МДК.01.01 Разработка программных модулей

Задеба Александр Анатольевич

zadaa

azadeba@kait20.ru

В основе объектно-ориентированного программирования (ООП) лежат следующие базовые понятия: *объект*, *свойство объекта*, *метод обработки*, *событие*, *класс объектов*.

https://drive.google.com/drive/folders/10_1rTDnP-wNm-f78LLjdS1zH_4_m0Agk?usp=sharing

account: kait2803@gmail.com password: _Kait20_

Определение 1. *Объект* - совокупность свойств (параметров) определенных сущностей и методов их обработки (программных средств), т.е. объект– это нечто целое, объединяющее некоторые данные, чем можно управлять с помощью кода C#.

- Например, любое приложение Windows – Word, Excel, Explorer и другие являются объектом языка, который называется Application (Приложение);
- окно программы – объект Windows;
- документ HTML – объект Document;
- диапазон ячеек в Excel – объект Range и т.д.

Объект содержит *инструкции* (программный код), определяющие действия, которые может выполнять объект, и обрабатываемые *данные.*

Один объект может выступать объединением вложенных в него по иерархии других объектов.

В частном случае, в C# *объектом* являются элементы пользовательского интерфейса, которые создаются на Форме пользователя (UserForm) или на рабочем листе, а также рабочая книга и её элементы.

Определение 2. *Класс* - совокупность объектов, характеризующихся общностью применяемых методов обработки или свойств.

Определение 3. *Свойство* - характеристика объекта, его параметр.

Определение 4. *Метод* - программа действий над объектом или его свойствами.

Метод рассматривается как **программный код**, связанный с определенным **объектом**; осуществляет преобразование свойств, изменяет поведение объекта.

Объект может обладать набором заранее определенных встроенных методов обработки, либо созданных пользователем или заимствованных в стандартных библиотеках, которые выполняются при наступлении *заранее определенных событий,* например, однократное нажатие левой кнопки мыши, вход в поле ввода, выход из поля ввода, нажатие определенной клавиши и т.п.

Определение 5. *Событие* - изменение состояния объекта.

Внешние события генерируются пользователем (например, клавиатурный ввод или нажатие кнопки мыши, выбор пункта меню, запуск макроса);

внутренние события генерируются системой.

Инкапсуляция (замыкание) свойств данных и программ в объекте. Инкапсуляция – это скрытие информации.
При ООП возможен доступ к объекту только через его методы и свойства. Внутренняя структура объекта скрыта от пользователя, т.е. объекты – это самостоятельные сущности, отделенные от внешнего мира.

Инкапсуляция позволяет изменять реализацию объектов любого класса без опасений, что это вызовет нежелательные побочные эффекты в программной системе. Это мощное средство обеспечивает многократное использование одного и того же программного кода, позволяя собирать программу из готовых модулей, как здание из отдельных кирпичиков, но различной архитектуры и функционального назначения.

Наследование – это возможность выделить свойства, методы и события одного объекта и приписать их другому объекту, иногда с их модификацией.

Класс может иметь образованные от него *подклассы.* При построении подклассов осуществляется наследование данных и методов обработки объектов исходного класса.

Механизм наследования позволяет переопределить или добавить новые данные и методы их обработки, создать иерархию классов. С точки зрения программиста, новый класс должен содержать только коды и данные для новых или изменяющихся методов.

Полиморфизм – способность объекта реагировать на запрос (вызов метода) сообразно своему типу, при этом одно и то же имя метода может использоваться для различных классов объектов. Например, команда Print, будет по-разному воспринята черно-белым или цветным принтером.

Создание С знаменует собой начало современной эпохи программирования. Язык С был разработан **Деннисом Ритчи** (Dennis Ritchie) в 1970-е годы для программирования на мини-ЭВМ DEC PDP-11 под управлением операционной системы Unix. Несмотря на то что в ряде предшествовавших языков, в особенности Pascal, был достигнут значительный прогресс, именно С установил тот образец, которому до сих пор следуют в программировании.

Язык С появился в результате революции в *структурном программировании* в 1960-е годы. До появления структурного программирования писать большие программы было трудно, поскольку логика программы постепенно вырождалась в так называемый "макаронный" код — запутанный клубок безусловных переходов, вызовов и возвратов, которые трудно отследить.

В *структурированных* языках программирования этот недостаток устранялся путем ввода строго определенных управляющих операторов, подпрограмм с локальными переменными и других усовершенствований. Благодаря применению методов *структурного* программирования сами программы стали более организованными, надежными и управляемыми.

Язык C++ был разработан в 1979 году **Бьярне Страуструпом** (Bjarne Stroustrup), работавшим в компании Bell Laboratories, базировавшейся в Мюррей-Хилл, шт. Нью- Джерси.

Первоначально новый язык назывался "С с классами", но в **1983** году он был переименован в C++. Язык С полностью входит в состав C++, а следовательно, С служит основанием, на котором зиждется C++. Большая часть дополнений, введенных Страуструпом, обеспечивала плавный переход к ООП. И вместо того чтобы изучать совершенно новый язык, программирующему на С требовалось лишь освоить ряд новых свойств, чтобы воспользоваться преимуществами методики ООП.

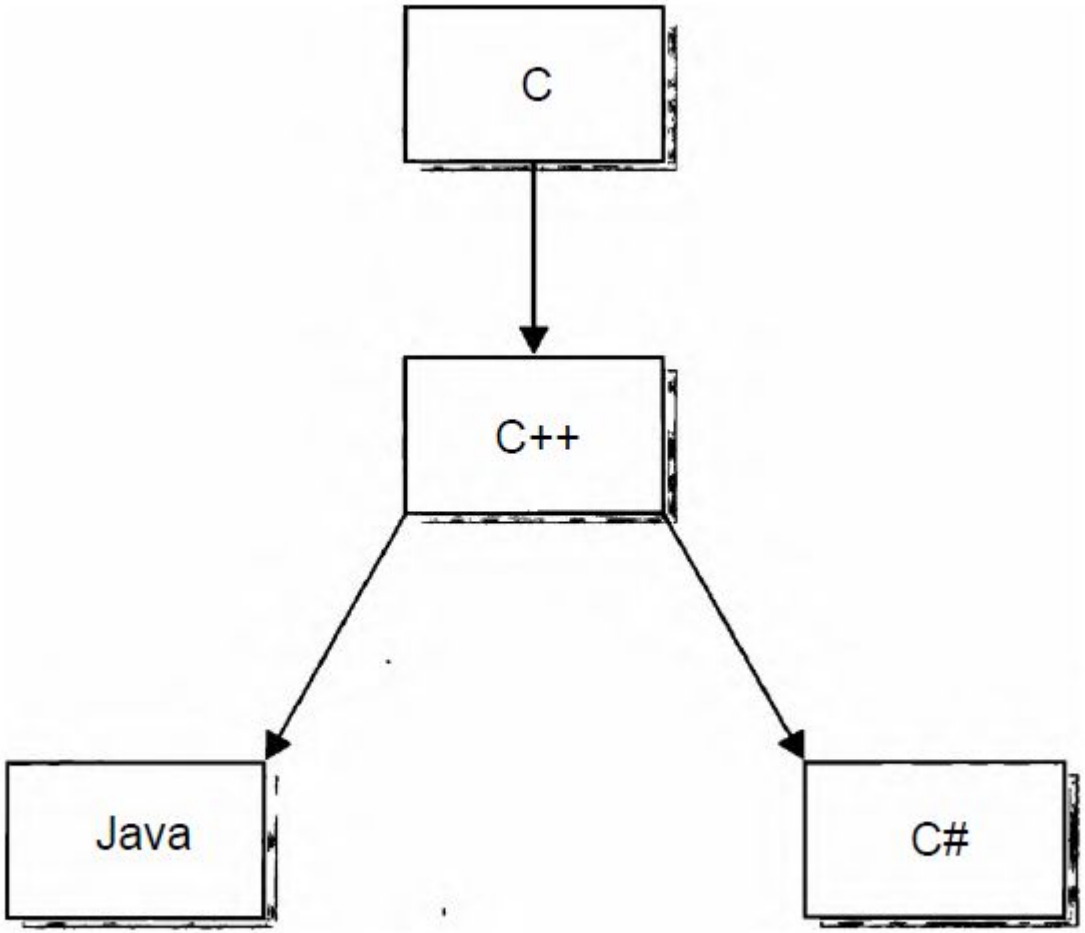


Рис. 1.1. Генеалогическое дерево C#

Появление Интернета и Java

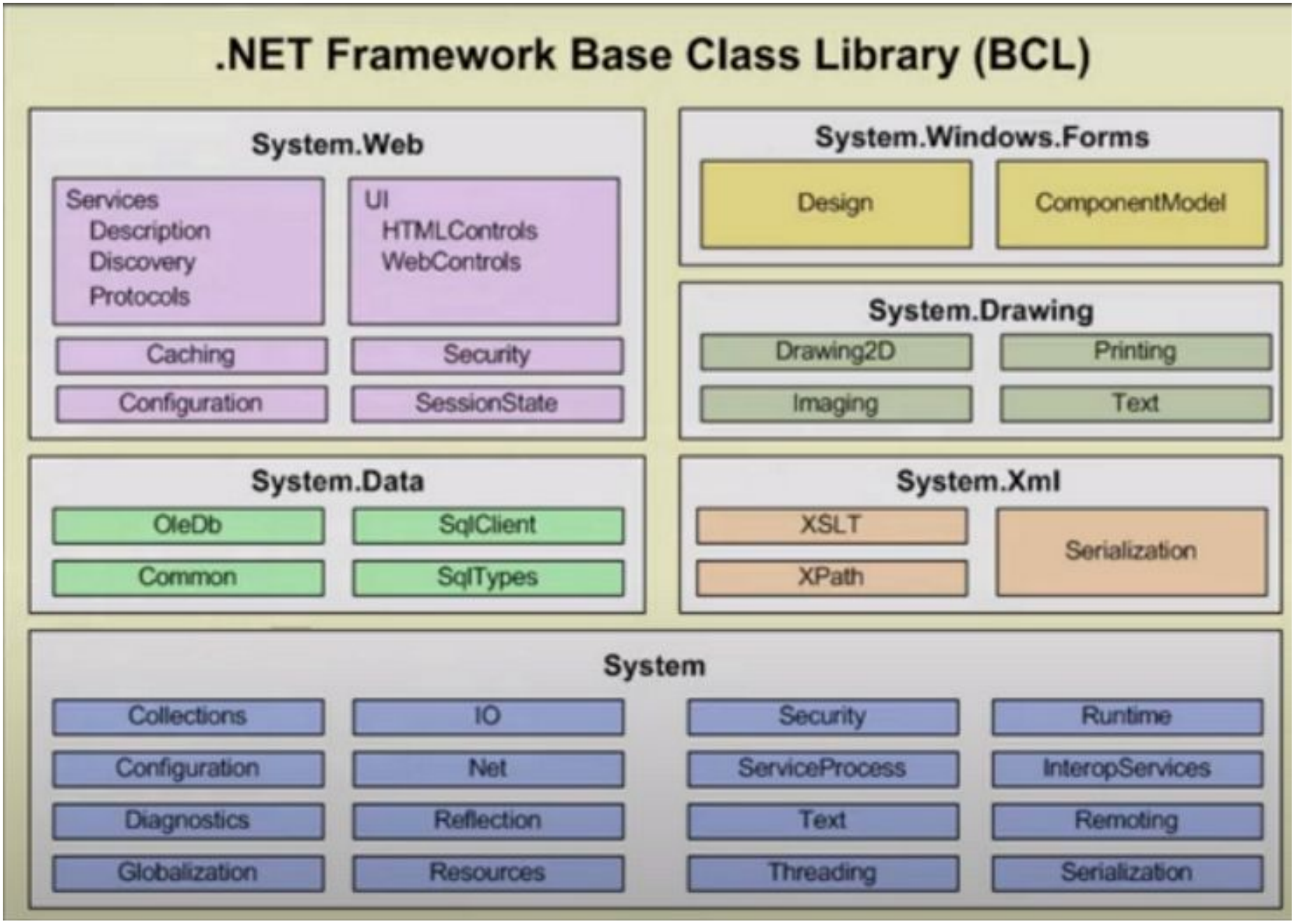
Следующим важным шагом в развитии языков программирования стала разработка **Java**. Работа над языком Java, который первоначально назывался Oak (Дуб), началась в 1991 году в компании Sun Microsystems. Главной "движущей силой" в разработке Java был Джеймс Гослинг.

Самым важным свойством (и причиной быстрого признания) **Java** является способность создавать межплатформенный, переносимый код, первоначальным толчком для разработки Java послужил не Интернет, а потребность в независящем от платформы языке, на котором можно было бы разрабатывать программы для встраиваемых контроллеров. В 1993 году стало очевидно, что вопросы межплатформенной переносимости, возникавшие при создании кода для встраиваемых контроллеров, стали актуальными и при попытке написать код для Интернета.

Переносимость программ на Java достигалась благодаря преобразованию исходного кода в промежуточный, называемый байт-кодом. Этот байт-код затем выполнялся виртуальной машиной Java (**JVM**) — основной частью исполняющей системы Java.

Java происходит от C и C++. В основу этого языка положен синтаксис C, а его объектная модель получила свое развитие из C++. И хотя код Java не
совместим с кодом C или C++ ни сверху вниз, ни снизу вверх, его синтаксис очень похож на эти языки, что позволяет большому числу программирующих на C или C++ без особого труда перейти на Java. Java построен по уже существующему образцу, что позволило разработчикам этого языка сосредоточить основное внимание на новых и передовых его свойствах.

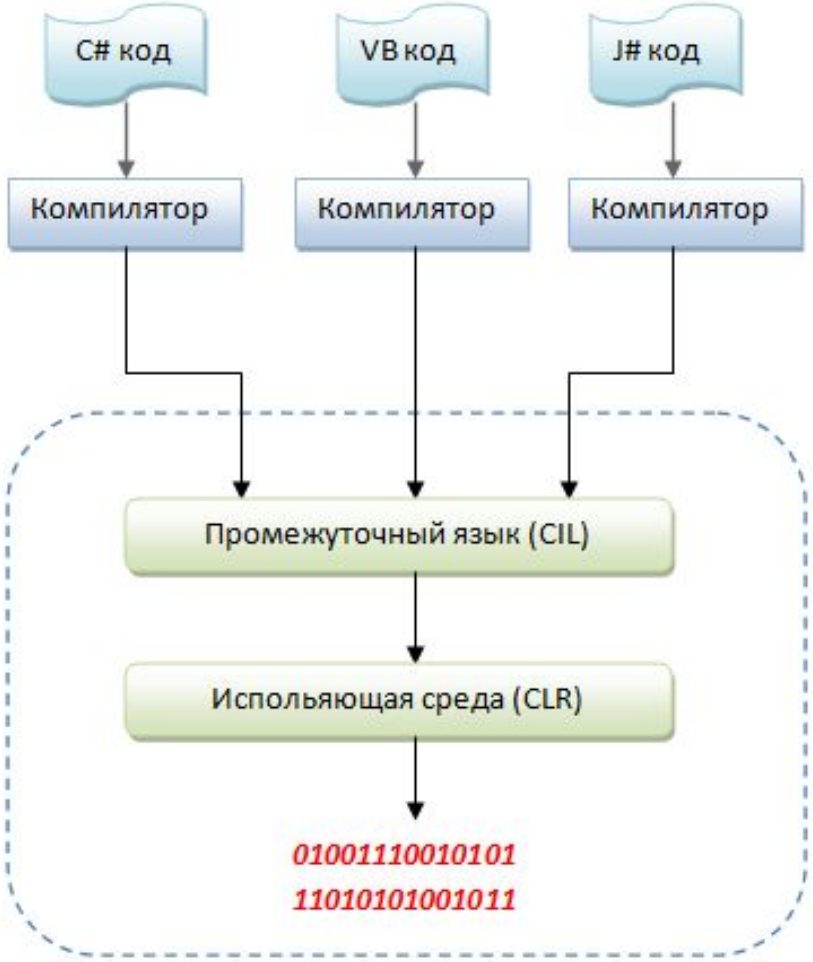
.Net (читается как «дот нет») – это кроссплатформенная среда выполнения приложений. Это то, что позволяет запускаться нашим приложениям в системе Microsoft Windows. Кросплатформенная – означает, что созданное приложение будет работать на всех процессорах и на всех операционных системах семейства Windows (за исключением самых ранних).


.NET Framework Base Class Library (BCL)
System.Web
Services
Description
Discovery
Protocols
UI
HTMLControls
WebControls
Caching
Security
Configuration
SessionState
System.Windows.Forms
Design
ComponentModel
System.Drawing
Drawing2D
Printing
Imaging
Text
System.Data
OleDb
SqlClient
Common
SqlTypes
System.Xml
XSLT
Serialization
XPath
System
Collections
IO
Security
Runtime
Configuration
Net
ServiceProcess
InteropServices
Diagnostics
Reflection
Text
Remoting
Globalization
Resources
Threading
Serialization

Назначение **.NET Framework** — служить средой для поддержки разработки и выполнения сильно распределенных компонентных приложений. Она обеспечивает совместное использование разных языков программирования, а также безопасность, переносимость программ и общую модель программирования для платформы Windows.

Для C# среда .**NET Framework** определяет два очень важных элемента.
Первым из них является *общеязыковая среда выполнения* (Common Language Runtime — **CLR**). Это система, управляющая выполнением программ. Среди прочих преимуществ — **CLR** как составная часть среды .**NET Framework** поддерживает многоязыковое программирование, а также обеспечивает переносимость и безопасное выполнение программ.

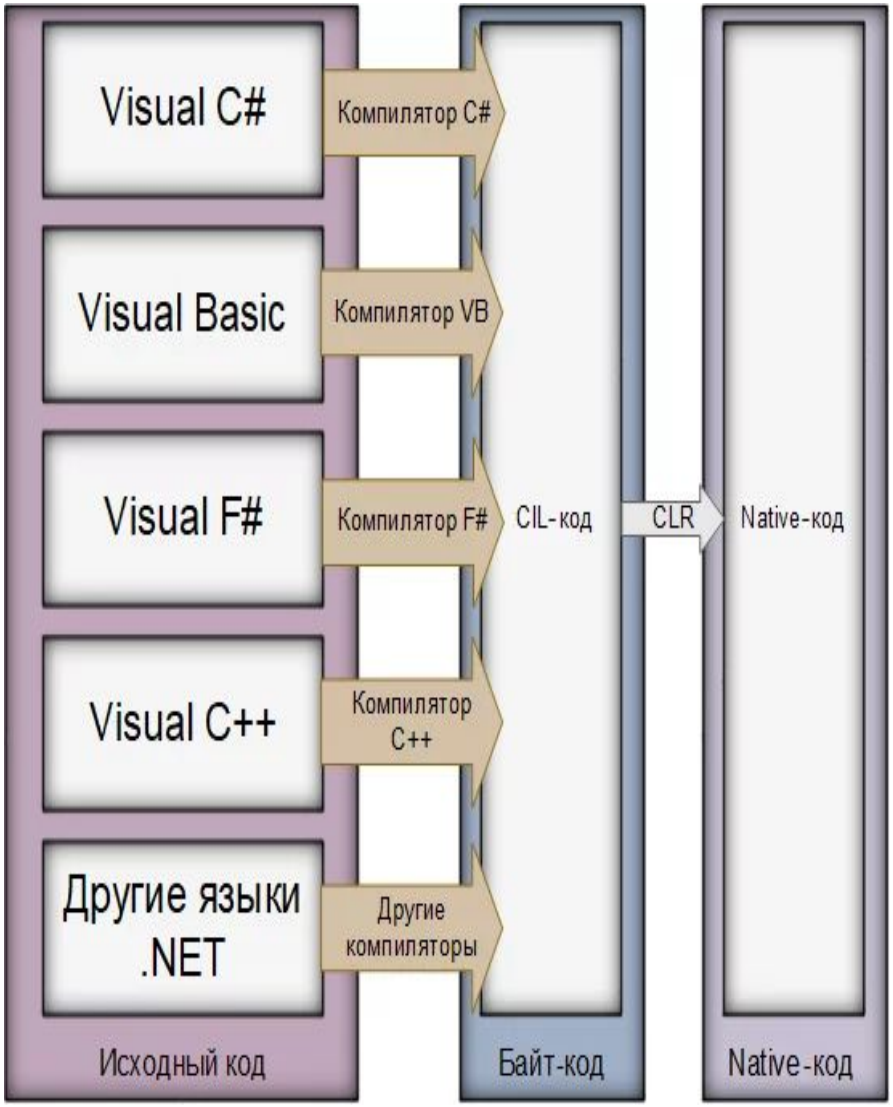
Вторым элементом среды .**NET Framework** является *библиотека классов*. Эта библиотека предоставляет программе доступ к среде выполнения. Так, если требуется выполнить операцию ввода-вывода, например вывести что-нибудь на экран, то для
этой цели используется библиотека классов .NET.



Код из любого языка преобразовывается в код, написанный на общем языке (Common intermediate language или CIL). Этот язык является языком низшего уровня, похожего по синтаксису на язык ассемблер.

После, этот код передаётся так называемой исполняющей среде (Common language runtime или CLR), которая берёт функции и методы из .**net Framework**

После этого конечный результат передаётся на процессор и выполняется программа.



CLI ***(Common Language Infrastructure*** — *общеязыковая инфраструктура)*. Она определяет, как работает *.NET* .
В *CLI* у каждого языка есть свой компилятор. Но программы компилируются не в нативный код *(исполняемый)*, а в промежуточный байт-код **CIL** ***(Common Intermediate Language*** — *общий промежуточный язык).*

Когда вы запускаете программу, написанную на одном из языков семейства *.NET,* её байт-код передаётся дальше по цепи в общеязыковую исполняющую среду CLR ***(Common Language Runtime)***. Там этот байт-код компилируется в нативный и уже начинает выполняться.
Почти по такому же принципу работает виртуальная машина Java, но программы на *.NET* быстрее запускаются, что делает их пригодными для работы не только на сервере, но и на персональных компьютерах.

F# (F Sharp или Эф шарп) - это функциональный статически типизированный язык программирования общего пользования, который создан и развивается компанией Microsoft и который предназначен для широкого круга задач.

Отличительной чертой F# является то, что он работает поверх платформы .NET и тем самым позволяет в определенной степени использовать возможности, предоставляемые этой платформой, например, систему типов, систему сборки мусора и т.д. Это также означает, что при компиляции код на F# компилируется в промежуточный язык IL (Intermediate Language), понятный для платформы .NET. И при запуске .NET управляет выполнением этого приложения.

Кроме того, благодаря этому мы можем в проекте на F# использовать вспомогательные библиотеки, написанные с помощью других .NET-языков (например, на C# или VB.NET). Подобным образом мы можем на F# написать библиотеку и затем подключить ее в проекты на других .NET-языках.

Принцип действия CLR

Среда CLR управляет выполнением кода **.NET**. Действует она по следующему принципу.

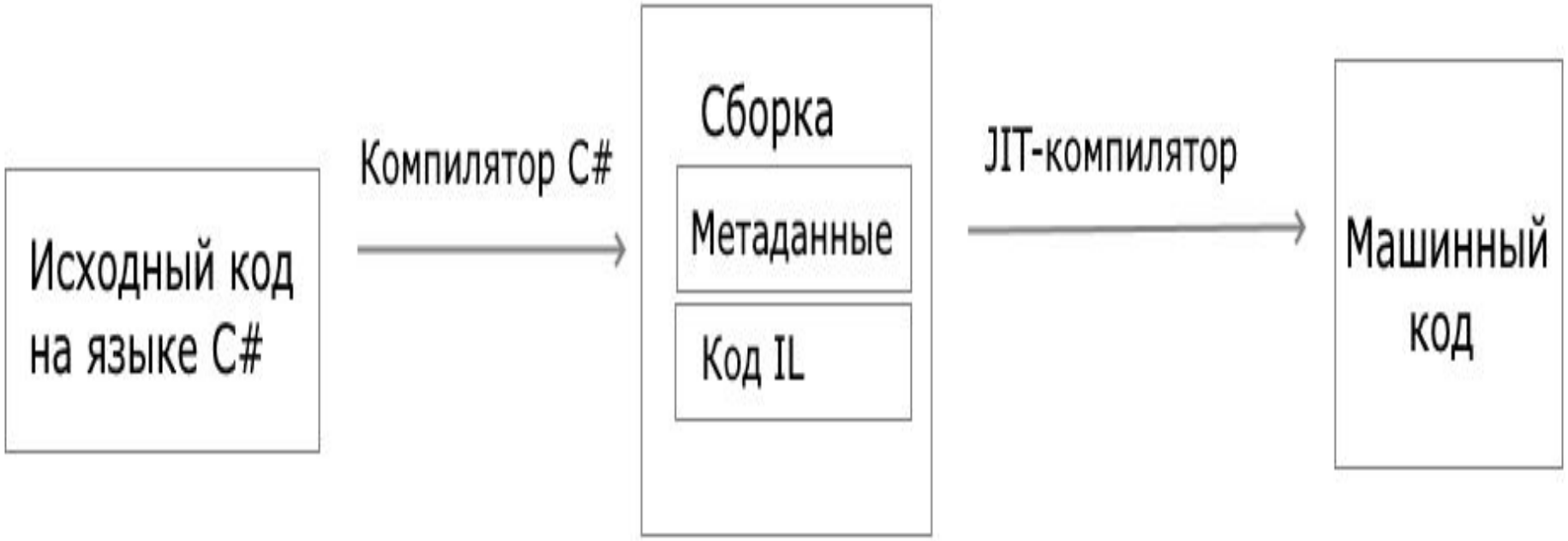
Результатом компиляции программы на C# является не исполняемый код, а файл, содержащий особого рода псевдокод, называемый *Microsoft Intermediate Language*, **MSIL** (промежуточный язык Microsoft, другое название - Common Intermediate Language - **CIL**).

Псевдокод **MSIL** определяет набор переносимых инструкций, не зависящих от конкретного процессора. По существу, **MSIL** определяет переносимый язык ассемблера.

CLR – это некая «виртуальная машина», которая собственно и управляет нашими приложениями, написанными для .net.

Назначение **CLR** — преобразовать промежуточный код в исполняемый код по ходу выполнения программы. Всякая программа, скомпилированная в псевдокод **MSIL**, может быть выполнена в любой среде, где имеется реализация CLR.

Таким образом достигается переносимость в среде **.NET Framework**.

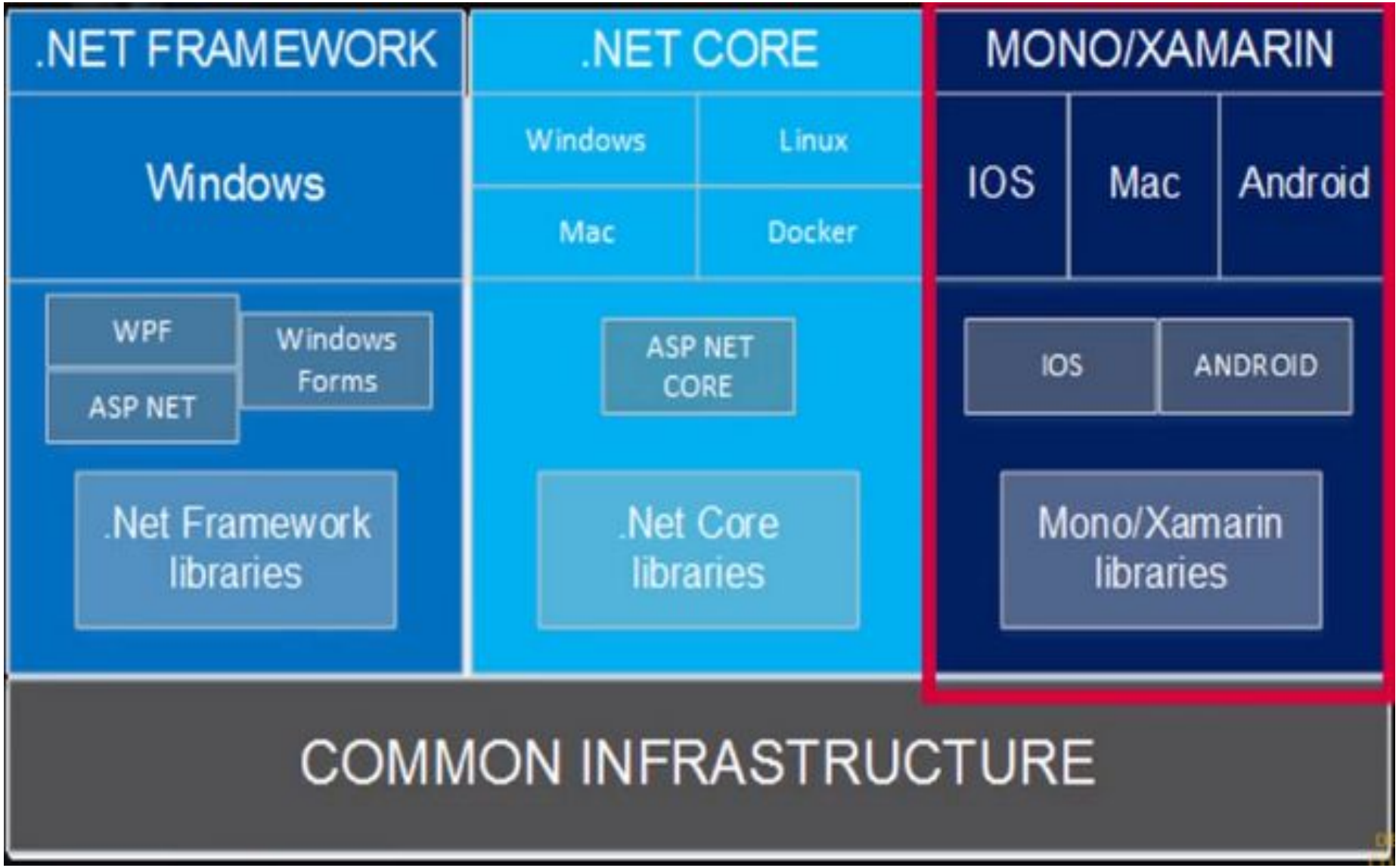

Исходный код на языке C#
Компилятор C#
Сборка
Метаданные
Код IL
JIT-компилятор
Машинный код

Псевдокод **MSIL** преобразуется в исполняемый код с помощью ***JIT-компилятора***. Сокращение **JIT** означает *точно в срок* и отражает оперативный характер данного компилятора.

Процесс преобразования кода происходит следующим образом. При выполнении программы среда **CLR** активизирует **JIT-компилятор**, который преобразует псевдокод **MSIL** в собственный код системы по требованию для каждой части программы.

Таким образом, программа на C# фактически выполняется как собственный код, несмотря на то, что первоначально она скомпилирована в псевдокод MSIL.

Это означает, что такая программа выполняется так же быстро, как и в том случае, когда она исходно скомпилирована в собственный код, но в то же время она приобретает все преимущества переносимости псевдокода MSIL.


.NET FRAMEWORK
Windows
WPF
ASP NET
Windows Forms
.Net Framework libraries
.NET CORE
Windows
Linux
Mac
Docker
ASP NET CORE
.Net Core libraries
MONO/XAMARIN
IOS
Mac
Android
IOS
ANDROID
Mono/Xamarin libraries
COMMON INFRASTRUCTURE

```
// A skeleton of a C# program
using System;
  namespace YourNamespace
  {
    class YourClass { }
    struct YourStruct { }
    interface IYourInterface { }
    delegate int YourDelegate();
    enum YourEnum { }
     namespace YourNestedNamespace{
     struct YourStruct { }
                    }
    class YourMainClass                    {
    static void Main(string[] args) {
     //Your program starts here...
                          }
                                 }
    }
```

```
using System;
class YouName{                                    }
class Example //произвольное имя класса
{
// Любая программа на C# начинается с вызова
метода Main().
static void Main() {
Console.WriteLine("Простая программа");
        }
}
```

Тип	Значение
bool	Логический, предоставляет два значения: “истина” или “ложь”
byte	8-разрядный целочисленный без знака
char	Символьный
decimal	Десятичный (для финансовых расчетов)
double	С плавающей точкой двойной точности
float	С плавающей точкой одинарной точности
int	Целочисленный
long	Длинный целочисленный
sbyte	8-разрядный целочисленный со знаком
short	Короткий целочисленный
uint	Целочисленный без знака
ulong	Длинный целочисленный без знака
ushort	Короткий целочисленный без знака

byte	8	0-255
sbyte	8	-128-127
ushort	16	0-65 535
short	16	-32 768-32 767
int	32	-2 147 483 648-2 147 483 647
uint	32	0-4 294 967 295
long	64	-9 223 372 036 854 775 808-9 223 372 036 854 775 807
ulong	64	0-18 446 744 073 709 551 615

Тип **decimal**, который предназначен для применения в финансовых расчетах. Этот тип имеет разрядность **128бит** для представления числовых значений в пределах от 1E-28 до 7,9E+28.

Для обычных арифметических вычислений с пла вающей точкой характерны ошибки округления дес ятичных значений. Эти ошибки исключаются при использовании типа decimal, который позволяет представить числа с точностью до 28 (а иногда и 29) десятичных разряд ов.

Благодаря тому что этот тип данных способен представлять десятичные значения без ошибок округления, он особенно удобен для расчетов, связанных с финансами.

```
decimal price;
decimal discount;
decimal discounted_price;
//
price = 19.95m;
discount = 0.15m; // норма скидки
составляет 15%
discounted_price = price - ( price *
discount);
Console.WriteLine ("Цена со скидкой: $" +
discounted_price);
```

Без суффикса m эти значения интерпретировались бы как стандартные константы с плавающей точкой, которые несовместимы с типом данных **decimal**. Тем не менее переменной типа **decimal** можно присвоить целое значение без суффикса m, например 10.

```
float x=3.14;

Console.WriteLine("Вы заказали " + 2 + "
предмета по цене $" + 3 + " каждый.“+x);
```

`Console.`**WriteLine(***"форматирующая строка", arg0, arg1, … , argN***);**

{*argnum, width: fmt*}

где *argnum* — номер выводимого аргумента, начиная с нуля; *width* — минимальная ширина поля; *fmt* — формат. Параметры *width* и *fmt* являются необязательными.

```
Console.WriteLine("В феврале {0} или {1} дней.", 28, 29);
Console.WriteLine("В феврале {0,10} или {1,5} дней.", 28, 29);
Console.WriteLine("{0}\t{1}\t{2}", i, i*i, i*i*i);
Console.WriteLine("{0:###,###.##}", 123456.56);
```

C Денежная единица Задает количество десятичных разрядов

D Целочисленный (используется только с целыми числами)Задает минимальное количество цифр. При необходимости результат дополняется начальными нуля

E Экспоненциальное представление чисел (в обозначении используется прописная буква E) Задает количество десятичных разрядов. По умолчанию используется шесть разрядов

F Представление чисел с фиксированной точкой Задает количество десятичных разрядов

N Представление чисел с фиксированной точкой (и запятой в качестве разделителя групп разрядов) Задает количество десятичных разрядов

P Проценты Задает количество десятичных разрядов P

X Шестнадцатеричный (в обозначении используются прописные буквы A-F) Задает минимальное количество цифр. При необходимости результат дополняется начальными нулями

```
int a = 12346;
Console.WriteLine(Convert.ToString(a,2));
Console.WriteLine(Convert.ToString(a, 8));
Console.WriteLine(Convert.ToString(a, 10));
Console.WriteLine(Convert.ToString(a, 16));

Console.WriteLine("{0:0.00}", 123.4567);     // "123.46"
Console.WriteLine("{0:0.00}", 123.4);    // "123.40"
Console.WriteLine("{0:0.00}", 123.0);          // "123.00"
Console.WriteLine("{0:0.##}", 123.4567);       // "123.46"
String.Format("{0:0.##}", 123.4);        // "123.4"
Console.WriteLine("{0:0.##}", 123.0);          // "123"
String.Format("{0:00.0}", 123.4567);       // "123.5"
Console.WriteLine("{0:00.0}", 23.4567);        // "23.5"
Console.WriteLine("{0:00.0}", 3.4567);         // "03.5"
Console.WriteLine("{0:00.0}", -3.4567);        // "-03.5"
```

Операция	**Значение**
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Не равно

Оператор	**Действие**
+	Сложение
-	Вычитание, унарный минус
*	Умножение
/	Деление
%	Деление по модулю
--	Декремент
++	Инкремент

Управляющая последовательность	**Описание**
\a	Звуковой сигнал (звонок)
\b	Возврат на одну позицию
\f	Перевод страницы (переход на новую страницу)
\n	Новая строка (перевод строки)
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\0	Пустой символ
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта

Оператор	Значение
&	Поразрядное И
|	Поразрядное ИЛИ
^	Поразрядное исключающее ИЛИ
>>	Сдвиг вправо
<<	Сдвиг влево
~	Дополнение до 1 (унарный оператор НЕ)

```
val = val >> 5; // сдвиг вправо на 5 бит
```

Таблица 4.2. Предшествование операторов в C#

Наивысший порядок									
()	[]	.	++ (постфиксный)	-- (постфиксный)	checked	new	sizeof	typeof	unchecked
!	~	(приведение типов)	+ (унарный)	– (унарный)	++ (префиксный)	-- префиксный)			
*	/	%							
+	–								
<<	>>								
<	>	<=	>=	is					
==	!=								
&									
^									
|									
&&									
||									
??									
?:									
=	op=	=>							
Наинизший порядок									

Упражнение1

1. Создать консольное приложение:

```
int a = 42;
        int b = 119;
        int c = a + b;
        Console.WriteLine(c);
        Console.ReadKey();
```

Задание

1. Объявить целые числа X и У и присвоить им значения 12 и 98.
2. Вычислить значение $У^2 - X^3$.
3. Вывести на экран сообщение в следующем виде:

X=12, У=98

Результат: *полученное число*

В C# можно явно указать другой тип с помощью **суффикса**. Так, для указания типа **long** к литералу присоединяется суффикс `l` или `L`. Например, `12` — это литерал типа **int**, а **`12L`** — литерал типа **long**.

Для указания целочисленного типа без знака к литералу присоединяется суффикс `u` или `U`. Следовательно, `100` — это литерал типа **int**, а **`100U`** — литерал типа `uint`.

А для указания длинного целочисленного типа без знака к литералу присоединяется суффикс `ul` или `UL`. Например, **`984375UL`** — это литерал типа `ulong`.

Для указания типа `float` к литералу присоединяется суффикс `F` или `f`. Например, **`10.19F`** — это литерал типа `float`. Можно указать тип `double`, присоединив к литералу суффикс `d` или `D`, хотя это излишне по умолчанию литералы с плавающей точкой относятся к типу `double`.

Для указания типа **decimal** к литералу присоединяется суффикс `m` или `M`. Например, **`9.95M`** — это десятичный литерал типа `decimal`.

Неявно типизированные переменные

Как правило, при объявлении переменной сначала указывается тип, например int или bool, а затем имя переменной. Но компилятору предоставляется возможность самому определить тип локальной переменной, исходя из значения, которым она инициализируется. Такая переменная называется *неявно типизированной*.

Неявно типизированная переменная объявляется с помощью ключевого слова **var** и должна быть непременно инициализирована. Для определения типа этой переменной компилятору служит тип ее инициализатора, т.е. значения, которым она инициализируется.

```
var e = 2.7183;
```

В данном примере переменная e инициализируется литералом с плавающей точкой, который по умолчанию имеет тип **double**, и поэтому она относится к типу **double**. Если бы переменная e была объявлена следующим образом:

```
var e = 2.7183F;
```

то она была бы отнесена к типу **float**.

Преобразование и приведение типов

```
int k;  float f;          k=10; f=k;
```

Вследствие строгого контроля типов далеко не все типы данных в C# оказываются полностью совместимыми, а следовательно, не все преобразования типов разрешены в неявном виде. Например, типы bool и int несовместимы. Преобразование несовместимых типов все-таки может быть осуществлено путем *приведения.*

Автоматическое преобразование типов

Когда данные одного типа присваиваются переменной другого типа, *неявное* преобразование типов происходит автоматически при следующих условиях:

- оба типа совместимы;
- диапазон представления чисел целевого типа шире, чем у исходного типа.

```
long L;
double D;
L = 100123285L;
D = L;
```

Приведение несовместимых типов

Несмотря на всю полезность неявных преобразований типов, они неспособны удовлетворить все потребности в программировании, поскольку допускают лишь расширяющие преобразования совместимых типов. А во всех остальных случаях приходится обращаться к приведению типов.

Приведение — это команда компилятору преобразовать результат вычисления выражения в указанный тип. А для этого требуется явное преобразование типов.

```
(целевой_тип) выражение
```

Здесь *целевой_тип* обозначает тот тип, в который желательно преобразовать указанное *выражение*.

Пример.

```
double x, y;
```

Если результат вычисления выражения x/y должен быть типа int, то следует записать следующее.

```
int i;
i=(int) (x / y);
```

Упражнение1*

1.Используя методы Write(), WriteLine() и управляющие последовательности (\t, \n, …) вывести на консоль таблицу умножения, выдерживая интервалы между столбцами.

```
*   |  1 |  2 |   3  |  4   |
-----------------------------------
1   |  1 |  2 |   3  |  4   |
2   | 2  |  4 |   6  |  8   |
3   |  3 |  6 |   9  |  12  |
4   | 4  |  8 |   12 |  16  |
```

```
if   (условие)
{ последовательность операторов A;
}
B;
```

```
if   (условие)
оператор;
else          if (условие)
        оператор;
          else          if (условие)
                  оператор;
                  else
                  оператор;
```

switch

```
int i;
for(i=0; i<10; i++)
switch(i) {
case 0: Console.WriteLine("i равно нулю");
break;
case 1: Console.WriteLine("i равно единице");
break;
case 2: Console.WriteLine("i равно двум");
break;
case 3: Console.WriteLine("i равно трем");
break;
case 4: Console.WriteLine("i равно четырем");
break;
default: Console.WriteLine("i равно или больше
пяти");
break;
   }
```

Оператор ?

Оператор **?** относится к числу самых примечательных в C#. Он представляет собой условный оператор и часто используется вместо определенных видов конструкций if-then-else.

Оператор **?** иногда еще называют *тернарным,* поскольку для него требуются три операнда. Ниже приведена общая форма этого оператора.

Выражение1 ? *Выражение2 : Выражение3;*

Здесь *Выражение1* должно относиться к типу bool, а *Выражение2* и *Выражение3* — к одному и тому же типу.

absval = val < 0 ? -val : val; // получить абсолютное значение переменной val

```
for(x = 100;   x > -100;   x -= 5){….}
```
c.131

```
for(i=0,   j=10;   i < j;   i++, j--){}
```

```
for(i=2, j=num/2; (i <= num/2) & (j >= 2); i++, j--){}
```

```
int i, j;                    bool done = false;
for(i=0, j=100; !done; i++, j--) {
if(i*i >= j) done = true;
Console.WriteLine("i, j: " + i + " " + j);
```

```
for(i = 0; i < 10; ) {Console.WriteLine("Проход №" + i);
i++; // инкрементировать переменную управления циклом}
```

```
for (;;){ ….      break;}
```

```
for(i = 1; i <= 5; sum += i++);
```

Упражнение2

1. Написать программу в которой переменной X присвоить значение 3,14.
2. Используя цикл вычислить куб и квадрат 5 чисел полученных из X с шагом 0,01.
3. Используя форматированный вывод на консоль вывести таблицу вида:

Значения:

3.14	**3.15**	**3.16**	**3.17**	**3.18**
квадрат	***квадрат***	***квадрат***	***квадрат***	***квадрат***
квадрат				
Куб	***куб***	***куб***	***куб***	***куб***

Упражнение3

1. Написать программу, которая присваивает переменной целое число 9037600125.
2. Подсчитать количество значащих цифр в числе и вывести результат на экран.
3. Необходимо вывести это число в заданном виде и с цифрами в обратном порядке.
4. Подсчитать сумму цифр этого числа.
5. Вывести на экран сообщение в следующем виде:

9037600125 ---> 5210067309
Число цифр 10.
Сумма цифр числа 9037600125 равна 33

```
int num=435679;          int mag=0;
while( num > 0) {
mag++;
num = num / 10;
};
```

```
int num=198;    int nextdigit;
do {
nextdigit = num % 10;
Console.Write(nextdigit); //Число в обратном порядке
num = num / 10;
} while (num > 0);
```

```
int sum = 0;              int[] nums = new int[10];
foreach (int x in nums) {
sum += x;
}
```

- **break**;

- С помощью оператора **continue** можно организовать преждевременное завершение шага итерации цикла в обход обычной структуры управления циклом.

```
        for (int i = 0; i <= 100; i++) {
if((i%2) != 0) continue; // перейти к следующему шагу итерации
     Console.WriteLine(i);              }
```

```
x = 1;
loop1:
x++;
if(x < 100) goto loop1;
```

Упражнение4

1. Написать цикл, который выводит все числа из диапазона от 1 до 3000, которые делятся без остатка одновременно и на 4 и на 7 и на 11.
2. Полученные числа вывести в одну строку на экран.
3. На второй строке вывести все простые числа из указанного диапазона.
4. На третьей строке вывести количество всех простых чисел.

```
char ch;
Console.Write("Нажмите клавишу, а затем – <ENTER>: ");
ch = (char) Console.Read(); // получить значение типа char

string str;
Console.WriteLine("Введите несколько символов.");
str = Console.ReadLine();
```

Имя структуры в .NET	Имя типа данных в C#
Decimal	decimal
Double	double
Single	float
Int16	short
Int32	int
Int64	long
UInt16	ushort
UInt32	uint
Uint64	ulong
Byte	byte
Sbyte	sbyte

```
string  str;      int  n;
str = Console.ReadLine();
n = Int32.Parse(str);
```

Упражнение5

1. Вывести сначала все четные числа, разделенные запятыми, затем нечетные в интервале от 0 до 20.
2. С клавиатуры ввести число 216, вывести на экран через запятую все множители этого числа кратные 3.
3. Разложить на сомножители целое число, веденное с клавиатуры и

Упражнение6

Задать с клавиатуры x, y.

$$z1 = \frac{3x}{y} - \frac{y}{7x}\left(\frac{9}{x-1} - \frac{13}{y+1}\right)$$

$$z2 = \frac{x^2 + y^2 - \frac{x}{2y}}{\frac{y^2 - 2{,}5}{x^2 + 1}}$$

Ответ вывести в формате с плавающей точкой и в экспоненциальной форме.

Упражнение7

1. Ввести 4 произвольных числа.
2. Вывести на экран наибольшее и наименьшее из этих чисел (встроенные методы и массивы не использовать).
3. Вывести вновь на экран эти числа в строку по возрастанию и по убыванию.
4. Для повторного ввода чисел

В C# используется спецификация класса для построения ***объектов***, которые являются *экземплярами* класса. Следовательно, класс, по существу, представляет *собой ряд схематических описаний способа построения объекта*. При этом очень важно подчеркнуть, что класс является логической **абстракцией**.

Физическое представление класса появится в оперативной памяти лишь после того, как будет создан объект этого класса.

```
class имя_класса {
// Объявление переменных экземпляра.
доступ тип переменная1;
доступ тип переменная2;
//...
доступ тип переменнаяN;

// Объявление методов.
доступ возвращаемый_тип метод1(параметры)
 {
// тело метода
 }
доступ возвращаемый_тип метод2(параметры)
{
// тело метода
}
// ...
доступ возвращаемый_тип методы(параметры) {
// тело метода
}
}
```

```
class Building {
public int Floors; // количество этажей
public int Area; // общая площадь здания
public int Occupants; // количество жильцов
}
```

Для того чтобы создать конкретный объект типа **Building**, придется воспользоваться следующим оператором.

```
Building house = new Building(); // создать объект типа Building

house.Floors = 2;
```

```
Building house1 = new Building();
Building house2 = house1;
```

На первый взгляд, переменные **house1** и **house2** ссылаются на совершенно разные объекты, но на самом деле это не так. Переменные **house1** и **house2**, напротив, ссылаются на один и тот же объект. Когда переменная house1 присваивается переменой house2, то в конечном итоге переменная house2 просто ссылается на тот же самый объект, что и переменная house1. Следовательно, этим объектом можно оперировать с помощью переменной **house1** или **house2**. Например, после очередного присваивания

```
house1.Area = 2600;

Console.WriteLine(house1.Area);
Console.WriteLine(house2.Area);
```

выводят одно и то же значение: 2600.

Модификаторы доступа

Управление доступом в языке C# организуется с помощью четырех *модификаторов доступа:* `public`, `private`, `protected` и `internal`.

Защищенный член создается с помощью модификатора доступа **protected**. Если

член класса объявляется как **protected**, он становится закрытым, но за исключением

одного случая, когда защищенный член наследуется. В этом случае защищенный член

базового класса становится защищенным членом производного класса, а значит, доступным для производного класса. Таким образом, используя модификатор доступа

protected, можно создать члены класса, являющиеся закрытыми для своего класса, но все же наследуемыми и доступными для производного класса.

internal этот модификатор определяет доступность члена во всех файлах сборки и его недоступность за пределами сборки. О члене, обозначенном как **internal**, известно только в самой программе, но не за ее пределами. Модификатор доступа **internal** особенно полезен для создания программных компонентов. Модификатор доступа **internal** можно применять к классам и их членам, а также к структурам и членам структур. Кроме того, модификатор **internal** разрешается использовать в объявлениях интерфейсов и перечислений. Из модификаторов **protected** и **internal** можно составить спаренный модификатор доступа **protected internal**. Уровень доступа **protected internal** может быть задан только для членов

protected

```
class B {
protected int i, j; // члены, закрытые для класса B, но
доступные для класса D
public void Set(int a, int b) {i = a;   j = b;   }
public void Show() {
Console.WriteLine(i + " " + j);       }            }
class D : B {
int k; // закрытый член члены i и j класса B доступны для
класса D
public void Setk() {k = i * j;   }
public void Showk() {Console.WriteLine(k);       } }

D ob = new D();
ob.Set(2, 3); // допустимо, поскольку доступно для класса D
ob.Show(); // допустимо, поскольку доступно для класса D
ob.Setk(); // допустимо, поскольку входит в класс D
ob.Showk(); // допустимо, поскольку входит в класс D
```

internal

```
class InternalTest {
internal int x;
}
class InternalDemo {
static void Main() {
InternalTest ob = new InternalTest();
ob.x = 10; // доступно, потому что находится в том же файле
Console.WriteLine("Значение ob.x: " + ob.x);
}
}
```

Упражнение9

1. Создать класс и в нем **метод** вычисления третьей степени произвольного числа.
2. Написать программу, которая при вводе числа вызывает созданный **метод** и выводит результат и только дробную часть полученного числа на экран .
3. Повторный ввод числа должен дать следующий результат.
4. При нажатии символа «q» программа прекращает свою работу.

```
double x;
string st=ConsoleReadLine();

while (st!="q")
{
    x=Double.Parse(st);
        ..............................
        Console.Write(x);
    st=ConsoleReadLine();
}
```

Упражнение10

$$Z(x, y)=\frac{x}{\sqrt[3]{e^2-(x+y)^{3/4}}}+\left[\frac{(x-y)^{\sin(x\cdot y)}}{\sqrt[3]{e^2-(x+y)^{3/4}}}\right]^{7/3}$$

Написать программу вычисления Z(x,y) из двух функций:

1. Создать свой класс.
2. В новом классе написать метод ввода дробных чисел x и y.
3. Написать метод вычисления одинаковых фрагментов формулы используя встроенный класс **Math**.
4. Написать метод вычисления всей формулы **Z_(double x, double y);**
5. Вычислить значение **Z(x, y**) путем вызова метода Z_(x, y), вывести на экран в виде:

Z(число x, число y)=ответ.

6. Важно следить чтобы не было деления на ноль и извлечение корня из отрицательного числа. Если решения нет вывести сообщение об ошибке.

Упражнение11

Задать исходные данные, произвести вычисления по заданным функциям* и вывести на экран в форматированном виде.

На экране в скобках вместо x
или должны выводится числовые
значения, **например**, cos(60)=0.5.

В результате на экране должны быть следующие записи:

Ввести значение угла в градусах: 45
sin(45)= cos(45)= tg(45)=

Ввести переменные, например:
X =2,3
Y=1,5
exp(2,3)= log(2,3)= log10(2,3)= pow(2,3,1,5)=
sqrt(2,3)= abs(2,3)= 2,3*2^1,5= остаток 1,3/1,5=

Конструктор инициализирует объект при его создании. У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу. Но у конструкторов нет возвращаемого типа, указываемого явно.

```
доступ имя_класса(список_параметров) {

// тело конструктора
}
```

Как правило, конструктор используется для задания первоначальных значений переменных экземпляра, определенных в классе, или же для выполнения любых других установочных процедур, которые требуются для создания полностью сформированного объекта.

Конструктор без параметров

```
class MyClass {
public int x;
public MyClass() {
x = 10;
}                                    }

class ConsDemo {
static void Main() {
MyClass t1 = new MyClass();
MyClass t2 = new MyClass();
Console.WriteLine(t1.x + " " + t2.x);
}                                     }
```

Параметризированные конструкторы

```
class MyClass {
public int x;
public MyClass(int i) {     x = i;     }
}
//-------------------------------------
--------
class ParmConsDemo {
static void Main() {

  MyClass t1 = new MyClass(10);
  MyClass t2 = new MyClass(88);

Console.WriteLine(t1.x + " " + t2.x);
} }
```

Деструктор

```
class Destruct {
public int x;
public Destruct (int i) {  x = i;          }
// Вызывается при утилизации объекта.
~Destruct() {Console.WriteLine("Уничтожить " +
x);}
// Создает объект и тут же уничтожает его.
public void Generator(int i) {
Destruct o = new Destruct(i);
}                              }

class DestructDemo {
static void Main() {
int count;
Destruct ob = new Destruct(0);
for(count=1; count < 100000; count++)
ob.Generator(count);
```

Ключевое слово **this** можно использовать в конструкторе. В этом случае оно обозначает объект, который конструируется.

```
class Rect {
public int Width;                    public int
Height;
public Rect (int w, int h) {
this.Width = w;
this.Height = h;
}
public int Area() {
return this.Width * this.Height;}
                      }
```

Упражнение*

Цех выпускает детали Т1 и Т2. Т1 – выпускает 90 штук в день, Т2 – выпускает 80 шт.

Для выполнения условий ассортиментности, которые предъявляются производством, продукция каждого из видов деталей должно быть выпущено не менее 20 процентов от возможного. Предполагая, что вся продукция цеха реализуется без остатка.

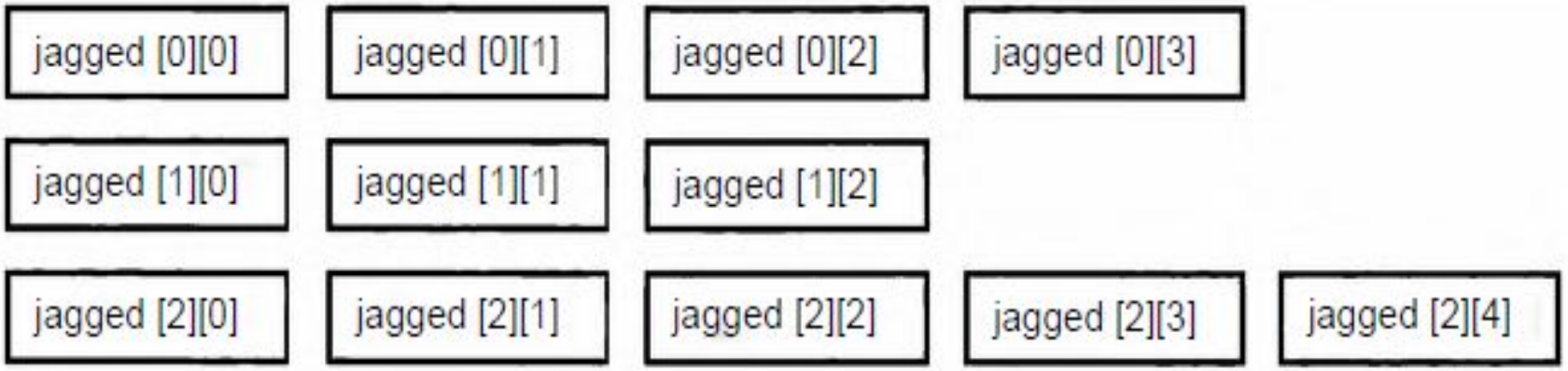
Написать программу в которой задаются параметры производства, себестоимость и цена деталей. Программа должна найдите максимально возможную прибыль цеха за один день (прибылью называется разница между отпуск ной стоимостью всей продукции и её себестоимостью). Вывести [illegible]тали Т1 и Т2.

Детали	Себестоимость	Отпускная цена
Т1	1500	2100
Т2	1100	1750

```
int[] sample = new int[10];

int[] sample;
sample = new int[10];

int[] nums = { 99, 10, 100, 18, 78,23, 63, 9, 87, 49 };

int[] nums = new int[] { 99,10,100,18,78,23,63,9};

int[,] table = new int[10, 20];
int[,] sqrs = { { 1, 1 },{ 2, 4 },{ 3, 9 },{ 4, 16 } };
```

```
var vals = new[] { 1, 2, 3, 4, 5 };
var vals = new[,] { {1.1, 2.2}, {3.3, 4.4},{ 5.5, 6.6} };
```

```
int[][] jagged = new int[3][];
jagged[0] = new int[4];
jagged[1] = new int[3];
jagged[2] = new int[5];
```



```
        for(int i = 0; i < 10; i++)nums[i] = i;
foreach (int x in nums) {
Console.Write("Значение элемента: " + x);
sum += x;
                    }
```

Присваивание ссылок на массивы

```
int[]x1=new int[]{2,7,4};        int[]x2=new int[3]{-1,-4,7};
x2=x1;// обе переменные ссылаются на один и тот же массив


int[] nums = new int[10];
Console.WriteLine("Длина массива nums равна " +
nums.Length);
for (int i=0; i < nums.Length; i++)
nums[i] = i * i;
```

Упражнение8

Задать массив MAS из 12 элементов формата int. Заполнить его числами в следующем порядке: -3, 10, 12, 1, 2, -4, 6, 12, -1, 2, -2, 12. Написать программу (*без использования класса Math!)*, которая выполняет следующие действия:

1) находит максимальное и минимальное число;
2) выводит сначала только четные, затем нечетные числа;
3) находит повторяющиеся числа выводит их и число их повторов.

Методы

```
доступ возвращаемый_тип имя(список_параметров)
{
// тело метода
}
```

где ***доступ*** — это модификатор доступа, определяющий те части программы, из которых может вызываться метод. Если он отсутствует, то метод оказывается закрытым (**private**) в пределах того класса, в котором он объявляется. Методы делают открытыми (**public**), чтобы вызывать их из любой другой части кода в программе.

возвращаемый_тип обозначает тип данных, возвращаемых методом. Этот тип должен быть действительным, в том числе и типом создаваемого класса. Если метод не возвращает значение, то в качестве возвращаемого для него следует указать тип **void**.

имя обозначает конкретное имя, присваиваемое методу. В качестве имени метода может служить любой допустимый идентификатор, не приводящий к конфликтам в текущей области объявлений.

список_параметров — это последовательность пар, состоящих из типа и идентификатора и разделенных запятыми. Параметры представляют собой переменные, получающие значение *аргументов,* передаваемых методу при его вызове. Если у метода отсутствуют параметры, то список параметров оказывается пустым.

```
class My_name
 {int x;  // x - по умолчанию private
public float My_func(int y)
{    x= y+10;
return (float)x*x/(4*x-11);
}

class Prog{
static void Main(){
     int a=13;
     My_name ob1=new My_name();
     ob1.x=11 // ошибка, здесь x – закрыто т.к. private
     float b=ob1.My_fun(a);
     }
}
```

```
using System;
class RandDice {
static void Main() {
Random ran = new Random();
Console.Write(ran.Next() + " ");
Console.WriteLine(ran.Next(1, 7));
}
    }
```

- public virtual int **Next**() Возвращает следующее случайное целое число, которое будет находиться в пределах от 0 до Int32. MaxValue-1 включительно.
- public virtual int **Next(int** maxValue) Возвращает следующее случайное целое число, которое будет находиться в пределах от 0 до maxValue-1 включительно.
- public virtual int **Next(int** minValue, int maxValue) Возвращает следующее случайное целое число, которое будет находиться в пределах от minValue до maxValue-1 включительно.
- public virtual void **NextBytes(byte**[] buffer) Заполняет массив buffer последовательностью случайных целых чисел. Каждый байт в массиве будет находиться в пределах от 0 до Byte.MaxValue-1 включительно.
- public virtual double **NextDouble**() Возвращает из последовательности следующее случайное число, которое представлено в форме с плавающей точкой, больше или равно 0,0 и меньше 1,0.
- protected virtual double **Sample**() Возвращает из последовательности следующее случайное число, которое представлено в форме с плавающей точкой, больше или равно 0,0 и меньше 1,0. Для получения несимметричного или специального распределения

Упражнение12

1. Написать программу, в которой задается начальное – A и конечное – B значения числового интервала для генерации целых случайных чисел.
2. Задать количество генерируемых случайных чисел.
3. Сгенерировать случайные числа и подсчитать сколько из них было кратно 3, 5, 8 и 11. Результат статистики вывести на экран:

Количество чисел кратные 3 – 354

Количество чисел кратные 5 – 178 и т.д.

4. Определить среднее значение всех полученных случайных чисел, а результат вывести на экран.

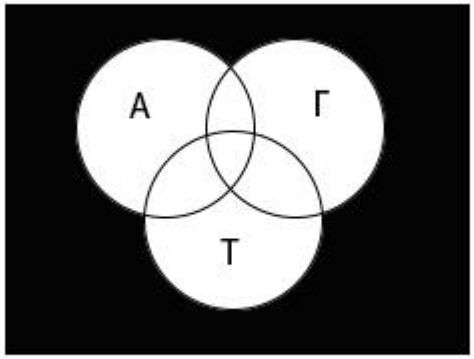
Упражнение12.1

1. Разработать свой алгоритм формирования последовательности случайных чисел с равномерным распределением в диапазоне: 0<=x<1.0 и написать метод его вычисления.
2. Написать программу, в которой с клавиатуры задается количество случайных чисел. Для полученной последовательности найти среднее значение (математическое ожидание) и вывести его на экран.
3. Подсчитать сколько чисел попадает в пять интервалов: A:0-0.2; B:0.2-0.4; C:0.4-0.6; D:0.6-0.8; E:0.8-1.0. Вывести на экран сколько чисел попадает диапазоны A, B, C, D, E.

Упражнение12*

Задать область на плоскости в диапазоне (0, x), (0, y). Разместить в этой области N окружностей случайным образом. Каждая окружность определяется координатами и случайным радиусом R, диапазон которого задается с клавиатуры.

Определить сколько окружностей по три попарно пересекаются. Для каждой такой тройки в строку вывести их координаты и их радиусы.



$$C_n^m = \frac{n!}{m!(n-m)!}$$

Упражнение13

1. Создать **ступенчатый** 2-х мерный массив. Количество строк и столбцов в каждой строке ввести с клавиатуры.
2. Заполнить этот массив случайными числами из интервала [-100, 100].
3. Найти сумму всех чисел в массиве и их среднее значение.

Упражнение

На конференции работников народного образования с докладами желает выступить несколько министров правительства. Но поскольку министры, очень заняты, то каждый из министров назначил время, когда он хочет выступить с докладом. Естественно, министры не смогли распределить время между собой, поэтому заявленные ими времена пересекаются. Перед вами стоит задача выбрать из всех поданных заявок несколько непересекающихся, при этом максимизировать число выбранных заявок.

Формат входных данных

Первая строка входных данных содержит целое число n, 0<n<=10000. Далее идет n строк, каждая из которых содержит два целых числа s_i и t_i, 0<=s_i<t_i<=30000. Каждая строка соответствует одной заявке с номером i (1<=i<=n), s_i является временем начала заявки, t_i – временем ее окончания.

Программа должна найти подмножество множества {1, 2, ..., n}, содержащее наибольшее число элементов, такое, что для любых двух элементов i и j из найденного подмножества верно одно из двух неравенств: t_i<=s_j или t_j<=s_i. Указанные неравенства означают, что заявки с номерами i и j не пересекаются, то есть одна из них заканчивается не позднее, чем начинается другая. При этом время окончания одной удовлетворенной заявки может совпадать со временем начала другой заявки (один министр на трибуне меняется на другого министра).

Формат выходных данных

Программа должна вывести номера удовлетворенных заявок в произвольном порядке, разделяя их пробелами. Заявки нумеруются, начиная с 1.

Пример

Входные данные

3 количество докладов

1 3

2 4

3 5 **Выходные данные:** 1 3

Входные данные

5 количество докладов

2 6

1 3

1 4

4 5

5 6 **Выходные данные:** 2 4 5

Упражнение14

1. Написать программу заполнения массива из 100 элементов случайными целыми числами от -110 до +110. Вывести на экран все **четные** числа, числа **меньше или равные нулю**, числа **кратные 5**, а также их количество в массиве.
2. Написать программу заполнения 2-х мерного массива 10X8 дробными числами из диапазона: -**20.63** … +**15.71**.

-подсчитать в полученном 2-х мерном массиве сколько отрицательных чисел и их сумму;
-Вывести на экран в виде 2-х мерной таблицы используя ‘\t’ и ‘\n’ .

Упражнение14*

1. Создать **ступенчатый** 2-х мерный массив. Количество строк и столбцов в каждой строке ввести с клавиатуры.
2. Заполнить этот массив случайными числами из интервала [-10, 10].
3. Вывести на экран все значения массива с использованием только одного цикла с использованием свойства **Length** или **foreach**. Подсчитать сумму и найти среднее значение.

Типы данных можно разделить на **типы** значений, еще называемые значимыми типами, (**value types**) и ссылочные типы (**reference types**). Важно понимать между ними различия.

Типы значений:

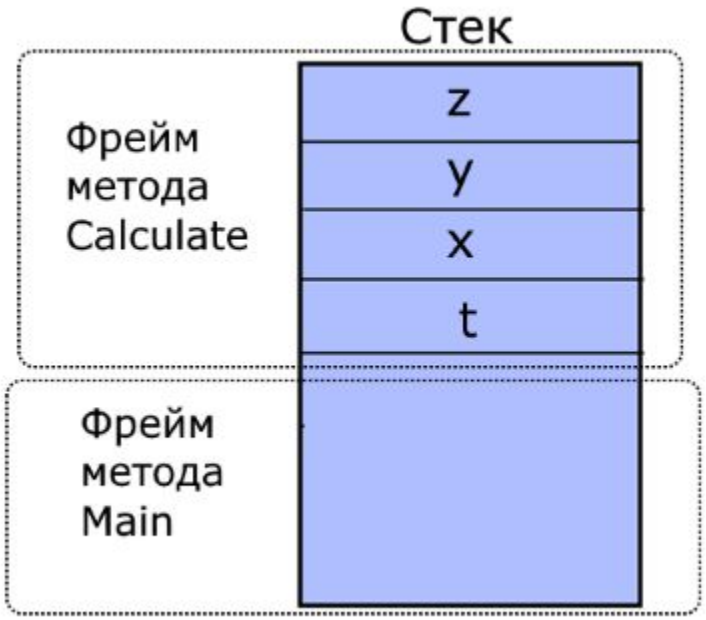
- Целочисленные типы (`byte, sbyte, short, ushort, int, uint, long, ulong`)
- Типы с плавающей запятой (`float, double`)
- Тип `decimal`
- Тип `bool`
- Тип `char`
- Перечисления `enum`
- Структуры (`struct`)

Ссылочные типы:

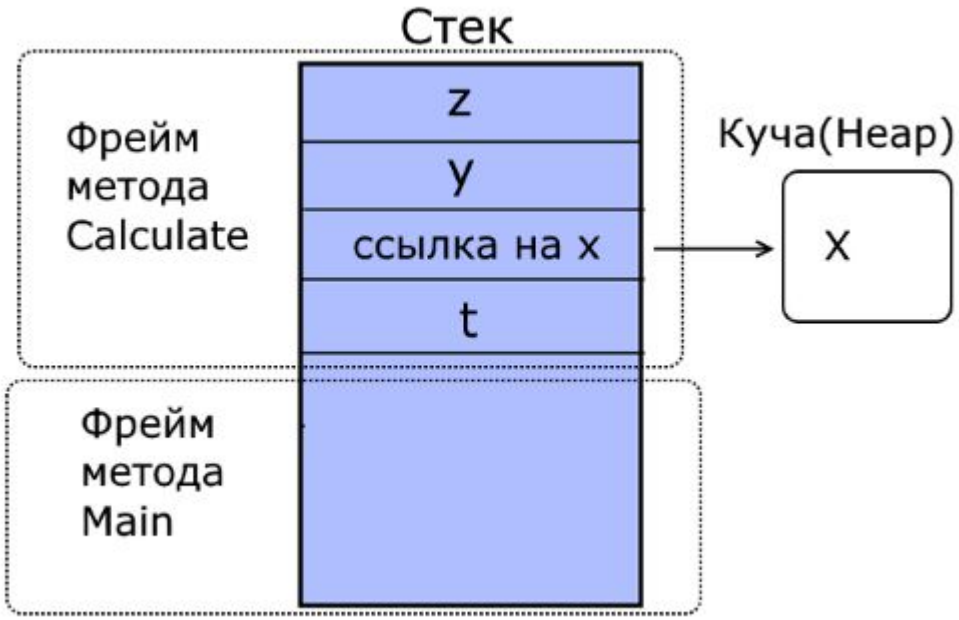
- Тип `object`
- Тип `string`
- Классы (`class`)
- Интерфейсы (`interface`)
- Делегаты (`delegate`)

Организацию памяти в .NET. Здесь память делится на два типа: **стек** и **куча (heap)**. Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу вверх: каждый новый добавляемый элемент помещается поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве.

Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для **стека** устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.



Ссылочные типы хранятся в **куче** или **хипе**, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.



Строки

С точки зрения регулярного программирования строковый тип данных string относится к числу самых важных в C#. Этот тип определяет и поддерживает символьные строки. В целом ряде других языков программирования строка представляет собой
массив символов. А в C# строки являются **объектами**.
Следовательно, тип string относится к числу ***ссылочных***.

```
char[] charray = {'t', 'e', 's', 't'};
string str = new string(charray);
```

```
string str1="error string";
for (int i=0; i < str1.Length; i++)   Console.Write(str1[i]);
```

```
string[] str = { "Это", "очень",
```

```
string orgstr = "В C# упрощается обращение со
строками.";
string substr = orgstr.Substring(5, 20); //
сформировать подстроку
```
substr содержит строку: “упрощается обращение”

```
public static string Concat (string str0, string
str1);
public static string Concat (string str0, string
str1,string str2);
public static string Concat (params string[]
values);

string result = String.Concat("Это","
метода", "сцепления " );


public string ToString("форматирующая строка");
Пример:
```

Для объединения строк также может использоваться метод **Join**.

```
string s5 = "apple";
string s6 = "a day";
string s7 = "keeps";
string s8 = "a doctor";
string s9 = "away";
string[] values = new string[] { s5, s6, s7, s8, s9 };

String s10 = String.Join(" ", values);
// результат: строка “apple a day keeps a doctor away”
```

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

```
string str1,str2;
strLow = str1.ToLower(CultureInfo.CurrentCulture);
strUp =   str2.ToUpper(CultureInfo.CurrentCulture);
```

результат = **string.Compare(*str1*, *str2, способ);**

где *способ* обозначает конкретный подход к сравнению символьных строк.

Для сравнения символьных строк с учетом культурной среды (т.е. языковых и региональных стандартов) применяется способ **StringComparison.CurrentCulture**.

Если требуется сравнить строки только на основании значений их символов **StringComparison.Ordinal**

Для сравнения строк без учета регистра: **StringComparison.CurrentCultureIgnoreCase** или **StringComparison.OrdinalIgnoreCase**.

```
if(str1 == str2)        Console.WriteLine("str1 равно str2");
else                    Console.WriteLine("str1 не равно str2");
```

Метод	Описание
static int **Compare**(string strA, string strB, StringComparison comparisonType)	Возвращает отрицательное значение, если строка strA меньше строки strB; положительное значение, если строка strA больше строки strB; и нуль, если сравниваемые строки равны. Способ сравнения определяется аргументом comparisonType
bool **Equals**(string value, StringComparison comparisonType)	> Возвращает логическое значение true, если вызывающая строка имеет такое же значение, как и у аргумента value. Способ сравнения определяется аргументом comparisonType
int **IndexOf**(char value)	> Осуществляет поиск в вызывающей строке первого вхождения символа, определяемого аргументом value. Применяется порядковый способ поиска.
int **IndexOf**(string value, StringComparison comparisonType)	Возвращает индекс первого совпадения с искомым символом или -1, если он не обнаружен > Осуществляет поиск в вызывающей строке первого вхождения подстроки, определяемой аргументом value. Возвращает индекс первого совпадения с искомой подстрокой или -1, если она не обнаружена. Способ поиска определяется аргументом

Метод	Описание
int **LastIndexOf**(char value)	Осуществляет поиск в вызывающей строке последнего вхождения символа, определяемого аргументом value.
int **LastIndexOf**(string value, StringComparison comparisonType)	Применяется порядковый способ поиска. Возвращает индекс последнего совпадени Осуществляет поиск в вызывающей строке последнего вхождения подстроки, определяемой аргументом value. Возвращает индекс последнего совпадения с искомой подстрокой или -1, если она не обнаружена. Способ поиска определяется аргументом ComparisonType
string **ToLower**(CultureInfo. CurrentCulture culture)	Возвращает вариант вызывающей строки в нижнем регистре. Способ преобразования определяется аргументом Culture
string **ToUpper**(CultureInfo. CurrentCulture culture)	Возвращает вариант вызывающей строки в верхнем регистре. Способ преобразования определяется аргументом culture

Метод	Описание
public string **Insert** (int startIndex, string value);	**value** String Строка, которую требуется вставить. Возвраты String Новая строка, эквивалентная данному экземпляру, но с тем отличием, что в положение `value` помещено значение `startIndex`.
public string **Remove** (int startIndex);	Отсчитываемая от нуля позиция, с которой начинается удаление знаков. Возвраты String Новая строка, эквивалентная данной строке за минусом удаленных знаков.
public string **Replace**(char oldChar, char newChar) public string **Replace**(string oldValue, string newValue)	Замены части строки служит метод Replace(). string s; s=s.Replace(" ","_");

Метод	Описание
public bool **Contains** (string *value*)	Возвращает логическое значение true, если вызывающая строка содержит подстроку *value.* Если же подстрока *value* не обнаружена, возвращается логическое значение false
public bool **EndsWith** (string *value*)	Возвращает логическое значение true, если вызывающая строка оканчивается подстрокой *value.* В противном случае возвращает логическое значение false
public bool **EndsWith** (string *value,* StringComparison *comparisonType*)	Возвращает логическое значение true, если вызывающая строка оканчивается подстрокой *value.* В противном случае возвращает логическое значение false. Параметр *comparisonType* определяет конкретный способ поиска
public bool **EndsWith** (string *value,* bool *ignoreCase,* CultureInfo *culture*)	Возвращает логическое значение true, если вызывающая строка оканчивается подстрокой *value,* иначе возвращает логическое значение false. Если параметр *ignoreCase* принимает логическое значение true, то при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. Поиск

Разделение и соединение строк

К основным операциям обработки строк относятся разделение и соединение. При *Разделении* строка разбивается на составные части, а при соединении строка составляется из отдельных частей. Для разделения строк в классе `String` определен метод **`Split`**`()`, а для соединения — метод **`Join`**`()`.

public string[ ] Split(params char[ ] *separator*)

public string[ ] Split(params char[ ] *separator,* int *count*)

В первой форме метода Split() вызывающая строка разделяется на составные части. В итоге возвращается массив, содержащий подстроки, полученные из вызывающей строки. Символы, ограничивающие эти подстроки, передаются в массиве *separator.* Если массив *separator* пуст или ссылается на пустую строку, то в качестве разделителя подстрок используется пробел. А во второй форме данного метода возвращается количество подстрок, определяемых параметром *count.*

```
string str = "Один на суше, другой на
море.";
char[] seps = {' ', '.', ','};
// Разделить строку на части.
string[] parts = str.Split(seps);
Console.WriteLine("Результат разделения
строки: ");
for(int i=0; i < parts.Length; i++)
Console.WriteLine(parts[i]);

// А теперь соединить части строки.
string whole = String.Join (" | ",
parts);
Console.WriteLine("Результат соединения
строки: ");
Console.WriteLine(whole);
```

Пример программы, где строки, содержащие такие бинарные математические операции, как 10 + 5, преобразуются в лексемы, а затем эти операции выполняются и выводится конечный результат.

```
string[] input = {"100 + 19","100 / 3,3","-3 * 9","100 - 87"};
char[] seps = {' '};
for(int i=0; i < input.Length; i++) {// разделить строку на части
string[] parts = input[i].Split (seps);
Console.Write("Команда: ");
for(int j=0; j < parts.Length; j++)
Console.Write(parts[j] + " ");
Console.Write(", результат: ");
double n = Double.Parse(parts[0]);      double n2 = Double.Parse(parts[2]);
switch(parts[1]) {
case "+": Console.WriteLine(n + n2);
break;
case “-”: Console.WriteLine(n - n2);
break;
case "*": Console.WriteLine(n * n2);
```

Упражнение15

1. Объявить строковую переменную **Str** содержащую строку "*Самый простой способ построить символьную строку. Самый легкий путь.*" и вывести ее на экран.
2. Написать процедуру ввода с клавиатуры произвольной строки.
3. Вывести на экран сколько раз, веденная строка входит в **Str.**
4. Найти первую и последнюю позицию вхождения этой строки в **Str**.

Упражнение16

Написать простую программу «Калькулятор командной строки» для целых чисел.

1. С клавиатуры вводится строка, например:

12+56= или 12+56-10=

Программа должна распознать числа и операцию, которую нужно выполнить, сделать расчеты.

2. В результате на экране должна быть запись:

12 + 56 = 68 или 12+56-10=58

3. Калькулятор должен выполнять все

Упражнение16*

1. Задан текст (строка):

«Я пошел к своему врачу. Он мой старый приятель; когда мне почудится, что я не здоров, он щупает у меня пульс, смотрит на мой язык, разговаривает со мной о погоде – и все это бесплатно; я подумал, что теперь моя очередь оказать ему услугу. «Главное для врача – практика», – решил я. Вот он ее и получит. В моем лице он получит такую практику, какой ему не получить от тысячи семисот каких-нибудь заурядных пациентов, у которых не наберется и двух болезней на брата.»

2. Написать программу, которая определяет следующие параметры текста:

- Сколько в тексте слов не считая предлогов и союзов. - - Сколько знаков препинания.
- Сколько предложений.
- Сколько гласных.

3. Вывести на экран словарь слов из этого текста и сколько раз они встречаются в нем.

В C# допускается, чтобы метод вызывал самого себя. Этот процесс называется *рекурсией*, а метод, вызывающий самого себя, — *рекурсивным*. Вообще, рекурсия представляет собой процесс, в ходе которого нечто определяет самое себя. В этом отношении она чем-то напоминает циклическое определение.

Рекурсивный метод отличается главным образом тем, что он содержит оператор, в котором этот метод вызывает самого себя. Рекурсия является эффективным механизмом управления программой. Классическим примером рекурсии служит вычисление факториала

$$e = \sum_{x=1}^{n} (1 + \frac{1}{x})^x$$

```
class F {public static double FN(double y)
        {    double z;
            if (y == 1) return (1 + 1 / y);
            z = FN(y - 1) + Math.Pow((1 + 1 / y), y);
            return z;
        }    }
//Рекурсия
Console.Write("e="+ F.FN(4));

//Итерация
 double s=0,x = 0;
for(int i = 1; i < 5; i++) { x = i;s+= Math.Pow((1 + 1 / x), x); }
 Console.Write("e="+s);
```

Упражнение17

1. Написать рекурсивную программу вычисления факториала n!=1*2*3*...n.
2. Написать итерационную программу вычисления факториала.

Число **n** задается с клавиатуры.

Передача объектов методам по ссылке

```
class MyClass {                    int alpha, beta;
 public MyClass (int i, int j) {  alpha = i;
beta = j;    }
    public bool SameAs (MyClass ob) {
     if((ob.alpha == alpha) & (ob.beta == beta)) return true;
     else return false;   }
public void Copy(MyClass ob) {alpha = ob.alpha;  beta = ob.beta;  }
public void Show() {Console.WriteLine("alpha: {0}, beta: {1}", alpha, beta);}}
//-------------------------------------------------------------------------

MyClass ob1 = new MyClass(4, 5);
MyClass ob2 = new MyClass(6, 7);
Console.Write("ob1: ");ob1.Show();     Console.Write("ob2: ");   ob2.Show();
if (ob1.SameAs(ob1)) Console.Write("ob1 и ob2 имеют одинаковые
значения.");
else     Console.Write("ob1 и ob2 имеют разные значения.");
ob1.Copy(ob2);
if (ob1.SameAs(ob2)) Console.Write("ob1 и ob2 имеют одинаковые
```

Использование **ref**

```
class RefTest {// Этот метод изменяет свой аргумент.
public void Sqr ( ref int i) {i = i * i;}
                    }
……….
….
static void Main() {
RefTest ob = new RefTest();
int a = 10;
 Console.WriteLine("a до вызова: " + a);
ob.Sqr(ref a);
Console.WriteLine("a после вызова: " + a);//здесь a
другое!!
```

Использование **out**

```
class Decompose {
//Разделить числовое значение на целую и дробную части
public int GetParts (double n,  out  double frac) {
int whole;  whole = (int) n;
frac = n - whole;
//передать дробную часть числа через параметр frac
return whole;  //возвратить целую часть числа } }
……….
……….
static void Main() {
Decompose ob = new Decompose();
int i; double f;
i = ob.GetParts (10.125, out f);
Console.WriteLine("Целая часть числа" + i);
 Console.WriteLine("Дроб часть числа " + f);}}
```

Возврат массива из метода

```
class Factor {
public int[] FindFactors  (int num, out int numfactors)
{
int[] facts = new int[80]; // размер массива 80 выбран
произвольно
int i, j;
// Найти множители и поместить их в массив facts.
for(i=2, j=0; i < num/2 + 1; i++) if( (num%i)==0 ) {
facts[j] = i; j++; }
numfactors = j;        return facts;    }
}
```

……

```
Factor f = new Factor();
int numfastors;
int[] factors;
factors = f.FindFactors(1000, out numfactors);
Console.WriteLine("Множители числа 1000: ");
for(int i=0; i < numfactors; i++)
Console.Write(factors[i] + " ");                Console.WriteLine();
```

Использование переменного числа аргументов params

При создании метода обычно заранее известно число аргументов, которые будут переданы ему, но так бывает не всегда. Иногда возникает потребность создать метод, которому можно было бы передать произвольное число аргументов.

Допустим, что требуется метод, обнаруживающий наименьшее среди ряда значений. Такому методу можно было бы передать не менее двух, трех, четырех или еще больше значений. Но в любом случае метод должен возвратить наименьшее из этих значений. Такой метод нельзя создать, используя обычные параметры. Вместо этого придется воспользоваться специальным типом параметра, обозначающим произвольное число параметров.

params

```
class Min {
public int MinVal( params int[] nums) {
int m;
if(nums.Length == 0) {Console.WriteLine("Ошибка: нет
аргументов.");
return 0;}
m = nums[0];
for(int i=1; i < nums.Length; i++)
if(nums[i] < m) m = nums[i];                    return m;}
. . . . . .
. . . . . .
Min ob = new Min();
int min;
int a = 10, b = 20;
min = ob.MinVal(a, b);

min = ob.MinVal(a, b, -1);

min = ob.MinVal(18, 23, 3, 14, 25);

int[] args = { 45, 67, 34, 9, 112, 8 };
min = ob.MinVal(args);
```

Упражнение18

1. Создать свой класс с методами.
2. Написать метод, который меняет значение аргумента, например, вычисляет кубический корень числа (использовать **ref**).
3. Написать метод, который возвращает целую и дробную часть любого числа (использовать **out**).
4. Написать метод с переменным числом аргументов, который возвращает максимальное число из введенных аргументов (аргументов должно быть больше двух, использовать **params**)

Возврат объектов из методов

```
class Rect {
int width;                    int height;
public Rect(int w, int h) {  width = w;   height = h;   }
public int Area() {  return width * height; }
public void Show() { Console.WriteLine(width + " " + height); }
// Метод возвращает прямоугольник со сторонами, пропорционально увеличенными на указанный коэффициент
public Rect Enlarge(int factor) {
return new Rect(width * factor, height * factor);
}        }
. . . . . .
Rect r1 = new Rect(4, 5);

r1.Show();
Console.WriteLine("Площадь прямоугольника r1: " + rl.Area());

Rect r2 = r1.Enlarge(2);

r2.Show();
Console.WriteLine("Площадь прямоугольника r2: " + r2.Area());
```

```
class MyClass {
int a, b;
public MyClass Factory (int i, int j) { MyClass t = new
MyClass();
t.a = i;   t.b = j;                    return t; //
возвратить объект    }
public void Show() {
Console.WriteLine("a и b: " + a + " " + b); }
                              }
```

……

```
MyClass ob = new MyClass();
int i, j;
// Сформировать объекты, используя фабрику класса.
for(i=0, j=10; i < 10; i++, j--) {
MyClass anotherOb = ob.Factory(i, j); // создать объект anotherOb.Show();
}
Console.WriteLine();
```

Алгоритм перебора

Имеется массив символов: [**a, b, c, d**]. Необходимо определить сколько можно построить массивов по 3 элемента из исходного массива, без повторений. Это задача определения количество вариантов выбора m = 3 из n = 4.

$$C_n^m = \frac{n!}{m!(n-m)!}$$

$$C = \frac{n!}{m!(n-m)!} = \frac{4!}{3!(4-3)!} = \frac{4}{1} = 4 \text{ варианта.}$$

a	b	c	d	Варианты		
0	1	1	1	b	c	d
1	0	1	1	a	c	d
1	1	0	1	a	b	d
1	1	1	0	a	b	c

Программа определяет все числа в диапазоне от 1 до 15, в которых число двоичных единиц ровно 3

```
byte x;
 int sum=0;
string ss;
for (x=1; x<15;x++)
{ss= Convert.Tostring (x,2);
      for(int j=0; j< ss.Length; j++){
       if (ss[j]==‘1’)sum++;              }
       if (sum==3) Console.WritLine(“x= “+x);
   sum=0;
 }
```

Упражнение18*

1. С клавиатуры ввести элементы алфавита, например: *a, b, c, d*.
2. Задать длину слов N, которые формируются из заданного алфавита.
3. Подсчитать сколько таких слов без повторений букв можно получить.
4. Вывести на экран все возможные слова, которые можем получить из заданного алфавита заданной длины. Например при N=3:

abc, bcd, cda, dab, …..

и так далее.

Перегрузка методов

В C# допускается совместное использование одного и того же имен двумя или более методами одного и того же класса, при условии, что и параметры объявляются по-разному. В этом случае говорят, что **методы перегружаются**, а сам процесс называется **перегрузкой методов**. Перегрузка методов относится к одному из способов реализации полиморфизма в C#.

```
class Overload {
public void OvlDemo() {Console.WriteLine("Без параметров"); }
// Перегрузка метода OvlDemo с одним целочисленным параметром.
public void OvlDemo( int a ) { Console.WriteLine(Один параметр:"+a);          }
// Перегрузка метода OvlDemo с двумя целочисленными параметрам
public int OvlDemo( int a, int b ) {   Console.WriteLine("Два параметра: " + a +
" + b);              return a + b;   }
```

Упражнение19

1. С консоли вводится число.
2. Написать перегружаемый метод, который возвращает целое число при вычислении формулы:

$$2*(x-4)/(x+5)$$

если аргумент **целое** число.
3. Возвращает дробное число если аргумент **дробное** число.
4. Выводит на экран сообщение, если не было введено значение с консоли.

Перегрузка конструкторов

```
class MyClass {
public int x;
public MyClass() {Console.Write("В конструкторе MyClass().");x = 0;}
public MyClass( int i ) {Console.Write("В конструкторе MyClass(int)."); x = i;}
public MyClass(double d) {Console.Write("В конструкторе MyClass(double).");x = (int) d;}
public MyClass(int i, int j) {Console.Write("В конструкторе MyClass(int, int).");x = i * j;}}

//.....................................................................
MyClass t1 = new MyClass();
MyClass t2 = new MyClass(88);
MyClass t3 = new MyClass(17.23);
MyClass t4 = new MyClass(2, 4);
Console.WriteLine("t1.x: " + t1.x);
```

Упражнение19*

1. Написать программу с одним перегружаемым методом, который выполняет операцию с вводимыми с консоли с различными типами переменными (от 2-х до нескольких переменных).
2. Если числа целые выполняется операция перемножения переменных и выводится на экран результат.
3. Если числа дробные, то выводится на экран результат деления.
4. Если введены строковые переменные то выполняется операция сцепления строк и выводится результат на экран.

Основы перегрузки операторов (-,+,/,*…)

public static возвращаемый_тип **operator** знак_оператора (тип_параметра операнд)

{// *операции* …….. }

// *Общая форма перегрузки* ***бинарного*** *оператора*.

public static возвращаемый_тип **operator** знак_оператора (тип_параметра1 **операнд1**, тип_параметра2 **операнд2**)

{// *операции* …….. }

```
// Класс для хранения двухмерных координат.
class ThreeD {          int x, y; // двухмерные координаты
public ThreeD() { x = y= 0; }
public ThreeD(int i, int j) { x = i; y = j; }

// Перегрузить бинарный оператор +.

    public static ThreeD operator +(ThreeD op1, ThreeD op2)
    {ThreeD result = new ThreeD();
/* Сложить координаты двух точек и возвратить результат. */
    result.x = op1.x + op2.x; // Эти операторы выполняют
    result.y = op1.y + op2.y; // целочисленное сложение,
    return result;}
// Вывести координаты X, Y.
public void Show(){Console.WriteLine("X="+x + " Y= " + y );}}
//   Главная программа ----------------------------------------------
 static void Main() {
ThreeD a = new ThreeD(1, 2);   ThreeD b = new ThreeD(10, 10);
ThreeD c;
c = a + b; // сложить координаты точек a и b
Console.Write("Результат сложения a + b: ");
c.Show();
}}
```

```
// Перегрузить оператор унарного минуса.
public static ThreeD operator -  (ThreeD op)
{
ThreeD result = new ThreeD ();
result.x = -op.x;
result.y = -op.y;
result.z = -op.z;
return result;
}
```

На перегрузку операторов отношения накладывается следующее важное ограничение: они должны перегружаться попарно.

== !=

< >

<= >=

Операторы, которые **нельзя** перегружать.

&&	()	.	?
??	[]	||	=
=>	->	as	checked
default	is	new	sizeof
typeof	unchecked		

Упражнение20

1. Ввести с клавиатуры координаты 2 точек в 3-х мерном пространстве.
2. Написать программу с перегрузкой оператора – (знак минус), который должен выполнять операцию вычитания координат одной точки от другой в 3-х мерном пространстве.
3. После получения ответа программа вновь должна быть готова для ввода новых координат. Для выхода из программы нажать символ ”q”.

Упражнение20*

1. Написать программу анализа координат 3-х мерной точки в пространстве с перегрузкой операторов **true** и **false**.
2. С клавиатуры вводятся координаты точки (x, y, z). Если **все** координаты равны нулю, то вывести на экран сообщение False, иначе True.
3. Произвести перегрузку унарных операторов ++ инкремент и -- декремент для 3-х мерных координат. Если надо увеличить все координаты точки на один (x+1, y+1, z+1) или уменьшить на один (x-1, y-1, z-1), то вместо трех действий сложения, вычитания использовать перегруженные унарные операторы ++, --.
4. Вывести на экран значения координат после операций ++ и --. Если **все** координаты равны нулю, то вывести на экран сообщение False, иначе True.

Индексатор позволяет проиндексировать объекты класса, так же как массив. При определении индексатора для класса, этот класс ведет себя как виртуальный массив, поэтому можно получить доступ к экземпляру этого класса с использованием оператора доступа к массиву [ ]

Преимущество индексатора заключается, в том, что он позволяет полностью управлять доступом к массиву, избегая нежелательного доступа.

```
тип_элемента this [int индекс] {
// Аксессор для получения данных.
get { // Возврат значения, которое определяет
индекс.
}
// Аксессор для установки данных.
set {  // Установка значения, которое
определяет индекс.
```

Индексатор без базового массива

```
class PwrOfTwo {/* Доступ к логическому массиву, содержащему степени числа 2 от 0 до 15. */

public int this[int index] {// Вычислить и возвратить степень числа 2.
get {
if((index >= 0) && (index < 16)) return pwr(index); else
return -1;        }
      // Аксессор set отсутствует.
                }
int pwr (int p) {
int result = 1;
for(int i=0; i < p; i++)  result *= 2;   return result;}
}
```

```
static void Main() {
PwrOfTwo pwr = new PwrOfTwo();
Console.Write("Первые 8 степеней числа 2: ");
for(int i=0; i < 8; i++)
Console.Write(pwr[i] + " ");
}
```

```
class IndexedNames {
private string[] namelist = new string[size];
static public int size = 10;
public IndexedNames() { //конструктор класса
    for (int i = 0; i < size; i++) namelist[i] = "N. A."; }

    public string this[int index] {
get { string tmp;
if(index >= 0 && index <= size-1) { tmp = namelist[index]; }
else { tmp = ""; } return (tmp); }

set { if(index >= 0 && index <= size-1) { namelist[index] = value; } } }
//-------------------------------------------------------
static void Main() {
IndexedNames names = new IndexedNames();
names[0] = "Zara";
names[1] = "Riz";
names[2] = "Nuha";
for (int i = 0; i < IndexedNames.size; i++) { Console.WriteLine(names[i]); }
      Console.ReadKey(); } }
```

Упражнение21

1. Написать программу с **одномерным индексатором** для хранения массива строковых переменных. Размер массива ввести с консоли.
2. Предварительно заполнить массив произвольными символами, но не нулями.
3. При обращении к данным по индексу за пределами массива выводить сообщение об ошибке на экран.
4. Заполнить в программе через консоль (форму) класс несколькими строками (например, «один», «два» и т.д.).
5. Вывести на экран (форму) содержание массива.

Свойство

Одной из разновидностей члена класса является *свойство*. Как правило, свойство сочетает в себе поле с методами доступа к нему. Поле часто создается, чтобы стать доступным для пользователей объекта, но при этом желательно сохранить управление над операциями, разрешенными для этого поля, например, ограничить диапазон значений, присваиваемых данному полю. Этой цели можно, конечно, добиться и с помощью закрытой переменной, а также методов доступа к ее значению, но свойство предоставляет более совершенный и рациональный путь для достижения той же самой цели.

тип ***имя*** { //круглых скобок нет!

get {

// код **аксессора** для чтения из поля

}

set {

// код **аксессора** для записи в поле

}

где *тип* обозначает конкретный тип свойства, например int, а *имя* — присваиваемое
свойству имя. Как только свойство будет определено, любое обращение к свойству по
имени приведет к автоматическому вызову соответствующего аксессора. Кроме того,

```
class SimpProp {
int prop; // поле, управляемое свойством MyProp
public SimpProp() { prop = 0; }

/* Это свойство обеспечивает доступ к закрытой  переменной
экземпляра prop. Оно допускает присваивание только положительных
значений. */
public int MyProp {
get {      return prop;                          }
set {      if(value >= 0) prop = value;     }     }
}
```

В Main()

```
SimpProp ob = new SimpProp();
ob.MyProp = 100; // присвоить значение
// Переменной prop нельзя присвоить отрицательное
значение.
```

Автоматически реализуемые свойства имеют следующую общую форму:

***тип имя* { get; set; }**

где *тип* обозначает конкретный тип свойства, а *имя* — присваиваемое свойству имя. После обозначений аксессоров **get** и **set** сразу же следует точка с запятой, а тело у них отсутствует. Такой синтаксис предписывает компилятору создать автоматически переменную, иногда еще называемую *поддерживающим полем*, для хранения значения. Такая переменная недоступна непосредственно и не имеет имени. Но в то же время она может быть доступна через свойство. Пример:
public int UserCount { get; set; }

Если необходимо ограничить право изменения или чтения переменной используют модификатор **privet**.

```
public bool Error { get; private set; }
public int Length { get; private set; }
public string Len {private get; set; }
```

```
class MyClass {
// это свойства.
public int Count { get; set; }
public string Str { get; set; }
}
class ObjInitDemo {
static void Main() {
// Сконструировать объект типа MyClass с помощью инициализаторов объектов.
MyClass obj =
new MyClass { Count = 100, Str = "Тестирование" };
Console.WriteLine(obj.Count + " " + obj.Str);
}                    }
```

Произвольное индексирование в массиве

```
class RangeArray {// Закрытые данные.
int[ ] a; // ссылка на базовый массив
    int lowerBound; // наименьший индекс
    int upperBound; // наибольший индекс
// Автоматически реализуемое и доступное только для чтения свойство
Length.
public int Length { get; private set; }
public bool Error { get; private set; }
public RangeArray (int low, int high) {// Построить массив по
заданному размеру.
high++;      if(high <= low) {Console.Write("Неверные индексы");
high = 1; low = 0; } // создать для надежности минимально допустимый массив
     a = new int[high - low];
Length = high - low;      lowerBound = low;       upperBound = --high;
public int this [int index] {// Это индексатор для класса RangeArray.
// Это аксессор get.
get {if(ok(index)) {Error = false;
return a[index - lowerBound];} else {Error = true; return 0;}}
// Это аксессор set.
set {if(ok(index)) {a[index - lowerBound] = value;Error = false;}else Error = true;}}
// Возвратить логическое значение true, если  индекс находится в установленных
границах.
private bool ok(int index) {
```

Реализация массива

```
   static void Main() {
RangeArray ra = new RangeArray(-5, 5);
RangeArray ra2 = new RangeArray(1, 10);
RangeArray ra3 = new RangeArray(-20,-12);

// Использовать ra в качестве массива.
Console.WriteLine("Длина массива ra: " + ra.Length);
for(int i = -5; i <= 5; i++) ra[i] = i;Console.Write("Содержимое
массива ra: ");
for(int i = -5; i <= 5; i++)Console.Write(ra[i] + " "); Console.WriteLine("\n");

Console.WriteLine("Длина массива ra2: " + ra2.Length);
for(int i = 1; i <= 10; i++)ra2[i] = i; Console.Write("Содержимое массива ra2: ");
for(int i = 1; i <= 10; i++) Console.Write(ra2[i] + " ");Console.WriteLine("\n");

Console.WriteLine("Длина массива ra3: " + ra3.Length);
for(int i = -20; i <= -12; i++)ra3[i] = i; Console.Write("Содержимое массива ra3:
");
for(int i = -20; i <= -12; i++) Console.Write(ra3[i] + " "); Console.WriteLine("\n");}
```

Упражнение22

1. Написать программу с **одномерным массивом** с индексами от -10 до -3.
2. Заполнить этот массив целыми случайными числами из интервала [-100, 100].
3. Написать процедуру ввода с клавиатуры индекса и вывод значения массива с этим индексом.
4. Если введенный индекс выходит за пределы заданного диапазона, выводить сообщение “Индекс за пределами

Пример объявления двумерного индексатора

В примере объявляется двумерный индексатор, который реализует двумерный массив чисел типа double.

```
// класс, который реализует двумерный индексатор
class TwoDimIndexes
{   double[,] A; // внутренний массив элементов
   int m, n; // размерность массива
   // конструктор
   public TwoDimIndexes(int _m, int _n)
   {      m = _m;        n = _n;
      A = new double[m, n];
      for (int i = 0; i < m; i++)          for (int j = 0; j < n; j++)             A[i, j] = 0;
   }
   // двумерный индексатор
   public double this[int i, int j]
   {
      get { return A[i, j]; }
      set { A[i, j] = value; }
   }
}}  }
```

Использование класса TwoDimIndexes в некотором программном коде

```
// использование класса TwoDimIndexes
TwoDimIndexes dI = new TwoDimIndexes(3, 5);
double x;

// формирование массива с помощью индексатора
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 5; j++)
        dI[i, j] = i * 2 + j;

x = dI[1, 3]; // x = 1*2 + 3 = 5
x = dI[2, 0]; // x = 4
```

Упражнение22*

1. Написать программу с **двухмерным** массивом начальное и конечное значения индексов устанавливается с клавиатуры и могут принимать любые значение при условии что конечный индекс должен быть всегда больше начального.
2. Заполнить этот массив целыми случайными числами из интервала [-100, 100].
3. Написать процедуру ввода с клавиатуры индексов и вывод значения массива с этими индексами.
4. Если введенный индекс выходит за пределы заданного диапазона, выводить сообщение “Индекс за пределами диапазона”

Как правило, доступ к члену класса организуется посредством **объекта** этого класса, но в то же время можно создать член класса для самостоятельного применения без ссылки на конкретный экземпляр объекта. Для того чтобы создать такой член класса, достаточно указать в самом начале его объявления ключевое слово **static**.

Важно!!!

- В методе типа **`static`** должна отсутствовать ссылка **`this`**, поскольку такой метод не выполняется относительно какого-либо объекта.
- В методе типа **static** допускается непосредственный вызов только других методов типа **static**, но не метода экземпляра из того самого же класса. Дело в том, что методы экземпляра оперируют конкретными объектами, а метод типа **static** не вызывается для объекта. Следовательно, у такого метода отсутствуют объекты, которыми он мог бы оперировать.
- Аналогичные ограничения накладываются на данные типа **static**. Для метода типа **static** непосредственно доступными оказываются только другие данные типа **static**, определенные в его классе. Он, в частности, не может оперировать переменной экземпляра своего класса.

```
class Static_ {
public int Denom = 3; // обычная переменная экземпляра
public static int Val = 1024; // статическая переменная

/* Ошибка! Непосредственный доступ к нестатической переменной  из статического метода недопустим. */
static int ValDivDenom() {
return Val/Denom; // не подлежит компиляции!
}                    }
```

Статический конструктор

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы имеют следующие отличительные черты:

1. Статические конструкторы *не должны* иметь модификатор доступа и не принимают параметров
2. Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово **this** для ссылки на текущий объект класса и можно обращаться только к статическим членам класса
3. Статические конструкторы *нельзя вызвать в программе вручную*. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются)

Статические конструкторы обычно используются для инициализации статических данных, либо же выполняют действия, которые требуется выполнить только один раз

Статические классы

Класс можно объявлять как **static**. Статический класс обладает двумя основными свойствами. Во-первых, объекты статического класса создавать нельзя. Во-вторых, статический класс должен содержать только статические члены. Статический класс создается по приведенной ниже форме объявления класса, видоизмененной с помощью ключевого слова **static**.

static class имя_класса { ...

В таком классе все члены должны быть объявлены как **static**. Ведь если класс становится статическим, то это совсем не означает, что статическими становятся и все его члены.

Статические классы применяются главным образом в двух случаях.

Во-первых, статический класс требуется при создании метода расширения. Методы расширения связаны в основном с языком LINQ. Во-вторых, статический класс служит для хранения совокупности связанных друг с другом

```
static class NumericFn {
// Возвратить обратное числовое значение.
static public double Reciprocal(double num) {
return 1/num;                                    }
// Возвратить дробную часть числового значения.
static public double FracPart(double num) {
return num - (int) num;                          }
// Возвратить логическое значение true, если числовое значение переменной num окажется четным.
static public bool IsEven(double num) {
return (num % 2) == 0 ? true : false;      }
// Возвратить логическое значение true, если числовое // значение переменной num окажется нечетным.
static public bool IsOdd(double num) {
return !IsEven(num);                             }
}
```

В рамках ООП статика обладает рядом недостатков.

1. Полиморфизм.

Статические классы не поддерживают наследование, т.е. вы не можете наследоваться от интерфейса или другого класса и таким образом расширить функциональность.

2. Тестирование.

При использование статики тестирование достаточно затруднено. Нельзя оперативно подменять код, основываясь на интерфейсах. Если нужно менять, то серьёзно, переписывая значительные куски кода.

3. Единственная ответственность.

Статические методы, зачастую, применяются для служебных целей, будь то вывод в лог ошибок или сортировка массива. Поэтому иногда возникает желание запихнуть всю статическую функциональность в один класс, обозвав его MegaUtils. Так поступать не стоит, лучше создать целую кучу маленьких классов, отвечающих каждый за свою область деятельности.

Пример: написать программу со статическими членами класса, в которой с консоли вводится две произвольные строковый переменные. В результате выполнения программы на экран выводится соединение двух введенных строк

class StaticDemo {

```
// Переменная типа static.
public static int Val = 100;
// Метод типа static.
public static int ValDiv2() {
return Val/2;
}                          }
//-----------------------------------------
static void Main() {
Console.WriteLine("Исходное значение переменной " +
"StaticDemo.Val равно " + StaticDemo.Val);
StaticDemo.Val = 8;
Console.WriteLine("Текущее значение переменной" +
"StaticDemo.Val равно " + StaticDemo.Val);
Console.WriteLine("StaticDemo.ValDiv2(): " +
StaticDemo.ValDiv2());
```

Упражнение23

1. Написать статический метод, который соединяет строковые переменные. Аргументами этого метода должны быть две строковые переменные.
2. С клавиатуры вводится произвольное число строк. Если последняя строка пустая, то программа должна вывести на консоль соединение всех ранее введенных строк.

Наследование

Благодаря **наследованию** можно создать общий класс, в котором определяются характерные особенности, присущие множеству связанных элементов. От этого класса могут затем наследовать другие, более конкретные классы, добавляя в него свои индивидуальные особенности.

В языке C# класс, который наследуется, называется *базовым*, а класс, который наследует, — *производным*.

Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе, добавляя к ним свои собственные элементы.

```
class имя_производного_класса : имя_базового_класса  {
// тело класса                                        }
```

```
class B {
protected int i, j; // члены, закрытые для класса B, но доступные для класса D
public void Set(int a, int b) {
                i = a; j = b;        }
public void Show() {   Console.Write(i + " " + j);   }
}

class D : B { int k; // закрытый член
// члены i и j класса B доступны для класса D
public void Setk() {  k = i * j;    }
public void Showk() { Console.WriteLine(k);}            }
```

```
class B {
protected int i, j; // члены, закрытые для класса B, но
доступные для класса D
public void Set (int a, int b) {
i = a;              j = b;                    }
public void Show() {Console.WriteLine(i + " " + j); } }

class D : B {
int k; // закрытый член
// члены i и j класса B доступны для класса D
public void Setk() { k = i * j; }
public void Showk() { Console.WriteLine(k); } }

static void Main() {
D ob = new D();
ob.Set(2, 3); // допустимо, поскольку доступно для
класса D
ob.Show(); // допустимо, поскольку доступно для класса D
ob.Setk(); // допустимо, поскольку входит в класс D
ob.Showk(); // допустимо, поскольку входит в класс D
}
```

```
class Two {
double p_w; double p_h; // теперь это закрытая переменная
public double Width {// Свойства ширины и высоты двумерного
объекта.
    get { return p_w; }
    set { p_w = value < 0 ? -value : value; } }
public double Height {
    get { return p_h; }
    set { p_h = value < 0 ? -value : value; } }
public void Show() {Console.Write("Ширина высота равны"
+Width + " и " + Height);}}

class Tri : Two {//Класс для треугольников, производный от класса
Two.
public string Style; // тип треугольника
public double Area() {// Возвратить площадь треугольника.
    return Width * Height / 2;}
public void ShowStyle() {// Показать тип треугольника.
Console.WriteLine("Треугольник " + Style);}}
//……..……
Tri t1 = new Tri();Tri t2 = new Tri();
t1.Width = 4.0;t1.Height = 4.0;    t1.Style = "равнобедренный";
t2.Width = 8.0;t2.Height = 12.0;   t2.Style = "прямоугольный";
t1.ShowStyle();   t1.Show();       Console.Write("Площадь равна " + t1.Area());
```

В иерархии классов допускается, чтобы у базовых и производных классов были свои собственные конструкторы.

Конструктор базового класса конструирует базовую часть объекта, а конструктор производного класса — производную часть этого объекта, их конструирование должно происходить раздельно.

Если конструктор определен только в производном классе, то конструируется объект производного класса, а базовая часть объекта автоматически конструируется его конструктором, используемым по умолчанию.

Когда конструкторы определяются как в базовом, так и в производном классе, процесс построения объекта несколько усложняется.

Ключевое слово языка **: base**, которое находит двоякое применение: во-первых, для вызова конструктора базового класса; и во-вторых, для доступа к члену базового класса, скрывающегося за членом производного класса.

конструк_производ_класса **(*список_парамет*):base(*список_аргументов*) {//** тело конструктора **}**

где *список_аргументов* обозначает любые аргументы, необходимые конструктору в базовом классе.

```
class Two { double pri_width; double pri_height;
public Two (double w, double h) { Width = w; Height = h;
}
public double Width {// Свойства ширины и высоты объекта.
    get { return pri_width; }
    set { pri_width = value < 0 ? -value : value; } }
public double Height {
    get { return pri_height; }
    set { pri_height = value < 0 ? -value : value; } }
public void ShowDim() {Console.Write("Ширина высота"+Width+"и"+Height); } }

class Tri : Two {          string Style;

public Tri(string s, double w, double h) : base(w, h)  {Style = s;}

public double Area() {return Width * Height / 2;}
public void ShowStyle() {Console.Write("Треугольник"+Style);}
//.....
Tri t1 = new Tri("равнобедренный", 4.0, 4.0);
Tri t2 = new Tri("прямоугольный", 8.0, 12.0);
t1.ShowStyle();                    t1.ShowDim();
Console.Write("Площадь равна "+t1.Area());
t2.ShowStyle();                    t2.ShowDim();
Console.Write("Площадь равна " + t2.Area());
```

Упражнение24

1. Имеется базовый класс:

```
class A{
   protected int x,y,z;
     public A(){x=y=z=1;}
  public void Show(){Console.WriteLine("x={0} y={1} z={2}",x,y,z);}
}
```

Создать новый класс на базе класса **A**, в котором вычисляется функция:

$$F(x,y,z) = (x-1)/(y^2+z^{0.5})$$

и выводится результат вычисления на консоль (форму).

2. Написать программу, в которой вводится три числа: x, y, z. Вывести на консоль приглашение x= , y=, z= и задать значения с клавиатуры. Вывести значения x, y, z с помощью Show().

Вычислить функцию F(x,y,z) и вывести результат на консоль.

Перегрузка операций преобразования типов

```
int x = 50;
byte y = (byte)x; // явное преобразование от int к byte
int z = y;  // неявное преобразование от byte к int
```

```
public static implicit|explicit operator
Тип_в_который_надо_преобразовать(исходный_тип
param)
{     // логика преобразования
      return значение;                    }
```

После модификаторов **public static** идет ключевое слово **explicit** (если преобразование явное, то есть нужна операция приведения типов) или **implicit** (если преобразование неявное). Затем идет ключевое слово **operator** и далее возвращаемый тип, в который надо преобразовать объект. В скобках в качестве параметра передается объект, который надо преобразовать

```
class Counter
{    public int Seconds { get; set; }
     public static implicit operator Counter(int x)
    {        return new Counter { Seconds = x };    }
    public static explicit operator int(Counter counter)
    {        return counter.Seconds;    }
}
```

в главной программе.

```
Counter counter1 = new Counter { Seconds = 23 };
     int x = (int)counter1;
    Console.WriteLine(x);   // 23
    Counter counter2 = x;
    Console.WriteLine(counter2.Seconds);  // 23
```

Виртуальные методы и их переопределение

Виртуальным называется такой метод, который объявляется как **virtual** в базовом классе. Виртуальный
метод отличается тем, что он может быть переопред елен в одном или нескольких производных классах. Следовательно, у каждого производного
класса может быть свой вариант виртуального метода.

Метод объявляется как виртуальный в базовом классе с помощью ключевого слова **virtual**, *указываемого*
перед его именем. Когда же виртуальный метод пе реопределяется в производном классе, то для этого используется модификатор **override**. *А сам*

```
public virtual double Area() {
Console.WriteLine("Метод Area() должен
быть переопределен");
return 0.0;
}
```

Переопределить метод Area() для производного класса.

```
pulplic override double Area() {
return Width * Height / 2;
}
```

```
class Base {
// Создать виртуальный метод в базовом классе.
public virtual void Who() {
Console.WriteLine("Метод Who() в классе Base");  }    }
class Derivedl : Base {
// Переопределить метод Who() в производном классе.
public override void Who() {
Console.WriteLine("Метод Who() в классе Derived1");    }       }
//….
static void Main() {
      Base baseOb = new Base();
      Derivedl dOb1 = new Derivedl();
      Base baseRef; // ссылка на базовый класс
      baseRef = baseOb;
      baseRef.Who();

      baseRef = dOb1;
      baseRef.Who();                          }                    }
```

Переопределение метода служит основанием для воплощения одного из самых эффективных в C# принципов: **динамической диспетчеризации методов,** которая представляет собой механизм разрешения вызова во время выполнения, а не компиляции. Значение динамической диспетчеризации методов состоит в том, что именно благодаря ей в C# реализуется динамический **полиморфизм**.

Применение абстрактных классов

Иногда требуется создать базовый класс, в котором определяется *лишь* самая общая форма для всех его производных классов, а наполнение ее деталями предоставляется каждому из этих классов.

В таком классе определяется *лишь характер методов*, которые должны быть конкретно реализованы в производных классах, а не в самом базовом классе. Подобная ситуация возникает, например, в связи с невозможностью получить содержательную реализацию метода в базовом классе.

Для определения абстрактного метода служит общая форма.

abstract *тип имя*
(список_параметров);

Модификатор `abstract` может применяться только в методах экземпляра, но не в статических методах (`static`).

Абстрактными могут быть также индексаторы и свойства.

Класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный, и для этого перед его объявлением `class` указывается модификатор **`abstract`**. А поскольку реализация абстрактного класса не определяется полностью, то у него не может быть объектов. Следовательно, попытка создать объект абстрактного класса с помощью оператора **`new`** приведет к ошибке во время компиляции.

Когда производный класс наследует абстрактный класс, в нем должны быть реализованы все абстрактные методы базового класса. В противном случае производный класс должен быть также определен как `abstract`. Таким образом, атрибут `abstract` наследуется до тех пор, пока не будет достигнута полная реализация класса.

```
abstract class Person
{    public string Name { get; set; }
     public Person(string name)    {         Name = name;    }
     abstract public void Display();
}

 class Client : Person
{    public int Sum { get; set; }    // сумма на счету
     public Client(string name, int sum): base(name)
    {         Sum = sum;    }
    public override void Display()
    {Console.WriteLine($"{Name} имеет счет на сумму{Sum}");    }
}

 class Employee : Person //сотрудник
{    public string Position { get; set; } // должность
     public Employee(string name, string position): base(name)
    {         Position = position;    }
     public override void Display()
    {         Console.WriteLine($"{Position} {Name}");  }
}
```

```
Client client = new Client ("Tom", 500);
Employee employee = new Employee ("Bob", "Apple");
client.Display();
employee.Display();
```

Упражнение25

1. Создать абстрактный класс для вычисления площади 2-х мерной фигуры. В классе должен быть конструктор по умолчанию, конструктор с параметром и абстрактный метод вычисления площади фигуры. Конструктор присваивает значения высоты и основания фигуры, а также тип фигуры: «треугольник» или «прямоугольник».
2. Создать два производных класса, которые вычисляют соответственно площадь треугольника и прямоугольника.
3. Написать программу, в которой с консоли вводятся значения высоты, основания фигуры и тип фигуры. На консоль выводится тип фигуры и ее площадь затем повторно приглашаются ввести новые данные

Класс **object**

В C# предусмотрен специальный класс **object**, который неявно считается базовым классом для всех остальных классов и типов, включая и типы значений.

Иными словами, все остальные типы являются прои зводными от **object**. Это, в частности, означает, что переменная ссылочного типа **object** может ссылаться на объект любого другого типа. Кроме того, переменная типа object может ссылаться на любой массив, поскольку в C# массивы реализуются как объекты.

Формально имя **object** считается в C# еще одним обозначением класса System.Object, входящего в библиотеку классов для среды .NET Framework.

Класс object как универсальный тип данных

Если **object** является базовым классом для всех остальных типов и упаковка значений простых типов происходит автоматически, то класс object можно вполне использовать в качестве "универсального" типа данных. Пример программы, в которой сначала создается массив типа **object**, элементам которого затем присваиваются значения различных типов данных.

```
static void Main() {
object[] ga = new object[10];
for(int i=0; i < 3; i++) ga[i] = i; // Сохранить целые значения.
for(int i=3; i< 6; i++) ga[i] = (double) i / 2;// Сохранить значения типа doub
le.
// Сохранить две строки, а также значения типа bool и char.
ga[6] = "Привет";
ga[7] = true;
ga[8] = 'X';
ga[9] = "Конец";                                    }
```

Для того чтобы предотвратить наследование класса, достаточно указать ключевое слово **sealed** перед определением класса. Как и следовало ожидать, класс не допускается объявлять одновременно как **abstract** и **sealed**, поскольку сам абстрактный класс реализован не полностью и опирается в этом отношении на свои производные классы, обеспечивающие полную реализацию.

```
sealed class A {
// ...
}
// Следующий класс недопустим.
class B : A ( // ОШИБКА! Наследовать класс A нельзя
//
```

Упражнение26

1. Создать базовый класс **A** с виртуальным методо м, который выводит на экран сообщение - "Это рабо тает объект базового класса A" .

2. Создать производный класс **B** от **A**, в котором пе реопределяемый метод должен выводить сообщен ие - "Это работает объект производного класса B" .

3. В главной программе вызвать указанные два мет ода.

Интерфейс

С точки зрения синтаксиса интерфейсы подобны **абстрактным** классам. Но в интерфейсе **ни у одного** из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации. В нем указывается только, что именно следует делать, но не как это делать.

```
interface имя{
возвращаемый_тип имя_метода1(список_параметров);
возвращаемый_тип имя_метода2(список_параметров);
// ...
возвращаемый_тип имя_методаN{список_параметров);
                                }
```

Особенности интерфейсов

- в интерфейсе нельзя вписывать реализацию его элементов;
- невозможно создать экземпляр интерфейса;
- можно создать ссылку на интерфейс;
- в интерфейсе не может быть конструкторов;
- интерфейс не может содержать поля;
- в интерфейсе не может быть осуществлена перегрузка операторов;
- все методы интерфейса по умолчанию объявлены как public.
- В интерфейсах можно указывать: методы; свойства; индексаторы; события.

При использовании интерфейсов **в классах-наследниках**:

- запрещено изменять модификатор доступа для метода при его реализации;
- невозможно объявить методы интерфейса как virtual;
- запрещено объявлять методы интерфейса с ключевым словом static (как статические).

```
public interface ISeries {
int GetNext(); // возвратить следующее по порядку
число
void Reset(); // перезапустить
void SetStart(int x); // задать начальное
значение }
```

```
class ByTwos : ISeries {
int start;        int val;
public ByTwos() { start = 0; val = 0; }
public int GetNext() {val += 2; return val; }
public void Reset() { val = start; }
public void SetStart(int x) { start = x; val = start; }}
```

В классах, реализующих интерфейсы, разрешается и часто практикуется определять их собственные дополнительные члены.

```
public interface ISeries {
int Next {
get; // возвратить следующее по порядку число
set; // установить следующее число}   }
                // Реализовать интерфейс ISeries.
class ByTwos : ISeries {
int val;
public ByTwos() {val = 0;     }
public int Next {// Получить или установить значение.
       get {    val += 2;   return val;     }
    set {val = value;     }}}

             // применение интерфейсного свойства.
ByTwos ob = new ByTwos();
// Получить доступ к последовательному ряду чисел с помощью свойства.
for(int i=0; i < 5; i++)
Console.Write("Следующее число равно " + ob.Next);

Console.Write("\nНачать с числа 21");
ob.Next = 21;
for(int i=0; i < 5; i++)
Console.WriteLine("Следующее число равно " + ob.Next);
```

```
// Интерфейсное свойство
тип имя{
get;
set;
}
// Интерфейсный индексатор
тип_элемента this [int индекс]{
get;
set;
}
```

```
public interface ISeries {  // Интерфейсное свойство.
int Next {
get; // возвратить следующее по порядку число
set; // установить следующее число
}
// Интерфейсный индексатор.
int this[int index] {
get; // возвратить указанное в ряду число
}                    }

                // Реализовать интерфейс ISeries.
class ByTwos : ISeries {
int val;
public ByTwos() {    val = 0;    }
// Получить или установить значение с помощью свойства.
public int Next {
get {   val += 2;                        return val;   }
set {   val = value;}   }
// Получить значение по индексу.
public int this[int index] {
get {val = 0;
for(int i=0; i < index; i++)val += 2;  return val;}}}
```

```
// В главной программе
ByTwos ob = new ByTwos();
// Получить доступ к последовательному ряду чисел с помощью
свойства.
for(int i=0; i < 5; i++)
Console.Write("Следующее число равно " + ob.Next);

Console.Write("\nНачать с числа 21");
ob.Next = 21;
for (int i=0; i < 5; i++)
Console.Write("Следующее число равно " + ob.Next);

Console.Write("\nСбросить в 0");
ob.Next = 0;
// Получить доступ к последовательному ряду чисел с помощью
индексатора
for(int i=0; i < 5; i++)
Console.WriteLine("Следующее число равно " + ob[i]);
```

Пример наследования интерфейсов.

```
public interface IA {
void Meth1();
void Meth2();
}
// В базовый интерфейс включены методы Meth1()
и Meth2().
//   в производный интерфейс добавлен еще один
метод — Meth3().
public interface IB : IA {
void Meth3();
}
```

```
public interface IMyInterface  {
int MyGetInt(); // метод, возвращающий число типа int
double MyGetPi(); // метод, возвращающий число Pi
int MySquare(int x); // метод, возвращающий x в квадрате
double MySqrt(double x); // метод, возвращающий корень квадратный из x
}
public interface IMyInterface2
{  double MySqrt2(double x); // корень квадратный из x  }

public class MyClass : IMyInterface, IMyInterface2
{  // методы из интерфейса MyInterface
public int MyGetInt()  {  return 25;  }
public double MyGetPi()  {  return Math.PI;  }
public int MySquare(int x)  {  return (int)(x * x);  }
public double MySqrt(double x)  {  return (double)Math.Sqrt(x);  }
// метод из интерфейса MyInterface2
public double MySqrt2(double x)  {  return (double)Math.Sqrt(x);  }  }
```

Выбор между интерфейсом и абстрактным классом

Одна из самых больших трудностей программирования на C# состоит в правильном выборе между интерфейсом и абстрактным классом в тех случаях, когда требуется описать функциональные возможности, но не реализацию.

В подобных случаях рекомендуется придерживаться следующего общего правила: если какое-то понятие можно описать с точки зрения функционального назначения, не уточняя конкретные детали реализации, то следует использовать интерфейс. А если требуются некоторые детали реализации, то данное понятие следует представить абстрактным классом.

Упражнение27

1. Создать класс на базе интерфейса вида:

```
public interface ISeries {
int GetNext();
void Reset();
void SetStart(int x);}
```

В этом классе написать метод, при обращении к которому возвращалось число уменьшенное на 10. Значение начального числа по умолчанию должно быть равно единице.

Написать метод, который устанавливает начальное значение числа. Это число вводится с консоли.

2. Написать программу с циклом вызова метода возвращающего каждый раз число уменьшенное на 10. Количество циклов задается с консоли. В программе должен вызываться метод установки начального значения первого числа.

Упражнение27*

1. Создать два интерфейса, в первом объявить 4 абстрактных метода для арифметических операций, во втором 2 метода возведение в степень целых чисел и возведение в степень дробных чисел.
2. В производных классах переменные объявить через свойства, а задавать значения переменных через конструктор.
3. Написать программу, где переменные и показатели степени вводятся с консоли и на экран выводятся результат арифметических операция и операций по возведению в степень.

Структуры

Классы относятся к ссылочным типам данных. Это означает, что объекты конкретного класса доступны по ссылке, в отличие от значений простых типов, доступных непосредственно.

Однако, прямой доступ к объектам как к значениям простых типов оказывается полезно иметь, например, ради повышения эффективности программы. Ведь каждый доступ к объектам (даже самым мелким) по ссылке связан с дополнительными издержками на расход вычислительных ресурсов и оперативной памяти. Для разрешения подобных затруднений в C# предусмотрена *структура,* которая подобна классу, но относится к типу значения, а не к ссылочному типу данных.

Структуры объявляются с помощью ключевого слова struct и с точки зрения синтаксиса подобны классам.

```
struct имя : интерфейсы {
// объявления членов
}
```

где *имя* обозначает конкретное имя структуры.

Назначение структур

Назначение структур в повышении эффективности и производительности программ. Структуры относятся к **типам значений**, и поэтому ими можно оперировать непосредственно, а не по ссылке. Следовательно, для работы со структурой вообще не требуется переменная ссылочного типа, а это означает в ряде случаев существенную экономию оперативной памяти. Более того, работа со структурой не приводит к ухудшению производительности, столь характерному для обращения к объекту класса. Ведь доступ к структуре осуществляется непосредственно, а к объектам — по ссылке, поскольку классы относятся к данным ссылочного типа. Косвенный характер доступа к объектам подразумевает дополнительные издержки вычислительных ресурсов на каждый такой доступ, тогда как обращение к структурам не влечет за собой подобные издержки.

Если нужно **просто сохранить группу связанных вместе данных, не требующих наследования и обращения по ссылке**, то с точки зрения производительности для них лучше выбрать **структуру**.

```
struct Book {
public string Author;
public string Title;
public int Copyright;
public Book (string a, string t, int c) { Author = a;
Title = t;   Copyright = c; }
}
```

Применение struct

```
Book book1 = new Book("Герберт Шилдт", "Справочник по C#", 2010); // вызов явно заданного конструктора
Book book2 = new Book(); // вызов конструктора по умолчанию
Book book3; // конструктор не вызывается

Console.WriteLine(book1.Author + ", " + book1.Title + ", (c) " + book1.Copyright);

if(book2.Title == null)
// А теперь ввести информацию в структуру book2.
book2.Title = "О дивный новый мир"; book2.Author = "ОХаксли"; book2.Copyright = 1932;
Console.WriteLine(book2.Author + ", " + book2.Title + ", (c) " + book2.Copyright);
?? Console.WriteLine(book3.Title); // неверно, этот член структуры  нужно сначала инициализировать
book3.Title = "Красный шторм";
```

```
// структура, описывающая точку на координатной плоскости
struct Point { int x, y;
public void SetXY(int nx, int ny) { x = nx; y = ny; }
 // свойство X
public int X { get { return x; } set { x = value; } }
 // свойство Y
public int Y { get { return y; } set { y = value; } } }
```

Объявление массива MP и его использование в программном коде некоторого обработчика события (приложение типа Windows Forms).

```
// одномерный массив структур типа Point
Point[] MP; // объявление переменной типа "массив Point"
// выделить память для 10 структур типа Point
MP = new Point[10];
// Обнулить значение всех точек массива MP
for (int i = 0; i < 10; i++)
    MP[i].SetXY(0, 0); // использовать метод SetXY структуры
// Записать некоторые значения в массив точек Point
for (int i = 0; i < 10; i++)
{    // использовать свойства X и Y
    MP[i].X = i * 2;        MP[i].Y = i + 1;    }
```

Упражнение28

1. Объявить структуру, которая содержит информацию о книге: автор, название, год издания.
2. В главной программе создать массив структур из трех элементов. В первом элементе массива при инициализации внести данные, во втором использовать конструктор по умолчанию, третий не инициализировать.
3. Во второй и третий элементы массива структур внести с консоли данные о книгах.
4. Вывести на экран на отдельной строчке содержимое всех элементов массива.

Перечисления

Перечисление представляет собой множество именованных целочисленных констант. Перечислимый тип данных объявляется с помощью ключевого слова **enum**. Общая форма объявления перечисления:

```
enum имя {список_перечисления};
```

где *имя* — это имя типа перечисления, а *список_перечисления* — список идентификаторов, разделяемый запятыми.
Перечисление `Apple` различных сортов яблок:

```
enum Apple { Jonathan, GoldenDel, RedDel, Winesap,
Cortland, McIntosh };
```

Каждая символически обозначаемая константа в перечислении имеет целое значение. Однако, неявные преобразования перечислимого типа во встроенные целочисленные типы и обратно в C# не определены, а значит, в подобных случаях требуется явное приведение типов. Поскольку перечисления обозначают целые значения, то их можно, например, использовать для управления оператором выбора **switch** или же оператором цикла **for**.

Перечисления очень полезны, когда в программе требуется одна или несколько специальных символически обозначаемых констант.

Допустим, что требуется написать программу для управления лентой конвейера на фабрике. Для этой цели можно создать метод `Conveyor()`, принимающий в качестве параметров следующие команды: "старт", "стоп", "вперед" и "назад". Вместо того чтобы передавать методу Conveyor() целые значения, например, 1 — в качестве команды "старт", 2 — в качестве команды "стоп" и так далее, что чревато ошибками, можно создать перечисление, чтобы присвоить этим значениям содержательные символические обозначения.

```
class EnumDemo {
enum Apple { Jonathan, GoldenDel, RedDel, Winesap,Cortland};

static void Main() {
string[] color = {"красный","желтый","красный","красный","
красный"};
Apple i; // объявить переменную перечислимого типа Использовать
переменную i для циклического обращения к членам перечисления.
for(i = Apple.Jonathan; i <= Apple.Cortland; i++)
Console.Write(i + " имеет значение " + (int)i);

// Использовать перечисление для индексирования массива.
for(i = Apple.Jonathan; i <= Apple.Cortland; i++)
Console.Write("Цвет сорта " + i + " — " +color[ (int) i]);
}}
```

Упражнение29

1. Создать класс в котором объявить перечисление включающее список сортов.
2. В этом классе написать метод, который выводит числовое значение эквивалентное каждому члену перечисления.
3. Объявить массив строк, содержащий набор цветов, присущие каждому сорту. Сопоставить сорта с их цветами и вывести на консоль.
4. В главной программе вызвать разработанный метод.

```
class EnumDemo
{    enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland, McIntosh };
   string[] color = { "красный", "желтый", "красный", "красный", "красный", "красновато- зеленый"};
public void fun()
    {
        Apple i; // объявить переменную перечислимого типа
                 // Использовать переменную i для циклического обращения к членам перечисления.
        for (i = Apple.Jonathan; i <= Apple.McIntosh; i++)
            Console.WriteLine(i + " имеет значение " + (int)i);
        Console.WriteLine();
        // Использовать перечисление для индексирования массива.
        for (i = Apple.Jonathan; i <= Apple.McIntosh; i++)
            Console.WriteLine("Цвет сорта " + i + " – " +
            color[(int)i]);
    }
}
```

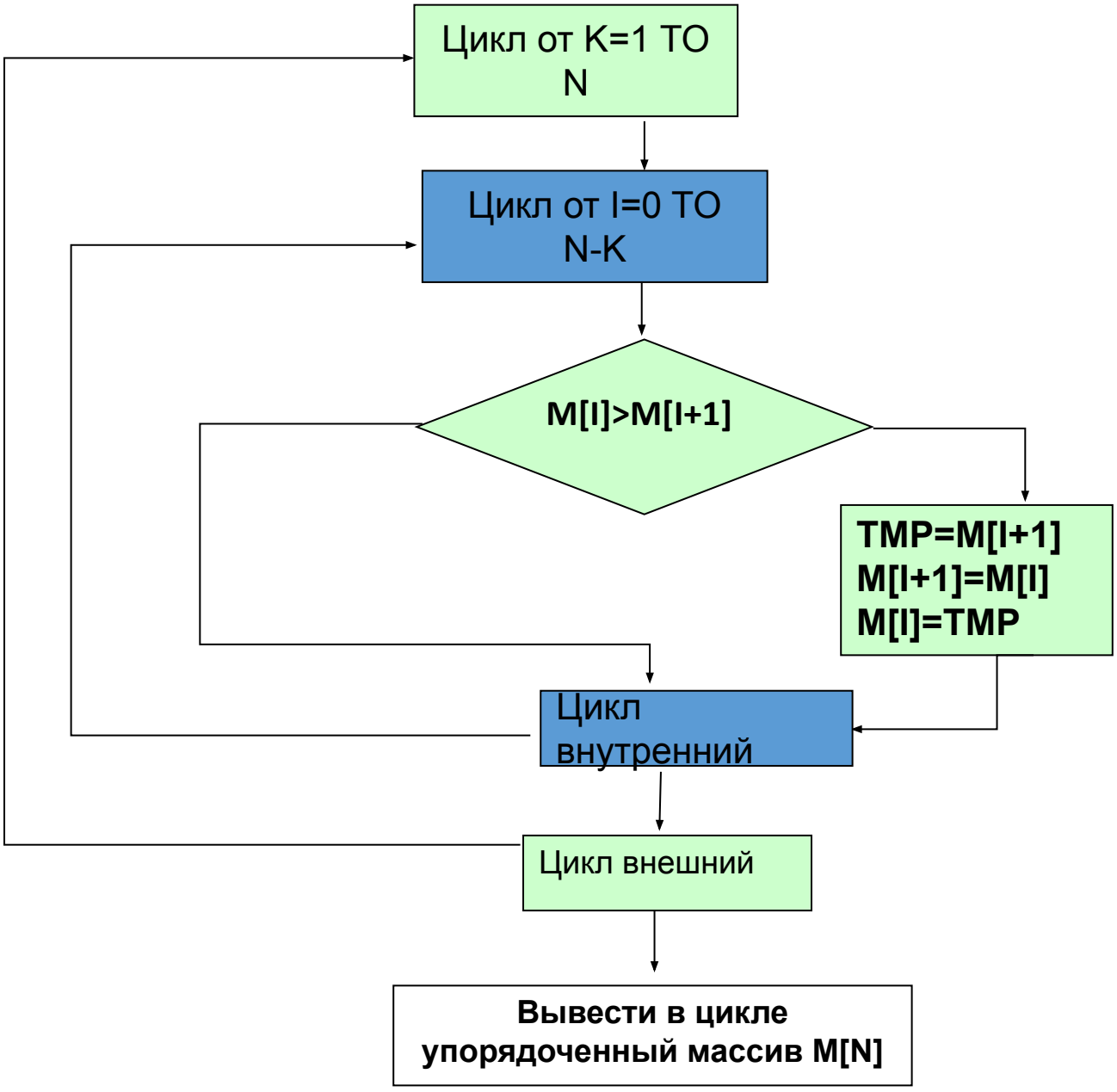
Вызов метода в главной программе:

```
EnumDemo cm = new EnumDemo();
    cm.fun();
```

Сортировка массивов

Сортировку массива по возрастанию можно провести методом «**всплывающего пузырька**», когда большие величины последовательно меняются местами с меньшими величинами и занимают позиции с большим индексом.

Исходный массив	1-й шаг	2-й шаг	3-й шаг	4-й шаг	5-й шаг	6-й шаг
7	7	3	3	3	3	1
3	3	7	7	7	1	3
8	8	8	8	1	7	7
1	1	1	1	8	8	8


Цикл от K=1 TO N
Цикл от I=0 TO N-K
M[I]>M[I+1]
TMP=M[I+1]
M[I+1]=M[I]
M[I]=TMP
Цикл внутренний
Цикл внешний
Вывести в цикле упорядоченный массив M[N]

Упражнение30

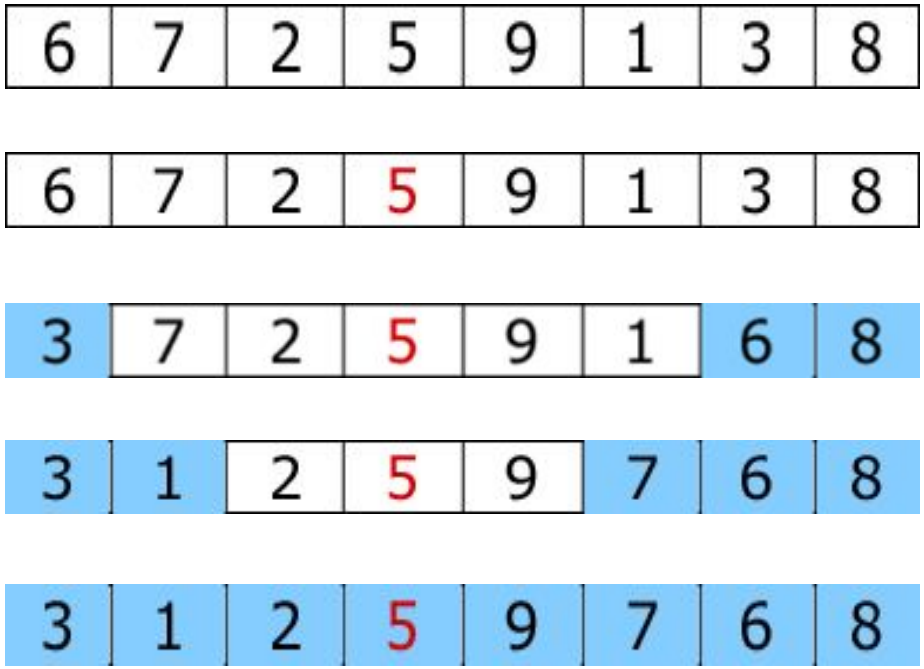
1. Объявить массив целых чисел, размер задать с клавиатуры.
2. Заполнить массив случайными целыми числам из диапазона от A до B (задать с клавиатуры A<B). Можно использовать, например, метод:

```
rand.Next(int A, int B);
```

Где rand экземпляр класса **Random.**
3. Вывести на экран, полученный массив.
4. Отсортировать массив по возрастанию методом «всплывающего пузырька», используя свой алгоритм и вывести на экран.
5. Аналогично отсортировать массив по убыванию и вывести на экран.
6. Вывести на экран элементы, которые встречаются более одного раза в массиве и указать сколько раз они встречаются.

Общая схема разбиения массива (сортировка по возрастанию):

1. вводятся указатели *first* и *last* для обозначения начального и конечного элементов последовательности, а также опорный элемент *mid*;
2. вычисляется значение опорного элемента (*first*+*last*)/2, и заноситься в переменную *mid*;
3. указатель *first* смещается с шагом в 1 элемент к концу массива до тех пор, пока *Mas*[*first*]>*mid*. А указатель *last* смещается от конца массива к его началу, пока *Mas*[*last*]<*mid*;
4. каждые два найденных элемента меняются местами;
5. пункты 3 и 4 выполняются до тех пор, пока first<last.



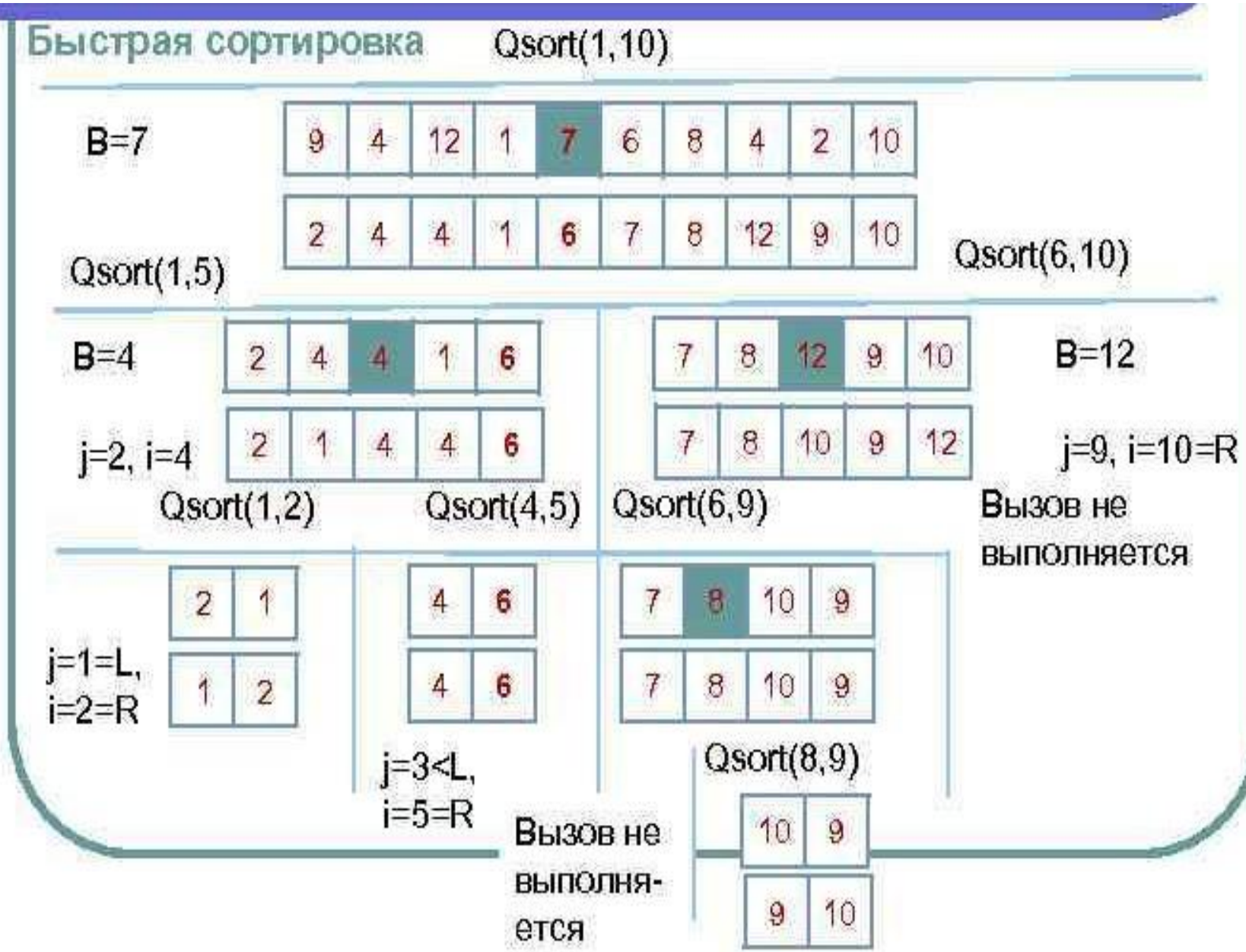
Чарльз Хоар

Рекурсивное доупорядочивание

Если в какой-то из получившихся в результате разбиения массива частей находиться больше одного элемента, то следует произвести рекурсивное упорядочивание этой части, то есть выполнить над ней операцию разбиения, описанную выше. Для проверки условия «количество элементов > 1», нужно действовать примерно по следующей схеме:

Имеется массив *Mas*[*L*..*R*], где *L* и *R* – индексы крайних элементов этого массива. По окончанию разбиения, указатели *first* и *last* оказались примерно в середине последовательности, тем самым образуя два отрезка: левый от *L* до *last* и правый от *first* до *R*. Выполнить рекурсивное упорядочивание левой части нужно в том случае, если выполняется условие *L*<*last*. Для правой части условие аналогично: *first*<*R*.

Быстрая сортировка



Упражнение31

1. Объявить и заполнить массив, как в предыдущем упражнении. Вывести на экран, полученный массив. Сделать дубликат полученного массива, чтобы его сортировать другим методом.
2. Отсортировать первый массив по возрастанию методом «всплывающего пузырька», подсчитать количество операций обмена местами чисел в массиве. Результат сортировки и количество обменов вывести на экран.
3. Отсортировать второй массив по возрастанию методом **«qsort»**, подсчитать количество операций обмена местами чисел в массиве. Результат сортировки и количество обменов вывести на экран.

qsort

```
int partition (int[] array, int start, int end)
    {
        int marker = start;
        for ( int i = start; i <= end; i++ )
        {
            if ( array[i] <= array[end] )
            {
                int temp = array[marker]; // swap
                array[marker] = array[i];
                array[i] = temp;
                marker += 1;
            }
        }
        return marker - 1;
    }

    void quicksort (int[] array, int start, int end)
    {
        if ( start >= end )
        {
            return;
        }
        int pivot = partition (array, start, end);
        quicksort (array, start, pivot-1);
        quicksort (array, pivot+1, end);
    }
```

Класс System.Exception

В C# исключения представлены в виде классов. Все классы исключений должны быть производными от встроенного в C# класса `Exception`, являющегося частью пространства имен `System`, все исключения являются подклассами класса `Exception`. К числу самых важных подклассов `Exception` относится класс **`SystemException`**. От этого класса являются производными все исключения, генерируемые исполняющей системой C#. Класс `SystemException` ничего не добавляет к классу `Exception`, а просто определяет вершину иерархии стандартных исключений.

В среде .NET Framework определено несколько встроенных исключений, являющихся производными от класса SystemException. Например, при попытке выполнить деление на нуль генерируется исключение **DivideByZeroException**.

Основу обработки исключительных ситуаций в C# составляет пара ключевых слов **try** и **catch**. Эти ключевые слова действуют совместно и не могут быть использованы порознь.

```
try {Блок кода, проверяемый на наличие ошибок. }
catch (ExcepType1 exOb) { // Обработчик исключения типа ExcepTypel. }
catch (ExcepType2 exOb) { // Обработчик исключения типа ExcepType2. }
```

где ExcepType — это тип возникающей исключительной ситуации. Когда исключение генерируется оператором **try**, оно перехватывается составляющим ему пару оператором **catch**, который затем обрабатывает это исключение. В зависимости от типа исключения выполняется и соответствующий оператор **catch**. Так, если типы генерируемого исключения и того, что указывается в операторе **catch**, совпадают, то выполняется именно этот оператор, а все остальные пропускаются.

```
static void Main() {
int[] nums = new int[4];
try {
// Сгенерировать исключение в связи с выходом индекса
за границы массива.
for(int i=0; i < 10; i++) { nums[i] = i;
Console.WriteLine("nums[{0}]: {1}", i,
nums[i]);}
. . . . .
}
catch ( IndexOutOfRangeException ) {
// Перехватить исключение.
Console.Write("Индекс вышел за границы
массива!");
}                                    }
```

- **`ArrayTypeMismatchException`** - Тип сохраняемого значения несовместим с типом массива
- **`DivideByZeroException`** - Попытка деления на нуль
- **`IndexOutOfRangeException`** - Индекс оказался за границами массива
- **`InvalidCastException`** - Неверно выполнено динамическое приведение типов
- **`OutOfMemoryException`** - Недостаточно свободной памяти для дальнейшего выполнения программы. Это исключение может быть, например, сгенерировано, если для создания объекта с помощью оператора new не хватает памяти
- **`OverflowException`** - Произошло арифметическое переполнение
- **`NullReferenceException`** - Попытка использовать пустую ссылку, т.е. ссылку, которая не указывает ни на один из объектов

Перехват всех исключений

Время от времени возникает потребность в перехвате всех исключений независимо от их типа. Для этой цели служит оператор **`catch`**, в котором тип и переменная исключения не указываются.

```
catch {// обработка исключений

}
```

С помощью такой формы создается "универсальный" обработчик всех исключений, перехватываемых в программе.

Вложение блоков try

```
int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512
};
int[] denom = ( 2, 0, 4, 4, 0, 8 );
try { // внешний блок try
      for(int i=0; i < numer.Length; i++) {
   try { // вложенный блок try
   Console.Write(numer[i]/denom[i]);}
   catch (DivideByZeroException) {
   Console.WriteLine("Делить на нуль нельзя!");}}
         }
catch (IndexOutOfRangeException) {
Console.Write("Подходящий элемент не найден.");
Console.Write("Неисправимая ошибка - программа
прервана.");}
```

*Как правило, внешний блок **try** служит для обнаружениям обработки самых серьезных ошибок, а во внутренних блоках **try** обрабатываются менее серьезные ошибки. Кроме того, внешний блок **try***

Генерирование исключений вручную

Исключение может быть сгенерировано и вручную с помощью оператора **`throw`**.

```
throw new exceptOb();
```

где в качестве *`exceptOb`* должен быть обозначен объект класса исключений, производного от класса `Exception`.

```
try {
Console.Write("До генерирования исключения.");
throw new DivideByZeroException();
}
catch (DivideByZeroException) {
Console.Write("Исключение перехвачено.");
}
```

Повторное генерирование исключений

```
int[] numer = { 4, 8, 16, 32, 64, 128,
256, 512 };
int[] denom = { 2, 0, 4, 4, 0, 8 };
for(int i=0; i<numer.Length; i++) {
   try {
Console.WriteLine(numer[i]/denom[i]);
}
   catch (DivideByZeroException) {
Console.WriteLine("Делить на нуль
нельзя!");
}
   catch (IndexOutOfRangeException) {
Console.Write("Подходящий элемент не найден.");
throw; // сгенерировать исключение
повторно
```

```
try {
//  Блок кода, предназначенный для обработки ошибок.
}
catch (ExcepType1 exOb) {
//  Обработчик исключения типа ExcepType1.
}
catch (ExcepType2 exOb) {
// Обработчик исключения типа ExcepType2.
}
finally {// Код завершения обработки исключений. }
```

*Блок **finally** будет выполняться всякий раз, когда происходит выход из блока **try/ catch**, независимо от причин, которые к этому привели. Это означает, что если блок **try** завершается нормально или по причине исключения, то последним выполняется код, определяемый в блоке **finally**.*

```
byte result;
a = 127;                                b = 127;
try {
result = unchecked ((byte)(a * b));
Console.Write("Непроверенный на переполнение
результат: " + result);
result = checked((byte)(a * b)); // эта
операция приводит к исключительной ситуации
Console.Write("Проверенный на переполнение
результат: " + result); //не подлежит
выполнению}
catch (OverflowException exc) {
Console.WriteLine(exc);
```

Упражнение32

1. Задать два массива numer и denom, первый массив в 2 раза больше второго. Заполнить массивы случайными числами первый из интервал [1,50], второй из [-3, 3].
2. Создать цикл длиной как размер массива numer.Length, в котором выполняется деление numer[i]/denom[i] и вывод результата на экран.
3. Осуществить перехват исключений **DivideByZeroException** - деление на ноль и **IndexOutOfRangeException** – выход индекса за пределы массива и вывести соответствующие сообщения на экран.
4. Объявить две переменные x и y типа byte. Присвоить им значение 125. Объявить третью переменную z типа byte. Выполнить операцию z=(byte)x*y; используя директиву **`checked`** вызвать исключительное состояние **`OverflowException`** `(переполнение). Вывести на экран` технологическое сообщение о типе и месте ошибки.

Организация системы ввода- вывода в C# на потоках

Ввод-вывод в программах на C# осуществляется посредством потоков.

Поток — это некая абстракция производства или потребления информации. С физическим устройством поток связывает система **ввода-вывода**. Все потоки действуют одинаково — даже если они связаны с разными физическими устройствами. Поэтому классы и методы ввода-вывода могут применяться к самым разным типам устройств. Например, методами вывода на консоль можно пользоваться и для вывода в файл на диске.

Байтовые и символьные потоки

На самом низком уровне ввод-вывод в C# осуществляется **байтами**. И делается это потому, что многие устройства ориентированы на операции ввода-вывода отдельными байтами. Но человеку больше свойственно общаться символами. Напомним, что в C# тип **`char`** является 16-разрядным, а тип **`byte`** — 8-разрядным.

Так, если в целях ввода- вывода используется набор символов в коде ASCII, то для преобразования типа **`char`** в тип **`byte`** достаточно отбросить старший байт значения типа **`char`**.

Классы потоков

Основные классы потоков определены в пространстве имен **System.IO**. Для того чтобы воспользоваться этими классами, как правило, достаточно добавить оператор в самом начале программы: `using System.IO;` Пространство имен **System.IO** не указывается для консольного ввода-вывода потому, что для него определен класс **Console** в пространстве имен **System**.

Класс Stream

- `void Close()` – Закрывает поток
- `void Flush()` – Выводит содержимое потока на физическое устройство
- `int ReadByte()` – Возвращает целочисленное представление следующего байта, доступного для ввода из потока. При обнаружении конца файла возвращает значение **-1**
- `int Read(byte[ ] buffer, int offset, int count)` – Делает попытку ввести ***count*** байтов в массив ***buffer***, начиная с элемента ***buffer*[*offset*]**. Возвращает количество успешно введенных байтов
- `long Seek(long offset, SeekOrigin origin)` – Устанавливает текущее положение в потоке по указанному смещению ***offset*** относительно заданного начала отсчета ***origin***. Возвращает новое положение в потоке
- `void WriteByte(byte value)` – Выводит один байт в поток вывода
- `void Write(byte[ ]buffer, int offset, int count)` – Выводит подмножество *count* байтов из массива *buffer*, начиная с элемента *buffer*[*offset*]. Возвращает количество выведенных

Некоторые из методов, генерируют исключение **`IOException`** при появлении ошибки ввода-вывода. Если же предпринимается попытка выполнить неверную операцию, например вывести данные в поток, предназначенный только для чтения, то генерируется исключение **`NotSupportedException`**.

- **`bool CanRead`** - Принимает значение true, если из потока можно ввести данные. Доступно только для чтения
- **`bool CanSeek`** - Принимает значение true, если поток поддерживает запрос текущего положения в потоке. Доступно только для чтения
- **`bool CanWrite`** - Принимает значение true, если в поток можно вывести данные. Доступно только для чтения
- **`long Length`** - Содержит длину потока. Доступно только для чтения
- **`long Position`** - Представляет текущее положение в потоке. Доступно как для чтения, так и для записи
- **`int ReadTimeout`** - Представляет продолжительность времени ожидания в операциях ввода. Доступно как для чтения, так и для записи
- **`int WriteTimeout`** - Представляет продолжительность времени ожидания в операциях вывода. Доступно как для чтения, так и для записи

- **`int Peek()`** – Получает следующий символ из потока ввода, но не удаляет его. Возвращает значение -1, если ни один из символов не доступен
- **`int Read()`** – Возвращает целочисленное представление следующего доступного символа из вызывающего потока ввода. При обнаружении конца потока возвращает значение -1
- **`int Read(char[]buffer, int index, int count)`** – Делает попытку ввести количество *`count`* символов в массив *`buffer`*, начиная с элемента *`buffer`*`[`*`index`*`]`, и возвращает количество успешно введенных символов
- **`int ReadBlock(char[]buffer, int index, int count)`** – Делает попытку ввести количество *`count`* символов в массив *`buffer`*, начиная с элемента *`buffer`*`[`*`index`*`]`, и возвращает количество успешно введенных символов
- **`string ReadLine()`** – Вводит следующую текстовую строку и возвращает ее в виде объекта типа `string`. При попытке прочитать признак конца файла возвращает пустое значение
- **`string ReadToEnd()`** – Вводит все символы, оставшиеся в потоке, и возвращает их в виде объекта типа `string`

Классы байтовых потоков

`BufferedStream` - Заключает в оболочку байтовый поток и добавляет буферизацию. Буферизация, как правило, повышает производительность

`FileStream` - Байтовый поток, предназначенный для файлового ввода- вывода

`MemoryStream` - Байтовый поток, использующий память для хранения данных

`UnmanagedMemoryStream` - Байтовый поток, использующий неуправляемую память для хранения данных

MemoryStream

```
byte[] data = new byte[255];
 // Создаём запоминающий поток
      MemoryStream mStream = new MemoryStream(data);
 // Создаём объекты чтения и записи данных в потоки и передаём mStream в качестве параметра
      StreamReader sReader = new StreamReader(mStream);
      StreamWriter sWriter = new StreamWriter(mStream);
       // Записываем данные в память
      for (int i = 0; i < 10; i++) {sWriter.WriteLine("byte [" + i + "]: " + i); }
 // Записываем в память символ для того, чтобы в цикле  при достижении этого символа,
выйти из цикла. В противном случае пробежимся по всем 255 символам.
      sWriter.WriteLine("!");
 // Данный метод нужен для того, чтобы содержимое буфера этого объекта записалось
непосредственно в массив data. Если этого не сделать, то в массиве data ничего не будет.
      sWriter.Flush();
       // Читаем данные прямо из массива data.
 foreach (char ch in data) { if (ch == '!') break; Console.Write(ch);    }
// Читаем данные из потока с помощью StreamReader Ставим указатель файла в начала
запоминающего потока MemoryStream. Если этого не сделать, то вывод будет пустым.
      mStream.Seek(0, SeekOrigin.Begin);
       string str = "";
       while ((str = sReader.ReadLine()) != null)
       {       if (str == "!") break;       Console.WriteLine(str);}
```

Открытие и закрытие файла

Для формирования байтового потока, привязанного к файлу, создается объект класса `FileStream`. В этом классе определено несколько конструкторов, самый распространенный среди них:

`FileStream(string путь, FileMode режим)`

где *путь* обозначает имя открываемого файла, включая полный путь к нему; а *режим* — порядок открытия файла. В последнем случае указывается одно из значений, определяемых в перечислении FileMode

FileMode.Append - Добавляет выводимые данные в конец файла

FileMode.Create - Создает новый выходной файл. Существующий файл с таким же именем будет разрушен

FileMode.CreateNew - Создает новый выходной файл. Файл с таким же именем не должен существовать

FileMode.Open - Открывает существующий файл

FileMode.OpenOrCreate - Открывает файл, если он существует. В противном случае создает новый файл

FileMode.Truncate - Открывает существующий файл, но сокращает его длину до нуля

Конструктор класса **FileStream** открывает файл, доступный для чтения или записи. Если же требуется ограничить доступ к файлу только для чтения или же только для записи, то в таком случае следует использовать такой конструктор.

FileStream(string *путь*, FileMode *режим*, FileAccess *доступ*)

путь обозначает имя открываемого файла, включая и полный путь к нему, а *режим* — порядок открытия файла, *доступ* обозначает конкретный способ доступа к файлу. В последнем случае указывается одно из значений, определяемых в перечислении FileAccess и приведенных ниже.

FileAccess.Read FileAccess.Write FileAccess.ReadWrite

Например, кода файл test.dat открывается только для чтения.

```
FileStream fin = new FileStream("test.dat",
FileMode.Open, FileAccess.Read);
```

! По завершении работы с файлом его следует закрыть, вызвав

```
FileStream fin;
try {
fin = new FileStream ("test", FileMode.Open);
}
catch (IOException exc) {//перехватить все исключения, связанные с вводом-выводом
Console.WriteLine(exc.Message);
// Обработать ошибку.
}
catch (Exception exc) { // перехватить любое другое исключение.
Console.WriteLine(exc.Message);// Обработать ошибку, если это возможно.}
```

В первом блоке catch из данного примера обрабатываются ошибки, возникающие в том случае, если файл не найден, путь к нему слишком длинен, каталог не существует, а также другие ошибки ввода-вывода. Во втором блоке catch, который является "универсальным" для всех остальных типов исключений, обрабатываются другие вероятные ошибки (возможно, даже путем повторного генерирования исключения). Кроме того, каждую ошибку можно проверять отдельно, уведомляя более подробно о ней и принимая конкретные меры по ее исправлению.

В классе **FileStream** определены два метода для чтения байтов из файла:

ReadByte() и **Read()**.

Так, для чтения одного байта из файла используется метод **ReadByte()**, общая форма которого `int ReadByte()`

Для чтения блока байтов из файла служит метод **Read**(), общая форма которого

выглядит так.

int Read(byte[ ] *array*, int *offset*, int *count*)

В методе Read() предпринимается попытка считать количество *count* байтов в массив *array*, начиная с элемента *array*[*offset*]. Он возвращает количество байтов, успешно считанных из файла. Если же возникает ошибка ввода-вывода, то генерируется исключение **IOException**. К числу других вероятных исключений, которые генерируются при этом, относится **NotSupportedException**. Это исключение генерируется в том случае, если чтение из файла не поддерживается в потоке.

```
static void Main(string[] args) {
int i;
FileStream fin;
if(args.Length != 1) {Console.WriteLine("Применение: ShowFile
Файл"); return;}
try {
fin = new FileStream (args[0], FileMode.Open);}
 catch (IOException exc) {Console.Write("Не удается открыть
файл");
Console.Write(exc.Message);
return;       // Файл не открывается, завершить программу
}// Читать байты до конца файла.
try {
do {
i = fin.ReadByte();
if(i != -1) Console.Write((char) i);
} while(i != -1);
} catch (IOException exc) {Console.Write("Ошибка чтения
файла");
Console.WriteLine(exc.Message);
} finally  {
fin.Close(); }}
```

Запись в файл

Для записи байта в файл служит метод `WriteByte()`.

```
void WriteByte(byte value)
```

Этот метод выполняет запись в файл байта, обозначаемого параметром *value*. Если базовый поток не открывается для вывода, то генерируется исключение **NotSupportedException**. А если поток закрыт, то генерируется исключение **ObjectDisposedException**.

Для записи в файл целого массива байтов может быть вызван метод **Write**().

```
void Write(byte[ ] array, int offset, int
count)
```

В методе **Write**() предпринимается попытка записать в файл количество *count* байтов из массива *array*, начиная с элемента *array*[*offset*]. Он возвращает количество байтов, успешно записанных в файл. Если во время записи возникает ошибка, то генерируется исключение **IOException**. А если базовый поток не открывается для вывода, то генерируется исключение **NotSupportedException**. Кроме того, может быть сгенерирован ряд других исключений.

Выводимые данные обычно буферизуются до тех пор, пока не появится возможность записать на диск сразу весь сектор. Но если данные требуется записать на физическое устройство без предварительного накопления в буфере, то для этой цели можно вызвать метод **Flush**.

```
void Flush()
```

При неудачном исходе данной операции генерируется исключение **IOException.** Если же поток закрыт, то генерируется исключение **ObjectDisposedException**.

По завершении вывода в файл следует закрыть его с помощью метода `Close()`.

```
FileStream fout = null;
try {
// Открыть выходной файл.
fout = new FileStream ("test.txt", FileMode.CreateNew);
for(char c = 'A'; c <= 'Z'; c++)
fout.WriteByte((byte) c);
} catch(IOException exc) {
Console.Write("Ошибка ввода-вывода:\n" + exc.Message);
} finally {
if (fout != null) fout.Close();
}
```

//CopyFile FIRST.DAT SECOND.DAT запуск с командной строки

```
static void Main (string[] args) {                    int i;
FileStream fin = null;
    FileStream fout = null;
if(args.Length != 2) {
Console.Write("Применение: CopyFile Откуда Куда");
                    return;}
try {
// Открыть файлы.
fin =   new FileStream (args[0], FileMode.Open);
fout = new FileStream (args[1], FileMode.Create);
// Скопировать файл.
do {
i = fin.ReadByte();
if(i != -1) fout.WriteByte((byte)i);
} while (i != –1);
} catch(IOException exc) {
Console.Write("Ошибка ввода-вывода:\n" + exc.Message);
} finally {
if(fin != null) fin.Close();
if(fout != null) fout.Close();
}}
```

Упражнение33

1. Задать массив байтов и присвоить ему значения: 239, 224,208, 224, 193, 238, 199.

- Создать в текущем каталоге файл “test33.txt”.
- Записать в байтовом потоке полученный массив.
- Найти созданный файл, открыть и прочитать содержимое с помощью блокнота, исправить исходный массив, чтобы исправить орфографическую ошибку и в существующий файл дописать исправленный массив с новой строки.

2. Создать байтовый массив не менее 1000 элементов.

- Заполнить случайными байтовыми числами.
- Записать массив в память с помощью **`MemoryStream.`**
- Прочитать из памяти записанный массив и подсчитать сумму чисел байтового потока. Вывести на экран результат.

Console

`static ConsoleKeyInfo ReadKey()`

`static ConsoleKeyInfo ReadKey(bool intercept)`

В первой форме данного метода ожидается нажатие клавиши. Когда оно происходит, метод возвращает введенный с клавиатуры символ и выводит его на экран. Во второй форме также ожидается нажатие клавиши, и затем возвращается введенный с клавиатуры символ. Но если значение параметра ***`intercept`*** равно **`true`**, то введенный символ не отображается. А если значение параметра ***`intercept`*** равно **`false`**, то введенный символ отображается.

Метод **ReadKey**() возвращает информацию о нажатии клавиши в объекте типа **ConsoleKeyInfo**, который представляет собой структуру, состоящую из свойств, доступных только для чтения.

```
char KeyChar
ConsoleKey Key
ConsoleModifiers Modifiers
```

- Свойство **KeyChar** содержит эквивалент **char** введенного с клавиатуры символа,
- свойство **Key** — значение из перечисления ConsoleKey всех клавиш на клавиатуре, а
- свойство **Modifiers** — описание одной из модифицирующих клавиш (<Alt>, <Ctrl> или <Shift>), которые были нажаты, если это действительно имело место, при формировании ввода с клавиатуры.

Эти модифицирующие клавиши представлены в перечислении **ConsoleModifiers** следующими значениями: **Control**, **Shift** и **Alt**. В свойстве Modifiers может присутствовать несколько значений нажатых модифицирующих клавиш.

```
int origWidth = Console.WindowWidth;//текущая ширина консоли
int origHeight=Console.WindowHeight;//текущая высота консоли
```

```
ConsoleKeyInfo keypress;
Console.Write("Введите несколько символов,"+"а по
окончании - <Q>.");
do {
keypress = Console.ReadKey (); //считать данные о нажатых
клавишах
Console.Write(" Вы нажали клавишу: " + keypress.KeyChar);
// Проверить нажатие модифицирующих клавиш.
if((ConsoleModifiers.Alt & keypress.Modifiers) != 0)
   Console.Write("Нажата клавиша <Alt>.");
if((ConsoleModifiers.Control & keypress.Modifiers) != 0)
   Console.Write("Нажата клавиша <Control>.");
if((ConsoleModifiers.Shift & keypress.Modifiers) != 0)
   Console.Write("Нажата клавиша <Shift>.");
} while(keypress.KeyChar != 'Q');
```

//Console.CursorVisible = false; скрыть курсор

```
ConsoleKeyInfo u;
u = Console.ReadKey(true);
```

u.Key.ToString() - возвращает одно из строковых значений при нажатии соответствующей кнопки: `UpArrow`, `DownArrow`, `LeftArrow`, `RightArrow`, `PageUp`, `PageDown`, `Home`, `End`, `Enter` и символы.

u.Modifiers.ToString() - возвращает одно из строковых значений при нажатии дополнительно соответствующей кнопки: `Shift`, `Control`, `Alt`

```
class Sample {
protected static int origRow;
protected static int origCol;
protected static void WriteAt(string s, int x, int y) {
try { Console.SetCursorPosition(origCol+x, origRow+y);
Console.Write(s); }
catch (ArgumentOutOfRangeException e) { Console.Clear();
Console.WriteLine(e.Message); }

static void Main() {
origRow = Console.CursorTop;
origCol = Console.CursorLeft;
// Draw the left side of a 5x5 rectangle, from top to bottom.
WriteAt("+", 0, 0);            WriteAt("|", 0, 1);
WriteAt("|", 0, 2);            WriteAt("|", 0, 3);
WriteAt("+", 0, 4);
// Draw the bottom side, from left to right.
WriteAt("-", 1, 4);
WriteAt("-", 2, 4);
WriteAt("-", 3, 4);
WriteAt("+", 4, 4);
                  WriteAt("All done!", 0, 6);}
}
```

Упражнение34

1. Написать программу управления перемещения курсора с помощью кнопок со стрелками, используя метод: **Console.ReadKey()**. Правая позиция границы консоли имеет значение 79, нижняя граница значение 100.
2. При нажатой пары кнопок: Alt+i движение вверх, Alt+j движение влево, Alt+k движение в право, Alt+m движение вниз должно заполнятся символом "+".
3. При нажатой кнопке **Ctrl** и стрелки должно заполнятся символом "-".

```
int Row = Console.CursorTop; //присвоить текущее положение курсора по оси Y
int Col = Console.CursorLeft;//присвоить текущее положение курсора по оси X
Console.SetCursorPosition(Col, Row);//установить новое расположение курсора
Console.SetWindowSize(100, 80);
```

Применение класса `StreamWriter`

Для создания символьного потока вывода достаточно заключить объект класса **`Stream`**, например `FileStream`, в оболочку класса **`StreamWriter`**. В классе `StreamWriter` определено несколько конструкторов.

`StreamWriter``(Stream` *`поток`*`)`

где *`поток`* обозначает имя открытого потока. Этот конструктор генерирует исключение **`ArgumentException`**, если *`поток`* не открыт для вывода, а также исключение **`ArgumentNullException`**, если *`поток`* оказывается пустым. После создания объекта класс `StreamWriter` выполняет автоматическое преобразование символов в байты.

Класс StreamWriter является производным от абстрактного класса **TextWriter**, а класс StreamReader — производным от абстрактного класса **TextReader**. Следовательно, в классах StreamReader и StreamWriter доступны методы и свойства, определенные в их базовых классах.

```
string str;
FileStream fout;
// Открыть сначала поток файлового ввода-вывода.
    try {
fout = new FileStream("test.txt", FileMode.Create);
}
    catch (IOException exc) {
Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
return;    }
// Заключить поток файлового ввода-вывода в оболочку класса
StreamWriter.
StreamWriter fstr_out = new StreamWriter (fout);
    try {
Console.WriteLine("Введите текст, а по окончании — 'стоп'.");
do {
Console.Write(": ");
str = Console.ReadLine();
if(str != "стоп") {   str = str + "\r\n"; // добавить новую
строку
fstr_out.Write(str);    }//для построчной записи в файл
fstr_out.WriteLine(str);
} while(str != "стоп");
}   catch (IOException exc) {
Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
```

Применение класса StreamReader

Для создания символьного потока ввода достаточно заключить байтовый поток в оболочку класса StreamReader. В классе StreamReader определено несколько конструкторов.

```
StreamReader(Stream поток)
```

где поток обозначает имя открытого потока. Этот конструктор генерирует исключение **ArgumentNullException**, если поток оказывается пустым, а также исключение **ArgumentException**, если поток не открыт для ввода. После своего создания объект класса StreamReader выполняет автоматическое преобразование байтов в символы. По завершении ввода из потока типа StreamReader его нужно закрыть. При этом закрывается и базовый поток.

```
string s;
try {
fin = new FileStream("test.txt",
FileMode.Open);
}
catch (IOException exc) {
Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
return;                    }
StreamReader fstr_in = new StreamReader (fin);
try {
while((s = fstr_in.ReadLine()) != null) {
Console.WriteLine(s);                    }
} catch (IOException exc) {
Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
} finally {
fstr_in.Close();
}
```

Когда метод **ReadLine()** возвращает пустую ссылку, это означает, что достигнут конец файла. Такой способ вполне работоспособен, но в классе **StreamReader** предоставляется еще одно средство для обнаружения конца потока — **EndOfStream**. Это доступное для чтения свойство имеет логическое значение **true**, когда достигается конец потока, в противном случае — логическое значение **false**. Следовательно, свойство **EndOfStream** можно использовать для отслеживания конца файла.

Другой способ организации цикла `while` для чтения из файла.

```
while ( !fstr_in.EndOfStream ) {
s = fstr_in.ReadLine();
Console.WriteLine(s);
}
```

Упражнение35

1. Написать программу записи текста вводимого с консоли в несколько строк в текстовый файл “Upr35.txt”. Для окончания ввода ввести строку “Stop”.
2. Затем после нажатия кнопки **Enter** прочитать текстовый файл “Upr35.txt” и вывести на экран содержимое.

Чтение и запись двоичных данных

Кроме возможности чтения и записи байтов или символов имеется также возможность (и ею пользуются часто) читать и записывать другие типы данных.

Например, можно создать файл, содержащий Данные типа **int**, **double** или **short**. Для чтения и записи двоичных значений встроенных в C# типов данных служат классы потоков **BinaryReader** и **BinaryWriter**.

Используя эти потоки, следует иметь в виду, что данные считываются и записываются во внутреннем двоичном формате, а не в удобочитаемой текстовой форме.

Класс BinaryWriter

Класс BinaryWriter служит оболочкой, в которую заключается байтовый поток, управляющий выводом двоичных данных.

BinaryWriter(Stream output)

где *output* обозначает поток, в который выводятся записываемые данные.

Для записи в выходной файл в качестве параметра output может быть указан объект, создаваемый средствами класса **FileStream**. Если же параметр output оказывается пустым, то генерируется исключение **ArgumentNullException**. А если поток, определяемый параметром output, не был открыт для записи данных, то генерируется исключение **ArgumentException**. По завершении вывода в поток типа BinaryWriter его нужно закрыть.

void Write(sbyte value) - Записывает значение типа sbyte со знаком
void Write(byte value) - Записывает значение типа byte без знака
void Write(byte[] buffer) - Записывает массив значений типа byte
void Write(short value) - Записывает целочисленное значение типа short (короткое целое)
void Write(ushort value) - Записывает целочисленное значение типа ushort (короткое целое без знака)
void Write(int value) - Записывает целочисленное значение типа int
void Write(uint value) - Записывает целочисленное значение типа uint (целое без знака)
void Write(long value) - Записывает целочисленное значение типа long (длинное целое)
void Write(ulong value) - Записывает целочисленное значение типа ulong (длинное целое без знака)
void Write(float value) - Записывает значение типа float (с плавающей точкой одинарной точности)
void Write(double value) - Записывает значение типа double (с плавающей точкой двойной точности)
void Write(decimal value) - Записывает значение типа decimal (с двумя десятичными разрядами после запятой)
void Write(char ch) - Записывает символ
void Write(char[] buffer) - Записывает массив символов
void Write(string value) - Записывает строковое значение типа

Класс BinaryReader

Класс BinaryReader служит оболочкой, в которую заключается байтовый поток, управляющий вводом двоичных данных.

BinaryReader(Stream input)

где input обозначает поток, из которого вводятся считываемые данные. Для чтения из входного файла в качестве параметра input может быть указан объект, создаваемый средствами класса **FileStream**. Если же поток, определяемый параметром input, не был открыт для чтения данных или оказался недоступным по иным причинам, то генерируется исключение **ArgumentException**.

При неудачном исходе операции чтения эти методы генерируют исключение **IOException**. Кроме того, в классе BinaryReader определен стандартный метод Close().

bool ReadBoolean() - Считывает значение логического типа bool
byte ReadByte() - Считывает значение типа byte
sbyte ReadSByte() - Считывает значение типа sbyte
byte[] ReadBytes(int count) - Считывает количество count байтов и возвращает их в виде массива
char ReadChar() - Считывает значение типа char
char[] ReadChars(int count) - Считывает количество count символов и возвращает их в виде массива
decimal ReadDecimal() - Считывает значение типа decimal
double ReadDouble() - Считывает значение типа double
float ReadSingle() - Считывает значение типа float
short ReadInt16() - Считывает значение типа short
int ReadInt32() - Считывает значение типа int
long ReadInt64() - Считывает значение типа long
ushort ReadUIntl6() - Считывает значение типа ushort
uint ReadUInt32() - Считывает значение типа uint
ulong ReadUInt64() - Считывает значение типа ulong
string ReadString() - Считывает значение типа string, представленное во внутреннем двоичном формате с указанием длины строки. Этот метод следует использовать для считывания строки, которая была записана средствами класса BinaryWriter
int Read() - Возвращает целочисленное представление следующего доступного символа из вызывающего потока ввода. При обнаружении конца файла возвращает значение -1
int Read(byte[] buffer, int offset, int count) - Делает попытку прочитать количество count байтов в
массив buffer, начиная с элемента buffer[offset], и возвращает количество успешно считанных байтов

```
BinaryReader dataIn; BinaryWriter dataOut;
int i = 10;  double d = 1023.56;   bool b = true;   string
str = "Это тест";
try {
dataOut = new BinaryWriter (new FileStream("testdat",
FileMode.Create));  }
catch (IOException exc) {
Console.Write("Ошибка открытия файла:\n" + exc.Message);
return;   }

try {
dataOut.Write(i);          dataOut.Write(d);
dataOut.Write(b);          dataOut.Write(12.2 * 7.4);
dataOut.Write(str);                }
catch(IOException exc) {
Console.Write("Ошибка ввода-вывода:\n" + exc.Message);
}
finally { dataOut.Close(); }
```

```
// А теперь прочитать данные из файла.
try { dataIn = new BinaryReader(new FileStream("testdata", FileMode.Open));
}
catch(IOException exc)
Console.Write("Ошибка открытия файла:\n" + exc.Message);
return; }
try {
i = dataIn.ReadInt32();  Console.Write("Чтение " + i);
d = dataIn.ReadDouble();  Console.Write("Чтение " + d);
b = dataIn.ReadBoolean () ;  Console.Write("Чтение " + b);
d = dataIn.ReadDouble();   Console.Write("Чтение " + d);
str = dataIn.ReadString();  Console.Write("Чтение " + str);      }
catch (IOException exc) {
 Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
} finally {
dataIn.Close();  }
```

Файлы с произвольным доступом

К содержимому файла может быть и произвольным. Для этого служит, в частности, метод **Seek**(), определенный в классе **FileStream**. Этот метод позволяет установить *указатель положения в файле*, или так называемый *указатель файла*, на любое место в файле. Общая форма метода **Seek**():

```
long Seek(long offset, SeekOrigin origin)
```

где *offset* обозначает новое положение указателя файла в байтах относительно заданного начала отсчета (*origin*). В качестве *origin* может быть указано одно из приведенных ниже значений, определяемых в перечислении SeekOrigin.

SeekOrigin.Begin - Поиск от начала файла

SeekOrigin.Current -Поиск от текущего положения

SeekOrigin.End - Поиск от конца файла

```
FileStream f = null;                                    char
ch;
try {
f = new FileStream("random.dat", FileMode.Create);
// Записать английский алфавит в файл.
for (int i=0; i < 26; i++)
f.WriteByte((byte)('A'+i));
// А теперь считать отдельные буквы английского
алфавита.
f.Seek(0, SeekOrigin.Begin); // найти первый байт
ch = (char) f.ReadByte();
Console.WriteLine("1-я буква: " + ch);
f.Seek(8, SeekOrigin.Begin); // найти 9-й байт
ch = (char) f.ReadByte();
Console.WriteLine("9-я буква: " + ch);
catch (IOException exc) {
Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
} finally {
if(f != null) f.Close();
}
```

Position

Несмотря на то что метод **Seek**() имеет немало преимуществ при использовании с файлами, существует другой способ установки текущего положения в файле с помощью свойства **Position**. Свойство **Position** доступно как для чтения, так и для записи. Поэтому с его помощью можно получить или же установить текущее положение в файле.

```
f = new FileStream("random.dat", FileMode.Create);
Console.WriteLine("Буквы алфавита через одну: ");
for(int i=0; i < 26; i += 2) {
f.Position = i; // найти i-й символ посредством свойства Position
ch = (char) f.ReadByte();}
```

Упражнение36

1. Написать программу, которая записывает в текстовый файл "test36.txt" строку: “Congratulations on your holiday” и выводит эту строку на экран.
2. В этом файле не читая строки, используя Seek() или Position, найти позицию слова “on” и вписать замену вместо “on your holiday” на “birthday to your”.
3. Прочитать содержимое файла и вывести стоку на экран.

- Для преобразования строки в массив байт можно использовать метод:

```
byte[] input = Encoding.Default.GetBytes(text);
```

- Для преобразования массива байт в строку можно использовать метод:

```
string text = Encoding.Default.GetString(input);
```

Делегаты

Делегат представляет собой класс, который может ссылаться на метод. Следовательно, когда создается делегат, то в итоге получается объект, содержащий ссылку на метод, метод можно вызывать по этой ссылке. Иными словами, делегат позволяет вызывать метод, на который он ссылается.

Делегат создан для возможности передавать методы в качестве аргументов функций.

Делегат в C# подобен указателю на функцию в C/C++.

Несмотря на то, что .NET использует концепцию функционального указателя посредством делегатов, есть несколько существенных отличий:

- делегаты нечувствительны к ошибкам ввода;
- объектно-ориентированы;
- безопасны.

Делегаты C# обладают следующими свойствами:

1. позволяют обрабатывать методы в качестве аргумента;
2. могут быть связаны вместе;
3. несколько методов могут быть вызваны по одному событию;
4. тип делегата определяется его именем;
5. не зависят от класса объекта, на который ссылается;
6. сигнатура метода должна совпадать с сигнатурой делегата.

Тип делегата объявляется с помощью ключевого слова **delegate**. Общая форма объявления делегата:

```
delegate   возвращаемый_тип  имя(список_параметров);
```

где *возвращаемый_тип* обозначает тип значения, возвращаемого методами, которые будут вызываться делегатом; *имя* — конкретное имя делегата; *список_параметров* — параметры, необходимые для методов, вызываемых делегатом. Как только будет создан экземпляр делегата, он может вызывать и ссылаться на те методы, возвращаемый тип и параметры которых соответствуют указанным в объявлении делегата.

Делегат может служить для вызова любого метода с соответствующей сигнатурой и возвращаемым типом. Вызываемый метод может быть методом экземпляра, связанным с отдельным объектом, или же статическим методом, связанным с конкретным классом. Значение имеет лишь одно: возвращаемый тип и сигнатура метода должны быть согласованы с теми, которые указаны в объявлении делегата.

```
// Объявление делегата
delegate void MyDelegate(string a);
class DelegateExample {
static void Func(string param) {
Console.WriteLine("Вызвана функция с параметром{0}.",param);
}
public static void Main() {
// Создание экземпляра делегата
MyDelegate f = new MyDelegate(Func);
// Вызов функции
f("hello"); }
}
```

```
delegate string StrMod (string str); // Объявить тип делегата.

class DelegateTest {
static string ReplaceSpaces (string s) {// Заменить пробелы
дефисами.
Console.WriteLine("Замена пробелов дефисами.");  return
s.Replace(' ', '-');}
static string RemoveSpaces(string s) {// Удалить пробелы.
string temp = "";    int i;    Console.WriteLine("Удаление
пробелов.");
for(i=0; i < s.Length; i++)  if(s[i] != ' ') temp += s[i];
return temp;          }
static string Reverse (string s) {// Обратить строку.
string temp = "";   int i, j;   Console.WriteLine("Обращение строки.");
for(j=0, i=s.Length-1; i >= 0; i--, j++)  temp += s[i];     return temp;   }

static void Main() {
StrMod strOp = new StrMod (ReplaceSpaces); // Сконструировать делегат.

string  str =strOp("Это простой тест."); // Вызвать методы с помощью
делегата.
Console.WriteLine("Результирующая строка: " + str);

strOp = new StrMod(RemoveSpaces);
str = strOp("Это простой тест.");
```

Групповая адресация

Одним из самых примечательных свойств делегата является поддержка групповой адресации. *Групповая адресация* — это возможность создать *список*, или *цепочку вызовов*, для методов, которые вызываются автоматически при обращении к делегату. Создать такую цепочку нетрудно.

Для этого достаточно получить экземпляр делегата, а затем добавить методы в цепочку с помощью оператора **+** или **+=**.

Для удаления метода из цепочки служит оператор **-** или **-=**.

Если делегат возвращает значение, то им становится значение, возвращаемое последним методом в списке вызовов. Поэтому делегат, в котором используется групповая адресация, обычно имеет возвращаемый тип **void**.

Делегаты, включающие в себя более одного метода, называются **мультикаст-делегатами**.

```
delegate void StrMod (ref string str);
class MultiCastDemo {
static void ReplaceSpaces(ref string s) {// Заменить пробелы
дефисами.
  Console.WriteLine("Замена пробелов дефисами."); s =
s.Replace(' ', '-');}
static void RemoveSpaces(ref string s) {// Удалить пробелы.
  string temp = "";          int i;
Console.WriteLine("Удаление пробелов.");
for(i=0; i < s.Length; i++)   if(s[i] != ' ') temp += s[i];
s = temp;  }
static void Reverse (ref string s) {// Обратить строку.
  string temp = "";  int i, j;  Console.WriteLine("Обращение
строки.");
for(j=0, i=s.Length-1; i >= 0; i--, j++) temp += s[i];  s =
temp; }
static void Main()      {//……
StrMod strOp; // Сконструировать делегаты.
StrMod replaceSp = ReplaceSpaces;        StrMod removeSp =
RemoveSpaces;
StrMod reverseStr = Reverse;                  string str =
"Это простой тест.";
strOp = replaceSp;
strOp += reverseStr;    // Организовать групповую адресацию.
strOp(ref str); // Обратиться к делегату с групповой адресацией.
```

Метод **Invoke()** для передачи аргументов делегату.

```
class Program
{   delegate int Operation(int x, int y);
    delegate void Message();

    static void Main(string[] args)
    {   Message mes = Hello;
        mes.Invoke();
        Operation op = Add;
        op.Invoke(3, 4);
        Console.Read();
    }
     static void Hello() { Console.WriteLine("Hello"); }
     static int Add(int x, int y) { return x + y; }
}
```

Другой способ передачи аргументов.

```
        Message mes = Hello;
        mes();
        Operation op = Add;
        op(3, 4);
```

Если делегат принимает параметры, то в метод **Invoke()** передаются значения для этих параметров.

Делегаты как параметры методов

```
class Program
{    delegate void GetMessage();

    static void Main(string[] args)
{if (DateTime.Now.Hour < 12) Show_Message(GoodMorning);
      else                   Show_Message(GoodEvening);
}
    private static void Show_Message(GetMessage _del)
    {       _del.Invoke();    }

    private static void GoodMorning()
    {       Console.WriteLine("Good Morning");    }

    private static void GoodEvening()
    {       Console.WriteLine("Good Evening");
    }}
```

Если делегат возвращает некоторое значение, то возвращается значение последнего метода из списка вызова (если в списке вызова несколько методов).

```
class Program
{
    delegate int Operation(int x, int y);

    static void Main(string[] args)
    {
        Operation op = Subtract;
        op += Multiply;
        op += Add;
        Console.WriteLine(op(7, 2));    // Add(7,2) = 9
        Console.Read();
 }
    private static int Add(int x, int y) { return x + y; }
    private static int Subtract(int x, int y) { return x - y; }
    private static int Multiply(int x, int y) { return x * y; }
}
```

Упражнение37

1. Создать делегат обработки строковой переменной.
2. Написать метод, который в строке вместо русских букв «а», «А», «о» и «О» записывал символ «+».
3. Написать программу, которая позволяет написать строку на консоли, а затем на следующей строке выводит преобразованную строку.

Анонимные функции

Метод, на который ссылается делегат, нередко используется только для этой цели. Иными словами, единственным основанием для существования метода служит то обстоятельство, что он может быть вызван посредством делегата, но сам он не вызывается вообще. В подобных случаях можно воспользоваться **анонимной функцией**, чтобы не создавать отдельный метод.

Анонимная функция, по существу, представляет собой безымянный кодовый блок, передаваемый конструктору делегата. Преимущество анонимной функции состоит, в частности, в ее простоте. Благодаря ей отпад ает необходимость объявлять отдельный метод, единственное назначение которого состоит в том, что он передается делегату.

```
// Объявить тип делегата.
delegate void CountIt();
class AnonMethDemo {
static void Main() {
// Далее следует код для подсчета чисел, передаваемый делегату
// в качестве анонимного метода.
CountIt count = delegate {
// Этот кодовый блок передается делегату.
for(int i=0; i <= 5; i++)          Console.WriteLine(i);

                                  }; обратите внимание на точку с запятой
count();
}
```

```
//  теперь у делегата CountIt имеется параметр.
delegate void CountIt (int end);
class AnonMethDemo2 {
        static void Main() {
// Здесь конечное значение для подсчета передается анонимному методу.
    CountIt count = delegate (int end) {
    for(int i=0; i <= end; i++)
    Console.WriteLine(i);
                                                    };
count(3);
Console.WriteLine();
count(5);
}
```

Возврат значения из анонимного метода

```
// Этот делегат возвращает значение.
delegate int CountIt (int end);
class AnonMethDemo3 {
                static void Main() {
int result;
// Здесь конечное значение для подсчета передается анонимному методу.
// А возвращается сумма подсчитанных чисел.
CountIt count = delegate (int end) {
int sum = 0;
for(int i=0; i <= end; i++) {    Console.WriteLine (i);
sum += i;                        }
return sum; // возвратить значение из анонимного метода
                                 };
result = count(3);
Console.WriteLine("Сумма 3 равна " + result);
Console.WriteLine();
result = count (5);
Console.WriteLine("Сумма 5 равна " + result);
}
```

```
// Этот делегат возвращает значение типа int и принимает аргумент типа int.
delegate int CountIt(int end);
class VarCapture {
static CountIt Counter() { int sum = 0;// Здесь сумма сохраняется в переменной sum.
CountIt ctObj = delegate (int end) {
for(int i=0; i <= end; i++) {Console.WriteLine(i);
sum += i;}  return sum;   };
return ctObj;
}
static void Main() {
// Получить результат подсчета.
CountIt count = Counter();
int result;
result = count(3);       Console.WriteLine("Сумма 3 равна " + result);
result = count (5);       Console.WriteLine("Сумма 5 равна " + result);
}
```

Упражнение38

1. Написать программу (два варианта) с использованием делегата, анонимной функции и лямбда выражение (второй вариант), которая вычисляет число целых чисел на интервале кратных числу 6.
2. Границы интервала задаются как аргументы делегата, делегат возвращает найденное число.
3. Границы интервала задаются с консоли.

Лямбда-выражения

На смену анонимных методов пришел более совершенный подход: *лямбда-выражение*. Лямбда- выражение относится к одним из самых важных нововведений в C#.

Лямбда-оператор

Во всех лямбда-выражениях применяется новый лямбда-оператор **=>**, который разделяет лямбда-выражение на две части. В левой его части указывается входной параметр (или несколько параметров), а в правой части — тело лямбда-выражения. Оператор **=>** иногда описывается такими словами, как "переходит" или "становится".

параметр => выражение

Если же требуется указать несколько параметров, то используется следующая форма.

(список_параметров) => выражение

Пример одиночного лямбда-выражения.

```
count => count + 2
```

```
// Объявить делегат, аргумент типа int и возвращающий
результат типа int.
delegate int Incr (int v);
// Объявить делегат,  типа int и возвращающий результат типа
bool.
delegate bool IsEven (int v);
class SimpleLambdaDemo {
static void Main() {
    // Создать делегат Incr, ссылающийся на лямбда-выражение.
    // увеличивающее свой параметр на 2.
Incr incr = count => count + 2;    //или Incr incr = (int)
count => count + 2;
    // А теперь использовать лямбда-выражение incr.
int x = -10;
while(x <= 0) {
Console.Write(x + " "); x = incr (x); // увеличить значение x
на 2
            }
```

// Создать экземпляр делегата IsEven, ссылающийся на лямбда-выражение,
// возвращающее логическое значение true, если его параметр имеет четное
// значение, а иначе — логическое значение false.

```
IsEven isEven = n => n % 2 == 0;
for(int i=1; i <= 10; i++)
if( isEven(i) ) Console.WriteLine(i + " четное.");
}
```

У лямбда-выражений может быть любое количество параметров, в том числе и нулевое. Если в лямбда-выражении используется несколько параметров, их *необходимо* заключить в скобки.

Пример использования лямбда- выражения с целью определить, находится ли значение в заданных пределах.

```
(low, high, val) => val >= low && val <= high;
```

А вот как объявляется тип делегата, совместимого с этим лямбда- выражением.

```
delegate bool InRange(int lower, int upper, int
                      v);
```

Экземпляр делегата **InRange** может быть создан следующим образом.

```
InRange rangeOK = (low, high, val) => val >= low
&& val <= high;
```

После этого одиночное лямбда-выражение может быть выполнено так

Блочные лямбда-выражения

```
// Делегат IntOp принимает один аргумент типа int
возвращает результат типа int.
delegate int IntOp(int end);
class StatementLambdaDemo {
static void Main() {// Блочное лямбда-выражение
возвращает факториал  значения.
IntOp fact = n => {
int r = 1;
for(int i=1; i <= n; i++)
r = i * r;
return r;
};
Console.WriteLine("Факториал 3 равен " + fact(3));
Console.WriteLine("Факториал 5 равен " + fact(5));
}
```

События

Еще одним важным средством C#, основывающимся на делегатах, является ***событие***.

Событие, по существу, представляет собой автоматическое уведомление о том, что произошло некоторое действие. События действуют по следующему принципу: *объект, проявляющий интерес к событию, регистрирует обработчик этого события. Когда же событие происходит, вызываются все зарегистрированные обработчики этого события.* Обработчики событий обычно представлены **делегатами**.

События являются членами класса и объявляются с помощью ключевого слова **event**.

Чаще всего для этой цели используется следующая форма:

```
event делегат_события имя_события;
```

где *делегат_события* обозначает имя делегата, используемого для поддержки события, а *имя_события* — конкретный объект объявляемого события.

```
delegate void MyEventHandler();

class MyEvent {// Объявить класс, содержащий событие.
public event MyEventHandler SomeEvent;
// Этот метод вызывается для запуска события.
public void OnSomeEvent() {
if(SomeEvent != null)  SomeEvent();}
}
class EventDemo {
// Обработчик события.
static void Handler() {Console.WriteLine("Произошло событие");        }

static void Main() {
            MyEvent evt = new MyEvent();
// Добавить метод Handler() в список событий.
      evt.SomeEvent += Handler;
// Запустить событие.
      evt.OnSomeEvent();
}
```

Групповая адресации события

```
delegate void MyEventHandler(); // Объявить тип делегата для
события.
class MyEvent {// Объявить делегат, содержащий событие.
public event MyEventHandler SomeEvent;
public void OnSomeEvent() {// Этот метод вызывается для запуска
события.
if(SomeEvent != null) SomeEvent(); }
                              }
class X {
public void Xhandler() { Console.Write("Событие получено объектом
классаX ");} }
class Y {
public void Yhandler() { Console.Write("Событие получено объектом
класса Y");}}
class EventDemo2 {
static void Handler() { Console.Write("Событие получено  класса
EventDemo"); }

static void Main() {
    MyEvent evt = new MyEvent();
    X xOb = new X();                Y yOb = new Y();
evt.SomeEvent += Handler; // Добавить обработчики в список
событий.
```

Пример

```
class Account
{    // Объявляем делегат
public delegate void AccountStateHandler(string message);
public event AccountStateHandler WithdrawN;//Событие, возникающее при выводе
                          денег
public event AccountStateHandler Added;//Событие, возникающее при добавление на
                          счеТ
     int _sum; // Переменная для хранения суммы
public Account (int sum){        _sum = sum;    }
public int CurrentSum {        get { return _sum; }    }
public void Put (int sum)
    {         _sum += sum;
        if (Added != null)
            Added($"На счет поступило {sum}");
    }
public void Withdraw_ (int sum)
{  if (sum <= _sum)        {
            _sum -= sum;
   if (WithdrawN != null)  WithdrawN ($"Сумма{sum}снята со счета"); }
else
 {if (WithdrawN != null) WithdrawN ("Недостаточно денег на счете"); }
 }
}
```

Основная программа

```
class Program
{
    static void Main (string[] args)
    {
        Account account = new Account(200);
        // Добавляем обработчики события
        account.Added += Show_Message;
        account.WithdrawN += Show_Message;

        account.Withdraw_ (100);
        // Удаляем обработчик события
        account.WithdrawN -= Show_Message;

        account.Withdraw_(50);
        account.Put (150);
                        Console.ReadLine();
    }
    private static void Show_Message(string message)
    {
        Console.WriteLine(message);
    }
}
```

Упражнение39

1. Написать программу контроля счета с использованием **событий**. Начальное значение счета – 0.
2. Для пополнения счета на консоли набрать число со знаком +. Вывести на консоль сообщение: «На счету: *число*.»
3. Для списания средств со счета на консоли набрать число со знаком -. Вывести на консоль сообщение: «На счету: *число*.» Если средств не хватает вывести сообщение: «Средств недостаточно», но при этом счет не обнуляется

Timer

```
using System.Threading;// добавить
      static void Main(string[] args)      {
Timer timer = new Timer(showTime, null, 0, 2000); //Время указывается в миллисекундах
  Console.ReadLine();      }

static void showTime(object obj)//    делегат
      {    Console.WriteLine("Текущие дата и время: {0}",
DateTime.Now.ToString());  }
```

showTime — первый параметр передаёт делегат, который предоставляет выполняемый метод.

Вторым параметром передаётся объект, в котором содержится информация для метода ответного вызова. Если ничего не передаём, ставим **null**.

Третьим параметром передаётся количество времени, которое пройдёт, прежде чем заработает метод **showTime()**.

4-ый параметр — это количество времени между запусками метода, т.е. **showTime** будет отрабатывать раз в 2 секунды

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading; //!!!

static int con=0;// общая переменная
static void Main(string[] args){
int num = 10;
  TimerCallback tm = new TimerCallback(Count); // устанавливаем
метод обратного вызова
 Timer timer = new Timer(tm, num, 0, 2000); // создаем таймер
 Console.ReadLine();
}

    public static void Count(object obj) {
    int x = (int)obj;
Console.Write(con++);
for (int i = 1; i < 9; i++, x++){
Console.WriteLine("{0}", x*i);  }
}
```

```
using System;
using System.Timers;

class myApp  {
public static void Main()
{      Timer myTimer = new Timer();
 myTimer.Elapsed += new ElapsedEventHandler( DisplayTimeEvent );
 myTimer.Interval = 1000;
myTimer.Start();
       while ( Console.Read() != 'q' );
}


public static void DisplayTimeEvent(object source,ElapsedEventArgs e)
{   Console.Write("\r{0}", DateTime.Now);    }
}
```

```
using System.Timers;
//изменение времени срабатывания таймера при работе. Выход – нажать esc.
static int t=0,n = 0;
        static void Main(string[] args)
        { int s = 500;
            Timer tm = new Timer();
            tm.Elapsed += TimerEvent;
 tm.Interval = s;
tm.Start();
            for(; ; )   {
if (n != 0) { s += n; n = 0; tm.Interval = s; Console.WriteLine(s); }
                if (t != 0) return;            }          }


        public static void TimerEvent (Object s, System.Timers.ElapsedEventArgs e)
{  Console.Write(e.SignalTime);
   n = 100;
            if (Console.ReadKey().Key == ConsoleKey.Escape) t = 1;
}
```

```
System.Diagnostics.Stopwatch myStopwatch = new
System.Diagnostics.Stopwatch();
 myStopwatch.Start(); //запуск
//....
myStopwatch.Stop(); //остановить
```

Публичный API класса **Stopwatch** выглядит следующий образом:

Свойства

Elapsed — возвращает общее затраченное время;

ElapsedMilliseconds — возвращает общее затраченное время в миллисекундах;

ElapsedTicks — возвращает общее затраченное время в тактах таймера;

IsRunning — возвращает значение, показывающее, запущен ли таймер Stopwatch.

Методы

Reset — останавливает измерение интервала времени и обнуляет затраченное время

Restart — останавливает измерение интервала времени, обнуляет затраченное время
начинает измерение затраченного времени;

Start — запускает или продолжает измерение затраченного времени для интервала;

StartNew — инициализирует новый экземпляр Stopwatch, задает свойство затраченног
времени равным нулю и запускает измерение затраченного времени;

Stop — останавливает измерение затраченного времени для интервала.

Поля

Frequency — возвращает частоту таймера, как число тактов в секунду;

IsHighResolution — указывает, зависит ли таймер от счетчика производительности
высокого разрешения.

Упражнение40

1. Написать программу, которая запускает таймер при нажатии клавиши “s” или “S” и останавливает при нажатии той же клавиши.
2. На консоли должно отображаться отсчитанное время в формате **часы:минуты:секунды:сотые доли секунды**
3. Для окончания работы программы нажать клавишу “q”.

Что такое обобщения

Термин ***обобщение***, по существу, означает *параметризированный тип*. Особая роль параметризированных типов состоит в том, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра.

С помощью **обобщений** можно, например, создать единый класс, который автоматически становится пригодным для обработки разнотипных данных.

Класс, структура, интерфейс, метод или делегат, оперирующий параметризированным типом данных, называется *обобщенным*, как, например, *обобщенный класс* или *обобщенный метод*.

В C# всегда имеется возможность создавать обобщенный код, оперируя ссылками типа **`object`**. А поскольку класс **`object`** является базовым для всех остальных классов, то по ссылке типа **`object`** можно обращаться к объекту любого типа. Таким образом, до появления обобщений для оперирования разнотипными объектами в программах служил обобщенный код, в котором для этой цели использовались ссылки типа **`object`**.

Обобщения как языковое средство позволяют создавать классы, структуры, интерфейсы, методы и делегаты для обработки разнотипных данных с соблюдением типовой безопасности.

Многие алгоритмы очень похожи по своей логике независимо от типа данных, к которым они применяются. Например, механизм, поддерживающий очередь, остается одинаковым независимо от того, предназначена ли очередь для хранения элементов типа `int`, `string`, `object` или для класса, определяемого пользователем.

До появления обобщений для обработки данных разных типов приходилось создавать различные варианты одного и того же алгоритма. А благодаря обобщениям можно сначала выработать единое решение независимо от конкретного типа данных, а затем применить его к обработке данных самых разных типов без каких-либо дополнительных усилий.

```
class Account
{   public int Id { get; set; }
    public int Sum { get; set; }
}
class Account
{   public object Id { get; set; }
    public int Sum { get; set; }
}


Account account1 = new Account { Sum = 5000 };
Account account2 = new Account { Sum = 4000 };
account1.Id = 2;
account2.Id = "4356";
int id1 = (int)account1.Id;
string id2 = (string)account2.Id;
Console.WriteLine(id1);
Console.WriteLine(id2);
```

```
class Account<T>
{
    public T Id { get; set; }
    public int Sum { get; set; }
}

Account<int> account1 = new Account<int> { Sum = 5000 };
Account<string> account2 = new Account<string> { Sum = 4000 };
account1.Id = 2;          // упаковка не нужна
account2.Id = "4356";
int id1 = account1.Id;  // распаковка не нужна
string id2 = account2.Id;
Console.WriteLine(id1);
Console.WriteLine(id2);
```

Статические поля обобщенных классов

```
class Account<T>
{
    public static T session; //статическое поле

    public T Id { get; set; }
    public int Sum { get; set; }
}


Account<int> account1 = new Account<int> { Sum = 5000 };
Account<int>.session = 5436;

Account<string> account2 = new Account<string> { Sum = 4000 };
Account<string>.session = "45245";

Console.WriteLine(Account<int>.session);      // 5436
Console.WriteLine(Account<string>.session);   // 45245
```

Обобщенные методы

```
class Program{
    private static void Main(string[] args)
    {   int x = 7;
        int y = 25;
        Swap<int>(ref x, ref y);
        Console.WriteLine($"x={x}    y={y}");   // x=25   y=7

        string s1 = "hello";
        string s2 = "bye";
        Swap<string>(ref s1, ref s2);
        Console.WriteLine($"s1={s1}    s2={s2}"); // s1=bye
s2=hello

        Console.Read();    }

    public static void Swap<T> (ref T x, ref T y)
    {
        T temp = x;
        x = y;
        y = temp;
    }
}
```

```
// В классе Gen параметр типа T заменяется реальным типом
данных при создании объекта типа Gen.
class Gen<T> {
T ob; // объявить переменную типа T
public Gen(T o) {ob = o;} // у этого конструктора имеется
параметр типа T.
// Возвратить переменную экземпляра ob, которая относится к
типу T.
public T GetOb() {return ob;}
   // Показать тип T.
public void ShowType() {Console.WriteLine("К типу T относится "
+ typeof(T));}
}
Gen <int> iOb; // Создать переменную ссылки на объект Gen типа
int.
// Создать объект типа Gen<int> и присвоить ссылку на него
переменной iOb.
iOb = new Gen<int>(102);
iOb.ShowType(); // Показать тип данных, хранящихся в переменной
iOb.
int v = iOb.GetOb(); // Получить значение переменной iOb.
Console.WriteLine("Значение: " + v);
   // Создать объект типа Gen для строк.
```

Обобщенный класс с двумя параметрами типа

```
class TwoGen<T, V> {
T ob1;
V ob2;
public TwoGen(T o1, V o2) {
ob1 = o1;
ob2 = o2;                                    }
// Показать типы T и V.
public void showTypes() {Console.WriteLine("К типу T относится " +
typeof(T));
Console.WriteLine("К типу V относится " + typeof(V));                    }
public T getob1() {return ob1; }
public V GetObj2() { return ob2;  }
                    }
```

Общая форма обобщенного класса

Синтаксис обобщений может быть сведен к общей форме. Ниже приведена общая форма объявления обобщенного класса.

class *имя_класса<список_параметров_типа>* { ...}

Синтаксис объявления ссылки на обобщенный класс.

имя_класса<список_аргументов_типа> *имя_переменной* **new** *имя_класса<список_параметров_типа>*
(список_аргументов_конструктора);

Использование нескольких универсальных параметров

```
class Transaction<U, V>
{   public U FromAccount { get; set; }  // с какого счета перевод
    public U ToAccount { get; set; }    // на какой счет перевод
    public V Code { get; set; }         // код операции
    public int Sum { get; set; }        // сумма перевода
}


Account<int> acc1 = new Account<int> { Id = 1857, Sum = 4500 };
Account<int> acc2 = new Account<int> { Id = 3453, Sum = 5000 };

Transaction<Account<int>, string> transaction1 = new
Transaction<Account<int>, string>
{
    FromAccount = acc1,
    ToAccount = acc2,
    Code = "45478758",
    Sum = 900
};
```

Ограничения универсальных типов

С помощью выражения **where T : Account** указываем, что используемый тип T обязательно должен быть классом Account или его наследником. Благодаря подобному ограничению можно использовать внутри класса Transaction все объекты типа T именно как объекты Account и соответственно обращаться к их свойствам и методам.

```
class Account
{   public int Id { get; private set;} // номер счета
    public int Sum { get; set; }
    public Account(int _id){      Id = _id;     }}

class Transaction<T> where T: Account
{   public T FromAccount { get; set; }  // с какого счета перевод
    public T ToAccount { get; set; }    // на какой счет перевод
    public int Sum { get; set; }        // сумма перевода
  public void Execute()    {
if (FromAccount.Sum > Sum){
            FromAccount.Sum -= Sum;
            ToAccount.Sum += Sum;
Console.WriteLine($"Счет {FromAccount.Id}: {FromAccount.Sum}$ \nСчет
{ToAccount.Id}: {ToAccount.Sum}$");}
      else
{Console.WriteLine($"Недостаточно денег на счете {FromAccount.Id}");}
}}
```

```
class Program
{
    static void Main(string[] args)
    {
        Account acc1 = new Account(1857) { Sum = 4500 };
        Account acc2 = new Account(3453) { Sum = 5000 };
        Transaction<Account> transaction1 = new Transaction<Account>
        {
            FromAccount = acc1,
            ToAccount = acc2,
            Sum = 6900
        };
        transaction1.Execute();

        Console.ReadLine();
    }
}
```

Упражнение41

1. Написать программу учета счетов и перевода средств с использованием обобщенного класса. Номер 1-го счета RS34, сумма 1500. Номер 2-го счета 547, сумма 2300.
2. Счет должен описывается с помощью класса и состоять из номера счета: у первого счета строковая переменная у второго счета числовая переменная. Номер счета должен быть закрытым.
3. На основе полученного класса создать обобщенный класс для описания перевода денег со счета на счет.
4. Вывести приглашение на выполнение действия: «Перевести со счета сумму», с консоли задать переводимую сумму и номер счета, затем вывести «на счет». Если номера счетов не совпадают, вывести сообщение: «Счета №XXX нет».
5. Если средств на счете не хватает вывести сообщение: «Средств на счете №XXX для списывания не достаточно». Если операция прошла успешно вывести на экран номера счетов и значения счетов.

LINQ

Аббревиатура **LINQ** означает **Language-Integrated Query**, т. е. язык интегрированных запросов. Это понятие охватывает ряд средств, позволяющих извлекать информацию из источника данных.

Извлечение данных составляет важную часть многих программ. Например, программа может получать информацию из списка заказчиков, искать информацию в каталоге продукции или получать доступ к учетному документу, заведенному на работника.

Как правило, такая информация хранится в базе данных, существующей отдельно от приложения. Так, каталог продукции может храниться в реляционной базе данных. В прошлом для взаимодействия с такой базой данных приходилось формировать запросы на языке структурированных запросов (SQL). А для доступа к другим источникам данных, например в формате XML, требовался отдельный подход.

Основы LINQ

В основу LINQ положено понятие запроса, в котором определяется информация, получаемая из источника данных. Как только запрос будет сформирован, его можно выполнить. Это делается, в частности, в цикле **foreach**. В результате выполнения запроса выводятся его результаты.

Поэтому использование запроса может быть разделено на две главные стадии. На первой стадии запрос формируется, а на второй — выполняется. Таким образом, при формировании запроса определяется, что именно следует извлечь из источника данных. А при выполнении запроса выводятся конкретные результаты.

Для обращения к источнику данных по запросу, сформированному средствами LINQ, в этом источнике должен быть реализован интерфейс IEnumerable. Он имеет две формы: обобщенную и необобщенную. Как правило, работать с источником данных легче, если в нем реализуется обобщенная форма **IEnumerable<T>**, где T обозначает обобщенный тип перечисляемых данных.

Простой запрос

Все запросы начинаются с оператора **from**, определяющего два элемента. Первым из них является переменная диапазона, принимающая элементы из источника данных. Вторым элементом является источник данных (например, массив nums). Тип переменной диапазона выводится из источника данных. Поэтому переменная n относится к типу int.

from переменная_диапазона **in** источник_данных

Далее следует оператор **where**, обозначающий условие, которому должен удовлетворять элемент в источнике данных, чтобы его можно было получить по запросу.

where булево_выражение

В этой форме булево_выражение должно давать результат типа bool. Такое выражение иначе называется предикатом. В запросе можно указывать несколько операторов **where**.

Все запросы оканчиваются оператором **select** или **group,** в которых точно определяются, что именно должно быть получено по запросу.

```
using System;
using System.Linq;
class SimpQuery {
    static void Main() {
int[] nums = { 1, -2, 3, 0, -4, 5 };
// Сформировать простой запрос на получение только положительных значений.
var posNums = from n in nums
    where n > 0
    select n;
Console.Write("Положительные значения из массива nums: ");
// Выполнить запрос и отобразить его результаты.
foreach (int i in posNums) Console.Write(i + " ");
}
}
```

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

class SimpQuery {
static void Main() {
            int[] nums = { 1, -2, 3, 0, -4, 5 };
// Сформировать простой запрос на получение только положительных значений.
    var posNums = from n in nums
    where n > 0
     select n;
// Выполнить запрос и отобразить его результаты.
                        foreach (int i in posNums) Console.Write(i + " ");
// Внести изменения в массив nums.
Console.WriteLine("\nЗадать значение 99 для элемента массива nums[1].");
nums[1] = 99;
Consofe.Write("Положительные значения из массива nums после изменений в нем: ");
// Выполнить запрос второй раз.
                        foreach (int i in posNums) Console.Write(i + " ");
}}
```

Тип переменной диапазона должен соответствовать типу элементов, хранящихся в источнике данных. Следовательно, тип переменной диапазона зависит от типа источника данных. Но выводимость типов может быть осуществлена при условии, что в источнике данных реализована форма интерфейса IEnumerable<T>, где **T** обозначает тип элементов в источнике данных. Форма интерфейса **IEnumerable<T>** реализуется во всех массивах.

Но если в источнике данных реализован необобщенный вариант интерфейса **IEnumerable**, то тип переменной диапазона придется указывать явно.

И это делается в операторе **from**.

```
var posNums = from int n in nums
// ...
```

Очевидно, что явное указание типа здесь не требуется, поскольку все массивы неявно преобразуются в форму интерфейса **IEnumerable<T>**, которая позволяет вывести тип переменной диапазона.

Тип объекта, возвращаемого по запросу, представляет собой экземпляр интерфейса **IEnumerable<T>**, где T — тип получаемых элементов. Следовательно, тип переменной запроса должен быть экземпляром интерфейса **IEnumerable<T>**, а значение T должно определяться типом значения, указываемым в операторе select.

С учетом явного указания типа **IEnumerable<int>** запрос можно было бы составить следующим образом.

```
IEnumerable<int> posNums = from n in nums
where n >0
 select n;
```

```
int[] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };
// Сформировать запрос на получение положительных значений меньше 10.
var posNums = from n in nums
where n > 0
where n < 10
select n;
Console.Write("Положительные значения меньше 10: ");
// Выполнить запрос и вывести его результаты.
foreach (int i in posNums) Console.Write (i + " ");
```

Или так

```
var posNums = from n in nums
where n > 0  &&  n < 10
select n;
```

```
string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер
Сити", "ПСЖ", "Барселона"};
var selectedTeams=from t in teams //определяем каждый объект из teams как t
     where t.ToUpper().StartsWith("Б") //фильтрация по критерию
                    orderby t  // упорядочиваем по возрастанию
                    select t; // выбираем объект
 foreach (string s in selectedTeams)
Console.WriteLine(s);
//---------------------------------------
    Console.WriteLine(s);
string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер
Сити", "ПСЖ", "Барселона" };
var selectedTeams = teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(
=> t);
 foreach (string s in selectedTeams)
    Console.WriteLine(s);
//---------------------------------------
```

Запрос

```
teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t => t)
```

будет аналогичен предыдущему. Он состоит из цепочки методов **Where** и **OrderBy**. В качестве аргумента эти методы принимают делегат или лямбда-выражение

Оператор **orderby** можно использовать для сортировки результатов запроса по одному или нескольким критериям.

orderby ***элемент*** ***порядок***

```
int[] nums = { 10, -19, 4, 7, 2, -5, 0 };
// Сформировать запрос на получение значений в отсортированном порядке.
var posNums = from n in nums
orderby n
select n;
// Выполнить запрос и вывести его результаты.
foreach (int i in posNums) Console.Write(i + " ");
```

Для того чтобы изменить порядок сортировки по нарастающей на сортировку по убывающей, достаточно указать ключевое слово **descending**, как показано ниже.

```
var posNums = from n in nums
orderby n descending
select n;
```

Для сортировки по нескольким критериям служит форма оператора **orderby**.

orderby *элемент_A направление, элемент_B направление, элемент_C направление, ...*

В данной форме *элемент_A* обозначает конкретный элемент, по которому проводится основная сортировка; *элемент_B* — элемент, по которому производится сортировка каждой группы эквивалентных элементов; *элемент_C* — элемент, по которому производится сортировка всех этих групп, и т.д. Таким образом, каждый последующий *элемент* обозначает дополнительный критерий сортировки. Во всех этих критериях указывать *направление* сортировки необязательно, но по умолчанию сортировка проводится по нарастающей.

```
class Account {
public string FirstName { get; private set; }
public string LastName { get; private set; }
public double Balance { get; private set; }
public string AccountNumber { get; private set; }
public Account(string fn, string ln, string accnum, double b) {
FirstName = fn;
LastName = ln;
AccountNumber = accnum;
Balance = b;
}   }
```

// Сформировать исходные данные.

```
Account[] accounts =
{ new Account("Том", "Смит", "132СК", 100.23),
new Account("Том", "Смит", "132CD", 10000.00),
new Account("Ральф", "Джонс", "436CD", 1923.85),
new Account("Ральф", "Джонс", "454ММ", 987.132),
new Account("Тед", "Краммер", "897CD", 3223.19),
new Account("Ральф", "Джонс", "434СК", -123.32),
new Account("Сара", "Смит", "543ММ", 5017.40),
new Account("Сара", "Смит", "547CD", 34955.79),
new Account("Сара", "Смит", "843СК", 345.00),
new Account("Альберт", "Смит", "445СК", -213.67),
new Account("Бетти", "Краммер","968ММ",5146.67),
new Account("Карл", "Смит", "078CD", 15345.99),
new Account("Дженни", "Джонс", "108СК", 10.98) };
```

```
/* Сформировать запрос на получение сведений о  банковских
счетах в отсортированном порядке. Отсортировать эти сведения
сначала по имени, затем по фамилии и, наконец, по остатку на
счете,*/

var accInfo = from acc in accounts
orderby acc.LastName, acc.FirstName, acc.Balance
select acc;
string str = "";
// Выполнить запрос и вывести его результаты.
foreach (Account acc in accInfo) {
if(str != acc.FirstName) { str = acc.FirstName;
}
Console.WriteLine("{0}, {1}\tНомер счета: {2},
{3,10:C}",
acc.LastName, acc.FirstName, acc. AccountNumber,
acc.Balance);
}
Console.WriteLine();
}
```

Упражнение42

1. Создать класс со свойствами для хранения строковой переменной и дробной переменной.
2. Класс должен иметь конструктор для присвоения значений переменным класса.
3. Создать массив экземпляров класса, размер которого задать с клавиатуры.
4. Заполнить в массиве строковые переменные случайным образом одной буквой от «а» до «я», числовые переменные дробными числами из интервала от -100.0 до +100.0. Вывести на экран полученный массив в виде: {B; 54,32} {c; -4,35} …
5. При нажатии **Enter** вывести массив отсортированный по строковым переменным в порядке возрастания, при повторном нажатии **Enter** вывести массив в порядке убывания по числовым переменным.
6. Для выхода из программы нажать на символ “q”.

Оператор **select** определяет конкретный тип элементов, получаемых по запросу.

select *выражение*

Здесь *выражение* просто обозначало имя переменной диапазона. Но применение оператора select не ограничивается только этой простой функцией. Он может также возвращать отдельную часть значения переменной диапазона, результат выполнения некоторой операции или преобразования переменной диапазона и даже новый тип объекта, конструируемого из отдельных фрагментов информации, извлекаемой из переменной диапазона. Такое преобразование исходных данных называется ***проецированием***.

```
double[] nums={ -10.0, 16.4, 12.125, 100.85, -2.2, 25.25, -3.5 };
// Сформировать запрос на получение квадратных корней всех  положительных значений, содержащихся в массиве nums.
var sqrRoots = from n in nums
where n > 0
select Math.Sqrt(n); // Он возвращает квадратный корень значения переменной диапазона.
// Выполнить запрос и вывести его результаты.
foreach (double r in sqrRoots)
Console.WriteLine("{0:#.##}", r);
```

```
class EmailAddress {
public string Name { get; set; }
public string Address { get; set; }
public EmailAddress(string n, string a) {
Name = n;
Address = a;}}
//-------------------------------------------------------------------------------
EmailAddress[] addrs = {
new EmailAddress("Герберт", "Herb@HerbSchildt.com"),
new EmailAddress("Tom", "Tom@HerbSchildt.com"),
new EmailAddress("Сара", "Sara@HerbSchildt.com")};
// Сформировать запрос на получение адресов электронной почты.
var eAddrs = from entry in addrs
select entry.Address;
// Выполнить запрос и вывести его результаты.
foreach (string s in eAddrs)  Console.WriteLine(" " + s);
}
```

Herb@HerbSchildt.com
Tom@HerbSchildt.com
Sara@HerbSchildt.com

Применение вложенных операторов from

```
class ChrPair {
public char First;
public char Second;
public ChrPair(char c, char c2) {First = c;Second = c2; } }
//-------------------------------------------------------------------------
char[] chrs = { 'A', 'B', 'C' };
char[] chrs2 = { 'X', 'Y', 'Z' };
```

// В первом операторе from организуется циклическое обращение к массиву символов chrs, а во втором операторе from — циклическое обращение к массиву символов chrs2.

```
var pairs = from ch1 in chrs
            from ch2 in chrs2
select new ChrPair(ch1, ch2);
foreach (var p in pairs)
Console.WriteLine("{0} {1}", p.First, p.Second);  }
```

```
A X
A Y
A Z
B X
B Y
B Z
C X
C Y
C Z
```

Группирование результатов с помощью оператора group

Одним из самых эффективных средств формирования запроса является оператор
group, поскольку он позволяет группировать полученные результаты по ключам. Используя последовательность сгруппированных результатов, можно без особого труда получить доступ ко всем данным, связанным с ключом. Благодаря этому свойству оператора **group** доступ к данным, организованным в последовательности связанных элементов, осуществляется просто и эффективно.

Оператор group является одним из двух операторов, которыми может оканчиваться запрос. (Вторым оператором, завершающим запрос, является select.)

group переменная_диапазона **by** ключ

Этот оператор возвращает данные, сгруппированные в последовательности, причем каждую последовательность обозначает общий ключ.

Результатом выполнения оператора **group** является последовательность, состоящая из элементов типа **IGrouping<TKey, TElement>,** т.е. обобщенного интерфейса, объявляемого в пространстве имен System.Linq. В этом интерфейсе определена коллекция объектов с общим ключом. Типом переменной запроса, возвращающего группу, является **IEnumerable<IGrouping<TKey, TElement>>**. В интерфейсе IGrouping определено также доступное только для чтения свойство

```
string[] websites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",
"hsNameD.com", "hsNameE.org", "hsNameF.org",
"hsNameG.tv", "hsNameH.net", "hsNameI.tv" };

// Сформировать запрос на получение списка веб-сайтов, группируемых по имени домена самого верхнего уровня.
var webAddrs = from addr in websites
where addr.LastIndexOf('.') != -1
group addr by addr.Substring(addr.LastIndexOf('.'));
// Выполнить запрос и вывести его результаты.
foreach (var sites in webAddrs) {
Console.WriteLine("Веб-сайты, сгруппированные " + "по имени домена" + sites.Key);
foreach (var site in sites)   Console.WriteLine(" " + site);
Console.WriteLine();              }
```

```
Веб-сайты, сгруппированные по имени домена .com
hsNameA.com
hsNameD.com
Веб-сайты, сгруппированные по имени домена .net
hsNameB.net
HsNameC.net
Веб-сайты, сгруппированные по имени домена .org
hsNameE.org
HsNameF.org
. . . .
```

Продолжение запроса с помощью оператора into

При использовании в запросе оператора **select** или **group** иногда требуется сформировать временный результат, который будет служить продолжением запроса для получения окончательного результата. Такое продолжение осуществляется с помощью оператора **into** в комбинации с оператором **select** или **group**.

into имя тело_запроса

где имя обозначает конкретное имя переменной диапазона, используемой для циклического обращения к временному результату в продолжении запроса, на которое указывает тело_запроса. Когда оператор into используется вместе с оператором **select** или **group**, то его называют продолжением запроса, поскольку он продолжает запрос. По существу, продолжение запроса воплощает в себе принцип построения нового запроса по результатам предыдущего.

```
string[] websites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",
"hsNameD.com", "hsNameE.org", "hsNameF.org",
"hsNameG.tv", "hsNameH.net", "hsNameI.tv" };

/* Сформировать запрос на получение списка веб-сайтов, группируемых по
имени домена самого верхнего уровня, но выбрать только те группы, которые
состоят более чем из двух членов. Здесь ws — это переменная диапазона для
ряда групп, возвращаемых при выполнении первой половины запроса.*/
var webAddrs = from addr in websites
where addr.LastIndexOf('.') != -1
group addr by addr.Substring(addr.LastIndexOf('.'))
                    into ws
where ws.Count() > 2
select ws;
Console.WriteLine("Домены самого верхнего уровня " +"с более чем двумя
членами.\n");
foreach(var sites in webAddrs) {
Console.WriteLine("Содержимое домена: " + sites.Key);
foreach(var site in sites)
Console.WriteLine(" " + site);        Console.WriteLine();}
```

```
Содержимое домена: .net
hsNameB.net
HsNameC.net
hsNameH.net
```

Применение оператора **let** для создания временной переменной в запросе

Иногда возникает потребность временно сохранить некоторое значение в самом запросе. Допустим, что требуется создать переменную перечислимого типа, которую можно будет затем запросить, или же сохранить некоторое значение, чтобы в дальнейшем использовать его в операторе **where**. Независимо от преследуемой цели, эти виды функций могут быть осуществлены с помощью оператора **let**. Общая форма оператора **let**:

let *имя = выражение*

где *имя* обозначает идентификатор, получающий значение, которое дает выражение. Тип имени выводится из типа выражения.

```
string[] strs = ( "alpha", "beta", "gamma" };
/* Сформировать запрос на получение символов, возвращаемых
из строк в отсортированной последовательности. Обратите внимание
на применение вложенного оператора from.*/
var chrs = from str in strs
let chrArray = str.ToCharArray()
from ch in chrArray
orderby ch
select ch;
Console.WriteLine("Отдельные символы, отсортированные по порядку:");
// Выполнить запрос и вывести его результаты.
foreach(char c in chrs) Console.Write(c + " "); Console.WriteLine();
```

```
Отдельные символы, отсортированные по порядку:
a a a a a b e g h l m m p t
```

ToCharArray() и возвращает массив, содержащий символы в строке. Полученный результат присваивается переменной **chrArray**, которая затем используется во вложенном операторе **from** для извлечения отдельных символов из массива.

Объединение двух последовательностей с помощью оператора join

Когда приходится иметь дело с базами данных, то часо требуется формировать последовательность, увязывающую данные из разных источников. Например, в Интернет-магазине может быть организована одна база данных, связывающая наименование
товара с его порядковым номером, и другая база данных, связывающая порядковый номер товара с состоянием его запасов на складе. В подобной ситуации
может возникнуть потребность составить список, в котором состояние запасов товаров на складе отображается по их наименованию, а не порядковому номеру. Для этой цели придется каким-то образом "увязать" данные из двух разных источников (баз данных).
И это нетрудно сделать с помощью такого средства LINQ, как оператор **join**.
Общая форма оператора **join** (совместно с оператором from).

from переменная_диапазона_A **in** источник_данных_A
join переменная_диапазона_B **in** источник_данных_B
on переменная_диапазона_A. свойство **equals** переменная_диапазона_B. Свойство

Применяя оператор **join**, следует иметь в виду, что каждый источник должен содержать
общие данные, которые можно сравнивать. Поэтому в приведенной форме этого оператора источник_данных_A и источник_данных_B должны иметь нечто общее, что подлежит сравнению. Сравниваемые элементы данных указываются в части on

// Класс, связывающий наименование товара с его порядковым номером.

```
class Item {
public string Name { get; set; }
public int ItemNumber { get; set; }
public Item(string n, int inum) {
Name = n;
ItemNumber = inum;}   }
```

// Класс, связывающий наименование товара с состоянием его запасов на складе.

```
class InStockStatus { public int ItemNumber { get; set; }
public bool InStock { get; set; }
public InStockStatus(int n, bool b) {
ItemNumber = n;
InStock = b;
}   }
```

// Класс, инкапсулирующий наименование товара и состояние его запасов на складе.

```
class Temp {
public string Name { get; set; }
public bool InStock { get; set; }
public Temp(string n, bool b) {
Name = n;
InStock = b;
}   }
```

```
Item[] items = {
new Item("Кусачки", 1424),
new Item("Тиски",     7892),
new Item("Молоток", 8534),
new Item("Пила",      6411)
};
InStockStatus[] statusList = {
new InStockStatus(1424, true),
new InStockStatus(7892, false),
new InStockStatus(8534, true),
new InStockStatus(6411, true)
};
```

/* Сформировать запрос, объединяющий объекты классов Item и InStockStatus для составления списка наименований товаров и их наличия на складе. Обратите внимание на формирование последовательности объектов класса Temp.*/

```
var inStockList = from item in items
join entry in statusList
on item.ItemNumber equals entry.ItemNumber
select new Temp(item.Name, entry.InStock);
Console.WriteLine("Товар\tНаличие\n");
// Выполнить запрос и вывести его результаты.
foreach(Temp t in inStockList)
Console.WriteLine("{0}\t{1}”,t.Name, t.InStock);
}
```

```
Товар     Наличие
Кусачки   True
Тиски     False
Молоток   True
Пила      True
```

Упражнение43

1. Имеются сведения об индексе популярности: Париж:29; Берлин:25; Варшава:20; Лондон:29; Рим:29; Познань:20; Хельсенки:25; Осло:20; Стокгольм:25; Флоренция:29; Антверпен:20.
2. Создать запрос и вывести в столбик список объектов в порядке убывания по названию.
3. Создать запрос группирующий список с одинаковыми индексами популярности.

Основные методы запроса

Методы запроса определяются в классе `System.Linq.Enumerable` и реализуются в виде *методов расширения* функций обобщенной формы интерфейса `IEnumerable<T>` в классе **`System.Linq.Queryable`**, расширяющем функции обобщенной формы интерфейса `IQueryable<T>`.

Метод расширения дополняет функции другого класса, но без наследования. В классе `Enumerable` предоставляется немало методов запроса, но основными считаются те методы, которые соответствуют операторам запроса.

Эти методы имеют также перегружаемые формы, <u>именно эти формы используется чаще всего.</u>

Оператор запроса	**Эквивалентный метод запроса**
select	Select(*selector*)
where	Where(*predicate*)
orderby	OrderBy(*keySelector*) или OrderByDescending(*keySelector*)
join	Join(*inner, outerKeySelector, innerKeySelector, resultSelector*)
group	GroupBy(*keySelector*)

Список используемых методов расширения LINQ

Select: определяет проекцию выбранных значений
Where: определяет фильтр выборки
OrderBy: упорядочивает элементы по возрастанию
OrderByDescending: упорядочивает элементы по убыванию
ThenBy: задает дополнительные критерии для упорядочивания элементов возрастанию
ThenByDescending: задает дополнительные критерии для упорядочивания элементов по убыванию
Join: соединяет две коллекции по определенному признаку
GroupBy: группирует элементы по ключу
ToLookup: группирует элементы по ключу, при этом все элементы добавляются в словарь
GroupJoin: выполняет одновременно соединение коллекций и группировку элементов по ключу
Reverse: располагает элементы в обратном порядке
All: определяет, все ли элементы коллекции удовлятворяют определенному условию
Any: определяет, удовлетворяет хотя бы один элемент коллекции определенному условию
Contains: определяет, содержит ли коллекция определенный элемент
Distinct: удаляет дублирующиеся элементы из коллекции
Except: возвращает разность двух коллекцию, то есть те элементы, которые содерятся только в одной коллекции
Union: объединяет две однородные коллекции
Intersect: возвращает пересечение двух коллекций, то есть те элементы, которые встречаются в обоих коллекциях
Count: подсчитывает количество элементов коллекции, которые удовлетворяют определенному условию
Sum: подсчитывает сумму числовых значений в коллекции
Average: подсчитывает среднее значение числовых значений в коллекции
Min: находит минимальное значение
Max: находит максимальное значение
Take: выбирает определенное количество элементов
Skip: пропускает определенное количество элементов
TakeWhile: возвращает цепочку элементов последовательности, до тех пор, пока условие истинно
SkipWhile: пропускает элементы в последовательности, пока они удовлетворяют заданному условию, и затем возвращает оставшиеся элементы
Concat: объединяет две коллекции
Zip: объединяет две коллекции в соответствии с определенным условием
First: выбирает первый элемент коллекции
FirstOrDefault: выбирает первый элемент коллекции или возвращает значение по умолчанию
Single: выбирает единственный элемент коллекции, если коллекция содердит больше или меньше одного элемента, то генерируется исключение
SingleOrDefault: выбирает первый элемент коллекции или возвращает значение по умолчанию
ElementAt: выбирает элемент последовательности по определенному индексу
ElementAtOrDefault: выбирает элемент коллекции по определенному индексу или возвращает значение по умолчанию, если индекс вне допустимого диапазона
Last: выбирает последний элемент коллекции
LastOrDefault: выбирает последний элемент коллекции или возвращает значение по умолчанию

За исключением метода **`Join()`**, методы запроса принимают единственный аргумент, который представляет собой объект некоторой разновидности обобщенного типа `Func<T, TResult>`. Это тип встроенного делегата, объявляемый следующим образом:

```
delegate TResult Func<in T, out TResult> (T arg)
```

где **`TResult`** обозначает тип результата, который дает делегат, а **`T`** — тип элемента.

В методах запроса аргументы *`selector`*, *`predicate`* или *`keySelector`* определяют действие, которое предпринимает метод запроса. Например, в методе `Where()` аргумент *`predicate`* определяет порядок отбора данных в запросе. Каждый метод запроса возвращает перечислимый объект. Поэтому результат выполнения одного метода запроса можно использовать для вызова другого, соединяя эти методы в цепочку.

Join(*inner, outerKeySelector, innerKeySelector, resultSelector*)

Метод **`Join()`** принимает четыре аргумента. Первый аргумент (***`inner`***) представляет собой ссылку на вторую объединяемую последовательность, а первой является последовательность, для которой вызывается метод `Join()`.

Селектор ключа для первой последовательности передается в качестве аргумента ***`outerKeySelector`***, а селектор ключа для второй последовательности — в качестве аргумента ***`innerKeySelector`***.

Результат объединения обозначается как аргумент ***`resultSelector`***.

Аргумент ***`outerKeySelector`*** имеет тип `Func<TOuter, TKey>`,

аргумент ***`innerKeySelector`*** — тип `Func<TInner, TKey>`, тогда как аргумент ***`resultSelector`*** — тип `Func<TOuter, Tinner, TResult>`,

где `TOuter` — тип элемента из вызывающей последовательности;

`Tinner` — тип элемента из передаваемой последовательности;

`TResult` — тип элемента из объединяемой в итоге последовательности, возвращаемой в виде перечислимого объекта.

Используя методы запроса одновременно с лямбда-выражениями, можно формировать запросы, вообще не пользуясь синтаксисом, предусмотренным в C# для запросов. Вместо этого достаточно вызвать соответствующие методы запроса.

```
int[] nums = { 1, -2, 3, 0, -4, 5 };
// Использовать методы Where() и Select() для формирования простого запроса.
var posNums = nums.Where(n => n > 0).Select(r => r);
Console.Write("Положительные значения из массива nums: ");
// Выполнить запрос и вывести его результаты.
foreach (int i in posNums) Console.Write(i + " ");
```

```
string[ ] websites = {
"hsNameA.com", "hsNameB.net", "hsNameC.net",
"hsNameD.com", "hsNameE.org", "hsNameF.org",
"hsNameG.tv", "hsNameH.net", "hsNameI.tv"
};

// Использовать методы запроса для группирования веб-
сайтов по имени домена самого верхнего уровня.
var webAddrs = websites.Where(w => w.LastIndexOf('.') !=
1).
GroupBy(x => x.Substring(x.LastIndexOf(".", x.Length)));
// Выполнить запрос и вывести его результаты.
foreach (var sites in webAddrs) {
Console.WriteLine("Веб-сайты сгруппированные по имени домена"
+ sites.Key);
foreach(var site in sites) Console.WriteLine(" " + site);
Console.WriteLine();}
```

Применения оператора **join**.

```
var inStockList = from item in items
join entry in statusList
on item.ItemNumber equals entry.ItemNumber
select new Temp(item.Name, entry.InStock);
```

// Использовать метод запроса **Join**() для составления списка наименований товаров и состояния их запасов на складе.

```
var inStockList = items.Join(statusList,
k1 => k1.ItemNumber,
k2 => k2.ItemNumber,
(k1, k2) => new Temp(k1.Name, k2.InStock)
);
```

Запросы в C# можно формировать двумя способами, используя синтаксис запросов или методы запроса. Оба способа связаны друг с другом более тесно, чем кажется, глядя на исходный код программы.

Дело в том, что синтаксис запросов компилируется в вызовы методов запроса. Поэтому код

```
where x < 10
```

будет преобразован компилятором в следующий вызов.

```
Where(x => x < 10)
```

Таким образом, оба способа формирования запросов в конечном итоге сходятся на одном и том же. Но если оба способа оказываются в конечном счете равноценными, то какой из них лучше для программирования на C#?

В целом, рекомендуется чаще пользоваться синтаксисом запросов, поскольку он полностью интегрирован в язык C#, поддерживается соответствующими ключевыми словами и синтаксическим конструкциями.

Дополнительные методы расширения, связанные с запросами

Метод	Описание
All(*predicate*)	Возвращает логическое значение true, если все элементы в последовательности удовлетворяют условию, задаваемому параметром *predicate*
Any(*predicate*)	Возвращает логическое значение true, если любой элемент в последовательности удовлетворяет условию, задаваемому параметром *predicate*
Average()	Возвращает среднее всех значений в числовой последовательности
Contains(*value*)	Возвращает логическое значение true, если в последовательности содержится указанный объект
Count()	Возвращает длину последовательности, т.е. количество составляющих ее элементов
First()	Возвращает первый элемент в последовательности
Last()	Возвращает последний элемент в последовательности
Max()	Возвращает максимальное значение в последовательности
Min()	Возвращает минимальное значение в последовательности
Sum()	Возвращает сумму значений в числовой последовательности

```
int[] nums ={ 3, 1, 2, 5, 4 };

Console.WriteLine("Минимальное значение равно " +
nums.Min());

Console.WriteLine("Максимальное значение равно " +
nums.Max());

Console.WriteLine("Первое значение равно " +
nums.First());

Console.WriteLine("Последнее значение равно " +
nums.Last());

Console.WriteLine("Суммарное значение равно " +
nums.Sum());

Console.WriteLine("Среднее значение равно " +
nums.Average());

if(nums.All(n => n > 0)) Console.WriteLine("Все значения
больше нуля.");

if(nums.Any(n => (n % 2) == 0)) Console.WriteLine("По
крайней мере одно значение является четным.");
```

Использовать метод Average() вместе с синтаксисом запросов.

```
int[] nums = { 1, 2, 4, 8, 6, 9, 10, 3, 6, 7 };
var ltAvg = from n in nums
let x = nums.Average()
where n < x
select n;
Console.WriteLine("Среднее значение равно
nums.Average());
Console.Write("Значения меньше среднего: ");
// Выполнить запрос и вывести его результаты.
foreach(int i in ltAvg) Console.Write(i + " ");
```

Упражнение44

1. Написать программу, которая задает массив целых чисел, размер массива определяется случайным образом в пределах от 0 до 1000.
2. Значения чисел разыгрываются случайным образом на интервале от A до B, при этом строго A<B. A и B определяются случайным образом из интервала от --1000 до плюс +1000.
3. Используя запросы и методы запросов определить следующее:

- Минимальное значение;
- Максимальное значение;
- Первое значение;
- Последнее значение;
- Суммарное значение;
- Среднее значение;
- Сколько четных чисел;
- Размер массива;
- Сколько чисел больше нуля;
- Сколько меньше нуля;
- Сколько равно нулю.

Вариант выполнения операций над комплексными числами

```
class Complex
{   double re,im ; // двухмерные координаты
    public Complex() { re = im = 0; }
    public Complex(double re, double im) { this.re = re; this.im=im; }
    // Перегрузить бинарный оператор +.
    public static Complex operator +(Complex A, Complex B)
    {   Complex result = new Complex();
        /* Сложить координаты двух точек и возвратить результат. */
        result.re = A.re + B.re; // Эти операторы выполняют
        result.im = A.im + B.im; // целочисленное сложение,
        return result;     }
    public static Complex operator *(Complex A, Complex B)
    {       Complex result = new Complex();
         result.re = A.re * B.re-A.im*B.im;
        result.im = A.re*B.im + A.im*B.re;
        return result;     }
    // Вывести значения re и im
    public void Show() {    Console.WriteLine("Re ="+re + " Im = " + im );}
//-------------------------------------------------------------------------------
Complex a,b,c;
        a = new Complex(3, 5); b = new Complex(-2, -5);
        c = a * b;
        c.Show();
```

Упражнение45

1. Написать программу, в которой с консоли вводятся два комплексных числа.
2. Вычислить результат выполнения операций:+, -, *, /.
3. Вычислить куб этих чисел.
4. Найти угол на комплексной поверхности этих чисел.
5. Найти модули этих чисел.

Нечёткий поиск в тексте и словаре

Алгоритмы нечеткого поиска, также известного как *поиск по сходству* или *fuzzy string search*, являются основой систем проверки орфографии и полноценных поисковых систем вроде Google или Yandex. Например, такие алгоритмы используются для функций наподобие «Возможно вы имели в виду …» в тех же поисковых системах.

Расстояние Левенштейна (*редакционное расстояние*, *дистанция редактирования*) — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки второй кратчайшим образом. Обычно действия обозначаются так: **D** (*delete*) — удалить, **I** (insert) — вставить, **R** (*replace*) — заменить, **M** (*match*) — совпадение. Например, для 2-х строк «CONNECT» и «CONEHEAD» можно построить следующую таблицу преобразований:

M	**M**	**M**	**R**	**I**	**M**	**R**	**R**
C	**O**	**N**	**N**		**E**	**C**	**T**
C	O	N	**E**	**H**	E	**A**	**D**

Вычисление метрики Левенштейна

Метрика Левенштейна вычисляется с помощью алгоритма Вагнера-Фишера

		А	Р	Е	С	Т	А	Н	Т
	0	1	2	3	4	5	6	7	8
Д	1	1	2	3	4	5	6	7	8
А	2	1	2	3	4	5	5	6	7
Г	3	2	2	3	4	5	6	6	7
Е	4	3	3	2	3	4	5	6	7
С	5	4	4	3	2	3	4	5	6
Т	6	5	5	4	3	2	3	4	5
А	7	6	6	5	4	3	2	3	4
Н	8	7	7	6	5	4	3	2	3

Матрица процесса вычисления метрики

$X_з$	$X_у$
$X_в$	$D_{i,j}$

$$D_{ij} = min(X_в + 1, X_у + 1, X_з + C_{замены})$$

$$C_{замены} = \begin{cases} 1, & если\ S_1[i] \neq S_2[j] \\ 0, & иначе \end{cases}$$

Формула вычисления значения очередного элемента матрицы

$X_в$, $X_з$, $X_у$ – предыдущие значения метрики, соответственно, для вставок, замен и удалений символов.
$C_{замены}$ – «стоимость» замены символа
D_{ij} – результирующее значение метрики

Алгоритм Вагнера — Фишера

```
public static int LevenshteinDistance(string string1,string string2)
{if (string1==null) throw new ArgumentNullException("string1");
if (string2==null) throw new ArgumentNullException("string2");
int diff;
int [,] m = new int[string1.Length+1,string2.Length+1];
for (int i=0;i<=string1.Length;i++) { m[i,0]=i; }
for (int j=0;j<=string2.Length;j++) { m[0,j]=j; }


for (int i=1;i<=string1.Length;i++) {
    for (int j=1;j<=string2.Length;j++){
diff=(string1[i-1]==string2[j-1])?0:1;
m[i,j]=Math.Min(Math.Min(m[i-1,j]+1, m[i,j-1]+1),m[i-1,j-1]+diff);}
}
return m[string1.Length,string2.Length];                              }
```

Упражнение46

1. Имеется словарь: {обвернутый, обвернуть, обвернуться, обвертеть, обвертеться, обвертка, обвертывание, обвертывать, обвертываться, обверченный, обверчивать, свернуть, вернул}.
2. Вывести на экран слова у которых **расстояние Левенштейна** не более 2 (≤ 2) соответственно для слов: *завертка, вернуть, обвернуто*.

В C# *коллекция* представляет собой **совокупность объектов**.

В среде .NET Framework имеется немало интерфейсов и классов, в которых определяются и реализуются различные типы коллекций.

Коллекции упрощают решение многих задач программирования благодаря тому, что предлагают готовые решения для создания целого ряда типичных, но порой трудоемких для разработки структур данных.

Например, в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связных списков, стеков, очередей и хеш-таблиц.

Хотя в языке C# есть массивы, которые хранят в себе наборы однотипных объектов, но работать с ними не всегда удобно. Например, массив хранит фиксированное количество объектов, однако что если мы заранее не знаем, сколько нам потребуется объектов. И в этом случае намного удобнее применять коллекции.

Еще один плюс **коллекций** состоит в том, что некоторые из них реализует стандартные структуры данных, например, стек, очередь, словарь, которые могут пригодиться для решения различных специальных задач.

В среде .NET Framework поддерживаются пять типов коллекций: необобщенные, специальные, с поразрядной организацией, обобщенные и параллельные.

Необобщенные коллекции

Реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также словари, в которых можно хранить пары "ключ-значение". В отношении необобщенных коллекций важно иметь в виду следующее: они оперируют данными типа **object**. Таким образом, необобщенные коллекции могут служить для хранения данных любого типа, причем в одной коллекции допускается наличие разнотипных данных. Такие коллекции не типизированы, поскольку в них хранятся ссылки на данные типа object. Классы и интерфейсы необобщенных коллекций находятся в пространстве имен **System.Collections**.

Специальные коллекции

Оперируют данными конкретного типа или же делают это каким-то особым образом. Например, имеются специальные коллекции для символьных строк, а также специальные коллекции, в которых используется однонаправленный список. Специальные коллекции объявляются в пространстве имен **System.Collections.Specialized**.

Поразрядная коллекция

В прикладном интерфейсе Collections API определена одна коллекция с поразрядной организацией — это **BitArray**. Коллекция типа BitArray поддерживает поразрядные операции, т.е. операции над отдельными двоичными разрядами, например И, ИЛИ, исключающее ИЛИ, а следовательно, она существенно отличается своими возможностями от остальных типов коллекций. Коллекция типа BitArray объявляется в пространстве имен System.Collections.

Обобщенные коллекции

Обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связные списки, стеки, очереди и словари. Такие коллекции являются типизированными в силу их обобщенного характера. Это означает, что в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Благодаря этому исключается случайное несовпадение типов. Обобщенные коллекции объявляются в пространстве имен **System.Collections.Generic**.

Параллельные коллекции

Поддерживают многопоточный доступ к коллекции. Это обобщенные коллекции, определенные в пространстве имен **System.Collections.Concurrent**.

Большая часть классов коллекций содержится в пространствах имен **System**.**Collections** (простые необобщенные классы коллекций),

System.**Collections**.**Generic** (обобщенные или типизированные классы коллекций)

System.**Collections**.**Specialized**(специальные классы коллекций).

Также для обеспечения параллельного выполнения задач и многопоточного доступа применяются классы коллекций из пространства имен **System**.**Collections.Concurrent**

Основой для создания всех коллекций является реализация интерфейсов **IEnumerator** и **IEnumerable**

и их обобщенных двойников

IEnumerator<T> и **IEnumerable<T>**.

Интерфейс **IEnumerator** представляет перечислитель, с помощью которого становится возможен последовательный перебор коллекции, например, в цикле **foreach**.

А интерфейс **IEnumerable** через свой метод GetEnumerator предоставляет перечислитель всем классам, реализующим данный интерфейс. Поэтому интерфейс **Ienumerable** (**IEnumerable<T>**) является базовым для всех коллекций.

Методы, определенные в интерфейсе IList

`int Add(object value)` - Добавляет объект *value* в вызывающую коллекцию. Возвращает индекс, по которому этот объект сохраняется

void Clear() - Удаляет все элементы из вызывающей коллекции

bool Contains (object *value)* - Возвращает логическое значение true, если вызывающая коллекция содержит объект *value,* а иначе — логическое значение false

int IndexOf(object *value)* - Возвращает индекс объекта *value,* если этот объект содержится в вызывающей коллекции. Если же объект *value* не обнаружен, то метод возвращает значение -1

void Insert(int *index,*object *value)* - Вставляет в вызывающую коллекцию объект *value* по индексу *index.* Элементы, находившиеся до этого по индексу *index* и дальше, смещаются вперед, чтобы освободить место для вставляемого объекта *value*

void Remove(object *value)* - Удаляет первое вхождение объекта *value* в вызывающей коллекции. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы устранить образовавшийся “пробел"

void RemoveAt(int *index)* - Удаляет из вызывающей коллекции объект, расположенный по указанному индексу *index.* Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы устранить образовавшийся “пробел”

Наиболее часто используемые методы, определенные в классе ArrayList

`public virtual void AddRange(Icollection c)` - Добавляет элементы из коллекции **`c`** в конец вызывающей коллекции типа `ArrayList`

`public virtual int BinarySearch(object value)` - Выполняет поиск в вызывающей коллекции значения *`value`*. Возвращает индекс найденного элемента. Если искомое значение не найдено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован

public virtual int BinarySearch(object *value,* Icomparer *comparer)* - Выполняет поиск в вызывающей коллекции значения *value,* используя для сравнения способ, определяемый параметром *comparer.* Возвращает индекс совпавшего элемента. Если искомое значение не найдено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован

public virtual int BinarySearch(int *index,*int *count,* object *value,*IComparer *comparer)* - Выполняет поиск в вызывающей коллекции значения *value,* используя для сравнения способ, определяемый параметром *comparer.* Поиск начинается с элемента, указываемого по индексу *index,* и включает количество элементов, определяемых параметром *count.* Метод возвращает индекс совпавшего элемента. Если искомое значение не найдено, метод возвращает отрицательное значение. Вызывающий список должен быть отсортирован

public virtual void CopyTo(Array *array)* - Копирует содержимое вызывающей коллекции в массив *array,* который должен быть одномерным и совместимым по типу с элементами коллекции

public virtual void CopyTo(Array *array,* int *arrayIndex)* - Копирует содержимое вызывающей

public static ArrayList FixedSize(ArrayList *list)* - Заключает коллекцию *list* в оболочку типа ArrayList с фиксированным размером и возвращает результат
public virtual ArrayList GetRange(int *ndex,* int *count) - Возвращает часть вызывающей коллекции типа ArrayList. Часть возвращаемой коллекции начинается с элемента, указываемого по индексу *index,* и включает количество элементов, определяемое параметром *count.* Возвращаемый объект ссылается на те же элементы, что и вызывающий объект
public virtual int IndexOf(object *value) - Возвращает индекс первого вхождения объекта *value* в вызывающей коллекции. Если искомый объект не обнаружен, возвращает значение -1
public virtual void InsertRange(int *index,* ICollection c) - Вставляет элементы коллекции с в вызывающую коллекцию, начиная с элемента, указываемого по индексу *index*
public virtual int LastIndexOf(object *value) - Возвращает индекс последнего вхождения объекта *value* в вызывающей коллекции. Если искомый объект не обнаружен, метод возвращает значение -1
public static ArrayList Readonly(ArrayList *list) - Заключает коллекцию *list* в оболочку типа ArrayList, доступную только для чтения, и возвращает результат
public virtual void RemoveRange(int *index,* int *count) - Удаляет часть вызывающей коллекции, начиная с элемента, указываемого по индексу *index,* и включая количество элементов, определяемое параметром *count*
public virtual void Reverse() - Располагает элементы вызывающей коллекции в обратном Порядке
public virtual void Reverse(int *index,* int *count) - Располагает в обратном порядке часть вызывающей
коллекции, начиная с элемента, указываемого по индексу *index,* и включая количество элементов, определяемое параметром *count*
public virtual void SetRange(int *index,* ICollection c) - Заменяет часть вызывающей коллекции, начиная с элемента, указываемого по индексу index, элементами коллекции с
public virtual void Sort() - Сортирует вызывающую коллекцию по нарастающей
public virtual void Sort(Icomparer *comparer) - Сортирует вызывающую коллекцию, используя для сравнения способ, определяемый параметром *comparer.* Если параметр *comparer* имеет пустое

public virtual void Sort(int *index*, int *count*, Icomparer *comparer)* - Сортирует вызывающую коллекцию, используя для сравнения способ, определяемый параметром *comparer*. Сортировка начинается с элемента, указываемого по индексу *index*, и включает количество элементов, определяемых параметром *count*. Если параметр comparer имеет пустое значение, то для сравнения используется способ, выбираемый по умолчанию

public static ArrayList Synchronized(ArrayList *list)* - Возвращает синхронизированный вариант коллекции типа ArrayList, передаваемой в качестве параметра *list*

public virtual object[ ] ToArray() - Возвращает массив, содержащий копии элементов вызывающего объекта

public virtual Array ToArray(Type *type)* - Возвращает массив, содержащий копии элементов вызывающего объекта. Тип элементов этого массива определяется

параметром *type*

public virtual void TrimToSize() - Устанавливает значение свойства Capacity равным значению свойства Count

```
using System;
using System.Collections;
using System.Collections.Generic;
 class Program
   {       static void Main(string[] args)       {
          // необобщенная коллекция ArrayList
          ArrayList objectList = new ArrayList() { 1, 2, "string", 'c', 2.0f };
           object obj = 45.8;
          objectList.Add(obj);
          objectList.Add("string2");
          objectList.RemoveAt(0); // удаление первого элемента
   foreach (object o in objectList) {  Console.WriteLine(o);  }
Console.WriteLine("Общее число элементов коллекции: {0}", objectList.Count);

          // обобщенная коллекция List
          List<string> countries = new List<string>(){"Россия","США","
Великобритания", "Китай" };
          countries.Add("Франция");
          countries.RemoveAt(1); // удаление второго элемента
  foreach (string s in countries) { Console.WriteLine(s); }
}   }
```

Классы коллекций в пространстве имен **`System.Collections`**:

- **ArrayList**: класс простой коллекции объектов. Реализует интерфейсы IList, ICollection, IEnumerable
- **BitArray**: класс коллекции, содержащей массив битовых значений. Реализует интерфейсы ICollection, IEnumerable
- **Hashtable**: класс коллекции, представляющей хэш-таблицу и хранящий набор пар "ключ-значение"
- **Queue**: класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел"). Реализует интерфейсы ICollection, IEnumerable
- **SortedList**: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы ICollection, IDictionary, IEnumerable
- **Stack**: класс стека

```
ArrayList list = new ArrayList();
            list.Add(2.3); // заносим в список объект типа double
            list.Add(55); // заносим в список объект типа int
list.AddRange(new string[] {"Hello","world"});//заносим в список строковый массив
             // перебор значений
            foreach (object o in list)
             { Console.WriteLine(o);           }

            // удаляем первый элемент
            list.RemoveAt(0);
            // переворачиваем список
            list.Reverse();
            // получение элемента по индексу
            Console.WriteLine(list[0]);
            // перебор значений
            for (int i = 0; i < list.Count; i++)
            {
                Console.WriteLine(list[i]);
            }
```

Классы коллекций в пространстве имен `System.Collections.Generic`:

- **List<T>**: класс, представляющий последовательный список. Реализует интерфейсы IList<T>, ICollection<T>, IEnumerable<T>
- **Dictionary<TKey, TValue>**: класс коллекции, хранящей наборы пар "ключ-значение". Реализует интерфейсы ICollection<T>, IEnumerable<T>, IDictionary<TKey, TValue>
- **LinkedList<T>**: класс двухсвязанного списка. Реализует интерфейсы ICollection<T> и IEnumerable<T>
- **Queue<T>**: класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел"). Реализует интерфейсы ICollection, IEnumerable<T>
- **SortedSet<T>**: класс отсортированной коллекции однотипных объектов. Реализует интерфейсы ICollection<T>, ISet<T>, IEnumerable<T>
- **SortedList<TKey, TValue>**: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы ICollection<T>, IEnumerable<T>, IDictionary<TKey, TValue>
- **SortedDictionary<TKey, TValue>**: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. В общем похож на класс SortedList<TKey, TValue>, основные отличия состоят лишь в использовании памяти и в скорости вставки и удаления
- **Stack<T>**: класс стека однотипных объектов. Реализует интерфейсы ICollection<T> и IEnumerable<T>

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

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

      numbers.Insert (0,666);//вставляем на первое место в списке число 666
      numbers.RemoveAt (1); //  удаляем второй элемент

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

   List<Person> people = new List<Person>(3);
            people.Add(new Person() { Name = "Том" });
            people.Add(new Person() { Name = "Билл" });

    foreach (Person p in people){  Console.WriteLine(p.Name);}
                      }     }
     class Person
    {
        public string Name { get; set; }
    }
```

Упражнение47

1. Создать необобщенную коллекцию содержащую значения: 1, 2, "string", 'c', 2.0f.
2. Вывести на консоль значения коллекции.
3. С клавиатуры ввести номер позиции коллекции куда вставить новое значение.
4. Ввести новое значение, которое может быть число или строка и вставить в указанную коллекцию, вывести на консоль новый набор значений коллекции.
5. Для удаления значения ввести символ "d", затем номер позиции и удалить член коллекции.
6. Вывести на консоль значения коллекции.

Коллекция Dictionary<T, V>

Словарь хранит объекты, которые представляют пару ключ-значение. Каждый такой объект является объектом структуры **KeyValuePair<TKey, TValue>**. Благодаря свойствам **Key** и **Value**, которые есть у данной структуры, можем получить ключ и значение элемента в словаре.

```
Dictionary<int, string> countries = new Dictionary<int, string>(5);
countries.Add(1, "Russia");
countries.Add(3, "Great Britain");
countries.Add(2, "USA");
countries.Add(4, "France");
countries.Add(5, "China");

foreach (KeyValuePair<int, string> keyValue in countries)
{    Console.WriteLine(keyValue.Key + " - " + keyValue.Value);     }
// получение элемента по ключу
string country = countries[4];
// изменение объекта
countries[4] = "Spain";
// удаление по ключу
countries.Remove(2);
```

Коллекция Hashtable

```
Hashtable myHash = new Hashtable();
  // добавляем элементы (ключом может быть и строка, любой тип)
  myHash.Add(20/*ключ*/, "Computer" /*значение*/);
  myHash.Add(30, "TV" );
  myHash.Add(5, "House" );
// в myHash элементы: {key=5, value="House"} {key=20, value="Computer"} {key=30, value="TV"}
  foreach (object key in myHash.Keys)   Console.WriteLine("Ключ="+key+" , значение="+myHash[key]);
    // установить значение по ключу
  myHash[33] = "aaa";  // в myHash элементы: {key=5, value="House"} {key=20, value="Computer"} {key=30, value="TV"} {key=33, value="aaa"}
  // получим значение по ключу
  object value = myHash[99];//Exception не будет если ключа такого нет value = null
  // узнать есть ли такой ключ
  bool isFound = myHash.ContainsKey(5);//Exception не будет если ключа такого нет   isFound = true
  // удаляем элемент по ключу 30 {key=30, value="TV"}
  myHash.Remove(30);
  // в myHash элементы: {key=5, value="House"} {key=20, value="Computer"} {key=33, value="aaa"}
```

Двухсвязный список LinkedList<T>

Класс `LinkedList<T>` представляет двухсвязный список, в котором каждый элемент хранит ссылку одновременно на следующий и на предыдущий элемент.
Если в простом списке List<T> каждый элемент представляет объект типа T, то в LinkedList<T> каждый узел представляет объект класса `LinkedListNode<T>`. Этот класс имеет следующие свойства:

Value: само значение узла, представленное типом T

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

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

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

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

```
class Program    {
  static void Main(string[] args){
            LinkedList<int> numbers = new LinkedList<int>();
 numbers.AddLast(1); // вставляем узел со значением 1 на последнее место
           // так как в списке нет узлов, то последнее будет также и первым
 numbers.AddFirst(2); // вставляем узел со значением 2 на первое место
 numbers.AddAfter(numbers.Last,3);//вставляем после последнего узла новый узел со
значением 3 теперь у нас список имеет следующую последовательность: 2, 1, 3
 foreach (int i in numbers)  { Console.WriteLine(i);        }
//.....................................................................
 LinkedList<Person> persons = new LinkedList<Person>();
// добавляем persona в список и получим объект LinkedListNode<Person>, в котором хранится имя
Tom
 LinkedListNode<Person> tom = persons.AddLast(new Person() { Name = "Tom" });
          persons.AddLast(new Person() { Name = "John" });
          persons.AddFirst(new Person() { Name = "Bill" });

 Console.WriteLine(tom.Previous.Value.Name);//получаем узел перед томом и
его значение
 Console.WriteLine(tom.Next.Value.Name);//получаем узел после тома и его
значение }    }
     class Person
    {        public string Name { get; set; }
    }
```

Заполнение коллекции с консоли

Чтобы избежать генерирования исключений при преобразовании числовых строк при вводе с консоли Console.Readline(), можно воспользоваться методом **`TryParse`**`()`, определенным для всех числовых структур. Например, метода `TryParse()`, определяемых в структуре **`Int32`**:

```
static bool TryParse(string s, out int результат)
```

где *s* обозначает числовую строку, передаваемую данному методу, который возвращает соответствующий *результат* после преобразования с учетом выбираемой по умолчанию местной специфики представления чисел. При неудачном исходе преобразования, например, когда параметр *s* не содержит числовую строку в надлежащей форме, метод `TryParse()` возвращает логическое значение `false`. В противном случае он возвращает логическое значение `true`.

```
double n;      string s="A7";
if(!Double.TryParse(s, out n))
{
//обработка, если не число
```

Определение типа переменной

```
static void Main(string[] args)
        {   int x = 1;
            double y = 2.3456;
            method(x, y);
            Console.Read();
        }

        public static void method(object first, object second)
        {
   if(first.GetType() == typeof(int)  && second.GetType() == typeof(double))
   {
                Console.WriteLine((int)first + (double)second);
   }
        }
```

Упражнение48

1. Имеется список счетов, у которых ключ **ID** строчная переменная, а баланс дробное число **float**.
2. Создать коллекцию **Dictionary**, заполнить десять счетов ID, которых d1, d2, d3,….d10. Баланс счетов заполнить случайными дробными числами из интервала [-100.0; 100.0].
3. Для просмотра значения баланса с консоли ввести ключ - сторочный.
4. При добавлении счета с консоли автоматически сгенерировать и добавить ключ на единицу больше существующего и поместить в конец коллекции, например d11, и ввести значение баланса. Вывести на консоль получившийся весь список.
5. Подсчитать сумму всех счетов и вывести на консоль.

Пример, как выделить число из строки "str123"

```
string a = "str123"; string b = string.Empty; int val;
for (int i=0; i< a.Length; i++) { if (Char.IsDigit(a[i])) b += a[i]; } if (b.Length>0)
val = int.Parse(b);
```

Класс `String`

Класс `String` определен в пространстве имен `System`. В нем реализуются следующие интерфейсы: `IComparable`, `IComparable<string>`, `ICloneable`, `IConvertible`, `IEnumerable`, `IEnumerable<char>` и `IEquatable<string>`. Кроме того, `String` — герметичный класс, а это означает, что он не может наследоваться. В классе `String` предоставляются все необходимые функциональные возможности для обработки символьных строк в C#.

Конструкторы класса String

В классе String определено несколько конструкторов, позволяющих создавать строки самыми разными способами. Для создания строки из символьного массива служит один из следующих конструкторов.

public String(char[ ] *value*)

public String(char[ ] *value*, int *startIndex*, int *length*)

Первая форма конструктора позволяет создать строку, состоящую из символов массива *value*. *А* во второй форме для этой цели из массива *value* извлекается определенное количество символов (*length*), начиная с элемента, указываемого по индексу *startIndex*.

С помощью приведенного ниже конструктора можно создать строку, состоящую из отдельного символа, повторяющегося столько раз, сколько потребуется:

public String(char c, int *count*)

где *c* обозначает повторяющийся символ; а *count* — количество его повторений.

Поле, индексатор и свойство класса `String`

В классе `String` определено единственное поле.

```
public static readonly string Empty
```

Поле `Empty` обозначает пустую строку, т.е. такую строку, которая не содержит символы. Этим оно отличается от пустой ссылки типа `String`, которая просто делается на несуществующий объект.

Помимо этого, в классе `String` определен единственный индексатор, доступный только для чтения.

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

Этот индексатор позволяет получить символ по указанному индексу. Индексация строк, как и массивов, начинается с нуля. Объекты типа `String` отличаются постоянством и не изменяются, поэтому вполне логично, что в классе `String` поддерживается индексатор, доступный только для чтения.

В классе `String` определено единственное свойство, доступное только для чтения.

```
public int Length { get; }
```

Свойство **`Length`** возвращает количество символов в строке.

Операторы класса `String`

В классе `String` перегружаются два следующих оператора: `==` и `!=`. Оператор `==` служит для проверки двух символьных строк на равенство. Когда оператор `==` применяется к ссылкам на объекты, он обычно проверяет, делаются ли обе ссылки на один и тот же объект. А когда оператор `==` применяется к ссылкам на объекты типа `String`, то на предмет равенства сравнивается содержимое самих строк. Это же относится и к оператору `!=`. Когда он применяется к ссылкам на объекты типа `String`, то на предмет неравенства сравнивается содержимое самих строк.

В то же время другие операторы отношения, в том числе `<` и `>=`, сравнивают ссылки на объекты типа `String` таким же образом, как и на объекты других типов. А для того чтобы проверить, является ли одна строка больше другой, следует вызвать метод **`Compare`**`()`, определенный в классе **`String`**.

Во многих видах сравнения символьных строк используются сведения о культурной среде. Но это не относится к операторам `= =` и `!=`. Ведь они просто сравнивают порядковые значения символов в строках. (они сравнивают двоичные значения символов, не видоизмененные нормами культурной среды, т.е. региональными стандартами.) Следовательно, эти операторы выполняют сравнение строк без учета регистра и настроек культурной среды.

Сцепление строк

Строки можно сцеплять, т.е. объединять вместе, двумя способами.

Во-первых, с помощью оператора **+**, как было показано ранее. И во-вторых, с помощью одного изметодов сцепления, определенных в классе **String**. Конечно, для этой цели проще всего воспользоваться оператором **+**, тем не менее методы сцепления служат неплохой альтернативой такому подходу. Метод, выполняющий сцепление строк, называется Concat(). Одна из самых распространенных его форм:

public static string **Concat**(string str0, string str1)

Этот метод возвращает строку, состоящую из строки str1, присоединяемой путем сцепления в конце строки str0.

Форма метода Concat() сцепляется произвольное количество строк:

public static string **Concat**(params string[] values)

где values обозначает переменное количество аргументов, сцепляемых для получения возвращаемого результата.

Значения, определяемые в перечислении StringComparison

CurrentCulture - Сравнение строк производится с использованием текущих настроек параметров культурной среды

CurrentCultureIgnoreCase - Сравнение строк производится с использованием текущих настроек параметров культурной среды, но без учета регистра

InvariantCulture - Сравнение строк производится с использованием неизменяемых, т.е. универсальных данных о культурной среде

InvariantCultureIngoreCase - Сравнение строк производится с использованием неизменяемых, т.е. универсальных данных о культурной среде и без учета регистра

Ordinal - Сравнение строк производится с использованием порядковых значений символов в строке. При этом лексикографический порядок может нарушиться, а условные обозначения, принятые в отдельной культурной среде, игнорируются

OrdinalIgnoreCase - Сравнение строк производится с использованием порядк овых значений символов в строке, но без учета регистра.
При этом лексикографический порядок может нарушиться, а условные обозначения, принятые в отдельной культурной среде, игнорируются

Поиск в строке

В классе String предоставляется немало методов для поиска в строке. С их помощью можно, например, искать в строке отдельный символ, строку, первое или последнее вхождение того и другого в строке. Поиск может осуществляться либо с учетом культурной среды либо порядковым способом. Для обнаружения первого вхождения символа или подстроки в исходной строке служит метод **IndexOf**(). Для него определено несколько перегружаемых форм.

public int **IndexOf**(char value)

В этой форме метода IndexOf() возвращается первое вхождение символа value в вызывающей строке. Если символ value в ней не найден, то возвращается значение -1. При таком поиске символа настройки культурной среды игнорируются. Следовательно, в данном случае осуществляется порядковый поиск первого вхождения символа.

Еще две формы метода **IndexOf**(), позволяющие искать первое вхождение одной строки в другой.

public int **IndexOf**(String value)

public int **IndexOf**(String value, StringComparison comparisonType)

В первой форме рассматриваемого здесь метода поиск первого вхождения строки, обозначаемой параметром value, осуществляется с учетом культурной среды. А во торой форме предоставляется возможность указать значение типа StringComparison, обозначающее способ поиска. В если искомая строка не найдена, то в обеих формах данного метода возвращается значение -1.

Заполнение и обрезка строк

Иногда в строке требуется удалить начальные и конечные пробелы. Такая операция называется обрезкой и нередко требуется в командных процессорах. Например, программа ведения базы данных способна распознавать команду "print", но пользователь может ввести эту команду с одним иди несколькими начальными и конечными пробелами. Поэтому перед распознаванием введенной команды необходимо удалить все подобные пробелы. С другой стороны, строку иногда требуется заполнить пробелами, чтобы она имела необходимую минимальную длину. Так, если подготавливается вывод результатов в определенном формате, то каждая выводимая строка должна иметь определенную длину, чтобы сохранить выравнивание строк. Для упрощения подобных операций в C# предусмотрены соответствующие методы.

Для обрезки строк используется одна из форм метода **Trim**().

public string **Trim**()

public string **Trim**(params char[] trimChars)

В первой форме метода Trim() из вызывающей строки удаляются начальные и конечные пробелы. А во второй форме этого метода удаляются начальные и конечные вхождения в вызывающей строке символов из массива trimChars. В обеих формах возвращается получающаяся в итоге строка.

Строку можно заполнить символами слева или справа. Для заполнения строки слева служат такие формы метода **PadLeft**().

public string **PadLeft**(int totalWidth)

public string **PadLeft**(int totalWidth, char paddingChar)

В первой форме метода PadLeft() вводятся пробелы с левой стороны вызывающей строки, чтобы ее общая

Вставка, удаление и замена строк

Для вставки одной строки в другую служит приведенный ниже метод **Insert**():

public string **Insert**(int startIndex, string value)

где value обозначает строку, вставляемую в вызывающую строку по индексу startIndex. Метод возвращает получившуюся в итоге строку.

Для удаления части строки служит метод **Remove**().

public string **Remove**(int startIndex)

public string **Remove**(int startIndex, int count)

В первой форме метода **Remove**() удаление выполняется, начиная с места, ука зываемого по индексу tartIndex, и продолжается до конца строки. А во второй форм е данного метода из строки удаляется количество символов, определяемое парам етром count, начиная с места, указываемого по индексу startIndex. В обеих формах возвращается получающаяся в итоге строка.

Для замены части строки служит метод **Replace**().

public string **Replace**(char oldChar, char newChar)

public string **Replace**(string oldValue, string newValue)

В первой форме метода Replace() все вхождения символа oldChar в вызывающе й строке заменяются символом newChar. А во второй форме данного метода все вхождения строки oldValue в вызывающей строке заменяются строкой newValue. В обеих формах возвращается получающаяся в итоге строка.

Смена регистра

В классе String предоставляются два удобных метода, позволяющих сменить регистр

букв в строке, — **ToUpper**() и **ToLower**().

public string **ToLower**()

public string **ToUpper**()

Метод **ToLower**() делает строчными все буквы в вызывающей строке, а метод ToUpper() делает их прописными. В обоих случаях возвращается получающаяся в итоге строка. Имеются также следующие формы этих методов, в которых можно указывать информацию о культурной среде и способы преобразования символов.

public string **ToLower**(CultureInfo culture)

public string **ToUpper**(CultureInfo culture)

С помощью этих форм можно избежать неоднозначности в исходном коде по отношению к правилам смены регистра. Именно для таких целей эти формы и рекомендуется применять. Кроме того, имеются следующие методы T o U p p e r I n v a r i a n t ( ) и ToLowerInvariant().

public string **ToUpperInvariant**()

public string **ToLowerInvariant**()

Эти методы аналогичны методам ToUpper() и ToLower(), за исключением того, что

они изменяют регистр букв в вызывающей строке безотносительно к настройка м культурной среды.

Разделение и соединение строк

К основным операциям обработки строк относятся разделение и соединение.

При *разделении* строка разбивается на составные части, а при соединении строка составляется из отдельных частей. Для разделения строк в классе `String` определен метод **`Split`**`()`, а для соединения — метод **`Join`**`()`.

Существует несколько вариантов метода `Split()`.

```
public string[ ] Split(params char[ ] separator)
public string[ ] Split(params char[ ] separator, int count)
```

В первой форме метода `Split()` вызывающая строка разделяется на составные части. В итоге возвращается массив, содержащий подстроки, полученные из вызывающей строки. Символы, ограничивающие эти подстроки, передаются в массиве *`separator`*. Если массив *`separator`* пуст или ссылается на пустую строку, то в качестве разделителя подстрок используется пробел.

А во второй форме данного метода возвращается количество подстрок, определяемых параметром *`count`*.

Существует несколько форм метода **`Join().`**

public static string **Join**(string *separator,* string[] *value*)

public static string **Join**(string *separator,*string[]*value,*int *startIndex,* int *count*)

В первой форме метода **Join**() возвращается строка, состоящая из сцепляемых подстрок, передаваемых в массиве *value*. Во второй форме также возвращается строка, состоящая из подстрок, передаваемых в массиве value, но они сцепляются в определенном количестве *count,* начиная с элемента массива *value*[*startIndex*]. В обеих формах каждая последующая строка отделяется от предыдущей разделительной строкой, определяемой параметром *separator.*

```
string str = "Один на суше, другой на море.";
char[] seps = {' ', '.', ','};
// Разделить строку на части.
string[] parts = str.Split(seps);
Console.WriteLine("Результат разделения
строки: ");
for(int i=0; i < parts.Length; i++)
Console.WriteLine(parts[i]);
// А теперь соединить части строки.
string whole = String.Join(" | ", parts);
Console.WriteLine("Результат соединения
строки: ");
Console.WriteLine(whole);
```

Один
на
суше

другой
на
море

Существует ряд других форм метода **`Split`**`()`, принимающих параметр типа **`StringSplitOptions`**. Этот параметр определяет, являются ли пустые строки частью разделяемой в итоге строки. .

```
    public string[] Split(params char[ ]
separator,StringSplitOptions options)
    public string[] Split(string[ ] separator, StringSplitOptions
options)
    public string[] Split(params char[ ] separator, int count,
StringSplitOptions options)
    public string[] Split(string[ ] separator, int count,
StringSplitOptions options)
```

В двух первых формах метода **`Split`**`()` вызывающая строка разделяется на части и возвращается массив, содержащий подстроки, полученные из вызывающей строки. Символы, разделяющие эти подстроки, передаются в массиве *`separator`*. Если массив *`separator`* пуст, то в качестве разделителя используется пробел.

В третьей и четвертой формах данного метода возвращается количество строк, ограничиваемое параметром *`count`*. Но во всех формах параметр *`options`* обозначает конкретный способ обработки пустых строк, которые образуются в том случае, если два разделителя оказываются рядом. В перечислении `StringSplitOptions` определяются только два значения: `None` и `RemoveEmptyEntries`.

Если параметр *`options`* принимает значение `None`, то пустые строки включаются в конечный результат разделения исходной строки, как показано в предыдущем примере программы. А если параметр *`options`* принимает

Упражнение49

1. Дана строка: «Первая форма конструктора позволяет создать строку из символов, доступных из массива по указателю *value.* При этом предполагается, что массив, доступный по указателю *value*, завершается пустым символом, обозначающим конец строки.».
2. Преобразовать строку в массив из отдельных слов.
3. Вывести на консоль эту строку по словам в обратном порядке.
4. Вывести на консоль слова в алфавитном порядке.

Основы многопоточной обработки

Различают две разновидности многозадачности: на основе **процессов** и на основе **потоков**.

Процесс фактически представляет собой исполняемую программу. Поэтому *многозадачность на основе процессов* — это средство, благодаря которому на компьютере могут параллельно выполняться две программы и более. Так, многозадачность на основе процессов позволяет одновременно выполнять программы текстового редактора, электронных таблиц и просмотра содержимого в Интернете.

При организации многозадачности на основе *процессов* программа является наименьшей единицей кода, выполнение которой может координировать планировщик задач.

Поток представляет собой координируемую единицу исполняемого кода. Своим происхождением этот термин обязан понятию "поток исполнения". При организации многозадачности на основе потоков у каждого процесса должен быть по крайней мере один поток, хотя их может быть и больше. Это означает, что в одной программе одновременно могут решаться две задачи и больше. Например, текст может форматироваться в редакторе текста одновременно с его выводом на печать, при условии, что оба эти действия выполняются в двух отдельных потоках.

Отличия в многозадачности на основе **процессов** и **потоков** могут быть сведены к следующему: многозадачность на основе процессов организуется для параллельного выполнения программ, а многозадачность на основе потоков — для параллельного выполнения отдельных частей одной программы.

Главное преимущество многопоточной обработки заключается в том, что она позволяет писать программы, которые работают очень эффективно благодаря возможности выгодно использовать время простоя, неизбежно возникающее в ходе выполнения большинства программ.

Классы, поддерживающие многопоточное программирование, определены в пространстве имен **System.Threading**. Поэтому любая многопоточная программа на C# включает в себя следующую строку кода.

```
using System.Threading;
```

Класс Thread

Система многопоточной обработки основывается на классе **Thread**, который инкапсулирует поток исполнения. Класс Thread является *герметичным*, т.е. он не может наследоваться. В классе Thread определен ряд методов и свойств, предназначенных для управления потоками.

Создание и запуск потока

Для создания потока достаточно получить экземпляр объекта типа **Thread**, т.е. класса, определенного в пространстве имен System.Threading. Простейшая форма конструктора класса **Thread**:

public **Thread**(ThreadStart *запуск*)

где *запуск* — это имя метода, вызываемого с целью начать выполнение потока, а ThreadStart — делегат, определенный в среде .NET Framework,

public **delegate** void ThreadStart()

Метод, указываемый в качестве точки входа в поток, должен иметь возвращаемый тип **void** и не принимать никаких аргументов. Вновь созданный новый поток не начнет выполняться до тех пор, пока не будет вызван его метод **Start**(), определяемый в классе **Thread**.

Существуют две формы объявления метода Start().

public void Start()

Однажды начавшись, поток будет выполняться до тех пор, пока не произойдет возврат из метода, на который указывает *запуск.* Таким образом, после возврата из этого метода поток автоматически прекращается. Если же попытаться вызвать метод **Start**() для потока, который уже начался, это приведет к генерированию исключения ThreadStateException.

```
class MyThread {
public int Count;                    string thrdName;
public MyThread (string name) {Count = 0;  thrdName = name;
}
// Точка входа в поток.
public void Run() {Console.WriteLine(thrdName + " начат.");
do {           Thread.Sleep(500);
Console.WriteLine("В потоке " + thrdName + ", Count = " +
Count); Count++;
} while(Count < 10);
Console.WriteLine(thrdName + " завершен.");
}
////////////////////////////////////////////////////////////
////////////////////////////////
// Сначала сконструировать объект типа MyThread.
MyThread mt = new MyThread ("Потомок #1");
// Далее сконструировать поток из этого объекта.
Thread newThrd = new Thread (mt.Run);
// начать выполнение потока.
newThrd.Start();
      do {
      Console.Write(".");
      Thread.Sleep(100);
      } while (mt.Count != 10);
```

```
class MyThread { public int Count;            public Thread Thrd;
public MyThread(string name) { Count = 0;
Thrd = new Thread(this.Run);
Thrd.Name = name;
Thrd.Start();}
// Точка входа в поток.
void Run() { Console.WriteLine(Thrd.Name + " начат.");
do {
Thread.Sleep(500); Console.WriteLine("В потоке " + Thrd.Name + ",
Count = " + Count);
Count++;
} while(Count < 10);
Console.WriteLine(Thrd.Name + " завершен.");}
}
//////////////////////////////////////////////////////////////////
//////////
// Сконструировать три потока.
MyThread mt1 = new MyThread("Потомок #1");
MyThread mt2 = new MyThread("Потомок #2") ;
MyThread mt3 = new MyThread("Потомок #3");
do {Console.Write("."); Thread.Sleep(100);}
while(mt1.Count < 10 || mt2.Count < 10 || mt3.Count < 10);
Console.WriteLine("Основной поток завершен.");
} }
```

Определение момента окончания потока

Как именно завершается поток. В классе **Thread** имеются два средства для определения момента окончания потока. С этой целью можно, прежде всего, опросить доступное только для чтения свойство **IsAlive**, определяемое следующим образом.

```
public bool IsAlive { get; }
```

Свойство **IsAlive** возвращает логическое значение true, если поток, для которого оно вызывается, по-прежнему выполняется.

```
// Сконструировать три потока.
MyThread mt1 = new MyThread("Поток #1");
MyThread mt2 = new MyThread("Поток #2");
MyThread mt3 = new MyThread("Поток #3");
do {
Console.Write(".");
Thread.Sleep(100);
} while(mt1.Thrd.IsAlive && mt2.Thrd.IsAlive && mt3.Thrd.IsAlive);
Console.WriteLine("Основной поток завершен.");
}
```

Еще один способ отслеживания момента окончания состоит в вызове метода **Join**(). Ниже приведена его простейшая форма.

```
public void Join()
```

Метод **Join**() ожидает до тех пор, пока поток, для которого он был вызван, не завершится. Его имя отражает принцип ожидания до тех пор, пока вызывающий поток не *присоединится* к вызванному методу. Если же данный поток не был начат, то генерируется исключение ThreadStateException. В других формах метода **Join**() можно указать максимальный период времени, в течение которого следует ожидать завершения указанного потока.

```
class MyThread {
public int Count;
public Thread Thrd;
public MyThread(string name) {Count = 0;   Thrd = new
Thread(this.Run);
Thrd.Name = name;   Thrd.Start();   }
// Точка входа в поток.
void Run() {Console.WriteLine(Thrd.Name + " начат.");
do {  Thread.Sleep(500);
Console.Write("В потоке"+Thrd.Name+",Count="+ Count); Count++;} While(Count
< 10);
Console.WriteLine(Thrd.Name + " завершен.");}
}
//////////////////////////////////////////////////
// Сконструировать три потока.
MyThread mt1 = new MyThread("Потомок #1");
MyThread mt2 = new MyThread("Потомок #2");
MyThread mt3 = new MyThread("Потомок #3");
mt1.Thrd.Join();        Console.WriteLine("Потомок #1 присоединен.");
mt2.Thrd.Join();        Console.WriteLine("Потомок #2 присоединен.");
mt3.Thrd.Join();        Console.WriteLine("Потомок #3 присоединен.");
Console.WriteLine("Основной поток завершен.");
```

Передача аргумента потоку

Для этого можно воспользоваться другими формами метода `Start()`, конструктора класса `Thread`, а также метода, служащего в качестве точки входа в поток.

Аргумент передается потоку в следующей форме метода `Start()`.

```
public void Start(object параметр)
```

Объект, указываемый в качестве аргумента *параметр,* автоматически передается методу, выполняющему роль точки входа в поток. Следовательно, для того чтобы передать аргумент потоку, достаточно передать его методу Start(). Для применения параметризированной формы метода Start() потребуется следующая форма конструктора класса **Thread**:

```
public Thread(ParameterizedThreadStart запуск)
```

где *запуск* обозначает метод, вызываемый с целью начать выполнение потока.

Важно, что в этой форме конструктора запуск имеет тип ParameterizedThreadStart, а не ThreadStart. В данном случае ParameterizedThreadStart является делегатом, объявляемым следующим образом.

```
public delegate void ParameterizedThreadStart(object obj)
```

Этот делегат принимает аргумент типа object. Поэтому для правильного применения данной формы конструктора класса Thread у метода, служащего в качестве точки входа в поток, должен быть параметр типа **object**.

```
class MyThread {
public int Count;                                    public Thread
Thrd;
// Обратите внимание на то, что конструктору класса MyThread передается также
значение типа int.
public MyThread(string name, int num) {       Count = 0;
// Вызвать конструктор типа ParameterizedThreadStart явным образом только ради
наглядности примера.
Thrd = new Thread(this.Run);
Thrd.Name = name;
// Здесь переменная num передается методу Start() в качестве
аргумента.
Thrd.Start(num);}
//Обратите внимание на то, что в этой форме метода Run() указывается параметр
типа object.
void Run(object num) { Console.WriteLine(Thrd.Name + " начат со
счета " + num);
do {       Thread.Sleep(500);
Console.WriteLine("В потоке " + Thrd.Name + ", Count = " +
Count);
Count++;
} while(Count < (int) num);
Console.WriteLine(Thrd.Name + " завершен.");
} }
////////////////////////////////////////////////////////////////////
//////////////////////////
```

Упражнение50

1. Написать программу в которой будет два потока. Один поток выводит все буквы алфавита, второй числа от 28 до 0, промежуток между выводом символов 0.4 сек.
2. По окончания потока вывести сообщение об окончании соответствующего потока.

Приоритеты потоков

У каждого потока имеется свой приоритет, который определяет, насколько часто поток получает доступ к ЦП. Низкоприоритетные потоки получают доступ к ЦП реже, чем высокоприоритетные. В течение заданного промежутка времени низкоприоритетному потоку будет доступно меньше времени ЦП, чем высокоприоритетному. Помимо приоритета, на частоту доступа потока к ЦП оказывают влияние и другие факторы. Если высокоприоритетный поток ожидает доступа к некоторому ресурсу, например для ввода с клавиатуры, он блокируется, а вместо него выполняется низкоприоритетный поток.

Конкретное планирование задач на уровне ОС также оказывает влияние на время ЦП, выделяемое для потока. Когда порожденный поток начинает выполняться, он получает приоритет, устанавливаемый по умолчанию. Приоритет потока можно изменить с помощью свойства **`Priority`**, являющегося членом класса **`Thread`**.

```
public ThreadPriority Priority { get; set; }
```

где `ThreadPriority` обозначает перечисление, в котором определяются значения приоритетов.

ThreadPriority.Highest
ThreadPriority.AboveNormal
ThreadPriority.Normal
ThreadPriority.BelowNormal
ThreadPriority.Lowest

По умолчанию для потока устанавливается значение приоритета ThreadPriority. **Normal**.

```
class MyThread {   public int Count;        public Thread Thrd;
static bool stop = false;                          static string
currentName;
/* Сконструировать новый поток. Обратите внимание на то, что данный
конструктор еще не начинает выполнение потоков. */
public MyThread (string name) { Count = 0;
Thrd = new Thread(this.Run);      Thrd.Name = name;   currentName =
name;
}
void Run () { // Начать выполнение нового потока.
Console.WriteLine("Поток " + Thrd.Name + " начат.");
do { Count++;       if(currentName != Thrd.Name) { currentName =
Thrd.Name;
Console.WriteLine("В потоке " + currentName); }
} while(stop == false && Count < 1000000000);
stop = true;             Console.WriteLine("Поток " + Thrd.Name + "
завершен.");} }
////////////////////////////////////////////////////////////////////
////////////
MyThread mt1 = new MyThread("с высоким приоритетом");
MyThread mt2 = new MyThread("с низким приоритетом");
// Установить приоритеты для потоков.
mt1.Thrd.Priority = ThreadPriority.AboveNormal;
mt2.Thrd.Priority = ThreadPriority.BelowNormal;
// Начать потоки.
mt1.Thrd.Start();
mt2.Thrd.Start();
mt1.Thrd.Join();
```

Упражнение51

1. Написать программу в которой запускаются три потока с приоритетами - высокий, выше среднего и нормальный соответственно.
2. В каждом потоке происходит подсчет до 10^9.
3. Вывести на консоль сообщения об запуске и остановке потоков и конечные значения счетчиков для каждого из потоков.

Синхронизация

Когда используется несколько потоков, то приходится координировать действия двух или более потоков. Процесс достижения такой координации называется *синхронизацией*. Самой распространенной причиной применения синхронизации служит необходимость разделять среди двух или более потоков общий ресурс, который может быть одновременно доступен только одному потоку. Например, когда в одном потоке выполняется запись информации в файл, второму потоку должно быть запрещено делать это в тот же самый момент времени.

Синхронизация требуется и в том случае, если один поток ожидает событие, вызываемое другим потоком. В подобной ситуации требуются какие-то средства, позволяющие приостановить один из потоков до тех пор, пока не произойдет событие в другом потоке. После этого ожидающий поток может возобновить свое выполнение.

В основу синхронизации положено понятие ***блокировки***, посредством которой организуется управление доступом к кодовому блоку в объекте. Когда объект заблокирован одним потоком, остальные потоки не могут получить доступ к заблокированному кодовому блоку. Когда же блокировка снимается одним потоком, объект становится доступным для использования в другом потоке.

Средство блокировки встроено в язык C#. Благодаря этому все объекты могут быть синхронизированы. Синхронизация организуется с помощью ключевого слова **lock**. Синхронизация объектов во многих программах происходит практически незаметно.

Общая форма блокировки:

```
lock (lockObj) {
// синхронизируемые операторы
}
```

где ***lockObj*** обозначает ссылку на синхронизируемый объект. Если же требуется синхронизировать только один оператор, то фигурные скобки не нужны. Оператор **lock** гарантирует, что фрагмент кода, защищенный блокировкой для данного объекта, будет использоваться только в потоке, получающем эту блокировку. А все остальные потоки блокируются до тех пор, пока блокировка не будет снята. Блокировка снимается по завершении защищаемого ею фрагмента кода.

```
class SumArray {    int sum;
object lockOn = new object(); //закрытый объект, доступный для
последующей блокировки
public int SumIt(int[ ] nums) {lock(lockOn) { // заблокировать весь метод
      sum = 0; // установить исходное значение суммы
   for(int i=0; i < nums.Length; i++) { sum += nums[i];
Console.WriteLine("Текущая сумма для потока"+Thread.CurrentThread.Name+"равна"+sum);
Thread.Sleep(10); } // разрешить переключение задач
return sum;  } }
}
class MyThread { public Thread Thrd;        int[] a;              int answer;
// Создать один объект типа SumArray для всех экземпляров класса MyThread.
static SumArray sa = new SumArray();
public MyThread(string name, int[] nums) {// Сконструировать новый поток,
a = nums;    Thrd = new Thread(this.Run);   Thrd.Name = name;
Thrd.Start(); }  // начать поток
void Run() {// Начать выполнение нового потока.
Console.WriteLine(Thrd.Name + " начат.");
answer = sa.SumIt(a);
Console.WriteLine("Сумма для потока " + Thrd.Name + " равна " + answer);
Console.WriteLine(Thrd.Name + " завершен.");
} }
////////////////////////////////////////////////////////////////////////////
int[] a = {1, 2, 3, 4, 5};
MyThread mt1 = new MyThread("Потомок #1", a);
MyThread mt2 = new MyThread("Потомок #2", a);
mt1.Thrd.Join();
mt2.Thrd.Join();
```

Сообщение между потоками с помощью методов `Wait()`, `Pulse()` и `PulseAll()`

Рассмотрим следующую ситуацию. Поток *T* выполняется в кодовом блоке `lock`, и ему требуется доступ к ресурсу *R*, который временно недоступен. Если поток *T* войдет в организованный в той или иной форме цикл опроса, ожидая освобождения ресурса *R*, то тем самым он свяжет соответствующий объект, блокируя доступ к нему других потоков. Это далеко не самое оптимальное решение, поскольку оно лишает преимуществ программирования для многопоточной среды. Более совершенное решение заключается в том, чтобы временно освободить объект и тем самым дать возможность выполняться другим потокам. Такой подход основывается на некоторой форме сообщения между потоками, благодаря которому один поток может уведомлять другой о том, что он заблокирован и что другой поток может возобновить свое выполнение. Сообщение между потоками организуется с помощью методов **Wait**(), **Pulse**() и **PulseAll**().

Методы **Wait**(), **Pulse**() и **PulseAll**() определены в классе **Monitor** и могут вызываться только из заблокированного фрагмента блока. Когда выполнение потока временно заблокировано, он вызывает метод **Wait**(). В итоге поток переходит в состояние ожидания, а блокировка с соответствующего объекта снимается, что дает возможность использовать этот объект в другом потоке.

В дальнейшем ожидающий поток активизируется, когда другой поток войдет в аналогичное состояние блокировки, и вызывает метод **Pulse**() или **PulseAll**(). При вызове метода **Pulse**() возобновляется выполнение первого потока, ожидающего своей очереди на получение блокировки. А вызов метода **PulseAll**() сигнализирует о снятии блокировки всем ожидающим потокам.

Две наиболее часто используемые формы метода **Wait**().

```
public static bool Wait(object obj)
public static bool Wait(object obj, int миллисекунд_простоя)
```

В первой форме ожидание длится вплоть до уведомления об освобождении объекта, а во второй форме — как до уведомления об освобождении объекта, так и до истечения периода времени, на который указывает количество *миллисекунд_простоя*.

В обеих формах ***obj*** обозначает объект, освобождение которого ожидается.

Общие формы методов **Pulse**() и **PulseAll**():

```
public static void Pulse(object obj)
public static void PulseAll(object obj)
```

где *obj* обозначает освобождаемый объект.

Если методы **Wait**(), **Pulse**() и **PulseAll**() вызываются из кода, находящегося за пределами синхронизированного кода, например из блока lock, то генерируется исключение SynchronizationLockException.

```
class TickTock {   object lockOn = new object();
public void Tick (bool running) {   lock(lockOn) {
if(!running) { // остановить часы
Monitor.Pulse(lockOn); return; }// уведомить любые ожидающие
потоки
Console.Write("тик ");
Monitor.Pulse(lockOn); // разрешить выполнение метода Tock()
Monitor.Wait(lockOn); } }  // ожидать завершения метода Tock()
public void Tock (bool running) {   lock(lockOn) {
if(!running) { // остановить часы
Monitor.Pulse(lockOn); return;} // уведомить любые ожидающие
потоки
Console.WriteLine("так");
Monitor.Pulse(lockOn); // разрешить выполнение метода Tick()
Monitor.Wait(lockOn); } } } // ожидать завершения метода Tick()

class MyThread {   public Thread Thrd;   TickTock ttOb;
// Сконструировать новый поток.
public MyThread (string name, TickTock tt) {
Thrd = new Thread(this.Run);                              ttOb =
tt;
Thrd.Name = name;
Thrd.Start(); }
void Run() {// Начать выполнение нового потока.
```

```
TickTock tt = new TickTock();
MyThread mt1 = new MyThread("Tick", tt);
MyThread mt2 = new MyThread("Tock", tt);
mt1.Thrd.Join();
mt2.Thrd.Join();
Console.WriteLine("Часы остановлены");
```

Упражнение52

1. Написать программу, которая запускает два независимых потока.
2. Первый поток выводит на консоль все четные числа от 10 до 0, второй все нечетные числа от 0 до 20.
3. По окончанию работы каждого из потока в конце выводится сообщение какой из потоков закончился.

FormsApp

Справочник: https://metanit.com/sharp/windowsforms/

TextBox

int x = 123;

textBox1.**Text**=x.ToString("f"); или

textBox1.**Text**=x.ToString();

- Если надо добавить к строке:

textBox1.**Text+**=“руб.”

- Если надо перейти на следующую строку:

textBox1.**Text+**=**Environment.NewLine;**

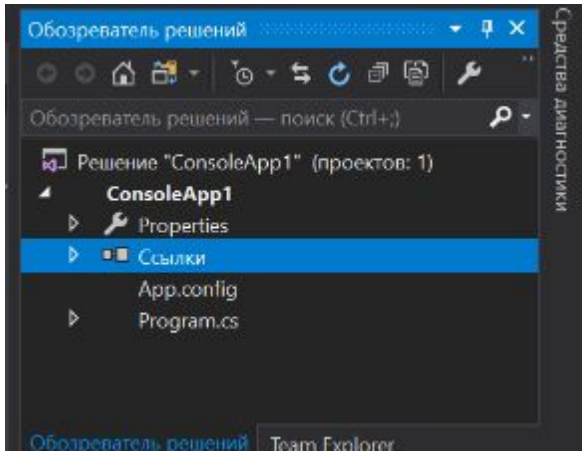
ListBox

Элемент ListBox представляет собой простой список. Ключевым свойством этого элемента является свойство **Items**, которое как раз и хранит набор всех элементов списка.

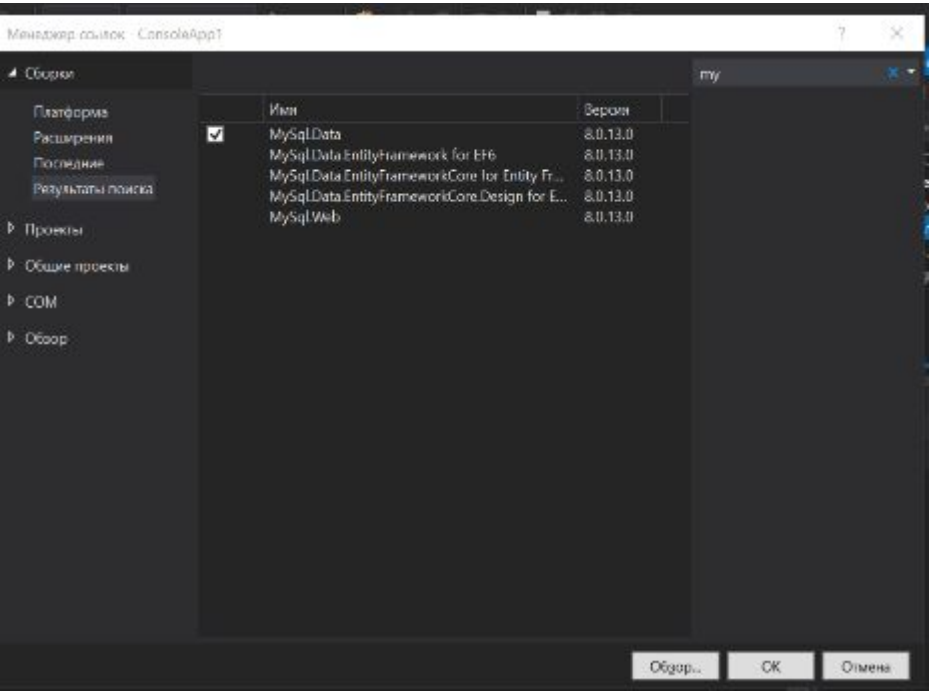
```
string[]countries={"Бразилия","Аргентина","Чили","Уругвай"};
listBox1.Items.AddRange(countries);

 for (float i = 0.1f; i < 20; i += 0.2f)
                listBox1.Items.Add(i);

listBox1.Items.Insert(1, "Парагвай");

listBox1.Items.Remove("Чили");
listBox1.Items.RemoveAt(1);
```

Очистить сразу весь список, применив метод Clear:

```
listBox1.Items.Clear();
```



1. Для работы с БД MySQL необходимо подключить библиотеку по ссылке.
2. Встать на **Ссылки** правой кнопкой выбрать: **Добавить ссылку…**
3. Выбрать **MySql.Data**.



4. В тексте программы добавить библиотеку using **MySql.Data.MySqlClient**;

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using MySql.Data.MySqlClient;
```

```
// готовим строку для подключения к БД
string connStr= "server=192.168.4.211; user=student; database=bd1; password=12345
//вариант:"server=192.168.4.211; user=student; database=bd1;password=123456;SslMode=non
// conn – экземпляр класса подключения к БД
MySqlConnection conn = new MySqlConnection(connStr);
// устанавливаем соединение с БД
conn.Open();
// запрос MySql
string sql = "SELECT id, name FROM tab1 WHERE $ >= 12";
// объект для выполнения SQL-запроса
MySqlCommand command = new MySqlCommand(sql, conn);
// объект для чтения ответа сервера
MySqlDataReader reader = command.ExecuteReader();
// читаем результат метод Read
while (reader.Read())
{// элементы массива [] - это значения столбцов из запроса SELECT здесь их два
Console.WriteLine(reader[0].ToString() + " " + reader[1].ToString());
}                reader.Close(); // закрываем reader

// Добавление новой записи в таблицу
sql = "INSERT INTO `bd1`.`tab1` (`id`,`name`,`$`) VALUES('15', 'Petr', '230')";

command = new MySqlCommand(sql, conn);
// проверяем сколько было изменений в БД
int iz=command.ExecuteNonQuery();
Console.WriteLine(iz);
conn.Close(); // закрываем соединение с БД
```

Упражнение

1. Создать базу данных в MySql, в ней создать таблицу с полями: порядковый номер (ключевое поле), ФИО, сумма (денежная единица).
2. Заполнить таблицу несколькими записями.
3.Написать программу подключения к БД и вывести в ListBox содержимое таблицы.

```
SELECT * FROM db1.tab1;
```

Простые сомножители

```
int k, x = Int32.Parse(Console.ReadLine());
        for (int i = 2; i < x; i++)
        {
           if (x % i == 0 & (i == 2 | i == 3 )) { Console.Write(i + " "); continue; }
           if (x % i == 0)
           {
                    for (k = 2; k < i; k++) if (i % k == 0) break;
              if (k == i) Console.Write(i + " ");
           }          }
```

Вхождение s0 in s **16**

```
public int CountWords(string s, string s0) { int count = (s.Length - s.Replace(s0, "").Length) / s0.Length;
return count; }
```

Сортировка

```
int tmp, N = размер массива;

   for (int k = 1; k < N; k++)
        {            for (int i = 0; i <N - k; i++)
             if (m[i] > m[i + 1]) {
                                  tmp = m[i + 1];
                                  m[i + 1] = m[i];
                                  m[i] = tmp;                                  }        }
```

2. qsort

```
int partition (int[] array, int start, int end)
   {
       int marker = start;
       for ( int i = start; i <= end; i++ )
       {
           if ( array[i] <= array[end] )
           {
               int temp = array[marker]; // swap
               array[marker] = array[i];
               array[i] = temp;
               marker += 1;
           }
       }
       return marker - 1;
   }

   void quicksort (int[] array, int start, int end)
   {
       if ( start >= end )
       {
           return;
       }
       int pivot = partition (array, start, end);
       quicksort (array, start, pivot-1);
       quicksort (array, pivot+1, end);
   }
```

```
string text = "hello world";
  // запись в файл
        using (FileStream fstream = new FileStream(@"D:\note.dat", FileMode.OpenOrCreate))
 { // преобразуем строку в байты
            byte[] input = Encoding.Default.GetBytes(text);
// запись массива байтов в файл
            fstream.Write(input, 0, input.Length);
            Console.WriteLine("Текст записан в файл");
 // перемещаем указатель в конец файла, до конца файла- пять байт
            fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца потока
 // считываем четыре символов с текущей позиции
            byte[] output = new byte[4];
            fstream.Read(output, 0, output.Length);
 // декодируем байты в строку
            string textFromFile = Encoding.Default.GetString(output);
            Console.WriteLine("Текст из файла: {0}", textFromFile); // worl
 // заменим в файле слово world на слово house
            string replaceText = "house";
            fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца потока
            input = Encoding.Default.GetBytes(replaceText);
            fstream.Write(input, 0, input.Length);
// считываем весь файл возвращаем указатель в начало файла
            fstream.Seek(0, SeekOrigin.Begin);
            output = new byte[fstream.Length];
            fstream.Read(output, 0, output.Length);
// декодируем байты в строку
            textFromFile = Encoding.Default.GetString(output);
            Console.WriteLine("Текст из файла: {0}", textFromFile);   }
Console.Read();    }
```

```
class prog
{
   static void Main()
   {
      Complex a,b,c;
      a = new Complex(3, 5); b = new Complex(-2, -5);
      c = a * b;
      c.Show();            Console.Read();
   }
}

class Complex
{
   double re,im ; // двухмерные координаты
   public Complex() { re = im = 0; }
   public Complex(double re, double im) { this.re = re; this.im=im; }
   // Перегрузить бинарный оператор +.
   public static Complex operator +(Complex A, Complex B)
   {
      Complex result = new Complex();
      /* Сложить координаты двух точек и возвратить результат. */
      result.re = A.re + B.re; // Эти операторы выполняют
      result.im = A.im + B.im; // целочисленное сложение,
      return result;
   }
   public static Complex operator *(Complex A, Complex B)
   {
      Complex result = new Complex();
      /*  */
      result.re = A.re * B.re-A.re*B.im; // Эти операторы выполняют
      result.im = A.re*B.im + A.im*B.re; // целочисленное сложение,
      return result;
   }
   // Вывести координаты X, Y.
   public void Show()
   {
      Console.WriteLine("Re ="+re + " Im = " + im );}
```

//Имеются сведения об индексе популярности: Париж:29; Берлин:25; Варшава:20; Лондон:29; Рим:29; Познань:20; Хельсенки:25; Осло:20; Стокгольм:25; Флоренция:29; Антверпен:20.

//Создать запрос и вывести в столбик список объектов в порядке убывания по названию.

//Создать запрос групирующий список с одинаковыми индексами популярности.

```
string[] Pop = { "Париж 29", "Берлин 25", "Варшава 20", "Лондон 29", "Рим 29", "Познань 20", "Хельсенки 25", "Осло 20", "Стокгольм 25", "Флоренция 29", "Антверпен 20"};
        Console.WriteLine("По популярности");
        var popular = from gorod in Pop
                      where gorod.LastIndexOf(' ') != -1
                      group gorod by gorod.Substring(gorod.LastIndexOf(' '));
        foreach (var aa in popular)
        {
           Console.WriteLine();
           var pos = from n in aa
                     orderby n descending
                     select n;
           foreach (var aaa in pos) {
              Console.WriteLine(aaa); }
        }
        Console.Read();
```

https://server.179.ru/tasks/olymp/

14*

```
Random rr = new Random();
 int[][] s = new int[3][]; s[0] = new int[2]; s[1] = new int[3]; s[2] = new int[4];
 for (int i = 0; i < 3; i++)
 for (int j = 0; j < s[i].Length; j++)  s[i][j] = rr.Next()/10000;
                                foreach (var x in s) { Console.WriteLine(String.Join( "   ",x)); }
          Console.ReadKey();
```

15

```
string sourcestring = "abcd1234abcd";
string substring = "abcd";
int count = (sourcestring.Length - sourcestring.Replace(substring,  "").Length)/substring.Length;
```

```
 public int fac(int i)//factorial
         {   int f = 1;
             if (i != 1)                  f = fac(i - 1) * i;
              return f;
         }
         public int fac_it(int a)
         {   int fac = 1;
             for (int i = 1; i <= a; i++)   fac *= i;
             return fac;
         }
```

20*1. Написать программу с перегрузкой арифметических операторов +, -, *, / так, чтобы при использовании этих операторов в программе на экран выводилась строка, например:

4-10=-6

2. С консоли вводятся первое число потом арифметический оператор затем второе число и при нажатии символа = на экран выводится строка результата.

23*

```
Class Timer
{   public int Hours { get; set; }
    public int Minutes { get; set; }
    public int Seconds { get; set; }}
class Counter
{    public int Seconds { get; set; }
     public static implicit operator Counter(int x){
        return new Counter { Seconds = x };    }
    public static explicit operator int(Counter counter)
    {        return counter.Seconds;    }
    public static explicit operator Counter(Timer timer)
    {   int h = timer.Hours * 3600;
        int m = timer.Minutes * 60;
        return new Counter { Seconds = h + m + timer.Seconds };    }
    public static implicit operator Timer(Counter counter)
    {   int h = counter.Seconds / 3600;
        int m = (counter.Seconds % 3600) / 60;
        int s = counter.Seconds % 60;
        return new Timer { Hours = h, Minutes = m, Seconds = s };
    }
//-----------------------------------------
Counter counter1 = new Counter { Seconds = 115 };
     Timer timer = counter1;
    Console.WriteLine($"{timer.Hours}:{timer.Minutes}:{timer.Seconds}"); // 0:1:55
     Counter counter2 = (Counter)timer;
    Console.WriteLine(counter2.Seconds);  //115
```

считать сразу 4 числа через пробел в одной строке.

```
string[] nums_strings = Console.ReadLine().Split(); int[] nums = new int[nums_strings.Length]; for(int i = 0; i < nums_strings.Length; i++)     nums[i] = Convert.ToInt32(nums_strings[i]); Console.WriteLine(nums);
```

```
string str = "5;10;7;52";//ввели с консоли несколько чисел
        var array = str.Split(';').Select(int.Parse).ToArray();
        foreach (var item in array)          {              Console.Write(item+" ");          }
```