**МИНОБРНАУКИ РОССИИ**

**Федеральное государственное бюджетное**

**образовательное учреждение высшего образования**

**«Тульский государственный университет»**

**Институт «Прикладной математики и компьютерных наук»**

**Кафедра «Вычислительной техники»**

**Утверждено на заседании кафедры**

**«Вычислительной техники»**

**«28» июня 2019 г., протокол №12**

**Заведующий кафедрой**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.Н.Ивутин**

**Сборник методических указания к лаборатоным работам**

**учебной дисциплины**

**«*Программирование*»**

**основной профессиональной образовательной программы**

**высшего** **образования – программы бакалавриата**

по направлению подготовки

***09.03.01 Информатика и вычислительная техника***

с направленностью

***Системы автоматизированного проектирования; Вычислительные машины, комплексы, системы и сети;***

***Программное обеспечение средств вычислительной техники и автоматизированных систем; Автоматизированные системы обработки информации и управления***

Форма обучения: *очная*

Идентификационный номер образовательной программы: 090301-02-19

Тула 2019 год

Сборник МУ составлен доц. каф ВТ ***Савиным Н.И.*** и обсужден на заседании кафедры Вычислительной техники института Прикладной математики и компьютерных наук (протокол заседания кафедры №12 от «28» июня 2019 г.)

***Зав. кафедрой А.Н.Ивутин***

# Оглавление

[Оглавление 3](#_Toc3870162)

[Лабораторная работа 1. Платформа Microsoft .Net Framework 4](#_Toc3870163)

[Лабораторная работа 2. Технология объектно-ориентированного программирования 11](#_Toc3870164)

[Лабораторная работа 3. Операции в программах на языке C# 24](#_Toc3870165)

[Лабораторная работа 4. Операторы языка C# 31](#_Toc3870166)

[Лабораторная работа 5. C#. Введение в методы 45](#_Toc3870167)

[Лабораторная работа 6. C#. Рекурсия 56](#_Toc3870168)

[Лабораторная работа 7. Обработка исключений 65](#_Toc3870169)

[Лабораторная работа 8. Массивы 76](#_Toc3870170)

[Лабораторная работа 9. Символы и строки 91](#_Toc3870171)

[Лабораторная работа 10. С#. Классы 102](#_Toc3870172)

[Лабораторная работа 11. Классы: деструкторы, индексаторы, операции класса, операции преобразования типов 116](#_Toc3870173)

[Лабораторная работа 12. Наследование 128](#_Toc3870174)

[Лабораторная работа 13. Коллекции 140](#_Toc3870175)

[Лабораторная работа 14. С#-.Организация системы ввода-вывода 154](#_Toc3870176)

[Лабораторная работа 15. C#. Файловая система 166](#_Toc3870177)

[Лабораторная работа 16. Интерфейсы. Контейнерные классы. 172](#_Toc3870178)

[Лабораторная работа 17. Делегаты и события 191](#_Toc3870179)

[Лабораторная работа 18. Регулярные выражения 207](#_Toc3870180)

[Приложение Оформление лабораторных работ 213](#_Toc3870181)

[1. Цель работы 2](#_Toc3870182)

[2. Теоретические сведения. 2](#_Toc3870183)

[3. Задания на лабораторную работу. 2](#_Toc3870184)

[Список литературы 3](#_Toc3870185)

[Приложения 4](#_Toc3870186)

[Список литературы 5](#_Toc3870187)

# Платформа Microsoft .Net Framework

## Теоретические сведения.

Совокупность средств, с помощью которых программы пишутся, корректируются, преобразуются в машинные коды, отлаживаются и запускаются, называют средой разработки или оболочкой. Платформа .Net или .Net Framework – это объединение разрозненных технологий, которые позволяют разрабатывать разнотипные приложения на различных языках программирования под различные операционные системы.

.*NET* Framework является надстройкой над операционной системой, в качестве которой может выступать любая версия *Windows*, и состоит из ряда компонентов. .*NET* Framework включает в себя:

1. Четыре официальных языка: С#, VB.NET, Managed C++ и JScript .NET.
2. Общеязыковую объектно-ориентированную среду выполнения CLR (Common Language Runtime), совместно используемую этими языками для создания приложений.
3. Ряд связанных между собой библиотек классов под общим именем *FCL* (Framework *Class Library*).

Основным компонентом платформы .*NET* Framework является общеязыковая среда выполнения программ *CLR*. К функциям *CLR* относятся:

1. двухшаговая компиляция: преобразование программы, написанной на одном из языков программирования в управляемый код на *промежуточном языке* (*Microsoft Intermediate Language*, *MSIL*, или просто IL), а затем преобразование IL-кода в машинный код конкретного процессора, который выполняется с помощью виртуальной машины или JIT-компилятора (Just In *Time compiler* - компилирование точно к нужному моменту);
2. управление кодом: загрузка и выполнение уже готового IL-кода с помощью JIT-компилятора;
3. осуществление доступа к метаданным с целью проверки *безопасности кода*;
4. управление памятью при размещении объектов с помощью сборщика мусора (*Garbage Collector*);
5. обработка исключений и исключительных ситуаций, включая межъязыковые исключения;
6. осуществление взаимодействия между управляемым кодом (код, созданный для СLR) и неуправляемым кодом;
7. поддержка сервисов для разработки разнотипных приложений.

Следующим компонентом .*Net* Framework является *FCL* – библиотека классов платформы. Эта библиотека разбита на несколько модулей таким образом, что имеется возможность использовать ту или иную ее часть в зависимости от требуемых результатов. Так, например, в одном из модулей содержатся классы, из которых можно построить *Windows*-приложения, в другом — классы, необходимые для организации работы в сети и т.д.

Часть *FCL* посвящена описанию *базисных типов*. Тип — это способ представления данных; *определение* наиболее фундаментальных из них облегчает совместное использование языков программирования с помощью .*NET* Framework. Все вместе это называется *Common Type System* (*CTS* — единая *система типов*).

Кроме того, *библиотека FCL* включает в себя Common *Language* *Specification* (*CLS* – общая языковая спецификация), которая устанавливает: основные правила языковой интеграции. Спецификация *CLS* определяет минимальные требования, предъявляемые к языку платформы .*NET*. Компиляторы, удовлетворяющие этой спецификации, создают объекты, способные взаимодействовать друг с другом. Поэтому любой язык, соответствующий требованиям *CLS*, может использовать все возможности библиотеки *FCL*.

Как уже отмечалось, основными языками, предназначенными для платформы .*NET* Framework, являются С#, VB.*NET*, Managed C++ и JScript .*NET*. Для данных языков Microsoft предлагает собственные компиляторы, переводящие программу в IL-код, который выполняется JIT-компилятором среды *CLR*. Кроме Microsoft, еще несколько компаний и академических организаций создали свои собственные компиляторы, генерирующие код, работающий в *CLR*. На сегодняшний момент известны компиляторы для *Pascal*, Cobol, *Lisp*, Perl, Prolog и т.д. Это означает, что можно написать программу, например, на языке *Pascal*, а затем, воспользовавшись соответствующим компилятором, создать *управляемый код*, который будет работать в среде *CLR*.

## Приложение, проект, решение

.NET Framework не налагает никаких ограничений на возможные типы создаваемых приложенийю. Наиболее часто встречающиеся типы приложений:

1. Консольные приложения позволяют выполнять вывод на "консоль", то есть в окно командного процессора.
2. Windows-приложения, использующие элементы интерфейса Windows, включая формы, кнопки, флажки и т.д.
3. Web-приложения представляют собой web-страницы, которые могут просматриваться любым web-браузером.
4. Web-сервисы представляют собой распределенные приложения, которые позволяют обмениваться по Интернету практически любыми данными с использованием единого синтаксиса независимо от того, какой язык программирования применялся при создании web-службы и на какой системе она размещена.

Приложение, находящееся в процессе разработки, называется **проектом**. Несколько приложений могут быть объединены в **решение** (solution).

Средой разработки приложений является Visual Studio .Net.

## Среда разработки Visual Studio .Net

## Консольные приложения на языке C#

Можно создавать файлы с исходным кодом на С# с помощью обычного текстового редактора, например, Блокнота, и компилировать их в *управляемые модули* с помощью компилятора командной строки, который является составной частью .*NET* Framework. Другой подход - использовать VS, потому что:

1. VS автоматически выполняет все шаги, необходимые для компиляции исходного кода.
2. Текстовый редактор VS настроен для работы с теми языками, которые поддерживаются VS, например С#, поэтому он *может интеллектуально обнаруживать ошибки и подсказывать в процессе ввода*, какой именно код необходим.
3. В состав VS входят программы, позволяющие создавать Windows- и Web-приложения путем простого перетаскивания мышью элементов пользовательского интерфейса.
4. Многие типы проектов, создание которых возможно на С#, могут разрабатываться на основе "каркасного" кода, заранее включаемого в программу. Вместо того чтобы каждый раз начинать с нуля, VS позволяет использовать уже имеющиеся файлы с исходным кодом, что уменьшает временные затраты на создание проекта.

Создание первого проекта

Для создания проекта следует запустить VS, а затем в главном меню VS выбрать команду **File – New - Project.** После чего откроется диалоговое меню **New Project**(см. [рис.1](https://www.intuit.ru/studies/courses/486/342/lecture/8225?page=1#image.1.1)).

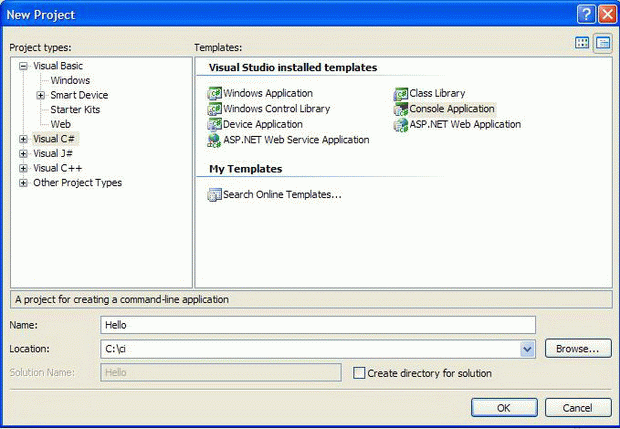


Рис. 1.1.

В поле **Project types** следует выбрать *Visual C#*, в поле **Templates** – *Console Application*.

В строчке **Name**введите имя приложения *Hello*. Обратите внимание на то, что это же имя появится в строчке **Solution Name**. Уберите галочку в поле **Create directory for Application** (пока мы создаем простое приложение, и нам нет необходимости усложнять его структуру).

В строке **Location** определите положение на диске, куда нужно сохранять ваш проект. И нажмите кнопку **OK**. Примерный вид экрана изображен на [рис.2](https://www.intuit.ru/studies/courses/486/342/lecture/8225?page=1#image.1.2)

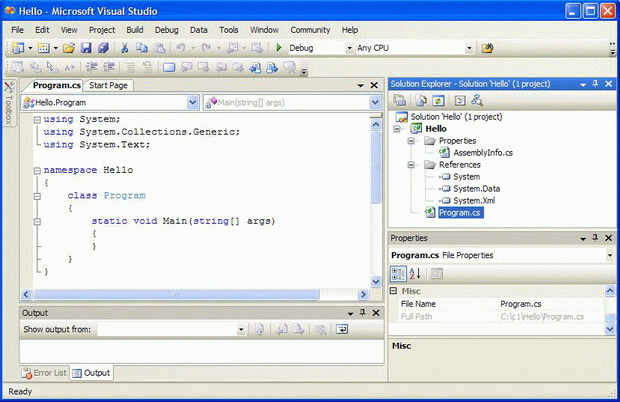


Рис. 2.

В правой верхней части располагается окно управления проектом **Solution Explorer**. Если оно закрыто, то его можно включить командой **View - Solution Explorer**. В этом окне перечислены все ресурсы, входящие в проект:

1. AssemblyInfo.cs – информация о *сборке*.

Компилятор в качестве результата своего выполнения создает так называемую *сборку* – файл с расширением exe или dll, который содержит IL-код и метаданные.

1. System, System.Data, System.Xml – ссылки на стандартные библиотеки.
2. Program.cs - текст программы на языке C#.

В правой нижней части экрана располагается окно свойств **Properties**. Если оно закрыто, то его можно включить командой **View - Properties**. В этом окне отображаются важнейшие характеристики выделенного элемента.

Основное пространство экрана занимает окно редактора, в котором располагается текст программы, созданный средой автоматически. Текст представляет собой каркас, в который программист будет добавлять нужный код. При этом зарезервированные слова отображаются синим цветом, комментарии – зеленым, основной текст – черным.

Текст структурирован. Щелкнув на знак минус, мы скроем блок кода, щелкнув на знаке плюс – откроем.

Структура папки, содержащую проект (см. [рис. 3](https://www.intuit.ru/studies/courses/486/342/lecture/8225?page=1#image.1.3)). Файлы, выделенные жирным шрифтом, появятся только после компиляции.

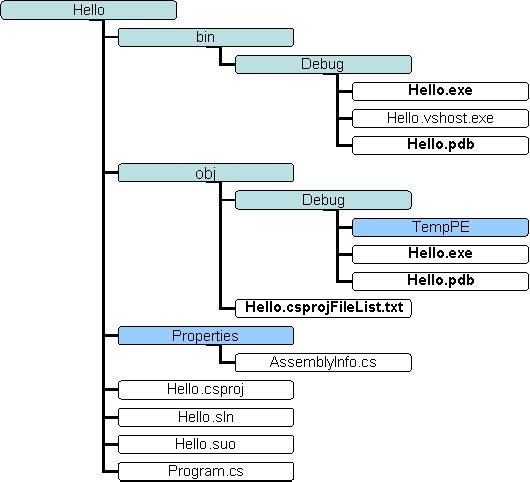


Рис. 3.

На данном этапе особый интерес будут представлять следующие файлы:

1. *Hello.sln*– основной файл, отвечающий за весь проект. Если необходимо открыть проект для редактирования, то нужно выбрать именно этот файл. Остальные файлы откроются автоматически.
2. *Program.cs* – файл, в котором содержится исходный код - код, написанный на языке С#. Именно с этим файлом предстоит непосредственно работать.
3. *Hello.exe* – файл, в котором содержатся сгенерированный IL-код и метаданные проекта. Другими словами, этот файл и есть готовое приложение, которое может выполняться на любом компьютере, на котором установлена платформа .Net.

Текст программы.

**using System** – это директива, которая разрешает использовать имена стандартных классов из пространства имен **System** непосредственно без указания имени пространства, в котором они были определены.

## Пространство имен

Ключевое слово **namespace** создает для проекта свое собственное пространство имен, которое по умолчанию называется именем проекта. В данном случае пространство имен называется Hello. Однако программист вправе указать другое имя. Пространство имен ограничивает область применения имен, делая его осмысленным только в рамках данного пространства. Это сделано для того, чтобы можно было давать имена программным объектам, не заботясь о том, что они совпадут с именами в других приложениях. Таким образом, пространства имен позволяют избегать конфликта имен программных объектов, что особенно важно при взаимодействии приложений.

С# - объектно-ориентированный язык, поэтому написанная на нем программа будет представлять собой совокупность взаимодействующих между собой классов. Автоматически был создан класс с именем Program (в других версиях среды может создаваться класс с именем Class1 ).

Данный класс содержит только один метод - метод Main(). Метод Main() является точкой входа в программу, т.е. именно с данного метода начнется выполнение приложения. Каждая программа на языке С# должна иметь метод Main ().

**Методы Main()** .Технически возможно иметь несколько методов Main() в одной программе, в этом случае потребуется с помощью параметра командной строки сообщить компилятору С#, какой именно метод Main() является точкой входа в программу.

Метод Main() имеет одну важную особенность. Перед объявлением типа возвращаемого значения void (который означает, что метод не возвращает значение) стоит ключевое слово static, которое означает что метод Main() можно вызывать, не создавая объект типа Program.

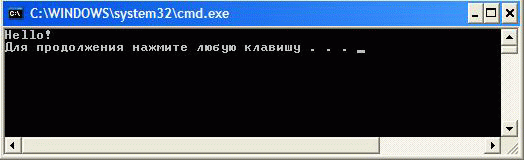
**Замечание**. В некоторых версиях требуется, чтобы перед словом static стояло слово public.

Добавим в метод следующий код:

Console.WriteLine("Hello!");

Здесь Console имя стандартного класса из пространства имен System. Его метод WriteLine выводит на экран текст, заданный в кавычках

Для запуска программы следует нажать клавишу F5 или выполнить команду **Debug-Start Debugging**.Если программа выполнена без ошибок, то сообщение выведется в консольное окно, которое мелькнет и быстро закроется. Чтобы просмотреть сообщение в нормальном режиме нужно нажать клавиши Ctrl+F5 или выполнить команду **Debug-Start Without Debugging**.В данном случае откроется следующее консольное окно:



Если код программы будет содержать ошибки, например, пропущена точка с запятой после команды вывода, то после нажатия клавиши F5 откроется диалоговое окно, в котором выведется сообщение о том, что обнаружена ошибка, и вопрос, продолжать ли работу дальше. Если вы ответите **Yes**, то будет выполнена предыдущая удачно скомпилированная версия программы. Иначе процесс будет остановлен и управление передано окну списка ошибок **Error List**.

## Компиляция и выполнение программы в среде CLR

Все *CLR*-совместимые компиляторы генерируют IL-код, который также называется управляемым модулем, потому что *CLR*  управляет его жизненным циклом и выполнением. Составные части управляемого модуля:

1. *Заголовок* *PE32 или PE32+*: Файл с заголовком в формате PE32 может выполняться в 32- или 64-разрядной ОС, а с заголовком PE32+ только в 64-разрядной ОС. Заголовок показывает тип файла: GUI, *CUI* или DLL, он также имеет временную метку, показывающую, когда файл был собран. Для модулей, содержащих только IL-код, основной объем информации в РЕ-заголовке игнорируется, Для модулей, содержащих процессорный код, этот заголовок содержит сведения о процессорном коде.
2. *Заголовок CLR*: Содержит информацию, которая превращает этот модуль в управляемый. Заголовок включает нужную версию СLR, некоторые флаги, метку метаданных, точки входа в *управляемый модуль* (метод Main), месторасположение и размер метаданных модуля, ресурсов и т.д.
3. *Метаданные* - это набор таблиц данных, описывающих то, что определено в модуле. Есть два основных вида таблиц: описывающие типы и члены, определенные в вашем исходном коде, и описывающие типы и члены, на которые имеются ссылки в вашем исходном коде. Метаданные служат многим целям:
   * устраняют необходимость в заголовочных и библиотечных файлах при компиляции, так как все сведения о типах и членах, на которые есть ссылки, содержатся в файле с IL-кодом, в котором они реализованы. Компиляторы могут читать метаданные прямо из управляемых модулей.
   * при компиляции IL-кода в машинный код CLR выполняет верификацию (проверку "безопасности" выполнения кода) используя метаданные, например, нужное ли число параметров передается методу, корректны ли их типы, правильно ли используется возвращаемое значение и т.д.
   * позволяют сборщику мусора отслеживать жизненный цикл объектов и т.д.
4. IL-код: управляемый код, создаваемый компилятором при компиляции исходного кода. Во время исполнения CLR компилирует IL-код в команды процессора.

*По* умолчанию *CLR*-совместимые компиляторы генерируют *управляемый код*, *безопасность* выполнения которого поддается проверке средой *CLR*. Вместе с тем возможно разрабатывать неуправляемый или "небезопасный" код, которому разрешается работать непосредственно с адресами памяти и управлять байтами в этих адресах. Эта возможность, обычно полезна при взаимодействии с неуправляемым кодом или при необходимости добиться максимальной производительности при выполнении критически важных алгоритмов. Однако использовать неуправляемый код довольно рискованно, т.к. он способен разрушить существующие структуры данных.

Чтобы понять принцип выполнения программы в среде *CLR* рассмотрим небольшой пример (Рис. 4).

Непосредственно перед исполнением функции Main CLR находит все типы, на которые ссылается ее код. В нашем случае метод Mainссылается на единственный тип — Console, и CLR выделяет единственную внутреннюю структуру WriteLine.

Когда Main первый раз обращается к WriteLine, вызывается *функция* JITCompiler (условное название), которая отвечает за компиляцию IL-кода вызываемого метода в собственные команды процессора. Функции JITCompiler известен вызываемый метод и тип, в котором он определен. JITCompiler ищет в метаданных соответствующей сборки IL-код вызываемого метода, затем проверяет и компилирует IL-код в собственные команды процессора, которые сохраняются в динамически выделенном блоке памяти. После этого JITCompiler возвращается к внутренней структуре данных типа и заменяет *адрес* вызываемого метода адресом блока памяти, содержащего собственные команды процессора. В завершение JITCompiler передает управление коду в этом блоке памяти. Далее управление возвращается в Main, который продолжает работу в обычном порядке.

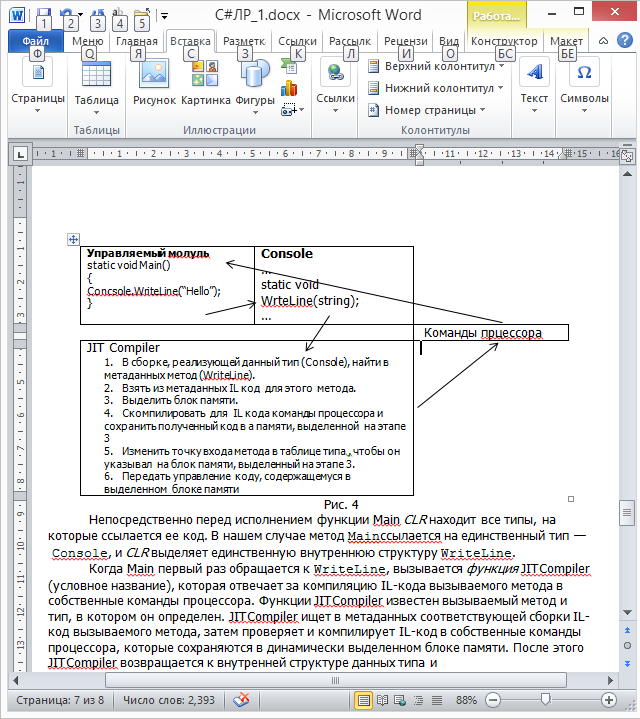


Рис. 4

Затем Main обращается к WriteLine вторично. К этому моменту код WriteLine уже проверен и скомпилирован, так что производится обращение к блоку памяти, минуя вызов JITCompiler. Отработав, метод WriteLine возвращает управление Main.

## Динамическая отладка программ.

Для динамической отладки программ используется механизм контрольных точек.

Контрольная точка – установленная с помощью средств отладчика отметка оператора программы, на котором программа остановится при достижении потоком кода данного оператора. После останова программы становится возможным доступ к переменным программы – в *окнах локальных и видимых переменных* а также в окне *контрольного значения*.

Для контрольной точки может быть установлено условие останова.

Вопросы.

1. Чем отличается метод Console.WriteLine() от метода Console.Write()?
2. Чем отличается метод Console.ReadLine() от метода Console.Read()?
3. Какой тип имеет литеральная константа 3.2?
4. Как явным образом уточнить тип литеральной константы?
5. Что обозначается константой NaN? И в каких случаях компилятором используется данная константа?

# Технология объектно-ориентированного программирования

## Теоретические сведения.

В основе языка С# лежит технология объектно-ориентированного программирования (*ООП*). Все программы на языке С# в большей или меньшей степени являются объектно-ориентированными.

### Основные понятия в рамках *ООП*.

*ООП* основано на таких понятиях как "*класс*", "*объект*", "*интерфейс*", "*инкапсуляция*", "*наследование*", "*полиморфизм*", "событие".

*Объект*в программе *—*это *абстракция* реального объекта. *Объект* обладает атрибутами, поведением и индивидуальностью. *Атрибуты* определяют основные черты объекта, *поведение* — действия над объектом, *индивидуальность* — отличие одного объекта от другого с такими же атрибутами по их конкретным значениям. Например: два вектора, у обоих компоненты x и y, но у одного x=2, y=3. А у другого x= -2, y=4.

*Класс* – это множество объектов с одинаковыми атрибутами и поведением, представляемое в языке программирования в виде абстрактного типа данных, который включает в себя члены класса.

### Члены класса:

* *поля* – непосредственно данные определенного типа для описания атрибутов;
* *методы* - функции, предназначенные для обработки внутренних данных объекта данного класса;
* *свойства* – это специальные поля данных, с помощью которых, можно управлять поведением объектов данного класса.

*Класс*  служит типом для создания объектов или, другими словами, *объект* является экземпляром класса.

Важным свойством объекта является его обособленность. Детали реализации объекта, то есть внутренние структуры данных и алгоритмы их обработки, скрыты от пользователя и недоступны для непреднамеренного изменения. *Объект* используется через его *интерфейс* - совокупность правил доступа. Скрытие деталей реализации называется *инкапсуляцией*.

В *ООП* данные и методы одного класса могут передаваться другим классам с помощью механизма *наследования*. Порожденный *класс*(*потомок*), наследующий характеристики другого класса, обладает теми же возможностями, что и *класс* (предок), от которого он порожден. При этом *класс-предок* остается без изменения, а классу-потомку можно добавлять новые элементы (поля, методы, свойства) или изменять унаследованные методы. Благодаря этому *класс*-*потомок* обладает большими возможностями, чем предок. Так, например, в программе на C# все классы порождены от корневого класса *System.Object*.

Классы-потомки некоторого класса являются разновидностями этого класса-предка. (Другими словами, *класс-предок* является обобщением своих потомков). Это означает, что к объектам классов-потомков можно обращаться с помощью одного и того же имени (но при этом могут выполняться различные действия) — что составляет суть *полиморфизма*.

Программу, построенную на принципах *ООП*, можно представить как совокупность взаимодействующих объектов. *Объект* А воздействует на *объект* Б, и для Б возникает *событие*, на которое Б отреагирует либо ответным воздействием на А, либо воздействием на *объект* В. Если А — внешний для системы *объект*, то Б — интерфейсный *объект* (отвечающий за взаимодействие системы с внешним миром).

*Операционная система* *Windows* — объектно-ориентированная система, в которой определены классы для производства объектов, обеспечивающих, в частности, *интерфейс* с пользователем. *Программа*, написанная под *Windows*, обращается к ней с предложением какой интерфейсный *объект* на каком именно месте создать — так строится внешний вид (*интерфейс*) программы. Эти интерфейсные объекты кажутся принадлежащими программе, но на самом деле они — часть *Windows*: именно она отвечает за их базовый внешний вид и поведение. Поэтому, когда *пользователь* воздействует на интерфейсный *объект* программы (нажимает кнопку, выбирает *пункт* *меню* и т.п.), для этого объекта происходит событие и *Windows* переводит это событие в сообщение для программы. При написании программы предусматривается однозначная *реакция* на это сообщение в виде метода, а в методе вызываются методы других объектов. Т.е. воздействие на интерфейсный *объект* приводит к появлению в недрах *Windows* сообщения, которое, приходя в программу, запускает цепочку взаимодействий внутренних для нее объектов. Таким образом, *ООП* *программа* фактически встраивается в ОС *Windows*.

Если рассмотреть программу:

class Program //класс

{

static void Main () //метод класса

{

Console.WriteLine("Hello!!!");

}

}

*Программа* содержит *класс* Program с единственным статическим (static) методом Main, что позволяет обращаться к данному методу класса без создания его экземпляра.

**Состав языка**

**Алфавит и лексемы**

Все тексты на языке пишутся с помощью его *алфавита*. В C# используется кодировка символов Unicode. *Кодировкой*, или *кодовой таблицей* (character set), называется соответствие между символами и кодирующими их числами. Кодировка Unicode позволяет представить символы всех существующих алфавитов одновременно. Каждому символу соответствует свой уникальный код.

Алфавит C# включает:

* *буквы*(латинские и национальных алфавитов) и *символ подчеркивания* ( \_ ), который употребляется наряду с буквами;
* *цифры* ;
* *специальные символы*, например +, \*, { и & ;
* *пробельные символы* (пробел и символы табуляции);
* *символы перевода строки*.
* Из символов составляются более крупные строительные блоки: лексемы, директивы препроцессора и комментарии.
* *Лексема* (token) — это минимальная единица языка, имеющая самостоятельный смысл. Существуют следующие виды лексем:
* *имена* ( *идентификаторы* );
* *ключевые слова* ;
* *знаки операций* ;
* *разделители* ;
* *литералы*( *константы* ).

Лексемы языка программирования аналогичны словам естественного языка. Например, лексемами являются число 128 (но не его часть 12), имя Vasia, ключевое слово goto и знак операции сложения  +. Далее мы рассмотрим лексемы подробнее.

*Директивы препроцессора* пришли в C# из его предшественника — языка С++. Препроцессором называется предварительная стадия компиляции, на которой формируется окончательный вид исходного текста программы. Например, с помощью директив (инструкций, команд) препроцессора можно включить или выключить из процесса компиляции фрагменты кода. Директивы препроцессора не играют в C# такой важной роли, как в С++.

*Комментарии* предназначены для записи пояснений к программе и формирования документации.

Из лексем составляются выражения и операторы. *Выражение* задает правило вычисления некоторого значения. Например, выражение a + b задает правило вычисления суммы двух величин.

*Оператор* задает законченное описание некоторого действия, данных или элемента программы. Например:

int a;

Это — оператор описания целочисленной переменной a.

**Идентификаторы**

*Имена*, или *идентификаторы*, служат для того чтобы обращаться к программным объектам и различать их, то есть идентифицировать. В идентификаторе могут использоваться буквы, цифры и символ подчеркивания. Прописные и строчные буквы различаются, например, *hacker*, *Hacker* и *hAcKeR*  — три разных имени.

Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Длина идентификатора не ограничена. Пробелы внутри имен не допускаются.

В идентификаторах C# разрешается использовать, помимо латинских букв, буквы национальных алфавитов. Например, правильными являются идентификаторы Фёкла и calc. Более того, можно применять даже так называемые *escape-последовательности Unicode*, то есть представлять символ с помощью его кода в шестнадцатеричном виде с префиксом \u, например, \u00F2.

Имена даются элементам программы, к которым требуется обращаться: переменным, типам, константам, методам, меткам и т. д. Идентификатор создается на этапе объявления переменной (метода, типа и т. п.), после этого его можно использовать в последующих операторах программы. При выборе идентификатора необходимо следить, чтобы он не совпадал с ключевыми словами.

**Ключевые слова**

*Ключевые слова* — это зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Список ключевых слов C# приведен в [таблице 1](https://www.intuit.ru/studies/courses/629/485/lecture/11001?page=1#table.2.1). Как видите, их не так уж и много!

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Таблица 1. Ключевые слова C# | | | | | |
| bstract | as | base | bool | break |
| byte | case | catch | char | checked |
| class | const | continue | decimal | default |
| delegate | do | double | else | enum |
| event | explicit | extern | false | finally |
| fixed | float | for | foreach | goto |
| if | implicit | in | int | interface |
| internal | is | lock | long | namespace |
| new | null | object | operator | out |
| override | params | private | protected | public |
| readonly | ref | return | sbyte | sealed |
| short | sizeof | stackalloc | static | string |
| struct | switch | this | throw | true |
| try | typeof | uint | ulong | unchecked |
| *unsafe* | ushort | using | virtual | void |
| volatile | while |  |  |  |

**Знаки операций и разделители**

*Знак операции* — это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Например, в выражении a += b знак += является знаком операции, а a и b  — операндами. Символы, составляющие знак операций, могут быть специальными, например, +, &&, | и <, и буквенными, такими как as или new.

Операции делятся на *унарные*, *бинарные* и *тернарную* по количеству участвующих в них операндов (один, два и три операнда соответственно). Один и тот же знак может интерпретироваться по-разному в зависимости от контекста.

*Разделители* используются для разделения или, наоборот, группирования элементов. Примеры разделителей: скобки, точка, запятая. Ниже перечислены все знаки операций и разделители, использующиеся в C#:

{ } [ ] ( ) . , : ; + - \* / % & | ^ ! ~ =

< > ? ++ -- && || << >> == != <= >= += -= \*= /= %=

&= |= ^= <<= >>= ->

**Литералы (константы)**

*Литералами*, или *константами*, называют неизменяемые величины. В C# есть логические, целые, вещественные, символьные и строковые константы, а также константа null. Компилятор, выделив константу в качестве лексемы, относит ее к одному из типов данных по ее внешнему виду. Программист может задать тип константы и самостоятельно.

Описание и примеры констант каждого типа приведены в [таблице 2.](https://www.intuit.ru/studies/courses/629/485/lecture/11001?page=1#table.2.2)

|  |  |  |
| --- | --- | --- |
| Таблица 2.. Константы в C# | | |
| **Константа** | **Описание** | **Примеры** |
| Логическая | true (истина) или false (ложь) | true  false |
| Целая | *Десятичная*: последовательность десятичных цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), за которой может следовать суффикс (U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu) | 8 0 199226  8u 0Lu 199226L |
|  | *Шестнадцатеричная*: символы 0х или 0Х, за которыми следуют шестнадцатеричные цифры (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F), а за цифрами, в свою очередь, может следовать суффикс (U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu) | 0xA 0x1B8 0X00FF  0xAU 0x1B8LU  0X00FFl |
| Вещественная | *С фиксированной точкой*: [цифры][.][цифры][суффикс] Суффикс — один из символов F, f, D, d, M, m | 5.7 .001 35.  5.7F .001d 35.  5F .001f 35m |
|  | *С порядком*: [цифры][.][цифры]{E|e}[+|–][цифры] [суффикс] Суффикс — один из символов F, f, D, d, M, m | 0.2E6 .11e+3 5E-10  0.2E6D .11e–3  5E10 |
| Символьная | Символ, заключенный в апострофы | 'A' 'ю' '\*'  '\0' '\n'  '\xF' '\x74'  '\uA81B' |
| Строковая | Последовательность символов, заключенная в кавычки | "Здесь был Vasia"  "\tЗначение r = \xF5 \n"  "Здесь был \u0056\u0061"  "C:\\temp\\file1.txt"  @"C:\temp\file1.txt" |
| Константа null | Ссылка, которая не указывает ни на какой объект | null |

*Логических* литералов всего два: true и false. Они широко используются в качестве признаков наличия или отсутствия чего-либо.

*Целые литералы* могут быть представлены либо в десятичной, либо в шестнадцатеричной системе счисления, а *вещественные*  — только в десятичной системе, но в двух формах: с фиксированной точкой и с порядком. Вещественная константа с порядком представляется в виде *мантиссы* и *порядка*. Мантисса записывается слева от знака экспоненты ( E или e ), порядок — справа от знака. Значение константы определяется как произведение мантиссы и возведенного в указанную в порядке степень числа 10 (например, 1.3e2 = 1,3 x 102 = 130). Пробелы внутри константы не допускаются.

Если требуется сформировать *отрицательную* целую или вещественную константу, то перед ней ставится знак унарной операции изменения знака ( – ), например, –218.

*Символьная константа* представляет собой любой символ в кодировке Unicode. Символьные константы записываются в одной из четырех форм:

* "обычный" символ, имеющий графическое представление (кроме апострофа и символа перевода строки) — 'A', 'ю', '\*' ;
* управляющая последовательность — '\0', '\n' ;
* символ в виде шестнадцатеричного кода — '\xF', '\x74' ;
* символ в виде escape-последовательности Unicode — '\uA81B'.
* *Управляющей последовательностью*, или *простой escape-последовательностью*, называют определенный символ, предваряемый обратной косой чертой. Управляющая последовательность интерпретируется как одиночный символ и используется для представления:
* кодов, не имеющих графического изображения (например, \n  — переход в начало следующей строки);
* символов, имеющих специальное значение в строковых и символьных литералах, например, апострофа (  '  ).

Допустимые значения последовательностей приведены в [таблице 3](https://www.intuit.ru/studies/courses/629/485/lecture/11001?page=1#table.2.3).

|  |  |
| --- | --- |
| Таблица 3. Управляющие последовательности в С# | |
| **Вид** | **Наименование** |
| \a | Звуковой сигнал |
| \b | Возврат на шаг |
| \f | Перевод страницы (формата) |
| \n | Перевод строки |
| \r | Возврат каретки |
| \t | Горизонтальная табуляция |
| \v | Вертикальная табуляция |
| \\ | Обратная косая черта |
| \' | Апостроф |
| \" | Кавычка |
| \0 | *Нуль-символ* |

Символ, представленный в виде *шестнадцатеричного кода*, начинается с префикса \x, за которым следует код символа. Числовое значение должно находиться в диапазоне от 0 до 216 – 1, иначе возникает ошибка компиляции.

*Escape-последовательности Unicode* служат для представления символа в кодировке Unicode с помощью его кода в шестнадцатеричном виде с префиксом \u или \U, например, \u00F2, \U00010011. Коды в диапазоне от \U10000 до \U10FFFF представляются в виде двух последовательных символов; коды, превышающие \U10FFFF, не поддерживаются.

Управляющие последовательности обоих видов могут использоваться и в *строковых константах*, называемых иначе *строковыми литералами*. Например, если требуется вывести несколько строк, можно объединить их в один литерал, отделив одну строку от другой символами \n:

"Никто не доволен своей\nвнешностью, но каждый доволен\nсвоим умом"

Этот литерал при выводе будет выглядеть так:

Никто не доволен своей

внешностью, но каждый доволен

своим умом

В C# есть и второй вид литералов — *дословные*(verbatim strings). Эти литералы предваряются символом @, который отключает обработку управляющих последовательностей и позволяет получать строки в том виде, в котором они записаны. Чаще всего дословные литералы применяются при задании полного пути файла. Посмотрите, насколько лучше воспринимается второй вариант записи одного и того же пути:

"C:\\app\\bin\\debug\\a.exe"

@"C:\app\bin\debug\a.exe"

Строка может быть пустой (записывается парой смежных двойных кавычек "" ). Пустая символьная константа недопустима.

Константа null представляет собой значение, задаваемое по умолчанию для величин так называемых ссылочных типов, которые мы рассмотрим далее в этой лекции.

**Комментарии**

*Комментарии* предназначены для записи пояснений к программе и формирования документации. Компилятор комментарии игнорирует. Внутри комментария можно использовать любые символы. В C# есть два вида комментариев: однострочные и многострочные.

*Однострочный* комментарий начинается с двух символов прямой косой черты ( // ) и заканчивается символом перехода на новую строку, *многострочный* заключается между символами-скобками /\* и \*/ и может занимать часть строки, целую строку или несколько строк. Комментарии не вкладываются друг в друга.

Кроме того, в языке есть еще одна разновидность комментариев, которые начинаются с трех подряд идущих символов косой черты ( /// ). Они предназначены для формирования документации к программе в формате XML. Компилятор извлекает эти комментарии из программы, проверяет их соответствие правилам и записывает их в отдельный файл.

**Типы данных**

Данные, с которыми работает *программа*, хранятся в оперативной памяти. Естественно, что компилятору необходимо точно знать, сколько места они занимают, как именно закодированы и какие действия с ними можно выполнять. Все это задается при описании данных с помощью типа.

*Тип данных* однозначно определяет:

* *внутреннее представление* данных, а следовательно и *множество их возможных значений* ;
* *допустимые действия* над данными (операции и функции).

Например, целые и вещественные числа, даже если они занимают одинаковый объем памяти, имеют совершенно разные диапазоны возможных значений.

Каждое *выражение* в программе имеет определенный тип. *Компилятор* использует информацию о типе при проверке допустимости описанных в программе действий.

*Память*, в которой хранятся данные во *время выполнения* программы, делится на две области: *стек* (stack) и *динамическая* область, или хип (*heap*), чаще называемый кучей (поскольку этот термин мне не нравится, я его использовать не буду). *Стек* используется для хранения величин, *память* под которые выделяет *компилятор*, а в *динамической области* *память* резервируется и освобождается во *время выполнения* программы с помощью специальных команд. Основным местом для хранения данных в C# является хип.

**Классификация типов**

Типы можно классифицировать по разным признакам. Если принять за основу строение элемента, все типы можно разделить на *простые* (не имеют внутренней структуры) и *структурированные* (состоят из элементов других типов). По своему "создателю" типы можно разделить на *встроенные* (стандартные) и *определяемые программистом*. По способу хранения значений типы делятся на *значимые*, или типы-значения, и *ссылочные*. Рассмотрим в первую очередь встроенные типы C#.

**Встроенные типы**

Встроенные типы не требуют предварительного определения. Для каждого типа существует ключевое слово, которое используется при описании переменных, констант и т. д. Встроенные типы C# приведены в [таблице 4](https://www.intuit.ru/studies/courses/629/485/lecture/11001?page=2#table.2.4). Они однозначно соответствуют стандартным классам библиотеки .NET, определенным в пространстве имен System. Как видно из таблицы, существуют несколько вариантов представления целых и вещественных величин. Программист выбирает тип каждой величины, используемой в программе, с учетом необходимого ему диапазона и точности представления данных.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Таблица 4. Встроенные типы C# | | | | | |
| **Название** | **Ключевое слово** | **Тип .NET** | **Диапазон значений** | **Описание** | **Размер, битов** |
| Логический тип | bool | Boolean | true, false |  |  |
| Целые типы | sbyte | SByte | От –128 до 127 | Со знаком | 8 |
| byte | Byte | От 0 до 255 | Без знака | 8 |
| short | Int16 | От –32768 до 32767 | Со знаком | 16 |
| ushort | UInt16 | От 0 до 65535 | Без знака | 16 |
| int | Int32 | От –2 x 109 до 2 x 109 | Со знаком | 32 |
| uint | UInt32 | От 0 до 4 x 109 | Без знака | 32 |
| long | Int64 | От –9 x 1018 до 9 x 1018 | Со знаком | 64 |
| ulong | UInt64 | От 0 до 18 x 1018 | Без знака | 64 |
| Символьный тип | char | Char | От U+0000 до U+ffff | Unicode-символ | 16 |
| Вещественные | float | Single | От 1.5 x 10-45 до 3.4 x 1038 | 7 цифр | 32 |
| double | Double | От 5.0 x 10-324 до 1.7 x 10308 | 15–16 цифр | 64 |
| Финансовый тип | decimal | Decimal | От 1.0 x 10-28 до 7.9 x 1028 | 28–29 цифр | 128 |
| Строковый тип | string | String | Длина ограничена объемом доступной памяти | Строка из Unicode-символов |  |
| Тип object | object | Object | Можно хранить все, что угодно | Всеобщий предок |  |

*Логический*, или булев, тип содержит всего два значения: true (истина) и false (ложь). Их внутреннее представление программиста интересовать не должно. Все целые и вещественные типы вместе с символьным и финансовым можно назвать *арифметическими типами*.

Внутреннее представление величины *целого типа*  — целое число в двоичном коде. В знаковых типах старший бит числа интерпретируется как знаковый (0 — положительное число, 1 — отрицательное).

*Вещественные типы*, или *типы данных с плавающей точкой*, хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей — мантиссы и порядка, причем каждая часть имеет знак. Длина мантиссы определяет *точность* числа, а длина порядка — его *диапазон*. Все вещественные типы могут представлять как положительные, так и отрицательные числа. Чаще всего в программах используется тип double, поскольку его диапазон и точность покрывают большинство потребностей. Этот тип имеют вещественные литералы и многие стандартные математические функции.

Тип decimal предназначен для *денежных вычислений*, в которых критичны ошибки округления. Величины типа decimalпозволяют хранить 28–29 десятичных разрядов. Тип decimal не относится к вещественным типам, у них различное внутреннее представление. Величины *денежного типа* даже нельзя использовать в одном выражении с вещественными без явного преобразования типа. Использование величин финансового типа в одном выражении с целыми допускается.

Любой встроенный тип C# соответствует стандартному классу библиотеки .NET, определенному в пространстве имен System. Имя этого класса приведено в третьем столбце таблицы 2.4. Везде, где используется имя встроенного типа, его можно заменить именем класса библиотеки.

**Типы литералов**

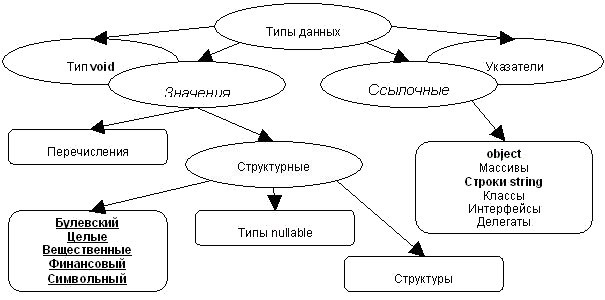
Литералы (константы) тоже имеют тип. Если значение целого литерала находится внутри диапазона допустимых значений типа int, литерал рассматривается как int, иначе он относится к наименьшему из типов uint, long или ulong, в диапазон значений которого он входит. Вещественные литералы по умолчанию относятся к типу double.

Например, константа 10 относится к типу int, хотя для ее хранения достаточно и байта, а константа 2147483648 будет определена как uint. Для явного задания типа литерала служит суффикс, например, 1.1f, 1UL, 1000m. Явное задание применяется в основном для уменьшения количества *неявных преобразований типа*, выполняемых компилятором.

**Типы-значения и ссылочные типы**

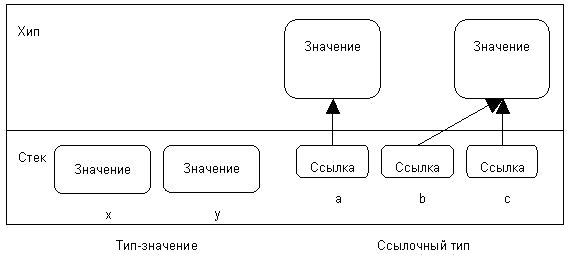
Чаще всего типы C# разделяют *по способу хранения элементов* на типы-значения и ссылочные типы ([рис. 1](https://www.intuit.ru/studies/courses/629/485/lecture/11001?page=2#image.2.1)). Элементы *типов-значений*, или *значимых типов* (value types), представляют собой просто последовательность битов в памяти, необходимый объем которой выделяет компилятор. Иными словами, величины значимых типов хранят свои значения непосредственно. Величина *ссылочного типа* хранит не сами данные, а ссылку на них (адрес, по которому расположены данные). Сами данные хранятся в хипе.

*Несмотря на различия в способе хранения, и типы-значения, и ссылочные типы являются потомками общего базового класса* object.



**Рис. 1.**Классификация типов данных C# по способу хранения

[Рис. 2](https://www.intuit.ru/studies/courses/629/485/lecture/11001?page=2#image.2.2) иллюстрирует разницу между величинами значимого и ссылочного типов. Одни и те же действия над ними выполняются по-разному. Рассмотрим в качестве примера проверку на равенство. Величины значимого типа равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одни и те же данные (на рисунке b и c равны, но a не равно b даже при одинаковых значениях). Из этого следует, что если изменить значение одной величины ссылочного типа, это может отразиться на другой.



**Рис. 2.**Хранение в памяти величин значимого и ссылочного типов

**Упаковка и распаковка**

Для того чтобы величины ссылочного и значимого типов могли использоваться совместно, необходимо иметь возможность преобразования из одного типа в другой. Преобразование из типа-значения в ссылочный тип называется *упаковкой* (boxing), обратное преобразование — *распаковкой* (unboxing).

Если величина значимого типа используется в том месте, где требуется ссылочный тип, автоматически выполняется создание промежуточной величины ссылочного типа. При необходимости обратного преобразования с величины ссылочного типа "снимается упаковка", и в дальнейших действиях участвует только ее значение.

## Организация ввода-вывода данных. Форматирование

*Программа* при вводе данных и выводе результатов взаимодействует с внешними устройствами. Совокупность стандартных устройств ввода (клавиатура) и вывода (экран) называется консолью. В языке С# нет операторов ввода и вывода. Вместо них для обмена данными с внешними устройствами используются специальные объекты. В частности, для работы с консолью используется стандартный *класс* Console, определенный в пространстве имен System.

### Вывод данных

В приведенных выше примерах рассматрен метод WriteLine, реализованный в классе Console, который позволяет организовывать вывод данных на экранСуществует несколько способов применения данного метода:

1. Console.WriteLine(x); //на экран выводится значение идентификатора х
2. Console.WriteLine("x=" + x +"y=" + y); /\* на экран выводится строка, образованная последовательным слиянием строки "x=", значения x, строки "у=" и значения у \*/
3. Console.WriteLine("x={0} y={1}", x, y); /\* на экран выводится строка, формат которой задан первым аргументом метода, при этом вместо параметра {0} выводится значение x, а вместо {1} – значение y\*/

Замечание.

Рассмотрим следующий фрагмент программы:

int i=3, j=4; Console.WriteLine("{0} {1}", i, j);

При обращении к методу WriteLine через запятую перечисляются три аргумента: "{0} {1}", i, j. Первый аргумент определяет формат выходной строки. Следующие аргументы нумеруются с нуля, так переменная i имеет номер 0, j – номер 1. Значение переменной i будет помещено в выходную строку на место {0}, а значение переменной j – на место {1}. В результате на экран будет выведена строка: 3 4. Если обратимся к методу WriteLine следующим образом Console.WriteLine("{0} {1} {2)", j, i, j), то на экран будет выведена строка: 4 3 4.

Последний вариант использования метода WriteLine является наиболее универсальным, потому что он позволяет не только выводить данные на экран, но и управлять форматом их вывода. Рассмотрим несколько примеров:

1. Использование управляющих последовательностей:

Управляющей последовательностью называют определенный символ, предваряемый обратной косой чертой. Данная совокупность символов интерпретируется как одиночный символ и используется для представления кодов символов, не имеющих графического обозначения (например, символа перевода курсора на новую строку) или символов, имеющих специальное обозначение в символьных и строковых константах (например, апостроф).

Управляющие символы:

|  |  |
| --- | --- |
| Вид | Наименование |
| \a | Звуковой сигнал |
| \b | Возврат на шаг назад |
| \f | Перевод страницы |
| \n | Перевод строки |
| \r | Возврат каретки |
| \t | Горизонтальная табуляция |
| \v | Вертикальная табуляция |
| \\ | Обратная косая черта |
| \' | Апостроф |
| \" | Кавычки |

Пример:

static void Main()

{

Console.WriteLine("Пример 1.\r\ Управляющие символы ");

}

1. Управление размером поля вывода:

Первым аргументом WriteLine указывается строка вида {n, m} – где n определяет номер идентификатора из списка аргументов метода WriteLine, а m – количество позиций (размер поля вывода), отводимых под значение данного идентификатора. При этом значение идентификатора выравнивается по правому краю. Если выделенных позиций для размещения значения идентификатора окажется недостаточно, то автоматически добавится необходимое количество позиций. Пример:

|  |  |
| --- | --- |
| static void Main()  {  double x= Math.E;  Console.WriteLine("E={0,20}", x);  Console.WriteLine("E={0,10}", x);  } | 02_02 |

1. Управление размещением вещественных данных:

Первым аргументом WriteLine указывается строка вида {n: ##.###} – где n определяет номер идентификатора из списка аргументов метода WriteLine, а ##.### определяет *формат вывода* вещественного числа. В данном случае под целую часть числа отводится две позиции, под дробную – три. Если выделенных позиций для размещения целой части значения идентификатора окажется недостаточно, то автоматически добавиться необходимое количество позиций. Пример:

|  |  |
| --- | --- |
| static void Main()  {  double x= Math.E;  Console.WriteLine("E={0:##.###}", x);  Console.WriteLine("E={0:.####}", x);  } | 02_03 |

1. Управление форматом числовых данных:

Первым аргументом WriteLine указывается строка вида

{n: <спецификатор>m} – где n определяет номер идентификатора из списка аргументов метода WriteLine, <спецификатор> - определяет формат данных, а m – количество позиций для дробной части значения идентификатора. В качестве спецификаторов могут использоваться следующие значения:

|  |  |  |
| --- | --- | --- |
| Параметр | Формат | Значение |
| C или c | Денежный. По умолчанию ставит знак р. Изменить его можно с помощь объекта NumberFormatInfo | Задается количество десятичных разрядов. |
| D или d | Целочисленный (используется только с целыми числами) | Задается минимальное количество цифр. При необходимости результат дополняется начальными нулями |
| E или e | Экспоненциальное представление чисел | Задается количество символов после запятой. По умолчанию используется 6 |
| F или f | Представление чисел с фиксированной точкой | Задается количество символов после запятой |
| G или g | Общий формат (или экспоненциальный, или с фиксированной точкой) | Задается количество символов после запятой. По умолчанию выводится целая часть |
| N или n | Стандартное форматирование с использованием запятых и пробелов в качестве разделителей между разрядами | Задается количество символов после запятой. По умолчанию – 2, если число целое, то ставятся нули |
| X или x | Шестнадцатеричный формат |  |
| P или p | Процентный |  |

Пример:

static void Main()

{

Console.WriteLine("C Format:{0,14:C} \t{0:C2}", 12345.678);

Console.WriteLine("D Format:{0,14:D} \t{0:D6}", 123);

Console.WriteLine("E Format:{0,14:E} \t{0:E8}", 12345.6789);

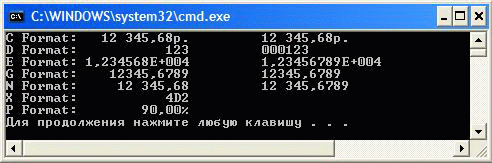
Console.WriteLine("G Format:{0,14:G} \t{0:G10}", 12345.6789);

Console.WriteLine("N Format:{0,14:N} \t{0:N4}", 12345.6789);

Console.WriteLine("X Format:{0,14:X} ", 1234);

Console.WriteLine("P Format:{0,14:P} ", 0.9);

}



### Ввод данных

Для ввода данных обычно используется метод ReadLine, реализованный в классе Console. Особенностью данного метода является то, что в качестве результата он возвращает строку ( string ). Пример:

static void Main()

{

string s = Console.ReadLine();

Console.WriteLine(s);

}

Для того чтобы получить числовое значение необходимо воспользоваться преобразованием данных. Пример:

static void Main()

{

string s = Console.ReadLine();

int x = int.Parse(s); //преобразование строки в число

Console.WriteLine(x);

}

Или сокращенный вариант:

static void Main()

{

//преобразование введенной строки в число

int x = int.Parse(Console.ReadLine());

Console.WriteLine(x);

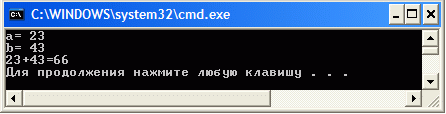
}

Для преобразования строкового представления целого числа в тип int мы используем метод int.Parse(), который реализован для всех числовых типов данных. Таким образом, если нам потребуется преобразовать строковое представление в вещественное, мы можем воспользоваться методом float.Parse() или double.Parse(). В случае, если соответствующее преобразование выполнить невозможно, то выполнение программы прерывается и генерируется исключение System.FormatExeption (*входная строка* имела неверный формат).

## Задания на лабораторную работу.

Написать программу, которая, реализует диалог с пользователем:

1. запрашивает с клавиатуры два целых числа, и выводит на экран сумму данных чисел:



Пример:

using System;

namespace Hello

{

class Program

{

static void Main()

{

Console.Write("a= ");

int a = int.Parse(Console.ReadLine());

Console.Write("b= ");

int b = int.Parse(Console.ReadLine());

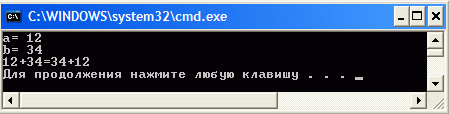
Console.WriteLine("{0}+{1}={2}", a, b, a + b);

}

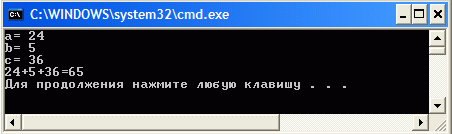
}

}

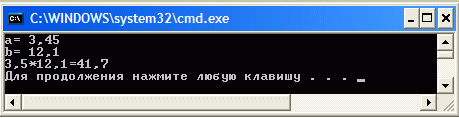
1. запрашивает с клавиатуры два целых числа, и выводит на экран сумму данных чисел в прямом и обратном порядке:



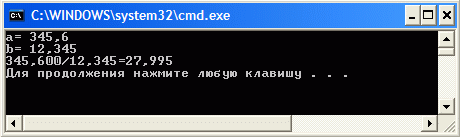
1. запрашивает с клавиатуры три целых числа, и выводит на экран сумму данных чисел:



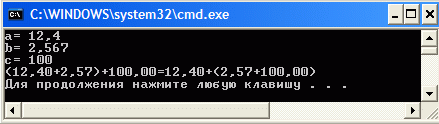
1. запрашивает с клавиатуры два вещественных числа, и выводит на экран произведение данных чисел (вещественные числа выводятся с точностью до 1 знака после запятой):



1. запрашивает с клавиатуры два вещественных числа, и выводит на экран результат деления первого числа на второе (вещественные числа выводятся с точностью до 3 знаков после запятой):



1. запрашивает с клавиатуры три вещественных числа, и выводит на экран следующее сообщение (вещественные числа выводятся с точностью до 2 знаков после запятой):



## Контрольные вопросы.

1. Перечислите и опишите лексемы языка C#.
2. Перечислите и опишите литералы языка C#.
3. Перечислите встроенные типы данных и опишите их.
4. Приведите классификацию типов данных C#.
5. Какие элементы стандартных классов .NET, соответствующих встроенным типам языка, вы знаете?
6. Опишите отличия значимых и ссылочных типов. В чем состоят процессы упаковки и распаковки?

# Операции в программах на языке C#

## Теоретические сведения.

Операции можно классифицировать по количеству операндов на: унарные - воздействуют на один операнд, бинарные - воздействуют на два операнда, тернарные - воздействует на три операнда. Некоторые символы используются для обозначения как унарных, так и бинарных операций. Например, символ "минус" используется как для обозначения унарной операции - арифметического отрицания, так и для обозначения бинарной операции вычитание. Будет ли данный символ обозначать унарную или бинарную операцию, определяется контекстом, в котором он используется.

### Инкремент (++) и декремент(--).

Эти операции имеют две формы записи - *префиксную*,когда операция записывается перед операндом, и *постфиксную* - операция записывается после операнда. *Префиксная операция* инкремента (декремента) увеличивает (уменьшает) свой операнд и возвращает измененное значение как результат. Постфиксные версии инкремента и декремента возвращают первоначальное значение операнда, а затем изменяют его.

Рассмотрим эти операции на примере.

|  |  |
| --- | --- |
| static void Main()  {  int i = 3, j = 4;  Console.WriteLine("{0} {1}", i, j);  Console.WriteLine("{0} {1}", ++i, --j);  Console.WriteLine("{0} {1}", i++, j--);  Console.WriteLine("{0} {1}", i, j);  } | Результат работы программы:  3 4  4 3  4 3  5 2 |

Допустимы ли следующие способы записи ++(++i), (i--)--, ++(i--) и т.д. И почему.

Префиксная версия требует существенно меньше действий: она изменяет значение переменной и запоминает результат в ту же переменную. Постфиксная операция должна отдельно сохранить исходное значение, чтобы затем вернуть его как результат. Для сложных типов подобные дополнительные действия могут оказаться трудоемки. Поэтому постфиксную форму имеет смысл использовать только при необходимости.

### **Операция new**

Используется для создания нового объекта. С помощью ее можно создавать как объекты ссылочного типа, так и размерные, например:

* object z=new object();
* int i=new int(); // то же самое, что и int i =0;

### Отрицание:

* + Арифметическое отрицание (-) - меняет знак операнда на противоположный.
  + Логическое отрицание (!) - определяет операцию инверсия для логического типа.

Рассмотрим эти операции на примере.

|  |  |
| --- | --- |
| static void Main()  {  int i = 3, j=-4;  bool a = true, b=false;  Console.WriteLine("{0} {1}", -i, -j);  Console.WriteLine("{0} {1}", !a, !b);  } | Результат работы программы:  -3 4  False True |

### **Явное преобразование типа.**

Используется для явного преобразования из одного типа в другой. Формат операции:

(тип) выражение;

Рассмотрим эту операцию на примере.

|  |  |
| --- | --- |
| static void Main()  {  int i = -4;  byte j = 4;  int a = (int)j; //преобразование без потери точности  byte b = (byte)i; //преобразование с потерей точности  Console.WriteLine("{0} {1}", a, b);  } | Результат работы программы:  4 252 |

### Умножение (\*), деление (/) и деление с остатком (%).

Операции умножения и деления применимы для целочисленных и вещественных типов данных. Для других типов эти операции применимы, если для них возможно *неявное преобразование* к целым или вещественным типам. При этом тип результата равен "наибольшему" из типов операндов, но не менее int. Если оба операнда при делении целочисленные, то и результат тоже целочисленный.

Рассмотрим эти операции на примере.

static void Main()

{

int i = 100, j = 15;

double a = 14.2, b = 3.5;

Console.WriteLine("{0} {1} {2}", i\*j, i/j, i%j);

Console.WriteLine("{0} {1} {2}", a \* b, a / b, a % b);

}

Результат работы программы:

1500 6 10

49.7 4.05714285714286 0.1999999999999999

*Вопрос*. Чему будет равен результат выполнения программы:

double hmm = 1.0 / 0.0;

double hmm2 = -1.0 / 0.0;

double hmm3 = 0.0 / 0.0;

Console.WriteLine("1/0 == {0}", hmm);

Console.WriteLine("-1/0 == {0}", hmm2);

Console.WriteLine("0/0 == {0}", hmm3);

double dd = 1.0 / 0;

dd = 1.0 / 0.0;

if (double.IsNaN(dd)) dd = 9;

if (double.IsInfinity(dd)) dd = 8;

Console.WriteLine("dd == {0}", dd);

Объясните, как получился данный результат.

### Сложение (+) и вычитание (-).

Операции сложения и вычитания применимы для целочисленных и вещественных типов данных. Для других типов эти операции применимы, если для них возможно *неявное преобразование* к целым или вещественным типам.

### Операции отношения ( <, <=, >, >=, ==, !=).

Операции отношения сравнивают значения левого и правого операндов. Результат операции логического типа: true - если значения совпадают, false - в противном случае. Рассмотрим операции на примере:

|  |  |
| --- | --- |
| static void Main()  {  int i = 15, j = 15;  Console.WriteLine(i<j); //меньше  Console.WriteLine(i<=j); //меньше или равно  Console.WriteLine(i>j); //больше  Console.WriteLine(i>=j); //больше или равно  Console.WriteLine(i==j); //равно  Console.WriteLine(i!=j); //не равно  } | Результат работы программы:  False  True  False  True  True  False |

### Логические операции: И ( && ), ИЛИ ( || ).

Логические операции применяются к операндам логического типа.

Результат логической операции И имеет значение истина тогда и только тогда, когда оба операнда принимают значение истина.

Результат логической операции ИЛИ имеет значение истина тогда и только тогда, когда хотя бы один из операндов принимает значение истина.

Рассмотрим операции на примере:

static void Main()

{

Console.WriteLine("x y x и y x или y");

Console.WriteLine("{0} {1} {2} {3}", false, false, false&&false, false||false);

Console.WriteLine("{0} {1} {2} {3}", false, true, false&&true, false||true);

Console.WriteLine("{0} {1} {2} {3}", true, false, true&&false, true||false);

Console.WriteLine("{0} {1} {2} {3}", true, true, true&&true, true||true);

}

Результат работы программы:

x y x и y x или y

False False False False

False True False True

True False False True

True True True True

Фактически была построена таблица истинности для логических операций И и ИЛИ.

### Условная операция.

Формат: (<операнд1>)? <операнд2> : <операнд3> ;

Операнд1 - это логическое выражение, которое оценивается с точки зрения его эквивалентности константам true и false. Если результат вычисления операнда1 равен true, то результатом условной операции будет значение операнда2, иначе - операнда3. Фактически условная операция является сокращенной формой условного оператора if, который будет рассмотрен позже.

Пример использования условной операции:

static void Main()

{

int x=5; int y=10;

int max = (x > y) ? x : y;

Console.WriteLine(max);

}

### Операции присваивания: =, +=, -= и т.д.

Формат операции *простого присваивания* ( = ):

операнд\_2 = операнд\_1;

В результате выполнения этой операции вычисляется значение операнда\_1, и результат записывается в операнд\_2. Возможно связать воедино сразу несколько операторов присваивания, записывая такие цепочки: a=b=c=100. Выражение такого вида выполняется справа налево: результатом выполнения c=100 является число 100, которое затем присваивается переменной b, результатом чего опять является 100, которое присваивается переменной a.

Кроме простой операции присваивания существуют *сложные операции присваивания*, например, умножение с присваиванием ( \*= ), деление с присваиванием ( /= ), остаток от деления с присваиванием ( % =), сложение с присваиванием ( += ), вычитание с присваиванием ( -= ) и т.д.

В сложных операциях присваивания, например, при *сложении с присваиванием*, к операнду\_2 прибавляется операнд\_1, и результат записывается в операнд\_2. То есть, выражение с += а является более компактной записью выражения с = с + а. Кроме того, сложные операции присваивания позволяют сгенерировать более эффективный код, за счет того, что в простой операции присваивания для хранения значения правого операнда создается временная переменная, а в сложных операциях присваивания значение правого операнда сразу записывается в левый операнд.

Объясните, какие значения примут переменные t и b после выполнения данного фрагмента программы: int a=10, b=3; int t=(a++)-b; b+=t\*a;

Рассмотренные *операции* приведены с учетом убывания приоритета. Если в одном выражении соседствуют *операции* одного приоритета, то *операции* присваивания и условная *операции* выполняются справа налево, а остальные наоборот. Если необходимо изменить порядок выполнения операций, то в выражении необходимо поставить круглые скобки.

### Выражения и преобразование типов

*Выражение* - это синтаксическая *единица* языка, определяющая способ вычисления некоторого значения. Выражения состоят из операндов, операций и скобок. Каждый *операнд* является в свою *очередь* выражением или одним из его частных случаев - константой, переменной или функций.

Примеры выражений:

(а + 0.12)/6

х && у || !z

(t \* Math.Sin(x)-l.05e4)/((2 \* k + 2) \* (2 \* k + 3))

*Операции* выполняются в соответствии с приоритетами. Для изменения порядка выполнения операций используются круглые скобки. Если в одном выражении записано несколько операций одинакового приоритета, то унарные *операции*, условная операция и *операции* присваивания выполняются *справа налево*, остальные - *слева направо*. Например,

а = b = с означает a=(b=c),

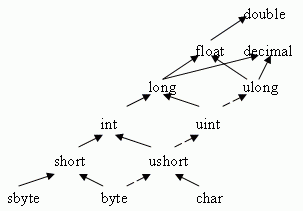
a+b+c означает (а + b) + с.

Результат вычисления выражения характеризуется значением и типом. Например, если а и b - переменные целого типа и описаны так:

int а = 2, b = 5;

то *выражение* а + b имеет *значение* 7 и тип int.

В *выражение* могут входить операнды различных типов. Если операнды имеют одинаковый тип, то результат *операции* будет иметь тот же тип. Если операнды разного типа, то перед вычислениями выполняются преобразования более коротких типов в более длинные для сохранения значимости и точности. *Иерархия типов* данных приведена в следующей схеме:



*Преобразование типов* в выражениях происходит *неявно* (без участия программистов) следующим образом: Если один из операндов имеет тип, изображенный на более низком уровне, чем другой, то он приводится к типу второго операнда при наличии пути между ними. Если пути нет, то возникает ошибка компиляции (чтобы ее избежать, необходимо воспользоваться операцией явного преобразования). Если путей преобразования несколько, то выбирается наиболее короткий, не содержащий пунктирных линий.

## Задания на лабораторную работу.

1. Написать программу, которая подсчитывает:
   1. периметр квадрата, площадь которого равна а ;

Пример:

using System;

namespace Example

{

class Program

{

static void Main()

{

Console.Write("s= ");

float s = float.Parse(Console.ReadLine());

double p = 4 \* Math.Sqrt(s);

Console.WriteLine("p=" + p);

}

}

}

* 1. площадь равностороннего треугольника, периметр которого равен p ;
  2. расстояние между точками с координатами a, b и с, d ;
  3. среднее арифметическое кубов двух данных чисел;
  4. среднее геометрическое модулей двух данных чисел;
  5. гипотенузу прямоугольного треугольника по двум данным катетам a, b.
  6. площадь прямоугольного треугольника по двум катетам a, b.
  7. периметр прямоугольного треугольника по двум катетам a, b.
  8. ребро куба, площадь полной поверхности которого равна s ;
  9. ребро куба, объем которого равен v ;
  10. периметр треугольника, заданного координатами вершин x1, y1, x2, y2, x3, y3 ;
  11. площадь треугольника, заданного координатами вершин x1, y1, x2, y2, x3, y3 ;
  12. радиус окружности, длина которой равна l ;
  13. радиус окружности, площадь круга которой равна s ;
  14. площадь равнобедренной трапеции с основаниями a и b и углом \alpha при большем основании;
  15. площадь кольца с внутренним радиусом r1 и внешним r2 ;
  16. радиус окружности, вписанной в равносторонний треугольник со стороной а ;
  17. радиус окружности, описанной около равностороннего треугольника со стороной а ;
  18. сумму членов арифметической прогрессии, если известен ее первый член, разность и число членов прогрессии;
  19. сумму членов геометрической прогрессии, если известен ее первый член, знаменатель и число членов прогрессии.

1. Написать программу, которая определяет:
   1. максимальное значение для двух различных вещественных чисел;

Пример:

using System;

namespace Hello

{

class Program

{

static void Main()

{

Console.Write("a= "); float a = float.Parse(Console.ReadLine());

Console.Write("b= "); float b = float.Parse(Console.ReadLine());

float max=(a>b)?a:b;

Console.WriteLine("max=" + max);

}

}

}

* 1. является ли заданное целое число четным;
  2. является ли заданное целое число нечетным;
  3. если целое число М делится на целое число N, то на экран выводится частное от деления, в противном случае выводится сообщение " M на N нацело не делится";
  4. оканчивается ли данное целое число цифрой 7 ;
  5. имеет ли уравнение ax2+bx+c=0 решение, где a, b, c – данные вещественные числа;
  6. какая из цифр двухзначного числа больше: первая или вторая;
  7. одинаковы ли цифры данного двухзначного числа;
  8. является ли сумма цифр двухзначного числа четной;
  9. является ли сумма цифр двухзначного числа нечетной;
  10. кратна ли трем сумма цифр двухзначного числа;
  11. кратна ли числу А сумма цифр двухзначного числа;
  12. какая из цифр трехзначного числа больше: первая или последняя;
  13. какая из цифр трехзначного числа больше: первая или вторая;
  14. какая из цифр трехзначного числа больше: вторая или последняя;
  15. все ли цифры трехзначного числа одинаковые;
  16. существует ли треугольник с длинами сторон a, b, c ;
  17. является ли треугольник с длинами сторон a, b, c прямоугольным;
  18. является ли треугольник с длинами сторон a, b, c равнобедренным;
  19. является ли треугольник с длинами сторон a, b, c равносторонним.

## Контрольные вопросы.

1. Допустимы ли следующие способы записи ++(++i), (i--)--, ++(i--) .
2. Приведите примеры операций: сдвиг влево ( << ), сдвиг вправо ( >> );
3. Приведите примеры операций: поразрядные операции И ( & ), исключающее ИЛИ ( ^ ) и ИЛИ ( | );
4. Приведите примеры операций: сложные операции присваивания: <<=, >>=, &=, ^=, |=.
5. Укажите последовательность выполнения операций в данном выражении:

(x\*x+Math.Sin(x+1))/x-2.

1. Запишите заданное математическое выражение по правилам языка С#:
2. Допустимы ли следующие способы записи !(-i), -(!a). И почему.
3. Объясните, какие значения примут переменные t и b после выполнения данного фрагмента программы: int a=10, b=3; int t=(a++)-b; b+=t\*a;
4. Измените программу так, чтобы:
   1. вычислялось наименьшее значение из двух вещественных чисел x и y ;
   2. если число двузначное, то на экран выводилось "Да", и "Нет" в противном случае.
5. Объясните, какое значение примет переменная t в данном фрагменте программы: int a=10, b=3; bool t=(a>=b && a!=2\*b || a<0);
6. Объясните, почему операция (byte)i вместо *ожидаемого значения* -4 дала нам в качестве результата значение 252.
7. Выясните, чему равен результат данного выражения:
   1. i<j<k
   2. true<false

Объясните, как получился данный результат.

# Операторы языка C#

## Теоретические сведения.

*Программа* на языке С# состоит из последовательности операторов, каждый из которых определяет законченное описание некоторого действия и заканчивается точкой с запятой. Все *операторы* можно разделить на 4 группы: *операторы* следования, *операторы*ветвления, *операторы цикла* и *операторы передачи управления*.

### Операторы следования

Операторы следования выполняются компилятором в естественном порядке: начиная с первого до последнего. К операторам следования относятся: выражение и составной оператор.

Любое *выражение*, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении значения выражения или выполнении законченного действия, например, вызова метода. Например:

++i; //оператор инкремента

x+=y; //оператор сложение с присваиванием

Console.WriteLine(x); //вызов метода

x=Math.Pow(a,b)+a\*b; //вычисление сложного выражения

Частным случаем оператора выражения является пустой оператор -;. Он используется тогда, когда по синтаксису оператор требуется, а по смыслу - нет. В этом случае лишний символ ; является пустым оператором и вполне допустим, хотя и не всегда безопасен. Например, случайный символ ; после условия оператора while или if может совершенно поменять работу этого оператора.

*Составной оператор* или *блок* представляет собой последовательность операторов, заключенных в фигурные скобки {}. Блок обладает собственной *областью видимости*: объявленные внутри блока имена доступны только внутри данного блока или блоков, вложенных в него. Составные операторы применяются в случае, когда правила языка предусматривают наличие только одного оператора, а логика программы требует нескольких операторов. Например, тело цикла while должно состоять только из одного оператора. Если заключить несколько операторов в фигурные скобки, то получится блок, который будет рассматриваться компилятором как единый оператор.

### Операторы ветвления

Операторы ветвления позволяют изменить порядок выполнения операторов в программе. К операторам ветвления относятся условный оператор if и оператор выбора switch.

### Условный оператор if

Условный оператор *if* используется для разветвления процесса обработки данных на два направления. Он может иметь одну из форм: *сокращенную* или *полную*.

#### Форма сокращенного оператора if:

if (B) S;

где В - логическое или арифметическое выражение, истинность которого проверяется; S - оператор: простой или составной.

При выполнении сокращенной формы оператора if сначала вычисляется выражение B, затем проводится анализ его результата: если B истинно, то выполняется оператор S ; если B ложно, то оператор S пропускается. Таким образом, с помощью сокращенной формы оператора if можно либо выполнить оператор S, либо пропустить его.

#### Форма полного оператора if:

if (B) S1; else S2;

где B - логическое или арифметическое выражение, истинность которого проверяется; S1, S2 - оператор: простой или составной.

При выполнении полной формы оператора if сначала вычисляется выражение B, затем анализируется его результат: если Bистинно, то выполняется оператор S1, а оператор S2 пропускается; если B ложно, то выполняется оператор S2, а S1 - пропускается. Таким образом, с помощью полной формы оператора if можно выбрать одно из двух альтернативных действий процесса обработки данных.

#### Примеры записи условного оператора if:

if (a > 0) x=y; // Сокращенная форма c простым оператором

if (++i) {x=y; y=2\*z;} // Сокращенная форма c составным оператором

if (a > 0 || b<0) x=y; else x=z; // Полная форма с простым оператором

if (i+j-1) { x= 0; y= 1;}

else {x=1; y:=0;} // Полная форма с составными операторами

Пример использования условного оператора.

static void Main()

{

Console.Write("x= ");

float x = float.Parse(Console.ReadLine());

Console.Write("y=");

float y = float.Parse(Console.ReadLine());

if (x < y ) Console.WriteLine("min= "+x);

else Console.WriteLine("min= "+y);

}

Результат работы программы:

x y min

0 0 0

1 -1 -1

-2 2 -2

Операторы S1 и S2 могут также являться операторами if. Такие операторы называют вложенными. При этом ключевое слово else связывается с ближайшим предыдущим словом if, которое еще не связано ни с одним else.

### Оператор выбора switch

Оператор выбора switch предназначен для разветвления процесса вычислений по нескольким направлениям. Формат оператора:

switch ( <выражение> )

{

case <константное\_выражение\_1>:

[<оператор 1>]; <оператор перехода>;

case <константное\_выражение\_2>:

[<оператор 2>]; <оператор перехода>;

...

case <константное\_выражение\_n>:

[<оператор n>]; <оператор перехода>;

[default: <оператор>; ]

}

Выражение, записанное в квадратных скобках, является необязательным элементом в *операторе switch*. Если оно отсутствует, то может отсутствовать и оператор перехода.

Выражение, стоящее за ключевым словом switch, должно иметь арифметический, символьный, строковый тип или тип указатель. Все константные выражения должны иметь разные значения, но их тип должен совпадать с типом выражения, стоящим после switchили приводиться к нему. Ключевое слово case и расположенное после него константное выражение называют также меткой case.

Выполнение оператора начинается с вычисления выражения, расположенного за ключевым словом switch. Полученный результат сравнивается с меткой case. Если результат выражения соответствует метке case, то выполняется оператор, стоящий после этой метки, за которым обязательно должен следовать оператор перехода: break, goto и т.д. При использовании оператора break происходит выход из switch и управление передается оператору, следующему за switch. Если же используется оператор goto, то управление передается оператору, помеченному меткой, стоящей после goto.

**Пример**. По заданному виду арифметической операции (сложение, вычитание, умножение и деление) и двум операндам, вывести на экран результат применения данной операции к операндам.

static void Main()

{

Console.Write("OPER= ");

char oper=char.Parse(Console.ReadLine());

bool ok=true;

Console.Write("A= ");

int a=int.Parse(Console.ReadLine());

Console.Write("B= ");

int b=int.Parse(Console.ReadLine());

float res=0;

switch (oper)

{

case '+': res = a + b; break; //1

case '-': res = a - b; break;

case '\*': res = a \* b; break;

case ':': if (b != 0)

{

res = (float)a / b; break;

}

else goto default;

default: ok = false; break;

}

if (ok) Console.WriteLine("{0} {1} {2} = {3}", a, oper, b, res);

else Console.WriteLine("error");

}

Результат выполнения программы:

oper x y rez

+ 4 5 9

: 4 0 error

% 4 3 error

Если необходимо, чтобы для разных меток выполнялось одно и тоже действие, то метки перечисляются через двоеточие. Например:

switch (oper)

{

case '+': res = a + b; break;

case '-': res = a - b; break;

case '\*': res = a \* b; break;

case '/': case ':': if (b != 0) // перечисление меток

{

res = (float)a / b; break;

}

else goto default;

default: ok = false; break;

}

### Операторы цикла

Операторы цикла используются для организации многократно повторяющихся вычислений. К операторам цикла относятся: *цикл с предусловием* while, *цикл с постусловием* do while, цикл с параметром for и цикл перебора foreach..

#### Цикл с предусловием while

Оператор цикла while организует выполнение одного оператора (простого или составного) неизвестное заранее число раз. Формат цикла while:

while (B) S;

где B - выражение, истинность которого проверяется (условие завершения цикла); S - тело цикла - оператор (простой или составной).

Перед каждым выполнением тела цикла анализируется значение выражения В: если оно истинно, то выполняется тело цикла, и управление передается на повторную проверку условия В ; если значение В ложно - цикл завершается и управление передается на оператор, следующий за оператором S.

Если результат выражения B окажется ложным при первой проверке, то тело цикла не выполнится ни разу. Отметим, что если условие B во время работы цикла не будет изменяться, то возможна ситуация зацикливания, то есть невозможность выхода из цикла. Поэтому внутри тела должны находиться операторы, приводящие к изменению значения выражения B так, чтобы цикл мог корректно завершиться.

В качестве иллюстрации выполнения цикла while рассмотрим программу вывода на экран целых чисел из интервала от 1 до n.

static void Main()

{

Console.Write("N= ");

int n=int.Parse(Console.ReadLine());

int i = 1;

while (i <= n) //пока i меньше или равно n

Console.Write(" "+ i++ ); //выводим i на экран, затем увеличиваем его на 1

}

Результаты работы программы:

n ответ

10 1 2 3 4 5 6 7 8 9 10

#### Цикл с постусловием do while

Оператор цикла do while также организует выполнение одного оператора (простого или составного) неизвестное заранее число раз. Однако в отличие от цикла while условие завершения цикла проверяется после выполнения тела цикла. Формат цикла dowhile:

do S while (B);

где В - выражение, истинность которого проверяется (условие завершения цикла); S - тело цикла - оператор (простой или блок).

Сначала выполняется оператор S, а затем анализируется значение выражения В: если оно истинно, то управление передается оператору S, если ложно - цикл завершается, и управление передается на оператор, следующий за условием B. Так как условие Впроверяется после выполнения тела цикла, то в любом случае тело цикла выполнится хотя бы один раз.

В операторе do while, так же как и в операторе while, возможна ситуация зацикливания в случае, если условие В всегда будет оставаться истинным.

В качестве иллюстрации выполнения цикла do while рассмотрим программу вывода на экран целых чисел из интервала от 1 до n.

static void Main()

{

Console.Write("N= ");

int n=int.Parse(Console.ReadLine());

int i = 1;

do

Console.Write(" " + i++);

//выводим i на экран, затем увеличиваем его на 1

while (i <= n); //пока i меньше или равно n

}

#### Цикл с параметром for

Цикл с параметром имеет следующую структуру:

for ( <инициализация>; <выражение>; <модификация>) <оператор>;

*Инициализация* используется для объявления и/или присвоения начальных значений величинам, используемым в цикле в качестве параметров (счетчиков). В этой части можно записать несколько операторов, разделенных запятой. Областью действия переменных, объявленных в части инициализации цикла, является цикл и *вложенные блоки*. Инициализация выполняется один раз в начале исполнения цикла.

*Выражение* определяет условие выполнения цикла: если его результат истинен, цикл выполняется. Истинность выражения проверяется перед каждым выполнением тела цикла, таким образом, цикл с параметром реализован как *цикл с предусловием*. В блоке выражение через запятую можно записать несколько логических выражений, тогда запятая равносильна операции логическое И ( && ).

*Модификация* выполняется после каждой итерации цикла и служит обычно для изменения параметров цикла. В части модификация можно записать несколько операторов через запятую.

*Оператор*(простой или составной) представляет собой тело цикла.

Любая из частей оператора *for* (инициализация, выражение, модификация, оператор) может отсутствовать, но точку с запятой, определяющую позицию пропускаемой части, надо оставить.

static void Main()

{

Console.Write("N= ");

int n=int.Parse(Console.ReadLine());

for (int i=1; i<=n;) //блок модификации пустой

Console.Write(" " + i++);

}

#### Вложенные циклы

Циклы могут быть простые или вложенные (кратные, циклы в цикле). Вложенными могут быть циклы любых типов: while, do while, for. Каждый внутренний цикл должен быть полностью вложен во все внешние циклы. "Пересечения" циклов не допускаются.

Пример использования вложенных циклов, который позволит вывести на экран числа следующим образом:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |

static void Main()

{

for (int i = 1; i <= 4; ++i, Console.WriteLine()) //1

for (int j=1; j<=5; ++j)

Console.Write(" " + 2);

}

В строке 1 в блоке модификации содержится два оператора ++i и Console.WriteLine(). В данном случае после каждого увеличения параметра i на 1 курсор будет переводиться на новую строку.

### Операторы безусловного перехода

В С# есть несколько операторов, изменяющих естественный порядок выполнения команд: *оператор безусловного перехода* goto, оператор выхода break, оператор перехода к следующей итерации цикла continue, оператор возврата из метода return и оператор генерации исключения throw.

#### Оператор безусловного перехода goto

*Оператор безусловного перехода* goto имеет формат:

goto <метка>;

В теле той же функции должна присутствовать ровно одна конструкция вида:

<метка>: <оператор>;

Оператор *goto* передает управление на помеченный меткой оператор. Рассмотрим пример использования оператора goto:

static void Main()

{

float x;

metka: Console.WriteLine("x="); //оператор, помеченный меткой

x = float.Parse(Console.ReadLine());

if (x!=0) Console.WriteLine("y({0})={1}", x, 1 / x );

else

{

Console.WriteLine("функция не определена");

goto metka; // передача управление метке

}

}

Следует учитывать, что использование оператора goto затрудняет чтение больших по объему программ, поэтому использовать метки нужно только в крайних случаях, например, в операторе switch.

#### Оператор выхода break

Оператор break используется внутри операторов ветвления и цикла для обеспечения перехода в точку программы, находящуюся непосредственно за оператором, внутри которого находится break.

Ранее применяли оператор break для выхода из оператора switch, аналогичным образом он может применяться для выхода из других операторов.

#### Оператор перехода к следующей итерации цикла continue

Оператор перехода к следующей итерации цикла continue пропускает все операторы, оставшиеся до конца тела цикла, и передает управление на начало следующей итерации (повторение тела цикла). Рассмотрим оператор continue на примере.

static void Main()

{

Console.WriteLine("n=");

int n = int.Parse(Console.ReadLine());

for (int i = 1; i <= n; i++)

{

if (i % 2 == 0) continue;

Console.Write(" " + i);

}

}

## Задания на лабораторную работу.

1. Дана точка на плоскости с координатами (х, у). Составить программу, которая выдает одно из сообщений "Да", "Нет", "На границе" в зависимости от того, лежит ли точка внутри заштрихованной области, вне заштрихованной области или на ее границе.

Пример:

|  |  |
| --- | --- |
| 1.04_01 | using System;  namespace Hello  {  class Program  {  static void Main()  {  Console.Write("x=");  float x = float.Parse(Console.ReadLine());  Console.Write("y=");  float y = float.Parse(Console.ReadLine());  if (x \* x + y \* y < 9 && y > 0)  Console.WriteLine("внутри");  else if (x \* x + y \* y > 9 || y < 0)  Console.WriteLine("вне");  else Console.WriteLine("на границе");  }  }  } |

Области задаются графически следующим образом:

|  |  |  |  |
| --- | --- | --- | --- |
| 2.04_02 | 3.04_03 | 4.04_04 |  |
| 5.04_05 | 6.04_06 | 7.04_07 | 8.04_08 |
| 9.04_09 | 10.04_10 | 11.04_11 | 12.04_12 |
| 13.04_13 | 14.04_14 | 15.04_15 | 16.04_16 |
| 17.04_17 | 18.04_18 | 19.04_19 | 20.04_20 |

Составить программу.

При решении данных задач возможно использовать оператор switch или вложенные операторы if.

1. Дан порядковый номер дня недели, вывести на экран его название.

Пример:

using System;

namespace Hello

{

class Program

{

static void Main()

{

Console.Write("n=");

byte n = byte.Parse(Console.ReadLine());

switch (n)

{

case 1: Console.WriteLine("понедельник"); break;

case 2: Console.WriteLine("вторник"); break;

case 3: Console.WriteLine("среда"); break;

case 4: Console.WriteLine("четверг"); break;

case 5: Console.WriteLine("пятница"); break;

case 6: Console.WriteLine("суббота"); break;

case 7: Console.WriteLine("воскресенье"); break;

default: Console.WriteLine("ВЫ ОШИБЛИСЬ"); break;

}

}

}

}

* 1. Дан порядковый номер месяца, вывести на экран количество месяцев оставшихся до конца года.
  2. Дан порядковый номер дня месяца, вывести на экран количество дней оставшихся до конца месяца.
  3. Дан номер масти m ( 1 <= m <= 4 ), определить название масти. Масти нумеруются: "пики" - 1, "трефы" - 2, "бубны" - 3, "червы" - 4.
  4. Дан номер карты k ( 6 <= k <= 14 ), определить достоинство карты. Достоинства определяются по следующему правилу: "туз" - 14, "король" - 13, "дама" - 12, "валет" - 11, "десятка" - 10, \dots, "шестерка" - 6.
  5. Дан номер масти m ( 1 <= m <= 4 ) и номер достоинства карты k ( 6 <= k <= 14 ). Определить полное название соответствующей карты в виде "дама пик", "шестерка бубен" и т.д.
  6. С 1 января 1990 года по некоторый день прошло n дней, определить название текущего месяца.
  7. С 1 января 1990 года по некоторый день прошло m месяцев, определить название текущего месяца.
  8. С некоторой даты по настоящий день прошло m месяцев, определить название месяца неизвестной даты.
  9. С некоторой даты по настоящий день прошло m месяцев, найти неизвестную дату.
  10. С некоторой даты по настоящий день прошло n дней, найти неизвестную дату.
  11. С 1 января 1990 года по некоторый день прошло m месяцев и n дней, определить название текущего месяца.
  12. Дано расписание приемных часов врача. Вывести на экран приемные часы врача в заданный день недели (расписание придумать самостоятельно).
  13. Проведен тест, оцениваемый в целочисленный баллах от нуля до ста. Вывести на экран оценку тестируемого в зависимости от набранного количества баллов: от 90 до 100 - "отлично", от 70 до 89 - "хорошо", от 50 до 69 - "удовлетворительно", менее 50 - "неудовлетворительно".
  14. Дан год. Вывести на экран название животного, символизирующего заданный год по восточному календарю.
  15. Дан возраст человека мужского пола в годах. Вывести на экран возрастную категорию: до года - "младенец", от года до 11лет - "ребенок", от 12 до 15 лет - "подросток", от 16 до 25 лет - "юноша", от 26 до 70 лет - "мужчина", более 70 лет - "старик".
  16. Дан пол человека: м - мужчина, ж - женщина. Вывести на экран возможные мужские и женские имена в зависимости от введенного пола.
  17. Дан признак транспортного средства: a - автомобиль, в - велосипед, м - мотоцикл, с - самолет, п - поезд. Вывести на экран максимальную скорость транспортного средства в зависимости от введенного признака.
  18. Дан номер телевизионного канала. Вывести на экран наиболее популярные программы заданного канала.
  19. Дан признак геометрической фигуры на плоскости: к - круг, п - прямоугольник, т - треугольник. Вывести на экран периметр и площадь заданной фигуры (данные, необходимые для расчетов, запросить у пользователя).

1. Вывести на экран:

целые числа 1, 3, 5, ..., 21 в строчку через пробел:

Пример:

using System;

namespace Hello

{

class Program

{

static void Main()

{

Console.Write("n=");

byte n = byte.Parse(Console.ReadLine());

Console.Write("while: ");

int i = 1;

while (i <= n)

{

Console.Write(" " + i);

i += 2;

}

Console.Write("do while: ");

i = 1;

do

{

Console.Write(" " + i);

i += 2;

}

while (i <= n);

Console.Write("For: ");

for (i = 1; i<=n; i+=2)

{

Console.Write(" " + i);

}

}

}

}

* 1. целые числа 10, 12, 14, ..., 60 в обратном порядке в столбик;

|  |  |
| --- | --- |
| числа следующим образом:  10 10.4  11 11.4  ...  25 25.4 | числа следующим образом:  25 25.5 24.8  26 26.5 25.8  ...  35 35.5 34.8 |

* 1. таблицу соответствия между весом в фунтах и весом в килограммах для значений 1, 2, 3, ..., 10 фунтов (1 фунтов = 453г);
  2. таблицу перевода 5, 10, 15, ..., 120 долларов США в рубли по текущему курсу (значение курса вводится с клавиатуры);
  3. таблицу стоимости для 10, 20, 30,..., 100 штук товара, при условии, что одна штука товара стоит х руб (значение *х*водится с клавиатуры);
  4. таблицу перевода расстояний в дюймах в сантиметры для значений 2, 4, 6, ..., 12 дюймов (1 дюйм = 25.4 мм);
  5. кубы всех целых чисел из диапазона от А до В ( А<=В ) в обратном порядке;
  6. все целые числа из диапазона от А до В ( А<=В ), оканчивающиеся на цифру Х ;
  7. все целые числа из диапазона от А до В ( А<=В ), оканчивающиеся на цифру Х или У ;
  8. все целые числа из диапазона от А до В ( А<=В ), оканчивающиеся на любую четную цифру;
  9. только положительные целые числа из диапазона от А до В ( А<=В );
  10. все целые числа из диапазона от А до В , кратные трем ( А<=В );
  11. все четные числа из диапазона от А до В , кратные трем ( А<=В );
  12. только отрицательные четные числа из диапазона от А до В ( А<=В );
  13. все двухзначные числа, в записи которых все цифры разные;
  14. все двухзначные числа, в которых старшая цифра отличается от младшей не больше чем на 1;
  15. все трехзначные числа, которые начинаются и заканчиваются на одну и ту же цифру;
  16. все трехзначные числа, в которых хотя бы две цифры повторяются.
  17. Вывести на экран числа следующим образом:

Пример :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1) | 1 | 1 | 1 | 1 | 1 | 1 |
|  | 2 | 2 | 2 | 2 | 2 | 2 |
|  | 3 | 3 | 3 | 3 | 3 | 3 |
|  | 4 | 4 | 4 | 4 | 4 | 4 |
|  |  |  |  |  |  |  |

using System;

namespace Hello

{

class Program

{

static void Main()

{

for (int i = 1; i<=4; ++i, Console.WriteLine())

{

for(int j=1; j<=6; ++j)

Console.Write(" " + i);

}

}

}

}

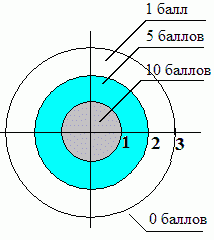
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2) | 1 | 2 | 3 | … | 10 |  | 3) | -10 | -9 | -8 | … | 12 |  |  | | | | | | |
| 1 | 2 | 3 | … | 10 | -10 | -9 | -8 | … | 12 |
| 1 | 2 | 3 | … | 10 | -10 | -9 | -8 | … | 12 |
| 1 | 2 | 3 | … | 10 | -10 | -9 | -8 | … | 12 |
|  |  |  |  |  | -10 | -9 | -8 | … | 12 |
|  | | | | | |  |  | | | | | |  |  | | | | | | |
| 4) | 41 | 42 | 43 | … | 50 |  | 5) | 5 |  |  |  |  |  | 6) | 1 | 1 | 1 | 1 | 1 |  |
| 51 | 52 | 53 | … | 60 | 5 | 5 |  |  |  | 1 | 1 | 1 | 1 |  |  |
| 61 | 62 | 63 | … | 70 | 5 | 5 | 5 |  |  | 1 | 1 | 1 |  |  |  |
| … |  |  |  |  | 5 | 5 | 5 | 5 |  | 1 | 1 |  |  |  |  |
| 71 | 72 | 73 | … | 80 | 5 | 5 | 5 | 5 | 5 | 1 |  |  |  |  |  |
|  | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7) | 1 |  |  |  |  |  | 8) | 6 | 6 | 6 | 6 | 6 |  | 9) | 7 |  |  |  |  |  |
| 2 | 2 |  |  |  | 7 | 7 | 7 | 7 |  | 6 | 6 |  |  |  |  |
| 3 | 3 | 3 |  |  | 8 | 8 | 8 |  |  | 5 | 5 | 5 |  |  |  |
| 4 | 4 | 4 | 4 |  | 9 | 9 |  |  |  | 4 | 4 | 4 | 4 |  |  |
| 5 | 5 | 5 | 5 | 5 | 10 |  |  |  |  | 3 | 3 | 3 | 3 | 3 |  |
|  | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 10) | 8 | 8 | 8 | 8 | 8 |  | 11) | 1 |  |  |  |  |  | 12) | 1 |  |  |  |  |  |
| 7 | 7 | 7 | 7 |  | 1 | 2 |  |  |  | 2 | 1 |  |  |  |  |
| 6 | 6 | 6 |  |  | 1 | 2 | 3 |  |  | 3 | 2 | 1 |  |  |  |
| 5 | 5 |  |  |  | 1 | 2 | 3 | 4 |  | 4 | 3 | 2 | 1 |  |  |
| 4 |  |  |  |  | 1 | 2 | 3 | 4 | 5 | 5 | 4 | 3 | 2 | 1 |  |
|  | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 13) | 0 | 1 | 2 | 3 | 4 |  | 14) | 4 | 3 | 2 | 1 | 0 |  | 15) | 1 |  |  |  |  |  |
| 0 | 1 | 2 | 3 |  | 3 | 2 | 1 | 0 |  | 0 |  |  |  |  |  |
| 0 | 1 | 2 |  |  | 2 | 1 | 0 |  |  | 2 | 2 |  |  |  |  |
| 0 | 1 |  |  |  | 1 | 0 |  |  |  | 0 | 0 |  |  |  |  |
| 0 |  |  |  |  | 0 |  |  |  |  | 3 | 3 | 3 |  |  |  |
|  |  |  |  |  |  |  |  |  |  | 0 | 0 | 0 |  |  |  |
|  |  |  |  |  |  |  |  |  |  | 4 | 4 | 4 | 4 |  |  |
|  |  |  |  |  |  |  |  |  |  | 0 | 0 | 0 | 0 |  |  |
|  |  |  |  |  |  |  |  |  |  | 5 | 5 | 5 | 5 | 5 |  |
|  |  |  |  |  |  |  |  |  |  | 0 | 0 | 0 | 0 | 0 | 0 |
|  | | | | | |  |  | | | | | |  |  | | | | | | |
| 16) | 8 |  |  |  |  |  | 17) | 1 |  |  |  |  |  | 18) | 9 |  |  |  |  |  |
| 7 |  |  |  |  | 6 |  |  |  |  | 4 |  |  |  |  |  |
| 7 | 7 |  |  |  | 2 | 2 |  |  |  | 8 | 8 |  |  |  |  |
| 6 | 6 |  |  |  | 7 | 7 |  |  |  | 3 | 3 |  |  |  |  |
| 6 | 6 | 6 |  |  | 3 | 3 | 3 |  |  | 7 | 7 | 7 |  |  |  |
| 5 | 5 | 5 |  |  | 8 | 8 | 8 |  |  | 2 | 2 | 2 | 2 |  |  |
| 5 | 5 | 5 | 5 |  | 4 | 4 | 4 | 4 |  | 6 | 6 | 6 | 6 | 6 |  |
| 4 | 4 | 4 | 4 |  | 9 | 9 | 9 | 9 |  | 1 | 1 | 1 | 1 | 1 |  |
|  | | | | | |  |  | | | | | |  |  | | | | | | |
| 19) | 3 |  |  |  |  |  | 20) | 2 | 2 | 2 | 2 | 2 |  |  | | | | | | |
| 0 |  |  |  |  | 3 | 4 | 5 | 6 | 7 |
| 2 | 3 |  |  |  | 2 | 2 | 2 | 2 |  |
| 9 | 0 |  |  |  | 2 | 3 | 4 | 5 |  |
| 2 | 2 | 3 |  |  | 2 | 2 | 2 |  |  |
| 8 | 9 | 0 |  |  | 1 | 2 | 3 |  |  |
| 2 | 2 | 2 | 3 |  | 2 | 2 |  |  |  |
| 7 | 8 | 9 | 0 |  | 0 | 1 |  |  |  |
| 2 | 2 | 2 | 2 | 3 | 2 |  |  |  |  |
| 6 | 7 | 8 | 9 | 0 | -1 |  |  |  |  |

## Контрольные вопросы.

1. Задана программа

static void Main()  
{  
Console.Write("x= ");  
float x = float.Parse(Console.ReadLine());  
Console.Write("y=");  
float y = float.Parse(Console.ReadLine());  
if (x < y ) Console.WriteLine("min= "+x);  
else Console.WriteLine("min= "+y);  
}  
Измените программу так, чтобы вычислялось наибольшее значение из х и у.

1. Дана мишень.

Составить программу подсчета количество баллов после выстрела по данной мишени.  
  


1. Задана программа

static void Main()

{

Console.Write("N= ");

int n=int.Parse(Console.ReadLine());

int i = 1;

do

Console.Write(" " + i++);

//выводим i на экран, затем увеличиваем его на 1

while (i <= n); //пока i меньше или равно n

}

Измените программу так, чтобы числа выводились в обратном порядке;

1. Задана программа из п.3

Измените программу так, чтобы выводились только четные числа.

1. Задана программа

static void Main()

{

Console.Write("N= ");

int n=int.Parse(Console.ReadLine());

for (int i=1; i<=n;) //блок модификации пустой

Console.Write(" " + i++);

}

Измените программу так, чтобы: числа выводились в обратном порядке;

1. Задана программа из п.5

Измените программу так, чтобы: выводились квадраты чисел.

1. Программа выводит числа следующим образом

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |

static void Main()

{

for (int i = 1; i <= 4; ++i, Console.WriteLine()) //1

for (int j=1; j<=5; ++j)

Console.Write(" " + 2);

}

Измените программу так, чтобы таблица содержала n и m столбцов (значения n и m вводятся с клавиатуры).

# C#. Введение в методы

## Теоретические сведения.

## Методы: основные понятия

Метод – это функциональный элемент класса, который реализует вычисления или другие действия, выполняемые классом или его экземпляром (объектом). Метод представляет собой законченный фрагмент кода, к которому можно обратиться по имени. Он описывается один раз, а вызываться может многократно. Совокупность методов класса определяет, что конкретно может делать класс. Например, стандартный класс Math содержит методы, которые позволяют вычислять значения математических функций.

### Синтаксис метода:

[атрибуты] [спецификторы] тип\_возвращаемого\_результата имя\_метода ([список\_параметров])

{

тело\_метода;

return значение

}

где:

* *Атрибуты и спецификторы* являются необязательными элементами синтаксиса описания метода. На данном этапе атрибуты нами использоваться не будут, а из всех спецификаторов мы в обязательном порядке будем использовать спецификатор static, который позволит обращаться к методу класса без создания его экземпляра.
* *Тип\_возвращаемого\_результата*определяет тип значения, возвращаемого методом. Это может быть любой тип, включая типы классов, создаваемые программистом. Если метод не возвращает никакого значения, необходимо указать тип void (в этом случае в теле метода отсутсвует оператор return ).
* *Имя\_метода* – идентификатор, заданный программистом с учетом требований, накладываемыми на идентификаторы в С#, отличный от тех, которые уже использованы для других элементов программы в пределах текущей области видимости.
* *Список\_параметров*представляет собой последовательность пар, состоящих из типа данных и идентификатора, разделенных запятыми. Параметры — это переменные или константы, которые получают значения, передаваемые методу при вызове. Если метод не имеет параметров, то *список\_параметров*остается пустым.
* *Значение*определяет значение, возвращаемое методом. Тип значения должен соответствовать *типу\_возвращаемого\_результата*или приводится к нему.

Пример метода:

class Program

{

static void Func() //дополнительный метод

{

Console.Write("x= ");

double x=double.Parse(Console.ReadLine());

double y = 1 / x;

Console.WriteLine("y({0})={1}", x,y );

}

static void Main() //точка входа в программу

{

Func(); //первый вызов метода Func

Func(); //второй вызов метода Func

}

}

В даном примере в метод Func не передаются никакие значения, поэтому список параметров пуст. Кроме того метод ничего не возвращает, поэтому тип возвращаемого значения void. В основном методе Main мы вызвали метод Func два раза. Если будет необходимо, то данный метод можно будет вызвать еще столько раз, сколько потребуется для решения задачи.

Изменим исходный пример так, чтобы в него передавалось значение х, а сам метод возвращал значение y.

class Program

{

static double Func( double x) //дополнительный метод

{

return 1 / x; //Возвращаемое значение

}

static void Main() //точка входа в программу

{

Console.Write("a=");

double a=double.Parse(Console.ReadLine());

Console.Write("b=");

double b=double.Parse(Console.ReadLine());

for (double x = a; x <= b; x += 0.5)

{

double y = Func(x); //вызов метода Func

Console.WriteLine("y({0:f1})={1:f2}", x, y);

}

}

В данном примере метод Func содержит параметр х, тип которого double. Для того, чтобы метод Func возвращал в вызывающий его метод Main значение выражения 1/x (тип которого double ), перед именем метода указывается тип возвращаемого значения – double, а в теле метода используется *оператор передачи управления* – return. Оператор returnзавершает выполнение метода и передает управление в точку его вызова.

Другой пример:

class Program

{

static int Func( int x, int y) //строка 1

{

return (x>y)? x:y;

}

static void Main()

{

Console.Write("a=");

int a = int.Parse(Console.ReadLine());

Console.Write("b=");

int b = int.Parse(Console.ReadLine());

Console.Write("c=");

int c = int.Parse(Console.ReadLine());

int max = Func(Func(a, b), c); //строка 2 - вызовы метода Func

Console.WriteLine("max({0}, {1}, {2})={3}", a, b, c, max);

}

}

В данном примере метод Func имеет два целочисленных параметра – x, y, а в качестве результата метод возвращает наибольшее из них. На этапе описания метода (строка 1) указываются *формальные* параметры, на этапе вызова (строка 2) в метод передаются *фактические* параметры, которые по количеству и по типу совпадают с формальными параметрами. Если количество фактических и формальных параметров будет различным, то компилятор выдаст соответствующее сообщение об ошибке. Если параметры будут отличаться типами, то компилятор попытается выполнить *неявное преобразование типов*. Если *неявное преобразование* невозможно, то также будет сгенерирована ошибка.

Обратите внимание на то, что при вызове метода Func использовалось вложение одного вызова в другой.

В общем случае параметры используются для обмена информацией между вызывающим и вызываемым методами.

### Типы параметров

В С# для обмена предусмотрены следующие типы параметров: параметры-значения, параметры-ссылки, выходные параметры, параметры-массивы.

При передаче параметра *по значению* метод получает копии параметров, и операторы метода работают с этими копиями. Доступа к исходным значениям параметров у метода нет, а, следовательно, нет и возможности их изменить.

Все примеры, рассмотренные ранее, использовали передачу данных по значению.

Пример:

class Program

{

static void Func(int x)

{

x += 10; // изменили значение параметра

Console.WriteLine("In Func: " + x);

}

static void Main()

{

int a=10;

Console.WriteLine("In Main: "+ a);

Func(a);

Console.WriteLine("In Main: " + a);

}

}

Результат работы программы:

In Main: 10

In Func: 20

In Main: 10

В данном примере значение формального параметра х было изменено в методе Func, но эти изменения не отразились на фактическом параметре а метода Main.

При передаче параметров *по ссылке* метод получает копии адресов параметров, что позволяет осуществлять доступ к ячейкам памяти по этим адресам и изменять исходные значения параметров. Для того чтобы параметр передавался по ссылке, необходимо при описании метода перед формальным параметром и при вызове метода перед соответствующим фактическим параметром поставить служебное слово ref.

class Program

{

static void Func(int x, ref int y)

{

x += 10; y += 10; //изменение параметров

Console.WriteLine("In Func: {0}, {1}", x, y);

}

static void Main()

{

int a=10, b=10; // строка 1

Console.WriteLine("In Main: {0}, {1}", a, b);

Func(a, ref b);

Console.WriteLine("In Main: {0}, {1}", a, b);

}

}

Результат работы программы:

In Main: 10 10

In Func: 20 20

In Main: 10 20

В данном примере в методе Func были изменены значения формальных параметров х и y. Эти изменения не отразились на фактическом параметре а, т.к. он передавался по значению, но значение b было изменено, т.к. он передавался по ссылке.

Передача параметра по ссылке требует, чтобы аргумент был инициализирован до вызова метода (см. строку 1). Если в этой строке не проводить инициализацию переменных, то компилятор выдаст сообщение об ошибке.

Однако в некоторых случаях бывает невозможно инициализировать параметр до вызова метода. Тогда параметр следует передавать как выходной, используя спецификатор out.

class Program

{

static void Func(int x, out int y)

{

x += 10; y = 10; // определение значения выходного параметра y

Console.WriteLine("In Func: {0}, {1}", x, y);

}

static void Main()

{

int a=10, b;

Console.WriteLine("In Main: {0}", a);

Func(a, out b);

Console.WriteLine("In Main: {0}, {1}", a, b);

}

}

Результат работы программы:

In Main: 10

In Func: 20 10

In Main: 10 10

В данном примере в методе Func формальный параметр y и соответствующий ему фактический параметр b метода Main были помечены спецификатором out. Поэтому значение b до вызова метода Func можно было не определять, но изменение параметра y отразились на изменении значения параметра b.

### Использование делегатов в качестве параметров

Если в метод требуется передать в качестве парметра ссылку на друглй метод, то для этой цели испльзуется соответствующий делегат метода.

Описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

[ атрибуты ] [ спецификаторы ] delegate тип имя\_делегата ( [ параметры ] )

Допускаются спецификаторы new, public, protected, internal и private. *Тип* описывает возвращаемое значение методов, вызываемых с помощью делегата, а необязательными *параметрами* делегата являются параметры этих методов. Делегат может хранить ссылки на несколько методов и вызывать их поочередно; естественно, что сигнатуры всех методов должны совпадать.

Пример описания делегата:

public delegate void D ( int i );

Делегаты применяются в основном для следующих целей:

* получения возможности определять вызываемый метод не при компиляции, а во время выполнения программы;
* обеспечения связи между объектами по типу "источник — наблюдатель";
* создания универсальных методов, в которые можно передавать другие методы;
* поддержки механизма обратных вызовов.

Пример.

using System;

namespace ConsoleApplication1

{

delegate void Del ( ref string s ); // объявление делегата

class Class1

{

public static void C00l ( ref string s ) // метод 1

{

string temp = "";

for ( int i = 0; i < s.Length; ++i )

{

if ( s[i] == 'o' || s[i] == 'O') temp += '0';

else if ( s[i] == 'l' ) temp += '1';

else temp += s[i];

}

s = temp;

}

public static void Hack ( ref string s ) // метод 2

{

string temp = "";

for ( int i = 0; i < s.Length; ++i )

if ( i / 2 \* 2 == i ) temp += char.ToUpper( s[i] );

else temp += s[i];

s = temp;

}

static void Main()

{

string s = "cool abcd";

Del d; // экземпляр делегата

for ( int i = 0; i < 2; ++i )

{

d = new Del( C00l ); // инициализация методом 1

if ( i == 1 ) d = new Del(Hack); // инициализация методом 2

d( ref s ); // использование делегата для вызова методов

Console.WriteLine( s );

}

}

}

}

Результат работы программы:

c001 abcd

C001 aBcD

### Перегрузка методов

Иногда бывает удобно, чтобы методы, реализующие один и тот же алгоритм для различных типов данных, имели одно и то же имя. Использование нескольких методов с одним и тем же именем, но различными типами и количеством параметров называется *перегрузкой методов*.Компилятор определяет, какой именно метод требуется вызвать, по типу и количеству фактических параметров.

Пример:

class Program

{

static int max(int a) //первая версия метода max

{

int b = 0;

while (a > 0)

{

if (a % 10 > b) b = a % 10;

a /= 10;

}

return b;

}

static int max(int a, int b) //вторая версия метода max

{

if (a > b) return a;

else return b;

}

static int max(int a, int b, int c) //третья версия метода max

{

if (a > b && a > c) return a;

else if (b > c) return b;

else return c;

}

static void Main()

{

int a = 1283, b = 45, c = 35740;

Console.WriteLine(max(a));

Console.WriteLine(max(a, b));

Console.WriteLine(max(a, b, c));

}

}

При вызове метода max компилятор выбирает вариант, соответствующий типу и количеству передаваемых в метод аргументов. Если точного соответствия не найдено, выполняются *неявные преобразования типов* в соответствии с общими правилами. Если преобразование невозможно, выдается сообщение об ошибке. Если выбор *перегруженного метода* возможен более чем одним способом, то выбирается "лучший" из вариантов (вариант, содержащий меньшие количество и длину преобразований в соответствии с правилами преобразования типов). Если существует несколько вариантов, из которых невозможно выбрать лучший, выдается сообщение об ошибке.

*Перегрузка методов* является проявлением *полиморфизма*, одного из основных свойств ООП. Программисту гораздо удобнее помнить одно имя метода и использовать его для работы с различными типами данных, а решение о том, какой вариант метода вызвать, возложить на компилятор. Этот принцип широко используется в классах библиотеки .NET. Например, в стандартном классе Console метод WriteLine перегружен 19 раз для вывода величин разных типов.

## Задания на лабораторную работу.

* 1. Разработать метод min(a,b) для нахождения минимального из двух чисел. Вычислить с помощью него значение выражения z=min(3x,2y)+min(x-y,x+y).

Пример.

using System;

namespace Hello

{

class Program

{

static double min(double a, double b)

{

return (a < b) ? a : b;

}

static void Main(string[] args)

{

Console.Write("x=");

double x = double.Parse(Console.ReadLine());

Console.Write("y=");

double y = double.Parse(Console.ReadLine());

double z = min(3 \* x, 2 \* y) + min(x - y, x + y);

Console.WriteLine("z=" + z);

}

}

}

* 1. Разработать метод \min{(a,b)} для нахождения минимального из двух чисел. Вычислить с помощью него минимальное значение из четырех чисел x, y, z, v.
  2. Разработать метод \max{(a,b)} для нахождения максимального из двух чисел. Вычислить с помощью него значение выражения z=\max(x,2y-x)+\max(5x+3y,y).
  3. Разработать метод f(x), который вычисляет значение по следующей формуле: f(x)=x^{3}-\sin x. Определить, в какой из точек а или b, функция принимает наибольшее значение.
  4. Разработать метод f(x), который вычисляет значение по следующей формуле: f(x)=\cos{(2x)}+\sin{(x-3)}. Определить, в какой из точек а или b, функция принимает наименьшее значение.
  5. Разработать метод f(x), который возвращает младшую цифру натурального числа x. Вычислить с помощью него значение выражения z=f(a)+f(b).
  6. Разработать метод f(x), который возвращает вторую справа цифру натурального числа x. Вычислить с помощью него значение выражения z=f(a)+f(b)-f(c).
  7. Разработать метод f(n), который для заданного натурального числа n находит значение \sqrt{n}+n. Вычислить с помощью него значение выражения \cfrac{\sqrt{6}+6 }{2}+\cfrac{\sqrt{13}+13 }{2}+\cfrac{\sqrt{21}+21 }{2}.
  8. Разработать метод f(n, x), который для заданного натурального числа n и вещественного х находит значение выражения \cfrac{x^n}{n}. Вычислить с помощью данного метода значение выражения \cfrac{x^2}{2}+\cfrac{x^4}{4}+\cfrac{x^6}{6}.
  9. Разработать метод f(x), который нечетное число заменяет на 0, а четное число уменьшает в два раза. Продемонстрировать работу данного метода на примере.
  10. Разработать метод f(x), который число, кратное 5, уменьшает в 5 раз, а остальные числа увеличивает на 1. Продемонстрировать работу данного метода на примере.
  11. Разработать метод f(x), который в двузначном числе меняет цифры местами, а остальные числа оставляет без изменения. Продемонстрировать работу данного метода на примере.
  12. Разработать метод f(x), который в трехзначном числе меняет местами первую с последней цифрой, а остальные числа оставляет без изменения. Продемонстрировать работу данного метода на примере.

|  |  |
| --- | --- |
| 05_05Разработать метод f(a, b), который по катетам a и b вычисляет гипотенузу. С помощью данного метода найти периметр фигуры ABCD по заданным сторонам AB, AC и DC. |  |

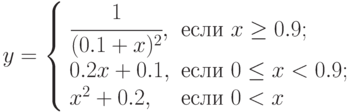
|  |  |
| --- | --- |
| 05_06Разработать метод f(x, y, z), который по длинам сторон треугольника x, y, z  вычисляет его площадь. С помощью данного метода по заданным вещественным числам a, b, c, d, e, f, g найти площадь пятиугольника, изображенного на рисунке. |  |

* 1. Разработать метод f(x_{1}, y_{1}, x_{2}, y_{2}), который вычисляет длину отрезка по координатам вершин (x_{1}, y_{1}) и (x_{2}, y_{2}), и метод d(a, b, c), который вычисляет периметр треугольника по длинам сторон a, b, c. С помощью данных методов найти периметр треугольника, заданного координатами своих вершин.
  2. Разработать метод f(x_{1}, y_{1}, x_{2}, y_{2}), который вычисляет длину отрезка по координатам вершин (x\_{1}, y\_{1}) и (x_{2}, y_{2}), и метод max(a, b), который вычисляет максимальное из чисел a, b. С помощью данных методов определить, какая из трех точек на плоскости наиболее удалена от начала координат.
  3. Разработать метод f(x_{1}, y_{1}, x_{2}, y_{2}), который вычисляет длину отрезка по координатам вершин (x_{1}, y_{1}) и (x_{2}, y_{2}), и метод min(a, b), который вычисляет минимальное из чисел a, b. С помощью данных методов найти две из трех заданных точек на плоскости, расстояние между которыми минимально.
  4. Разработать метод f(x_{1}, y_{1}, x_{2}, y_{2}), который вычисляет длину отрезка по координатам вершин (x_{1}, y_{1}) и (x_{2}, y_{2}), и метод t(a, b, c), который проверяет, существует ли треугольник с длинами сторон a, b, c. С помощью данных методов проверить, можно ли построить треугольник по трем заданным точкам на плоскости.
  5. Разработать метод f(x_{1}, y_{1}, x_{2}, y_{2}), который вычисляет длину отрезка по координатам вершин (x_{1}, y_{1}) и (x_{2}, y_{2}), и метод t(a, b, c), который проверяет, существует ли треугольник с длинами сторон a, b, c. С помощью данных методов проверить, сколько различных треугольников можно построить по четырем заданным точкам на плоскости.

Задания повышенной сложности

1. Постройте таблицу значений функции y=f(x)  для х \in [a, b] с шагом h.

Для решения задачи использовать вспомогательный метод.



Пример:

using System;

namespace Hello

{

class Program

{

static double f (double x)

{

double y;

if (x >= 0.9) y = 1 / Math.Pow(1 + x, 2);

else if (x >= 0) y = 0.2 \* x + 0.1;

else y = x \* x + 0.2;

return y;

}

static void Main(string[] args)

{

Console.Write("a=");

double a = double.Parse(Console.ReadLine());

Console.Write("b=");

double b = double.Parse(Console.ReadLine());

Console.Write("h=");

double h = double.Parse(Console.ReadLine());

for (double i = a; i <= b; i += h)

Console.WriteLine("f({0:f2})={1:f4}", i, f(i));

}

}

}

|  |  |
| --- | --- |
|  | 2. y=\left\{\begin{array}{ll} \sin{(x)}, & \text{если } |x|<3; \\ \cfrac{\sqrt{x^2+1}}{\sqrt{x^2+5}}, &  \text{если } 3\le |x| < 9; \\ \sqrt{x^2+1}-\sqrt{x^2+5} &  \text{если } |x|\ge9 \end{array} |
| 3. y=\left\{\begin{array}{ll}0, & \text{если } x<a; \\\cfrac{x-a}{x+a}, &  \text{если } x > a; \\1 &  \text{если } x=a\end{array} | 4. y=\left\{\begin{array}{ll} x^3-0.1, & \text{если } |x|\le0.1; \\ 0.2x-0.1, &  \text{если } 0.1< x \le 0.2; \\ x^3+0.1 &  \text{если } |x|>0.2\end{array} |
| 5. y=\left\{\begin{array}{ll} a+b, & \text{если } x^2-5x<0; \\ a-b, &  \text{если } 0\le (x^2-5x) <10; \\ ab &  \text{если } x^2-5x \ge10\end{array} | 6. y=\left\{\begin{array}{ll} x^2, & \text{если } (x^2+2x+1)<2; \\ \cfrac{1}{x^2-1}, &  \text{если } 2\le (x^2+2x+1) <3; \\ 0 &  \text{если } (x^2+2x+1) \ge3\end{array} |
| 7. y=\left\{\begin{array}{ll} -4, & \text{если } x<0; \\ x^2, &  \text{если } 0\le x < 1; \\ 2 &  \text{если } x\ge1 \end{array} | 8. y=\left\{\begin{array}{ll} x^2-1, & \text{если } |x|\le1; \\ 2x-1, &  \text{если } 1< x \le 2; \\ x^5-1 &  \text{если } |x|>2\end{array} |
| 9. y=\left\{\begin{array}{ll} (x^2-1)^2, & \text{если } x<1; \\ \cfrac{1}{(1+x)^2}, &  \text{если } x > 1; \\ 0 &  \text{если } x=1\end{array} | 10. y=\left\{\begin{array}{ll} x^2, & \text{если } (x+2)<1; \\ \cfrac{1}{x+2}, &  \text{если } 1\le (x+2) <10; \\ x+2 &  \text{если } (x+2) \ge10\end{array} |
| 11. y=\left\{\begin{array}{ll} x^2+5, & \text{если } x\le 5; \\ 0, &  \text{если } 5< x <20; \\ 1 &  \text{если } x \ge20\end{array} | 12. y=\left\{\begin{array}{ll} 0, & \text{если } x<0; \\ x^2+1, &  \text{если } x > 0 \text{ и } x \ne 1; \\ 1 &  \text{если } x=1\end{array} |
| 13. y=\left\{\begin{array}{ll} 1, & \text{если } x=1 \text{ или } x=-1; \\ \cfrac{1}{1-x}, &  \text{если } x \ge 0 \text{ и } x \ne 1; \\ \cfrac{1}{1+x}, &  \text{если } x < 0 \text{ и } x \ne -1; \end{array} | 14. y=\left\{\begin{array}{ll} 0.2x^2-x-0.1, & \text{если } x<0; \\ \cfrac{x^2}{x-0.1}, &  \text{если } x > 0 \text{ и } x \ne 0.1; \\ 0 &  \text{если } x=1\end{array} |
| 15. y=\left\{\begin{array}{ll} 1, & \text{если } (x-1)<1; \\ 0, &  \text{если } (x-1)=1; \\ -1 &  \text{если } (x-1)>1\end{array} | 16. y=\left\{\begin{array}{ll} x, & \text{если } x>0; \\ 0, &  \text{если } -1\le x \le 0; \\ x^2 &  \text{если } x<-1 \end{array} |
| 17. y=\left\{\begin{array}{ll} a+bx, & \text{если } x<93; \\ b-ac, &  \text{если } 93\le x \le 120; \\ abx &  \text{если } x>120 \end{array} | 18. y=\left\{\begin{array}{ll}x^2-0.3, & \text{если } x<3; \\ 0, &  \text{если } 3\le x \le 5; \\ x^2+1, &  \text{если } x>5\end{array} |
| 19. y=\left\{\begin{array}{ll}\sqrt{5x^2+5}, & \text{если } |x|< 2; \\\cfrac{|x|}{\sqrt{5x^2+5}}, &  \text{если } 2\le |x| < 10; \\0 &  \text{если } |x|\ge10\end{array} | 20. y=\left\{\begin{array}{ll}\sin{(x)}, & \text{если } |x|< \cfrac{\pi}{2}; \\\cos{(x)}, &  \text{если } \cfrac{\pi}{2}\le |x| \le \pi; \\0 &  \text{если } |x|>\pi\end{array} |

1. Перегрузите метод f из предыдущего раздела так, чтобы его сигнатура (заголовок) соответствовала виду static void f (double x, out double y).

## Контрольные вопросы.

1. Задан метод Func

class Program  
 {  
 static double Func( double x) //дополнительный метод  
 {  
 return 1 / x; //Возвращаемое значение  
 }  
 static void Main() //точка входа в программу  
 {  
 Console.Write("a=");  
 double a=double.Parse(Console.ReadLine());  
 Console.Write("b=");  
 double b=double.Parse(Console.ReadLine());  
 for (double x = a; x <= b; x += 0.5)  
 {  
 double y = Func(x); //вызов метода Func  
 Console.WriteLine("y({0:f1})={1:f2}", x, y);   
 }  
 }

Преобразуйте программу так, чтобы метод *Func* возвращал значение выражения:

1. Преобразуйте программу п. 1 так, чтобы метод *Func* возвращал значение выражения:
2. Задана программа

class Program  
 {  
 static int Func( int x, int y) //строка 1  
 {  
 return (x>y)? x:y;  
 }  
  
 static void Main()  
 {  
 Console.Write("a=");  
 int a = int.Parse(Console.ReadLine());  
 Console.Write("b=");  
 int b = int.Parse(Console.ReadLine());  
 Console.Write("c=");  
 int c = int.Parse(Console.ReadLine());  
 int max = Func(Func(a, b), c); //строка 2 - вызовы метода Func  
 Console.WriteLine("max({0}, {1}, {2})={3}", a, b, c, max);   
 }  
 }

Преобразуйте программу так, чтобы с помощью метода Func можно было найти наибольшее значение из четырех чисел: a, b, c, d. Метод Func при этом не изменять.

1. Что означает передача параметра по значению
2. Что означает передача параметра по ссылке
3. Как использовать выходной параметр
4. Что означает перегрузка метода

# C#. Рекурсия

## Теоретические сведения.

Рекурсивным называют метод, если он вызывает сам себя в качестве вспомогательного. В основе *рекурсивного метода* лежит так называемое "рекурсивное *определение*" какого-либо понятия. Классическим примером *рекурсивного метода* является метод, вычисляющий *факториал*.

Из курса математики известно, что 0!=1!=1, n!=1\*2\*3…\*n. С другой стороны n!=(n-1)!\*n. Таким образом, известны два частных случая параметра n, а именно n= 0 и n=1, при которых мы без каких-либо дополнительных вычислений можем определить *значение* факториала. Во всех остальных случаях, то есть для n>1, *значение* факториала может быть вычислено через *значение* факториала для параметра n-1. Таким образом, *рекурсивный метод* будет иметь вид:

{

static long F(int n) //рекурсивный метод

{

if (n==0 || n==1)

return 1; //нерекурсивная ветвь

else return n\*F(n-1); //шаг рекурсии - повторный вызов метода с другим

//параметром

}

static void Main()

{

Console.Write("n=");

int n =int.Parse( Console.ReadLine());

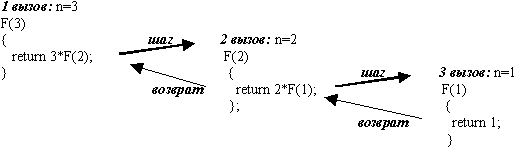
long f=F(n); //нерекурсивный вызов метода F

Console.WriteLine("{0}!={1}",n, f);

}

}

Рассмотрим работу описанного выше *рекурсивного метода* для n=3.



Первый *вызов метода* осуществляется из метода Main, в нашем случае командой f=F(3). Этап вхождения в рекурсию обозначим жирными стрелками. Он продолжается до тех пор, пока *значение* переменной n не становится равной 1. После этого начинается *выход* из рекурсии (тонкие стрелки). В результате вычислений получается, что F(3)=3\*2\*1.

Рассмотренный вид рекурсии называют *прямой*. Метод с *прямой* рекурсией обычно содержит следующую структуру:

if (<условие>)

<оператор>;

else <вызов данного метода с другими параметрами>;

В качестве <условия> обычно записываются некоторые граничные случаи параметров, передаваемых рекурсивному методу, при которых результат его работы заранее известен, поэтому далее следует простой оператор или блок, а в ветви else происходит *рекурсивный вызов* данного метода с другими параметрами.

Со входом в рекурсию осуществляется *вызов метода*, а для выхода необходимо помнить точку возврата, т.е. то *место* программы откуда пришли и куда нужно будет возвратиться после завершения метода. *Место* хранения точек возврата называется стеком вызовов и для него выделяется определенная область оперативной памяти. В этом стеке запоминаются не только адреса точек возврата, но и копии значений всех параметров. *По* этим копиям восстанавливается при возврате вызывающий метод. При развертывании рекурсии за счет создания копий параметров возможно *переполнение* стека. Это является основным недостатком *рекурсивного метода*. С другой стороны, *рекурсивные методы* позволяют перейти к более компактной записи алгоритма.

Любой *рекурсивный метод* можно преобразовать в обычный метод. И практически любой метод можно преобразовать в рекурсивный, если выявить *рекуррентное соотношение* между вычисляемыми в методе значениями.

Далее для сравнения каждую задачу будем решать с использованием обычного и рекурсивного методов:

**Пример 1**. Найти сумму цифр числа А.

Известно, что любое *натуральное число* A=an an-1... a1 a0, где an an-1... a1 a0 - цифры числа, можно представить следующим образом:

A=an an-1... a1 a0 = A=an\*10n + an-1\*10n-1 + ... a1\*101 + a0\*100 = ((...((an\*10 + an-1)\*10+ an-2)\*10...)\*10 + a1)\*10 + a0

Например, число 1234 можно представить как:

1234 = 1\*103+ 2\*102+ 3\*101+ 4\*100= ((1\*10 + 2)\*10 + 3)\*10 + 4

Из данного представления видно, что получить последнюю цифру можно, если найти *остаток от деления* числа на 10. В связи с этим для разложения числа на составляющие его цифры можно использовать следующий *алгоритм*:

1. Получить остаток при делении числа А на 10, т.е. получаем крайнюю правую цифру числа.
2. Получить целую часть числа при делении A на 10, т.е. отбрасываем от числа A крайнюю правую цифру.
3. Если преобразованное A > 0, то переход на пункт 1. Иначе число равно нулю и отделять от него больше нечего.

Данный *алгоритм* будет использоваться при разработке нерекурсивного метода.

С другой стороны, сумму цифр числа 1234 можно представить следующим образом sum(1234)=sum(123)+4=(sum(12)+3)+4=(((sum(1)+2)+3)+4)=(((sum(0)+1)+2)+3)+4. Таким образом, если А=0, то сумма цифр числа также равна нулю, т.е. sum=0. В противном случае сумму цифр числа A можно представить *рекуррентным соотношением* sum(A)=sum(A/10)+A%10. Полученное *рекуррентное соотношение* будем использовать при разработке *рекурсивного метода*.

class Program

{

static long Sum(long a) //нерекурсивный метод

{

long sum=0;

while (a>0) //пока a больше нуля

{

sum+=a%10; //добавляем к сумме последнюю цифру числа а

a/=10; //отбрасываем от числа а последнюю цифру

}

return sum; //возвращаем в качестве результата сумму цифр числа a

}

static long SumR(long a) //рекурсивный метод

{

if (a==0) //если a =0, то

return 0; // возвращаем 0

else return SumR(a/10)+ a%10; //иначе обращаемся к рекуррентному

// соотношению

}

static void Main()

{

Console.Write("n=");

long n=long.Parse(Console.ReadLine());

Console.WriteLine("Нерекурсивный метод: "+Sum(n));

Console.WriteLine("Рекурсивный метод: "+SumR(n));

}

}

}

Рассмотренные выше *рекурсивные методы* возвращали некоторое *значение*, заданное *рекуррентным соотношением*. Однако не все методы возвращают *значение*. Кроме того, рассмотренные выше методы определяют простой вариант *рекурсивного метода*. В общем случае *рекурсивный метод* включает в себя некоторое множество операторов и один или несколько операторов рекурсивного вызова. Действия могут выполняться после рекурсивного вызова, до рекурсивного вызова, а также и до, и после рекурсивного вызова. Рассмотрим примеры "сложных" *рекурсивных методов*, не возвращающих *значение*.

**Пример 2**. Для заданного значения n вывести на экран n строк, в каждой из которых содержится n звездочек. Например, для n=5 на экран нужно вывести следующую таблицу:

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

class Program

{

static void Stroka(int n) //выводит на экран строку из n звездочек

{

for (int i=1; i<=n; ++i)

{

Console.Write('\*');

}

Console.WriteLine();

}

static void Star(int n) //нерекурсивный метод

{

for (int i=1; i<=n;++i) //выводит n строк по i звездочек в каждой

Stroka(i);

}

//рекурсивный метод, где i – номер текущей строки, n – номер последней строк

static void StarR(int i,int n)

{

if (i<=n ) //если номер текущей строки не больше номера последней строки, то

{

Stroka(i); //выводим i звездочек в текущей строке и

StarR(i+1,n); //переходим к формированию следующей строки

}

}

static void Main()

{

Console.Write("n=");

int n=int.Parse(Console.ReadLine());

Console.WriteLine("Нерекурсивный метод: ");

Star(n);

Console.WriteLine("Рекурсивный метод: ");

StarR(1,n); // параметр 1 – это номер первой строки, n – номер последней строки

}

}

Все примеры, рассмотренные ранее, относились к *прямой* рекурсии. Однако существует еще и косвенная *рекурсия*, в которой метод вызывает себя в качестве вспомогательного не непосредственно, а через другой вспомогательный метод. *Косвенную рекурсию* демонстрирует следующая *программа*, которая для заданного значения n выводит на экран следующее за ним *простое число*.

Данная *программа* содержит метод Prim, который возвращает true, если его *параметр* является простым числом, false – в противном случае. Чтобы установить, является ли число j простым, нужно проверить делимость числа j на все простые числа, не превышающие квадратный корень из j. Перебор таких простых чисел можно организовать так: рассмотреть первое *простое число* – 2, а затем, используя метод NextPrim, возвращающий следующее за значением ее параметра *простое число*, получить все простые числа, не превышающие квадрата числа j. В свою *очередь* метод NextPrim обращается к методу Prim для того, чтобы определить является ли заданное число простым.

Таким образом методы Prim и NextPrim перекрестно вызывают друг друга. В этом и проявляется косвенная *рекурсия*.

class Program

{

static bool Prim (int j)

{

int k=2; //первое простое число

//значение k "пробегает" последовательность простых чисел, начиная с 2 до корня из j, при

//этом проверяется делится ли j на одно из таких простых чисел

while (k\*k<=j && j%k!=0)

k=NextPrim(k); //вызов метода NextPrim

return (j%k==0)?false:true;

}

static int NextPrim(int i)

{

int p=i+1;

while (!Prim(p)) //вызов метода Prim

++p;

return p;

}

static void Main()

{

Console.Write("n=");

int n=int.Parse(Console.ReadLine());

Console.WriteLine("Следующее за {0} простое число равно {1}.", n, NextPrim(n));

}

}

*Рекурсия* является средством решения многих задач: сортировки числовых массивов, обхода таких структур данных как деревья и графы.

С другой стороны, применение *рекурсивных методов* в ряде случаев оказывается нерациональным.

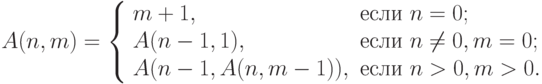
Например, *рекурсивный метод* подсчета n-ного члена последовательности Фибоначчи. Данный метод будет работать неэффективно. FbR(17) вычисляется в ней как FbR(16)+ FbR(15). В свою *очередь* FbR(16) вычисляется в ней как FbR(15)+ FbR(14). Таким образом, FbR(15) будет вычисляться 2 раза, FbR(14) – 3 раза, FbR(13) – 5 раз и т.д. Всего для вычисления FbR(17) потребуется выполнить более тысячи операций сложения. Для сравнения при вычислении Fb(17), т.е. используя не *рекурсивный метод*, потребуется всего лишь 15 операций сложения.

## Задания на лабораторную работу.

1. Разработать рекурсивный метод (возвращающий значение):
   1. для вычисления n -го члена следующей последовательности b_{1} = -10, b_{2} = 2, b_{n+2} = |b_{n}| – 6 b_{n+1}.
   2. для вычисления n -го члена следующей последовательности b_{1} = 5, b_{n+1} = b_{n}/(n^{2}+n+1)
   3. для нахождения наибольшего общего делителя методом Евклида:

НОД(a,b)=\left\{\begin{array}{ll}
a, & \text{если } a=b;\\
НОД(a-b,b), &  \text{если } a > b;\\
НОД(a,b-a), &  \text{если } b > a;\end{array}

* 1. для вычисления значения функции Аккермана для неотрицательных чисел n и m. Функция Аккермана определяется следующим образом:



* 1. для вычисления числа сочетаний C(n, m) где 0 \le m \le n, используя следующие свойства

C_n^0 = C_n^n =1; C_n^m = C_{n-1}^{m} + C_{n-1}^{m-1} =1; при 0<m<n.

* 1. вычисляющий число *а*, для которого выполняется неравенство 2^{a-1} \le n \le 2^{a}, где n  – натуральное число. Для подсчета числа а использовать формулу: a(n)=\left\{\begin{array}{ll}
     1, & \text{если } n=1;\\
     a(n/2), &  \text{если } n >1;
     \end{array}
  2. для вычисления x^{n} ( x –вещественное, x \ne 0, а n –целое) по формуле:

x^n=\left\{\begin{array}{ll}
1, & \text{если } n=0;\\
\frac{1}{x^{|n|}}, &  \text{если } n <0;\\
x \cdot{x^{n-1}}, &  \text{если } n >0;
\end{array}.

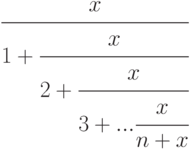
Вычислить значение x^{n} для различных x и n.

* 1. для вычисления \sum_{i=1}^n{i}, где n – натуральное число. Для заданных натуральных чисел m и k вычислить с помощью разработанного метода значение выражения \sum_{i=1}^n{i}+\sum_{i=1}^{2k}{i}.
  2. для вычисления значения функции

F(N)=\frac{N}{\sqrt{1+\sqrt{2+\sqrt{3+...\sqrt{N}}}}}.

Найти ее значение при заданном натуральном N.

* 1. для вычисления цепной дроби:

.

Найти значение данной дроби при заданном натуральном n.

1. Разработка рекурсивных методов ( не возвращающих значений):
   1. Даны первый член и разность арифметической прогрессии. Написать *рекурсивный метод* для нахождения n -го члена и суммы n  первых членов прогрессии.
   2. Даны первый член и знаменатель геометрической прогрессии. Написать *рекурсивный метод* для нахождения n-го члена и суммы n  первых членов прогрессии.
   3. Разработать *рекурсивный метод*, который по заданному натуральному числу N ( N \ge 1000 ) выведет на экран все натуральные числа не больше N в порядке возрастания. Например, для N=8, на экран выводится 1 2 3 4 5 6 7 8.
   4. Разработать *рекурсивный метод*, который по заданному натуральному числу N ( N \ge 1000 ) выведет на экран все натуральные числа не больше N  в порядке убывания. Например, для N=8, на экран выводится 8 7 6 5 4 3 2 1.
   5. Дано натуральное число n. Разработать *рекурсивный метод* для вывода на экран следующей последовательности чисел:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 |  |  |  |  |
| 2 | 2 |  |  |  |
| 3 | 3 | 3 |  |  |
| … |  |  |  |  |
| n | n | n | … | n |

* 1. Дано натуральное число n. Разработать *рекурсивный метод* для вывода на экран следующей последовательности чисел:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 |  |  |  |  |
| 2 | 1 |  |  |  |
| 3 | 2 | 1 |  |  |
| … |  |  |  |  |
| n | n-1 | n-2 | … | 1 |

* 1. Разработать *рекурсивный метод* для вывода на экран цифр натурального числа в прямом порядке. Применить эту процедуру ко всем числам из интервала от А до В.
  2. Разработать *рекурсивный метод* для перевода числа из десятичной системы счисления в двоичную.
  3. Разработать *рекурсивный метод* для перевода числа из двоичной системы счисления в десятичную.
  4. Разработать *рекурсивный метод* для вывода на экран всех делителей заданного натурального числа n.
  5. Дано натуральное четное число n. Разработать *рекурсивный метод* для вывода на экран следующей картинки:

|  |  |
| --- | --- |
| \*\*\*\*\*\*\*\*\* | (0 пробелов, n звездочек) |
| \*\*\*\*\*\*\*\* | (1 пробел, n-1 звездочка) |
| \*\*\*\*\*\*\* | (2 пробела, n-2 звездочки) |
| … |  |
| \* | (n-1 пробел, 1 звездочка) |

* 1. Дано натуральное четное число n. Разработать *рекурсивный метод* для вывода на экран следующей картинки:

|  |  |
| --- | --- |
| \* \* | (n пробелов между звездочками) |
| \*\* \*\* | (n-2 пробела) |
| \*\*\* \*\*\* | (n-4 пробела) |
| … | … |
| \*\*\*\*\* \*\*\*\*\* | (2 пробела) |
| \*\*\*\*\*\*\*\*\*\* | (0 пробелов) |
| \*\*\*\*\* \*\*\*\*\* | (2 пробела) |
| … | … |
| \*\*\* \*\*\* | (n-4 пробела) |
| \*\* \*\* | (n-2 пробела) |
| \* \* | (n пробелов |

* 1. Дано натуральное число n. Разработать *рекурсивный метод* для вывода на экран следующей картинки:

|  |  |
| --- | --- |
| 1 | (1 раз) |
| 222 | (3 раза) |
| 33333 | (5 раз) |
| … | (n раз) |
| 33333 | (5 раз) |
| 222 | (3 раза) |
| 1 | (1 раз) |

* 1. Разработать *рекурсивный метод* для вывода на экран следующей картинки:

|  |  |
| --- | --- |
| AAAAAAAAAA…AAAAAAAAAA | (80 раз) |
| BBBBBBBBB…BBBBBBBB | (78 раз) |
| СССССССС …СССССССС | (76 раз) |
| … | … |
| YYY…YYY | (32 раза) |
| ZZ...ZZ | (30 раз) |
| YYY…YYY | (32 раза) |
| … | … |
| СССССССС …СССССССС | (76 раз) |
| BBBBBBBBB…BBBBBBBB | (78 раз) |
| AAAAAAAAAA…AAAAAAAAAA | (80 раз) |

## Контрольные вопросы.

1. Задана программа

class Program

{

//нерекурсивный метод------------------------------

static long Sum(long a)

{

long sum=0;

while (a>0) //пока a больше нуля

{

sum+=a%10; //добавляем к сумме последнюю цифру числа а

a/=10; //отбрасываем от числа а последнюю цифру

}

return sum; //возвращаем в качестве результата сумму цифр числа a

}

//рекурсивный метод----------------------------------

static long SumR(long a) //рекурсивный метод

{

if (a==0) //если a =0, то

return 0; // возвращаем 0

else return SumR(a/10)+ a%10; //иначе обращаемся к рекуррентному

// соотношению

}

static void Main()

{

Console.Write("n=");

long n=long.Parse(Console.ReadLine());

Console.WriteLine("Нерекурсивный метод: "+Sum(n));

Console.WriteLine("Рекурсивный метод: "+SumR(n));

}

}

}

Изменить методы так, чтобы на экран выводилось количество цифр в числе n.

1. Задана программа

class Program

{

static int Fb(int n) //нерекурсивный алгоритм

{

int a, a1=1, a2=1;

if (n==1||n==2) return 1;

else

{

for (int i=2; i<=n; ++i)

{

a=a1+a2;

a1=a2;

a2=a;

}

return a1;

}

}

static int FbR(int n) //рекурсивный алгоритм

{

if (n==1 || n==2 )return 1;

else return FbR(n-1)+FbR(n-2);

}

static void Main()

{

Console.Write("n=");

int n=int.Parse(Console.ReadLine());

Console.WriteLine("Нерекурсивный метод: "+Fb(n));

Console.WriteLine("Рекурсивный метод: "+FbR(n));

}

}

Изменить методы так, чтобы на экран выводилась сумма n элементов последовательности Фибоначчи.

1. Для заданного значения n вывести на экран n строк, в каждой из которых содержится n звездочек. Например, для n=5 на экран нужно вывести следующую таблицу:

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

1. Для заданного значения n (например для n=7 ) вывести на экран следующую таблицу:

\* \* \* \* \* \* \*

\* \* \* \* \*

\* \* \*

\*

\*

\* \* \*

\* \* \* \* \*

\* \* \* \* \* \* \*

1. Для заданного значения n (в нашем случае для n=7 ) разработать рекурсивный метод, чтобы на экран выводилась следующая таблица:

\*

\* \* \*

\* \* \* \* \*

\* \* \* \* \* \* \*

\* \* \* \* \* \* \*

\* \* \* \* \*

\* \* \*

\*

1. Для заданного значения N  разработать рекурсивный метод, чтобы на экран выводились все простые числа меньшие N.

# Обработка исключений

## Теоретические сведения.

Язык С#, как и многие другие объектно-ориентированные языки, реагирует на ошибки и ненормальные ситуации с помощью механизма обработки *исключений*. *Исключение* - это *объект*, генерирующий информацию о "необычном программном происшествии". При этом важно проводить различие между ошибкой в программе, ошибочной ситуацией и исключительной ситуацией.

*Ошибка в программе*допускается программистом при ее разработке. Например, вместо *операции* сравнения ( == ) используется *операция присваивания* ( = ). Программист должен исправить подобные ошибки до передачи кода программы заказчику. Использование механизма обработки исключений не является защитой от ошибок в программе.

*Ошибочная ситуация*вызвана действиями пользователя. Например, *пользователь* вместо числа ввел строку. Такая ошибка способна вызывать *исключение*. Программист должен предвидеть ошибочные ситуации и предотвращать их с помощью операторов, проверяющих допустимость поступающих данных.

Даже если программист исправил все свои ошибки в программе, предвидел все ошибочные ситуации, он все равно может столкнуться с непредсказуемыми и неотвратимыми проблемами - *исключительными ситуациями*. Например, нехваткой доступной памяти или попыткой открыть несуществующий *файл*. Исключительные ситуации программист предвидеть не может, но он может отреагировать на них так, что они не приведут к краху программы.

Для обработки ошибочных и исключительных ситуаций в С# используется специальная подсистема обработки исключений. Преимущество данной подсистемы состоит в автоматизации создания большей части кода по обработке исключений. Раньше этот код приходилось вводить в программу "вручную". Кроме этого обработчик исключений способен распознавать и выдавать информацию о таких стандартных исключениях, как *деление* на нуль или попадание вне диапазона определения индекса.

### Оператор try

В С# исключения представляются классами. Все *классы исключений* порождены от *встроенного класса* исключений Exception, который определен в пространстве имен System.

Управление обработкой исключений основывается на использовании оператора try. Синтаксис оператора:

try // контролируемый блок

{

…

}

catch //один или несколько блоков обработки исключений

{

…

}

finally //блок завершения

{

…

}

Программные инструкции, которые нужно проконтролировать на предмет исключений, помещаются в блок try. Если исключение возникает в этом блоке, оно дает знать о себе *выбросом*определенного рода информации. *Выброшенная информация*может быть перехвачена и обработана соответствующим образом с помощью блока catch. Любой код, который должен быть обязательно выполнен при выходе из блока try, помещается в блок finally. Рассмотрим пример, демонстрирующий, как отследить и перехватить исключение.

static void Main()

{

int x = int.Parse(Console.ReadLine());

int y =1 / x;

Console.WriteLine(y);

}

Перечислим, какие исключительные ситуации могут возникнуть:

1. пользователь может ввести нечисловое значение
2. если ввести значение 0, то произойдет деление на 0.

Теперь попробуем обработать эти ситуации. Для этого изменим код следующим образом.

static void Main()

{

try

{

int x = int.Parse(Console.ReadLine());

int y =1 / x;

Console.WriteLine("y={0}", y);

Console.WriteLine("блок try выполнился успешно");

}

catch // \*

{

Console.WriteLine("возникла какая-то ошибка");

}

Console.WriteLine("конец программы");

}

Рассмотрим, как обрабатываются исключения в данном примере. Когда возникает исключение, выполнение программы останавливается и управление передается блоку catch. Этот блок *никогда*не возвращает управление в то место программы, где возникло исключение. Поэтому команды из блока try, расположенные ниже строки, в которой возникло исключение, никогда не будут выполнены. Блок catch обрабатывает исключение, и выполнение программы продолжается с оператора, следующего за этим блоком.

В нашем случае при вводе нечислового значения или 0 будет выведено сообщение "возникла ошибка", а затем сообщение "конец программы".

Обработчик исключений позволяет не только отловить ошибку, но и вывести полную информацию о ней. Для демонстрации сказанного заменим блок catch следующим фрагментом.

catch (Exception error)

{

Console.WriteLine("Возникла ошибка {0}", error);

}

Теперь, если возникнет исключительная ситуация, "выброшенная" информация будет записана в идентификатор error. Данную информацию можно просмотреть с помощью метода WriteLine. Такое сообщение очень полное и будет полезно только разработчику на этапе отладки проекта.

Для пользователя на этапе эксплуатации приложения достаточно более краткой информации о типе ошибке. С этой целью в С# выделены стандартные *классы исключений*, такие как DivideByZeroException, FormatException. Внесем изменения в программу.

static void Main()

{

try

{

int x = int.Parse(Console.ReadLine()); // 1 ситуация

int y =1 / x; // 2 ситуация

Console.WriteLine("y={0}", y);

Console.WriteLine("блок try выполнился успешно");

}

catch(FormatException) // обработка 1 ситуации

{

Console.WriteLine("Ошибка: введено нечисловое значение!");

}

catch (DivideByZeroException) // обработка 2 ситуации

{

Console.WriteLine("Ошибка: деление на 0!");

}

Console.WriteLine("конец программы");

}

В данном примере обрабатывается каждая ситуация в отдельности, при этом пользователю сообщается лишь минимальная информация об ошибке. В следующей таблице содержится описание наиболее часто используемых обработчиков стандартных исключений.

|  |  |
| --- | --- |
| Имя | Описание |
| ArithmeticException | Ошибка в арифметических операциях или преобразованиях |
| ArrayTypeMismatchException | Попытка сохранения в массиве элемента несовместимого типа |
| DivideByZeroException | Попытка деления на ноль |
| FormatException | Попытка передать в метод аргумент неверного формата |
| IndexOutOfRangeException | Индекс массива выходит за границу диапазона |
| InvalidCastException | Ошибка преобразования типа |
| OutOfMemoryException | Недостаточно памяти для нового объекта |
| OverflowException | Переполнение при выполнении арифметических операций |
| StackOverflowException | Переполнение стека |

Одно из основных достоинств обработки исключений состоит в том, что она позволяет программе отреагировать на ошибку и продолжить выполнение. Рассмотрим программу, которая строит таблицу значений для функции вида y(x)=100/(x2-1).

static void Main()

{

Console.WriteLine("a=");

int a = int.Parse( Console.ReadLine());

Console.WriteLine("b=");

int b = int.Parse(Console.ReadLine());

for (int i = a; i <= b; ++i)

{

try

{

Console.WriteLine("y({0})={1}", i, 100 / (i \* i - 1));

}

catch (DivideByZeroException)

{

Console.WriteLine("y({0})=Деление на 0", i);

}

}

}

Если встречается деление на нуль, генерируется исключение типа DivideByZeroException. В программе это исключение обрабатывается выдачей сообщения об ошибке, после чего выполнение программы продолжается. При этом попытка разделить на нуль не вызывает внезапную динамическую ошибку (т.к. блок обработки прерываний помещен внутрь цикла for ). Вместо этого исключение позволяет красиво выйти из ошибочной ситуации и продолжить выполнение программы.

### Операторы checked и unchecked

В С# предусмотрено специальное средство, которое связано с генерированием исключений, вызванных переполнением результата в арифметических вычислениях. Например, когда значение арифметического выражения выходит за пределы диапазона, определенного для типа данных выражения. Рассмотрим небольшой фрагмент программы:

static void Main()

{

byte x = 200; byte y = 200;

byte result = (byte) (x + y);

Console.WriteLine(result);

}

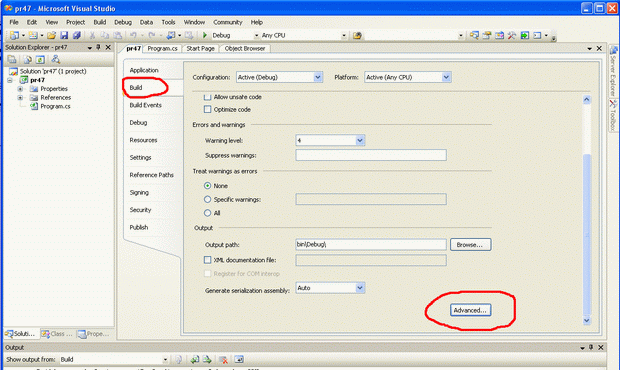
Здесь сумма значений а и b превышает диапазон представления значений типа byte. Следовательно, результат данного выражения не может быть записан в переменную result, тип которой byte.

Для управления подобными исключениями в С# используются операторы checked и *unchecked*. Чтобы указать, что некоторое выражение должно быть проконтролировано на предмет переполнения, используйте ключевое слово checked. А чтобы проигнорировать переполнение, используйте ключевое слово *unchecked*. В последнем случае результат будет усечен так, чтобы его тип соответствовал типу-результату выражения.

По умолчанию проверка переполнения отключена (галочка не стоит). В результате код выполняется быстро, но тогда программист должен быть уверен, что переполнения не случится или предусмотреть его возникновение. Как мы уже упоминали, можно включить проверку переполнения для всего проекта, однако она не всегда нужна. С помощью использования операторов checked и *unchecked* в С# реализуется механизм гибкого управления проверкой

Можно задать (или отключить) проверку переполнения сразу для всего проекта. Для этого необходимо выполнить следующие действия

1. Щелкнуть правой кнопкой мыши на имени проекта
2. В выпадающем меню выбрать Properties
3. В появившемся окне (см. рис.) выбрать слева страницу Build
4. Щелкнуть на кнопке Advanced
5. В появившемся окошке поставить или убрать галочку напротив Check for *arithmetic overflow*/*underflow* property.

[](https://www.intuit.ru/EDI/01_05_17_1/1493590891-23956/tutorial/509/objects/7/files/07_01.gif)

Оператор checked имеет две формы:

1. проверяет конкретное выражение и называется *операторной checked-формой*

checked ((тип-выражения) expr)

где expr — выражение, значение которого необходимо контролировать. Если значение контролируемого выражения переполнилось, генерируется исключение типа OverflowException.

1. проверяет блок инструкций

checked

**{**

**// Инструкции, подлежащие проверке.**

**}**

Оператор *unchecked* также имеет две формы:

1. операторная форма, которая позволяет игнорировать переполнение для заданного выражения

unchecked ((тип-выражения) expr)

где ехрr — выражение, которое не проверяется на предмет переполнения. В случае переполнения это выражение усекается.

1. игнорирует переполнение, которое возможно в блоке инструкций

unchecked

{

// Инструкции, для которых переполнение игнорируется.

}

Рассмотрим пример программы, которая демонстрирует использование checked и *unchecked*.

static void Main()

{

byte x = 200; byte y = 200;

try

{

byte result = unchecked((byte)(x + y));

Console.WriteLine("1: {0}", result);

result = checked((byte)(x + y));

Console.WriteLine("2: ", result);

}

catch (OverflowException)

{

Console.WriteLine("возникло переполнение");

}

}

Результат выполнения программы:

1: 144

возникло переполнение

В данном примере показано, как использовать checked и uncheсked для проверки выражения. Пример, как использовать их для контроля за блоком инструкций.

static void Main()

{

byte n = 1; byte i;

try

{

unchecked //блок без проверки

{

for (i = 1; i < 10; i++) n \*= i;

Console.WriteLine("1: {0}", n);

}

checked //блок с проверкой

{

n=1;

for (i = 1; i < 10; i++) n \*= i;

Console.WriteLine("2: ", n);

}

}

catch (OverflowException)

{

Console.WriteLine("возникло переполнение");

}

}

Результат выполнения программы:

1: 128

возникло переполнение

### Генерация собственных исключений

До сих пор рассматривали исключения, которые генерирует среда, но сгенерировать исключение может и сам программист. Для этого необходимо воспользоваться оператором throw, указав параметры, определяющие вид исключения. Параметром должен быть объект, порожденный от стандартного класса System.Exception. Этот объект используется для передачи информации об исключении обработчику.

static void Main()

{

try

{

int x = int.Parse(Console.ReadLine());

if (x < 0) throw new Exception(); //1

Console.WriteLine("ok");

}

catch

{

Console.WriteLine("введено недопустимое значение");

}

}

В строчке 1 c помощью команды new был создан объект исключения типа Exception. При необходимости можно генерировать исключение любого типа.

При генерации исключения можно определить сообщение, которое будет "выбрасываться" обработчиком исключений. Например:

static void Main()

{

try

{

int x = int.Parse(Console.ReadLine());

if (x < 0) throw new Exception("введено недопустимое значение"); //1

Console.WriteLine("ok");

}

catch (Exception error)

{

Console.WriteLine(error.Message);

}

}

Рассмотрим несколько приемов использования обработчиков исключений.

**Пример 1**. Один try -блок можно вложить в другой. Исключение, сгенерированное во внутреннем try -блоке и не перехваченное catch -инструкцией, которая связана с этим try -блоком, передается во внешний try -блок. Например, в следующей программе исключение типа ArithmeticException перехватывается не внутренним try -блоком, а внешним.

static void Main()

{

Console.WriteLine("a=");

byte a = byte.Parse(Console.ReadLine());

Console.WriteLine("b=");

byte b = byte.Parse(Console.ReadLine());

int f=1;

try //Внешний блок-try

{

for (byte i = a; i <= b; ++i)

{

try //Внутренний блок-try

{

f=checked((int)(f\*i));

Console.WriteLine("y({0})={1:f6}", i, 100 / (f - 1));

}

catch (DivideByZeroException)

{

Console.WriteLine("y({0})=Деление на 0", i);

}

}

}

catch (ArithmeticException)

{

Console.WriteLine("ERROR");

}

}

Использование вложенных try -блоков обусловлено желанием обрабатывать различные категории ошибок различными способами. Одни типы ошибок носят катастрофический характер и не подлежат исправлению. Другие — неопасны для дальнейшего функционирования программы, и с ними можно справиться прямо на месте их возникновения. Поэтому внешний try -блок можно использовать для перехвата самых серьезных ошибок, позволяя внутренним try -блокам обрабатывать менее опасные.

**Пример 2**. Исключение, перехваченное одной catch -инструкцией, можно сгенерировать повторно, чтобы обеспечить возможность его перехвата другой (внешней) catch -инструкцией. Это позволяет нескольким обработчикам получить доступ к исключению.

static void genException ()

{

Console.WriteLine("a=");

double a = double.Parse(Console.ReadLine());

Console.WriteLine("b=");

double b = double.Parse(Console.ReadLine());

int f = 1;

try //Внешний блок-try

{

for (double i = a; i <= b; ++i)

{

try //Внутренний блок-try

{

f = checked((int)(f \* i));

Console.WriteLine("y({0})={1:f6}", i, 100 / (f - 1));

}

catch (DivideByZeroException)

{

Console.WriteLine("y({0})=Деление на 0", i);

}

}

}

catch (ArithmeticException)

{

Console.WriteLine("ERROR");

throw ; //повторная генерация исключения

}

}

static void Main()

{

try

{

genException();

}

catch

{

Console.WriteLine("НЕИСПРАВИМАЯ ОШИБКА!!!");

}

}

Нужно помнить, что при повторном генерировании исключения оно не будет повторно перехватываться той же catch -инструкцией, а передается следующей (внешней) catch -инструкции.

**Пример 3**. Как упоминалось выше, *тип исключения* должен совпадать с типом, заданным в catch -инструкции. В противном случае это исключение не будет перехвачено. Можно перехватывать все исключения, используя catch -инструкцию без параметров. Кроме того, с try -блоком можно связать не одну, а несколько catch -инструкций. В этом случае все catch-инструкции должны перехватывать исключения различного типа. Если вы все же не уверены, что предусмотрели все ситуации, то последней можно добавить catch -инструкцию без параметров.

Иногда возникает потребность в обязательном выполнении каких-то действий, которые должны выполниться по выходу из try/catch -блока. Например, генерируется исключение и происходит преждевременное завершение выполнения программного фрагмента, но при этом остается открытым файл. Для выхода из такой ситуации С# предоставляет блок finally, который добавляется после всех блоков catch.

static void Main()

{

for (int i = 0; i < 5; i++)

{

try

{

Console.WriteLine("Введите два числа");

int a = int.Parse(Console.ReadLine());

int b = int.Parse(Console.ReadLine());

Console.WriteLine(a+"/"+b+"="+a/b);

}

catch (FormatException)

{

Console.WriteLine("Нужно ввести число!");

}

catch (DivideByZeroException)

{

Console.WriteLine("Делить на нуль нельзя!");

}

catch

{

Console.WriteLine("Какая-то ошибка");

}

finally

{

Console.WriteLine("после try-блока");

}

}

}

## Задания на лабораторную работу.

Постройте таблицу значений функции y=f(x) для х \in [a, b] с шагом h. Если в некоторой точке x *функция* не определена, то выведите на экран сообщение об этом.

При решении данной задачи использовать вспомогательный метод f(x), реализующий заданную функцию, а также проводить обработку возможных исключений.

Пример:

y=\cfrac{1}{(1+x)^2}

using System;

namespace Hello

{

class Program

{

static double f(double x)

{

try

{

//если х не попадает в область определения, то генерируется исключение

if (x == -1) throw new Exception();

else return 1 / Math.Pow(1 + x, 2);

}

catch

{

throw;

}

}

static void Main(string[] args)

{

try

{

Console.Write("a=");

double a = double.Parse(Console.ReadLine());

Console.Write("b=");

double b = double.Parse(Console.ReadLine());

Console.Write("h=");

double h = double.Parse(Console.ReadLine());

for (double i = a; i <= b; i += h)

try

{

Console.WriteLine("y({0})={1:f4}", i, f(i));

}

catch

{

Console.WriteLine("y({0})=error", i);

}

}

catch (FormatException)

{

Console.WriteLine("Неверный формат ввода данных");

}

catch

{

Console.WriteLine("Неизвестная ошибка");

}

}

}

}

|  |  |  |
| --- | --- | --- |
| 2. y=\cfrac{1}{x^2-1} ; | 3. y=\sqrt{x^2-1} ; |  |
| 4. y=\sqrt{5-x^3} ; | 5. y=\ln{(x-1)} ; | 6. y=\ln{(4-x^2)}  ; |
| 7. y=\cfrac{x}{\sqrt{2x-1}} ; | 8. y=\cfrac{3x+4}{\sqrt{x^2+2x+1}} ; | 9. y=\cfrac{1}{x-1}+\cfrac{2}{1-4x} ; |
| 10. y=\ln{|x-2|} ; | 11. y=\ln{\cfrac{x}{x-2}} ; | 12. y=\ln{(x^4-1)}\ln{(1+x)} ; |
| 13. y=\cfrac{\ln{(x-2)}}{\sqrt{5x+1}} ; | 14. y=\cfrac{\sqrt{x^2-2x+1}}{\ln{(4-2x)}} ; | 15. y=\ln{|3x|}\sqrt{2x^5-1} ; |
| 16. y=\cfrac{3}{|x^3+8|} ; | 17. y=\cfrac{x+4}{x^2-2}+\sqrt{x^3-1} ; | 18. y=\sqrt{x^2+1}+\sqrt{x^2+5} ; |
| 19. y=\cfrac{\sqrt{x^3-1}}{\sqrt{x^2-1}} ; | 20. y=\cfrac{1}{x+7}+\ln{(1-|x|)} ; |  |

## Контрольные вопросы.

1. Рассмотрите программу из Примера 2. Объясните почему не было сгенерировано исключение DivideByZeroException.
2. Рассмотрите программу из Примера 2. Сгенерируйте собственное исключение DivideByZeroException и его обработку для ситуации f-1<0.000001.
3. Рассмотрите программу из Примера 3. Протестируйте данную программу, вводя поочередно следующие значения:

a=4, b=2

a=3, b=g

a=d, b=1

a=2, b=0,

a=123456789987654321, b=1

## 

# Массивы

## Теоретические сведения.

*Массив* - набор элементов одного и того же типа, объединенных общим именем. Массивы в С# можно использовать по аналогии с тем, как они используются в других языках программирования. Однако С#-массивы имеют существенные отличия: они относятся к *ссылочным типам данных*, более того - реализованы как объекты. Фактически имя массива является ссылкой на область кучи (динамической памяти), в которой последовательно размещается набор элементов определенного типа. Выделение памяти под элементы происходит на этапе инициализации массива. А за освобождением памяти следит система сборки мусора - неиспользуемые массивы автоматически утилизируются данной системой.

### Одномерные массивы

Одномерный массив - это фиксированное количество элементов одного и того же типа, объединенных общим именем, где каждый элемент имеет свой номер. Нумерация элементов массива в С# начинается с нуля, то есть, если массив состоит из 10 элементов, то его элементы будут иметь следующие номера: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Одномерный массив в С# реализуется как объект, поэтому его создание представляет собой двухступенчатый процесс. Сначала объявляется ссылочная переменная на массив, затем выделяется память под требуемое количество элементов базового типа, и ссылочной переменной присваивается адрес нулевого элемента в массиве. Базовый тип определяет тип данных каждого элемента массива. Количество элементов, которые будут храниться в массиве, определяется размер массива.

В общем случае процесс объявления переменной типа массив, и выделение необходимого объема памяти может быть разделено. Кроме того на этапе объявления массива можно произвести его инициализацию. Поэтому для объявления одномерного массива может использоваться одна из следующих форм записи:

|  |  |
| --- | --- |
| Форма записи | Пояснения |
| базовый\_тип [] имя\_\_массива;  Например:  int [] a; | Описана ссылка на одномерный массив, которая в дальнейшем может быть использована:  для адресации на уже существующий массив;  передачи массива в метод в качестве параметра  отсроченного выделения памяти под элементы массива. |
| базовый\_тип [] имя\_\_массива = new базовый\_тип [размер];  Например:  int []a=new int [10]; | Объявлен одномерный массив заданного типа и выделена память под одномерный массив указанной размерности. Адрес данной области памяти записан в ссылочную переменную. Элементы массива равны нулю.  **Замечание**. Надо отметить, что в C# элементам массива присваиваются начальные значения по умолчанию в зависимости от базового типа. Для арифметических типов - нули, для ссылочных типов - null, для символов - пробел. |
| базовый\_тип [] имя\_\_массива={список инициализации};  Например:  int []a={0, 1, 2, 3}; | Выделена память под одномерный массив, размерность которого соответствует количеству элементов в списке инициализации. Адрес этой области памяти записан в ссылочную переменную. Значение элементов массива соответствует списку инициализации. |

Обращения к элементам массива происходит с помощью индекса, для этого нужно указать имя массива и в квадратных скобках его номер. Например, a[0], b[10], c[i].

Так как массив представляет собой набор элементов, объединенных общим именем, то обработка массива обычно производится в цикле. Рассмотрим несколько простых примеров работы с одномерными массивами.

Пример 1.

static void Main()

{

int[] myArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int i;

for (i = 0; i < 10; ++i)

Console.WriteLine(myArray[i]);

}

Пример 2.

static void Main()

{

int[] myArray = new int[10];

int i;

for (i = 0; i < 10; i++)

myArray[i] = i \* i;

for (i = 0; i < 10; i++)

Console.WriteLine(myArray[i]);

}

Хотя при инициализации массива нет необходимости использовать операцию new, все же массив можно инициализировать следующим образом:

int [ ] myArray = new int [ ] { 99, 10, 100, 18, 78, 23, 163, 9, 87, 49 };

Несмотря на избыточность, данная форма инициализации массива может оказаться полезной в том случае, когда уже существующей ссылке на одномерный массив присваивается ссылка на новый массив. Например:

static void Main()

{

int[] myArray = { 0, 1, 2, 3, 4, 5};

int i;

for (i = 0; i < 6; i++)

Console.Write(" "+myArray[i]);

Console.WriteLine("\nНовый массив: ");

myArray = new int[] { 99, 10, 100, 18, 78, 23, 163, 9, 87, 49 }; // 1

for (i = 0; i < 10; i++)

Console.Write(" " + myArray[i]);

}

Следует отметить, что первоначально переменная myArray ссылалась на 6-ти элементный массив. В строке 1 переменной myArrayбыла присвоена ссылка на новый 10-элементный массив, в результате чего исходный массив оказался неиспользуемым, т.к. на него теперь не ссылается ни один объект. Поэтому он автоматически будет удален сборщиком мусора.

### Массивы и исключения

Выход за границы массива в C# расценивается как ошибка, в ответ на которую генерируется исключение - IndexOutOfRangeException.

Рассмотрим следующий пример:

static void Main()

{

int[] myArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int i;

try

{

for (i = 0; i <= 10; i++) Console.WriteLine(myArray[i]);

}

catch (IndexOutOfRangeException)

{

Console.WriteLine("Exception: Выход за границу диапазона");

}

}

### Массив как параметр

Так как имя массива фактически является ссылкой, то он передается в метод по ссылке и, следовательно, все изменения элементов массива, являющегося формальным параметром, отразятся на элементах соответствующего массива, являющимся фактическим параметром.

Рассмотрим пример передачи массива как параметра:

class Program

{

static void Print(int n, int[] a) //n - размерность массива, а - ссылка на массив

{

for (int i = 0; i < n; i++) Console.Write("{0} ", a[i]);

Console.WriteLine();

}

static void Change(int n, int[] a)

{

for (int i = 0; i < n; i++)

if (a[i] > 0) a[i] = 0; // изменяются элементы массива

}

static void Main()

{

int[] myArray = { 0, -1, -2, 3, 4, 5, -6, -7, 8, -9 };

Print(10, myArray);

Change(10, myArray);

Print(10, myArray);

}

}

### Массив как объект

Массивы в С# реализованы как объекты. Если говорить более точно, то они реализованы на основе базового класса Array, определенного в пространстве имен System. Данный класс содержит различные свойства и методы. Например, свойство Length позволяет определять количество элементов в массиве. Преобразуем предыдущий пример:

class Program

{

static void Print(int[] a) // передаем только ссылку на массив

{

for (int i = 0; i < a.Length; i++) Console.Write("{0} ", a[i]);

Console.WriteLine();

}

static void Change(int[] a)

{

for (int i = 0; i < a.Length; i++)

if (a[i] > 0) a[i] = 0;

}

static void Main()

{

int[] myArray = { 0, -1, -2, 3, 4, 5, -6, -7, 8, -9 };

Print(myArray);

Change(myArray);

Print(myArray);

}

}

Другие свойства и методы класса Array приведены в следующей таблице:

|  |  |  |
| --- | --- | --- |
| Элемент | Вид | Описание |
| Length | свойство | Количество элементов массива (по всем размерностям) |
| *BinarySearch* | статический метод | Двоичный поиск в отсортированном массиве |
| Clear | статический метод | Присваивание элементам массива значений по умолчанию |
| Copy | статический метод | Копирование заданного диапазона элементов одного массива в другой |
| CopyTo | *экземплярный метод* | Копирование всех элементов текущего одномерного массива в другой массив |
| GetValue | *экземплярный метод* | Получение значения элемента массива |
| IndexOf | статический метод | Поиск первого вхождения элемента в одномерный массив |
| *LastIndexOf* | статический метод | Поиск последнего вхождения элемента в одномерный массив |
| Reverse | статический метод | Изменение порядка следования элементов на обратный |
| SetValue | *экземплярный метод* | Установка значения элемента массива |
| Sort | статический метод | Упорядочивание элементов одномерного массива |

Вызов статических методов происходит через обращение к имени класса, например, Array.Sort(myArray). В данном случае мы обращаемся к статическому методу Sort класса Array и передаем данному методу в качестве параметра объект myArray - экземпляр класса Array.

Обращение к свойству или вызов *экземплярного метода* производится через обращение к экземпляру класса, например, myArray.Length или myArray.GetValue(i).

Пример:

class Program

{

static void Main()

{

try

{

int[] MyArray;

Console.Write("Введите размерность массива: ");

int n = int.Parse(Console.ReadLine());

MyArray = new int[n];

for (int i = 0; i < MyArray.Length; ++i)

{

Console.Write("a[{0}]=",i);

MyArray[i] = int.Parse(Console.ReadLine());

}

PrintArray("исходный массив:", MyArray);

Array.Sort(MyArray);

PrintArray("массив отсортирован по возрастанию", MyArray);

Array.Reverse(MyArray);

PrintArray("массив отсортирован по убыванию", MyArray);

}

catch (FormatException)

{

Console.WriteLine("неверный формат ввода данных");

}

catch (OverflowException)

{

Console.WriteLine("переполнение");

}

catch (OutOfMemoryException)

{

Console.WriteLine("недостаточно памяти для создания нового объекта");

}

}

static void PrintArray(string a, int[] mas)

{

Console.WriteLine(a);

for (int i = 0; i < mas.Length; i++) Console.Write("{0} ", mas[i]);

Console.WriteLine();

}

}

}

### Многомерные массивы

Многомерные массивы имеют более одного измерения. Чаще всего используются двумерные массивы, которые представляют собой таблицы. Каждый элемент массива имеет два индекса, первый определяет номер строки, второй - номер столбца, на пересечении которых находится элемент. Нумерация строк и столбцов начинается с нуля.

Объявить двумерный массив можно одним из предложенных способов:

тип [,] имя\_\_массива;

тип [,] имя\_\_массива = new тип [размер1, размер2];

тип [,] имя\_\_массива={{элементы 1-ой строки}, … , {элементы n-ой строки}};

тип [,] имя\_\_массива= new тип [,]{{элементы 1-ой строки}, … ,{элементы n-ой строки}};

строки}};

Например:

int [,] a;

int [,] a= new int [3, 4];

int [,] a={{0, 1, 2}, {3, 4, 5}};

int [,] a= new int [,]{{0, 1, 2}, {3, 4, 5}};

Замечания.

1. *Последние два описания являются избыточными.*
2. *При обращении к свойству Length для двумерного массива мы получим общее количество элементов в массиве. Чтобы получить количество строк нужно обратиться к методу GetLength с параметром 0. Чтобы получить количество столбцов - к методу GetLength с параметром 1.*

Пример:

class Program

{

static void PrintArray(string a, int[,] mas)

{

Console.WriteLine(a);

for (int i = 0; i < mas.GetLength(0); i++)

{

for (int j = 0; j < mas.GetLength(1); j++)

Console.Write("{0} ", mas[i, j]);

Console.WriteLine();

}

}

static void Change(int[,] mas)

{

for (int i = 0; i < mas.GetLength(0); i++)

for (int j = 0; j < mas.GetLength(1); j++)

if (mas[i, j] % 2 == 0) mas[i, j] = 0;

}

static void Main()

{

try

{

int[,] MyArray ={ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

PrintArray("исходный массив:", MyArray);

Change(MyArray);

PrintArray("итоговый массив", MyArray);

}

catch (FormatException)

{

Console.WriteLine("неверный формат ввода данных");

}

catch (OverflowException)

{

Console.WriteLine("переполнение");

}

catch (OutOfMemoryException)

{

Console.WriteLine("недостаточно памяти для создания нового объекта");

}

}

}

### Ступенчатые массивы

В ступенчатых массивах количество элементов в разных строках может быть различным. В памяти ступенчатый массив хранится в виде массива массивов. Структура ступенчатого массива:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Массив | a |  | a[0] |  | a[0][0] | a[0][1] | … |  |  |  |
|  |  |  | a[1] |  |  |  |  |  |  |  |
|  |  |  | … |  | a[1][0] | a[1][1] | … |  |  |  | |
|  |  |  | a[n] |  |  |  |  |  |  |  |  |
|  |  |  |  |  | a[n][0] | a[n][1] | … |  |  |  |  |

#### Объявление ступенчатого массива:

тип [][] имя\_массива;

Например:

int [][]a;

Фактически мы объявили одномерный массив ссылок на целочисленные одномерные массивы. При таком описании потребуется не только выделять память под одномерный массив ссылок, но и под каждый из целочисленных одномерных массивов. Такое распределение памяти позволяет определять произвольную длину каждой строки массива (отсюда и произошло название массива - ступенчатый). Например:

int [][] a= new int [3][]; // Создаем три строки

a[0]=new int [2]; // 0-ая строка ссылается на 2-х элементый одномерный массив

a[1]=new int [3]; // 1-ая строка ссылается на 3-х элементый одномерный массив

a[2]=new int [10]; // 2-ая строка ссылается на 10-ти элементый одномерный массив

Другой способ выделения памяти:

int [][] a= {new int [2], new int [3], new int [10]};

Так как каждая строка ступенчатого массива фактически является одномерным массивом, то с каждой строкой можно работать как с экземпляром класса Array. Это является преимуществом ступенчатых массивов перед двумерными массивами.

Пример:

class Program

{

static void Main()

{

try

{

int[][] MyArray;

Console.Write("Ввведите количество строк: ");

int n = int.Parse(Console.ReadLine());

MyArray = new int[n][];

for (int i = 0; i < MyArray.Length; i++)

{

Console.Write("введите количество элементов в {0} строке: ", i);

int j = int.Parse(Console.ReadLine());

MyArray[i] = new int[j];

for (j = 0; j < MyArray[i].Length; j++)

{

Console.Write("a[{0}][{1}]= ", i, j);

MyArray[i][j] = int.Parse(Console.ReadLine());

}

}

PrintArray("исходный массив:", MyArray);

for (int i = 0; i < MyArray.Length; i++) Array.Sort(MyArray[i]);

PrintArray("измененный массив", MyArray);

}

catch (FormatException)

{

Console.WriteLine("неверный формат ввода данных");

}

catch (OverflowException)

{

Console.WriteLine("переполнение");

}

catch (OutOfMemoryException)

{

Console.WriteLine("недостаточно памяти для создания нового объекта");

}

}

static void PrintArray(string a, int[][] mas)

{

Console.WriteLine(a);

for (int i = 0; i < mas.Length; i++)

{

for (int j = 0; j < mas[i].Length; j++) Console.Write("{0} ", mas[i][j]);

Console.WriteLine();

}

}

}

### Оператор foreach и его использование при работе с массивами

Оператор foreach применяется для перебора элементов в специальным образом организованной группе данных, в том числе и в массиве. Удобство этого вида цикла заключается в том, что нам не требуется определять количество элементов в группе и выполнять перебор по индексу - мы просто указываем на необходимость перебрать все элементы группы. Синтаксис оператора:

foreach (<тип> <имя> in <группа>) <тело цикла>

где *имя* определяет локальную по отношению к циклу переменную, которая будет по очереди принимать все значения из указанной *группы*, а *тип* соответствует базовому типу элементов *группы*.

Ограничением оператора foreach является то, что с его помощью можно только просматривать значения элементов в группе данных, но нельзя их изменять.

Рассмотрим несколько примеров использования оператора foreach:

1. для работы с одномерными массивами:

static void PrintArray(string a, int [] mas)

{

Console.WriteLine(a);

foreach (int x in mas)Console.Write("{0} ", x);

Console.WriteLine();

}

1. для работы с двумерными массивами:

static int Sum (int [,] mas)

{

int s=0;

foreach (int x in mas) s += x;

return s;

}

1. для работы со ступенчатыми массивами:

static void PrintArray3(string a, int[][] mas)

{

Console.WriteLine(a);

for (int i = 0; i < mas.Length; i++)

{

foreach (int x in mas[i]) Console.Write("{0} ", x);

Console.WriteLine();

}

}

## Задания на лабораторную работу.

1. Дана последовательность целых чисел.

Задачи из данного пункта решить двумя способами, используя одномерный массив, а затем двумерный. Размерность массива вводится с клавиатуры.

Заменить все положительные элементы противоположными им числами.

Пример 1: для одномерного массива

using System;

namespace ConsoleApplication2

{

class Class

{

static int [] Input ()

{

Console.WriteLine("введите размерность массива");

int n=int.Parse(Console.ReadLine());

int []a=new int[n];

for (int i = 0; i < n; ++i)

{

Console.Write("a[{0}]= ", i);

a[i]=int.Parse(Console.ReadLine());

}

return a;

}

static void Print(int[] a)

{

for (int i = 0; i < a.Length; ++i) Console.Write("{0} ", a[i]);

Console.WriteLine();

}

static void Change(int[] a)

{

for (int i = 0; i < a.Length; ++i)

if (a[i] > 0) a[i] = -a[i];

}

static void Main()

{

int[] myArray=Input();

Console.WriteLine("Исходный массив:");

Print(myArray);

Change(myArray);

Console.WriteLine("Измененный массив:");

Print(myArray);

}

}

}

Пример 2: для двумерного массива

using System;

namespace ConsoleApplication

{

class Class

{

static int [,] Input (out int n, out int m)

{

Console.WriteLine("введите размерность массива");

Console.Write("n = ");

n=int.Parse(Console.ReadLine());

Console.Write("m = ");

m=int.Parse(Console.ReadLine());

int [,]a=new int[n, m];

for (int i = 0; i < n; ++i)

for (int j = 0; j < m; ++j)

{

Console.Write("a[{0},{1}]= ", i, j);

a[i, j]=int.Parse(Console.ReadLine());

}

return a;

}

static void Print(int[,] a)

{

for (int i = 0; i < a.GetLength(0); ++i,Console.WriteLine() )

for (int j = 0; j < a.GetLength(1); ++j)

Console.Write("{0,5} ", a[i, j]);

}

static void Change(int[,] a)

{

for (int i = 0; i < a.GetLength(0); ++i)

for (int j = 0; j < a.GetLength(1); ++j)

if (a[i, j] > 0) a[i, j] = -a[i, j];

}

static void Main()

{

int n,m;

int[,] myArray=Input(out n, out m);

Console.WriteLine("Исходный массив:");

Print(myArray);

Change(myArray);

Console.WriteLine("Измененный массив:");

Print(myArray);

}

}

}

Заменить все элементы, меньшие заданного числа, этим числом.

* 1. Заменить все элементы, попадающие в интервал [a, b], нулем.
  2. Заменить все отрицательные элементы, не кратные 3, противоположными им числами.
  3. Все элементы, меньшие заданного числа, увеличить в два раза.
  4. Подсчитать среднее арифметическое элементов.
  5. Подсчитать среднее арифметическое отрицательных элементов.
  6. Подсчитать количество нечетных элементов.
  7. Подсчитать сумму элементов, попадающих в заданный интервал.
  8. Подсчитать сумму элементов, кратных 9.
  9. Подсчитать количество элементов, не попадающих в заданный интервал.
  10. Подсчитать сумму квадратов четных элементов.
  11. Вывести на экран номера всех элементов больших заданного числа.
  12. Вывести на экран номера всех нечетных элементов.
  13. Вывести на экран номера всех элементов, которые не делятся на 7.
  14. Вывести на экран номера всех элементов, не попадающих в заданный интервал.
  15. Определить, является ли произведение элементов трехзначным числом.
  16. Определить, является ли сумма элементов двухзначным числом.
  17. Вывести на экран элементы с четными индексами (для двумерного массива - сумма индексов должна быть четной).
  18. Вывести на экран положительные элементы с нечетными индексами (для двумерного массива - первый индекс должен быть нечетным).

1. Дана последовательность из n действительных чисел.

Задачи из данного пункта решить, используя одномерный массив.

Подсчитать количество максимальных элементов.

Пример.

using System;

namespace ConsoleApplication

{

class Class

{

static int [] Input ()

{

Console.WriteLine("введите размерность массива");

int n=int.Parse(Console.ReadLine());

int []a=new int[n];

for (int i = 0; i < n; ++i)

{

Console.Write("a[{0}]= ", i);

a[i]=int.Parse(Console.ReadLine());

}

return a;

}

static int Max(int[] a)

{

int max=a[0];

for (int i = 1; i < a.Length; ++i)

if (a[i] > max) max=a[i];

return max;

}

static void Main()

{

int[] myArray=Input();

int max=Max(myArray);

int kol=0;

for (int i=0; i<myArray.Length;++i)

if (myArray[i]==max)++kol;

Console.WriteLine("Количество максимальных элементов = "+kol);

}

}

}

Вывести на экран номера всех минимальных элементов.

* 1. Заменить все максимальные элементы нулями.
  2. Заменить все минимальные элементы на противоположные.
  3. Поменять местами максимальный элемент и первый.
  4. Вывести на экран номера всех элементов, не совпадающих с максимальным.
  5. Найти номер первого минимального элемента.
  6. Найти номер последнего максимального элемента.
  7. Подсчитать сумму элементов, расположенных между максимальным и минимальным элементами (минимальный и максимальный элементы в массиве единственные). Если максимальный элемент встречается позже минимального, то выдать сообщение об этом.
  8. Найти номер первого максимального элемента.
  9. Найти номер последнего минимального элемента.
  10. Подсчитать сумму элементов, расположенных между первым максимальным и последним минимальными элементами. Если максимальный элемент встречается позже минимального, то выдать сообщение об этом.
  11. Поменять местами первый минимальный и последний максимальный элементы.
  12. Найти максимум из отрицательных элементов.
  13. Найти минимум из положительных элементов.
  14. Найти максимум из модулей элементов.
  15. Найти количество пар соседних элементов, разность между которыми равна заданному числу.
  16. Подсчитать количество элементов, значения которых больше значения предыдущего элемента.
  17. Найти количество пар соседних элементов, в которых предыдущий элемент кратен последующему.
  18. Найти количество пар соседних элементов, в которых предыдущий элемент меньше последующего.

1. Дан массив размером nxn, элементы которого целые числа.

При решении задач из данного пункта использовать двумерный массив.

Подсчитать среднее арифметическое нечетных элементов, расположенных выше главной диагонали.

Пример.

using System;

namespace ConsoleApplication

{

class Class

{

static int [,] Input (out int n)

{

Console.WriteLine("введите размерность массива");

Console.Write("n = ");

n=int.Parse(Console.ReadLine());

int [,]a=new int[n, n];

for (int i = 0; i < n; ++i)

for (int j = 0; j < n; ++j)

{

Console.Write("a[{0},{1}]= ", i, j);

a[i, j]=int.Parse(Console.ReadLine());

}

return a;

}

static void Print(int[,] a)

{

for (int i = 0; i < a.GetLength(0); ++i,Console.WriteLine() )

for (int j = 0; j < a.GetLength(1); ++j)

Console.Write("{0,5} ", a[i, j]);

}

static double Rezalt(int[,] a)

{

int k=0;

double s=0;

for (int i = 0; i < a.GetLength(0); ++i)

for (int j = i+1; j < a.GetLength(1); ++j)

if (a[i, j] %2!= 0) {++k; s+=a[i, j];}

if (k!=0) return s/k;

else return 0;

}

static void Main()

{

int n;

int[,] myArray=Input(out n);

Console.WriteLine("Исходный массив:");

Print(myArray);

double rez=Rezalt(myArray);

Console.WriteLine("Среднее арифметическое ={0:f2}", rez);

}

}

}

* 1. Подсчитать среднее арифметическое четных элементов, расположенных ниже главной диагонали.
  2. Подсчитать сумму элементов, расположенных на побочной диагонали.
  3. Подсчитать среднее арифметическое ненулевых элементов, расположенных над *побочной диагональю*.
  4. Подсчитать среднее арифметическое элементов, расположенных под *побочной диагональю*.
  5. Поменять местами столбцы по правилу: первый с последним, второй с предпоследним и т.д.
  6. Поменять местами две средних строки, если количество строк четное, и первую со средней строкой, если количество строк нечетное.
  7. Поменять местами два средних столбца, если количество столбцов четное, и первый со средним столбцом, если количество столбцов нечетное.
  8. Если количество строк в массиве четное, то поменять строки местами по правилу: первую строку со второй, третью - с четвертой и т.д. Если количество строк в массиве нечетное, то оставить массив без изменений.
  9. Если количество столбцов в массиве четное, то поменять столбцы местами по правилу: первый столбец со вторым, третий - с четвертым и т.д. Если количество столбцов в массиве нечетное, то оставить массив без изменений.
  10. Вычислить А^{n}, где n - натуральное число.
  11. Подсчитать норму матрицы по формуле

||A||=\sum_{i}{\max_j{a_{i,j}}}.

* 1. Подсчитать норму матрицы по формуле

||A||=\sum_{j}^{}{\max_i{a_{i,j}}}.

* 1. Вывести элементы матрицы в следующем порядке:08_03
  2. Выяснить, является ли матрица симметричной относительно главной диагонали.
  3. Заполнить матрицу числами от 1 до n (где n=m \times k, а m  - количесто строк, а k  - количество столбцов прямоугольной матрицы) следующим образом:

08_04

* 1. Определить, есть ли в данном массиве строка, состоящая только из положительных элементов.
  2. Определить, есть ли в данном массиве столбец, состоящий только из отрицательных элементов.
  3. В каждой строке найти максимум и заменить его на противоположный элемент.
  4. В каждом столбце найти минимум и заменить его нулем.

1. Дан массив размером nxn, элементы которого целые числа.

Для хранения массив n?n использовать ступенчатый массив.

Найти максимальный элемент в каждой строке и записать данные в новый массив.

Пример

using System;

namespace ConsoleApplication

{

class Class

{

static int [][] Input ()

{

Console.WriteLine("введите размерность массива");

Console.Write("n = ");

int n=int.Parse(Console.ReadLine());

int [][]a=new int[n][];

for (int i = 0; i < n; ++i)

{

a[i]=new int [n];

for (int j = 0; j < n; ++j)

{

Console.Write("a[{0},{1}]= ", i, j);

a[i][j]=int.Parse(Console.ReadLine());

}

}

return a;

}

static void Print1(int[] a)

{

for (int i = 0; i < a.Length; ++i)

Console.Write("{0,5} ", a[i]);

}

static void Print2(int[][] a)

{

for (int i = 0; i < a.Length; ++i,Console.WriteLine() )

for (int j = 0; j < a[i].Length; ++j)

Console.Write("{0,5} ", a[i][j]);

}

static int Max(int[] a)

{

int max=a[0];

for (int i = 1; i < a.Length; ++i)

if (a[i] >max) {max=a[i];}

return max;

}

static void Main()

{

int[][] myArray=Input();

Console.WriteLine("Исходный массив:");

Print2(myArray);

int[]rez=new int [myArray.Length];

for (int i=0;i<myArray.Length; ++i)

rez[i]=Max(myArray[i]);

Console.WriteLine("Новый массив:");

Print1(rez);

}

}

}

* 1. Найти минимальный элемент в каждом столбце и записать данные в новый массив.
  2. Четные столбцы таблицы заменить на вектор Х.
  3. Нечетные строки таблицы заменить на вектор Х.
  4. Вычислить А*Х, где А - двумерная матрица, X - вектор.
  5. Для каждой строки подсчитать количество положительных элементов и записать данные в новый массив.
  6. Для каждого столбца подсчитать сумму отрицательных элементов и записать данные в новый массив.
  7. Для каждого столбца подсчитать сумму четных положительных элементов и записать данные в новый массив.
  8. Для каждой строки подсчитать количество элементов, больших заданного числа, и записать данные в новый массив.
  9. Для каждого столбца найти первый положительный элемент и записать данные в новый массив.
  10. Для каждой строки найти последний четный элемент и записать данные в новый массив.
  11. Для каждого столбца найти номер последнего нечетного элемента и записать данные в новый массив.
  12. Для каждой строки найти номер первого отрицательного элемента и записать данные в новый массив.
  13. Для каждой строки найти сумму элементов с номерами от k1 до k2  и записать данные в новый массив.
  14. Для каждого столбца найти произведение элементов с номерами от k1  до k2  и записать данные в новый массив.
  15. Для каждой строки подсчитать сумму элементов, не попадающих в заданный интервал, и записать данные в новый массив.
  16. Подсчитать сумму элементов каждой строки и записать данные в новый массив. Найти максимальный элемент нового массива.
  17. Подсчитать произведение элементов каждого столбца и записать данные в новый массив. Найти минимальный элемент нового массива.
  18. Для каждой строки найти номер первой пары неравных элементов. Данные записать в новый массив.
  19. Для каждого столбца найти номер первой пары одинаковых элементов. Данные записать в новый массив.

## Контрольные вопросы.

# Символы и строки

## Теоретические сведения.

Обработка текстовой информации является одной из самых распространенных задач современного программировании. С# предоставляет для ее решения широкий набор средств: символы char, неизменяемые строки string, изменяемые строки StringBuider и регулярные выражения Regex. В данном разделе мы рассмотрим работу с символами, неизменяемыми и изменяемыми строками.

### Символы char

Символьный тип char предназначен для хранения символа в кодировке Unicode. Символьный тип относится к встроенным типам данных С# и соответствует стандартному классу Сhar библиотеки .Net из пространства имен System. В этом классе определены статические методы, позволяющие задавать вид и категорию символа, а также преобразовывать символ в верхний или нижний регистр, в число. Рассмотрим основные методы:

|  |  |
| --- | --- |
| Метод | Описание |
| GetNumericValue | Возвращает числовое значение символа, если он является цифрой, и -1 в противном случае. |
| GetUnicodeCategory | Возвращает категорию Unicode-символа. В Unicode символы разделены на категории, например цифры ( DecimalDigitNumber ), римские цифры ( LetterNumber ), разделители строк ( LineSeparator ), буквы в нижнем регистре ( LowercaseLetter ) и т.д. |
| IsControl | Возвращает true, если символ является управляющим. |
| IsDigit | Возвращает true, если символ является десятичной цифрой. |
| IsLetter | Возвращает true, если символ является буквой. |
| IsLetterOrDigit | Возвращает true, если символ является буквой или десятичной цифрой. |
| IsLower | Возвращает true, если символ задан в нижнем регистре. |
| IsNumber | Возвращает true, если символ является числом (десятичным или шестнадцатеричным). |
| IsPunctuation | Возвращает true, если символ является знаком препинания. |
| IsSeparator | Возвращает true, если символ является разделителем. |
| IsUpper | Возвращает true, если символ задан в верхнем регистре. |
| IsWhiteSpace | Возвращает true, если символ является пробельным (пробел, перевод строки, возврат каретки). |
| Parse | Преобразует строку в символ (строка должна состоять из одного символа). |
| ToLower | Преобразует символ в нижний регистр |
| ToUpper | Преобразует символ в верхний регистр |

В следующем примере рассмотрим применение данных методов:

static void Main()

{

try

{

char b = 'B', c = '\x64', d = '\uffff';

Console.WriteLine("{0}, {1}, {2}", b, c, d);

Console.WriteLine("{0}, {1}, {2}", char.ToLower(b), char.ToUpper(c), char.GetNumericValue(d));

char a;

do //цикл выполнятеся до тех пор, пока не ввели символ e

{

Console.WriteLine("Введите символ: ");

a = char.Parse(Console.ReadLine());

Console.WriteLine("Введен символ {0}, его код {1}, его категория {2}", a,

(int)a, char.GetUnicodeCategory(a));

if (char.IsLetter(a)) Console.WriteLine("Буква");

if (char.IsUpper(a)) Console.WriteLine("Верхний регистр");

if (char.IsLower(a)) Console.WriteLine("Нижний регистр");

if (char.IsControl(a)) Console.WriteLine("Управляющий символ");

if (char.IsNumber(a)) Console.WriteLine("Число");

if (char.IsPunctuation(a)) Console.WriteLine("Разделитель");

} while (a != 'e');

}

catch

{

Console.WriteLine("Возникло исключение");

}

}

Используя символьный тип можно оргранизовать массив символов и работать с ним на основе базового класса Array:

static void Main()

{

char[] a ={ 'm', 'a', 'Х', 'i', 'M', 'u', 'S' , '!', '!', '!' };

char [] b="кол около колокола".ToCharArray(); //преобразование строки в массив символов

PrintArray("Исходный массив а:", a);

for (int x=0;x<a.Length; x++)

if (char.IsLower(a[x])) a[x]=char.ToUpper(a[x]);

PrintArray("Измененный массив а:", a);

PrintArray("Исходный массив b:", b);

Array.Reverse(b);

PrintArray("Измененный массив b:", b);

}

static void PrintArray(string line, Array a)

{

Console.WriteLine(line);

foreach( object x in a) Console.Write(x);

Console.WriteLine('\n');

}

### Неизменяемые строки string

Тип string, предназначенный для работы со строками символов в кодировке Unicode, является встроенным типом С#. Ему соответствует базовый тип класса System.String библиотеки .Net. Каждый объект string - это неизменяемая последовательность символов Unicode, т.е. методы, предназначенные для изменения строк, возвращают измененные копии, исходные же строки остаются неизменными.

Создать строку можно несколькими способами:

1. string s; // инициализация отложена
2. string s=''кол около колокола''; //инициализация строковым литералом
3. string s=new string (' ', 20); //конструктор создает строку из 20 пробелов
4. int x = 12344556; //инициализировали целочисленную переменную
5. string s = x.ToString(); //преобразовали ее к типу string
6. char [] a={'a', 'b', 'c', 'd', 'e'}; //создали массив символов
7. string v=new string (a); // создание строки из массива символов
8. char [] a={'a', 'b', 'c', 'd', 'e'};
9. // создание строки из части массива символов, при этом: 0

string v=new string (a, 0, 2)

// показывает с какого символа, 2 - сколько символов

// использовать для инициализации

Класс string обладает богатым набором методов для сравнения строк, поиска в строке и других действий со строками. Рассмотрим эти методы.

|  |  |  |
| --- | --- | --- |
| Название | Вид | Описание |
| Compare | Статический метод | Сравнение двух строк в лексикографическом (алфавитном) порядке. Разные реализации метода позволяют сравнивать строки с учетом или без учета регистра. |
| CompareTo | Метод | Сравнение текущего экземпляра строки с другой строкой. |
| Concat | Статический метод | Слияние произвольного числа строк. |
| Copy | Статический метод | Создание копии строки |
| Empty | Статическое поле | Открытое статическое поле, представляющее пустую строку |
| Format | Статический метод | Форматирование строки в соответствии с заданным форматом |
| IndexOf, IndexOfAny, *LastIndexOf*, LastIndexOfAny | Экземплярные методы | Определение индексов первого и последнего вхождения заданной подстроки или любого символа из заданного набора в данную строку. |
| Insert | Экземплярный метод | Вставка подстроки в заданную позицию |
| Join | Статический метод | Слияние массива строк в единую строку. Между элементами массива вставляются разделители. |
| Length | Свойство | Возвращает длину строки |
| PadLeft, PadRigth | Экземплярные методы | Выравнивают строки по левому или правому краю путем вставки нужного числа пробелов в начале или в конце строки. |
| Remove | Экземплярный метод | Удаление подстроки из заданной позиции |
| Replace | Экземплярный метод | Замена всех вхождений заданной подстроки или символа новыми подстрокой или символом. |
| Split | Экземплярный метод | Разделяет строку на элементы, используя разные разделители. Результаты помещаются в массив строк. |
| StartWith, EndWith | Экземплярные методы | Возвращают true или false в зависимости от того, начинается или заканчивается строка заданной подстрокой. |
| Substring | Экземплярный метод | *Выделение подстроки*, начиная с заданной позиции |
| ToCharArray | Экземплярный метод | Преобразует строку в массив символов |
| ToLower, ToUpper | Экземплярные методы | Преобразование строки к нижнему или верхнему регистру |
| Trim, TrimStart, TrimEnd | Экземплярные методы | Удаление пробелов в начале и конце строки или только с одного ее конца. |

Напоминаем, что вызов статических методов происходит через обращение к имени класса, например, String.Concat(str1, str2), в остальных случаях через обращение к экземплярам класса, например, str.ToLower(). На примере рассмотрим использование данных свойств и методов.

static void Main()

{

string str1 ="Первая строка";

string str2 = string.Copy(str1);

string str3 = "Вторая строка";

string str4 = "ВТОРАЯ строка";

string strUp, strLow;

int result, idx;

Console.WriteLine("str1: " + str1);

Console.WriteLine("Длина строки str1: " +str1.Length);

// Создаем прописную и строчную версии строки str1.

strLow = str1.ToLower();

strUp = str1.ToUpper();

Console.WriteLine("Строчная версия строки str1: " +strLow);

Console.WriteLine("Прописная версия строки str1: " +strUp);

Console.WriteLine();

// Сравниваем строки,

result = str1.CompareTo(str3);

if (result == 0) Console.WriteLine("str1 и str3 равны.");

else if (result < 0) Console.WriteLine("str1 меньше, чем str3");

else Console.WriteLine("str1 больше, чем str3");

Console.WriteLine();

//сравниваем строки без учета регистра

result = String.Compare(str3,str4,true);

if (result == 0) Console.WriteLine("str3 и str4 равны без учета регистра.");

else Console.WriteLine("str3 и str4 не равны без учета регистра.");

Console.WriteLine();

//сравниваем части строк

result = String.Compare(str1, 4, str2, 4, 2);

if (result == 0) Console.WriteLine("часть str1 и str2 равны");

else Console.WriteLine("часть str1 и str2 не равны");

Console.WriteLine();

// Поиск строк.

idx = str2.IndexOf("строка");

Console.WriteLine("Индекс первого вхождения подстроки строка: " + idx);

idx = str2.LastIndexOf("о");

Console.WriteLine("Индекс последнего вхождения символа о: " + idx);

//конкатенация

string str=String.Concat(str1, str2, str3, str4);

Console.WriteLine(str);

//удаление подстроки

str=str.Remove(0,str1.Length);

Console.WriteLine(str);

//замена подстроки "строка" на пустую подстроку

str=str.Replace("строка","");

Console.WriteLine(str);

}

Очень важными методами обработки строк, являются методы разделения строки на элементы Split и слияние массива строк в единую строку Join.

static void Main()

{

string poems = "тучки небесные вечные странники";

char[] div = { ' '}; //создаем массив разделителей

// Разбиваем строку на части,

string[] parts = poems.Split(div);

Console.WriteLine("Результат разбиения строки на части: ");

for (int i = 0; i < parts.Length; i++)

Console.WriteLine(parts[i]);

// Теперь собираем эти части в одну строку, в качестве разделителя используем символ |

string whole = String.Join(" | ", parts);

Console.WriteLine("Результат сборки: ");

Console.WriteLine(whole);

}

В общем случае строка может содержать и другие разделители:

static void Main()

{

string poems = "Тучки небесные, вечные странники...";

char[] div = { ' ', ',', '.'}; //создаем массив разделителей

// Разбиваем строку на части,

string[] parts = poems.Split(div);

Console.WriteLine("Результат разбиения строки на части: ");

for (int i = 0; i < parts.Length; i++)

Console.WriteLine(parts[i]);

// Теперь собираем эти части в одну строку,

string whole = String.Join(" | ", parts);

Console.WriteLine("Результат сборки: ");

Console.WriteLine(whole);

}

Рассмотрим другой пример - используя метод Split вводить двумерный массив можно не поэлементно, а построчно:

static void Main()

{

try

{

int[][] MyArray;

Console.Write("введите количество строк: ");

int n = int.Parse(Console.ReadLine());

MyArray = new int[n][];

for (int i = 0; i < MyArray.Length; i++)

{

string line = Console.ReadLine();

string[] mas = line.Split(' ');

MyArray[i] = new int[mas.Length];

for (int j = 0; j < MyArray[i].Length; j++)

{

MyArray[i][j] = int.Parse(mas[j]);

}

}

PrintArray("исходный массив:", MyArray);

for (int i = 0; i < MyArray.Length; i++) Array.Sort(MyArray[i]);

PrintArray("итоговый массив", MyArray);

}

catch

{

Console.WriteLine("возникло исключение");

}

}

static void PrintArray(string a, int[][] mas)

{

Console.WriteLine(a);

for (int i = 0; i < mas.Length; i++)

{

foreach (int x in mas[i]) Console.Write("{0} ", x);

Console.WriteLine();

}

}

В этом примере могут возникнуть исключительные ситуации, если введенная строка элементов массива будет содержать лишние пробелы. Следовательно, от этих пробелов нужно избавиться:

static void Main()

{

try

{

int[][] MyArray;

Console.Write("введите количество строк: ");

string line= Console.ReadLine()

int n = int.Parse(line.Trim());

MyArray = new int[n][];

for (int i = 0; i < MyArray.Length; i++)

{

line = Console.ReadLine();

line=line.Trim(); //удалаяем пробелы в начале и конце строки

//удаляем линшие пробелы внутри строки

n = line.IndexOf(" ");

while (n > 0)

{

line = line.Remove(n, 1);

n = line.IndexOf(" ");

}

string[] mas = line.Split(' ');

MyArray[i] = new int[mas.Length];

for (int j = 0; j < MyArray[i].Length; j++)

{

MyArray[i][j] = int.Parse(mas[j]);

}

}

PrintArray("исходный массив:", MyArray);

for (int i = 0; i < MyArray.Length; i++) Array.Sort(MyArray[i]);

PrintArray("итоговый массив", MyArray);

}

catch

{

Console.WriteLine("возникло исключение");

}

}

static void PrintArray(string a, int[][] mas)

{

Console.WriteLine(a);

for (int i = 0; i < mas.Length; i++)

{

foreach (int x in mas[i]) Console.Write("{0} ", x);

Console.WriteLine();

}

}

При работе с объектами класса string нужно учитывать их свойство неизменяемости, т.е. тот факт, что методы изменяют не сами строки, а их копии. Рассмотрим фрагмент программы:

string a="";

for (int i = 1; i <= 100; i++) a +="!";

Console.WriteLine(a);

В этом случае в памяти компьютера будет сформировано 100 различных строк вида:

!

!!

!!!

…

!!!...!!

И только последняя строка будет храниться в переменной а. Ссылки на все остальные строчки будут потеряны, но эти строки будут храниться в памяти компьютера и засорять память. Бороться с таким засорением придется сборщику мусора, что будет сказываться на производительности программы. Поэтому если нужно изменять строку, то лучше пользоваться классом StringBuilder.

### Изменяемые строки

Чтобы создать строку, которую можно изменять, в С# предусмотрен класс StringBuilder, определенный в пространстве имен System.Text. Объекты этого класса всегда объявляются с явным вызовом конструктора класса (через операцию new ) . Примеры создания изменяемых строк:

//создание пустой строки, размер по умолчанию 16 символов

StringBuilder a =new StringBuilder();

//инициализация строки и выделение необходимой памяти

StringBuilder b = new StringBuilder("abcd");

//создание пустой строки и выделение памяти под 100 символов

StringBuilder с = new StringBuilder(100);

//инициализация строки и выделение памяти под 100 символов

StringBuilder d = new StringBuilder("abcd", 100);

//инициализация подстрокой "bcd", и выделение памяти под 100 символов

StringBuilder d = new StringBuilder("abcd", 1, 3,100);

Основные элементы класса приведены в таблице:

|  |  |  |
| --- | --- | --- |
| Название | Вид | Описание |
| Append | *Экземплярный метод* | Добавление данных в конец строки. Разные варианты метода позволяют добавлять в строку величины любых встроенных типов, массивы символов, строки и подстроки string. |
| AppendFormat | *Экземплярный метод* | Добавление форматированной строки в конец строки |
| Capacity | свойство | Получение и установка емкости буфера. Если устанавливаемое значение меньше текущей длины строки или больше максимального, то генерируется исключение ArgumentOutOfRangeException |
| Insert | *Экземплярный метод* | Вставка подстроки в заданную позицию |
| Length | изменяемое свойство | Возвращает длину строки. Присвоение ему значения 0 сбрасывает содержимое и очищает строку |
| MaxCapacity | неизменное свойство | Возвращает наибольшее количество символов, которое может быть размещено в строке |
| Remove | *Экземплярный метод* | Удаление подстроки из заданной позиции |
| Replace | *Экземплярный метод* | Замена всех вхождений заданной подстроки или символа новой подстрокой или символом |
| ToString | *Экземплярный метод* | Преобразование в строку типа string |
| Chars | изменяемое свойство | Возвращает из массива или устанавливает в массиве символ с заданным индексом. Вместо него можно пользоваться квадратными скобками [] |
| Equals | *Экземплярный метод* | Возвращает true, только если объекты имеют одну и ту же длину и состоят из одних и тех же символов |
| CopyTo | *Экземплярный метод* | Копирует подмножество символов строки в массив char |

Как видим, методы класса StringBuilder менее развиты, чем методы класса String, но они позволяют более эффективно использовать память за счет работы с изменяемыми строками. Рассмотрим примеры использования данных методов.

static void Main()

{

try

{

StringBuilder str=new StringBuilder("Площадь");

PrintString(str);

str.Append(" треугольника равна");

PrintString(str);

str.AppendFormat(" {0:f2} см ", 123.456);

PrintString(str);

str.Insert(8, "данного ");

PrintString(str);

str.Remove(7, 21);

PrintString(str);

str.Replace("а", "о");

PrintString(str);

StringBuilder str1=new StringBuilder(Console.ReadLine());

StringBuilder str2=new StringBuilder(Console.ReadLine());

Console.WriteLine(str1.Equals(str2));

}

catch

{

Console.WriteLine("Вознико исключение");

}

}

static void PrintString(StringBuilder a)

{

Console.WriteLine("Строка: "+a);

Console.WriteLine("Текущая длина строки " +a.Length);

Console.WriteLine("Объем буфера "+a.Capacity);

Console.WriteLine("Максимальный объем буфера "+a.MaxCapacity);

Console.WriteLine();

}

С изменяемой строкой можно работать не только как с объектом, но как с массивом символов:

static void Main()

{

StringBuilder a = new StringBuilder("2\*3=3\*2");

Console.WriteLine(a);

int k=0;

for (int i = 0; i < a.Length; ++i )

if (char.IsDigit(a[i])) k+=int.Parse(a[i].ToString());

Console.WriteLine(k);

}

На практике часто комбинируют работу с изменяемыми и неизменяемыми строками. Однако если необходимо изменять строку, то в этом случае используют StringBuilder.

**Пример**. Дана строка, в которой содержится осмысленное текстовое сообщение. Слова сообщения разделяются пробелами и знаками препинания. Вывести все слова сообщения, которые начинаются и заканчиваются на одну и ту же букву.

static void Main()

{

Console.WriteLine("Введите строку: ");

StringBuilder a = new StringBuilder(Console.ReadLine());

Console.WriteLine("Исходная строка: "+a);

for (int i=0; i<a.Length;)

if (char.IsPunctuation(a[i])) a.Remove(i,1);

else ++i;

string str=a.ToString();

string []s=str.Split(' ');

Console.WriteLine("Искомые слова: ");

for (int i=0; i<s.Length; ++i)

if (s[i][0]==s[i][s[i].Length-1]) Console.WriteLine(s[i]);

}

## Задания на лабораторную работу.

При решении задач следует руководствоваться правилом: если в строке должны производится изменения, то лучше пользоваться классом StringBuilder ; если необходимо разбивать строки на слова, то классом String. При решении некоторых задач, потребуется использовать оба класса.

Разработать программу, которая для заданной строки s:

* 1. вставляет символ x после каждого вхождения символа y ;

Пример

using System;

using System.Text;

namespace ConsoleApplication

{

class Class

{

static void Main()

{

Console.WriteLine("Введите строку: ");

StringBuilder a = new StringBuilder(Console.ReadLine());

Console.WriteLine("Исходная строка: "+a);

Console.WriteLine("Введите символ x: ");

char x=char.Parse(Console.ReadLine());

Console.WriteLine("Введите символ y: ");

char y=char.Parse(Console.ReadLine());

for (int i=0; i<a.Length; ++i)

if (a[i]==y){a.Insert(i+1,x); ++i;}

Console.WriteLine("Измененная строка: "+a);

}

}

}

* 1. меняет местами первую букву со второй, третью с четвертой и т.д.
  2. определяет, какой из двух заданных символов встречается чаще в строке;
  3. подсчитывает общее число вхождений символов х и y ;
  4. подсчитывает количество букв в строке;
  5. определяет, имеются ли в строке два соседствующих одинаковых символа;
  6. удаляет среднюю букву, если длина строки нечетная, и две средних, если длина строки четная;
  7. удваивает каждое вхождение заданного символа x;
  8. удаляет все символы х;
  9. удаляет все подстроки substr ;
  10. заменяет все вхождения подстроки substr1 на подстроку substr2 ;
  11. подсчитывает сумму всех содержащихся в ней цифр;
  12. подсчитывает количество содержащихся в ней цифр;
  13. находит порядковые номера первого и последнего вхождения символа x ;
  14. заменяет все группы стоящих рядом точек на многоточие;
  15. выводит на экран последовательность символов, расположенных до первого двоеточия;
  16. выводит на экран последовательность символов, расположенных после последнего двоеточия;
  17. удаляет из нее последовательность символов, расположенных между круглыми скобками (считается, что в строке ровно одна пара круглых скобок).
  18. удаляет из нее последовательность символов, расположенных между двумя запятыми (считается, что в строке ровно две запятые);
  19. определяет, сколько различных символов встречается в строке.

1. Дана строка, в которой содержится осмысленное текстовое сообщение. Слова сообщения разделяются пробелами и знаками препинания.

Вывести только те слова сообщения, в которых содержится заданная подстрока.

Пример

using System;

using System.Text;

namespace ConsoleApplication

{

class Class

{

static void Main()

{

Console.WriteLine("Введите строку: ");

StringBuilder a = new StringBuilder(Console.ReadLine());

Console.WriteLine("Исходная строка: "+a);

Console.WriteLine("Введите заданную подстроку: ");

string x=Console.ReadLine();

for (int i=0; i<a.Length;)

if (char.IsPunctuation(a[i]))a.Remove(i,1);

else ++i;

string str=a.ToString();

str=str.Trim();

string []s=str.Split(' ');

Console.WriteLine("Искомые слова: ");

for (int i=0; i<s.Length; ++i)

if (s[i].IndexOf(x)!=-1) Console.WriteLine(s[i]);

}

}

}

1. Вывести только те слова сообщения, которые содержат не более чем n букв.
2. Вывести только те слова сообщения, которые начинаются с прописной буквы.
3. Вывести только те слова сообщения, которые содержат хотя бы одну цифру.
4. Удалить из сообщения все слова, которые заканчиваются на заданный символ.
5. Удалить из сообщения все слова, содержащие данный символ (без учета регистра).
6. Удалить из сообщения все однобуквенные слова (вместе с лишними пробелами).
7. Удалить из сообщения все повторяющиеся слова (без учета регистра).
8. Подсчитать сколько раз заданное слово встречается в сообщении.
9. Подсчитать сколько слов, состоящих только из прописных букв, содержится в сообщении.
10. Найти самое длинное слово сообщения.
11. Найти все самые длинные слова сообщения.
12. Найти самое короткое слово сообщения.
13. Найти все самые короткие слова сообщения.
14. Вывести на экран все слова-палиндромы, содержащиеся в сообщении.
15. По правилу расстановки знаков препинания перед каждым знаком препинания пробел отсутствует, а после него обязательно стоит пробел. Учитывая данное правило, проверьте текст на правильность расстановки знаков препинания и, если необходимо, внесите в текст изменения.
16. Вывести только те слова, которые встречаются в тексте ровно один раз.
17. Вывести только те слова, которые встречаются более n раз.
18. Вывести слова сообщения в алфавитном порядке.
19. Вывести слова сообщения в порядке возрастания их длин.

## Контрольные вопросы.

## 

# С#. Классы

## Теоретические сведения.

### Классы

#### Основные понятия

Класс - это обобщенное понятие, определяющие характеристики и поведение некоторого множества объектов, называемых экземплярами класса. "Классический" класс содержит данные, определяющие *свойства объектов класса*, и методы, определяющие их поведение. Для Windows-приложений в класс добавляется третья составляющая - события, на которые может *реагировать объект*класса. Все классы библиотеки .Net, а также все классы, которые создает программист в среде .Net, имеют одного общего предка - класс object.

Все программы, рассмотренные ранее, состояли из одного класса с одним методом Main и несколькими вспомогательными статическими методами. Теперь рассмотрим понятие "класс" более подробно.

Описание класса содержит ключевое слово class, за которым следует его *имя*, а далее в фигурных скобках - *тело* класса. Кроме того, для класса можно задать его базовые классы (предки) и ряд необязательных атрибутов и спецификаторов, определяющих различные характеристики класса:

[ атрибуты ] [ спецификаторы ] class имя\_класса [ : предки ]

{тело\_класса}

Пример класса:

class Demo{}

Спецификаторы определяют свойства класса, а также доступность класса для других элементов программы. Возможные значения спецификаторов перечислены в следующей таблице:

|  |  |  |
| --- | --- | --- |
| № | Спецификатор | Описание |
| 1 | new | Задает новое описание класса взамен унаследованного от предка. Используется для вложения классов (в иерархии объектов). |
| 2 | public | Доступ к классу не ограничен |
| 3 | protected | Доступ только из данного или производного класса. Используется для вложенных классов. |
| 4 | internal | Доступ только из данной программы (сборки). |
| 5 | protected internal | Доступ только из данного и производного класса и из данной программы (сборки). |
| 6 | private | Доступ только из элементов класса, внутри которых описан данный класс. Используется для вложенных классов. |
| 7 | static | Статический класс. Позволяет обращатся к методам класса без создания экземпляра класса |
| 8 | sealed | Бесплодный класс. Запрещает наследование данного класса. Применяется в иерархии объектов. |
| 9 | abstract | Абстрактный класс. Применяется в иерархии объектов. |

Спецификаторы 2-6 называются спецификаторами *доступа*. Они определяют, откуда можно непосредственно обращаться к данному классу. Спецификаторы доступа могут комбинироваться с остальными спецификаторами.

Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. В последнем случае класс называется вложенным. В зависимости от места описания класса некоторые из этих спецификаторов могут быть запрещены. В данном разделе мы рассмотрим классы, которые описываются непосредственно в пространстве имен. Для таких классов допускаются только два спецификатора: public и internal. По умолчанию, то есть если ни один спецификатор доступа не указан, подразумевается спецификатор internal.

Объекты создаются явным или неявным образом, то есть либо программистом, либо системой. Программист создает экземпляр класса с помощью операции new, например:

Demo a = new Demo (); // Создается экземпляр класса Demo

Если достаточный для хранения объекта объем памяти выделить не удалось, то генерируется исключение OutOfMemoryException.

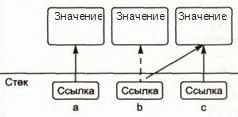
Для каждого объекта при его создании в памяти выделяется отдельная область, в которой хранятся его данные. В классе могут присутствовать *статические элементы*, которые существуют в единственном экземпляре для всех объектов класса. Статические данные часто называют данными класса, а остальные - данными экземпляра. Для работы с данными класса используются статические методы класса, для работы с данными экземпляра - *методы экземпляра*, или просто методы.

До сих пор использовали в программах только данные (переменные и константы) и методы. В общем случае класс может содержать следующие функциональные элементы:

1. Данные: переменные или константы.
2. Методы, реализующие не только вычисления, но и другие действия, выполняемые классом или его экземпляром.
3. Конструкторы (реализуют действия по *инициализации экземпляров* или класса в целом).
4. Свойства (определяют характеристики класса в соответствии со способами их задания и получения).
5. Деструкторы (определяют действия, которые необходимо выполнить до того, как объект будет уничтожен).
6. Индексаторы (обеспечивают возможность доступа к элементам класса по их порядковому номеру).
7. Операции (задают действия с объектами с помощью знаков операций).
8. События (определяют уведомления, которые может генерировать класс).
9. Типы (типы данных, внутренние по отношению к классу).

В данном разделе рассмотрим первые четыре категории элементов класса

Прежде чем приступить к проектированию классов, необходимо поговорить о присваивании и сравнении объектов. Механизм выполнения присваивания один и тот же для величин любого типа, как ссылочного, так и размерного, однако результаты различаются. При присваивании значения копируется значение, а при присваивании ссылки - ссылка, поэтому после присваивания одного объекта другому мы получим две ссылки, указывающие на одну и ту же область памяти:



Пусть были созданы три объекта а, b и с, а затем выполнено присваивание b = с. Теперь ссылки b и с указывают на один и тот же объект. Старое значение b становится недоступным и очищается сборщиком мусора.

Аналогичная ситуация с операцией проверки на равенство. Величины *значимого типа* равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одни и те же данные. Так, объекты b и с равны, т.к. они ссылаются на одну и ту же область памяти. Но а не равно b даже при равенстве их значений.

#### Данные: поля и константы

Данные, содержащиеся в классе, могут быть *переменными* или *константами* и задаются в соответствии с правилами, рассмотренными в теме "Идентификаторы". При описании данных также можно указывать атрибуты и спецификаторы, задающие различные характеристики элементов. Синтаксис описания элемента данных приведен ниже:

[атрибуты] [спецификаторы] [const] тип имя [ = начальное\_значение ]

Рассмотрим возможные спецификаторы для данных:

|  |  |  |
| --- | --- | --- |
| № | Спецификатор | Описание |
| 1 | new | Новое описание поля, скрывающее унаследованный элемент класса |
| 2 | public | Доступ к элементу не ограничен |
| 3 | protected | Доступ только из данного и производных классов |
| 4 | internal | Доступ только из данной сборки |
| 5 | protected internal | Доступ только из данного и производных классов и из данной сборки |
| 6 | private | Доступ только из данного класса |
| 7 | static | Одно поле для всех экземпляров класса |
| 8 | readonly | Поле доступно только для чтения (значение таких полей можно установить либо при описании, либо в конструкторе) |
| 9 | *volatile* | Поле может изменяться другим процессом или системой |

Для констант можно использовать только спецификаторы 1-6.

По умолчанию элементы класса считаются закрытыми private. Для полей класса этот вид доступа является предпочтительным, поскольку поля определяют внутреннее строение класса, которое должно быть скрыто от пользователя. Все методы класса имеют непосредственный доступ к его закрытым полям.

Поля, описанные со спецификатором static, а также константы существуют в единственном экземпляре для всех объектов класса, поэтому к ним обращаются не через имя экземпляра, а через имя класса. Обращение к полю класса выполняется с помощью операции доступа (точка). Справа от точки задается имя поля, слева - имя экземпляра для обычных полей или имя класса для статических. Рассмотрим пример создания класса Demo и два способа обращения к его полям.

class Circle

{

public int x=0;

public int y=0;

public int radius=3;

public const double pi = 3.14;

public static string name = "Окружность";

double p;

double s;

}

class Program

{

static void Main()

{

Circle cr = new Circle(); //создание экземпляра класса

Console.WriteLine("pi=" + Circle.pi);// обращение к константе

Console.Write(Circle.name);// обращение к статическому полю

//обращение к обычным полям

Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", cr.x, cr.y, cr.radius);

// Console.WriteLine(cr.p); - вызовет ошибку, т.к. поле p имеет тип private

Console.Write("Введите коэффициент= ");

int kof = int.Parse(Console.ReadLine());

cr.x -= kof; cr.y += kof; cr.radius \*= kof;

Console.WriteLine(" Новая окружность с центром в точке ({0},{1}) и радиусом {2}",

cr.x, cr.y, cr.radius);

//cr.s = 2 \* Circle.pi \* cr.radius; - вызовет ошибку, т.к. поле s имеет тип private

}

}

#### Методы

Создание и использование методов было рассмотрено ранее. Теперь рассмотрим использование методов в контексте создания классов.

Методы находятся в памяти в единственном экземпляре и используются всеми объектами одного класса совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается скрытый параметр this, в котором хранится ссылка на вызвавший функцию экземпляр.

В явном виде параметр this применяется для того, чтобы возвратить из метода ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода, например:

class Circle

{

public int x=0;

public int y=0;

public int radius=3;

public const double pi = 3.14;

public static string name = "Окружность";

public Circle T() //метод возвращает ссылку на экземпляр класса

{

return this;

}

public void Set(int x, int y, int r)

{

this.x = x;

this.y = y;

radius=r;

}

}

class Program

{

static void Main()

{

Circle cr = new Circle(); //создание экземпляра класса

Console.WriteLine("pi=" + Circle.pi);// обращение к константе

Console.Write(Circle.name);// обращение к статическому полю

//обращение к обычным полям

Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", cr.x, cr.y, cr.radius);

cr.Set(1, 1, 10);

Console.WriteLine("Новая окружность с центром в точке ({0},{1}) и радиусом {2}",

cr.x, cr.y, cr.radius);

Circle b=cr.T();//получаем ссылку на объект cr, аналог b=c

Console.WriteLine("Новая ссылка на окружность с центром в точке ({0},{1})

и радиусом {2}", b.x, b.y, b.radius);

}

}

#### Конструкторы

Конструктор предназначен для инициализации объекта. Конструкторы делятся на конструкторы класса (для статических классов) и конструкторы экземпляра класса (всех остальных классов).

#### Конструкторы экземпляра

Конструктор экземпляра вызывается автоматически при создании объекта класса с помощью операции new. Имя конструктора совпадает с именем класса. Рассмотрим основные свойства конструкторов:

1. Конструктор не возвращает значение, даже типа void.
2. Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации.
3. Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, полям *значимых типов*присваивается нуль, полям ссылочных типов - значение null.

До сих пор задавали начальные значения полей класса при описании класса. Это удобно в том случае, когда для всех экземпляров класса начальные значения некоторого поля одинаковы. Если же при создании объектов требуется присваивать полю разные значения, это следует делать с помощью явного задания конструктора. В следующем примере добавлен конструктор и метод Print для вывода информации об объекте:

class Circle

{

public int x;

public int y;

public int radius;

public const double pi = 3.14;

public static string name = "Окружность";

public Circle(int x, int y, int r)//конструктор

{

this.x = x;

this.y = y;

radius = r;

}

public void Print()

{

Console.Write(name);

Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", x, y, radius);

Console.WriteLine();

}

}

class Program

{

static void Main()

{

Circle a = new Circle(0, 0, 1); //вызов конструктора

a.Print();

Circle b=new Circle(10, 10, 5);//вызов конструктора

b.Print();

}

}

Часто бывает удобно задать в классе несколько конструкторов, чтобы обеспечить возможность инициализации объектов разными способами. Для этого конструкторы должны иметь разные сигнатуры. Рассмотрим это на примере:

class Circle

{

public int x;

public int y;

public int radius;

public const double pi = 3.14;

public static string name = "Окружность";

public Circle(int x, int y, int r)//конструктор 1

{

this.x = x;

this.y = y;

radius = r;

}

public Circle(int r)//конструктор 2

{

radius = r;

}

public void Print()

{

Console.Write(name);

Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", x, y, radius);

Console.WriteLine();

}

}

class Program

{

static void Main()

{

Circle a = new Circle(0, 0, 1); //вызов конструктора 1

a.Print();

Circle b=new Circle(5);//вызов конструктора 2

b.Print();

}

}

Обратите внимание на то, что в конструктое 2 не были инициализированы поля x, y, поэтому им присваивается значение 0.

Если один из конструкторов выполняет какие-либо действия, а другой должен делать то же самое плюс еще что-нибудь, то удобно вызвать первый конструктор из второго. Для этого используется уже известное вам ключевое слово this в другом контексте, например:

public Circle(int x, int y, int r):this(r) //конструктор 1

{

this.x = x;

this.y = y;

}

public Circle(int r) //конструктор 2

{

radius = r;

}

Конструкция, находящаяся после двоеточия, называется *инициализатором*, то есть тем кодом, который исполняется до начала выполнения тела конструктора. В нашем случае конструктор 1 до выполнения своего кода вызывает конструктор 2.

#### Конструкторы класса

Статические классы содержат только статические члены, в том числе и конструктор, которые хранятся в памяти в единственном экземпляре. Поэтому создавать экземпляры класса нет смысла.

Чтобы подчеркнуть этот факт, в первой версии С# для статических классов создавали два конструктора, один - пустой закрытый (private) конструктор, второй - статический конструктор, не имеющий параметров. Первый конструктор предотвращал попытки создания экземпляров класса. Второй конструктор автоматически вызывается системой до первого обращения к любому элементу статического класса, выполняя необходимые действия по инициализации. Вышесказанное отражено в следующем примере:

class Demo

{

static int a;

static int b;

private Demo(){} //закрытый конструктор

static Demo() //статический конструктор

{

a=10;

b=2;

}

public static void Print ()

{

Console.WriteLine("{0}+{1}={2}",a,b,a+b);

Console.WriteLine("{0}\*{1}={2}",a,b,a\*b);

Console.WriteLine("{0}-{1}={2}",a,b,a-b);

}

}

class Program

{

static void Main()

{

//Demo S=new Demo(); //ошибка содать экземпляр класса нельзя

Demo.Print();

}

}

В версию 2.0 введена возможность описывать статический класс, то есть класс с модификатором static. Экземпляры такого класса создавать запрещено, и кроме того, от него запрещено наследовать. Все элементы такого класса должны явным образом объявляться с модификатором static (константы и *вложенные типы* классифицируются как *статические элементы* автоматически). Конструктор экземпляра для статического класса задавать запрещается.

static class Demo

{

static int a=20;

static int b=10;

public static void Print ()

{

Console.WriteLine("{0}+{1}={2}",a,b,a+b);

Console.WriteLine("{0}\*{1}={2}",a,b,a\*b);

Console.WriteLine("{0}-{1}={2}",a,b,a-b);

}

}

class Program

{

static void Main()

{

Demo.Print();

}

}

#### Свойства

Иногда требуется создать поле, которое с одной стороны, должно быть доступно для использования, с другой стороны, возможность что-то сделать с этим полем имеет ограничения. Например, полю нельзя присваивать произвольное значение, а только значения из какого-то диапазона. Свойство предлагает простой и удобный способ решения этой проблемы.

Синтаксис свойства:

[атрибуты] [спецификаторы] тип имя\_свойства

{

[get код\_доступа]

[set код\_доступа]

}

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые (со спецификатором public ).

Код доступа представляет собой блоки операторов, которые выполняются при получении ( get ) или установке ( set ) свойства. Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, то свойство доступно только для чтения. Если отсутствует часть get, то свойство доступно только для записи.

Рассмотрим пример работы со свойствами, обращая внимание на то, что синтаксически чтение и запись свойства выглядят почти как методы. При этом get должен содержать оператор return, возвращающий выражение, для типа которого должно существовать *неявное преобразование* к типу свойства.

class Circle

{

//закрытые поля

int x;

int y;

int radius;

public static string name = "Окружность";

public Circle(int x, int y, int r):this(r)//конструктор 1

{

this.x = x;

this.y = y;

}

public Circle(int r)//конструктор 2

{

radius = r;

}

public void Print()

{

Console.Write(name);

Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", x, y, radius);

Console.WriteLine();

}

public int X //свойство для обращения к полю x

{

get

{

return x;

}

set

{

x = value;

}

}

public int Y //свойство для обращения к полю y

{

get

{

return y;

}

set

{

y = value;

}

}

public int R //свойство для обращения к полю radius

{

get

{

return radius;

}

set

{

radius = value;

}

}

public double P //свойство только для чтения

{

get

{

return 2\* Math.PI \*radius;

}

}

public double S //свойство только для чтения

{

get

{

return Math.PI \*radius\*radius;

}

}

}

class Program

{

static void Main()

{

Circle a = new Circle(0, 0, 1); //вызов конструктора

a.Print();

//установка новых значений

a.X=1;

a.Y=1;

a.R=10;

//a.S=100; //ошибка - свойство доступно только для чтения

Console.WriteLine("центр=({0},{1})

радиус={2}

периметр={3:f2}

площадь={4:f2}",

a.X, a.Y, a.R, a.P, a.S);

}

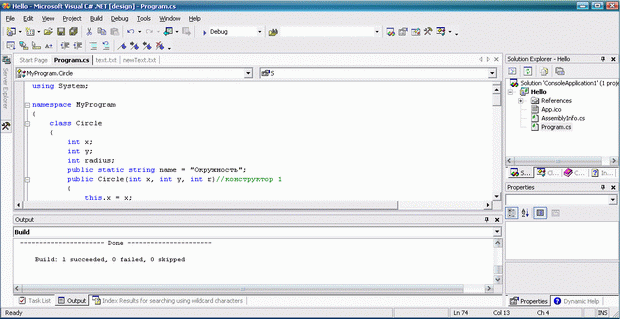
}

13.1.

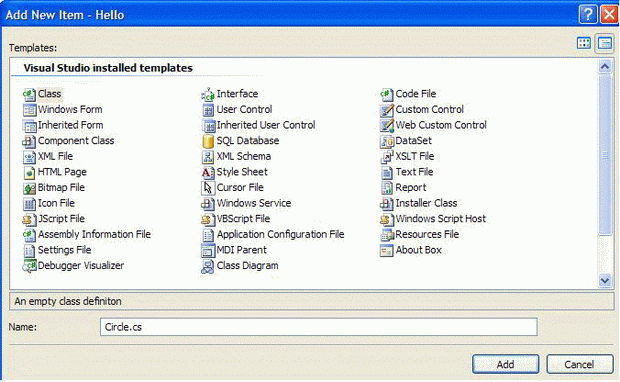
#### "Один класс - один файл",

С добавлением новых классов в программу резко увеличивается ее размер, что затрудняет ее прочтение. Поэтому следует руководствоваться одним простым принципом "один класс - один файл". Для того чтобы создать новый файл для класса Circleвыполним следующие действия:

1. В окне Solution Explorer щелкните правой кнопкой на имени проекта Hello (на рисунке выделен жирным).

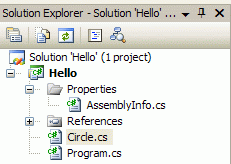


1. Выполните команду Add/Add Class…



В поле Name напишите Сircle.cs и нажмите кнопку Add.

Теперь окно Solution Explorer выглядит следующим образом:



А файл Circle.cs выглядит следующим образом:

using System;

namespace ConsoleApplication1

{

public class Circle

{

public Circle()

{

//

// TODO: Add constructor logic here

//

}

}

}

1. Замените namespace ConsoleApplication1 на namespace MyProgram, для того чтобы идентификаторы файлов Program.cs и Circle.cs были определены в одном пространстве имен.
2. Перенесите класс Circle из файла Program.cs в файл Circle.cs.
3. Теперь запустите программу и посмотрите, что она делает.

## Задания на лабораторную работу.

1. Создать класс Point, разработав следующие элементы класса:
   * Поля:
     + int x, y;
   * Конструкторы, позволяющие создать экземпляр класса:
     + с нулевыми координатами;
     + с заданными координатами.
   * Методы, позволяющие:
     + вывести координаты точки на экран;
     + рассчитать расстояние от начала координат до точки;
     + переместить точку на плоскости на вектор (a, b).
   * Свойства:
     + получить-установить координаты точки (доступное для чтений и записи);
     + позволяющие умножить координаты точки на скаляр (доступное только для записи).
2. Создать класс *Triangle*, разработав следующие элементы класса:
   * Поля:
     + int a, b, c;
   * Конструктор, позволяющий создать экземпляр класса с заданными длинами сторон.
   * Методы, позволяющие:
     + вывести длины сторон треугольника на экран;
     + расчитать периметр треугольника;
     + расчитать площадь треугольника.
   * Свойства:
     + позволяющее получить-установить длины сторон треугольника (доступное для чтения и записи);
     + позволяющее установить, существует ли треугольник с данными длинами сторон (доступное только для чтения).
3. Создать класс Rectangle, разработав следующие элементы класса:
   * Поля:
     + int a, b;
   * Конструктор, позволяющий создать экземпляр класса с заданными длинами сторон.
   * Методы, позволяющие:
     + вывести длины сторон прямоугольника на экран;
     + расчитать периметр прямоугольника;
     + расчитать площадь прямоугольника.
   * Свойства:
     + получить-установить длины сторон прямоугольника (доступное для чтения и записи);
     + позволяющее установить, является ли данный прямоугольник квадратом (доступное только для чтения).
4. Создать класс Money, разработав следующие элементы класса:
   * Поля:
     + int first;//номинал купюры
     + int second; //количество купюр
   * Конструктор, позволяющий создать экземпляр класса с заданными значениям полей.
   * Методы, позволяющие:
     + вывести номинал и количество купюр;
     + определить, хватит ли денежных средств на покупку товара на сумму N рублей.
     + определить, сколько шт товара стоимости n рублей можно купить на имеющиеся денежные средства.
   * Свойства:
     + позволяющее получить-установить значение полей (доступное для чтения и записи);
     + позволяющее расчитатать сумму денег (доступное только для чтения).
5. Создать класс для работы с одномерным массивом целых чисел. Разработать следующие элементы класса:
   * Поля:
     + int [] IntArray;
     + int n.
   * Конструктор, позволяющий создать массив размерности n.
   * Методы, позволяющие:
     + ввести элементы массива с клавиатуры;
     + вывести элементы массива на экран;
     + отсортировать элементы массива в порядке возрастания.
   * Свойства:
     + возвращающее размерность массива (доступное только для чтения);
     + позволяющее домножить все элементы массива на скаляр (доступное только для записи).
6. Создать класс для работы с двумерным массивом целых чисел. Разработать следующие элементы класса:
   * Поля:
     + int [,] IntArray;
     + int n.
   * Конструктор, позволяющий создать массив размерности nxn.
   * Методы, позволяющие:
     + ввести элементы массива с клавиатуры;
     + вывести элементы массива на экран;
     + вычислить сумму элеметов i -того столбца.
   * Свойства:
     + позволяющее вычислить количество нулевых элементов в массиве (доступное только для чтения);
     + позволяющее установить значение всех элементы главной диагонали массива равное скаляру (доступное только для записи).
7. Создать класс для работы с двумерным массивом вещественных чисел. Разработать следующие элементы класса:
   * Поля:
     + double [][] DoubelArray;
     + int n, m.
   * Конструктор, позволяющий создать массив размерности nxm.
   * Методы, позволяющие:
     + ввести элементы массива с клавиатуры;
     + вывести элементы массива на экран;
     + отсортировать элементы каждой строки массива в порядке убывания.
   * Свойства:
     + возвращающее общее количество элементов в массиве (доступное только для чтения);
     + позволяющее увеличить значение всех элементов массива на скаляр (доступное только для записи).
8. Создать класс для работы сo строками. Разработать следующие элементы класса:
   * Поля:
     + StringBuilder Line ;
     + int n.
   * Конструктор, позволяющий создать строку из n символов.
   * Методы, позволяющие:
     + подсчитать количество пробелов в строке;
     + заменить в строке все прописные символы на строчные;
     + удалить из строки все знаки препинания.
   * Свойства:
     + возвращающее общее количество элементов в строке (доступное только для чтения);
     + позволяющее установить значение поля, в соответствии с введенным значением строки с клавиатуры, а также получить значение данного поля (доступно для чтения и записи)
9. Создать класс для работы с регулярными выражениями. Разработать следующие элементы класса:
   * Поля:
     + Regex r ;
     + string text ;
   * Методы, позволяющие:
     + определить, содержит ли текст фрагменты, соответствующие шаблону поля;
     + вывести на экран все фрагменты текста, соответствующие шаблону поля;
     + удалить из текста все фрагменты, соответствующие шаблону поля;
   * Свойства:
     + позволяющее установить или получить строковое поле класса (доступно для чтения и записи)
     + позволяющее установить или получить регулярное выражение, хранящееся в соответствующем поле класса (доступно для чтения и записи)
10. Создать класс для работы с датой. Разработать следующие элементы класса:
    * Поле DataTime data.
    * Конструкторы, позволяющие установить:
      + заданную дату
      + дату 1.01.2009
    * Методы, позволяющие:
      + вычислить дату предыдущего дня;
      + вычислить дату следующего дня;
      + определить сколько дней осталось до конца месяца.
    * Свойства:
      + позволяющее установить или получить значение поле класса (доступно для чтения и записи)
      + позволяющее определить год высокосным (доступно только для чтения)

## Контрольные вопросы.

# Классы: деструкторы, индексаторы, операции класса, операции преобразования типов

## Теоретические сведения.

### Деструкторы

В С# существует специальный вид метода, называемый деструктором, который вызывается сборщиком мусора непосредственно перед удалением объекта из памяти.

В деструкторе описываются действия, гарантирующие корректность последующего удаления объекта. Например, проверяется все ли ресурсы, используемые объектом, освобождены (файлы закрыты, *удаленное соединение* разорвано и т. п.).

Синтаксис деструктора:

[атрибуты] [extern] ~имя\_класса()

{тело\_деструктора}

Деструктор не имеет параметров, не возвращает значения и не требует указания спецификаторов доступа. Его имя совпадает с именем класса и предваряется *тильдой* ( ~ ), символизирующей обратные по отношению к конструктору действия. Тело деструктора представляет собой блок или просто точку с запятой. Если деструктор определен как внешний, то используется спецификатор extern. Пример работы деструктора:

class DemoArray

{

int[] MyArray;//закрытый массив

string name; //закрытое поле

public DemoArray(int size,int x, string name)//конструктор

{

MyArray = new int[size];

this.name = name;

for (int i=0;i<size; ++i) MyArray[i]=x;

}

public void Print ()//метод

{

Console.Write(name+ " : ");

foreach (int a in MyArray) Console.Write(a+" ");

Console.WriteLine();

}

public int LengthN //свойство

{

get { return MyArray.Length; }

}

~DemoArray()//деструктор

{

Console.WriteLine("сработал деструктор для объекта "+this.name);

}

}

class Program

{

static void Main()

{

DemoArray a= new DemoArray(5,2, "один");

a.Print();

DemoArray b = new DemoArray(6,1, "два");

b.Print();

a = b;

a.Print();

}

}

Обратите внимание на то, что деструкторы были вызваны автоматически. Дайте объяснение тому, в какой последовательности были вызваны деструкторы.

В общем случае применение деструкторов замедляет процесс сборки мусора. Поэтому создавать деструкторы следует только тогда, когда необходимо освободить какие-то ресурсы перед удалением объекта.

### Индексаторы

Индексатор представляет собой разновидность свойства и обычно применяется для организации доступа к скрытым полям класса по индексу, например, так же, как мы обращаемся к элементу массива. Синтаксис индексатора аналогичен синтаксису свойства:

[атрибуты] [спецификаторы] тип this [список параметров] // последние [ ] являются элементами синтаксиса

{

[get код\_доступа]

[set код\_доступа]

}

Спецификаторы аналогичны спецификаторам свойств и методов. Индексаторы чаще всего объявляются со спецификатором public, поскольку они входят в *интерфейс объекта*. Атрибуты и спецификаторы могут отсутствовать.

Код доступа представляет собой блоки операторов, которые выполняются при получении ( get ) или установке ( set ) значения некоторого элемента класса. Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, индексатор доступен только для чтения, если отсутствует часть get, индексатор доступен только для записи.

Список параметров содержит одно или несколько описаний индексов, по которым выполняется доступ к элементу. Чаще всего используется один индекс целого типа.

В качестве примера рассмотрим индексатор, который позволяет получить n-член последовательности Фиббоначи:

class DemoFib

{

public int this[int i] //индексатор, доступный только для чтения

{

get

{

if (i <=0) throw new Exception("недопустимое значение индекса");

else if (i==1 || i==2) return 1;

else

{int a=1, b=1, c;

for (int j=3; j<=i; ++j)

{

c=a+b;

a=b;

b=c;

}

return b;

}

}

}

}

class Program

{

static void Main()

{

Console.Write("n=");

int n=int.Parse(Console.ReadLine());

DemoFib a=new DemoFib();

try

{

Console.WriteLine("a[{0}]={1}",n,a[n]);

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

Индексаторы очень удобно применять для создания специализированных массивов, на работу с которыми накладываются какие-либо ограничения. Рассмотрим в качестве примера класс-массив, значения элементов которого находятся в диапазоне [0, 100]. Кроме того, при доступе к элементу проверяется, не вышел ли индекс за допустимые границы.

class DemoArray

{

int[] MyArray;//закрытый массив

public DemoArray(int size)//конструктор

{

MyArray = new int[size];

}

public int LengthArray //свойство, возвращающее размерность

{

get { return MyArray.Length; }

}

public int this[int i] //индексатор

{

get

{

if (i <0 || i >= MyArray.Length) throw new Exception("выход за границы массива");

else return MyArray[i];

}

set

{

if (i <0 || i >= MyArray.Length) throw new Exception("выход за границы массива");

else if (value >= 0 && value <= 100) MyArray[i] = value;

else throw new Exception("присваивается недопустимое значение");

}

}

}

class Program

{

static void Main()

{

DemoArray a = new DemoArray(10);

for (int i=0; i<a.LengthArray; i++)

{

a[i] = i \* i; // использование индексатора в режиме записи

Console.Write(a[i]+" ");// использование индексатора в режиме чтения

}

Console.WriteLine();

try

{

//a[10]=100;

//a[0]=200;

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

В блоке try содержатся две закомментированные команды. Посмотрите, как проведет себя программа, если убрать комментарии: вначале один, затем другой.

Язык С# допускает использование многомерных индексаторов. Они применяются для работы с многомерными массивами. Рассмотрим на примере предыдущую задачу при условии, что организуется двумерный массив.

class DemoArray

{

int[,] MyArray;//закрытый массив

int n, m;//закрытые поля: размерность массива

public DemoArray(int sizeN, int sizeM)//конструктор

{

MyArray = new int[sizeN, sizeM];

this.n = sizeN;

this.m = sizeM;

}

public int LengthN //свойство, возвращающее количество строк

{

get { return n; }

}

public int LengthM //свойство, возвращающее количество строк

{

get { return m; }

}

public int this[int i, int j] //индексатор

{

get

{

if (i < 0 || i >= n || j < 0 || j >= m) throw new Exception("выход за границы массива");

else return MyArray[i, j];

}

set

{

if (i < 0 || i >= n || j < 0 || j >= m) throw new Exception("выход за границы массива");

else if (value >= 0 && value <= 100) MyArray[i, j] = value;

else throw new Exception("присваивается недопустимое значение");

}

}

}

class Program

{

static void Main()

{

DemoArray a = new DemoArray(3, 3);

for (int i = 0; i < a.LengthN; i++,Console.WriteLine())

{

for (int j = 0; j < a.LengthM; j++)

{

a[i, j] = i \*j; // использование индексатора в режиме записи

Console.Write("{0,5}", a[i, j]);// использование индексатора в режиме чтения

}

}

Console.WriteLine();

try

{

//Console.WriteLine(a[3,3]);

//a[0,0]=200;

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

В блоке try содержатся две закомментированные команды. Посмотрите, как проведет себя программа если убрать комментарии - вначале один, затем другой.

### Операции класса

С# позволяет переопределить большинство операций так, чтобы при использовании их объектами конкретного класса выполнялись действия, отличные от стандартных. Это дает возможность применять объекты собственных типов данных в составе выражений, например:

newObject x, y, z;

...

z = x+y; // используется операция сложения, переопределенная для класса newObject

*Определение* собственных операций класса называют *перегрузкой операций*. *Перегрузка операций* обычно применяется для классов, для которых *семантика* операций делает программу более понятной. Если назначение *операции* интуитивно непонятно, перегружать такую операцию не рекомендуется.

*Операции* класса описываются с помощью методов специального вида, *синтаксис* которых выглядит следующим образом:

[ атрибуты] спецификаторы объявитель\_операции

{тело}

В качестве спецификаторов одновременно используются ключевые слова public и static. Кроме того, операцию можно объявить как внешнюю - extern. Объявление *операции* может выглядеть по-разному, в зависимости от того, что мы перегружаем: унарную или бинарную операцию.

При описании операций необходимо соблюдать следующие правила:

1. операция должна быть описана как открытый статический метод класса ( public static );
2. параметры в операцию должны передаваться по значению (то есть недопустимо использовать параметры ref и out );
3. сигнатуры всех операций класса должны различаться;
4. типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции).

#### Унарные операции

В классе можно переопределять следующие унарные операции: + - ! ~ ++ --, а также константы true и false. При этом, если была перегружена константа true, то должна быть перегружена и константа false, и наоборот.

Синтаксис объявителя унарной операции:

тип operator унарная\_операция (параметр)

Примеры заголовков унарных операций:

public static int operator + (DemoArray m)

public static DemoArray operator --(DemoArray m)

public static bool operator true (DemoArray m)

Параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. При этом операция должна возвращать:

1. для операций +, -, !, ~ величину любого типа;
2. для операций ++, -- величину типа класса, для которого она определяется;
3. для операций true и false величину типа bool.

Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата.

В качестве примера рассмотрим класс DemoArray, реализующий одномерный массив, в котором содержатся следующие функциональные элементы:

1. конструктор, позволяющий создать объект-массив заданной размерности;
2. конструктор, позволяющий инициализировать объект-массив обычным массивом;
3. свойство, возвращающее размерность массива;
4. индексатор, позволяющий просматривать и устанавливать значение по индексу в закрытом поле-массиве;
5. метод вывода закрытого поля-массива;
6. *перегрузка операции* унарный минус (все элементы массива меняют свое значение на противоположное);
7. *перегрузка операции* инкремента (все элементы массива увеличивают свое значение на 1);
8. перегруза констант true и false (при обращении к объекту будет возвращаться значение true, если все элементы массива положительные, в противном случае, будет возвращаться значение false).

class DemoArray

{

int[] MyArray;//закрытый массив

public DemoArray(int size)//конструктор 1

{

MyArray = new int[size];

}

public DemoArray(params int[] arr)//конструктор 2

{

MyArray = new int[arr.Length];

for (int i = 0; i < MyArray.Length; i++) MyArray[i] = arr[i];

}

public int LengthArray //свойство, возвращающее размерность

{

get { return MyArray.Length; }

}

public int this[int i] //индексатор

{

get

{

if (i < 0 || i >= MyArray.Length) throw new Exception("выход за границы массива");

return MyArray[i];

}

set

{

if (i < 0 || i >= MyArray.Length) throw new Exception("выход за границы массива");

else MyArray[i] = value;

}

}

public static DemoArray operator -(DemoArray x) //перегрузка операции унарный минус

{

DemoArray temp = new DemoArray(x.LengthArray);

for (int i = 0; i < x.LengthArray; ++i)

temp[i] = -x[i];

return temp;

}

public static DemoArray operator ++(DemoArray x) //перегрузка операции инкремента

{

DemoArray temp = new DemoArray(x.LengthArray);

for (int i = 0; i < x.LengthArray; ++i)

temp[i] = x[i]+1;

return temp;

}

public static bool operator true(DemoArray a) //перегрузка константы true

{

foreach (int i in a.MyArray)

{

if (i<0)

{

return false;

}

}

return true;

}

public static bool operator false(DemoArray a)//перегрузка константы false

{

foreach (int i in a.MyArray)

{

if (i>0)

{

return true;

}

}

return false;

}

public void Print(string name) //метод - выводит поле-массив на экран

{

Console.WriteLine(name + ": ");

for (int i = 0; i < MyArray.Length; i++)

Console.Write(MyArray[i] + " ");

Console.WriteLine();

}

}

class Program

{

static void Main()

{

try

{

DemoArray Mas = new DemoArray(1, -4, 3, -5, 0); //вызов конструктора 2

Mas.Print("Исходный массив");

Console.WriteLine("\nУнарный минус");

DemoArray newMas=-Mas; //применение операции унарного минуса

Mas.Print("Mассив Mas"); //обратите внимание, что создается новый объект и знаки меняются

newMas.Print("Массив newMas"); //только у нового массива

Console.WriteLine("\nОперация префиксного инкремента");

DemoArray Mas1=++Mas;

Mas.Print("Mассив Mas");

Mas1.Print("Mассив Mas1=++Mas");

Console.WriteLine("\nОперация постфиксного инкремента");

DemoArray Mas2=Mas++;

Mas.Print("Mассив Mas");

Mas2.Print("Mассив Mas2=Mas++");

if (Mas)

Console.WriteLine("\nВ массиве все элементы положительные\n");

else Console.WriteLine("\nВ массиве есть не положительные элементы\n");

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

Добавьте в класс DemoArray переопределение унарного плюса (все элементы массива преобразуются в положительные), унарного декремента (все элементы массива уменьшаются на единицу)

#### Бинарные операции

При разработке класса можно перегрузить следующие бинарные операции: + - \* / % & | ^ << >> == != < > <= >=. Обратите внимание, **операций присваивания** в этом списке **нет**.

Синтаксис объявителя бинарной операции:

тип operator бинарная\_операция (параметр1, параметр 2)

Примеры заголовков бинарных операций:

public static DemoArray operator + (DemoArray a, DemoArray b)

public static bool operator == (DemoArray a, DemoArray b)

При переопределении бинарных операций нужно учитывать следующие правила:

1. Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется.
2. Операция может возвращать величину любого типа.
3. Операции отношений определяются только парами и обычно возвращают логическое значение. Чаще всего переопределяются операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение значения некоторых полей объектов, а не ссылок на объект. Для того чтобы переопределить операции отношений, требуется знание стандартных интерфейсов, которые будут рассматриваться чуть позже.

В качестве примера вернемся к классу DemoArray, реализующему одномерный массив, и добавим в него две версии *переопределенной операции* +:

* Вариант 1: добавляет к каждому элементу массива заданное число;
* Вариант 2: поэлементно складывает два массива

class DemoArray

{

...

public static DemoArray operator +(DemoArray x, int a) //вариант 1

{

DemoArray temp = new DemoArray(x.LengthArray);

for (int i = 0; i < x.LengthArray; ++i)

temp[i]=x[i]+a;

return temp;

}

public static DemoArray operator +(DemoArray x, DemoArray y) //вариант 2

{

if (x.LengthArray == y.LengthArray)

{

DemoArray temp = new DemoArray(x.LengthArray);

for (int i = 0; i < x.LengthArray; ++i)

temp[i] = x[i] + y[i];

return temp;

}

else throw new Exception("несоответствие размерностей");

}

}

class Program

{

static void Main()

{

try

{

DemoArray a = new DemoArray(1, -4, 3, -5, 0);

a.Print("Массива a");

DemoArray b=a+10;

b.Print("\nМассива b");

DemoArray c = a+b;

c.Print("\nМассива c");

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

#### Операции преобразования типов

Операции преобразования типов обеспечивают возможность явного и *неявного преобразования* между *пользовательскими типами данных*. Синтаксис объявителя операции преобразования типов выглядит следующим образом:

explicit operator целевой\_тип (параметр) //явное преобразование

implicit operator целевой\_тип (параметр) //неявное преобразование

Эти операции выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого выполняется преобразование.

*Неявное преобразование* выполняется автоматически в следующих ситуациях:

1. при присваивании объекта переменной целевого типа;
2. при использовании объекта в выражении, содержащем переменные целевого типа;
3. при передаче объекта в метод параметра целевого типа;
4. при явном приведении типа.

Явное преобразование выполняется при использовании операции приведения типа.

При определении операции преобразования типа следует учитывать следующие особенности:

1. тип возвращаемого значения (целевой\_тип) включается в сигнатуру объявителя операции;
2. ключевые слова explicit и *implicit* не включаются в сигнатуру объявителя операции.

Следовательно, для одного и того класса нельзя определить одновременно и явную, и неявную версию. Однако, т.к. *неявное преобразование* автоматически выполнятся при явном использовании операции приведения типа, то достаточно разработать только неявную версию операции преобразования типа.

В качестве примера вернемся к классу DemoArray, реализующему одномерный массив, и добавим в него неявную версию переопределения типа DemoArray в тип одномерный массив и наоборот:

class DemoArray

{

…

public static implicit operator DemoArray (int []a) //неявное преобразование типа int [] в DemoArray

{

return new DemoArray(a);

}

public static implicit operator int [](DemoArray a) //неявное преобразование типа DemoArray в int []

{

int []temp=new int[a.LengthArray];

for (int i = 0; i < a.LengthArray; ++i)

temp[i] = a[i];

return temp;

}

}

class Program

{

static void arrayPrint(string name, int[]a) //метод, который позволяет вывести на экран одномерный массив

{

Console.WriteLine(name + ": ");

for (int i = 0; i < a.Length; i++)

Console.Write(a[i] + " ");

Console.WriteLine();

}

static void Main()

{

try

{

DemoArray a = new DemoArray(1, -4, 3, -5, 0);

int []mas1=a; //неявное преобразование типа DemoArray в int []

int []mas2=(int []) a; //явное преобразование типа DemoArray в int []

DemoArray b1 =mas1; //неявное преобразование типа int [] в DemoArray

DemoArray b2 =(DemoArray)mas2; //явное преобразование типа int [] в DemoArray

//изменение значений

mas1[0]=0; mas2[0]=-1;

b1[0]=100; b2[0]=-100;

//вывод на экран

a.Print("Массива a");

arrayPrint("Maccив mas1", mas1);

arrayPrint("Maccив mas2", mas2);

b1.Print("Массива b1");

b2.Print("Массива b2");

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

## Задания на лабораторную работу.

*Замечания*.

1. *Во всех задачах данного раздела подразумевается, что исходная информация хранится в текстовом файле input.txt, каждая строка которого содержит полную информацию о некотором объекте, результирующая информация должна быть записана в файл output.txt.*
2. *Для хранения данных внутри программы организовать массив структур.*
3. *В типе структура реализуется метод CompareTo интерфейса IComparable, перегружается метод ToString базового класса object и необходимые операции отношения, поля данных и дополнительные методы продумайте самостоятельно.*
4. На основе данных входного файла составить список студентов группы, включив следующие данные: ФИО, год рождения, домашний адрес, какую школу окончил. Вывести в новый файл информацию о студентах, окончивших заданную школу, отсортировав их по году рождения.
5. На основе данных входного файла составить список студентов группы, включив следующие данные: ФИО, номер группы, результаты сдачи трех экзаменов. Вывести в новый файл информацию о студентах, успешно сдавших сессию, отсортировав по номеру группы.
6. На основе данных входного файла составить багажную ведомость камеры хранения, включив следующие данные: ФИО пассажира, количество вещей, общий вес вещей. Вывести в новый файл информацию о тех пассажирах, средний вес багажа которых превышает заданный, отсортировав их по количеству вещей, сданных в камеру хранения.
7. На основе данных входного файла составить автомобильную ведомость, включив следующие данные: марка автомобиля, номер автомобиля, фамилия его владельца, год приобретения, пробег. Вывести в новый файл информацию об автомобилях, выпущенных ранее определенного года, отсортировав их по пробегу.
8. На основе данных входного файла составить список сотрудников учреждения, включив следующие данные: ФИО, год принятия на работу, должность, зарплата, рабочий стаж. Вывести в новый файл информацию о сотрудниках, имеющих зарплату ниже определенного уровня, отсортировав их по рабочему стажу.
9. На основе данных входного файла составить инвентарную ведомость склада, включив следующие данные: вид продукции, стоимость, сорт, количество. Вывести в новый файл информацию о той продукции, количество которой менее заданной величины, отсортировав ее по количеству продукции на складе.
10. На основе данных входного файла составить инвентарную ведомость игрушек, включив следующие данные: название игрушки, ее стоимость (в руб.), возрастные границы детей, для которых предназначена игрушка. Вывести в новый файл информацию о тех игрушках, которые предназначены для детей от N до M лет, отсортировав их по стоимости.
11. На основе данных входного файла составить список вкладчиков банка, включив следующие данные: ФИО, № счета, сумма, год открытия счета. Вывести в новый файл информацию о тех вкладчиках, которые открыли вклад в текущем году, отсортировав их по сумме вклада.
12. На основе данных входного файла составить список студентов, включающий фамилию, факультет, курс, группу, 5 оценок. Вывести в новый файл информацию о тех студентах, которые имеют хотя бы одну двойку, отсортировав их по курсу.
13. На основе данных входного файла составить список студентов, включающий ФИО, курс, группу, результат забега. Вывести в новый файл информацию о студентах, показавших три лучших результата в забеге. Если окажется, что некоторые студенты получили такие же высокие результаты, то добавить их к списку победителей.

## Контрольные вопросы.

# Наследование

## Теоретические сведения.

Управлять большим количеством разрозненных классов достаточно сложно. С этой проблемой можно справиться путем упорядочивания и ранжирования классов, то есть объединяя общие для нескольких классов свойства в одном классе и используя его в качестве базового.

Эту возможность предоставляет механизм *наследования*, который является мощнейшим инструментом *ООП*. Он позволяет строить иерархии, в которых классы-потомки получают свойства классов-предков и могут дополнять их или изменять. Таким образом, *наследование* обеспечивает важную возможность многократного использования кода.

Классы, расположенные ближе к началу иерархии, объединяют в себе общие черты для всех нижележащих классов. *По* мере продвижения вниз *по* иерархии классы приобретают все больше конкретных особенностей.

### Описание класса-потомка

*Класс* в C# может иметь *произвольное* количество потомков и только одного предка. При описании класса имя его предка записывается в заголовке класса после двоеточия. Если имя предка не указано, предком считается *базовый класс* всей иерархии System.Object:

[ атрибуты ] [ спецификаторы ] class имя\_класса [ : предки ]

тело класса

Примечание

*Обратите внимание, что слово "предки" присутствует в описании класса во множественном числе, хотя класс может иметь только одного предка. Причина в том, что класс наряду с единственным предком может наследовать от интерфейсов — специального вида классов, не имеющих реализации. Интерфейсы рассматриваются на следующей лекции*.

Рассмотрим *наследование*  классов на примере. Ранее был описан *класс* Monster, моделирующий персонаж компьютерной игры. Допустим, нам требуется ввести в игру еще один тип персонажей, который должен обладать свойствами объекта Monster, а кроме того, уметь думать. Будет логично сделать новый *объект* потомком объекта Monster.

using System;

namespace ConsoleApplication1

{

class Monster

{

...

}

class Daemon : Monster

{

public Daemon()

{

brain = 1;

}

public Daemon( string name, int brain ) : base( name ) // 1

{

this.brain = brain;

}

public Daemon( int health, int ammo, string name, int brain )

: base( health, ammo, name ) // 2

{

this.brain = brain;

}

new public void Passport() // 3

{

Console.WriteLine(

"Daemon {0} \t health = {1} ammo = {2} brain = {3}",

Name, Health, Ammo, brain );

}

public void Think() // 4

{

Console.Write( Name + " is" );

for ( int i = 0; i < brain; ++i ) Console.Write( " thinking" );

Console.WriteLine( "..." );

}

int brain; // закрытое поле

}

class Class1

{ static void Main()

{

Daemon Dima = new Daemon( "Dima", 3 ); // 5

Dima.Passport(); // 6

Dima.Think(); // 7

Dima.Health -= 10; // 8

Dima.Passport();

}

}

}

Листинг 1. Класс Daemon, потомок класса Monster

В классе Daemon введены закрытое *поле* brain и метод Think, определены собственные конструкторы, а также переопределен метод Passport. Все поля и свойства класса Monster наследуются в классе Daemon.

Результат работы программы:

Daemon Dima health = 100 ammo = 100 brain = 3

Dima is thinking thinking thinking...

Daemon Dima health = 90 ammo = 100 brain = 3

Как видите, экземпляр класса Daemon с одинаковой легкостью использует как собственные (*операторы* 5–7), так и унаследованные (оператор 8) элементы класса. Рассмотрим общие правила наследования.

*Конструкторы* не наследуются, поэтому производный *класс* должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными ниже правилами.

Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса без параметров.

Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса. Таким образом, каждый конструктор инициализирует свою часть объекта.

Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации. Вызов выполняется с помощью ключевого слова base. Вызывается та версия конструктора, список параметров которой соответствует списку аргументов, указанных после слова base.

*Поля*, *методы* и *свойства* класса наследуются, поэтому при желании заменить элемент базового класса новым элементом следует явным образом указать компилятору свое намерение с помощью ключевого слова new. В [листинге 1](https://www.intuit.ru/studies/courses/629/485/lecture/11013?page=1#example.8.1) таким образом переопределен метод вывода информации об объекте Passport. Метод Passport класса Daemon замещает соответствующий метод базового класса, однако возможность доступа к методу базового класса из метода производного класса сохраняется. Для этого перед вызовом метода указывается все то же волшебное *слово* base, например:

base.Passport();

Элементы базового класса, определенные как private, в производном классе недоступны. Поэтому в методе Passport для доступа к полям name, health и ammo пришлось использовать соответствующие свойства базового класса. Другое решение заключается в том, чтобы определить эти поля со спецификатором protected, в этом случае они будут доступны методам всех классов, производных от Monster. Оба решения имеют свои достоинства и недостатки.

Во *время выполнения* программы объекты хранятся в отдельных переменных, массивах или других коллекциях. Во многих случаях удобно *оперировать объектами одной иерархии единообразно*, то есть использовать один и тот же программный код для работы с экземплярами разных классов. Это возможно благодаря тому, что *объекту базового класса можно присвоить объект производного класса*.

Попробуем описать *массив* объектов базового класса и занести туда объекты производного класса. В [листинге 2](https://www.intuit.ru/studies/courses/629/485/lecture/11013?page=1#example.8.2) в массиве типа Monster хранятся два объекта типа Monster и один — типа Daemon.

using System;

namespace ConsoleApplication1

{

class Monster

{

...

}

class Daemon : Monster

{

... //

}

class Class1

{ static void Main()

{

const int n = 3;

Monster[] stado = new Monster[n];

stado[0] = new Monster( "Monia" );

stado[1] = new Monster( "Monk" );

stado[2] = new Daemon ( "Dimon", 3 );

foreach ( Monster elem in stado ) elem.Passport(); // 1

for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0; // 2

Console.WriteLine();

foreach ( Monster elem in stado ) elem.Passport(); // 3

}

}

}

Листинг 2. Массив объектов разных типов

Результат работы программы:

Monster Monia health = 100 ammo = 100

Monster Monk health = 100 ammo = 100

Monster Dimon health = 100 ammo = 100

Monster Monia health = 100 ammo = 0

Monster Monk health = 100 ammo = 0

Monster Dimon health = 100 ammo = 0

Результат радует нас только частично: *объект* типа Daemon действительно можно поместить в *массив*, состоящий из элементов типа Monster, но для него вызываются только методы и свойства, унаследованные от предка. Это устраивает нас в операторе 2, а в операторах 1 и 3 хотелось бы, чтобы вызывался метод Passport, переопределенный в потомке.

Итак, присваивать объекту базового класса *объект* производного класса можно, но вызываются для него только методы и свойства, определенные в базовом классе. Иными словами, возможность доступа к элементам класса определяется типом ссылки, а не типом объекта, на который она указывает.

Это и понятно: ведь *компилятор* должен еще до выполнения программы решить, какой метод вызывать, и вставить в код фрагмент, передающий управление на этот метод (этот процесс называется *ранним связыванием* ). При этом *компилятор* может руководствоваться только типом переменной, для которой вызывается метод или свойство (например, stado[i].Ammo ). То, что в этой переменной в разные моменты времени могут находиться ссылки на объекты разных типов, *компилятор* учесть не может.

Следовательно, если мы хотим, чтобы вызываемые методы соответствовали типу объекта, необходимо отложить процесс связывания до этапа выполнения программы, а точнее — до момента вызова метода, когда уже точно известно, на *объект* какого типа указывает *ссылка*. Такой механизм в C# есть — он называется *поздним связыванием* и реализуется с помощью так называемых виртуальных методов.

### Виртуальные методы

При раннем связывании *программа*, готовая для выполнения, представляет собой структуру, логика выполнения которой жестко определена. Если же требуется, чтобы решение о том, какой из одноименных методов разных объектов иерархии использовать, принималось в зависимости от конкретного объекта, для которого выполняется вызов, то заранее жестко связывать эти методы с остальной частью кода нельзя.

Следовательно, надо каким-то образом дать знать компилятору, что эти методы будут обрабатываться *по*-другому. Для этого в C# существует ключевое *слово* virtual. Оно записывается в заголовке метода базового класса, например:

virtual public void Passport() ...

Объявление метода виртуальным означает, что все ссылки на этот метод будут разрешаться в момент его вызова во *время выполнения*программы. Этот механизм называется *поздним связыванием*.

Для его реализации необходимо, чтобы адреса виртуальных методов хранились там, где ими можно будет в любой момент воспользоваться, поэтому *компилятор* формирует для этих методов *таблицу виртуальных методов* (*Virtual* *Method* *Table*, *VMT*). В нее записываются адреса виртуальных методов (в том числе унаследованных) в порядке описания в классе. Для каждого класса создается одна *таблица*.

Каждый *объект* во *время выполнения* должен иметь *доступ* к *VMT*. *Связь* экземпляра объекта с *VMT* устанавливается с помощью специального кода, автоматически помещаемого компилятором в *конструктор* объекта.

Если в производном классе требуется переопределить *виртуальный метод*, используется ключевое *слово* override, например:

override public void Passport() ...

Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса.

Добавим в [листинге 2](https://www.intuit.ru/studies/courses/629/485/lecture/11013?page=1#example.8.2) два слова — virtual и override — в описания методов Passport соответственно базового и производного классов ([листинг 3](https://www.intuit.ru/studies/courses/629/485/lecture/11013?page=1#example.8.3)).

using System;

namespace ConsoleApplication1

{

class Monster

{

...

virtual public void Passport()

{

Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",

name, health, ammo );

}

...

}

class Daemon : Monster

{

...

override public void Passport()

{

Console.WriteLine(

"Daemon {0} \t health = {1} ammo = {2} brain = {3}",

Name, Health, Ammo, brain );

}

...

}

class Class1

{ static void Main()

{

const int n = 3;

Monster[] stado = new Monster[n];

stado[0] = new Monster( "Monia" );

stado[1] = new Monster( "Monk" );

stado[2] = new Daemon ( "Dimon", 3 );

foreach ( Monster elem in stado ) elem.Passport();

for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0;

Console.WriteLine();

foreach ( Monster elem in stado ) elem.Passport();

}

}

}

Листинг 3. Виртуальные методы

Результат работы программы:

Monster Monia health = 100 ammo = 100

Monster Monk health = 100 ammo = 100

Daemon Dimon health = 100 ammo = 100 brain = 3

Monster Monia health = 100 ammo = 0

Monster Monk health = 100 ammo = 0

Daemon Dimon health = 100 ammo = 0 brain = 3

Теперь в циклах 1 и 3 вызывается метод Passport, соответствующий типу объекта, помещенного в *массив*.

*Виртуальные методы базового класса определяют интерфейс всей иерархии*. Этот *интерфейс* может расширяться в потомках за счет добавления новых виртуальных методов. Переопределять *виртуальный метод* в каждом из потомков не обязательно: если он выполняет устраивающие потомка действия, метод наследуется.

С помощью виртуальных методов реализуется один из основных принципов объектно-ориентированного программирования — *полиморфизм*. Это *слово* в переводе с греческого означает "много форм", что в данном случае означает "один вызов — много методов".

Виртуальные методы незаменимы и при передаче объектов в методы в качестве параметров. В параметрах метода описывается *объект*базового типа, а при вызове в нее передается *объект* производного класса. Виртуальные методы, вызываемые для объекта из метода, будут соответствовать типу аргумента, а не параметра.

**Абстрактные классы**

При создании иерархии объектов для исключения повторяющегося кода часто бывает логично выделить их общие свойства в один родительский *класс*. При этом может оказаться, что создавать экземпляры такого класса не имеет смысла, потому что никакие реальные объекты им не соответствуют. Такие классы называют абстрактными.

*Абстрактный класс служит только для порождения потомков*. Как правило, в нем задается набор методов, которые каждый из потомков будет реализовывать *по*-своему. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах.

*Абстрактный класс задает интерфейс для всей иерархии*, при этом методам класса может не соответствовать никаких конкретных действий. В этом случае методы имеют пустое тело и объявляются со спецификатором abstract.

Если в классе есть хотя бы один *абстрактный метод*, весь *класс* также должен быть описан как абстрактный, например:

abstract class Spirit

{

public abstract void Passport();

}

class Monster : Spirit

{

...

override public void Passport()

{

Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",

name, health, ammo );

}

...

}

class Daemon : Monster

{

...

override public void Passport()

{

Console.WriteLine(

"Daemon {0} \t health = {1} ammo = {2} brain = {3}",

Name, Health, Ammo, brain );

}

... // полный текст этих классов приведен в главе 12

}

Абстрактные классы используются при работе со структурами данных, предназначенными для хранения объектов одной иерархии, и в качестве параметров методов.

**Бесплодные классы**

В C# есть ключевое *слово* sealed, позволяющее описать *класс*, от которого, в противоположность абстрактному, наследовать запрещается:

sealed class Spirit

{

...

}

// class Monster : Spirit { ... } ошибка!

Большинство встроенных типов данных описано как sealed. Если необходимо использовать функциональность бесплодного класса, применяется не *наследование*, а *вложение*, или *включение*: в классе описывается *поле* соответствующего типа.

Вложение классов, когда один *класс* включает в себя поля, являющиеся классами, является *альтернативой наследованию* при проектировании. Например, если есть *объект* "двигатель", а требуется описать *объект* "самолет", логично сделать двигатель полем этого объекта, а не его предком.

**Виды взаимоотношений между классами**

Механизм наследования классов предоставляет программисту богатейшие возможности организации кода и его многократного использования. Выбор наиболее подходящих средств для целей конкретного проекта основывается на знании механизма их работы и взаимодействия.

Существует следующая классификация форм наследования. Форма наследования определяет, с какой целью оно используется. Порождение дочернего класса может быть выполнено *по* следующим причинам.

* **Специализация**. Класс-наследник является специализированной формой родительского класса — в наследнике просто переопределяются методы.
* **Спецификация**. *Дочерний класс* реализует поведение, описанное в родительском классе. В С# эта форма реализуется наследованием от абстрактного класса.
* **Конструирование**. Класс-наследник использует методы базового класса, но не является его подтипом.
* **Расширение**. В *класс-потомок* добавляют новые методы, расширяя поведение родительского класса.
* **Обобщение**. *Дочерний класс* обобщает поведение базового класса. Обычно такое наследование используется в тех случаях, когда изменить поведение базового класса невозможно (например, базовый класс является библиотечным классом).
* **Ограничение**. Класс-наследник ограничивает поведение родительского класса.
* **Варьирование**. Базовый класс и *класс-потомок* являются вариациями на одну тему, однако связь "класс-подкласс" произвольна, например, "квадрат-прямоугольник" или "прямоугольник-квадрат". Эта форма фактически не отличается от "конструирования", так как класс-наследник, очевидно, "использует методы базового класса, но не является его подтипом".
* **Комбинирование**. *Дочерний класс* наследует черты нескольких классов — это множественное наследование (в C# не используется, поскольку множественное наследование запрещено, а наследование от нескольких интерфейсов имеет иной смысл).

Альтернативой наследованию при проектировании классов является **вложение**, когда один *класс* включает в себя поля, являющиеся классами. Например, если есть *класс* "двигатель", а требуется описать *класс* "самолет", логично сделать двигатель полем этого класса, а не его предком. Вложение представляет отношения классов "Y содержит X" или "Y реализуется посредством Х" и обычно реализуется с помощью модели "включение-делегирование", которая иллюстрируется в [листинге 4](https://www.intuit.ru/studies/courses/629/485/lecture/11013?page=2#example.8.4).

using System;

namespace ConsoleApplication1

{

class Двигатель

{ public void Запуск()

{

Console.WriteLine( "вжжжж!!" );

}

}

class Самолет

{ public Самолет()

{

левый = new Двигатель();

правый = new Двигатель();

}

public void Запустить\_двигатели()

{

левый.Запуск();

правый.Запуск();

}

Двигатель левый, правый;

}

class Class1

{ static void Main()

{

Самолет АН24\_1 = new Самолет();

АН24\_1.Запустить\_двигатели();

}

}

}

Листинг 8.4. Модель включения-делегирования

Результат работы программы:

вжжжж!!

вжжжж!!

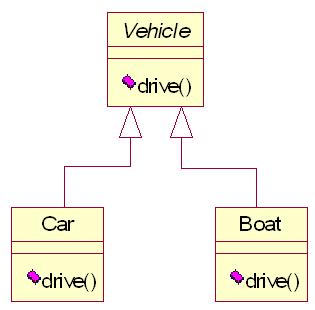
В методе Запустить\_двигатели *запрос* на *запуск* двигателей передается, или, как принято говорить, *делегируется* вложенному классу.

В процессе проектирования объектно-ориентированных программ для описания различного рода взаимоотношений классов и объектов часто используется *UML*.

**UML — Unified Modeling Language** — является языком для специфицирования, визуализации, конструирования и документирования программных продуктов, а также используется в бизнес-моделировании и моделировании любых иных (не программных) систем.

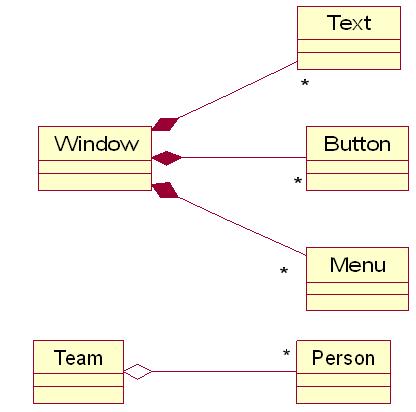
*UML* позволяет задавать различные аспекты представления системы, например, поведение системы с точки зрения ее внешних пользователей, внутреннюю структуру и различные виды взаимодействия элементов системы. Взаимоотношения классов отображают на диаграмме классов.

*Отношение наследования* изображается на диаграмме классов в виде незакрашенного треугольника, направленного к базовому классу (предку). Пример изображения наследования с помощью *диаграммы классов* приведен на [рис. 1](https://www.intuit.ru/studies/courses/629/485/lecture/11013?page=2#image.8.1).



**Рис. 1.**Пример диаграммы UML для наследования

Вложение классов изображается в виде закрашенного или незакрашенного ромба, направленного к владельцу части. Другой конец линии может быть снабжен цифрой, указывающей количество вложенных частей, или звездочкой, если таких частей может быть *произвольное* количество. Примеры приведены на [рис. 2](https://www.intuit.ru/studies/courses/629/485/lecture/11013?page=2#image.8.2).



**Рис. 2.**Примеры диаграмм UML для вложения классов

При проектировании классов следует выбирать модель, наиболее точно отражающую смысл взаимоотношений классов, например, моделируемых объектов *предметной области*.

**Класс object**

Корневой *класс* System.Object всей иерархии объектов .*NET*, называемый в C# object, обеспечивает всех наследников несколькими важными методами. *Производные* классы могут использовать эти методы непосредственно или переопределять их.

*Класс* object часто используется и непосредственно при описании типа параметров методов для придания им общности, а также для хранения ссылок на объекты различного типа — таким образом реализуется *полиморфизм*.

#### Открытые методы класса System.Object :.

Метод Equals с одним параметром возвращает значение true, если параметр и вызывающий объект ссылаются на одну и ту же область памяти. Синтаксис:

public virtual bool Equals(object obj);

Метод Equals с двумя параметрами возвращает значение true, если оба параметра ссылаются на одну и ту же область памяти. Синтаксис:

public static bool Equals(object ob1, object ob2);

Метод GetHashCode формирует хэш-код объекта и возвращает число, однозначно идентифицирующее объект. Это число используется в различных структурах и алгоритмах библиотеки. Если переопределяется метод Equals, необходимо перегрузить и метод GetHashCode. Синтаксис:

public virtual int GetHashCode();

Метод GetType возвращает текущий полиморфный тип объекта, то есть не тип ссылки, а тип объекта, на который она в данный момент указывает. Возвращаемое значение имеет тип Type. Это абстрактный базовый класс иерархии, использующийся для получения информации о типах во время выполнения. Синтаксис:

public Type GetType();

Метод ReferenceEquals возвращает значение true, если оба параметра ссылаются на одну и ту же область памяти. Синтаксис:

public static bool ReferenceEquals(object ob1, object ob2);

Метод ToString по умолчанию возвращает для ссылочных типов полное имя класса в виде строки, а для значимых — значение величины, преобразованное в строку. Этот метод переопределяют для того, чтобы можно было выводить информацию о состоянии объекта. Синтаксис:

public virtual string ToString()

В *производных объектах* эти методы часто переопределяют. Например, можно переопределить метод Equals для того, чтобы задать собственные критерии сравнения объектов.

Пример применения и переопределения методов класса object для класса Monster приведен в [листинге 5](https://www.intuit.ru/studies/courses/629/485/lecture/11013?page=3#example.8.5).

using System;

namespace ConsoleApplication1

{

class Monster

{

public Monster( int health, int ammo, string name )

{

this.health = health;

this.ammo = ammo;

this.name = name;

}

public override bool Equals( object obj )

{

if ( obj == null || GetType() != obj.GetType() ) return false;

Monster temp = (Monster) obj;

return health == temp.health &&

ammo == temp.ammo &&

name == temp.name;

}

public override int GetHashCode()

{

return name.GetHashCode();

}

public override string ToString()

{

return string.Format( "Monster {0} \t health = {1} ammo = {2}",

name, health, ammo );

}

string name;

int health, ammo;

}

class Class1

{ static void Main()

{

Monster X = new Monster( 80, 80, "Вася" );

Monster Y = new Monster( 80, 80, "Вася" );

Monster Z = X;

if ( X == Y ) Console.WriteLine(" X == Y ");

else Console.WriteLine(" X != Y ");

if ( X == Z ) Console.WriteLine(" X == Z ");

else Console.WriteLine(" X != Z ");

if ( X.Equals(Y) ) Console.WriteLine( " X Equals Y " );

else Console.WriteLine( " X not Equals Y " );

Console.WriteLine(X.GetType());

}

}

}

Листинг 5. Перегрузка методов класса object

Результат работы программы:

X != Y

X == Z

X Equals Y

ConsoleApplication1.Monster

Анализируя результат работы программы, можно увидеть, что в *операции* сравнения на *равенство* сравниваются ссылки, а в перегруженном методе Equals — значения. Для концептуального единства можно переопределить и *операции отношения*.

## Задания на лабораторную работу.

Создать абстрактный класс Pair (пара значений) с виртуальными арифметическими операциями и методом вывода на экран. На его основе реализовать классы Money (деньги) и Complex (комплексное число).

1. В классе Money денежная сумма представляется в виде двух целых, в которых хранятся рубли и копейки соответственно. При выводе части числа снабжаются словами "руб." и "коп.".
2. В классе Complex предусмотреть при выводе символ мнимой части (i).
3. Создать класс Series (набор), содержащий список (или массив) объектов этих классов в динамической памяти. Предусмотреть возможность вывода объектов списка. Написать демонстрационную программу, в которой будут использоваться все методы классов.

## Контрольные вопросы.

1. Для чего используется наследование?
2. Опишите синтаксис производного класса. Какие спецификаторы доступа применяются в иерархиях?
3. Как вызвать метод базового класса из производного?
4. Опишите порядок вызова конструкторов базовых классов при работе конструктора производного класса.
5. Какие ключевые слова используется при переопределении методов базового класса в производном?
6. Опишите механизмы раннего и позднего связывания.
7. Опишите процесс вызова виртуального метода. Для чего применяются виртуальные методы?
8. Все ли методы следует описывать как виртуальные?
9. Для чего используются абстрактные классы?
10. Какие методы и операции класса object часто перегружают в его потомках?
11. Назовите альтернативы наследованию классов.
12. Изучите по справочной системе свойства и методы класса object.
13. Изучите разделы стандарта C#, касающиеся наследования.

# Коллекции

## Теоретические сведения.

### Коллекции

В С# под *коллекцией*понимается некоторая группа объектов. Коллекции упрощают реализацию многих задач программирования, предлагая уже готовые решения для построения структур данных. Все коллекции разработаны на основе четко определенных интерфейсов, поэтому стандартизируют способ обработки группы объектов. Среда .NET Framework поддерживает три основных *типа коллекций*: общего назначения, специализированные и ориентированные на побитовую организацию данных.

*Коллекции общего* *назначения* определены в пространстве имен System.Collection и реализуют такие структуры данных, как стеки, очереди, динамические массивы, словари (хеш-таблицы, предназначенные для хранения пар ключ/значение), отсортированный список для хранения пар ключ/значение. Коллекции общего назначения работают с данными типа object, поэтому их можно использовать для хранения данных любого типа.

*Коллекции специального назначения* определены в пространстве имен System.Collection.Specialized и ориентированы на обработку данных конкретного типа или на обработку данных уникальным способом. Например, существуют специализированные коллекции, предназначенные только для обработки строк.

В пространстве имен System.Collection определена единственная коллекция, ориентированная на побитовую организацию данных, которая служит для хранения групп битов и поддерживает такой набор операций, который не характерен для коллекций других типов.

#### Коллекции общего назначения

Классы коллекций общего назначения:

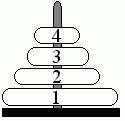
|  |  |
| --- | --- |
| Класс | Описание |
| Stack | Стек - частный случай однонаправленного списка, действующий по принципу: последним пришел - первым вышел |
| Queue | Очередь - частный случай однонаправленного списка, действующего по принципу:первым пришел - первым вышел |
| *ArrayList* | *Динамический массив*, т.е. массив который при необходимости может увеличивать свой размер |
| Hashtable | Хеш-таблица для пар ключ/значение |
| SortedList | Отсортированный список пар ключ/значение |

Рассмотрим данные коллекции более подробно.

Абстрактный тип данных (АТД) *список* - это последовательность элементов a1, а2,...,аn(n >= 0) одного типа. Количество элементов *n* называется *длиной списка*. Если n > 0, то а1 называется *первым элементом списка*, а аn - *последним элементом списка*. В случае n = 0 имеем *пустой список*, который не содержит элементов. Важное *свойство списка* заключается в том, что его элементы линейно упорядочены в соответствии с их позицией в списке. Так элемент аi *предшествует* аi+1 для i=1, 2, …n-1 и аi *следует* за аi-1 для i=2, ...n. Список называется *однонаправленным*, если каждый элемент списка содержит ссылку на следующий элемент. Если каждый элемент списка содержит две ссылки (одну на следующий элемент в списке, вторую - на предыдущий элемент), то такой список называется *двунаправленным*(*двусвязным*). А если последний элемент связать указателем с первым, то получится кольцевой список.

##### Класс Stack

АТД стек - это частный случай однонаправленного списка, добавление элементов в который и выборка элементов из которого выполняются с одного конца, называемого вершиной стека (головой - head ). При выборке элемент исключается из стека. Другие операции со стеком не определены. Говорят, что стек реализует принцип обслуживания LIFO (last in - fist out, последним пришел - первым вышел). Стек проще всего представить себе в виде пирамиды, на которую надевают кольца.



Достать первое кольцо можно только после того, как будут сняты все верхние кольца.

В С# реализацию АТД стек представляет класс Stack, который реализует интерфейсы *ICollection*, *IEnumerable* и *ICloneable*. Stack - это динамическая коллекция, размер которой изменяется.

В классе Stack определены следующие конструкторы:

public Stack(); //создает пустой стек, начальная вместимость которого равна 10

public Stack(int capacity); // создает пустой стек, начальная вместимость которого равна capacity

public Stack(ICollection c); //создает стек, который содержит элементы коллекции, заданной

//параметром с, и аналогичной (аналогичной - с чем?) вместимостью

Кроме методов, определенных в интерфейсах, реализуемых классом Stack, в этом классе определены *собственные методы*:

|  |  |
| --- | --- |
| Метод | Описание |
| public virtual bool Contains(object v) | Возвращает значение true, если объект v содержится в вызывающем стеке, в противном случае возвращает значение false. |
| public virtual void Clear() | Устанавливает свойство Сount равным нулю, тем самым очищая стек. |
| public virtual object *Peek*() | Возвращает элемент, расположенный в вершине стека, но не извлекая его из стека |
| public virtual object Pop() | Возвращает элемент, расположенный в вершине стека, и извлекает его из стека |
| public virtual void Push(object v) | Помещает объект v в стек |
| public virtual object[] ToArray() | Возвращает массив, который содержит копии элементов вызывающего стека |

Рассмотрим несколько примеров использования стека.

**Пример 1**.Для заданного значения n запишем в стек все числа от 1 до n, а затем извлечем из стека:

using System;

using System.Collections;

namespace ConsoleApplication

{

class Program

{

public static void Main()

{

Console.Write("n= ");

int n=int.Parse(Console.ReadLine());

Stack intStack = new Stack();

for (int i = 1; i <= n; i++)

intStack.Push(i);

Console.WriteLine("Размерность стека " + intStack.Count);

Console.WriteLine("Верхний элемент стека = " + intStack.Peek());

Console.WriteLine("Размерность стека " + intStack.Count);

Console.Write("Содержимое стека = ");

while (intStack.Count != 0)

Console.Write("{0} ", intStack.Pop());

Console.WriteLine("\nНовая размерность стека " + intStack.Count);

}

}

}

**Пример 2**.В текстовом файле содержится математическое выражение. Проверить баланс круглых скобок в данном выражении.

using System;

using System.Collections;

using System.IO;

namespace MyProgram

{

class Program

{

public static void Main()

{

StreamReader fileIn=new StreamReader("t.txt",);

string line=fileIn.ReadToEnd();

fileIn.Close();

Stack skobki=new Stack();

bool flag=true;

//проверяем баланс скобок

for ( int i=0; i<line.Length;i++)

{

//если текущий символ открывающаяся скобка, то помещаем ее в стек

if (line[i] == '(') skobki.Push(i);

else if (line[i] == ')') //если текущий символ закрывающаяся скобка, то

{

//если стек пустой, то для закрывающейся скобки не хватает парной открывающейся

if (skobki.Count == 0)

{ flag = false; Console.WriteLine("Возможно в позиции " + i + "лишняя ) скобка"); }

else skobki.Pop(); //иначе извлекаем парную скобку

}

}

//если после просмотра строки стек оказался пустым, то скобки сбалансированы

if (skobki.Count == 0) { if (flag)Console.WriteLine("скобки сбалансированы"); }

else //иначе баланс скобок нарушен

{

Console.Write("Возможно лишняя ( скобка в позиции:");

while (skobki.Count != 0)

{

Console.Write("{0} ", (int)skobki.Pop());

}

Console.WriteLine();

}

}

}

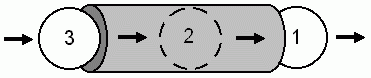
}

\_\_\_\_\_\_\_\_t.txt\_\_\_\_\_\_\_\_\_

(1+2)-4\*(a-3)/(2-7+6)

##### Класс Queue

АТД очередь - это частный случай однонаправленного списка, добавление элементов в который выполняется в один конец (хвост), а выборка производится с другого конца (головы). Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания FIFO (fist in - fist out, первым пришел - первым вышел). Очередь проще всего представить в виде узкой трубы, в один конец которой бросают мячи, а с другого конца которой они вылетают. Понятно, что мяч, который был брошен в трубу первым, первым и вылетит с другого конца.



В С# реализацию АТД очередь представляет класс Queue, который также как и стек реализует интерфейсы *ICollection*, *IEnumerable* и *ICloneable*. Queue - это динамическая коллекция, размер которой изменяется. При необходимости увеличение вместимости очереди происходит с коэффициентом роста по умолчанию равным 2.0.

В классе Queue определены следующие конструкторы:

public Queue(); //создает пустую очередь, начальная вместимость которой равна 32

public Queue (int capacity); // создает пустую очередь, начальная вместимость которой равна capacity

//создает пустую очередь, начальная вместимость которой равна capacity, и коэффициент роста

//устанавливается параметром n

public Queue (int capacity, float n);

//создает очередь, которая содержит элементы коллекции, заданной параметром с, и аналогичной

//вместимостью

public Queue (ICollection c);

Кроме методов, определенных в интерфейсах, реализуемых классом Queue, в этом классе определены *собственные методы*:

|  |  |
| --- | --- |
| Метод | Описание |
| public virtual bool Contains (object v) | Возвращает значение true, если объект v содержится в вызывающей очереди, в противном случае возвращает значение false |
| public virtual void clear () | Устанавливает свойство Count равным нулю, тем самым очищая очередь |
| public virtual object*Dequeue* () | Возвращает объект из начала вызывающей очереди, удаляя его из очереди |
| public virtual object *Peek*() | Возвращает объект из начала вызывающей очереди, не удаляя его из очереди |
| public virtual void*Enqueue*(object v) | Добавляет объект v в конец очереди |
| public virtual object [ ] ToArrау () | Возвращает массив, который содержит копии элементов из вызывающей очереди |
| public virtual void TrimToSizeO | Устанавливает свойство Capacity равным значению свойства Count |

Рассмотрим несколько примеров использования очереди.

**Пример 1**.Для заданного значения n запишем в очередь все числа от 1 до n, а затем извлечем их из очереди:

using System;

using System.Collections;

namespace MyProgram

{

class Program

{

public static void Main()

{

Console.Write("n= ");

int n=int.Parse(Console.ReadLine());

Queue intQ = new Queue();

for (int i = 1; i <= n; i++)

intQ.Enqueue(i);

Console.WriteLine("Размерность очереди " + intQ.Count);

Console.WriteLine("Верхний элемент очереди = " + intQ.Peek());

Console.WriteLine("Размерность очереди " + intQ.Count);

Console.Write("Содержимое очереди = " );

while (intQ.Count!=0)

Console.Write("{0} ", intQ.Dequeue());

Console.WriteLine("\nНовая размерность очереди " + intQ.Count);

}

}

}

**Пример 2**.В текстовом файле записана информация о людях (фамилия, имя, отчество, возраст, вес через пробел). Вывести на экран вначале информацию о людях младше 40 лет, а затем информацию о всех остальных.

using System;

using System.Collections;

using System.IO;

using System.Text;

namespace MyProgram

{

class Program

{

public struct one //структура для хранения данных об одном человеке

{

public string f;

public string i;

public string o;

public int age;

public float massa;

}

public static void Main()

{

StreamReader fileIn = new StreamReader("t.txt",Encoding.GetEncoding(1251));

string line;

Queue people = new Queue();

one a;

Console.WriteLine("ВОЗРАСТ МЕНЕЕ 40 ЛЕТ");

while ((line = fileIn.ReadLine()) != null) //читаем до конца файла

{

string [] temp = line.Split(' '); //разбиваем строку на составные элементы

//заполняем структуру

a.f = temp[0];

a.i = temp[1];

a.o = temp[2];

a.age = int.Parse(temp[3]);

a.massa = float.Parse(temp[4]);

// если возраст меньше 40 лет, то выводим данные на экран, иначе помещаем их в

//очередь для временного хранения

if (a.age<40)

Console.WriteLine(a.f + "\t"+ a.i + "\t"+ a.o + "\t"+a.age + "\t" + a.massa);

else people.Enqueue(a);

}

fileIn.Close();

Console.WriteLine("ВОЗРАСТ 40 ЛЕТ И СТАРШЕ");

while (people.Count != 0) //извлекаем из очереди данные

{

a = (one)people.Dequeue();

Console.WriteLine(a.f + "\t"+ a.i + "\t"+ a.o + "\t"+a.age + "\t" + a.massa);

}

}

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_t.txt\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Иванов Сергей Николаевич 21 64

Петров Игорь Юрьевич 45 88

Семёнов Михаил Алексеевич 20 70

Пиманов Александр Дмитриевич 53 101

##### Класс ArrayList

В С# стандартные массивы имеют фиксированную длину, которая не может измениться во время выполнения программы. Класс ***ArrayList***предназначен для поддержки динамических массивов, которые при необходимости могут увеличиваться или сокращаться.

Объект класса ***ArrayList***представляет собой массив переменной длины, элементами которого являются объектные ссылки. Любой объект класса ***ArrayList***создается с некоторым начальным размером. При превышении этого размера коллекция автоматически удваивается. В случае удаления объектов массив можно сократить.

Класс *ArrayList* реализует интерфейсы *ICollection*, *IList*, *IEnumerable* и *ICloneable*. В классе *ArrayList*определены следующие конструкторы:

//создает пустой массив с максимальной емкостью равной 16 элементам, при текущей размерности 0

public ArrayList()

public ArrayList(int capacity) //создает массив с заданной емкостью capacity, при текущей размерности 0

public ArrayList(ICollection с) //строит массив, который инициализируется элементами коллекции с

Помимо методов, определенных в интерфейсах, которые реализует класс *ArrayList*, в нем определены и *собственные методы*:

|  |  |
| --- | --- |
| Метод | Описание |
| public virtual void AddRange (ICollectionс) | Добавляет элементы из коллекции с в конец вызывающей коллекции |
| public virtual intBinarySearch (object v) | В вызывающей отсортированной коллекции выполняет поиск значения, заданного параметром v. Возвращает индекс найденного элемента. Если искомое значение не обнаружено, возвращает отрицательное значение. |
| public virtual intBinarySearch (object v, IComparer comp) | В вызывающей отсортированной коллекции выполняет поиск значения, заданного параметром v, на основе метода сравнения объектов, заданного параметром соmp. Возвращает индекс найденного элемента. Если искомое значение не обнаружено, возвращает отрицательное значение. |
| public virtual intBinarySearch (int startldx, int count, object v, IComparer comp) | В вызывающей отсортированной коллекции выполняет поиск значения, заданного параметром v, на основе метода сравнения объектов, заданного параметром comp. Поиск начинается с элемента, индекс которого равен значению startIdx, и включает count элементов. Метод возвращает индекс найденного элемента. Если искомое значение не обнаружено, возвращает отрицательное значение. |
| public virtual void CopyTo(Array ar, int startIdx) | Копирует содержимое вызывающей коллекции, начиная с элемента, индекс которого равен значению startIdx, в массив, заданный параметром аr. Приемный массив должен быть одномерным и совместимым по типу с элементами коллекции. |
| public virtual void CopyTo(int srcldx, Array ar, int destIdx, int count) | Копирует count элементов вызывающей коллекции, начиная с элемента, индекс которого равен значению srcIdx, в массив, заданный параметром аr, начиная с элемента, индекс которого равен значению destIdx. Приемный массив должен быть одномерным и совместимым по типу с элементами коллекции |
| public virtualArrayListGetRange(int idx, int count) | Возвращает часть вызывающей коллекции типа ArrayList. Диапазон возвращаемой коллекции начинается с индекса idx и включает count элементов. Возвращаемый объект ссылается на те же элементы, что и вызывающий объект |
| public staticArrayListFixedSize(ArrayListar) | Превращает коллекцию ar в ArrayList -массив с фиксированным размером и возвращает результат |
| public virtual void InsertRange(int startldx, ICollectionc) | Вставляет элементы коллекции, заданной параметром с, в вызывающую коллекцию, начиная с индекса, заданного параметром startIdx |
| public virtual int LastlndexOf(object v) | Возвращает индекс последнего вхождения объекта v в вызывающей коллекции. Если искомый объект не обнаружен, возвращает отрицательное значение |
| public staticArrayListReadonly(ArrayListar) | Превращает коллекцию ar в ArrayList -массив, предназначенный только для чтения |
| public virtual void RemoveRange(int idx, int count) | Удаляет count элементов из вызывающей коллекции, начиная с элемента, индекс которого равен значению idx |
| public virtual void Reverse() | Располагает элементы вызывающей коллекции в обратном порядке |
| public virtual void Reverse(int startldx, int count) | Располагает в обратном порядке count элементов вызывающей коллекции, начиная с индекса startldx |
| public virtual void SetRange(int startldx, ICollectionc) | Заменяет элементы вызывающей коллекции, начиная с индекса startldx, элементами коллекции, заданной параметром с |
| public virtual void Sort() | Сортирует коллекцию по возрастанию |
| public virtual void Sort(IComparer comp) | Сортирует вызывающую коллекцию на основе метода сравнения объектов, заданного параметром comp. Если параметр comp имеет нулевое значение, для каждого объекта используется стандартный метод сравнения |
| public virtual void Sort ( int startidx, int endidx, icomparer comp) | Сортирует часть вызывающей коллекции на основе метода сравнения объектов, заданного параметром comp. Сортировка начинается с индекса startidx и заканчивается индексом endidx. Если параметр comp имеет нулевое значение, для каждого объекта используется стандартный метод сравнения |
| public virtual object [ ] ToArray () | Возвращает массив, который содержит копии элементов вызывающего объекта |
| public virtual Array ToArray (Type type) | Возвращает массив, который содержит копии элементов вызывающего объекта. Тип элементов в этом массиве задается параметром type |
| public virtual void TrimToSize() | Устанавливает свойство Capacity равным значению свойства Count |

Свойство Capacity позволяет узнать или установить емкость вызывающего динамического массива типа *ArrayList*. Емкость представляет собой количество элементов, которые можно сохранить в *ArrayList* -массиве без его увеличения. Если вам заранее известно, сколько элементов должно содержаться в *ArrayList* -массиве, то размерность массива можно установить используя свойство Capacity, съэкономив тем самым системные ресурсы. Если нужно уменьшить размер *ArrayList*-массива, то путем установки свойства Capacity можно сделать его меньшим. Но устанавливаемое значение не должно быть меньше значения свойства Count, иначе будет сгенерировано исключение ArgumentOutOfRangeException. Чтобы сделать емкость *ArrayList* -массива равной действительному количеству элементов, хранимых в нем в данный момент, установите свойство Capacity равным свойству Count. Того же эффекта можно добиться, вызвав метод TrimToSize ().

Рассмотрим несколько примеров использования динамического массива.

using System;

using System.Collections;

namespace MyProgram

{

class Program

{

static void ArrayPrint(string s, ArrayList a)

{

Console.WriteLine(s);

foreach (int i in a)

Console.Write(i + " ");

Console.WriteLine();

}

static void Main(string[] args)

{

ArrayList myArray = new ArrayList();

Console.WriteLine("Начальная емкость массива: " + myArray.Capacity);

Console.WriteLine("Начальное количество элементов: " + myArray.Count);

Console.WriteLine("\nДобавили 5 цифр");

for (int i = 0; i < 5; i++) myArray.Add(i);

Console.WriteLine("Текущая емкость массива: " + myArray.Capacity);

Console.WriteLine("Текущее количество элементов: " + myArray.Count);

ArrayPrint("Содержимое массива", myArray);

Console.WriteLine("\nОптимизируем емкость массива");

myArray.Capacity=myArray.Count;

Console.WriteLine("Текущая емкость массива: " + myArray.Capacity);

Console.WriteLine("Текущее количество элементов: " + myArray.Count);

ArrayPrint("Содержимое массива", myArray);

Console.WriteLine("\nДобавляем элементы в массив");

myArray.Add(10);

myArray.Insert(1, 0);

myArray.AddRange(myArray);

Console.WriteLine("Текущая емкость массива: " + myArray.Capacity);

Console.WriteLine("Текущее количество элементов: " + myArray.Count);

ArrayPrint("Содержимое массива", myArray);

Console.WriteLine("\nУдаляем элементы из массива");

myArray.Remove(0);

myArray.RemoveAt(10);

Console.WriteLine("Текущая емкость массива: " + myArray.Capacity);

Console.WriteLine("Текущее количество элементов: " + myArray.Count);

ArrayPrint("Содержимое массива", myArray);

Console.WriteLine("\nУдаляем весь массив");

myArray.Clear();

Console.WriteLine("Текущая емкость массива: " + myArray.Capacity);

Console.WriteLine("Текущее количество элементов: " + myArray.Count);

ArrayPrint("Содержимое массива", myArray);

}

}

}

**Пример 2**.В текстовом файле записана информация о людях (фамилия, имя, отчество, возраст, вес через пробел). Вывести на экран информацию о людях, отсортированную по возрасту.

using System;

using System.Collections;

using System.IO;

using System.Text;

namespace MyProgram

{

class Program

{

public struct one //структура для хранения данных об одном человеке

{

public string f;

public string i;

public string o;

public int age;

public float massa;

}

public class SortByAge : IComparer //реализация стандартного интерфейса

{

int IComparer.Compare(object x, object y) //переопределение метода Compare

{

one t1 = (one)x;

one t2 = (one)y;

if (t1.age > t2.age) return 1;

if (t1.age < t2.age) return -1;

return 0;

}

}

static void ArrayPrint(string s, ArrayList a)

{

Console.WriteLine(s);

foreach (one x in a)

Console.WriteLine(x.f + "\t"+ x.i + "\t"+ x.o + "\t"+x.age + "\t" + x.massa);

}

static void Main(string[] args)

{

StreamReader fileIn = new StreamReader("t.txt",Encoding.GetEncoding(1251));

string line;

one a;

ArrayList people = new ArrayList();

string[] temp = new string[5];

while ((line=fileIn.ReadLine())!=null) //цикл для организации обработки файла

{

temp = line.Split(' ');

a.f = temp[0];

a.i = temp[1];

a.o = temp[2];

a.age = int.Parse(temp[3]);

a.massa = float.Parse(temp[4]);

people.Add(a);

}

fileIn.Close();

ArrayPrint("Исходные данные: ", people);

people.Sort(new Program.SortByAge()); //вызов сортировки

ArrayPrint("Отсортированные данные: ", people);

}

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_t.txt\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Иванов Сергей Николаевич 21 64

Петров Игорь Юрьевич 45 88

Семёнов Михаил Алексеевич 20 70

Пиманов Александр Дмитриевич 53 101

В данном примере был разработан вложенный класс SortByAge, реализующий стандартный интерфейс IComparer. В этом классе был *перегружен метод* Compare, позволяющий сравнивать между собой два объекта типа one. Созданный класс использовался для сортировки коллекции по заданному критерию (по возрасту).

##### Класс Hashtable

Класс Hashtable предназначен для создания коллекции, в которой для хранения объектов используется хеш-таблица. В хеш-таблице для хранения информации используется механизм, именуемый хешированием (hashing). Суть хеширования состоит в том, что для определения уникального значения, которое называется хеш-кодом, используется информационное содержимое соответствующего ему ключа. Хеш-код затем используется в качестве индекса, по которому в таблице отыскиваются данные, соответствующие этому ключу. Преобразование ключа в хеш-код выполняется автоматически, т.е. сам хеш-код вы даже не увидите. Но преимущество хеширования - в том, что оно позволяет сокращать время выполнения таких операций, как поиск, считывание и запись данных, даже для больших объемов информации.

Класс Hashtable реализует стандартные интерфейсы IDictionary, *ICollection*, *IEnumerable*, ISerializable, IDeserializationCallback и *ICloneable*. Размер хеш-таблицы может динамически изменяться. Размер таблицы увеличивается тогда, когда количество элементов превышает значение, равное произведению вместимости таблицы и ее коэффициента заполнения, который может принимать значение на интервале от 0,1 до 1,0. По умолчанию установлен коэффициент равный 1,0.

В классе Hashtable определено несколько конструкторов:

public Hashtable() //создает пустую хеш-таблицу

// строит хеш-таблицу, которая инициализируется элементами коллекции с

public Hashtable(IDictionary с)

public Hashtable(int capacity) //создает хеш-таблицу с вместимостью capacity

//создает хеш-таблицу вместимостью capacity и коэффициентом заполнения n

public Hashtable(int capacity, float n)

Помимо методов, определенных в интерфейсах, которые реализует класс Hashtable, в нем определены и *собственные методы*:

|  |  |
| --- | --- |
| Метод | Описание |
| public virtual bool ContainsKey (object k) | Возвращает значение true , если в вызывающей хеш-таблице содержится ключ, заданный параметром k. В противном случае возвращает значение false |
| public virtual bool ContainsValue (object v) | Возвращает значение true, если в вызывающей хеш-таблице содержится значение, заданное параметром v. В противном случае возвращает значение false |
| public virtual IDictionaryEnumerator GetEnumerator() | Возвращает для вызывающей хеш-таблицы нумератор типа IDictionaryEnumerator |

В классе Hashtable, помимо свойств, определенных в реализованных им интерфейсах, определены два собственных public-свойства:

public virtual ICollection Keys { get; } //позволяет получить коллекцию ключей

public virtual ICollection Values { get; } //позволяет получить коллекцию значений

Для добавления элемента в хеш-таблицу необходимо вызвать метод Add(), который принимает два отдельных аргумента: ключ и значение. Важно отметить, что хеш-таблица не гарантирует сохранения порядка элементов, т.к хеширование обычно не применяется к отсортированным таблицам.

Рассмотрим пример, который демонстрирует использование Hashtable коллекции:

**Пример 1**:рассмотрим простые операции с хеш-таблицей

using System;

using System.Collections;

namespace MyProgram

{

class Program

{

static void printTab(string s, Hashtable a)

{

Console.WriteLine(s);

ICollection key = a.Keys; //Прочитали все ключи

foreach (string i in key)//использование ключа для получения значения

{

Console.WriteLine(i+"\t"+a[i]);

}

Console.WriteLine();

}

static void Main(string[] args)

{

Hashtable tab = new Hashtable();

Console.WriteLine("Начальное количество элементов: " + tab.Count);

printTab("Содержимое таблицы: ", tab);

Console.WriteLine("Добавили в таблицу записи");

tab.Add("001","ПЕРВЫЙ");

tab.Add("002","ВТОРОЙ");

tab.Add("003","ТРЕТИЙ");

tab.Add("004", "ЧЕТВЕРТЫЙ");

tab.Add("005", "ПЯТЫЙ");

Console.WriteLine("Текущее количество элементов: " + tab.Count);

printTab("Содержимое заполненной таблицы", tab);

tab["005"] = "НОВЫЙ ПЯТЫЙ";

tab["001"] = "НОВЫЙ ПЕРВЫЙ";

printTab("Содержимое измененной таблицы", tab);

}

}

}

**Пример 2**. Разработаем простейшую записную книжку, в которую можно добавлять и удалять телефоны, а также осуществлять поиск номера телефона по фамилии и фамилии по номеру телефона.

using System;

using System.Collections;

using System.IO;

using System.Text;

namespace MyProgram

{

class Program

{

static void printTab(string s, Hashtable a)

{

Console.WriteLine(s);

ICollection key = a.Keys; //Прочитали все ключи

foreach (string i in key)//использование ключа для получения значения

{

Console.WriteLine(i + "\t" + a[i]);

}

}

static void Main(string[] args)

{

StreamReader fileIn = new StreamReader("t.txt",Encoding.GetEncoding(1251));

string line;

Hashtable people = new Hashtable();

while ((line = fileIn.ReadLine()) != null) //цикл для организации обработки файла

{

string [] temp = line.Split(' ');

people.Add(temp[0],temp[1]);

}

fileIn.Close();

printTab("Исходные данные: ", people);

Console.WriteLine("Введите номер телефона");

line = Console.ReadLine();

if (people.ContainsKey(line)) Console.WriteLine(line + "\t" + people[line]);

else

{

Console.WriteLine("Такого номера нет в записной книжке.\nВведите фамилию: ");

string line2=Console.ReadLine();

people.Add(line,line2);

}

printTab("Исходные данные: ", people);

Console.WriteLine("Введите фамилию для удаления");

line = Console.ReadLine();

if (people.ContainsValue(line))

{

ICollection key =people.Keys; //Прочитали все ключи

Console.WriteLine(line);

string del="";

foreach (string i in key)//использование ключа для получения значения

if (string.Compare((string)people[i], line) == 0)

{

del = i;

break;

}

Console.WriteLine(del + "\t" + people[del] + "- данные удалены!!!");

people.Remove(del);

printTab("Измененные данные: ", people);

}

else Console.WriteLine("Такого абонента в записной книжке нет ");

}

}

}

\_\_\_\_\_\_\_\_\_t.txt\_\_\_\_\_\_\_\_\_\_\_\_

12-34-56 Иванов

78-90-12 Петров

34-56-78 Семёнов

90-11-12 Пиманов

## Задания на лабораторную работу.

1. Решить следующие задачи с использованием класса Stack:
   1. Дан файл, в котором записан набор чисел. Переписать в другой файл все числа в обратном порядке.
   2. Создать текстовый файл. Распечатать гласные буквы этого файла в обратном порядке.
   3. Напечатать содержимое текстового файла t, выписывая литеры каждой его строки в обратном порядке.
   4. Даны 2 строки s1 и s2. Из каждой можно читать по одному символу. Выяснить, является ли строка s2 обратной s1.
   5. Написать программу подсчета выражения в префиксной форме.
   6. Написать программу, которая преобразует выражение из инфиксной формы в префиксную.
   7. Написать программу преобразования выражения из постфиксной формы в инфиксную.
   8. В текстовом файле записана без ошибок формула вида:

<формула>=<цифра>|M(<формула>,<формула>)|m(<формула>,<формула>)  
<цифра>=0|1|2|3|4|5|6|7|8|9  
M обозначает вычисление максимума, m – минимума  
Вычислить значение этой формулы  
Например M(m(3,5),M(1,2))=3

* 1. В текстовом файле записана без ошибок формула вида:  
     <формула>=  
     <цифра>|p(<формула>,<формула>)|m(<формула>,<формула>)  
     <цифра>=0|1|2|3|4|5|6|7|8|9   
     m (a, b) = (a-b) mod 10,  
     p (a, b) = (a+b) mod 10.  
     Вычислить значение этой формулы. Например, m (9, p (p (3, 5), m (3, 8))) = 6.
  2. Пусть символ # определен в текстовом редакторе как стирающий символ Backspace, т.е. строка abc#d##c в действительности является строкой ac.

Дан текст, в котором встречается символ #. Преобразовать его с учетом действия этого символа.

1. Решить следующие задачи с использованием класса Queue:
   1. Дан текстовый файл. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все символы, отличные от цифр, а затем все цифры, сохраняя исходный порядок в каждой группе символов.
   2. Дан файл, содержащий числа. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все числа, из интервала [a,b], потом все числа, меньшие a, потом все числа, большие b, сохраняя исходный порядок в каждой группе чисел.
   3. Дан текстовый файл. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все слова, начинающиеся на гласную букву, потом все слова, начинающиеся на согласную букву, сохраняя исходный порядок в каждой группе слов.
   4. Дан файл, содержащий числа. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все положительные числа, потом все отрицательные числа, сохраняя исходный порядок в каждой группе чисел.
   5. Дан текстовый файл. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все слова, начинающиеся с прописной буквы, потом все слова, начинающиеся со строчной буквы, сохраняя исходный порядок в каждой группе слов.
   6. Дан файл, содержащий информацию о сотрудниках фирмы: фамилия, имя, отчество, пол, возраст, размер зарплаты. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о мужчинах, потом все данные о женщинах, сохраняя исходный порядок в каждой группе сотрудников.
   7. Дан файл, содержащий информацию о сотрудниках фирмы: фамилия, имя, отчество, пол, возраст, размер зарплаты. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о сотрудниках, зарплата которых меньше 10000, потом данные об остальных сотрудниках, сохраняя исходный порядок в каждой группе сотрудников.
   8. Дан файл, содержащий информацию о сотрудниках фирмы: фамилия, имя, отчество, пол, возраст, размер зарплаты. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о сотрудниках младше 30 лет, потом данные об остальных сотрудниках, сохраняя исходный порядок в каждой группе сотрудников.
   9. Дан файл, содержащий информацию о студентах: фамилия, имя, отчество, номер группы, оценки по трем предметам текущей сессии. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о студентах, успешно сдавших сессию, потом данные об остальных студентах, сохраняя исходный порядок в каждой группе сотрудников.
   10. Дан файл, содержащий информацию о студентах: фамилия, имя, отчество, номер группы, оценки по трем предметам текущей сессии. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о студентах, успешно обучающихся на 4 и 5, потом данные об остальных студентах, сохраняя исходный порядок в каждой группе сотрудников.
2. Решить задачи из заданий 1, используя класс ArrayList.
3. Решить задачи из заданий 2, используя класс ArrayList.
4. Решить задачу, используя класс HashTable:  
   реализовать простейший каталог музыкальных компакт-дисков, который позволяет:

Добавлять и удалять диски.  
Добавлять и удалять песни.  
Просматривать содержимое целого каталога и каждого диска в отдельности.  
Осуществлять поиск всех записей заданного исполнителя по всему каталогу.

## Контрольные вопросы.

# С#-.Организация системы ввода-вывода

## Теоретические сведения.

С#-программы выполняют *операции* ввода-вывода посредством потоков, которые построены на иерархии классов. *Поток*(*stream*) - это *абстракция*, которая генерирует и принимает данные. С помощью потока можно читать данные из различных источников (клавиатура, *файл*) и записывать в различные источники (принтер, экран, *файл*). Несмотря на то, что потоки связываются с различными физическими устройствами, характер поведения всех потоков одинаков. Поэтому классы и методы ввода-вывода можно применить ко многим типам устройств.

На самом низком уровне иерархии *потоков ввода-вывода* находятся потоки, оперирующие байтами. Это объясняется тем, что многие устройства при выполнении операций ввода-вывода ориентированы на байты. Однако для человека привычнее оперировать символами, поэтому разработаны символьные потоки, которые фактически представляют собой оболочки, выполняющие преобразование байтовых потоков в символьные и наоборот. Кроме этого, реализованы потоки для работы с int -, double -, short - значениями, которые также представляют оболочку для байтовых потоков, но работают не с самими значениями, а с их внутренним представлением в виде двоичных кодов.

Центральную часть потоковой С#-системы занимает *класс* Stream пространства имен System.IO. Класс Stream представляет байтовый *поток* и является базовым для всех остальных потоковых классов. Из класса *Stream* выведены такие байтовые классы потоков как:

1. FileStream - байтовый поток, разработанный для *файлового ввода-вывода*
2. BufferedStream - заключает в оболочку байтовый поток и добавляет буферизацию, которая во многих случаях увеличивает производительность программы;
3. MemoryStream - байтовый поток, который использует память для хранения данных.

Программист может вывести собственные потоковые классы. Однако для подавляющего большинства приложений достаточно встроенных потоков.

Подробно мы рассмотрим *класс* FileStream, классы StreamWriter и StreamReader, представляющие собой оболочки для класса FileStream и позволяющие преобразовывать байтовые потоки в символьные, а также классы BinaryWriter и BinaryReader, представляющие собой оболочки для класса FileStream и позволяющие преобразовывать байтовые потоки в двоичные для работы с int -, double -, short - и т.д. значениями.

### Байтовый поток

Чтобы создать байтовый поток, связанный с файлом, создается объект класса FileStream. При этом в классе определено несколько конструкторов. Чаще всего используется конструктор, который открывает поток для чтения и/или записи:

FileStream(string filename, FileMode mode)

где:

1. параметр filename определяет имя файла, с которым будет связан поток ввода-вывода данных; при этом filenameопределяет либо полный путь к файлу, либо имя файла, который находится в папке bin/debug вашего проекта.
2. параметр mode определяет режим открытия файла, который может принимать одно из возможных значений, определенных перечислением FileMode:
   * FileMode.Append - предназначен для добавления данных в конец файла;
   * FileMode.Create - предназначен для создания нового файла, при этом если существует файл с таким же именем, то он будет предварительно удален;
   * FileMode.*CreateNew* - предназначен для создания нового файла, при этом файл с таким же именем не должен существовать;
   * FileMоde.Open - предназначен для открытия существующего файла;
   * FileMode.ОpenOrCreate - если файл существует, то открывает его, в противном случае создает новый
   * FileMode.Truncate - открывает существующий файл, но усекает его длину до нуля

Если попытка открыть файл оказалась неуспешной, то генерируется одно из исключений: FileNotFoundException - файл невозможно открыть по причине его отсутствия, IOException - файл невозможно открыть из-за ошибки ввода-вывода, ArgumentNullException - имя файла представляет собой null -значение, ArgumentException - некорректен параметр mode, SecurityException - пользователь не обладает правами доступа, DirectoryNotFoundException - некорректно задан каталог.

Другая версия конструктора позволяет ограничить доступ только чтением или только записью:

FileStream(string filename, FileMode mode, FileAccess how)

где:

1. параметры filename и mode имеют то же назначение, что и в предыдущей версии конструктора;
2. параметр how, определяет способ доступа к файлу и может принимать одно из значений, определенных перечислением FileAccess:
3. FileAccess.Read - только чтение;
4. FileAccess.Write - только запись;
5. FileAccess.ReadWrite - и чтение, и запись.

После установления связи байтового потока с физическим файлом внутренний указатель потока устанавливается на начальный байт файла.

Для чтения очередного байта из потока, связанного с физическим файлом, используется метод ReadByte(). После прочтения очередного байта внутренний указатель перемещается на следующий байт файла. Если достигнут конец файла, то метод ReadByte() возвращает значение -1.

Для побайтовой записи данных в поток используется метод WriteByte().

По завершении работы с файлом его необходимо закрыть. Для этого достаточно вызвать метод Close (). При закрытии файла освобождаются системные ресурсы, ранее выделенные для этого файла, что дает возможность использовать их для работы с другими файлами.

Рассмотрим пример использования класса FileStream, для копирования одного файла в другой. Но вначале создадим текстовый файл text.txt в папке bin/debug текущего проекта. И внесем в него произвольную информацию, например:

12 456

Hello!

23,67 4: Message

using System;

using System.Text;

using System.IO; //для работы с потоками

namespace MyProgram

{

class Program

{

static void Main()

{

try

{

FileStream fileIn = new FileStream("text.txt",

FileMode.Open,

FileAccess.Read);

FileStream fileOut = new FileStream("newText.txt",

FileMode.Create,

FileAccess.Write);

int i;

while ((i = fileIn.ReadByte())!=-1)

{

//запись очередного файла в поток, связанный с файлом fIleOut

fileOut.WriteByte((byte)i);

}

fileIn.Close();

fileOut.Close();

}

catch (Exception EX)

{

Console.WriteLine(EX.Message);

}

}

}

}

**Задание**. Подумайте, почему для переменной i указан тип int. Можно было бы указать тип byte?

### Символьный поток

Чтобы создать символьный поток нужно поместить объект класса Stream (например, FileStream ) "внутрь" объекта класса StreamWriter или объекта класса StreamReader. В этом случае байтовый поток будет автоматически преобразовываться в символьный.

**Класс StreamWriter** предназначен для организации выходного символьного потока. В нем определено несколько конструкторов. Один из них записывается следующим образом:

StreamWriter(Stream stream);

где параметр stream определяет имя уже открытого байтового потока.

Например, создать экземпляр класса StreamReader можно следующим образом:

StreamWriter fileOut=new StreamWriter(new FileStream("text.txt",

FileMode.Create,

FileAccess.Write));

Этот конструктор генерирует исключение типа ArgumentException, если поток stream не открыт для вывода, и исключение типа ArgumentNullException, если он (поток) имеет null-значение.

Другой вид конструктора позволяет открыть поток сразу через обращения к файлу:

StreamWriter(string name);

где параметр name определяет имя открываемого файла.

Например, обратиться к данному конструктору можно следующим образом:

StreamWriter fileOut=new StreamWriter("c:\temp\t.txt");

И еще один вариант конструктора StreamWriter:

StreamWriter(string name, bool appendFlag);

где параметр name определяет имя открываемого файла;

параметр appendFlag может принимать значение true - если нужно добавлять данные в конец файла, или false - если файл необходимо перезаписать.

Например:

StreamWriter fileOut=new StreamWriter("t.txt", true);

Теперь для записи данных в поток fileOut можно обратиться к методу WriteLine. Это можно сделать следующим образом:

fileOut.WriteLine("test");

В данном случае в конец файла t.txt будет дописано слово test.

**Класс StreamReader** предназначен для организации входного символьного потока. Один из его конструкторов выглядит следующим образом:

StreamReader(Stream stream);

где параметр stream определяет имя уже открытого байтового потока.

Этот конструктор генерирует исключение типа ArgumentException, если поток stream не открыт для ввода.

Например, создать экземпляр класса StreamWriter можно следующим образом:

StreamReader fileIn = new StreamReader(new FileStream("text.txt",

FileMode.Open,

FileAccess.Read));

Как и в случае с классом StreamWriter у класса StreamReader есть и другой вид конструктора, который позволяет открыть файл напрямую:

StreamReader (string name);

где параметр name определяет имя открываемого файла.

Обратиться к данному конструктору можно следующим образом:

StreamReader fileIn=new StreamReader ("c:\temp\t.txt");

В C# символы реализуются кодировкой Unicode. Для того, чтобы можно было обрабатывать текстовые файлы, содержащие русские символы, созданные, например, в Блокноте, рекомендуется вызывать следующий вид конструктора StreamReader:

StreamReader fileIn=new StreamReader ("c:\temp\t.txt",

Encoding.GetEncoding(1251));

Параметр Encoding.GetEncoding(1251) говорит о том, что будет выполняться преобразование из кода Windows-1251 (одна из модификаций кода ASCII, содержащая русские символы) в Unicode. Encoding.GetEncoding(1251) реализован в пространстве имен System.Text.

Теперь для чтения данных из потока fileIn можно воспользоваться методом ReadLine. При этом если будет достигнут конец файла, то метод ReadLine вернет значение null.

Рассмотрим пример, в котором данные из одного файла копируются в другой, но уже с использованием классов StreamWriter и StreamReader.

static void Main()

{

StreamReader fileIn = new StreamReader("text.txt",

Encoding.GetEncoding(1251));

StreamWriter fileOut=new StreamWriter("newText.txt", false);

string line;

while ((line=fileIn.ReadLine())!=null) //пока поток не пуст

{

fileOut.WriteLine(line);

}

fileIn.Close();

fileOut.Close();

}

**Задание**. Выясните, для чего предназначен метод ReadToEnd() и когда имеется смысл его применять.

Таким образом, данный способ копирования одного файла в другой, даст нам тот же результат, что и при использовании байтовых потоков. Однако, его работа будет менее эффективной, т.к. будет тратиться дополнительное время на преобразование байтов в символы. Но у символьных потоков есть свои преимущества. Например, мы можем использовать регулярные выражения для поиска заданных фрагментов текста в файле.

static void Main()

{

StreamReader fileIn = new StreamReader("text.txt");

StreamWriter fileOut=new StreamWriter("newText.txt", false);

string text=fileIn.ReadToEnd();

Regex r= new Regex(@"[-+]?\d+");

Match integer = r.Match(text);

while (integer.Success)

{

fileOut.WriteLine(integer);

integer = integer.NextMatch();

}

fileIn.Close();

fileOut.Close();

}

### Двоичные потоки

Двоичные файлы хранят данные в том же виде, в котором они представлены в оперативной памяти, то есть во внутреннем представлении. Двоичные файлы не применяются для просмотра человеком, они используются только для программной обработки.

Выходной поток BinaryWriter поддерживает произвольный доступ, т.е. имеется возможность выполнять запись в произвольную позицию двоичного файла. Наиболее важные методы потока BinaryWriter:

|  |  |
| --- | --- |
| Член класса | Описание |
| BaseStream | Определяет базовый поток, с которым работает объект BinaryWriter |
| Close | Закрывает поток |
| Flush | Очищает буфер |
| Seek | Устанавливает позицию в текущем потоке |
| Write | Записывает значение в текущий поток |

Наиболее важные методы выходного потока BinaryReader:

|  |  |
| --- | --- |
| Член класса | Описание |
| BaseStream | Определяет базовый поток, с которым работает объект BinaryReader |
| Close | Закрывает поток |
| PeekChar | Возвращает следующий символ потока без перемещения внутреннего указателя в потоке |
| Read | Считывает очередной поток байтов или символов и сохраняет в массиве, передаваемом во входном параметре |
| ReadBoolean, ReadByte, ReadInt32 и т.д | Считывает из потока данные определенного типа |

Двоичный поток открывается на основе базового потока (например, FileStream ), при этом двоичный поток будет преобразовывать байтовый поток в значения int -, double -, short - и т.д.

Рассмотрим пример формирования двоичного файла:

static void Main()

{

//открываем двоичный поток

BinaryWriter fOut=new BinaryWriter(new FileStream("t.dat",FileMode.Create));

//записываем данные в двоичный поток

for (int i=0; i<=100; i+=2)

{

fOut.Write(i);

}

fOut.Close(); //закрываем двоичный поток

}

Попытка просмотреть двоичный файл через текстовый редактор неинформативна. Двоичный файл просматривается программным путем, например следующим образом:

static void Main()

{

FileStream f=new FileStream("t.dat",FileMode.Open);

BinaryReader fIn=new BinaryReader(f);

long n=f.Length/4; //определяем количество чисел в двоичном потоке

int a;

for (int i=0; i<n; i++)

{

a=fIn.ReadInt32();

Console.Write(a+" ");

}

fIn.Close();

f.Close();

}

Двоичные файлы являются файлами с произвольным доступом, при этом нумерация элементов в двоичном файле ведется с нуля. Произвольный доступ обеспечивает метод Seek. Рассмотрим его синтаксис:

Seek(long newPos, SeekOrigin pos)

где параметр newPos определяет новую позицию внутреннего указателя файла в байтах относительно исходной позиции указателя, которая определяется параметром pos. В свою очередь параметр pos должен быть задан одним из значений перечисления SeekOrigin:

|  |  |
| --- | --- |
| Значение | Описание |
| SeekOrigin.Begin | Поиск от начала файла |
| SeekOrigin.Current | Поиск от текущей позиции указателя |
| SeekOrigin.End | Поиск от конца файла |

После вызова метода Seek следующие операции чтения или записи будут выполняться с новой позиции внутреннего указателя файла.

Рассмотрим пример организации произвольного доступа к двоичному файлу (на примере файла t.dat ):

static void Main()

{

//изменение данных в двоичном потоке

FileStream f=new FileStream("t.dat",FileMode.Open);

BinaryWriter fOut=new BinaryWriter(f);

long n=f.Length; //определяем количество байт в байтовом потоке

int a;

for (int i=0; i<n; i+=8) //сдвиг на две позиции, т.к. тип int занимает 4 байта

{

fOut.Seek(i,SeekOrigin.Begin);

fOut.Write(0);

}

fOut.Close();

//чтение данных из двоичного потока

f=new FileStream("t.dat",FileMode.Open);

BinaryReader fIn=new BinaryReader(f);

n=f.Length/4; //определяем количество чисел в двоичном потоке

for (int i=0; i<n; i++)

{

a=fIn.ReadInt32();

Console.Write(a+" ");

}

fIn.Close();

f.Close();

}

Поток BinaryReader не имеет метода Seek, однако используя возможности потока FileStream можно организовать произвольный доступ при чтении двоичных файлов. Рассмотрим следующий пример:

static void Main()

{

//Записываем в файл t.dat целые числа от 0 до 100

FileStream f=new FileStream("t.dat",FileMode.Open);

BinaryWriter fOut=new BinaryWriter(f);

for (int i=0; i<100; ++i)

{

fOut.Write(i);

}

fOut.Close();

//Объекты f и fIn связаны с одним и тем же файлом

f=new FileStream("t.dat",FileMode.Open);

BinaryReader fIn=new BinaryReader(f);

long n=f.Length; //определяем количество байт потоке

//Читаем данные из файла t.dat, перемещая внутренний указатель на 8 байт, т.е. на два целых числа

for (int i=0; i<n; i+=8)

{

f.Seek(i,SeekOrigin.Begin);

int a=fIn.ReadInt32();

Console.Write(a+" ");

}

fIn.Close();

f.Close();

}

### Перенаправление стандартных потоков

Тремя стандартными потоками, доступ к которым осуществляется через свойства Console.Out, Console.In и Console.Error, могут пользоваться все программы, работающие в пространстве имен System. Свойство Console.Out относится к стандартному выходному потоку. По умолчанию это консоль. Например, при вызове метода Console.WriteLine() информация автоматически передается в поток Console.Out. Свойство Console.In относится к стандартному входному потоку, источником которого по умолчанию является клавиатура. Например, при вводе данных с клавиатуры информация автоматически передается потоку Console.In, к которому можно обратиться с помощью метода Console.ReadLine(). Свойство Console.Error относится к ошибкам в стандартном потоке, источником которого также по умолчанию является консоль. Однако эти потоки могут быть перенаправлены на любое совместимое устройство ввода-вывода, например, на работу с физическими файлами.

Перенаправить стандартный поток можно с помощью методов SetIn(), SetOut() и SetError(), которые являются членами класса Console:

static void Setln(TextReader input)

static void SetOut(TextWriter output)

static void SetError(TextWriter output)

Пример *перенаправления потоков* проиллюстрирован следующей программой, в которой двумерный массив вводится из файла input.txt, а выводится в файл output.txt

static void Main()

{

try

{

int[,] MyArray;

StreamReader file=new StreamReader("input.txt");

Console.SetIn(file); // перенаправляем стандартный входной поток на file

string line=Console.ReadLine();

string []mas=line.Split(' ');

int n=int.Parse(mas[0]);

int m=int.Parse(mas[1]);

MyArray = new int[n,m];

for (int i = 0; i < n; i++)

{

line = Console.ReadLine();

mas = line.Split(' ');

for (int j = 0; j < m; j++)

{

MyArray[i,j] = int.Parse(mas[j]);

}

}

PrintArray("исходный массив:", MyArray, n, m);

file.Close();

}

static void PrintArray(string a, int[,] mas, int n, int m)

{

StreamWriter file=new StreamWriter("output.txt"); // перенаправляем стандартный входной поток на file

Console.SetOut(file);

Console.WriteLine(a);

for (int i = 0; i < n; i++)

{

for (int j=0; j<m; j++) Console.Write("{0} ", mas[i,j]);

Console.WriteLine();

}

file.Close();

}

\_\_\_input.txt\_\_\_\_\_\_\_\_\_

3 4

1 4 2 8

4 9 0 1

5 7 4 2

При необходимости восстановить исходное состояние потока Console.In можно следующим образом:

TextWriter str = Console.In; // первоначально сохраняем исходное состояние входного потока

…

Console.SetIn(str); // при необходимости восстанавливаем исходное состояние входного потока

Аналогичным образом можно восстановить исходное состояние потока Console.Out:

TextWriter str = Console.Out; // первоначально сохраняем исходное состояние выходного потока

…

// при необходимости восстанавливаем исходное состояние выходного потока

Console.SetOut(str);

Вопрос**.** Для чего нужно два потока Console.Out и Console.Error, если они оба при стандартной работе выводят информацию на экран.

## Задания на лабораторную работу.

1. Работа с двоичными файлами:

Создать файл и записать в него вещественные числа из диапазона от a до b с шагом h. Вывести на экран все компоненты файла с нечетными порядковыми номерами.

Пример:

using System;

using System.Text;

using System.IO;

namespace MyProgram

{

class Program

{

static void Main()

{

Console.Write("a= ");

double a=double.Parse(Console.ReadLine());

Console.Write("b= ");

double b=double.Parse(Console.ReadLine());

Console.Write("h= ");

double h=double.Parse(Console.ReadLine());

//Записываем в файл t.dat вещественные числа из заданного диапазона

FileStream f=new FileStream("t.dat",FileMode.Open);

BinaryWriter fOut=new BinaryWriter(f);

for (double i=a; i<=b; i+=h)

{

fOut.Write(i);

}

fOut.Close();

//Объекты f и fIn связаны с одним и тем же файлом

f=new FileStream("t.dat",FileMode.Open);

BinaryReader fIn=new BinaryReader(f);

long m=f.Length; //определяем количество байт в потоке

//Читаем данные из файла t.dat начиная с элемента с номером 1, т.е с 8 байта,

//перемещая внутренний указатель на 16 байт, т.е. на два вещественных числа

for (long i=8; i<m; i+=16)

{

f.Seek(i,SeekOrigin.Begin);

a=fIn.ReadDouble();

Console.Write("{0:f2} ",a);

}

fIn.Close();

f.Close();

}

}

}

* 1. Создать файл и записать в него степени числа 3. Вывести на экран все компоненты файла с четным порядковым номером.
  2. Создать файл и записать в него обратные натуральные числа 1,1/2,...,1/n. Вывести на экран все компоненты файла с порядковым номером, кратным 3.
  3. Создать файл и записать в него n первых членов последовательности Фибоначчи. Вывести на экран все компоненты файла с порядковым номером, не кратным 3.
  4. Дана последовательность из n целых чисел. Создать файл и записать в него все четные числа последовательности. Вывести содержимое файла на экран.
  5. Дана последовательность из n целых чисел. Создать файл и записать в него все отрицательные числа последовательности. Вывести содержимое файла на экран.
  6. Дана последовательность из n целых чисел. Создать файл и записать в него числа последовательности, попадающие в заданный интервал. Вывести содержимое файла на экран.
  7. Дана последовательность из n целых чисел. Создать файл и записать в него числа последовательности, не кратные заданному числу. Вывести содержимое файла на экран.
  8. Дана последовательность из n вещественных чисел. Записать все эти числа в файл. Вывести на экран все компоненты, не попадающие в данный диапазон.
  9. Дана последовательность из n вещественных чисел. Записать все эти числа в файл. Вывести на экран все компоненты файла с нечетными номерами, большие заданного числа.
  10. Дана последовательность из n вещественных чисел. Записать все эти числа в файл. Вывести на экран все компоненты файла с четными номерами, меньшие заданного числа.
  11. Дана последовательность из n вещественных чисел. Записать все эти числа в файл. Вывести на экран все положительные компоненты файла.
  12. Дана последовательность из n вещественных чисел. Записать все эти числа в файл. Подсчитать среднее арифметическое компонентов файла, стоящих на четных позициях.
  13. Дана последовательность из n вещественных чисел. Записать все эти числа в файл. Найти максимальное значение среди компонентов файла, стоящих на нечетных позициях.
  14. Дано предложение. Создать файл и записать в него все символы данного предложения, отличные от знаков препинания. Вывести содержимое файла на экран.
  15. Дано предложение. Создать файл и записать в него все символы данного предложения, отличные от цифр. Вывести содержимое файла на экран.
  16. Создать файл, состоящий из слов. Вывести на экран все слова, которые начинаются на заданную букву.
  17. Создать файл, состоящий из слов. Вывести на экран все слова, длина которых равна заданному числу.
  18. Создать файл, состоящий из слов. Вывести на экран все слова, которые начинаются и заканчиваются одной буквой.
  19. Создать файл, состоящий из слов. Вывести на экран все слова, которые начинаются на ту же букву, что и последнее слово.

1. Работа с текстовым (символьным) файлом.

Дан текстовый файл. Найти количество строк, которые начинаются с данной буквы.

Пример:

**Замечание**: Напоминаем, что файл text.txt должен содержать английский текст

using System;

using System.Text;

using System.IO;

using System.Text.RegularExpressions;

namespace MyProgram

{

class Program

{

static void Main()

{

Console.Write("Введите заданную букву: ");

char a=char.Parse(Console.ReadLine());

StreamReader fileIn = new StreamReader("text.txt");

string text=fileIn.ReadToEnd(); //считываем из файла весь текст

fileIn.Close();

int k=0;

//разбиваем текст на слова используя регулярные выражения

string []newText=Regex.Split(text,"[ ,.:;]+");

//подсчитываем количество слов, начинающихся на заданную букву

foreach( string b in newText)

if (b[0]==a)++k;

Console.WriteLine("k= "+k);

}

}

}

* 1. Дан текстовый файл. Найти количество строк, которые начинаются и заканчиваются одной буквой.
  2. Дан текстовый файл. Найти самую длинную строку и ее длину.
  3. Дан текстовый файл. Найти самую короткую строку и ее длину.
  4. Дан текстовый файл. Найти номер самой длинной строки.
  5. Дан текстовый файл. Найти номер самой короткой строки.
  6. Дан текстовый файл. Выяснить, имеется ли в нем строка, которая начинается с данной буквы. Если да, то напечатать ее.
  7. Дан текстовый файл. Напечатать первый символ каждой строки.
  8. Дан текстовый файл. Напечатать символы с k1 по k2 в каждой строке.
  9. Дан текстовый файл. Напечатать все нечетные строки.
  10. Дан текстовый файл. Напечатать все строки, в которых имеется хотя бы один пробел.
  11. Дан текстовый файл. Напечатать все строки, длина которых равна данному числу.
  12. Дан текстовый файл. Напечатать все строки, длина которых меньше заданного числа.
  13. Дан текстовый файл. Напечатать все строки с номерами от k1 до k2.
  14. Дан текстовый файл. Получить слово, образованное k-ыми символами каждой строки.
  15. Дан текстовый файл. Переписать в новый файл все его строки, вставив в конец каждой строки ее номер.
  16. Дан текстовый файл. Переписать в новый файл все его строки, вставив в конец каждой строки количество символов в ней.
  17. Дан текстовый файл. Переписать в новый файл все его строки, длина которых больше заданного числа.
  18. Дан текстовый файл. Переписать в новый файл все его строки четной длины.
  19. Дан текстовый файл. Переписать в новый файл все его строки, удалив из них символы, стоящие на четных местах.

## Контрольные вопросы.

# C#. Файловая система

## Теоретические сведения.

### Работа с файловой системой

В пространстве имен System.IO предусмотрено четыре класса, которые предназначены для работы с файловой системой компьютера, т.е для создания, удаления переноса и т.д. файлов и каталогов.

Первые два типа - Directory и Fi1е реализуют свои возможности с помощью статических методов, поэтому данные классы можно использовать без создания соответствующих объектов (экземпляров классов).

Следующие типы - DirectoryInfo и FileInfo обладают схожими функциональными возможностями c Directory и Fi1е, но порождены от класса FileSystemInfo и поэтому реализуются путем создания соответствующих экземпляров классов.

### Работа с каталогами

##### Абстрактный класс FileSystemInfo

Значительная часть членов FileSystemInfo предназначена для работы с общими характеристиками файла или каталога (метками времени, атрибутами и т. п.). Рассмотрим некоторые свойства FileSystemInfo:

|  |  |
| --- | --- |
| Свойство | Описание |
| Attributes | Позволяет получить или установить атрибуты для данного объекта файловой системы. Для этого свойства используются значения и перечисления FileAttributes |
| CreationTime | Позволяет получить или установить время создания объекта файловой системы |
| Exists | Может быть использовано для того, чтобы определить, существует ли данный объект файловой системы |
| Extension | Позволяет получить расширение для файла |
| FullName | Возвращает имя файла или каталога с указанием пути к нему в файловой системе |
| LastAccessTime | Позволяет получить или установить время последнего обращения к объекту файловой системы |
| LastWriteTime | Позволяет получить или установить время последнего внесения изменений в объект файловой системы |
| Name | Возвращает имя указанного файла. Это свойство доступно только для чтения. Для каталогов возвращает имя последнего каталога в иерархии, если это возможно. Если нет, возвращает полностью определенное имя |

В FileSystemInfo предусмотрено и несколько методов. Например, метод Delete() - позволяет удалить объект файловой системы с жесткого диска, a Refresh() - обновить информацию об объекте файловой системы.

##### Класс DirectoryInfo

Данный класс наследует члены класса FileSystemInfo и содержит дополнительный набор членов, которые предназначены для создания, перемещения, удаления, получения информации о каталогах и подкаталогах в файловой системе. Наиболее важные члены класса содержатся в следующей таблице:

|  |  |
| --- | --- |
| Член | Описание |
| Create()CreateSubDirectory() | Создают каталог (или подкаталог) по указанному пути в файловой системе |
| Delete() | Удаляет пустой каталог |
| GetDirectories() | Позволяет получить доступ к подкаталогам текущего каталога (в виде массива объектов DirectoryInfo ) |
| GetFiles() | Позволяет получить доступ к файлам текущего каталога (в виде массива объектов FileInfo ) |
| MoveTo() | Перемещает каталог и все его содержимое на новый адрес в файловой системе |
| Parent | Возвращает *родительский каталог* в иерархии файловой системы |

Работа с типом DirectoryInfo начинается с того, что мы создаем экземпляр класса (объект), указывая при вызове конструктора в качестве параметра путь к нужному каталогу. Если мы хотим обратиться к текущему каталогу (то есть каталогу, в котором в настоящее время производится выполнение приложения), вместо параметра используется обозначение ".". Например:

// Создаем объект DirectoryInfo, которому будет обращаться к текущему каталогу

DirectoryInfo dir1 = new DirectoryInfo(".");

// Создаем объект DirectoryInfo, которому будет обращаться к каталогу d:\prim

DirectoryInfo dir2 = new DirectoryInfo(@"d:\prim");

Если мы попытаемся создать объект DirectoryInfo, связав его с несуществующим каталогом, то будет сгенерировано исключение System.IO.DirectoryNotFoundException. Если же все нормально, то мы сможем получить доступ к данному каталогу. В примере, который приведен ниже, мы создаем объект DlrectoryInfo, который связан с каталогом d:\prim, и выводим информацию о данном каталоге:

using System;

using System.Text;

using System.IO;

namespace MyProgram

{

class Program

{

static void Main(string[] args)

{

DirectoryInfo dir = new DirectoryInfo(@"d:\prim");

Console.WriteLine("\*\*\*\*\* "+dir.Name+" \*\*\*\*\*");

Console.WriteLine("FullName: {0}", dir.FullName);

Console.WriteLine("Name: {0}", dir.Name);

Console.WriteLine("Parent: {0}", dir.Parent);

Console.WriteLine("Creation: {0}", dir.CreationTime);

Console.WriteLine("Attributes: {0}", dir.Attributes.ToString());

Console.WriteLine("Root: {0}", dir.Root);

}

}

}

Свойство Attributes позволяет получить информацию об атрибутах объекта файловой системы. Возможные значения данного свойства приведены в следующей таблице:

|  |  |
| --- | --- |
| Значение | Описание |
| Archive | Этот атрибут используется приложениями при проведении резервного копирования, а в некоторых случаях - удаления старых файлов |
| Compressed | Определяет, что файл является сжатым |
| Directory | Определяет, что объект файловой системы является каталогом |
| Encrypted | Определяет, что файл является зашифрованным |
| Hidden | Определяет, что файл является скрытым (такой файл не будет выводиться при обычном просмотре каталога) |
| Normal | Определяет, что файл находится в обычном состоянии и для него установлены любые другие атрибуты. Этот атрибут не может использоваться с другими атрибутами |
| Offline | Файл (расположенный на сервере) кэширован в хранилище off-line на клиентском компьютере. Возможно, что данные этого файла уже устарели |
| Readonly | Файл доступен только для чтения |
| System | Файл является системным (то есть файл является частью операционной системы или используется исключительно операционной системой) |

Через DirectoryInfo можно не только получать доступ к информации о текущем каталоге, но получить доступ к информации о его подкаталогах:

class Program

{

static void printDirect( DirectoryInfo dir)

{

Console.WriteLine("\*\*\*\*\* "+dir.Name+" \*\*\*\*\*");

Console.WriteLine("FullName: {0}", dir.FullName);

Console.WriteLine("Name: {0}", dir.Name);

Console.WriteLine("Parent: {0}", dir.Parent);

Console.WriteLine("Creation: {0}", dir.CreationTime);

Console.WriteLine("Attributes: {0}", dir.Attributes.ToString());

Console.WriteLine("Root: {0}", dir.Root);

}

static void Main(string[] args)

{

DirectoryInfo dir = new DirectoryInfo(@"d:\prim");

printDirect(dir);

DirectoryInfo[] subDirects = dir.GetDirectories();

Console.WriteLine("Найдено {0} подкаталогов", subDirects.Length);

foreach (DirectoryInfo d in subDirects)

{

printDirect(d);

}

}

}

Метод CreateSubdirectory() позволяет создать в выбранном каталоге как единственный подкаталог, так и множество подкаталогов (в том числе, и вложенных друг в друга). Создадим в каталоге несколько дополнительных подкаталогов:

DirectoryInfo dir = new DirectoryInfo(@"d:\prim");

dir.CreateSubdirectory("doc"); //создали подкаталог

dir.CreateSubdirectory(@"book\2008"); //создали вложенный подкаталог

Метод MoveTo() позволяет переместить текущий каталог по заданному в качестве параметра адресу. При этом возможно произвести переименование каталога. Например:

DirectoryInfo dir = new DirectoryInfo(@"d:\prim\bmp");

dir.MoveTo(@"d:\prim\letter\bmp");

В данном случае каталог bmp перемещается в по адресу d:\prim\letter\bmp. Так как имя перемещаемого каталога совпадает с крайним правым именем в адресе нового местоположения каталога, то переименования не происходит. Следующий пример позволит нам переименовать текущий каталог:

DirectoryInfo dir = new DirectoryInfo(@"d:\prim\letter");

dir.MoveTo(@"d:\prim\archive");

##### Класс Directory

Работать с каталогами файловой системы компьютера можно и при помощи класса Directory, функциональные возможности которого во многом совпадают с возможностями DirectoryInfo. Но члены данного класса реализованы статически, поэтому для их использования нет необходимости создавать объект.

Рассмотрим работу с методами данного класса на примерах.

Directory.CreateDirectory(@"d:\prim\2008");//создали подкаталог 2008

Directory.Move(@"d:\prim\bmp",

@"d:\prim\2008\bmp");//перенесли каталог bmp в каталог 2008

Directory.Move(@"d:\prim\letter",

@"d:\prim\archives");//переименовали каталог letter в archives

Замечания.

1. Удаление каталога возможно только тогда, когда он пуст.
2. На практике комбинируют использование классов Directory и DirectoryInfo.

#### Работа с файлами

##### Класс Filelnfo

Класс Filelnfo предназначен для организации доступа к физическому файлу, который содержится на жестком диске компьютера. Он позволяет получать информацию об этом файле (например, о времени его создания, размере, атрибутах и т. п.), а также производить различные операции, например, по созданию файла или его удалению. Класс FileInfo наследует члены класса FileSystemInfo и содержит дополнительный набор членов, который приведен в следующей таблице:

|  |  |
| --- | --- |
| Член | Описание |
| AppendText() | Создает объект StreamWriter для добавления текста к файлу |
| CopyTo() | Копирует уже существующий файл в новый файл |
| Create() | Создает новый файл и возвращает объект FileStream для взаимодействия с этим файлом |
| CreateText() | Создает объект StreamWriter для записи текстовых данных в новый файл |
| Delete() | Удаляет файл, которому соответствует объект FileInfo |
| Directory | Возвращает каталог, в котором расположен данный файл |
| DirectoryName | Возвращает полный путь к данному файлу в файловой системе |
| Length | Возвращает размер файла |
| MoveTo() | Перемещает файл в указанное пользователем место (этот метод позволяет одновременно переименовать данный файл) |
| Name | Позволяет получить имя файла |
| Ореn() | Открывает файл с указанными пользователем правами доступа на чтение, запись или совместное использование с другими пользователями |
| OpenRead() | Создает объект FileStream, доступный только для чтения |
| OpenText() | Создает объект StreamReader (о нем также будет рассказано ниже), который позволяет считывать информацию из существующего текстового файла |
| OpenWrite() | Создает объект FileStream, доступный для чтения и записи |

Большинство методов FileInfo возвращает объекты ( FIleStream, StreamWriter, StreamReader и т. п.), которые позволяют различным образом взаимодействовать с файлом, например, производить чтение или запись в него. Приемы работы с данными потоками нам уже известны. Поэтому рассмотрим другие возможности класса FileInfo.

using System;

using System.Text;

using System.IO; //для работы с файловым вводом-выводом

using System.Text.RegularExpressions;

namespace MyProgram

{

class Program

{

static void Main()

{

//создаем новый файл и связываем с ним строковый поток

FileInfo f = new FileInfo("text.txt");

StreamWriter fOut = new StreamWriter(f.Create());

//записываем в файл данные и закрываем строковый поток,

// при этом связь с физическим файлом для f не рвется

fOut.WriteLine("ОДИН ДВА ТРИ...");

fOut.Close();

//получаем информацию о файле

Console.WriteLine("\*\*\*\*\*\*\*"+f.Name File Inf+"\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

Console.WriteLine("File size: {0}", f.Length);

Console.WriteLine("Creation: {0}", f.CreationTime);

Console.WriteLine("Attributes: {0}", f.Attributes.ToString());

}

}

}

Рассмотрим следующий пример:

static void Main()

{

FileInfo f = new FileInfo(@"d:\prim\letter\letter1.txt");

f.CopyTo(@"d:\prim\bmp\letter.txt");

Directory.CreateDirectory(@"d:\prim\archives");

f.MoveTo(@"d:\prim\archives\letter1.txt");

f = new FileInfo(@"d:\prim\letter\letter2.txt");

f.Delete();

}

**Задание**. Посмотрите, что произойдет, если запустить эту программу еще раз. Дайте этому объяснение.

Рассмотрим еще один пример по удалению файлов:

static void printFile( FileInfo file)

{

Console.WriteLine("\*\*\*\*\* "+file.Name+" \*\*\*\*\*");

Console.WriteLine("File size: {0}", file.Length);

Console.WriteLine("Creation: {0}", file.CreationTime);

Console.WriteLine("Attributes: {0}", file.Attributes.ToString());

}

static void Main(string[] args)

{

DirectoryInfo dir = new DirectoryInfo(@"d:\prim\bmp");

FileInfo[] files = dir.GetFiles();

if (files.Length!=0)

{

Console.WriteLine("Найдено {0} файла", files.Length);

foreach (FileInfo f in files)

{

printFile(f);

f.Delete();

}

Console.WriteLine("\nТеперь в каталоге содержится {0} файлов и можно его удалить",

dir.GetFiles().Length);

dir.Delete();

}

}

##### Класс File

Доступ к физическим файлам можно получать и через статические методы класса File. Большинство методов объекта Fileinfo представляют в этом смысле зеркальное отражение методов объекта File.

Пример.

static void Main(string[] args)

{

File.Copy(@"d:\prim\letter\letter1.txt",@"d:\prim\bmp\letter1.txt");

Directory.CreateDirectory(@"d:\prim\archives");

File.Move(@"d:\prim\letter\letter1.txt",@"d:\prim\archives\letter1.txt");

File.Delete(@"d:\prim\letter\letter2.txt");

Directory.Delete(@"d:\prim\letter");

}

Имеет прямой смысл использовать статический класс File, когда требуется осуществить единственный вызов метода на объект. В этом случае вызов будет выполнен быстрее, поскольку .NET Framework не придется проходить через процедуру создания экземпляра нового объекта с последующим вызовом метода. Однако если приложение осуществляет несколько операций над файлом, то более разумным представляется создать экземпляр объекта FileInfo и использовать его методы. Это позволит сэкономить определенное время, поскольку объект будет заранее настроен на нужный файл в файловой системе, в то время как статическому классу придется каждый раз осуществлять его поиск заново.

Аналогичное правило действует и при выборе между классами Directory и DirectoryInfo.

## Задания на лабораторную работу.

Программным путем:

1. В папке С:\temp создайте папки К1 и К2.
2. В папке К1:
   1. создайте файл t1.txt, в который запишите следующий текст :

Иванов Иван Иванович, 1965 года рождения, место жительства г. Саратов

* 1. создайте файл t2.txt, в который запишите следующий текст:

Петров Сергей Федорович, 1966 года рождения, место жительства г.Энгельс

1. В папке К2 создайте файл t3.txt, в который перепишите вначале текст из файла t1.txt, а затем из t2.txt
2. Выведите развернутую информацию о созданных файлах.
3. Файл t2.txt перенесите в папку K2.
4. Файл t1.txt скопируйте в папку K2.
5. Папку K2 переименуйте в ALL, а папку K1 удалите.
6. Вывести полную информацию о файлах папки All.

## Контрольные вопросы.

# Интерфейсы. Контейнерные классы.

## Теоретические сведения.

### Синтаксис интерфейса

*Интерфейс* является "крайним случаем" абстрактного класса. В нем задается набор абстрактных методов, свойств и индексаторов, которые должны быть реализованы в производных классах. Иными словами, интерфейс определяет поведение, которое поддерживается реализующими этот интерфейс классами. Основная идея использования интерфейса состоит в том, чтобы к объектам таких классов можно было обращаться одинаковым образом.

Каждый класс может определять элементы интерфейса по-своему. Так достигается полиморфизм: объекты разных классов по-разному реагируют на вызовы одного и того же метода.

Синтаксис интерфейса аналогичен синтаксису класса:

[ атрибуты ] [ спецификаторы ] interface имя\_интерфейса [ : предки ]

тело\_интерфейса [ ; ]

Для интерфейса могут быть указаны спецификаторы new, public, protected, internal и private. Спецификатор newприменяется для вложенных интерфейсов и имеет такой же смысл, как и соответствующий модификатор метода класса. Остальные спецификаторы управляют видимостью интерфейса. По умолчанию интерфейс доступен только из сборки, в которой он описан ( internal ).

Интерфейс может наследовать свойства нескольких интерфейсов, в этом случае *предки* перечисляются через запятую. *Тело интерфейса* составляют абстрактные методы, шаблоны свойств и индексаторов, а также события.

В качестве примера рассмотрим интерфейс IAction, определяющий базовое поведение персонажей компьютерной игры, встречавшихся в предыдущих главах. Допустим, что любой персонаж должен уметь выводить себя на экран, атаковать и красиво умирать:

interface IAction

{

void Draw();

int Attack(int a);

void Die();

int Power { get; }

}

В интерфейсе IAction заданы заголовки трех методов и шаблон свойства Power, доступного только для чтения. Если бы требовалось обеспечить возможность установки свойства, в шаблоне следовало указать ключевое слово set, например:

int Power { get; set; }

##### Отличия интерфейса от абстрактного класса:

* элементы интерфейса по умолчанию имеют спецификатор доступа public и не могут иметь спецификаторов, заданных явным образом;
* интерфейс не может содержать полей и обычных методов — все элементы интерфейса должны быть абстрактными;
* класс, в списке предков которого задается интерфейс, должен определять *все* его элементы, в то время как потомок абстрактного класса может не переопределять часть абстрактных методов предка (в этом случае производный класс также будет абстрактным);
* класс может иметь в списке предков несколько интерфейсов, при этом он должен определять все их методы.

### Реализация интерфейса

В списке предков класса сначала указывается его базовый класс, если он есть, а затем через запятую интерфейсы, которые реализует этот класс. Например, реализация интерфейса IAction в классе Monster может выглядеть следующим образом:

using System;

namespace ConsoleApplication1

{

interface IAction

{

void Draw();

int Attack( int a );

void Die();

int Power { get; }

}

class Monster : IAction

{

public void Draw()

{

Console.WriteLine( "Здесь был " + name );

}

public int Attack( int ammo\_ )

{

ammo -= ammo\_;

if ( ammo > 0 ) Console.WriteLine( "Ба-бах!" );

else ammo = 0;

return ammo;

}

public void Die()

{

Console.WriteLine( "Monster " + name + " RIP" );

health = 0;

}

public int Power

{

get

{

return ammo \* health;

}

}

…

}

Сигнатуры методов в интерфейсе и реализации должны полностью совпадать. Для реализуемых элементов интерфейса в классе следует указывать спецификатор public. К этим элементам можно обращаться как через объект класса, так и через объект типа соответствующего интерфейса:

Monster Vasia = new Monster( 50, 50, "Вася" ); // объект класса Monster

Vasia.Draw(); // результат: Здесь был Вася

IAction Actor = new Monster( 10, 10, "Маша" ); // объект типа интерфейса

Actor.Draw(); // результат: Здесь был Маша

Существует второй способ реализации интерфейса в классе: *явное указание имени интерфейса* перед реализуемым элементом. Спецификаторы доступа при этом не указываются. К таким элементам можно обращаться в программе *только через объект типа интерфейса*, например:

class Monster : IAction

{

int IAction.Power

{

get

{

return ammo \* health;

}

}

void IAction.Draw()

{

Console.WriteLine( "Здесь был " + name );

}

...

}

...

IAction Actor = new Monster( 10, 10, "Маша" );

Actor.Draw(); // обращение через объект типа интерфейса

// Monster Vasia = new Monster( 50, 50, "Вася" );

// Vasia.Draw();

Таким образом, при явном задании имени реализуемого интерфейса соответствующий метод *не входит в интерфейс класса*. Это позволяет упростить его в том случае, если какие-то элементы интерфейса не требуются конечному пользователю класса.

### Работа с объектами через интерфейсы. Операции is и as

При работе с объектом через объект типа интерфейса бывает необходимо убедиться, что объект поддерживает данный интерфейс. Проверка выполняется с помощью бинарной операции is. Эта операция определяет, совместим ли текущий тип объекта, находящегося слева от ключевого слова is, с типом, заданным справа.

Результат операции равен true, если объект можно преобразовать к заданному типу, и false в противном случае. Операция обычно используется в следующем контексте:

{

// выполнить преобразование "объекта" к "типу"

// выполнить действия с преобразованным объектом

}

Допустим, мы оформили какие-то действия с объектами в виде метода с параметром типа object. Прежде чем использовать этот параметр внутри метода для обращения к методам, описанным в производных классах, требуется выполнить преобразование к производному классу. Для безопасного преобразования следует проверить, возможно ли оно, например так:

static void Act( object A )

{

if ( A is IAction )

{

IAction Actor = (IAction) A;

Actor.Draw();

}

}

В метод Act можно передавать любые объекты, но на экран будут выведены только те, которые поддерживают интерфейс IAction.

Недостатком использования операции is является то, что преобразование фактически выполняется дважды: при проверке и при собственно преобразовании. Более эффективной является другая операция — as. Она выполняет преобразование к заданному типу, а если это невозможно, формирует результат null, например:

static void Act( object A )

{

IAction Actor = A as IAction;

if ( Actor != null ) Actor.Draw();

}

Обе рассмотренные операции применяются как к интерфейсам, так и к классам.

### Интерфейсы и наследование

Интерфейс может не иметь или иметь сколько угодно интерфейсов-предков, в последнем случае он наследует все элементы всех своих базовых интерфейсов, начиная с самого верхнего уровня.

Как и в обычной иерархии классов, базовые интерфейсы определяют общее поведение, а их потомки конкретизируют и дополняют его. В интерфейсе-потомке можно также указать элементы, переопределяющие унаследованные элементы с такой же сигнатурой. В этом случае перед элементом указывается ключевое слово new, как и в аналогичной ситуации в классах. С помощью этого слова соответствующий элемент базового интерфейса скрывается. Класс, реализующий интерфейс, должен определять все его элементы, в том числе унаследованные.

Интерфейс, на собственные или унаследованные элементы которого имеется явная ссылка, должен быть указан в списке предков класса.

Класс наследует все методы своего предка, в том числе те, которые реализовывали интерфейсы. Он может переопределить эти методы с помощью спецификатора new, но обращаться к ним можно будет только через объект класса. Если использовать для обращения ссылку на интерфейс, вызывается не переопределенная версия:

interface IBase

{

void A();

}

class Base : IBase

{

public void A() { ... }

}

class Derived: Base

{

new public void A() { ... }

}

...

Derived d = new Derived ();

d.A(); // вызывается Derived.A();

IBase id = d;

id.A(); // вызывается Base.A();

Однако если интерфейс реализуется с помощью виртуального метода класса, после его переопределения в потомке любой вариант обращения (через класс или через интерфейс) приведет к одному и тому же результату. Метод интерфейса, реализованный явным указанием имени, объявлять виртуальным запрещается.

Существует возможность *повторно реализовать интерфейс*, указав его имя в списке предков класса наряду с классом-предком, уже реализовавшим этот интерфейс. При этом реализация переопределенных методов базового класса во внимание не принимается:

interface IBase

{

void A();

}

class Base : IBase

{

void IBase.A() { ... } // не используется в Derived

}

class Derived : Base, IBase

{

public void A() { ... }

}

Если класс наследует от класса и интерфейса, которые содержат методы с одинаковыми сигнатурами, унаследованный метод класса воспринимается как реализация интерфейса. Вообще при реализации интерфейса учитывается наличие "подходящих" методов в классе независимо от их происхождения. Это могут быть методы, описанные в текущем или базовом классе, реализующие интерфейс явным или неявным образом.

### Стандартные интерфейсы .NET

В библиотеке классов .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Например, интерфейс IComparable задает метод сравнения объектов на больше-меньше, что позволяет выполнять их сортировку. Реализация интерфейсов IEnumerable и IEnumerator дает возможность просматривать содержимое объекта с помощью конструкции foreach, а реализация интерфейса *ICloneable* — клонировать объекты.

Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки. Например, работа с массивами с помощью цикла foreach возможна именно потому, что тип Array реализует интерфейсы IEnumerable и IEnumerator. Можно создавать и собственные классы, поддерживающие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами.

#### Сравнение объектов

Интерфейс IComparable определен в пространстве имен System. Он содержит всего один метод CompareTo, возвращающий результат сравнения двух объектов — текущего и переданного ему в качестве параметра:

interface IComparable

{

int CompareTo( object obj )

}

Метод должен возвращать:

* 0, если текущий объект и параметр равны;
* отрицательное число, если текущий объект меньше параметра;
* положительное число, если текущий объект больше параметра.

Реализуем интерфейс IComparable в знакомом нам классе Monster. В качестве критерия сравнения объектов выберем поле health. В [листинге 9.1](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=2#example.9.1) приведена программа, сортирующая массив монстров по возрастанию величины, характеризующей их здоровье.

using System;

namespace ConsoleApplication1

{

class Monster : IComparable

{

public Monster( int health, int ammo, string name )

{

this.health = health;

this.ammo = ammo;

this.name = name;

}

virtual public void Passport()

{

Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",

name, health, ammo );

}

public int CompareTo( object obj ) // реализация интерфейса

{

Monster temp = (Monster) obj;

if ( this.health > temp.health ) return 1;

if ( this.health < temp.health ) return -1;

return 0;

}

string name;

int health, ammo;

}

class Class1

{ static void Main()

{

const int n = 3;

Monster[] stado = new Monster[n];

stado[0] = new Monster( 50, 50, "Вася" );

stado[1] = new Monster( 80, 80, "Петя" );

stado[2] = new Monster( 40, 10, "Маша" );

Array.Sort( stado ); // сортировка стала возможной

foreach ( Monster elem in stado ) elem.Passport();

}

}

}

Листинг 1. Пример реализации интерфейса IComparable

Результат работы программы:

Monster Маша health = 40 ammo = 10

Monster Вася health = 50 ammo = 50

Monster Петя health = 80 ammo = 80

Во многих алгоритмах требуется выполнять сортировку объектов по различным критериям. В C# для этого используется интерфейс IComparer, который мы рассмотрим далее.

#### Сортировка по разным критериям (интерфейс IComparer)

Интерфейс IComparer определен в пространстве имен System.Collections. Он содержит один метод Compare, возвращающий результат сравнения двух объектов, переданных ему в качестве параметров:

interface IComparer

{

int Compare( object ob1, object ob2 )

}

Принцип применения этого интерфейса состоит в том, что для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий этот интерфейс. Объект этого класса передается в стандартный метод сортировки массива в качестве второго аргумента.

Пример сортировки массива объектов из предыдущего листинга по именам (свойство Name, класс SortByName ) и количеству вооружений (свойство Ammo, класс SortByAmmo ) приведен в [листинге 2](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=3#example.9.2).

using System;

using System.Collections;

namespace ConsoleApplication1

{

class Monster

{

public Monster( int health, int ammo, string name )

{

this.health = health;

this.ammo = ammo;

this.name = name;

}

public int Ammo

{

get { return ammo; }

set

{

if (value > 0) ammo = value;

else ammo = 0;

}

}

public string Name

{

get { return name; }

}

virtual public void Passport()

{

Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",

name, health, ammo );

}

public class SortByName : IComparer //

{

int IComparer.Compare( object ob1, object ob2 )

{

Monster m1 = (Monster) ob1;

Monster m2 = (Monster) ob2;

return String.Compare( m1.Name, m2.Name );

}

}

public class SortByAmmo : IComparer //

{

int IComparer.Compare( object ob1, object ob2 )

{

Monster m1 = (Monster) ob1;

Monster m2 = (Monster) ob2;

if ( m1.Ammo > m2.Ammo ) return 1;

if ( m1.Ammo < m2.Ammo ) return -1;

return 0;

}

}

string name;

int health, ammo;

}

class Class1

{ static void Main()

{

const int n = 3;

Monster[] stado = new Monster[n];

stado[0] = new Monster( 50, 50, "Вася" );

stado[1] = new Monster( 80, 80, "Петя" );

stado[2] = new Monster( 40, 10, "Маша" );

Console.WriteLine( "Сортировка по имени:" );

Array.Sort( stado, new Monster.SortByName() );

foreach ( Monster elem in stado ) elem.Passport();

Console.WriteLine( "Сортировка по вооружению:" );

Array.Sort( stado, new Monster.SortByAmmo() );

foreach ( Monster elem in stado ) elem.Passport();

}

}

}

Листинг 2. Сортировка по двум критериям

Результат работы программы:

Сортировка по имени:

Monster Вася health = 50 ammo = 50

Monster Маша health = 40 ammo = 10

Monster Петя health = 80 ammo = 80

Сортировка по вооружению:

Monster Маша health = 40 ammo = 10

Monster Вася health = 50 ammo = 50

Monster Петя health = 80 ammo = 80

#### Перегрузка операций отношения

Если класс реализует интерфейс IComparable, его экземпляры можно сравнивать между собой на больше-меньше. Логично разрешить использовать для этого операции отношения, перегрузив их. Операции должны перегружаться парами: < и >, <= и >=, == и !=. *Перегрузка операций* обычно выполняется путем делегирования, то есть обращения к переопределенным методам CompareTo и Equals.

**Обязательно**

*Если класс реализует интерфейс*IComparable, *требуется переопределить метод*Equals *и связанный с ним метод* GetHashCode. *Оба метода унаследованы от базового класса*object.

В [листинге 3](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=3#example.9.3) операции отношения перегружены для класса Monster. В качестве критерия сравнения объектов на больше-меньше выступает поле health, а при сравнении на равенство попарно сравниваются все поля объектов

using System;

namespace ConsoleApplication1

{

class Monster : IComparable

{

public Monster( int health, int ammo, string name )

{

this.health = health;

this.ammo = ammo;

this.name = name;

}

public override bool Equals( object obj )

{

if ( obj == null || GetType() != obj.GetType() ) return false;

Monster temp = (Monster) obj;

return health == temp.health &&

ammo == temp.ammo &&

name == temp.name;

}

public override int GetHashCode()

{

return name.GetHashCode();

}

public static bool operator == ( Monster a, Monster b )

{

return a.Equals( b );

}

public static bool operator != ( Monster a, Monster b )

{

return ! a.Equals( b );

}

public static bool operator < ( Monster a, Monster b )

{

return ( a.CompareTo( b ) < 0 );

}

public static bool operator > ( Monster a, Monster b )

{

return ( a.CompareTo( b ) > 0 );

}

public static bool operator <= ( Monster a, Monster b )

{

return ( a.CompareTo( b ) <= 0 );

}

public static bool operator >= ( Monster a, Monster b )

{

return ( a.CompareTo( b ) >= 0 );

}

public int CompareTo( object obj )

{

Monster temp = (Monster) obj;

if ( this.health > temp.health ) return 1;

if ( this.health < temp.health ) return -1;

return 0;

}

string name;

int health, ammo;

}

class Class1

{ static void Main()

{

Monster Вася = new Monster( 70, 80, "Вася" );

Monster Петя = new Monster( 80, 80, "Петя" );

if ( Вася > Петя ) Console.WriteLine( "Вася больше Пети");

else if ( Вася == Петя ) Console.WriteLine( "Вася == Петя");

else Console.WriteLine( "Вася меньше Пети");

}

}

}

Листинг 3. Перегрузка операций отношения

Результат работы программы:

Вася меньше Пети

#### Клонирование объектов

*Клонирование* — это создание копии объекта. Копия объекта называется клоном. Как известно, при присваивании одного объекта ссылочного типа другому копируется ссылка, а не сам объект. Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом MemberwiseClone, который любой объект наследует от класса object. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются. Это называется *поверхностным клонированием*.

Для создания полностью независимых объектов необходимо *глубокое клонирование*, когда в памяти создается дубликат всего дерева объектов, то есть объектов, на которые ссылаются поля объекта, поля полей, и так далее. Алгоритм глубокого клонирования весьма сложен, поскольку требует *рекурсивного обхода* всех ссылок объекта и отслеживания циклических зависимостей.

Объект, имеющий собственные алгоритмы клонирования, должен объявляться как наследник интерфейса *ICloneable* и переопределять его единственный метод Clone. В [листинге 9.4](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=4#example.9.4) приведен пример создания поверхностной копии объекта класса Monster с помощью метода MemberwiseClone, а также реализован интерфейс *ICloneable*. В демонстрационных целях в имя клона объекта добавлено слово "Клон".

using System;

namespace ConsoleApplication1

{

class Monster : ICloneable

{

public Monster( int health, int ammo, string name )

{

this.health = health;

this.ammo = ammo;

this.name = name;

}

public Monster ShallowClone() // поверхностная копия

{

return (Monster)this.MemberwiseClone();

}

public object Clone() // пользовательская копия

{

return new Monster(this.health, this.ammo, "Клон " + this.name);

}

virtual public void Passport()

{

Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",

name, health, ammo );

}

string name;

int health, ammo;

}

class Class1

{ static void Main()

{

Monster Вася = new Monster( 70, 80, "Вася" );

Monster X = Вася;

Monster Y = Вася.ShallowClone();

Monster Z = (Monster)Вася.Clone();

...

}

}

}

Листинг 4. Клонирование объектов

Объект Х ссылается на ту же область памяти, что и объект Вася. Следовательно, если мы внесем изменения в один из этих объектов, это отразится на другом. Объекты Y и Z, созданные путем клонирования, обладают собственными копиями значений полей и независимы от исходного объекта.

#### Перебор объектов (интерфейс IEnumerable) и итераторы

Оператор foreach является удобным средством перебора элементов объекта. Массивы и все стандартные коллекции библиотеки .NET позволяют выполнять такой перебор благодаря тому, что в них реализованы интерфейсы IEnumerable и IEnumerator. Для применения оператора foreach к пользовательскому типу данных требуется реализовать в нем эти интерфейсы.

Интерфейс IEnumerable ( *перечислимый* ) определяет всего один метод — GetEnumerator, возвращающий объект типа IEnumerator ( *перечислитель* ), который можно использовать для просмотра элементов объекта.

Интерфейс IEnumerator задает три элемента:

* свойство Current, возвращающее текущий элемент объекта;
* метод MoveNext, продвигающий перечислитель на следующий элемент объекта;
* метод Reset, устанавливающий перечислитель в начало просмотра.

Цикл foreach использует эти методы для перебора элементов, из которых состоит объект.

Таким образом, если требуется, чтобы для перебора элементов класса мог применяться цикл foreach, необходимо реализовать четыре метода: GetEnumerator, Current, MoveNext и Reset..

*Итератор*  представляет собой блок кода, задающий последовательность перебора элементов объекта. На каждом проходе цикла foreach выполняется один шаг итератора, заканчивающийся выдачей очередного значения. Выдача значения выполняется с помощью ключевого слова *yield*.

Рассмотрим создание итератора на примере ([листинг 5](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=4#example.9.5)). Пусть требуется создать объект, содержащий боевую группу экземпляров типа Monster.

using System;

using System.Collections;

namespace ConsoleApplication1

{

class Monster { ... }

class Daemon { ... }

class Stado : IEnumerable // 1

{

private Monster[] mas;

private int n;

public Stado()

{

mas = new Monster[10];

n = 0;

}

public IEnumerator GetEnumerator()

{

for ( int i = 0; i < n; ++i ) yield return mas[i]; // 2

}

public void Add( Monster m )

{

if ( n >= 10 ) return;

mas[n] = m;

++n;

}

}

class Class1

{ static void Main()

{

Stado s = new Stado();

s.Add( new Monster() );

s.Add( new Monster("Вася") );

s.Add( new Daemon() );

foreach ( Monster m in s ) m.Passport();

}

}

}

Листинг 5. Класс с итератором

Все, что требуется сделать в версии 2.0 для поддержки перебора — указать, что класс реализует интерфейс IEnumerable (оператор 1), и описать итератор (оператор 2). Доступ к нему может быть осуществлен через методы MoveNext и Current интерфейса IEnumerator.

Преимущество использования итераторов заключается в том, что для одного и того же класса можно задать различный порядок перебора элементов. В [листинге 6](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=4#example.9.6) описаны две дополнительные стратегии перебора элементов класса Stado, введенного в [листинге 5](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=4#example.9.5) — перебор в обратном порядке и выборка только тех объектов, которые являются экземплярами класса Monster.

using System;

using System.Collections;

using MonsterLib;

namespace ConsoleApplication1

{

class Monster { ... }

class Daemon { ... }

class Stado : IEnumerable

{

private Monster[] mas;

private int n;

public Stado()

{

mas = new Monster[10];

n = 0;

}

public IEnumerator GetEnumerator()

{

for ( int i = 0; i < n; ++i ) yield return mas[i];

}

public IEnumerable Backwards() // в обратном порядке

{

for ( int i = n - 1; i >= 0; --i ) yield return mas[i];

}

public IEnumerable MonstersOnly() // только монстры

{

for ( int i = 0; i < n; ++i )

if ( mas[i].GetType().Name == "Monster" )

yield return mas[i];

}

public void Add( Monster m )

{

if ( n >= 10 ) return;

mas[n] = m;

++n;

}

}

class Class1

{ static void Main()

{

Stado s = new Stado();

s.Add( new Monster() );

s.Add( new Monster("Вася") );

s.Add( new Daemon() );

foreach ( Monster i in s ) i.Passport();

foreach ( Monster i in s.Backwards() ) i.Passport();

foreach ( Monster i in s.MonstersOnly() ) i.Passport();

}

}

}

Листинг 6. Реализация нескольких стратегий перебора

Теперь, когда вы получили представление об итераторах, рассмотрим их более формально. Блок итератора синтаксически представляет собой обычный блок и может встречаться в теле метода, операции или части get свойства, если соответствующее возвращаемое значение имеет тип IEnumerable или IEnumerator.

В теле блока итератора могут встречаться две конструкции:

* *yield* return формирует значение, выдаваемое на очередной итерации;
* *yield* break сигнализирует о завершении итерации.

Ключевое слово *yield* имеет специальное значение для компилятора только в этих конструкциях.

Код блока итератора выполняется не так, как обычные блоки. Компилятор формирует служебный *объект-перечислитель*, при вызове метода MoveNext которого выполняется код блока итератора, выдающий очередное значение с помощью ключевого слова *yield*. Следующий вызов метода MoveNext объекта-перечислителя возобновляет выполнение блока итератора с момента, на котором он был приостановлен в предыдущий раз.

### Контейнерные классы

#### Абстрактные структуры данных

Любая программа предназначена для обработки данных, от способа организации которых зависит ее алгоритм. Для разных задач необходимы различные способы хранения и обработки данных, поэтому выбор структур данных должен предшествовать созданию алгоритмов и основываться на требованиях к функциональности и быстродействию программы. Наиболее часто в программах используются массив, список, стек, очередь, бинарное дерево, хеш-таблица, граф и множество. Далее дана краткая характеристика каждой из этих структур данных.

*Массив* — это конечная совокупность однотипных величин. Массив занимает непрерывную область памяти и предоставляет прямой (произвольный) доступ к своим элементам по индексу. Память под массив выделяется до начала работы с ним и впоследствии не изменяется.

В *списке* каждый элемент связан со следующим и, возможно, с предыдущим. В первом случае список называется *односвязным*, во втором — *двусвязным*. Если последний элемент связать указателем с первым, получится *кольцевой список*. Количество элементов в списке может изменяться в процессе работы программы.

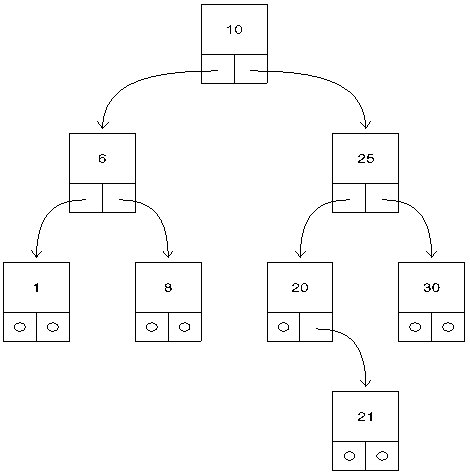
Каждый элемент списка содержит *ключ*, идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью данных, хранящихся в каждом элементе списка. В качестве ключа в процессе работы со списком могут выступать разные части данных. Например, если создается список из записей, содержащих фамилию, год рождения и стаж работы, любая часть записи может выступать в качестве ключа: при упорядочивании списка по алфавиту ключом будет фамилия, а при поиске, например, ветеранов труда ключом можно сделать стаж. Ключи разных элементов списка могут совпадать.

Над списками можно выполнять операции добавления, удаления и вставки элемента, чтения элемента с заданным ключом, упорядочивания списка по ключу (ключам). Список не обеспечивает произвольный доступ к элементу, поэтому при выполнении операций чтения, вставки и удаления выполняется последовательный перебор элементов, пока не будет найден элемент с заданным ключом.

*Стек* — частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняются с одного конца, называемого вершиной стека. Другие операции со стеком не определены. При выборке элемент исключается из стека. Говорят, что стек реализует принцип обслуживания LIFO (Last In — First Out, последним пришел — первым ушел).

*Очередь* — частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка — из другого конца. Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания FIFO (First In — First Out, первым пришел — первым ушел).

*Бинарное дерево* — динамическая структура данных, состоящая из узлов, каждый из которых содержит, помимо данных, не более двух ссылок на различные бинарные поддеревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется *корнем* дерева. Пример бинарного дерева приведен на [рис. 9.1](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=5#image.9.1) (корень обычно изображается сверху). Узел, не имеющий поддеревьев, называется *листом*. Исходящие узлы называются *предками*, входящие — *потомками. Высота дерева* определяется количеством уровней, на которых располагаются его узлы.



**Рис 1.**Пример бинарного дерева поиска

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, оно называется *деревом поиска*. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку время поиска определяется *высотой дерева*, а она пропорциональна двоичному логарифму количества узлов.

#### Хеш-таблица, ассоциативный массив, или словарь

— это массив, доступ к элементам которого осуществляется не по номеру, а по некоторому ключу. Можно сказать, что это таблица, состоящая из пар "ключ-значение" ([табл. 1](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=5#table.9.1)). Хеш-таблица эффективно реализует операцию поиска значения по ключу. При этом ключ преобразуется в число ( *хэш-код* ), которое используется для быстрого нахождения нужного значения в хеш-таблице.

|  |  |
| --- | --- |
| Таблица 1. Пример хэш-таблицы | |
| **Ключ** | **Значение** |
| boy | мальчик |
| girl | девочка |
| dog | собачка |

Преобразование выполняется с помощью *хэш-функции*, или *функции расстановки*. Эта функция обычно производит какие-либо преобразования внутреннего представления ключа. Если хеш-функция распределяет совокупность возможных ключей равномерно по множеству индексов массива, то доступ к элементу по ключу выполняется почти так же быстро, как в массиве.

Смысл хэш-функции состоит в том, чтоб отобразить более широкое множество ключей в более узкое множество индексов. При этом неизбежно возникают так называемые *коллизии*, когда хеш-функция формирует для двух разных элементов один и тот же хэш-код. В разных реализациях хэш-таблиц используются различные стратегии борьбы с коллизиями.

*Граф* — это совокупность узлов и ребер, соединяющих различные узлы. Например, можно представить себе карту автомобильных дорог как граф с городами в качестве узлов и шоссе между городами в качестве ребер. Множество реальных практических задач можно описать в терминах графов, что делает их структурой данных, часто используемой при написании программ.

*Множество* — это неупорядоченная совокупность элементов. Для множеств определены операции проверки *принадлежности элемента множеству*, включения и исключения элемента, а также объединения, пересечения и вычитания множеств.

Описанные структуры данных называются *абстрактными,* поскольку в них не задается реализация допустимых операций.

В библиотеках большинства современных объектно-ориентированных языков программирования представлены стандартные классы, реализующие основные абстрактные структуры данных. Такие классы называются *коллекциями*, или *контейнерами*. Для каждого *типа коллекции* определены методы работы с ее элементами, не зависящие от конкретного типа хранимых данных, поэтому один и тот же вид коллекции можно использовать для хранения данных различных типов. Использование коллекций позволяет сократить сроки разработки программ и повысить их надежность. Изучение возможностей стандартных коллекций и их грамотное применение является необходимым условием создания эффективных и профессиональных программ.

**Внимание**

*Каждый вид коллекции поддерживает свой набор операций над данными, и быстродействие этих операций может быть разным. Выбор вида коллекции зависит от того, что требуется делать с данными в программе и какие требования предъявляются к ее быстродействию. Например, при необходимости часто вставлять и удалять элементы из середины последовательности следует использовать список, а если включение элементов выполняется в конец последовательности — очередь*.

В библиотеке .NET определено множество стандартных классов, реализующих большинство перечисленных ранее абстрактных структур данных. Основные пространства имен, в которых описаны эти классы — System.Collections, System.Collections.Specialized и System.Collections.Generic .

#### Пространство имен System.Collections

В пространстве имен System.Collections определены наборы стандартных коллекций и интерфейсов, которые реализованы в этих коллекциях. В [таблице 2](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=5#table.9.2) приведены наиболее важные интерфейсы, часть из которых уже изучались в разделе "Стандартные интерфейсы .NET".

|  |  |
| --- | --- |
| Таблица 2. Интерфейсы пространства имен System.Collections | |
| Интерфейс | Назначение |
| ICollection | Определяет общие характеристики (например, размер) для набора элементов |
| IComparer | Позволяет сравнивать два объекта |
| IDictionary | Позволяет представлять содержимое объекта в виде пар "имя-значение" |
| IDictionaryEnumerator | Используется для нумерации содержимого объекта, поддерживающего интерфейс IDictionary |
| IEnumerable | Возвращает интерфейс IEnumerator для указанного объекта |
| IEnumerator | Обычно используется для поддержки оператора foreach в отношении объектов |
| IHashCodeProvider | Возвращает хэш-код для реализации типа с применением выбранного пользователем алгоритма хэширования |
| Ilist | Поддерживает методы добавления, удаления и индексирования элементов в списке объектов |

В [таблице 3](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=5#table.9.3) перечислены основные коллекции, определенные в пространстве System.Collections.

|  |  |  |
| --- | --- | --- |
| Таблица 3. Коллекции пространства имен System.Collections | | |
| Класс | Назначение | Важнейшие из реализованных интерфейсов |
| ArrayList | Массив, динамически изменяющий свой размер | *IList*, *ICollection*, IEnumerable, *ICloneable* |
| BitArray | Компактный массив для хранения битовых значений | *ICollection*, IEnumerable, *ICloneable* |
| Hashtable | Хэш-таблица | IDictionary, *ICollection*, IEnumerable, *ICloneable* |
| Queue | Очередь | *ICollection*, *ICloneable*, IEnumerable |
| SortedList | Коллекция, отсортированная по ключам. Доступ к элементам — по ключу или по индексу | IDictionary, *ICollection*, IEnumerable, *ICloneable* |
| Stack | Стек | *ICollection*, IEnumerable |

Пространство имен System.Collections.Specialized включает специализированные коллекции, например, коллекцию строк StringCollection и хэш-таблицу со строковыми ключами StringDictionary.

В качестве примера стандартной коллекции рассмотрим класс ArrayList.

#### Класс ArrayList

Основным недостатком обычных массивов является то, что объем памяти, необходимый для хранения их элементов, должен быть выделен до начала работы с массивом. Класс ArrayList позволяет программисту не заботиться о выделении памяти и хранить в одном и том же массиве элементы различных типов.

По умолчанию при создании объекта типа ArrayList строится массив из 16 элементов типа object. Можно задать желаемое количество элементов в массиве, передав его в конструктор или установив в качестве значения свойства Capacity, например:

ArrayList arr1 = new ArrayList(); // создается массив из 16 элементов

ArrayList arr2 = new ArrayList(1000); // создается массив из 1000 элементов

ArrayList arr3 = new ArrayList();

arr3.Capacity = 1000; // количество элементов задается

Класс ArrayList реализован через класс Array, то есть содержит закрытое поле этого класса. Поскольку все типы в C# являются потомками класса object, массив может содержать элементы произвольного типа. Даже если в массиве хранятся обычные целые числа, то есть элементы *значимого типа*, внутренний класс является массивом ссылок на экземпляры типа object, которые представляют собой упакованный тип-значение. Соответственно, при занесении в массив выполняется упаковка, а при извлечении — распаковка элемента.

Если при добавлении элемента в массив оказывается, что фактическое количество элементов массива превышает его емкость, она автоматически удваивается, то есть происходит повторное выделение памяти и переписывание туда всех существующих элементов. Пример занесения элементов в экземпляр класса ArrayList:

arr1.Add( 123 ); arr1.Add( -2 ); arr1.Add( "Вася" );

Доступ к элементу выполняется по индексу, однако при этом необходимо явным образом привести полученную ссылку к целевому типу, например:

int a = (int) arr1[0];

int b = (int) arr1[1];

string s = (string) arr1[2];

Попытка приведения к типу, не соответствующему хранимому в элементе, вызывает генерацию исключения InvalidCastException.

Для повышения надежности программ применяется следующий прием: экземпляр класса ArrayList объявляется закрытым полем класса, в котором необходимо хранить коллекцию значений определенного типа, а затем описываются методы работы с этой коллекцией, делегирующие свои функции методам ArrayList.

Недостатком этого решения является то, что для каждого метода стандартной коллекции приходится описывать метод-оболочку, вызывающий стандартный метод. Хотя это и несложно, но несколько неизящно. В C#, начиная с версии 2.0, появились классы-прототипы (generics), позволяющие решить эту проблему.

### Классы-прототипы (generics)

*Классы-прототипы* (generics) — это классы, имеющие в качестве параметров типы данных. Чаще всего их применяют для хранения данных, то есть в качестве *контейнерных классов*, или коллекций. Во вторую версию библиотеки .NET добавлены *параметризованные коллекции* для представления основных структур данных, применяющихся при создании программ — стека, очереди, списка, словаря и т. д. Эти коллекции, расположенные в пространстве имен System.Collections.Generic, дублируют аналогичные коллекции пространства имен System.Collections. В [таблице 4](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=6#table.9.4) приводится соответствие между обычными и параметризованными коллекциями библиотеки .NET (параметры, определяющие типы данных, хранимых в коллекции, указаны в угловых скобках).

|  |  |
| --- | --- |
| Таблица 9.4. Параметризованные коллекции библиотеки .NET версии 2.0 | |
| **Класс-прототип (версия 2.0)** | **Обычный класс** |
| Comparer<T> | Comparer |
| Dictionary<K,T> | HashTable |
| LinkedList<T> | — |
| List<T> | ArrayList |
| Queue<T> | Queue |
| SortedDictionary<K,T> | SortedList |
| Stack<T> | Stack<T> |

В качестве примера рассмотрим применение универсального "двойника" класса ArrayList — класса List<T> — для хранения коллекции объектов известных нам классов Monster и Daemon, а также для хранения целых чисел.

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

using MonsterLib; // библиотека, в которой хранятся классы Monster и Daemon

class Program

{

static void Main()

{

List<Monster> stado = new List<Monster>();

stado.Add( new Monster( "Monia" ) );

stado.Add( new Monster( "Monk" ) );

stado.Add( new Daemon ( "Dimon", 3 ) );

foreach ( Monster x in stado ) x.Passport();

List<int> lint = new List<int>();

lint.Add( 5 ); lint.Add( 1 ); lint.Add( 3 );

lint.Sort();

int a = lint[2];

Console.WriteLine( a );

foreach ( int x in lint ) Console.Write( x + " ");

}}}

Листинг 7. Использование универсальной коллекции List<T>

Результат работы программы:

Monster Monia health = 100 ammo = 100

Monster Monk health = 100 ammo = 100

Daemon Dimon health = 100 ammo = 100 brain = 3

5

1 3 5

В [листинге 7](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=6#example.9.7) две коллекции. Первая (stado) содержит элементы пользовательских классов, которые находятся в библиотеке MonsterLib.dll. В коллекции, для которой объявлен тип элементов Monster, благодаря полиморфизму можно хранить элементы любого производного класса, но не элементы других типов. Достоинством такого ограничения является то, что компилятор может выполнить контроль типов, что повышает надежность программы и упрощает поиск ошибок.

Коллекция lint состоит из целых чисел, причем для работы с ними не требуются ни операции упаковки и распаковки, ни явные преобразования типа при получении элемента из коллекции.

Классы-прототипы называют также родовыми или шаблонными, поскольку они представляют собой образцы, по которым во время выполнения программы строятся конкретные классы.

## Задания на лабораторную работу.

## Контрольные вопросы.

1. Изучите разделы стандарта C#, касающиеся интерфейсов.
2. Изучите по справочной системе основные стандартные интерфейсы .NET.
3. Чем интерфейс отличается от абстрактного класса?
4. Должен ли класс реализовывать все методы всех своих интерфейсов-предков?
5. Какие операции используются для проверки, реализует ли класс заданный интерфейс?
6. Перечислите стандартные интерфейсы .NET, которые определяют возможности сортировки, клонирования и просмотра объектов с помощью оператора foreach.
7. Опишите, как используется конструкция *yield*.
8. Изучите по справочной системе состав пространства имен System.Collections.
9. Изучите по справочной системе свойства и методы класса ArrayList.
10. Изучите разделы стандарта C#, касающиеся классов-прототипов.
11. Изучите по справочной системе состав пространства имен System.Collections.Generic. Опишите основные виды абстрактных структур данных.
12. Допускает ли структура "линейный список" прямой доступ к своим элементам?
13. В чем основная задача хеш-функции?
14. В каких задачах используется стек? Приведите примеры.
15. Сравните время поиска элемента в линейном списке и в дереве поиска.
16. В чем преимущество массива перед другими структурами данных?
17. Опишите состав пространства имен System.Collections и дайте характеристику основных типов-коллекций.
18. Какие новые контейнеры появились в библиотеке классов для версии С# 2.0?
19. В чем преимущества контейнеров версии С# 2.0?
20. Какие методы содержит класс ArrayList?

# Делегаты и события

## Теоретические сведения.

### Делегаты

*Делегат*— это вид класса, предназначенный для хранения ссылок на методы. Делегат, как любой *класс*, можно передать в качестве параметра, а затем вызвать инкапсулированный в нем метод. *Делегаты* используются для поддержки событий, а также как самостоятельная конструкция языка (в том числе в многопоточных приложениях).

#### Описание делегатов

Описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

[ атрибуты ] [ спецификаторы ] delegate тип имя\_делегата ( [ параметры ] )

Допускаются спецификаторы new, public, protected, internal и private. *Тип* описывает возвращаемое значение методов, вызываемых с помощью делегата, а необязательными *параметрами* делегата являются параметры этих методов. Делегат может хранить ссылки на несколько методов и вызывать их поочередно; естественно, что сигнатуры всех методов должны совпадать.

Пример описания делегата:

public delegate void D ( int i );

Здесь описан тип делегата, который может хранить ссылки на методы, возвращающие void и принимающие один параметр целого типа. Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса.

#### Использование делегатов

Чтобы воспользоваться делегатом, необходимо создать его экземпляр и задать имена методов, на которые он будет ссылаться. При вызове *экземпляра делегата* вызываются все заданные в нем методы.

Делегаты применяются в основном для следующих целей:

* получения возможности определять вызываемый метод не при компиляции, а во время выполнения программы;
* обеспечения связи между объектами по типу "источник — наблюдатель";
* создания универсальных методов, в которые можно передавать другие методы;
* поддержки механизма обратных вызовов.

Рассмотрим сначала пример реализации первой из этих целей. В [листинге  1](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=1#example.10.1) объявляется делегат, с помощью которого один и тот же оператор используется для вызова двух разных методов ( C00l и *Hack* ).

using System;

namespace ConsoleApplication1

{

delegate void Del ( ref string s ); // объявление делегата

class Class1

{

public static void C00l ( ref string s ) // метод 1

{

string temp = "";

for ( int i = 0; i < s.Length; ++i )

{

if ( s[i] == 'o' || s[i] == 'O') temp += '0';

else if ( s[i] == 'l' ) temp += '1';

else temp += s[i];

}

s = temp;

}

public static void Hack ( ref string s ) // метод 2

{

string temp = "";

for ( int i = 0; i < s.Length; ++i )

if ( i / 2 \* 2 == i ) temp += char.ToUpper( s[i] );

else temp += s[i];

s = temp;

}

static void Main()

{

string s = "cool hackers";

Del d; // экземпляр делегата

for ( int i = 0; i < 2; ++i )

{

d = new Del( C00l ); // инициализация методом 1

if ( i == 1 ) d = new Del(Hack); // инициализация методом 2

d( ref s ); // использование делегата для вызова методов

Console.WriteLine( s );

}

}

}

}

Листинг 1. Использование делегата

Результат работы программы:

c001 hackers

C001 hAcKeRs

Использование делегата имеет тот же синтаксис, что и вызов метода. Если делегат хранит *ссылки на несколько методов*, они вызываются последовательно в том порядке, в котором были добавлены в делегат.

Добавление метода в список выполняется либо с помощью метода Combine, унаследованного от класса System.Delegate, либо, что удобнее, с помощью перегруженной операции сложения. Вот как выглядит измененный метод Main из предыдущего листинга, в котором одним вызовом делегата выполняется преобразование исходной строки сразу двумя методами.

static void Main()

{

string s = "cool hackers";

Del d = new Del( C00l );

d += new Del( Hack ); // добавление метода в делегат

d( ref s );

Console.WriteLine( s ); // результат: C001 hAcKeRs

}

При вызове последовательности методов с помощью делегата необходимо учитывать следующее:

* сигнатура методов должна в точности соответствовать делегату;
* методы могут быть как статическими, так и обычными методами класса;
* каждому методу в списке передается один и тот же набор параметров;
* если параметр передается по ссылке, изменения параметра в одном методе отразятся на его значении при вызове следующего метода;
* если параметр передается с ключевым словом out или метод возвращает значение, результатом выполнения делегата является значение, сформированное последним из методов списка (в связи с этим рекомендуется формировать списки только из делегатов, имеющих возвращаемое значение типа void );
* если в процессе работы метода возникло исключение, не обработанное в том же методе, последующие методы в списке не выполняются, а происходит поиск обработчиков в объемлющих делегат блоках;
* попытка вызвать делегат, в списке которого нет ни одного метода, вызывает генерацию исключения System.NullReferenceException.

#### Паттерн "наблюдатель"

Рассмотрим применение делегатов для обеспечения связи между объектами по типу "источник — наблюдатель". В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных объектов. При этом желательно избежать жесткой связанности классов, так как это часто негативно сказывается на возможности многократного использования кода.

Для обеспечения связи между объектами во время выполнения программы применяется следующая стратегия. Объект, называемый *источником*, при изменении своего состояния, которое может представлять интерес для других объектов, посылает им уведомления. Эти объекты называются *наблюдателями*. Получив уведомление, наблюдатель опрашивает источник, чтобы синхронизировать с ним свое состояние. Примером такой стратегии может служить связь электронной таблицы с созданными на ее основе диаграммами.

Программисты часто используют одну и ту же схему организации и взаимодействия объектов в разных контекстах. За такими схемами закрепилось название *паттерны*, или *шаблоны* *проектирования*. Описанная стратегия известна под названием *паттерн "наблюдатель"*.

Наблюдатель (observer) определяет между объектами зависимость типа "один ко многим", так что при изменении состояния одного объекта все зависящие от него объекты получают извещение и автоматически обновляются. Рассмотрим пример ([листинг 2](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=1#example.10.2)), в котором демонстрируется схема оповещения источником трех наблюдателей. Гипотетическое изменение состояния объекта моделируется сообщением "ОЙ!". Один из методов в демонстрационных целях сделан статическим.

using System;

namespace ConsoleApplication1

{

public delegate void Del( object o ); // объявление делегата

class Subj // класс-источник

{

Del dels; // объявление экземпляра делегата

public void Register( Del d ) // регистрация делегата

{

dels += d;

}

public void OOPS() // что-то произошло

{

Console.WriteLine( "ОЙ!" );

if ( dels != null ) dels( this ); // оповещение наблюдателей

}

}

class ObsA // класс-наблюдатель

{

public void Do( object o ) // реакция на событие источника

{

Console.WriteLine( "Бедняжка!" );

}

}

class ObsB // класс-наблюдатель

{

public static void See( object o ) // реакция на событие источника

{

Console.WriteLine( "Да ну, ерунда!" );

}

}

class Class1

{

static void Main()

{

Subj s = new Subj(); // объект класса-источника

ObsA o1 = new ObsA(); // объекты

ObsA o2 = new ObsA(); // класса-наблюдателя

s.Register( new Del( o1.Do ) ); // регистрация методов

s.Register( new Del( o2.Do ) ); // наблюдателей в источнике

s.Register( new Del( ObsB.See ) ); // ( экземпляры делегата )

s.OOPS(); // инициирование события

}

}

}

Листинг 2. Оповещение наблюдателей с помощью делегата

В источнике объявляется *экземпляр делегата*, в этот экземпляр заносятся методы тех объектов, которые хотят получать уведомление об изменении состояния источника. Этот процесс называется *регистрацией делегатов*. При регистрации имя метода добавляется к списку. При наступлении "часа Х" все зарегистрированные методы поочередно вызываются через делегат.

Результат работы программы:

ОЙ!

Бедняжка!

Бедняжка!

Да ну, ерунда!

Для обеспечения обратной связи между наблюдателем и источником делегат объявлен с параметром типа object, через который в вызываемый метод передается ссылка на вызывающий объект. Следовательно, в вызываемом методе можно получать информацию о состоянии вызывающего объекта и посылать ему сообщения.

Связь "источник — наблюдатель" устанавливается во время выполнения программы для каждого объекта по отдельности. Если наблюдатель больше не хочет получать уведомления от источника, можно удалить соответствующий метод из списка делегата с помощью метода Remove или перегруженной операции вычитания, например:

public void UnRegister( Del d ) // удаление делегата

{

dels -= d;

}

#### Операции

Делегаты можно *сравнивать на равенство и неравенство*. Два делегата равны, если они оба не содержат ссылок на методы или если они содержат ссылки на одни и те же методы в одном и том же порядке. Сравнивать можно даже делегаты различных типов при условии, что они имеют один и тот же тип возвращаемого значения и одинаковые списки параметров.

С делегатами одного типа можно *выполнять операции* простого и сложного *присваивания*, например:

Del d1 = new Del( o1.Do ); // o1.Do

Del d2 = new Del( o2.Do ); // o2.Do

Del d3 = d1 + d2; // o1.Do и o2.Do

d3 += d1; // o1.Do, o2.Do и o1.Do

d3 -= d2; // o1.Do и o1.Do

Эти операции могут понадобиться, например, в том случае, если в разных обстоятельствах требуется вызывать разные наборы и комбинации наборов методов.

Делегат, как и строка string, является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.

#### Передача делегатов в методы

Поскольку делегат является классом, его можно передавать в методы в качестве параметра. Таким образом обеспечивается *функциональная параметризация*: в метод можно передавать не только различные данные, но и различные функции их обработки. Функциональная параметризация применяется для создания универсальных методов и обеспечения возможности обратного вызова.

В качестве простейшего примера *универсального метода* можно привести метод вывода таблицы значений функции, в который передается диапазон значений аргумента, шаг его изменения и вид вычисляемой функции. Пример приведен в [листинге 3](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=2#example.10.3).

using System;

namespace ConsoleApplication1

{

public delegate double Fun( double x ); // объявление делегата

class Class1

{

public static void Table( Fun F, double x, double b )

{

Console.WriteLine( " ----- X ----- Y -----" );

while (x <= b)

{

Console.WriteLine( "| {0,8:0.000} | {1,8:0.000} |", x, F(x));

x += 1;

}

Console.WriteLine( " ---------------------" );

}

public static double Simple( double x )

{

return 1;

}

static void Main()

{

Console.WriteLine( " Таблица функции Sin " );

Table( new Fun( Math.Sin ), -2, 2 );

Console.WriteLine( " Таблица функции Simple " );

Table( new Fun( Simple ), 0, 3 );

}

}

}

Листинг 3. Передача делегата через список параметров

Результат работы программы:

Таблица функции Sin

----- X ----- Y -----

| -2,000 | -0,909 |

| -1,000 | -0,841 |

| 0,000 | 0,000 |

| 1,000 | 0,841 |

| 2,000 | 0,909 |

---------------------

Таблица функции Simple

----- X ----- Y -----

| 0,000 | 1,000 |

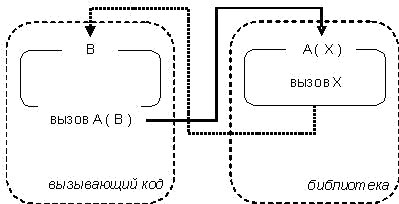
| 1,000 | 1,000 |

| 2,000 | 1,000 |

| 3,000 | 1,000 |

---------------------

*Обратный вызов*(callback) представляет собой вызов функции, передаваемой в другую функцию в качестве параметра. Рассмотрим [рисунок 1](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=2#image.10.1). Допустим, в библиотеке описана функция А, параметром которой является имя другой функции. В вызывающем коде описывается функция с требуемой сигнатурой ( В ) и передается в функцию А. Выполнение функции А приводит к вызову В, то есть управление передается из библиотечной функции обратно в вызывающий код.



**Рис. 1.**Механизм обратного вызова

Механизм обратного вызова широко используется в программировании. Например, он реализуется во многих стандартных функциях Windows.

Начиная с Visual Studio 2005, использующей версию 2.0 языка C#, можно применять упрощенный синтаксис для делегатов. Первое упрощение заключается в том, что в большинстве случаев явным образом создавать *экземпляр делегата* не требуется, поскольку он создается автоматически по контексту. Второе упрощение заключается в возможности создания так называемых *анонимных методов* — фрагментов кода, описываемых непосредственно в том месте, где используется делегат:

static void Main()

{

Console.WriteLine( " Таблица функции Sin " );

Table( Math.Sin, -2, 2 ); // упрощение 1

Console.WriteLine( " Таблица функции Simple " );

Table( delegate (double x ){ return 1; }, 0, 3 ); // упрощение 2

}

}

}

В первом случае *экземпляр делегата*, соответствующего функции Sin, создается автоматически. Чтобы это могло произойти, список параметров и тип возвращаемого значения функции должны быть совместимы с делегатом. Во втором случае не требуется оформлять простой фрагмент кода в виде отдельной функции Simple, как это было сделано в предыдущем листинге, — код функции оформляется как анонимный метод и встраивается прямо в место передачи.

### События

*Событие* — это элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния. При этом для объектов, являющихся наблюдателями события, активизируются методы-обработчики этого события. Обработчики должны быть зарегистрированы в объекте-источнике события. Таким образом, механизм событий формализует на языковом уровне *паттерн*"наблюдатель", который рассматривался в предыдущем разделе.

Механизм событий можно также описать с помощью модели "публикация — подписка": один *класс*, являющийся *отправителем* (*sender*) сообщения, публикует события, которые он может инициировать, а другие классы, являющиеся *получателями* (receivers) сообщения, подписываются на получение этих событий.

События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому *создание события* в классе состоит из следующих частей:

* описание делегата, задающего сигнатуру обработчиков событий;
* описание события;
* описание метода (методов), инициирующих событие.

*Синтаксис* события похож на *синтаксис* делегата:

[ атрибуты ] [ спецификаторы ] event тип имя\_события

Для событий применяются *спецификаторы* new, public, protected, internal, private, static, virtual, sealed, override, abstract и extern. Например, так же как и методы, событие может быть статическим ( static ), тогда оно связано с классом в целом, или обычным — в этом случае оно связано с экземпляром класса. *Тип* события — это тип делегата, на котором основано событие.

Пример описания делегата и соответствующего ему события:

public delegate void Del( object o ); // объявление делегата

class A

{

public event Del Oops; // объявление события

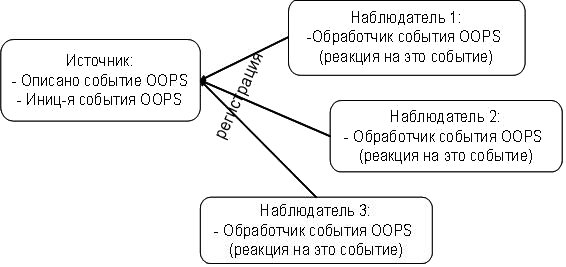
...

}

*Обработка событий* выполняется в классах-получателях сообщения. Для этого в них описываются методы-обработчики событий, *сигнатура* которых соответствует типу делегата. Каждый *объект* (не *класс*!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод.

Как видите, это в точности тот же самый механизм, который рассматривался в предыдущем разделе. Единственное отличие состоит в том, что при использовании событий не требуется описывать метод, регистрирующий обработчики, поскольку события поддерживают *операции* += и -=, добавляющие обработчик в *список* и удаляющие его из списка.

В [листинге.4](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=3#example.10.4) приведен код из [листинга 2](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=1#example.10.2), переработанный с использованием событий, а [рисунок 2](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=3#image.10.2) поясняет организацию обработки событий.



**Рис. 2.**Выполнение программы с двумя нулевыми коэффициентами

using System;

namespace ConsoleApplication1

{

public delegate void Del(); // объявление делегата

class Subj // класс-источник

{

public event Del Oops; // объявление события

public void CryOops() // метод, инициирующий событие

{

Console.WriteLine( "ОЙ!" );

if ( Oops != null ) Oops();

}

}

class ObsA // класс-наблюдатель

{

public void Do(); // реакция на событие источника

{

Console.WriteLine( "Бедняжка!" );

}

}

class ObsB // класс-наблюдатель

{

public static void See() // реакция на событие источника

{

Console.WriteLine( "Да ну, ерунда!" );

}

}

class Class1

{

static void Main()

{

Subj s = new Subj(); // объект класса-источника

ObsA o1 = new ObsA(); // объекты

ObsA o2 = new ObsA(); // класса-наблюдателя

s.Oops += new Del( o1.Do ); // добавление

s.Oops += new Del( o2.Do ); // обработчиков

s.Oops += new Del( ObsB.See ); // к событию

s.CryOops(); // инициирование события

}

}

}

Листинг 4. Оповещение наблюдателей с помощью событий

Внешний код может работать с событиями единственным образом: добавлять обработчики в *список* или удалять их, поскольку вне класса могут использоваться только *операции* += и -=. Тип результата этих операций — void, в отличие от операций сложного присваивания для арифметических типов.

Внутри класса, в котором описано событие, с ним можно обращаться, как с обычным полем, имеющим тип делегата: использовать *операции* отношения, присваивания и т. д. *Значение* события по умолчанию — null.

В библиотеке .*NET* описано огромное количество стандартных делегатов, предназначенных для реализации механизма обработки событий. Большинство этих классов оформлено по одним и тем же правилам:

* имя делегата заканчивается суффиксом EventHandler ;
* делегат получает два параметра:
  + первый параметр задает источник события и имеет тип object ;
  + второй параметр задает аргументы события и имеет тип EventArgs или производный от него.

Если обработчикам события требуется специфическая *информация* о событии, то для этого создают *класс*, производный от стандартного класса EventArgs, и добавляют в него необходимую информацию. Если делегат не использует такую информацию, можно обойтись стандартным классом делегата System.EventHandler.

Имя обработчика события принято составлять из префикса On и имени события. В [листинге 5](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=3#example.10.5) приведен пример из [листинга 4](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=3#example.10.4), оформленный в соответствии со стандартными соглашениями .*NET*.

using System;

namespace ConsoleApplication1

{

class Subj

{

public event EventHandler Oops;

public void CryOops()

{

Console.WriteLine( "ОЙ!" );

if ( Oops != null ) Oops( this, null );

}

}

class ObsA

{

public void OnOops( object sender, EventArgs e )

{

Console.WriteLine( "Бедняжка!" );

}

}

class ObsB

{

public static void OnOops( object sender, EventArgs e )

{

Console.WriteLine( "Да ну, ерунда!" );

}

}

class Class1

{ static void Main()

{

Subj s = new Subj();

ObsA o1 = new ObsA();

ObsA o2 = new ObsA();

s.Oops += new EventHandler( o1.OnOops );

s.Oops += new EventHandler( o2.OnOops );

s.Oops += new EventHandler( ObsB.OnOops );

s.CryOops();

}

}

}

Листинг 5. Использование стандартного делегата EventHandler

### Многопоточные приложения

*Приложение* .*NET* состоит из одного или нескольких *процессов*. Процессу принадлежат выделенная для него область оперативной памяти и ресурсы. Каждый процесс может состоять из нескольких *доменов* (частей) приложения, ресурсы которых изолированы друг от друга. В рамках домена может быть запущено несколько потоков выполнения. *Поток* (*thread*) представляет собой часть исполняемого кода программы. Иногда термин "*thread*" переводится буквально — "нить", чтобы отличить его от потоков ввода-вывода. Поэтому в литературе можно встретить и термин "многонитевые приложения".

В каждом процессе есть *первичный поток*, исполняющий роль точки входа в *приложение*. Для консольных приложений это метод Main.

Многопоточные приложения создают как для многопроцессорных, так и для однопроцессорных систем. Основной целью при этом являются повышение общей производительности и сокращение времени реакции приложения. Управление потоками осуществляет *операционная система*. Каждый *поток* получает некоторое количество квантов времени, по истечении которого управление передается другому потоку. Это создает у пользователя однопроцессорной машины впечатление одновременной работы нескольких потоков и позволяет, к примеру, выполнять ввод текста одновременно с длительной операцией по передаче данных.

Недостатки многопоточности:

большое количество потоков ведет к увеличению накладных расходов, связанных с переключением потоков, что снижает общую производительность;

возникают проблемы синхронизации данных, связанные с потенциальной возможностью доступа к одним и тем же данным со стороны нескольких потоков (например, если один поток начинает изменение общих данных, а отведенное ему время истекает, доступ к этим же данным может получить другой поток, который, изменяя данные, необратимо их повреждает).

#### Класс Thread

В .NET многопоточность поддерживается в основном с помощью пространства имен System.Threading. Некоторые типы этого пространства описаны в [таблице 1](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=4#table.10.1).

|  |  |
| --- | --- |
| Таблица 10.1. Некоторые типы пространства имен System.Threading | |
| **Тип** | **Описание** |
| Monitor | Класс, обеспечивающий синхронизацию доступа к объектам |
| *Mutex* | Класс-примитив синхронизации, который используется также для синхронизации между процессами |
| Thread | Класс, который создает поток, устанавливает его приоритет, получает информацию о состоянии |
| ThreadPool | Класс, используемый для управления набором взаимосвязанных потоков — пулом потоков |
| Timer | Класс, определяющий механизм вызова заданного метода в заданные интервалы времени для пула потоков |
| WaitHandle | Класс, инкапсулирующий объекты синхронизации, которые ожидают доступа к разделяемым ресурсам |
| ThreadStart | Делегат, представляющий метод, который должен быть выполнен при запуске потока |
| TimerCallback | Делегат, представляющий метод, обрабатывающий вызовы от класса Timer |
| WaitCallback | Делегат, представляющий метод для элементов класса ThreadPool |
| ThreadPriority | Перечисление, описывающее *приоритет потока* |
| ThreadState | Перечисление, описывающее состояние потока |

Первичный поток создается автоматически. Для запуска вторичных потоков используется класс Thread. При создании объекта-потока ему передается делегат, определяющий метод, выполнение которого выделяется в отдельный поток:

Thread t = new Thread ( new ThreadStart( имя\_метода ) );

После создания потока заданный метод начинает в нем свою работу, а первичный поток продолжает выполняться. В [листинге 6](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=4#example.10.6)приведен пример одновременной работы двух потоков.

using System;

using System.Threading;

namespace ConsoleApplication1

{ class Program

{

static public void Hedgehog() // метод для вторичного потока

{

for ( int i = 0; i < 6; ++i )

{

Console.Write(" " + i ); Thread.Sleep( 1000 );

}

}

static void Main()

{

Console.WriteLine( "Первичный поток " +

Thread.CurrentThread.GetHashCode() );

Thread ta = new Thread( new ThreadStart(Hedgehog) );

Console.WriteLine( "Вторичный поток " + ta.GetHashCode() );

ta.Start();

for ( int i = 0; i > -6; --i )

{

Console.Write( " " + i ); Thread.Sleep( 400 );

}

}

}

}

Листинг 6. Создание вторичного потока

Результат работы программы:

Первичный поток 1

Вторичный поток 2

0 0 -1 -2 1 -3 -4 2 -5 3 4 5

В листинге используется метод Sleep, останавливающий функционирование потока на заданное количество миллисекунд. Как видите, оба потока работают одновременно. Если бы они работали с одним и тем же файлом, он был бы испорчен так же, как и приведенный вывод на консоль, поэтому такой способ распараллеливания вычислений имеет смысл только для работы с различными ресурсами.

В [таблице 2](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=4#table.10.2) перечислены основные элементы класса Thread.

|  |  |  |
| --- | --- | --- |
| Таблица 10.2. Основные элементы класса Thread | | |
| Элемент | Вид | Описание |
| CurrentThread | Статическое свойство | Возвращает ссылку на выполняющийся поток (только для чтения) |
| Name | Свойство | Установка текстового имени потока |
| Priority | Свойство | Получить/установить *приоритет потока* (используются значения перечисления ThreadPriority ) |
| ThreadState | Свойство | Возвращает состояние потока (используются значения перечисления ThreadState ) |
| Abort | Метод | Генерирует исключение ThreadAbortException. Вызов этого метода обычно завершает работу потока |
| Sleep | Статический метод | Приостанавливает выполнение текущего потока на заданное количество миллисекунд |
| Interrupt | Метод | Прерывает работу текущего потока |
| Join | Метод | Блокирует вызывающий поток до завершения другого потока или указанного промежутка времени и завершает поток |
| Resume | Метод | Возобновляет работу после приостановки потока |
| Start | Метод | Начинает выполнение потока, определенного делегатом ThreadStart |
| Suspend | Метод | Приостанавливает выполнение потока. Если выполнение потока уже приостановлено, то игнорируется |

Можно создать несколько потоков, которые будут совместно использовать один и тот же код. Пример приведен в [листинге 7](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=4#example.10.7).

using System;

using System.Threading;

namespace ConsoleApplication1

{

class Class1

{ public void Do()

{

for ( int i = 0; i < 4; ++i )

{ Console.Write( " " + i ); Thread.Sleep( 3 ); }

}

}

class Program

{ static void Main()

{

Class1 a = new Class1();

Thread t1 = new Thread( new ThreadStart( a.Do ) );

t1.Name = "Second";

Console.WriteLine( "Поток " + t1.Name );

t1.Start();

Thread t2 = new Thread( new ThreadStart( a.Do ) );

t2.Name = "Third";

Console.WriteLine( "Поток " + t2.Name );

t2.Start();

}

}

}

Листинг 7. Потоки, использующие один объект

Результат работы программы:

Поток Second

Поток Third

0 0 1 1 2 2 3 3

Варианты вывода могут несколько различаться, поскольку один поток прерывает выполнение другого в неизвестные моменты времени.

Для того чтобы блок кода мог использоваться в каждый момент только одним потоком, применяется оператор lock. Формат оператора:

lock ( выражение ) блок\_операторов

*Выражение* определяет объект, который требуется заблокировать. Для обычных методов в качестве выражения используется ключевое слово this, для статических — typeof( *класс* ). *Блок операторов* задает критическую секцию кода, которую требуется заблокировать.

Например, блокировка операторов в приведенном ранее методе Do выглядит следующим образом:

public void Do()

{

lock( this )

{

for ( int i = 0; i < 4; ++i )

{ Console.Write( " " + i ); Thread.Sleep( 30 ); }

}

}

Для такого варианта метода результат работы программы изменится:

Поток Second

Поток Third

0 1 2 3 0 1 2 3

### Асинхронные делегаты

Делегат можно вызвать на выполнение либо синхронно, как во всех приведенных ранее примерах, либо асинхронно с помощью методов BeginInvoke и EndInvoke.

При вызове делегата с помощью метода BeginInvoke среда выполнения создает для исполнения метода отдельный поток и возвращает управление оператору, следующему за вызовом. При этом в исходном потоке можно продолжать вычисления.

Если при вызове BeginInvoke был указан метод обратного вызова, этот метод вызывается после завершения потока. Метод обратного вызова также задается с помощью делегата, при этом используется стандартный делегат AsyncCallback. В методе обратного вызова для получения возвращаемого значения и выходных параметров применяется метод EndInvoke.

Если метод обратного вызова не был указан в параметрах метода BeginInvoke, метод EndInvoke можно использовать в потоке, инициировавшем запрос.

В [листинге 8](https://www.intuit.ru/studies/courses/629/485/lecture/11017?page=4#example.10.8) приводится два примера асинхронного вызова метода, выполняющего разложение числа на множители. Листинг приводится по документации Visual Studio с некоторыми изменениями.

Класс Factorizer содержит метод Factorize, выполняющий разложение на множители. Этот метод асинхронно вызывается двумя способами: в методе Num1 метод обратного вызова задается в BeginInvoke, в методе Num2 имеют место ожидание завершения потока и непосредственный вызов EndInvoke.

using System;

using System.Threading;

using System.Runtime.Remoting.Messaging;

// асинхронный делегат

public delegate bool AsyncDelegate ( int Num, out int m1, out int m2 );

// класс, выполняющий разложение числа на множители

public class Factorizer

{

public bool Factorize( int Num, out int m1, out int m2 )

{

m1 = 1; m2 = Num;

for ( int i = 2; i < Num; i++ )

if ( 0 == (Num % i) ) { m1 = i; m2 = Num / i; break; }

if (1 == m1 ) return false;

else return true;

}

}

// класс, получающий делегат и результаты

public class PNum

{

private int Number;

public PNum( int number ) { Number = number; }

[OneWayAttribute()]

// метод, получающий результаты

public void Res( IAsyncResult ar )

{

int m1, m2;

// получение делегата из AsyncResult

AsyncDelegate ad = (AsyncDelegate)((AsyncResult)ar).AsyncDelegate;

// получение результатов выполнения метода Factorize

ad.EndInvoke( out m1, out m2, ar );

// вывод результатов

Console.WriteLine( "Первый способ : множители {0} : {1} {2}",

Number, m1, m2 );

}

}

// демонстрационный класс

public class Simple

{

// способ 1: используется функция обратного вызова

public void Num1()

{

Factorizer f = new Factorizer();

AsyncDelegate ad = new AsyncDelegate ( f.Factorize );

int Num = 1000589023, tmp;

// создание экземпляра класса, который будет вызван

// после завершения работы метода Factorize

PNum n = new PNum( Num );

// задание делегата метода обратного вызова

AsyncCallback callback = new AsyncCallback( n.Res );

// асинхронный вызов метода Factorize

IAsyncResult ar = ad.BeginInvoke(

Num, out tmp, out tmp, callback, null );

//

// здесь - выполнение неких дальнейших действий

// ...

}

// способ 2: используется ожидание окончания выполнения

public void Num2()

{

Factorizer f = new Factorizer();

AsyncDelegate ad = new AsyncDelegate ( f.Factorize );

int Num = 1000589023, tmp;

// создание экземпляра класса, который будет вызван

// после завершения работы метода Factorize

PNum n = new PNum( Num );

// задание делегата метода обратного вызова

AsyncCallback callback = new AsyncCallback( n.Res );

// асинхронный вызов метода Factorize

IAsyncResult ar = ad.BeginInvoke(

Num, out tmp, out tmp, null, null );

// ожидание завершения

ar.AsyncWaitHandle.WaitOne( 10000, false );

if ( ar.IsCompleted )

{

int m1, m2;

// получение результатов выполнения метода Factorize

ad.EndInvoke( out m1, out m2, ar );

// вывод результатов

Console.WriteLine( "Второй способ : множители {0} : {1} {2}",

Num, m1, m2 );

}

}

public static void Main()

{

Simple s = new Simple();

s.Num1();

s.Num2();

}

}

Листинг 8. Асинхронные делегаты

Результат работы программы:

Первый способ : множители 1000589023 : 7 142941289

Второй способ : множители 1000589023 : 7 142941289

*Атрибут* [OneWayAttribute()] *помечает метод как не имеющий возвращаемого значения и выходных параметров*.

На этом данный курс лекций завершается. В него не вошли многие темы, рассмотренные в учебнике [[4](https://www.intuit.ru/studies/courses/629/485/literature#literature.4)]: структуры, перечисления, работа с файлами, регулярные выражения, сборки, атрибуты, небезопасный код, основы программирования под Windows. Это связано с формулировкой договора, заключенного автором с издательством ПИТЕР при публикации книги.

## Задания на лабораторную работу.

## Контрольные вопросы.

1. Опишите синтаксис делегата. Как добавить метод в делегат?
2. Расскажите о способах использования делегатов.
3. Какие операции можно выполнять с делегатами?
4. Опишите реализацию событий в C#.
5. Опишите паттерн "наблюдатель".
6. Какой стандартный класс используется для передачи информации о событии?
7. Изучите разделы стандарта C#, касающиеся делегатов.
8. Изучите разделы стандарта C#, касающиеся событий.

# Регулярные выражения

## Теоретические сведения.

Стандартный *класс* *string* позволяет выполнять над строками различные *операции*, в том числе *поиск*, замену, вставку и удаление подстрок. Тем не менее, есть классы задач по обработке символьной информации, где стандартных возможностей явно не хватает. Чтобы облегчить решение подобных задач, в *Net* Framework встроен более мощный аппарат работы со строками, основанный на регулярных выражениях.

Регулярные выражения предназначены для обработки текстовой информации и обеспечивают:

1. Эффективный поиск в тексте по заданному шаблону;
2. Редактирование текста;
3. Формирование итоговых отчетов по результатам работы с текстом.

Подробно рассмотрим первые два аспекта применения регулярных выражений.

Метасимволы в регулярных выражениях

Регулярное выражение - это шаблон, по которому выполняется поиск соответствующего фрагмента текста. Язык описания регулярных выражений состоит из символов двух видов: обычных символов и метасимволов. Обычный символ представляет в выражении сам себя, а метасимвол - некоторый *класс символов*.

Рассмотрим наиболее употребительные метасимволы:

|  |  |  |
| --- | --- | --- |
| Класс символов | Описание | Пример |
| . | Любой символ, кроме \n. | Выражение c. t соответствует фрагментам: cat, cut, c#t, c{t и т.д. |
| [] | Любой одиночный символ из последовательности, записанной внутри скобок. Допускается использование *диапазонов символов*. | Выражение c[*aui*]t соответствует фрагментам: cat, cut, *cit*. Выражение c[a-c]t соответствует фрагментам: cat, *cbt*, cct. |
| [^] | Любой одиночный символ, не входящий в последовательность, записанную внутри скобок. Допускается использование *диапазонов символов*. | Выражение c[^*aui*]t соответствует фрагментам: *cbt*, cct, c2t и т.д. Выражение c[^a-c]t соответствует фрагментам: cdt, *cet*, c%t и т.д. |
| \w | Любой алфавитно - цифровой символ. | Выражение c\wt соответствует фрагментам: *cbt*, cct, c2t и т.д., но не соответствует фрагментам c%t, c{t и т.д. |
| \W | Любой не алфавитно - цифровой символ. | Выражение c\Wt соответствует фрагментам: c%t, c{t, c. t и т.д., но не соответствует фрагментам *cbt*, cct, c2t и т.д. |
| \s | Любой пробельный символ. | Выражение \s\w\w\w\s соответствует любому слову из трех букв, окруженному пробельными символами. |
| \S | Любой не пробельный символ. | Выражение \s\S\S\S\s соответствует любым трем непробельным символам, окруженным пробельными. |
| \d | Любая десятичная цифра | Выражение c\dt соответствует фрагментам: c1t, c2t, c3t и т.д. |
| \D | Любой символ, не являющийся десятичной цифрой | Выражение c\Dt не соответствует фрагментам: c1t, c2t, c3t и т.д. |

Кроме метасимволов, обозначающие классы символов, могут применяться уточняющие метасимволы:

|  |  |
| --- | --- |
| Уточняющие символы | Описание |
| ^ | Фрагмент, совпадающий с регулярными выражениями, следует искать только в начале строки |
| $ | Фрагмент, совпадающий с регулярными выражениями, следует искать только в конце строки |
| \А | Фрагмент, совпадающий с регулярными выражениями, следует искать только в начале многострочной строки |
| \Z | Фрагмент, совпадающий с регулярными выражениями, следует искать только в конце многострочной строки |
| \b | Фрагмент, совпадающий с регулярными выражениями, начинается или заканчивается на границе слова, т.е. между символами, соответствующими метасимволам \w и \W |
| \B | Фрагмент, совпадающий с регулярными выражениями, не должен встречаться на границе слов |

В регулярных выражениях часто используются повторители - метасимволы, которые располагаются непосредственно после обычного символа или группы символов и задают количество его повторений в выражении.

|  |  |  |
| --- | --- | --- |
| Повторители | Описание | Пример |
| \* | Ноль или более повторений предыдущего элемента | Выражение ca\*t соответствует фрагментам: ct, cat, caat, caaat и т.д. |
| + | Одно или более повторений предыдущего элемента | Выражение ca+t соответствует фрагментам: cat, caat, caaat и т.д. |
| ? | Не более одного повторения предыдущего элемента | Выражение ca?t соответствует фрагментам: ct, cat. |
| {n} | Ровно n повторений предыдущего элемента | Выражение ca{3}t соответствует фрагменту: cаааt. Выражение (cat){2}соответствует фрагменту: c а tcat. |
| {n,} | По крайней мере nповторений предыдущего элемента | Выражение ca{3, }t соответствует фрагментам: c ааа t, caaaat, caaaaaaat и т.д. Выражение (cat){2, } соответствует фрагментам: catcat, catcatcat и т.д. |
| {n, m} | От n до m повторений предыдущего элемента | Выражение ca{2, 4}t соответствует фрагментам: c аа t, caaat, caaaat. |

Регулярное выражение записывается в виде строкового литерала, причем перед строкой необходимо ставить символ @, который говорит о том, что строку нужно будет рассматривать и в том случае, если она будет занимать несколько строчек на экране. Однако символ @ можно не ставить, если в качестве шаблона используется шаблон без метасимволов.

Если нужно найти какой-то символ, который является метасимволом, например, точку, можно это сделать защитив ее обратным слэшем. Т.е. просто точка означает любой одиночный символ, а \. означает просто точку.

Примеры регулярных выражений:

1. слово rus -

@"rus" или "rus"

1. номер телефона в формате xxx-xx-xx - @"\d\d\d-\d\d-\d\d" или @"\d{3}(-\d\d){2}"
2. номер автомобиля - @"[A-Z]\d{3}[A-Z]{2}\d{2,3}RUS"

**Задания**. Запишите регулярное выражение, соответствующее:

1. дате в формате дд.мм.гг или дд.мм.гггг
2. времени в формате чч.мм или чч:мм
3. целому числу (со знаком и без)
4. вещественному числу (со знаком и без, с дробной частью и без, с целой частью и без)

Поиск в тексте по шаблону

Пространство имен библиотеки базовых классов System.Text.RegularExpressions содержит все объекты платформы .NET Framework, имеющие отношение к регулярным выражениям. Важнейшим классом, поддерживающим регулярные выражения, является класс Regex, который представляет неизменяемые откомпилированные регулярные выражения. Для описания регулярного выражения в классе определено несколько перегруженных конструкторов:

1. Regex() - создает пустое выражение;
2. Regex(String) - создает заданное выражение;
3. Regex(String, RegexOptions) - создает заданное выражение и задает параметры для его обработки с помощью элементов перечисления RegexOptions (например, различать или нет прописные и строчные буквы).

Поиск фрагментов строки, соответствующих заданному выражению, выполняется с помощью методов IsMatch, Match, Matchesкласса Regex.

Метод IsMatch возвращает true, если фрагмент, соответствующий выражению, в заданной строке найден, и false в противном случае. Например, попытаемся определить, встречается ли в заданном тексте слово собака:

static void Main()

{

Regex r = new Regex("собака",RegexOptions.IgnoreCase);

string text1 = "Кот в доме, собака в конуре.";

string text2 = "Котик в доме, собачка в конуре.";

Console.WriteLine(r.IsMatch(text1));

Console.WriteLine(r.IsMatch(text2));

}

*Замечание*. RegexOptions.IgnoreCase - означает, что регулярное выражение применяется без учета регистра символов.

Можно использовать конструкцию выбора из нескольких элементов. Варианты выбора перечисляются через вертикальную черту. Например, попытаемся определить, встречается ли в заданном тексте слов собака или кот:

static void Main(string[] args)

{

Regex r = new Regex("собака|кот",RegexOptions.IgnoreCase);

string text1 = "Кот в доме, собака в конуре.";

string text2 = "Котик в доме, собачка в конуре.";

Console.WriteLine(r.IsMatch(text1));

Console.WriteLine(r.IsMatch(text2));

}

Попытаемся определить, есть ли в заданных строках номера телефона в формате xx-xx-xx или xxx-xx-xx:

static void Main()

{

Regex r = new Regex(@"\d{2,3}(-\d\d){2}");

string text1 = "tel:123-45-67";

string text2 = "tel:no";

string text3 = "tel:12-34-56";

Console.WriteLine(r.IsMatch(text1));

Console.WriteLine(r.IsMatch(text2));

Console.WriteLine(r.IsMatch(text3));

}

**Задание**. Измените программу так, чтобы можно было определить, содержится в тексте дата в формате дд.мм.гг.

Метод Match класса Regex не просто определяет, содержится ли текст, соответствующий шаблону, а возвращает объект класса Match - последовательность фрагментов текста, совпавших с шаблоном. Следующий пример позволяет найти все номера телефонов в указанном фрагменте текста:

static void Main()

{

Regex r = new Regex(@"\d{2,3}(-\d\d){2}");

string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45

Контакты в Саратове tel:12-34-56; fax:12-56-45";

Match tel = r.Match(text);

while (tel.Success)

{

Console.WriteLine(tel);

tel = tel.NextMatch();

}

}

Следующий пример позволяет подсчитать сумму целых чисел, встречающихся в тексте:

static void Main()

{

Regex r = new Regex(@"[-+]?\d+");

string text = @"5\*10=50 -80/40=-2";

Match teg = r.Match(text);

int sum = 0;

while (teg.Success)

{

Console.WriteLine(teg);

sum += int.Parse(teg.ToString());

teg = teg.NextMatch();

}

Console.WriteLine("sum=" + sum);

}

**Задание**. Измените программу так, чтобы на экран дополнительно выводилось количество найденных чисел.

Метод Matches класса Regex возвращает объект класса MatchCollection - коллекцию всех фрагментов заданной строки, совпавших с шаблоном. При этом метод Matches многократно запускает метод Match, каждый раз начиная поиск с того места, на котором закончился предыдущий поиск.

static void Main(string[] args)

{

string text = @"5\*10=50 -80/40=-2";

Regex theReg = new Regex(@"[-+]?\d+");

MatchCollection theMatches = theReg.Matches(text);

foreach (Match theMatch in theMatches)

{

Console.Write("{0} ", theMatch.ToString());

}

Console.WriteLine();

}

}

### Редактирование текста

Регулярные выражения могут эффективно использоваться для редактирования текста. Например, метод Replace класса Regexпозволяет выполнять замену одного фрагмента текста другим или удаление фрагментов текста:

**Пример** *1*. Изменение номеров телефонов:

static void Main(string[] args)

{

string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45.

Контакты в Саратове tel:12-34-56; fax:11-56-45";

Console.WriteLine("Старые данные\n"+text);

string newText=Regex.Replace(text, "123-", "890-");

Console.WriteLine("Новые данные\n" + newText);

}

**Задание**. Измените программу так, чтобы шестизначные номера заменялись на семизначные добавлением 0 после первых двух цифр. Например, номер 12-34-56 заменился бы на 120-34-56.

**Пример 2**. Удаление всех номеров телефонов из текста:

static void Main()

{

string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45.

Контакты в Саратове tel:12-34-56; fax:12-56-45";

Console.WriteLine("Старые данные\n"+text);

string newText=Regex.Replace(text, @"\d{2,3}(-\d\d){2}", "");

Console.WriteLine("Новые данные\n" + newText);

}

}

**Задание**. Измените программу так, чтобы из текста удалялись слова tel и fax (если после данных слов стоят двоеточия, то их тоже следует удалить).

**Пример** 3. Разбиение исходного текста на фрагменты:

static void Main()

{

string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45.

Контакты в Саратове tel:12-34-56; fax:12-56-45";

string []newText=Regex.Split(text,"[ ,.:;]+");

foreach( string a in newText)

Console.WriteLine(a);

}

**Задание**. Разместите текст на одной строке и посмотрите, как изменится вывод данных. Объясните результаты.

## Задания на лабораторную работу.

Дана строка, в которой содержится осмысленное текстовое сообщение. Слова сообщения разделяются пробелами и знаками препинания.

1. Определите, содержится ли в сообщении заданное слово.
2. Выведите все слова заданной длины.
3. Выведите на экран все слова сообщения, записанные с заглавной буквы.
4. Удалите из сообщения все однобуквенные слова.
5. Удалите из сообщения все знаки препинания.
6. Удалите из сообщения все русские слова.
7. Удалите из сообщения только те русские слова, которые начинаются на гласную букву.
8. Заменить все английские слова на многоточие.
9. Найти максимальное целое число, встречающееся в сообщении.
10. Найти сумму всех имеющихся в тексте чисел (целых и вещественных, причем вещественное число может быть записано в экспоненциальной форме).
11. В сообщении могут встречаться номера телефонов, записанные в формате xx-xx-xx, xxx-xxx или xxx-xx-xx. Вывести все номера телефонов, которые содержатся в сообщении.
12. В сообщении может содержаться дата в формате дд.мм.гггг. В заданном формате дд - целое число из диапазона от 1 до 31, мм - целое число из диапазона от 1 до 12, а гггг - целое число из диапазона от 1900 до 2010 (если какая-то часть формата нарушена, то данная подстрока в качестве даты не рассматривается). Выведите на экран все даты, которые относятся к текущему году.
13. В сообщении могут содержаться IP-адреса компьютеров в формате d.d.d.d, где d - целое число из диапазона от 0 до 255. Вывести все IP-адреса содержащиеся в тексте.
14. В сообщении могут содержаться IP-адреса компьютеров в формате d.d.d.d, где d - целое число из диапазона от 0 до 255. Удалить из сообщения IP-адреса, в которых последнее число d начинается с заданной цифры (данная цифра вводится с клавиатуры).
15. Выведите на экран все адреса web-сайтов, содержащиеся в сообщении.
16. В сообщении может содержаться дата в формате дд.мм.гггг. В заданном формате дд - целое число из диапазона от 1 до 31, мм - целое число из диапазона от 1 до 12, а гггг - целое число из диапазона от 1900 до 2010 (если какая-то часть формата нарушена, то данная подстрока в качестве даты не рассматривается). Замените каждую дату сообщения на дату следующего дня.
17. В сообщении может содержаться дата в формате дд.мм.гггг. В заданном формате дд - целое число из диапазона от 1 до 31, мм - целое число из диапазона от 1 до 12, а гггг - целое число из диапазона от 1900 до 2010 (если какая-то часть формата нарушена, то данная подстрока в качестве даты не рассматривается). Замените каждую дату в сообщении на дату предыдущего дня.
18. В сообщении может содержаться время в формате чч:мм:сс. В заданном формате чч - целое число из диапазона от 00 до 24, мм и сс - целые числа из диапазона от 00 до 60 (если какая-то часть формата нарушена, то данная подстрока в качестве даты не рассматривается). Преобразуйте каждое время к формату чч:мм, применив правило округления до целого числа минут.
19. В сообщении может содержаться время в формате чч:мм. В заданном формате чч - целое число из диапазона от 00 до 24, мм - целое число из диапазона от 00 до 60 (если какая-то часть формата нарушена, то данная подстрока в качестве даты не рассматривается). Увеличите время на n минут.
20. В сообщении может содержаться время в формате чч:мм. В заданном формате чч - целое число из диапазона от 00 до 24, мм - целое число из диапазона от 00 до 60 (если какая-то часть формата нарушена, то данная подстрока в качестве даты не рассматривается). Уменьшите время на n часов.

## Контрольные вопросы.

## Самостоятельная работа

# Приложение Оформление лабораторных работ

Министерство образования …

Название университета …

Название кафедры…

Название дисциплины

Лабораторная работа n

Название лабораторной работы

Выполнил Проверил

Ст. гр. Хххххх ФИО Доц каф ВТ ФИО

*Тула 20хх г.*

1. Цель работы

Освоение целочичленных алгоритмов и приобретение практических навыков проектирования программ на языке C# для реализации указанных алгоритмов.

1. Теоретические сведения.

К целочисленным алгоритмам относятся:

1. Алгоритм нахождения наибольшего общего делителя двух натуральных чисел
2. Алгоритм нахождения наименьшего общего кратного двух натуральных чисел
3. Факторизация числа

… и т.д.

#### Алгоритм Евклида нахождения наибольшего общего делителя

Наиболее часто используются три разновидности алгоритма Евклида нахождения НОД(а,в).

* *Алгоритм на основе операции вычитания*
* *Алгоритм на основе операции получения остатка от деления*
* *Двоичный алгоритм*

Рассмотрим алгоритм на основе нахождения общих простых делителей для заданных чисел.

Обозначим НОД(а,в) – значение нахождения наибольшего общего делителя двух натуральных чисел а и в.

Справедливы следующие утвеждения.

1. *Всегда существует общий делитель чисел а и в равный единице.*
2. *Существует разложение чисел а и в на простые множители(основная теорема арифметики)  
   Пример . –факторизация числа а.  
   Для а=18, факторизация имеет вид 18=2×3×3*
3. *Тогда НОД(а, в) будет равен произведению общих делителей чисел а и в.  
   Пример. НОД(12,18)=НОД((2×2×3), (2×3×3))= 2×3=6*

… и т.д.

1. Задания на лабораторную работу.
   1. Задание 1

Составить алгоритм нахождения количества квадратов, на которые можно разрезать прямоугольник размером n×m в следующем порядке – выделяется максимальный квадрат, затем в оставшейся части вырезается очередной максимальных квадрат и т.д.

* + 1. Постановка задачи

Заданы два натуральных числа n×m – длины сторон прямоугольника.

Требуется найти значение N количество последовательных разрезов прямоугольника на квадраты с максимально возможной длиной стороны.

Пример. m=9, n=5.

|  |  |  |
| --- | --- | --- |
| Первый разрез. | Второй разрез. | … |
|  |  |  |

* + 1. Технические условия

1. Числа n,m <1000
2. Формат входных данных.  
   Числа n,m записаны в файле in.txt в одной строке и разделены одним пробелом.
3. Формат выходных данных  
   Результат запиывается в выходной файл out.txt в виде:  
   НОД(n,m)=число
4. Время работы приложения не превышает 1 с.
   * 1. Математическое обеспечение

#### Математические методы

Утверждается что число разрезов N=GCD(n,m).

Доказательство.

Процесс разрезания соответствует алгоритму Евклида нахождения НОД(n,m) на основе вычитания. Поэтому можно применить данный алгоритм для нахождения N.

#### Алгоритмы

Алгоритм нахождения N на основе алгоритма GCD(n,m)

1. Положить N=0
2. Положить R=|n-m|
3. Цикл Пока (R не равно нулю)
   1. N=N+1
   2. n=min{n,m}
   3. m=R
   4. R=|n-m|
   5. Конец цикла
4. Вернуть N
   * 1. Эксспериментальная часть
5. Проверить работоспособность приложения для нескольких пар чисел, удовлетворяющих техническим условиям. Резултаты тестирования см в Приложении 1.
6. Проверить работоспособность приложения для нескольких пар чисел, не удовлетворяющих техническим условиям. Резултаты тестирования см в Приложении 1.
   1. Задание n

Составить бинарный алгоритм нахождения GCD( n×m

* + 1. Постановка задачи
    2. Технические условия
    3. Математическое обеспечение

#### Математические методы

#### Алгоритмы

* + 1. Эксспериментальная часть

…

# Список литературы

* 1. Шилдт Г. ….
  2. Итернет ресурс. Сайт MetaNit.com …
  3. …

# Приложения

## Приложение 1

Задание 1. Листинг программы

…

Задание 1. Результаты тестирования

…

## Приложение 2

Задание 2. Листинг программы

…

Задание 2. Результаты тестирования

…

# Список литературы

1. **Программирование на языке высокого уровня C#/ Т.А. Павловская Т.А. Национальный Открытый Университет «ИНТУИТ», 2016**
2. **Г. Шилдт Полный справочник по С# М. : Издательский дом "Вильяме", 2004. — 752 с**
3. **Троелсен Э С# и платформа .NET. Библиотека программиста СПб.: Питер, 2004. —796 с**
4. **Павловская Т.А С#. Программирование на языке высокого уровня. Учебник для вузов СПб.: Питер, 2007. – 432 с**
5. **Джефри Рихтер Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. Мастер класс М.: Издательско-торговый дом "Русская Редакция", 2007. – 656 с**
6. [**Программирование на языке С#: разработка консольных приложений**](https://www.intuit.ru/goods_store/ebooks/8426) **/   
   Е.В. Кудрина , М.В. Огнева , М.С. Портенко Национальный Открытый Университет «ИНТУИТ», 2009**