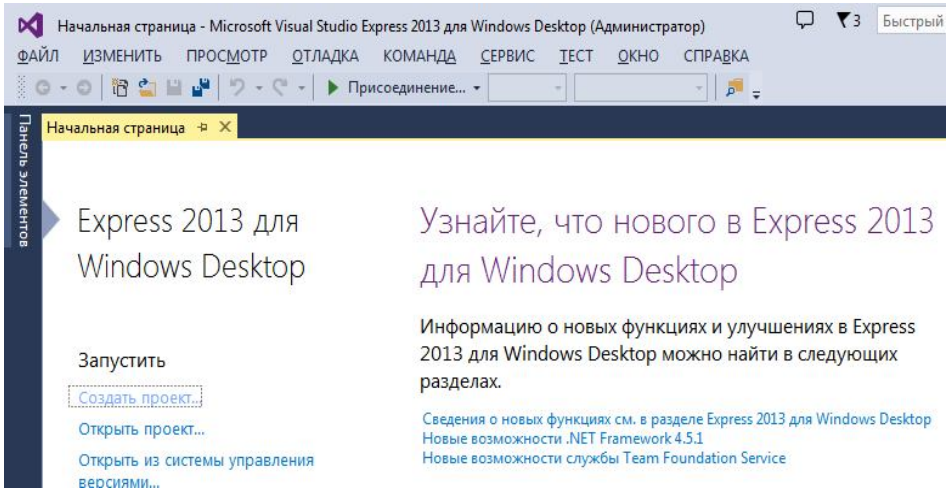
Оглавление

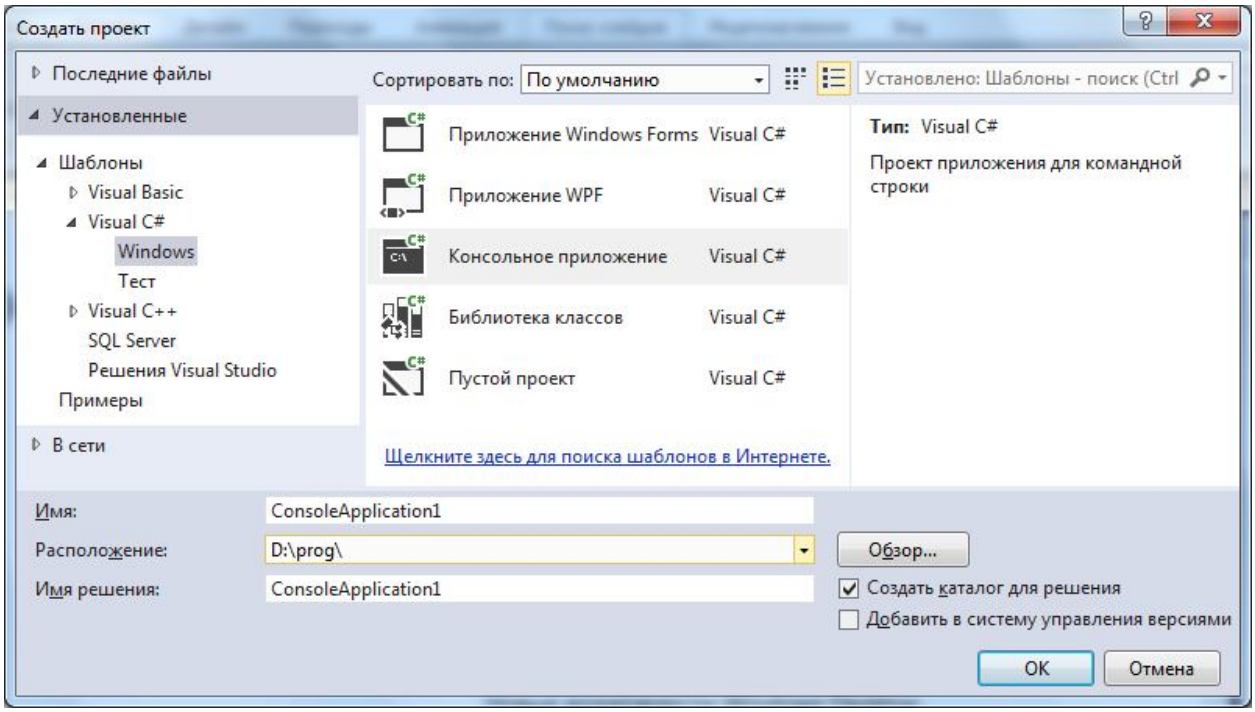
1. Создание консольного проекта. Преобразование типов

1.1. Первая программа на C Sharp

Пишем первую программу на C#. Запускайте Visual Studio.NET. Для создания нового пустого проекта C# на Начальной странице выбираем Создать проект (Create Project), или нажимаем комбинацию клавиш Ctrl+Shift+N, или просто заходим в меню ФАЙЛ (FILE) и далее выбираем Создать проект… (Create Project).



В появившемся окне "Создать проект" слева выбираем, естественно, Visual C# - Windows, а справа тип приложения - Console Application:



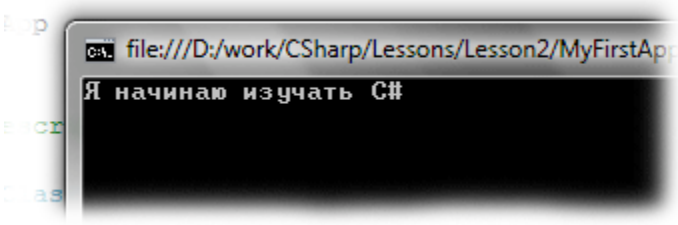
В качестве имени проекта (Имя - Name) напечатайте MyFirstApp или что-то в этом роде. Укажите нужно расположение. Нажмите кнопку Ok для закрытия данного диалогового окна.

Теперь приступаем к коду. Наша первая программа просто выведет некоторое фиксированное слово в консольное окошко. Вот ее листинг.

Пример1.

```
using System;
namespace first
{
    ///
    /// Summary description for MyFirstClass.
    ///
    class MyFirstClass
    {
        ///
        /// The main entry point for the application.
        ///
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //
            Console.WriteLine("Я начинаю изучать C#");
        }
    }
}
```

Запускаем программу, нажав **Ctrl+F5**. Результат будет таким:



Разберем текст программы поподробнее. В .NET Runtime существуют пространства имен, содержащие наборы типов, классов и методов. Одно из таких пространств - это System. Оно добавляется автоматически в любой проект на C#. Так как мы добавили в нашу программу строчку

```
using System;
```

то мы можем вместо длинных имен использовать более короткие. В частности, вместо System.Console можно писать просто Console. Что мы делаем в строчке

```
            Console.WriteLine("Я начинаю изучать C#");
```

Далее мы в нашей программе объявляем класс MyFirstClass. Что такое классы мы посмотрим в дальнейшем. Сейчас же достаточно сказать, что в C# не существует глобальных функций.

Функция Main обязательно должна быть в каждой программе на C#, и именно с этой функции и начинается выполнение программы. Обратите также внимание, что эта функция пишется с прописной (большой) буквы. C# различает строчные и прописные буквы, так что это важно. Кроме того, эта функция объявлена с модификатором static. Это означает, что она не относится к конкретному экземпляру класса MyFirstClass, а принадлежит всему классу. В нашей функции Main мы просто выводим на экран некоторую строчку методом *WriteLine*.

Как вы обратили внимание, программа выполнилась и быстро закрыла свое окно. Чтобы такого не произошло, добавим еще одну строку в нашу программу:

```
            Console.WriteLine("Я начинаю изучать C#");
            Console.ReadLine();
```

ReadLine, как видно из названия, считывает строку.

Теперь после запуска окно остается, и программа ждет ввода. Для завершения ее работы требуется нажать Enter.

1.2. Выражения, инструкции и разделители

1.2.1. Выражения (Expressions)

Выражение — это строка кода, которая определяет значение. Пример простого выражения:

```
myValue = 100;
```

Обратите внимание, что данная инструкция выполняет присвоение значения 100 переменной myValue. Оператор присвоения (=) не сравнивает стоящее справа от него значение (100) и значение переменной, которая находится слева от оператора (myValue). Оператор «=» устанавливает значение переменной myValue равным 100.

Поскольку myValue = 100, то myValue может использоваться другим оператором присвоения, как выражение, которое определяет значение 100.

Например:

```
mySecondValue = myValue = 100;
```

В данном выражении литеральное значение 100 присваивается переменной myValue, а затем оператором присвоения устанавливается вторая переменная mySecondValue с тем же значением 100. Таким образом, значение 100 будет присвоено обеим переменным одновременно. Инструкцией такого вида вы можете инициализировать любое число переменных с одним и тем же значением, например 20:

a = b = c = d = e = 20;

1.2.2. Инструкции (Statements)

Инструкция — это законченное выражение в коде программы. Программа на языке C# состоит из последовательностей инструкций. Каждая инструкция обязательно должна заканчиваться точкой с запятой (;). Например:

```
int x; // инструкция
x = 100; //другая инструкция
int y = x; //тоже инструкция
```

Кроме того, в C# существуют составные инструкции. Они, в свою очередь, состоят из набора простых инструкций, помещенных в фигурные скобки { }.

```
{
int x; // инструкция
x = 100; //другая инструкция
int y = x; //тоже инструкция
}
```

C# инструкции рассматриваются в соответствии с порядком их записи в тексте программы. Компилятор начинает рассматривать код программы с первой строки и заканчивает концом файла.

1.2.3. Разделители (Delemiters)

В языке C# пробелы, знаки табуляции и переход на новую строку рассматриваются как разделители. В инструкциях языка C# лишние разделители игнорируются. Таким образом, вы можете написать:

```
myValue = 100;
```

или:

```
myValue  =  100;
```

Компилятор обработает эти две инструкции как абсолютно идентичные. Исключение состоит в том, что пробелы в пределах строки не игнорируются. Если вы напишете:

```
Console.WriteLine("Я изучаю C# !");
```

каждый пробел между словами «Я», «изучаю», «C#»и знаком «!» будет обрабатываться как отдельный символ строки.

В большинстве случаев использование пробелов происходит чисто интуитивно. Обычно они применяются для того, чтобы сделать программу более читаемой для программиста; для компилятора разделители абсолютно безразличны.

Надо заметить, что есть случаи, в которых использование пробелов является весьма существенным. Например, выражение:

```
int myValue = 25;
```

то же самое, что и выражение:

```
int myValue=25;
```

но следующее выражение не будет соответствовать двум предыдущим:

```
intmyValue =25;
```

Компилятор знает, что пробел с обеих сторон оператора присвоения игнорируется (сколько бы много их не было), но пробел между объявлением типа int и именем переменной myValue должен быть обязательно.

Это не удивительно, пробелы в тексте программы позволяют компилятору находить и анализировать ключевые слова языка. В данном случае это int, а некоторый термин intmyValue для компилятора неизвестен. Вы можете свободно добавлять столько пробелов, сколько вам нравится, но между int и myValue должен быть, по крайней мере, один символ пробела или табуляции.

1.3. ВСТРОЕННЫЕ ТИПЫ

Язык C# предоставляет программисту широкий спектр встроенных типов, которые соответствуют CLS (Common Language Specification) и отображаются на основные типы платформы .NET. Это гарантирует, что объекты, созданные на C#, могут успешно использоваться наряду с объектами, созданными на любом другом языке программирования, поддерживающем .NET CLS (например, VB.NET).

Каждый тип имеет строго заданный для него размер, который не может изменяться. В отличие от языка C++, в C# тип int всегда занимает 4 байта, потому что отображается к Int32 в .NET CLS. Представленная ниже таблица содержит список всех встроенных типов, предлагаемых C#.

Тип	Область значений	Размер
sbyte	-128 до 127	Знаковое 8-бит целое
byte	0 до 255	Беззнаковое 8-бит целое
char	U+0000 до U+ffff	16-битовый символ Unicode
bool	true или false	1 байт
short	-32768 до 32767	Знаковое 16-бит целое
ushort	0 до 65535	Беззнаковое 16-бит целое
int	-2147483648 до 2147483647	Знаковое 32-бит целое
uint	0 до 4294967295	Беззнаковое 32-бит целое
long	-9223372036854775808 до 9223372036854775807	Знаковое 32-бит целое
ulong	0 до 18446744073709551615	Беззнаковое 32-бит целое
float	$\pm 1{,}5*10^{-45}$ до $\pm 3{,}4*10^{33}$	4 байта, точность – 7 разрядов
double	$\pm 5*10^{-324}$ до $\pm 1{,}7*10^{306}$	8 байт, точность – 16 разрядов
decimal		12 байт, точность – 28 разрядов

В дополнение к этим примитивным типам C# может иметь объекты типа enum и struct.

1.3.1. Преобразование встроенных типов

Объекты одного типа могут быть преобразованы в объекты другого типа неявно или явно. Неявные преобразования происходят автоматически, компилятор делает это вместо вас. Явные преобразования осуществляются, когда вы «приводите» значение к другому типу. Неявные преобразования гарантируют также, что данные не будут потеряны. Например, вы можете неявно приводить от short (2 байта) к int (4 байта).

Независимо от того, какой значение находится в short, оно не потеряется при преобразовании к int:

```
short x = 1;

int y = x; //неявное преобразование
```

Если вы делаете обратное преобразование, то, конечно же, можете потерять информацию. Если значение в int больше, чем 32.767, оно будет усечено при преобразовании. Компилятор не станет выполнять неявное преобразование от int к short:

```
short x;

int y = 5;

x = y; //не скомпилируете
```

Вы должны выполнить явное преобразование, используя оператор приведения:

```
short x;

int y = 5;

x = (short) y;//OK
```

1.4. Переменные

Переменная — это расположение в памяти объекта определенного типа. В приведенных выше примерах x и y — переменные. Переменные могут иметь значения, которыми они проинициализированы, или эти значения могут быть изменены программно.

1.4.1. Назначение значений переменным.

Чтобы создать переменную, вы должны задать тип переменной и затем дать этому типу имя. Вы можете проинициализировать переменную во время ее объявления или присвоить ей новое значение во время выполнения программы. Вот пример программы, который в первом случае использует инициализацию для присвоения значения переменной, во втором случае использует присвоение значения переменной с помощью оператора «=»:

Пример 2.

```
class Variables
{
   static void Main()
   {
       int myInt = 10;
       System.Console.WriteLine("Инициализированная переменная myInt: {0} ", myInt);
```

```
            myInt = 5;
            System.Console.WriteLine("Переменная myInt после присвоения значения: {0}",
myInt);
            Console.ReadLine();
        }
}
```

Результат работы этой программы будет следующий:

Инициализированная переменная myInt: 10

myInt после присвоения значения: 5

Здесь строка:

int myInt = 10;

означает объявление и инициализацию переменной myInt. Строка:

myInt= 5 ;

означает присвоение переменной myInt значения 5.

1.4.2. Определение значений переменных

C# требует определения значений, то есть переменные перед использованием должны быть инициализированы. Чтобы проверить это правило, рассмотрим следующий пример.

Пример 3.

```
class Variables
{
    static void Main()
    {
      int myInt;
      System.Console.WriteLine("Неинициализированная переменная myInt: {0}", myInt);
      myInt = 5;
      System.Console.WriteLine("После присвоения переменной myInt: {0}", myInt);
    }
}
```

Если вы попробуете скомпилировать этот пример, компилятор отобразит следующее сообщение об ошибке:

error CS3165: Use of unassigned local variable 'myInt'

Нельзя использовать неинициализированную переменную в C#. У вас может сложиться впечатление, что вы должны инициализировать каждую переменную в программе. Это не так. Вы должны лишь назначить переменной значение прежде, чем попытаетесь ее использовать. Ниже представлен пример правильной программы без использования инициализации переменной

Пример 4.

```
class Variables
    {
        static void Main()
        {
```

```
            int myInt;
            myInt = 10;
            System.Console.WriteLine("Инициализированная переменная myInt: {0}",
myInt);
            myInt = 5;
            System.Console.WriteLine("После присвоения значения, myInt: {0}", myInt);

        }
    }
```

В данном примере вместо инициализации выбирается присвоение значения переменной myInt до ее использования:

myInt = 10;

1.4.3. Константы

Константа — это переменная, значение которой не может быть изменено. Переменные — это более гибкий способ хранения данных. Однако иногда вы хотите гарантировать сохранение значения определенной переменной.

Например, число pi. Как известно, значение этого числа никогда не изменяется. Следовательно, вы должны гарантировать, что переменная, хранящая это число, не изменит своего значения на протяжении всей работы программы. Ваша программа будет лучше читаемой, если вы вместо записи:

y = x*3.14159265358979323846264338327 95

будете использовать переменную, которая хранит значение pi. В таком случае используемой переменной должна быть константа:

const double pi = 3.14159265358979323846264338327 95;

y = x * pi;

Существует три разновидности константы: литералы, символические константы и перечисления. Рассмотрим следующий случай:

x = 100;

Значение 100 — это литеральная константа. Значение 100 —это всегда 100. Вы не можете установить новое значение на 100. Вы не можете сделать так, чтобы 100 представляло значение 99. Символические константы устанавливают имя для некоторого постоянного значения. Вы объявляете символическую константу, используя ключевое слово const, и применяете следующий синтаксис для создания константы:

const тип идентификатор = значение;

Константа обязательно должна быть проинициализирована, и ее значение не может быть изменено во время выполнения программы. Например:

const double pi = 3.14159265358979323846264338327 95;

pi = 5.0; //недопустимая операция

В этом примере число 3.14...—литеральная константа, а pi — символическая константа типа double. Ниже представлен пример использования символических констант.

Пример 5.

```
class Constants
    {
        static void Main ()
        {
            const double pi = 3.14159265358979323846264338327795;
            const double g = 9.81; //гравитационная постоянная
            System.Console.WriteLine("Число пи: {0}" ,pi);
            System.Console.WriteLine("Гравитационная постоянная: {0}",g);
        }
    }
```

Результат работы программы будет следующий:

Число пи: 3.14159265358979323846264338327795

Гравитационная постоянная: 9.81

В последнем примере создаются две символические целочисленные константы: pi и д. Эти константы преследуют ту же цель, что и использование их литеральных значений. Но данные константы имеют имена, которые несут в себе гораздо больше информации об используемом значении, чем просто набор цифр.

Для того чтобы убедиться, что константные значения не могут быть изменены во время выполнения программы, добавьте в код следующую строку:

g = 10;

Во время компиляции вы получите следующее сообщение об ошибке:

error CS0131: The left-hand side of an assignment must be a variable, property or indexer.

1.4.4. Перечисления

Перечисления являются мощной альтернативой константам. Это особый тип значений, который состоит из набора именованных констант.

Допустим, у вас есть список констант, содержащих годы рождения ваших знакомых. Для того чтобы запрограммировать это при помощи констант, вам придется написать:

```
const int maryBirthday = 1955;

const int ivanBirthday = 1980;

const int pavelBirthday = 1976;
```

Получились три совершенно несвязанные константы. Для того чтобы установить логическую связь между ними, в C# предусмотрен механизм перечислений. Вот как выглядит тот же код, записанный при помощи перечислений:

```
enum FriendsBirthday

{
```

```
const int maryBirthday = 1955;

const int ivanBirthday = 1980;

const int pavelBirthday = 1976;

}
```

Теперь три символические константы являются элементами одного перечисления типа FriendsBirthday.

Каждое перечисление имеет свой базовый тип, которым может быть любой встроенный целочисленный тип C# (int, long, short и т. д.), за исключением char.

Перечисление задается следующим образом:

[атрибуты] [модификаторы] enum идентификатор[: базовый тип] (список перечислений);

Атрибуты и модификаторы будут рассматриваться далее. Пока давайте остановимся на второй части этого объявления. Перечисление начинается с ключевого слова enum, которое сопровождается идентификатором типа:

enum MyEnurnerators

Базовый тип — основной тип для перечисления. Если вы не учитываете этот описатель при создании перечисления, то будут использоваться значения по умолчанию int. Но вы можете применить любой из целочисленных типов (например, ushort, long), за исключением char. Например, следующий фрагмент кода объявляет перечисление целых чисел без знака (uint):

```
enum Sizes: uint

{

Small = 1,

Middle = 2,

Large = 3

}
```

Внутри каждого перечисления записывается список возможных значений перечисления, разделенных запятой. Каждое значение может представлять собой либо просто набор символических констант, либо набор символических констант в сочетании с литеральным целочисленным значением. Если вы не укажете для элементов перечисления целочисленных значений, то компилятор пронумерует их сам, начиная с 0. Например, следующие фрагменты кода аналогичны:

```
enum Sizes: uint

{

Small,
```

```
Middle,

Large

}
```

Если вы объявите свое перечисление следующим образом:

```
enum Sizes: uint

{

Small,

Middle = 20,

Large

}
```

то элементы перечисления будут иметь такие числовые значения:

Small = 0;

Middle = 20;

Large = 21.

Давайте рассмотрим пример, который наглядно показывает, каким образом перечисления помогают упростить код приложения.

Пример 5.

```
class ScreenResolutions
{
//перечисление размеров мониторов в дюймах
    enum Screens
        {
            Small = 14,
            Medium = 17,
            Large = 19,
            SuperLarge = 21
        }
    static void Main()
        {
            System.Console.WriteLine("Самые маленькие мониторы имеют размер: {0}",
(int) Screens.Small );
            System.Console.WriteLine("Самые большие мониторы имеют размер: {0}",
(int) Screens.SuperLarge);
        }
   }
```

Как видите, значение перечисления (SuperLarge) должно быть специфицировано именем перечисления (Screens). По умолчанию, значение перечисления представляется его символическим именем (типа Small или Medium). Если вы хотите отобразить значение константы перечисления, то должны привести константу к ее основному типу (в данном случае int).

Целочисленное значение передается в функцию WriteLine, и оно отображается на экране.

1.5. Консольный ввод-вывод

Пример 6: Методы вывода

```
using System;
namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            int i = 3;
            double y = 4.12;
            decimal d = 600m;
            string s = "Вася";

            Console.WriteLine("i = " + i);                  // 1
            Console.WriteLine("s = " + s);                  // 2
            Console.WriteLine("y = {0} \nd = {1}", y, d);   // 3
        }
    }
}
```

Результат работы программы:

```
i = 3
s = Вася
y = 4,12
d = 600
```

в классе `Console` существует несколько вариантов методов с именами `Write` и `WriteLine`, предназначенных для вывода значений различных типов.

Методы с одинаковыми именами, но разными параметрами называются *перегруженными*. Компилятор определяет, какой из методов вызван, по типу передаваемых в него величин. Методы вывода в классе `Console` перегружены для всех встроенных типов данных, кроме того, предусмотрены варианты форматного вывода.

Пример содержит два наиболее употребительных варианта вызова методов вывода. Если метод `WriteLine` вызван с *одним* параметром, он может быть любого встроенного типа. В строке, помеченной комментариями "1" и "2", нам требуется вывести в каждой строке не одну, а две величины, поэтому прежде чем передавать их для вывода, их требуется "склеить" в одну строку с помощью операции `+`.

Перед объединением строки с числом надо преобразовать число из его внутренней формы представления в последовательность символов, то есть в строку. *Преобразование в строку* определено во всех стандартных классах C# — для этого служит метод `ToString()`. В данном случае он выполняется неявно.

Оператор 3 иллюстрирует форматный вывод. В этом случае используется другой вариант метода `WriteLine`. Первым параметром методу передается строковый литерал, содержащий, помимо обычных символов, предназначенных для вывода на консоль,

параметры в фигурных скобках, а также *управляющие последовательности*. Параметры нумеруются с нуля, перед выводом они заменяются значениями соответствующих переменных в списке вывода.

Из управляющих последовательностей чаще всего используются символы перевода строки (`\n`) и горизонтальной табуляции (`\t`).

Рассмотрим простейшие способы *ввода с клавиатуры*. В классе `Console` определены методы ввода строки и отдельного символа, но нет методов, которые позволяют непосредственно считывать с клавиатуры числа. Ввод числовых данных выполняется в два этапа:

1. Символы, представляющие собой число, вводятся с клавиатуры в строковую переменную.
2. Выполняется преобразование из строки в переменную соответствующего типа.

Преобразование можно выполнить либо с помощью специального класса `Convert`, определенного в пространстве имен `System`, либо с помощью метода `Parse`, имеющегося в каждом стандартном арифметическом классе. В следующем примере используются оба способа.

Пример 7: Ввод с консоли

```
using System;
namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            Console.WriteLine("Введите строку");
            string s = Console.ReadLine();                  // 1
            Console.WriteLine("s = " + s);

            Console.WriteLine("Введите символ");
            char c = (char)Console.Read();                  // 2
            Console.ReadLine();                             // 3
            Console.WriteLine("c = " + c);

            string buf;        // строка – буфер для ввода чисел

            Console.WriteLine("Введите целое число");
            buf = Console.ReadLine();
            int i = Convert.ToInt32(buf);                 // 4
            Console.WriteLine(i);

            Console.WriteLine("Введите вещественное число");
            buf = Console.ReadLine();
            double x = Convert.ToDouble(buf);             // 5
            Console.WriteLine(x);

            Console.WriteLine("Введите вещественное число");
            buf = Console.ReadLine();
            double y = double.Parse(buf);                 // 6
            Console.WriteLine(y);
```

```
            Console.WriteLine("Введите вещественное число");
            buf = Console.ReadLine();
            decimal z = decimal.Parse(buf);              // 7
            Console.WriteLine(z);
        }
    }
}
```

Ввод строки выполняется в операторе 1. Длина строки не ограничена, ввод выполняется до символа перевода строки.

Ввод символа выполняется с помощью метода `Read`, который считывает один символ из входного потока (оператор 2). Метод возвращает значение типа `int`, представляющее собой код символа, или –1, если символов во входном потоке нет (например, пользователь нажал клавишу **Enter**). Поскольку нам требуется не `int`, а `char`, а неявного преобразования от `int` к `char` не существует, приходится применить операцию явного преобразования типа.

Оператор 3 считывает остаток строки и никуда его не передает. Это необходимо потому, что ввод данных выполняется через *буфер* — специальную область оперативной памяти. Фактически данные сначала заносятся в буфер, а затем считываются оттуда процедурами ввода. Занесение в буфер выполняется по нажатию клавиши **Enter** вместе с ее кодом. Метод `Read`, в отличие от `ReadLine`, не очищает буфер, поэтому следующий после него ввод будет выполняться с того места, на котором закончился предыдущий.

В операторах 4 и 5 используются методы класса `Convert`, в операторах 6 и 7 — методы `Parse` классов `Double` и `Decimal` библиотеки .NET, которые используются здесь через имена типов C# `double` и `decimal`.

Источники:

1. Изучаем C Sharp (C#). Программирование на C Sharp (C#) с нуля. Режим доступа: http://simple-cs.ru/
2. Павловская Т.А. Программирование на C#. Учебный курс. Режим доступа: http://ips.ifmo.ru/courses/csharp/index.html