем с помощью task2.Result

При этом при обращении к свойству Result программа текущий поток останавливает выполнение и ждет, когда будет получен результат из выполняемой задачи.

Задачи продолжения

Задачи продолжения или continuation task позволяют определить задачи, которые выполняются после завершения других задач. Благодаря этому мы можем вызвать после выполнения одной задачи несколько других, определить условия их вызова, передать из предыдущей задачи в следующую некоторые данные.

Задачи продолжения похожи на методы обратного вызова, но фактически являются обычными задачами Task. Посмотрим на примере:

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{

```

```

class Program
{
    static void Main(string[] args)
    {
        Task task1 = new Task(() => {
            Console.WriteLine($"Id задачи: {Task.CurrentId}");
        });

        // задача продолжения
        Task task2 = task1.ContinueWith(Display);

        task1.Start();

        // ждем окончания второй задачи
        task2.Wait();
        Console.WriteLine("Выполняется работа метода Main");
        Console.ReadLine();
    }

    static void Display(Task t)
    {
        Console.WriteLine($"Id задачи: {Task.CurrentId}");
        Console.WriteLine($"Id предыдущей задачи: {t.Id}");
        Thread.Sleep(3000);
    }
}

```

Первая задача задается с помощью лямбда-выражения, которое просто выводит id этой задачи. Вторая задача - задача продолжения задастся с помощью метода `ContinueWith`, который в качестве параметра принимает делегат `Action<Task>`. То есть метод `Display`, который передается в данный метод в качестве значения параметра, должен принимать параметр типа `Task`.

Благодаря передачи в метод параметра `Task`, мы можем получить различные свойства предыдущей задачи, как например, в данном случае получает ее `Id`.

И после завершения задачи `task1` сразу будет вызываться задача `task2`.

Также мы можем передавать конкретный результат работы предыдущей задачи:

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task<int> task1 = new Task<int>(() => Sum(4, 5));

            // задача продолжения
            Task task2 = task1.ContinueWith(sum => Display(sum.Result));

            task1.Start();

            // ждем окончания второй задачи
            task2.Wait();
            Console.WriteLine("End of Main");
            Console.ReadLine();
        }
    }
}

```

```

    }

    static int Sum(int a, int b) => a + b;
    static void Display(int sum)
    {
        Console.WriteLine($"Sum: {sum}");
    }
}

```

Подобным образом можно построить целую цепочку последовательно выполняющихся задач:

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task task1 = new Task(() => {
                Console.WriteLine($"Id задачи: {Task.CurrentId}");
            });

            // задача продолжения
            Task task2 = task1.ContinueWith(Display);

            Task task3 = task1.ContinueWith((Task t) =>
            {
                Console.WriteLine($"Id задачи: {Task.CurrentId}");
            });

            Task task4 = task2.ContinueWith((Task t) =>
            {
                Console.WriteLine($"Id задачи: {Task.CurrentId}");
            });

            task1.Start();

            Console.ReadLine();
        }

        static void Display(Task t)
        {
            Console.WriteLine($"Id задачи: {Task.CurrentId}");
            Console.WriteLine($"Id предыдущей задачи: {t.Id}");
            Thread.Sleep(3000);
        }
    }
}

```

Класс Parallel

Класс Parallel также является частью TPL и предназначен для упрощения параллельного выполнения кода. Parallel имеет ряд методов, которые позволяют распараллелить задачу.

Одним из методов, позволяющих параллельное выполнение задач, является метод `Invoke`:

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Parallel.Invoke(Display,
                () => {
                    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
                    Thread.Sleep(3000);
                },
                () => Factorial(5));

            Console.ReadLine();
        }

        static void Display()
        {
            Console.WriteLine($"Выполняется задача {Task.CurrentId}");
            Thread.Sleep(3000);
        }

        static void Factorial(int x)
        {
            int result = 1;

            for (int i = 1; i <= x; i++)
            {
                result *= i;
            }
            Console.WriteLine($"Выполняется задача {Task.CurrentId}");
            Thread.Sleep(3000);
            Console.WriteLine($"Результат {result}");
        }
    }
}
```

Метод `Parallel.Invoke` в качестве параметра принимает массив объектов `Action`, то есть мы можем передать в данный метод набор методов, которые будут вызываться при его выполнении. Количество методов может быть различным, но в данном случае мы определяем выполнение трех методов. Опять же, как и в случае с классом `Task` мы можем передать либо название метода, либо лямбда-выражение.

И таким образом, при наличии нескольких ядер на целевой машине данные методы будут выполняться параллельно на различных ядрах.

Parallel.For

Метод `Parallel.For` позволяет выполнять итерации цикла параллельно. Он имеет следующее определение: `For(int, int, Action<int>)`, где первый параметр задает начальный индекс элемента в цикле, а второй параметр - конечный индекс.

Третий параметр - делегат Action - указывает на метод, который будет выполняться один раз за итерацию:

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Parallel.For(1, 10, Factorial);

            Console.ReadLine();
        }

        static void Factorial(int x)
        {
            int result = 1;

            for (int i = 1; i <= x; i++)
            {
                result *= i;
            }
            Console.WriteLine($"Выполняется задача {Task.CurrentId}");
            Console.WriteLine($"Факториал числа {x} равен {result}");
            Thread.Sleep(3000);
        }
    }
}
```

В данном случае в качестве первого параметра в метод Parallel.For передается число 1, а в качестве второго - число 10. Таким образом, метод будет вести итерацию с 1 до 9 включительно. Третий параметр представляет метод, подсчитывающий факториал числа. Так как этот параметр представляет тип Action<int>, то этот метод в качестве параметра должен принимать объект int.

А в качестве значения параметра в этот метод передается счетчик, который проходит в цикле от 1 до 9 включительно. И метод Factorial, таким образом, вызывается 9 раз.

Parallel.ForEach

Метод Parallel.ForEach осуществляет итерацию по коллекции, реализующей интерфейс IEnumerable, подобно циклу foreach, только осуществляет параллельное выполнение перебора. Он имеет следующее определение: ParallelLoopResult ForEach<TSource>(IEnumerable<TSource> source, Action<TSource> body), где первый параметр представляет перебираемую коллекцию, а второй параметр - делегат, выполняющийся один раз за итерацию для каждого перебираемого элемента коллекции.

На выходе метод возвращает структуру ParallelLoopResult, которая содержит информацию о выполнении цикла.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
```

```

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            ParallelLoopResult result = Parallel.ForEach<int>(new List<int>() { 1, 3, 5,
8 },
                    Factorial);

            Console.ReadLine();
        }
        static void Factorial(int x)
        {
            int result = 1;

            for (int i = 1; i <= x; i++)
            {
                result *= i;
            }
            Console.WriteLine($"Выполняется задача {Task.CurrentId}");
            Console.WriteLine($"Факториал числа {x} равен {result}");
            Thread.Sleep(3000);
        }
    }
}

```

В данном случае поскольку мы используем коллекцию объектов `int`, то и метод, который мы передаем в качестве второго параметра, должен в качестве параметра принимать значение `int`.

Выход из цикла

В стандартных циклах `for` и `foreach` предусмотрен преждевременный выход из цикла с помощью оператора `break`. В методах `Parallel.ForEach` и `Parallel.For` мы также можем, не дожидаясь окончания цикла, выйти из него:

```

using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            ParallelLoopResult result = Parallel.For(1, 10, Factorial);

            if (!result.IsCompleted)
                Console.WriteLine($"Выполнение цикла завершено на итерации
{result.LowestBreakIteration}");
            Console.ReadLine();
        }
        static void Factorial(int x, ParallelLoopState pls)
        {
            int result = 1;

            for (int i = 1; i <= x; i++)
            {

```

```

        result *= i;
        if (i == 5)
            pls.Break();
    }
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Console.WriteLine($"Факториал числа {x} равен {result}");
}
}
}

```

Здесь метод Factorial, обрабатывающий каждую итерацию, принимает дополнительный параметр - объект ParallelLoopState. И если счетчик в цикле достигнет значения 5, вызывается метод Break. Благодаря чему система осуществит выход и прекратит выполнение метода Parallel.For при первом удобном случае.

Методы Parallel.ForEach и Parallel.For возвращают объект ParallelLoopResult, наиболее значимыми свойствами которого являются два следующих:

IsCompleted: определяет, завершилось ли полное выполнение параллельного цикла

LowestBreakIteration: возвращает индекс, на котором произошло прерывание работы цикла

Так как у нас на индексе равном 5 происходит прерывание, то свойство IsCompleted будет иметь значение false, а LowestBreakIteration будет равно 5.

Отмена задач и параллельных операций. CancellationToken

Параллельное выполнение задач может занимать много времени. И иногда может возникнуть необходимость прервать выполняемую задачу. Для этого .NET предоставляет класс CancellationToken:

```

using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            CancellationTokenSource cancellationTokenSource = new CancellationTokenSource();
            CancellationToken token = cancellationTokenSource.Token;
            int number = 6;

            Task task1 = new Task(() =>
            {
                int result = 1;
                for (int i = 1; i <= number; i++)
                {
                    if (token.IsCancellationRequested)
                    {
                        Console.WriteLine("Операция прервана");
                        return;
                    }
                }
            });
        }
    }
}

```

```

        }

        result *= i;
        Console.WriteLine($"Факториал числа {i} равен {result}");
        Thread.Sleep(5000);
    }
});
task1.Start();

Console.WriteLine("Введите Y для отмены операции:");
string s = Console.ReadLine();
if (s == "Y")
    cancellationTokenSource.Cancel();

Console.Read();
}
}
}

```

Для отмены операции нам надо создать и использовать токен. Вначале создается объект `CancellationTokenSource`:

```
CancellationTokenSource cancellationTokenSource = new CancellationTokenSource();
```

Затем из него получаем сам токен:

```
CancellationToken token = cancellationTokenSource.Token;
```

Чтобы отменить операцию, необходимо вызвать метод `Cancel()` у объекта `CancellationTokenSource`:

```
cancellationTokenSource.Cancel();
```

В самой операции мы можем отловить выставление токена с помощью условной конструкции:

```

if (token.IsCancellationRequested)
{
    Console.WriteLine("Операция прервана");
    return;
}

```

Если был вызван метод `cancellationTokenSource.Cancel()`, то выражение `token.IsCancellationRequested` возвращает `true`.

Если операция представляет внешний метод, то ему надо передавать в качестве одного из параметров токен:

```

using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            CancellationTokenSource cancellationTokenSource = new CancellationTokenSource();
            CancellationToken token = cancellationTokenSource.Token;

            Task task1 = new Task(() => Factorial(5, token));

```

```

        task1.Start();

        Console.WriteLine("Введите 0 для отмены операции:");
        string s = Console.ReadLine();
        if (s == "Y")
            cancelTokenSource.Cancel();

        Console.ReadLine();
    }

    static void Factorial(int x, CancellationToken token)
    {
        int result = 1;
        for (int i = 1; i <= x; i++)
        {
            if (token.IsCancellationRequested)
            {
                Console.WriteLine("Операция прервана токеном");
                return;
            }

            result *= i;
            Console.WriteLine($"Факториал числа {x} равен {result}");
            Thread.Sleep(5000);
        }
    }
}

```

Отмена параллельных операций Parallel

Для отмены выполнения параллельных операций, запущенных с помощью методов `Parallel.For()` и `Parallel.ForEach()`, можно использовать перегруженные версии данных методов, которые принимают в качестве параметра объект `ParallelOptions`. Данный объект позволяет установить токен:

```

using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
            CancellationToken token = cancelTokenSource.Token;

            new Task(() =>
            {
                Thread.Sleep(400);
                cancelTokenSource.Cancel();
            }).Start();

            try
            {
                Parallel.ForEach<int>(new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8 },
                                     new ParallelOptions { CancellationToken = token },
                Factorial);
            }
            // или так
        }
    }
}

```

```

        //Parallel.For(1, 8, new ParallelOptions { CancellationTokен = token },
Factorial);
    }
    catch (OperationCanceledException ex)
    {
        Console.WriteLine("Операция прервана");
    }
    finally
    {
        cancellationTokenSource.Dispose();
    }

    Console.ReadLine();
}
static void Factorial(int x)
{
    int result = 1;

    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    Console.WriteLine($"Факториал числа {x} равен {result}");
    Thread.Sleep(3000);
}
}
}

```

В параллельной запущенной задаче через 400 миллисекунд происходит вызов `cancellationTokenSource.Cancel()`, в результате программа выбрасывает исключение `OperationCanceledException`, и выполнение параллельных операций прекращается.