Урок №3

Содержание

18. Перегрузка логических операторов
19. Перегрузка операторов преобразования
20. Что такое свойства.
21. Синтаксис объявления свойств.
22. Примеры использования свойств.
23. Понятие индексатора.
24. Создание одномерного индексатора.
25. Создание многомерных индексаторов.
26. Перегрузка индексаторов.

1. Иерархия исключений

Код, не содержащий ошибок – это идеал, который редко встречается на практике. Существует даже специальная область в теории надежности, предназначенная для теоретической оценки количества ошибок в проекте в зависимости от количества модулей, используемой технологии разработки и других факторов.

Возникновение ошибок при выполнении приложения - не всегда следствие ошибок в коде приложения. Ошибки могут быть вызваны неправильными действиями пользователя или внешними причинами такими как аппаратные сбои, недоступность некоторых ресурсов (например, сетевого диска или сервера базы данных).

Таким образом, профессионально разработанное приложение должно быть готово к возникновению ошибок и должно обеспечивать их обработку.

Для обработки ошибок можно использовать различные подходы. В WinAPI многие функции при неуспешном выполнении возвращают признак ошибки (например, FALSE, INVALID_HANDLE_VALUE, NULL). Далее с помощью функции GetLastError() можно получить код ошибки и найти по этому коду ее описание. Такой способ обработки ошибок является трудоемким, т.к. после вызова функции надо проверять

результат возврата, и не вполне надежным, т.к. можно продолжить работу без проверки результата, так как будто функция завершилась успешно.

В .Net Framework способ обработки ошибок значительно улучшен. Все методы при неуспешном выполнении генерируют исключения. Этот подход имеет следующие преимущества:

- ✓ Обработку ошибок можно отделить от основной логики работы программы: всю обработку ошибок можно сосредоточить в одном блоке, а не выполнять проверку после каждого вызова метода.

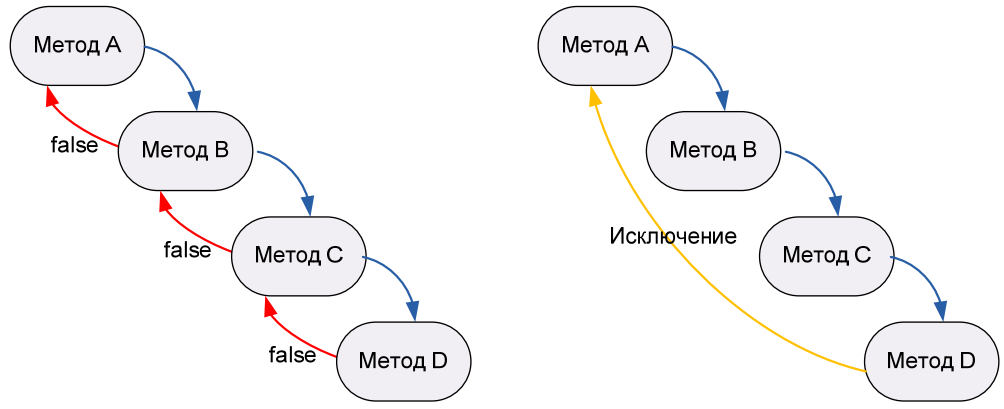


Рис. 1. Передача информации о возникновении ошибки по цепочке вызовов

- ✓ Возможна ситуация, когда имеется цепочка вызовов (рис 1.), и в последнем методе цепочки возникает ошибка. Без обработки исключений каждый из методов должен вернуть вызывающему методу код ошибки, в случае использования исключений при возникновении ошибки управление будет сразу передано методу, содержащему обработчик исключения.
- ✓ Исключение нельзя проигнорировать, т.к. необработанное исключение приведет к аварийному завершению программы.

Базовый класс System.Exception

В C# исключение является объектом, который создается и «выбрасывается» (throw) в случае возникновения ошибки. CLR позволяет генерировать исключения любого типа, например Int32, String и др. CLS-совместимый язык должен быть способен генерировать и перехватывать типы исключений, производные от базового класса SystemException. Эти исключения называются CLS-совместимыми. Такое исключение несет дополнительную информацию об ошибке, которая облегчает отладку программы.

Свойства класса System.Exception:

Название свойства	Описание
string Message	Содержит текст сообщения с указанием причины возникновения исключения.
IDictionary Data	Ссылка на набор пар «параметр-значение». Обычно код, генерирующий исключение, добавляет записи в этот набор. Код, перехвативший исключение, может использовать эти данные для получения дополнительной информации о причине возникновения исключения.
string Source	Содержит имя сборки, сгенерировавшей исключение.
string StackTrace	Содержит имена и сигнатуры методов, вызов которых привел к возникновению исключения.
MethodBase TargetSite	Содержит метод, сгенерировавший исключение.
string HelpLink	Содержит URL документа с описанием исключения
Exception InnerException	Указывает предыдущее исключение, если текущее было сгенерировано при обработке предыдущего исключения

Анализ иерархии стандартных исключений

В BCL существует множество классов исключений (наследников System.Exception), разработчик может использовать эти классы и также создавать собственные классы исключений. На рас.2. представлена небольшая часть иерархии исключений.

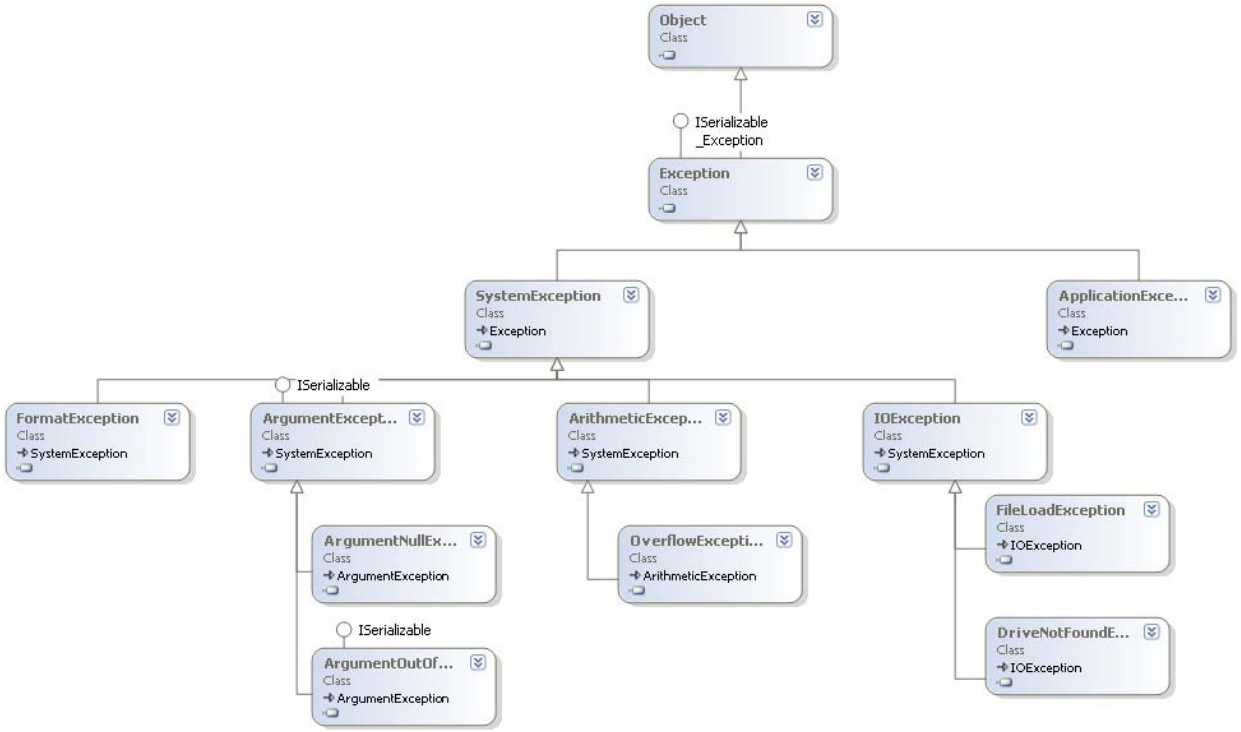


Рис. 2. Некоторые классы исключений.

Следует подчеркнуть, что на рис. 2. Представлены далеко не все исключения. Основное назначение этой схемы – показать общие закономерности иерархии исключений.

Как видно из рис. 2. Все исключения делятся на 2 группы: SystemException и ApplicationException.

SystemException – это класс исключений, которые обычно генерируются CLR или являются исключениями общей природы и могут быть сгенерированы любым приложением. Например, исключение StackOverflowException генерируется CLR при переполнении стека, исключение ArgumentException (и производные от него) могут быть сгенерированы любым приложением, если метод получает недопустимые значения аргументов.

ApplicationExcption – от этого класса должны наследоваться пользовательские исключения, специфичные для приложения.

Иерархия классов исключений является в некоторой степени необычной, т.к. производные классы в основном не добавляют новую

функциональность к возможностям базового класса. Эти классы используются для указания более специфических причин возникновения ошибки. Например, от класса ArgumentException наследуются классы ArgumentNullException (генерируется при передаче null в качестве параметра метода) и ArgumentOutOfRange (генерируется при выходе переменной за допустимый диапазон значений).

2. Основы обработки исключений

Синтаксис блока обработки исключений:

```
try
{
// код, в котором может возникнуть исключение
}
catch(тип исключения)
{
// обработка исключения
}
finally
{
// освобождение ресурсов
}
```

Блок ***try*** содержит код, требующий общей очистки ресурсов или восстановления после исключения.

Блок ***catch*** содержит код, который должен выполняться при возникновении исключения. При объявлении блока ***catch*** указывается тип исключения, для обработки которого он предназначен. Если блок ***try*** завершился без генерации исключения, блоки ***catch*** не выполняются. Если в блоке ***try*** не предполагается возникновение исключения, блок ***catch*** может отсутствовать, но тогда обязательно должен быть блок ***finally***.

Блок ***finally*** обычно содержит очистку ресурсов, а также другие действия, которые необходимо гарантировано выполнить после завершения блока ***try*** и ***catch***. Например, в этом блоке можно выполнить закрытие файла или закрытие соединения с БД. Блок ***finally*** выполняется всегда, независимо от возникновения исключения в блоке

try. Если нет необходимости выполнять очистку ресурсов, блок ***finally*** может отсутствовать, но тогда обязательно должен быть блок ***catch***. Блок ***try*** сам по себе не имеет смысла.

Для генерации исключения используется ключевое слово **throw**.

Синтаксис генерации исключения:

```
throw new Конструктор класса исключения()
```

В качестве типа исключения надо использовать производный класс иерархии исключений, который наиболее полно описывает возникшую проблему. Не рекомендуется использовать базовые классы, т.к. в этом случае при обработке исключения трудно будет точно определить причину его возникновения.

При возникновении исключения в блоке ***try*** выполнение этого блока прекращается и CLR выполняет поиск блока ***catch***, предназначенного для обработки исключений такого типа. Если этот блок найден, он выполняется, затем выполняется блок ***finally*** (если он есть). Если подходящий блок ***catch*** не найден, исключение считается необработанным и приводит к аварийному завершению программы.

Пример 1. Последовательность выполнения кода при возникновении исключения.

```
using System;

namespace ExceptionExample1
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Before Exception"); //1
                throw new Exception("Test Exception"); //2
                Console.WriteLine("After Exception.
                                   This line will never appear");
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception: {0}", e.Message); //3
            }
            finally
            {
                Console.WriteLine("In finally block"); //4
            }
        }
    }
}
```

```
Before Exception
Exception: Test Exception
In finally block
```

Текст «After Exception» не будет выведен, т.к. после генерации исключения выполнение блока ***try*** прекращается.

3. Тонкости обработки исключений

У одного блока ***try*** может быть несколько блоков ***catch*** для обработки исключений различных типов. При возникновении исключения поиск обработчика начинается с 1-ого блока ***catch***, поэтому следует, сначала размещать обработчики для исключений производных классов, затем базовых.

Перехват всех исключений

Для обработки всех типов CLS совместимых исключений можно использовать блок ***catch***, который перехватывает System.Exception, т.к. этот класс является базовым для всех классов исключений:

```
catch(Exception e)
{
}
```

Как уже отмечалось, в C# разрешается генерировать исключения только классов производных от System.Exception. Однако методы, разработанные на других языках программирования, например на C++, могут генерировать исключения любых типов. Для обработки всех типов исключений (в том числе и не CLS совместимых) в ***catch*** можно не указывать тип перехватываемого исключения:

```
catch
{
}
```

Пример 2. Использование нескольких блоков ***catch***: вычисляется выражение d = 100/ln(n), n вводится пользователем. Перехватываются следующие типы исключений:

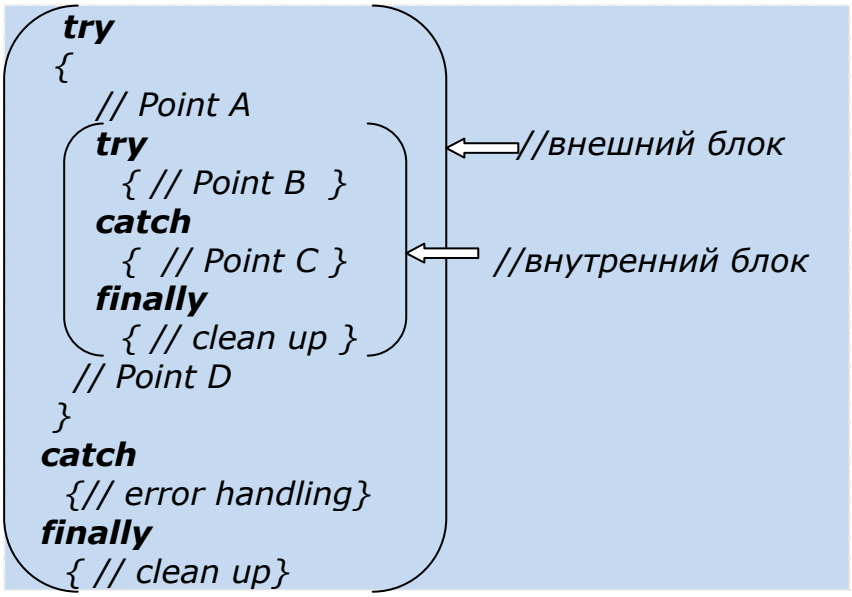
- FormatException – возникает, если введенную пользователем строку невозможно преобразовать в число.
- DivisionByZeroException – возникает, если Ln(n) = 0, т.е. n = 1
- Exception – все остальные исключения, наследуемые от Exception, например Overfloaw

```
using System;

namespace MultipleCatchBlocks
{
    class Program
    {
      static void Main(string[] args)
       {
         do
          {
           try
           {
             Console.Write("{0}Input int number: ",Environment.NewLine);
              //чтение ввода пользователя
               string s = Console.ReadLine();
               //условие выхода из цикла
               if (s== string.Empty) return;
               //преобразование строки в число
               int n = Convert.ToInt32(s);
              //проверка, что полученное число принадлежит
              //области определения функции ln()
              if (n <= 0) throw new ArgumentOutOfRangeException("n <= 0");
              double f = Math.Log(n);
              int d = 100 / (int)f;
              Console.WriteLine("d = {0} f = {1}", d, f);
          }
          catch (FormatException)
           {
             //происходит, если введенное пользователем значение
             //невозможно преобразовать в целое число
                Console.WriteLine("FormatException");
           }
          catch (DivideByZeroException)
           {
              //происходит, елси Log(n) = 0 (т.е. n = 1)
                Console.WriteLine("DivideByZeroException");
           }
          catch (Exception e)
           {
              //прехват всех остальных исключений
              //например, исключения ArgumentOutOfRangeException,
              //которое генерируется, если Log(n) не определен (т.е. n <= 0)
              Console.WriteLine("Exception: {0}",e.Message);            }
        }
           while (true);
       }
    }
}
```

```
Input int number: qwe
FormatException

Input int number: 0
Exception: Specified argument was out of the range of valid values.
Parameter name: n <= 0

Input int number: 1
DivideByZeroException

Input int number: 4
d = 100 f = 1.38629436111989
```

Вложенные блоки *try*

Блоки ***try*** могут быть вложенными:



Рассмотрим несколько сценариев возникновения и обработки исключений в этом блоке. Если исключение возникает в точке B, выполняется блок ***finally*** вложенного блока ***try***, затем выполняется поиск подходящего обработчика исключения. Если исключение может быть обработано внутренним блоков ***catch***, оно перехватывается и обрабатывается, после чего продолжается выполнение кода – точка D. Если не возникало нового исключения, блок ***catch*** внешнего блока игнорируется, блок ***finally*** внешнего блока выполняется в любом случае. Если исключение не может быть перехвачено внутренним блоком ***catch***, выполняется проверка внешнего блока ***catch*.** Если этот блок не в

состоянии обработать исключение, поиск подходящего обработчика выполняется выше по стеку вызовов.

Пример 3. Вложенные блоки ***try***. Во внутреннем блоке происходит 2 вида исключений:

- деление на 0

- обращение к массиву по недопустимому индексу.

1-ое исключение перехватывается внутренним блоком ***catch***, 2-ое – внешним.

```
using System;

namespace NestedTryBlocks
{
    class Program
    {
        static void Main()
        {
            int[] a = new int[5];
            int cnt = 0;
            try //внешний блок try
            {
                for (int i = -3; i <= 3; i++)
                {
                    //при делении на 0 не происходит выход из цикла:
                    //это исключение перехватывается
                    //и обрабатывается вложенным блоком try
                    try //вложенный блок try
                    {
                        a[cnt] = 100 / i;
                        Console.WriteLine(a[cnt]);
                        cnt++;
                    }
                    catch (DivideByZeroException e)
                    {
                        Console.WriteLine("In inner catch");
                        Console.WriteLine(e.Message);
                    }
                }
            }
            catch (IndexOutOfRangeException e)
            {
                Console.WriteLine("In outer catch");
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

```
-33
-50
-100
In inner catch
Attempted to divide by zero.
100
50
In outer catch
Index was outside the bounds of the array.
```

Повторное генерирование исключений

При обработке исключения в блоке ***catch*** возможна одна из следующих ситуаций:

✓ Приложение ожидало возникновения этого исключения и «знает» как его обработать. Например, пропала связь с сервером БД и приложение переходит в автономный режим работы. В этом случае в блоке ***catch*** выполняется обработка исключения, поток покидает блок ***catch*** и выполняет расположенный далее код.

✓ В блоке ***catch*** выполняется некоторая обработка, например откат транзакции в БД, и то же самое исключение генерируется повторно для уведомления вызывающего метода о возникшей проблеме.

✓ В блоке ***catch*** уточняется причина возникновения исключения и генерируется новое более информативное исключение. Например, когда происходит низкоуровневое системное исключение из контекста вызвавшего его метода можно понять причину его возникновения и сгенерировать исключение, более полно описывающее проблему.

Для повторной генерации исключения того же типа используется оператор throw;

Для генерации исключения другого типа используется обычный синтаксис генерации исключения. Исходное исключение, перехваченное блоком ***catch***, сохраняется в поле InnerException нового исключения.

*Пример 4.*Повторная генерация исключения. Исключение возникает в функции f(), перехватывается блоком ***catch***, расположенным в этой же функции, генерируется повторно и перехватывается блоком ***catch*** в функции Main().

```
using System;
```

```
namespace RethrowException
{
    class Program
    {
        static void f()
        {
            try
            {
                throw new Exception("MyException");
            }
            catch (Exception e)
            {
                Console.WriteLine("In catch block function f().
                                   Exception: {0}", e.Message);
                throw;
            }
        }

        static void Main()
        {
            try
            {
                f();
            }
            catch (Exception e)
            {
                Console.WriteLine("In catch block function Main().
                                   Exception: {0}",e.Message);
            }
        }
    }
}
```

```
In catch block function f(). Exception: MyException
In catch block function Main(). Exception: MyException
```

Пример 5. Генерации исключения другого типа в блоке ***catch***. Блок ***catch*** перехватывает исключение деления на 0 и генерирует исключение, поясняющее, кокой аргумент имел недопустимое значение.

```
using System;

namespace ThrowExceptionFromCatchBlock
{
    class Program
    {
        static public Int32 SomeMethod(Int32 x)
        {
            try
            {
                return 100 / x;
            }
            catch (DivideByZeroException e)//перехват исключения деления на 0
            {
                //генерация исключения недопустимого значения аргумента
                //исходное исключение передается как внутреннее исключение
```

```
                throw new ArgumentOutOfRangeException("x can't be 0", e);
            }
        }

        static void Main()
        {
            try
            {
                SomeMethod(0);
            }
            catch (ArgumentOutOfRangeException e)
            {
                Console.WriteLine(e.Message);
                //вывод внутреннего исключения
                Console.WriteLine("InnerException: {0}",
                                  e.InnerException.Message);
            }
        }
    }
}
```

```
x can't be 0
InnerException: Attempted to divide by zero.
```

Трассировка стека при исключениях

При генерации исключения CLR регистрируют метод, в котором была исполнена команда ***throw***. Когда обнаруживается блок ***catch***, способный обработать это исключение, CLR регистрирует место перехвата исключения. Теперь, если в блоке ***catch*** обратиться к свойству StackTrace сгенерированного объекта исключения, код этого свойства вызовет CLR. При этом CLR построит строку, идентифицирующую все методы, вызванные между возникновением исключения и его обработкой. Свойство StackTrace дает информацию об обработке исключения, включая метод, его сгенерировавший.

Пример 6. Трассировка стека при возникновении исключения: Исключение возникает в методе f2(), перехватывается в методе f().

```
using System;

namespace StackTrace
{
    class Program
    {
        static void f2()
        {
            throw new Exception("MyException");//генерация исключения
        }
```

```
        static void f1()
        {
            f2();
        }
        static void f()
        {
            try
            {
                f1();
            }
            catch (Exception e)//перехват исключения
            {
                Console.WriteLine(e.Message);
                //вывод стека трассировки
                Console.WriteLine("Stack trace:\n {0} ", e.StackTrace);
            }
        }
        static void Main(string[] args)
        {
            f();
        }
    }
}
```

```
MyException
Stack trace:
    at StackTrace.Program.f2()
   at StackTrace.Program.f1()
   at StackTrace.Program.f()
```

Методы f(), f1(), f()2 выводятся в трассировке, метод Main не выводится, т.к. он расположен в стеке вызовов выше, чем f(), в которой было перехвачено исключение.

Чтобы получить полную трассировку стека от начала потока до вызова обработчика исключения, надо воспользоваться типом System.Diagnostics.Stack.Trace, в котором определен ряд свойств и методов, позволяющих получить все методы, вызванные в текущий момент времени на стеке потока.

4. Применение конструкций *checked* и *unchecked*

При выполнении арифметических операций с целочисленными типами данных может возникать переполнение, т.е. ситуация при которой количество двоичных разрядов полученного результата превышает разрядность переменной, в которую этот результат записывается.

Переполнение может возникнуть в следующих ситуациях:

- ✓ в выражениях, которые используют арифметические операторы:

  ++ -- - (унарный) **+ - * /**

- ✓ при выполнении явного преобразования целочисленных типов

При возникновении переполнения CLR может использовать один из двух вариантов:

- ✓ проигнорировать переполнение и отбросить старшие разряды
- ✓ сгенерировать исключение OverflowException

По умолчанию при возникновении переполнения старшие разряды отбрасываются.

Например:

```
Byte b = 100;
b = (Byte)(b + 200);
//b = 44 – разраяды, которые не попадают в диапазон Byte отбрасываются
int a2 = 0x10000;
short s = (short)a2;
// s = 0 – разраяды, которые не попадают в диапазон Short отбрасываются
```

Можно явно задать режим контроля переполнения с помощью ключевых слов ***checked*** и ***unchecked***.

Ключевое слово ***checked*** задает режим контроля переполнения с генерацией исключения.

Ключевое слово ***unchecked*** задает игнорирование возникновения переполнения.

Пример 7. Оператор ++ выполняется для переменной b с начальным значением 255

- в блоке ***checked*** – возникает исключение, которое перехватывается в блоке ***catch***

- в блоке ***unchecked*** – исключение не возникает, в переменную b записывается значение 0.

```
using System;

namespace CheckOverflowException
{
    class Program
    {
        static void Main(string[] args)
        {
            byte b = 255;
```

```
            try
            {
                checked
                {
                    b++; //генерация OverflowException thrown
                }
                Console.WriteLine(b.ToString());
            }
            catch (OverflowException e)
            {
                Console.WriteLine(e.Message);
            }

            try
            {
                unchecked
                {
                    b++; //переполнение игнорируется
                }
                Console.WriteLine(b.ToString()); //0
            }
            catch (OverflowException e)
            {
                Console.WriteLine(e.Message);
            }
        }

    }
}
```

Ключевые слова ***checked*** и ***unchecked*** могут использоваться как операторы в строке преобразования.

Например:

```
Byte b = 100;
b = unchecked((Byte) (b + 200)); // b = 44
b = checked((Byte) (b + 200));// генерация OverflowException
```

В данном примере необходимо выполнять явное приведение к типу Byte, т.к. в .Net операции выполняет над 32 и 64 разрядными данными, т.е. Byte приводится к Int32

Если режим контроля переполнения не указан явно, он определяется опцией компилятора /checked. Эту опцию можно задать с помощью свойств проекта в меню

Properties->Debug->Advanced->Check for Arithmetic underflow/overflow. Если флажок установлен, при возникновении переполнения генерируется исключение.

5. Что такое пространство имён

А теперь поближе познакомимся с одним из основополагающих понятий языка **C#**, без которого так или иначе не обходится ни одна программа.

Речь пойдёт о так называемых пространствах имён. На самом деле, не совсем новое понятие, поскольку вы уже знакомы с ним из курса **C++**. В языке **C#** пространства имён представляют собой нечто подобное. Но…не будем торопиться.

Что же представляет собой пространство имён в **C#**?

Это некая область объявления данных, обеспечивающую логическую взаимосвязь типов.

Другими словами, пространство имён - это способ группирования ассоциированных типов, включая классы, структуры, делегаты, интерфейсы, перечисления, а также другие пространства имён.

Это логический способ объединения типов.

При создании класса в файле проекта вы можете поместить его в пространство имён, которое сами ему назначите. Если позднее вы решите включить в программу другой класс (возможно, уже в другом файле), выполняющий какую-то ассоциированную задачу, то поместите его в это же пространство имён, создавая логическую группу. Тем самым вы указываете на то, как данные классы взаимодействуют друг с другом.

Если какой-нибудь тип (пусть это будет класс **Poetry**) был объявлен в своём пространстве имён (пусть он называется **Inspiration**), то это обозначает, что этот тип будет мирно существовать в своём пространстве имён, не конфликтуя с одноимёнными типами, объявленными в пространствах имён, отличных от данного.

Честно говоря, понятие пространства имён очень жизненное.

Например, одного моего соседа зовут Алексей, а в школе у меня был одноклассник, которого тоже так звали. А в коллективе на работе есть один товарищ, которого (вы не поверите!!!) тоже зовут Алексей.

Список можно продолжать…

И хотя всех этих людей объединяет одно имя, их не спутать, потому что каждый из них ограничен логикой своего контекста. Для одного это класс в школе, для другого - дом, где я живу, а для третьего - коллектив, в котором сейчас работаю.

Переводя это в синтаксис языка **C#**, информацию об этих людях можно отобразить следующим образом:

1. ШкольныйКласс.Алексей
2. МойДом.Алексей
3. КоллективНаРаботе.Алексей

Программирование не отстаёт от жизни, и поэтому очень похожая ситуация существует и здесь. Представить себе невозможно, сколько библиотек, классов, функций, переменных и тому подобного богатства, обозначенного каждый своим идентификатором, уже существует на сегодняшний день. Не мудрено, что программисту, работающему над созданием новых библиотек, классов и тому подобного добра, легко дать своему детищу имя, которым уже когда-то наградили другой тип. Гигабайты человеческой памяти, к сожалению, не в состоянии удерживать такое количество именований (если тот, кто читает этот урок, может продемонстрировать обратное, заранее прошу прощения).

Та же ситуация просматривается в библиотеке **.NET**
В пространстве имён **System.Windows.Forms** есть тип **Button**, и в пространстве имён **System.Web.UI.Controls** таковой имеется. Однако каждый из них существует в рамках своего пространства имён, и конфликта не возникает.

6. Цели и задачи пространства имён

Итак, вы уже, наверное, догадались, что одной из основных задач создания пространств имён является предотвращение конфликтов по совпадению имён.

Рассмотрим следующий пример:

ClassFoo.cs

```
using System;

class Foo
{
//...
}
class Foo
{
//...

}
```

В ответ на такую запись компилятор выдаёт следующее сообщение

Error1 The namespace '<global namespace>' already contains a definition for 'Foo'

Оба класса объявлены в одном файле, соответственно они находятся в одной области видимости. А поскольку идентификаторы у них совпадают, возникает одна из самых распространённых ситуаций - конфликт имён.

Для решения этой ситуации нужно либо изменить название одного из классов, либо разграничить их области видимости с помощью пространств имён. В данной ситуации, конечно, можно обойтись и первым способом, примерно так:

```
using System;

class Foo
```

```
{
    //...
}
class Foo1
{
    //...

}
```

Но в другом случае это могло быть критично. Тогда конфликт может быть полностью исчерпан с помощью пространства имён:

```
using System;

namespace CSharpNamespace
{
    class Foo
    {
        //...
    }
}
class Foo
{
  //…

}
```

либо так:

```
using System;

namespace CSharpNamespace
{
    class Foo
    {
        //...
    }
}
namespace CSharpNamespace1
{
    class Foo
    {
        //...
    }

}
```

Ещё один пример:

Class1.cs

```
using System;

namespace A
{
     class Incrementer
     {
         private int count;

         public Incrementer(int count)
         {
             this.count = count;
         }
         public int MultyIncrement()
         {
             for (int i = 0; i < 5; i++)
                 count++;
             return count;
         }
     }

}
namespace B
{
     class Incrementer
     {
         private int var;

         public Incrementer(int var)
         {
             this.var = var;
         }
         public int AnotherMultyIncrement()
         {
             for (int i = 0; i < 5; i++)
                 var += 10;
             return var;
         }
     }
     class Tester
         {
             public static void Main()
             {
                 Incrementer obj1 = new Incrementer(10);
                 Console.WriteLine(obj1.AnotherMultyIncrement());
                 A.Incrementer obj2 = new A.Incrementer(5);
                 Console.WriteLine(obj2.MultyIncrement());
             }
         }
}
```

На выходе:
60
10

В данной программе в одном файле в двух разных пространствах имён

```
namespace A
```
и

```
namespace B
```

размещены два класса с одинаковым именем

```
class Incrementer
```

Точка входа программы метод **Main** размещён во втором из классов. В **Main** создаётся экземпляр класса **Incrementer** из пространства имён **B**, и для него вызывается метод, увеличивающий значение поля

```
Incrementer obj1 = new Incrementer(10);
Console.WriteLine(obj1.AnotherMultyIncrement());
```

Затем создаётся ещё один объект - экземпляр класса **Incrementer** из пространства имён **A**.

```
A.Incrementer ob2 = new A.Incrementer(5);
Console.WriteLine(ob2.MultyIncrement());
```

Обратим внимание на запись: здесь имя класса указывается в сочетании с названием пространства имён. Почему так? Пространство создаёт область видимости для всех объектов (читай типов), находящихся в его пределах. Внутри пространства имён **A** не видны объекты из пространства имён **B** и наоборот. Для того чтобы в пространстве имён **B** обратиться к типу из пространства имён **A**, необходимо явно указать имя второго.

А что будет, если мы поступим иначе и не добавим название пространства имён **A** перед именем класса **Incrementer**? Поменяем запись и заново запустим программу.

Код теперь выглядит так:

```
class Tester
     {
         public static void Main()
         {
             Incrementer obj1 = new Incrementer(10);
             Console.WriteLine(obj1.AnotherMultyIncrement());
             Incrementer obj2 = new Incrementer(5);
```

```
            Console.WriteLine(obj2.MultyIncrement());
        }
    }
```

А компилятор выдаёт сообщение об ошибке:

'B.Incrementer' does not contain a definition for 'MultyIncrement'

Это происходит потому, что компилятор воспринимает объявление второго объекта как попытку создать ещё один экземпляр класса **Incrementer** из пространства имён **B**. А поскольку в нём нет определения метода под названием **MultyIncrement**, генерируется ошибка.

Предположим, в обоих пространствах имён в соответствующих классах **Incrementer** определён метод с названием **MultyIncrement**. В методе **Main** создаётся два экземпляра класса **Incrementer** с последующим вызовом метода **MultyIncrement** для обоих. Всё это отображено в следующем коде:

```
using System;

namespace A
{
   public class Incrementer
   {
        private int count;

        public Incrementer(int count)
        {
            this.count = count;
        }
        public int MultyIncrement()
        {
            for (int i = 0; i < 5; i++)
                count++;
            return count;
        }
    }

}
namespace B
{
   public class Incrementer
    {
        private int var;

        public Incrementer(int var)
        {
            this.var = var;
```

```
            }
          public int MultyIncrement()
            {
                for (int i = 0; i < 5; i++)
                    var += 10;
                return var;
            }
        }
        class Tester
            {
                public static void Main()
                {
                    Incrementer obj1 = new Incrementer(10);
                    Console.WriteLine(obj1.MultyIncrement());
                    Incrementer obj2 = new Incrementer(5);
                    Console.WriteLine(obj2.MultyIncrement());
                }
            }
    }
```

На выходе:
60
55

Эта ситуация не вызовет никаких претензий компилятора, поскольку, с точки зрения синтаксиса, он совершенно корректный. Однако возникает вопрос целесообразности создания первого пространства имён и всего его содержимого, поскольку **namespace A** никак не задействовано.

7. Ключевое слово using

Для того чтобы подключить в программу нужное пространство имён, необходимо использовать ключевое слово **using**, после которого указывается название пространства имён.
В каждом Вашем консольном приложении присутствует строка

```
using System;
```

Она нужна, чтобы вы могли, к примеру, выводить на консоль текстовые сообщения

```
public static void Main()
{
        Console.WriteLine("Hello, everybody!");
 }
```

Дело в том, что класс **Console** размещён в пространстве имён **System**, и чтобы в программе не приходилось явно указывать название **System** при каждом обращении к этому классу, мы подключаем его через директиву **using**.

А вот как выглядела та же запись без применения директивы **using**:

```
public static void Main()
{
        System.Console.WriteLine("Hello, everybody!");
 }
```

И так на протяжение всей программы. Утомительно, не так ли?

Конечно, при создании собственных пространств имён также может применяться ключевое лово **using**.

Например, в проекте 2 файла - Class1.cs и Class2.cs, в каждом из них своё пространство имён, **CSharpNamespace1** и **CSharpNamespace2** соответственно. Мы хотим в методе класса из пространства имён **CSharpNamespace2** файла Class2.cs создать экземпляр класса из пространства имён **CSharpNamespace1** файла Class1.cs. Для этого нам всего-навсего нужно в файле Class2.cs подключить пространство имён из другого файла с помощью **using**. Вот как это выглядит:

Class1.cs

```
using System;

namespace CSharpNamespace1
{
    class Foo
    {
        //...
    }
}
```

Class2.cs

```
using System;
using CSharpNamespace1;
namespace CSharpNamespace2
{
    class Class2
    {
        public void Method()
        {
         Foo obj = new Foo();
    }
}
```

Если два пространства имён, указанные в операторах **using**, содержат тип с одинаковым именем, то в этом случае необходимо использовать полную (или как минимум более длинную) форму имени, чтобы компилятор знал, какой именно тип в каждом случае использовать. Если этого не сделать, компилятор не будет знать, к типу из какого пространства имён вы обращаетесь в данный момент.

Program1.cs

```
using System;
namespace NS
{
    public class Class
    {
        public int a = 1;

        public void Print()
        {
            Console.WriteLine("Printing from NS");
        }
    }
}
```

Program2.cs

```
using System;
namespace NS1
{
    public class Class
    {
        public int b = 2;
        public void Print()
        {
            Console.WriteLine("Printing from NS1");
        }
    }
}
```

Main.cs

```
using System;
using NS;
using NS1;
namespace M
{
    public class ClassM
    {
        public static void Main()
        {
            Class objA = new Class();
            Console.WriteLine("objA = {0}", objA.a);
            objA.Print();
            Class objB = new Class();
            Console.WriteLine("objB = {0}", objB.b);
            objB.Print();
        }
    }
}
```

```
Error 1    'Class'  is  an  ambiguous  reference  between  'NS.Class'  and
'NS1.Class' Error 2    'Class' is an ambiguous reference between 'NS.Class'
and 'NS1.Class' Error  3    'Class'  is  an  ambiguous  reference  between
'NS.Class' and 'NS1.Class' Error   4    'Class'  is  an  ambiguous  reference
between 'NS.Class' and 'NS1.Class'  Error     5    'NS.Class'    does    not
contain a definition for 'b'
```

Чтобы исправить данные ошибки, нужно просто добавить названия пространств имён перед идентификаторами **Class**:

```
        public static void Main()
        {
            NS.Class objA = new NS.Class();
            Console.WriteLine("objA = {0}", objA.a);
            objA.Print();

            NS1.Class objB = new NS1.Class();
            Console.WriteLine("objB = {0}", objB.b);
            objB.Print();
        }
```

На выходе:

```
objA = 1
Printing from NS
objB = 2
Printing from NS1
```

8. Объявление пространства имён

Для объявления пространства имён используется ключевое слово namespace, а всё, что определяется внутри него, заключается в фигурные скобки.

```
namespace dataBinding
{
    ...
}
```

В пространство имён могут включаться классы, структуры, делегаты, интерфейсы, перечисления, а также другие пространства имён (знакомство с некоторыми из вышеперечисленных средств языка **C#** Вам только предстоит в следующих уроках).

9. Вложенные пространства имен

Здесь всё предельно просто: пространства имён могут быть вложенными друг в друга. Это создаёт иерархические структуры для ваших типов:

```
namespace ITAcademy
{
    namespace ProgrammingDepartment
    {
        namespace CSharp
        {
            namespace Basics
            {
                class MyClass
                {
                    ///...
                }
            }
        }
    }
  }
}
```

Каждое имя в пространстве имён состоит из имён пространств, в которые оно вложено. Все имена разделяются между собой точками. Первым указывается самое внешнее имя, и далее вглубь по иерархии. Например, полным именем для сласса из предыдущего примера будет

```
ProgrammingDepartment.CSharp.Basics.MyClass
```

Иерархическое размещение характерно для **.NET**, и большая часть типов платформы имеет полное имя, состоящее из нескольких пространств имён.

```
System.Collections.Generic.List<>
System.Windows.Forms.Button
System.IO.StreamWriter
System.Xml.Serialization.XmlSerializationReader
```

Если вы разрабатываете классы для своей компании, Microsoft рекомендует применять хотя бы двойной уровень вложенности пространств имён: первое - имя вашей компании, а второе - название тенологии или пакета программного обеспечения, к которому относится класс.

Выглядит это примерно так:

```
namespace CompanyName
{
    namespace BankingSerices
    {
        class Client
        {
            //...
        }
    }
}
```

и полное имя типа будет

```
CompanyName.BankingSerices.Customer
```

В данном примере внешнее пространство имён **CompanyName** является корневым. Оно, как правило, служит лишь для расширения облаcи видимости, и в нём непосредственно не определяются типы.

Политика многоуровневой вложенности типов защищает от совпадения названий ваших типов с типами, разработанными другими компаниями.

N.B.
Нельзя объявлять составое пространство имён, вложенное в другое пространство имён.

По поводу вложенности пространств имён нужно понимать одно простое правило: вы создаёте области видимости, вложенные друг в друга, а это значит, что все типы, объявленные в пространстве имён на ступеньку ниже, не доступны из объемлющих пространств имён. В подтверждение сказанного расмотрим пример:

```
using System;
namespace NS
{
    namespace A
    {
        class Foo
        {
            public void Method()
            {
                Console.WriteLine("Hello from A.Foo");

            }
            static void Main()
            {
                Foo obj = new Foo();
                obj.Method();
            }
        }
    }
    class Foo
    {
        public void Method()
        {
            Console.WriteLine("Hello from NS.Foo");
        }
    }
}
```

Результатом этой программы будет запись:

Hello from A.Foo

10. Разбиение пространства имён на части.

Представим ситуацию: несколько программистов вашей фирмы работают над разработкой разных классов для одного приложения. Каждый из этих классов находится в своём файле, но помещён в одно и то же

пространство имён. Когда эти файлы добавятся в проект, они будут скомпилированы в одну сборку.

Также вы можете в пределах одного файла разделяете пространство имён на части.

Рассмотрим следующий пример:

```
Class1.cs


using System;
namespace A
{
    public class ClassA
    {
        public void Print()
        {
            Console.WriteLine("Printing from A.ClassA");
        }
    }

}


Class2.cs


using System;
using  A;
namespace A
{
    class ClassB
    {
        public void Print()
        {

            Console.WriteLine("Printing from A.ClassB");
        }
    }
}
namespace B
{
    public class Class
    {
        public static void Main()
        {
            ClassA a = new ClassA();
            a.Print();
            ClassB b = new ClassB();
            b.Print();
            ClassC c = new ClassC();
            c.Print();
        }
    }
```

```
}
namespace A
{
    class ClassC
    {
        public void Print()
        {
          Console.WriteLine("Printing from A.ClassC");
        }
    }

}
```

Здесь пространство имён **A** распределено по двум файлам, а фактически разбито на 3 части, поскольку в файле Class2.cs оно дополнительно разделено на 2 части.

Во втором файле включен оператор

```
using  A;
```

Это нужно для того, чтобы во втором пространстве имён **B** были доступны все типы, определённые в пространстве имён **A**.

Результатом работы программы является следующее:

Printing from A.ClassA

Printing from A.ClassB

Printing from A.ClassC

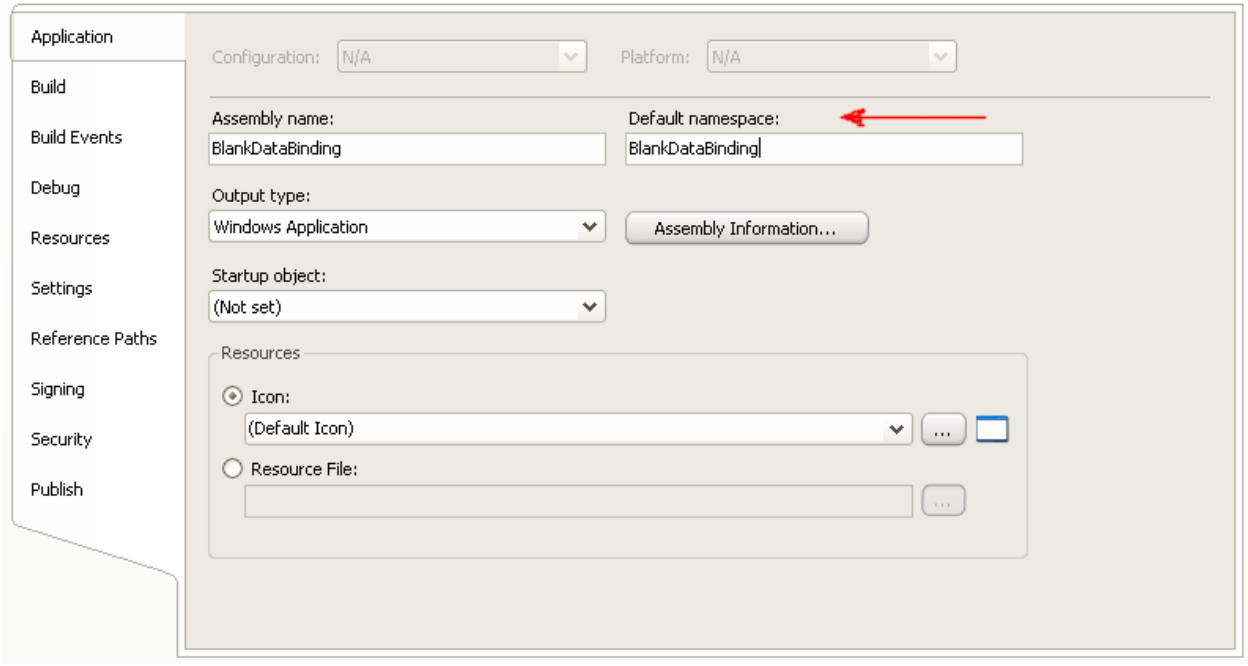
11. Пространство имён по умолчанию

По правилам платформы **.NET**, все имена должны быть объявлены в пределах пространства имён.

А что происходит, если мы не назначаем никакого пространства имён для своих типов?

В таком случае для них будет назначено пространство имён по умолчанию, и оно будет совпадать с именем проекта.

Если вас это не устраивает, вы можете изменить его имя на то, которое вам нужно. Для этого на вкладке **Application** окна свойства проекта воспользуйтесь опцией **Dafault** namespace и укажите там имя, которое нужно.



Если вы так поступите, то каждый последующий файл, добавляемый вами в проект, автоматически будет помещаться в пространство имён с этим именем.

12. Вторая форма using

Ключевое слово **using** может применяться не только для подключения необходимого пространства имён, но также для создания псевдонимов пространств имён.

Чем это удобно? Поскольку, как мы упоминали ранее, пространство имён может иметь какую угодно глубину вложенности, соответственно размеры идентификатора могут быть очень громоздкими, например:

```
namespace Wrox.ProCSharp.WindowsApplication.SimpleForm
{
//...
}
```

Чтобы каждый раз не переписывать такое название, мы создаём, по сути, ещё одно, более лаконичное имя для пространства имён следующим способом:

```
using псевдоним = полное название пространства имён;
using NS = Wrox.ProCSharp.WindowsApplication.SimpleForm;
```

При этом в дальнейшем желательно обращаться к типу через квалификатор :: . В случае, если в данном пространстве имён будет определён тип с таким же именем, как псевдоним, конфликт возникать не будет.

```
Program1.cs

using System;
namespace NS
{
    public class Class
    {
        public int a = 1;

        public void Print()
        {
            Console.WriteLine("Printing from NS");
        }
    }
}

Program2.cs

using System;
namespace NS1
{
    public class Class
```

```
        {
            public int b = 2;
            public void Print()
            {
                Console.WriteLine("Printing from NS1");
            }
        }
    }

    Main.cs

    using System;
    using  X = NS;
    using Y = NS1;
    namespace M
    {
        public class ClassM
        {
            public int m = 3;

            public static void Main()
            {
                X::Class objA = new X::Class();
                Console.WriteLine("objA = {0}", objA.a);
                objA.Print();
                Y::Class objB = new Y::Class();
                Console.WriteLine("objB = {0}", objB.b);
                objB.Print();
            }
        }
        public class X
        {
          //...
        }
    }
```

В этом примере представлен проект, состоящий из нескольких файлов: `Program1.cs, Program2.cs, Main.cs.`

В каждом файле код заключён в своё пространство имён. В файле `Main.cs` назначаются псевдонимы для пространств имён из двух других файлов

```
using  X = NS;
using Y = NS1;
```

И здесь же через псевдонимы идёт обращение к типам из пространств имён **NS** и **NS1**. Казалось бы, ничем не примечательная ситуация, но…

В пространстве имён **M** файла `Main.cs` объявлен класс **X**, чьё имя совпадает с псевдонимом, назначенным для пространства имён **NS**

```
public class X
{
  //...
}
```

Если бы в теле метода **Main** мы обратились к идентификатору **X**, то компилятор расценил бы это как обращение к классу с одноимённым названием. Обратиться через точку к классу Class из пространства имён X в этой ситуации невозможно. Ситуацию спасает квалификатор ::

```
X::Class objA = new X::Class();
```

И в заключение нашего знакомства с оператором **using** скажу, что оно ещё продолжится в одном из следующих уроков. Если вы заглянете в **MSDN**, то найдёте следующее замечание по поводу применения ключевого слова **using**:

The using keyword has two major uses:

- **As a directive, when it is used to create an alias for a namespace or to import types defined in other namespaces.**
- **As a statement, when it defines a scope at the end of which an object will be disposed**.

Первый пункт относится к способу применения **using**, описанному нами выше, а второй - к разделу языка **C#**, с которым вам только предстоит познакомиться в ближайшем будущем.

13. Введение в перегрузку операторов

Перегрузка операторов позволяет указать, как стандартные операторы будут использоваться с объектами класса. Перегрузка операторов используется для улучшения читаемости программ.

Например, в C++ при реализации строки в виде массива символов для выполнения действий со строками надо было использовать

специальные функции. Для сравнения строк вместо привычного оператора == надо было вызывать функцию strcmp(). Что особенно неприятно при использовании == программа компилировалась успешно, но выполнялось сравнение строк не по содержимому, а по адресу в памяти первого символа. В C# тип System.String использует перегрузку операторов == и != для проверки равенства строк по их содержимому, оператор + также перегружен и выполняет конкатенацию строк.

Требования к перегрузке операторов:

- ✓ перегрузка операторов должна выполняться открытыми статическими методами класса.
- ✓ у метода - оператора тип возвращаемого значения или одного из параметров должен совпадать с типом, в котором выполняется перегрузка оператора.
- ✓ параметры метода - оператора не должны включать модификатор out и ref.

Таким образом, невозможно изменить значение стандартных операций для стандартных типов данных.

При перегрузке оператора в классе генерируется метод со специальным именем. Например, при перегрузке оператора + генерируется метод op_Addition с флагом SpecialName. При компиляции кода, когда в тексте программы встречается оператор +, компилятор рассматривает типы переменных в выражении, содержащем этот оператор. Если в одном из типов определен метод op_Addition и его параметры совпадают с типами операндов, генерируется код, вызывающий этот метод.

Также как и с C++ в C# перегрузка операторов имеет некоторые ограничения:

- ✓ перегрузка не может изменить приоритет операторов
- ✓ при перегрузке невозможно изменить число операндов, с которыми работает оператор
- ✓ не все операторы можно перегружать

Операторы, допускающие перегрузку

Операторы	Категория операторов
-	Изменение знака переменной
!	Операция логического отрицания
~	Операция побитового дополнения, которая приводит к инверсии каждого бита
++, --	Инкремент и декремент
true, false	Критерий истинности объекта, определяется разработчиком класса
+, –, *, /, %	Арифметические операторы
&, |, ^, <<, >>	Битовые операции
==, !=, <, >, <=, >=	Операторы сравнения
&&, ||	Логические операторы
[]	Операции доступа к элементам массивов моделируются за счет индексаторов
()	Операции преобразования

Операторы, не допускающие перегрузку

Операторы	Категория операторов
+=, –=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Перегружаются автоматически при перегрузке соответствующих бинарных операций
=	Присвоение
.	Доступ к членам типа
?:	Оператора условия
new	Создание объекта
as, is, typeof	Используются для получения информации о типе
–>, sizeof,*,&	Доступны только в небезопасном коде

В отличие от C++ оператор присвоения = нельзя перегружать.

Перегрузку операторов можно использовать как в классах, так и в структурах.

Поскольку перегруженные операторы являются статическими методами, они не получают указателя this, поэтому унарные операторы должны получать 1 параметр, бинарные 2 (в отличие от C++, где одним параметром был указатель this).

Синтаксис перегрузки:

public static *<тип результата>* ***operator*** *<символ операции> (параметры)*

14. Перегрузка унарных операторов

Унарные операторы получают единственный операнд. Этот операнд должен иметь тип класса, в котором выполняется перегрузка оператора. То есть, если выполняется перегрузка унарного оператора в классе CPoint, то и тип операнда должен быть CPoint.

Для операторов ++ и -- возвращаемое значение также должно быть того же типа, в котором выполняется перегрузка оператора или производного от него.

Рассмотрим перегрузку унарных операторов на примере операторов инкремента, декремента и изменения знака -.

Класс Point описывает точку на плоскости, точка имеет координаты x и y. Оператор ++ увеличивает обе координаты на 1, оператор -- уменьшает, оператор – изменяет знак координат на противоположный.

```
using System;

namespace UnaryOperator
{
    //класс точки на плоскости - пример для перегрузки операторорв
   class CPoint
    {
        int x, y;
        public CPoint(int x, int y)
        {
            this.x = x; this.y = y;
        }

        //перегрузка инкремента
        public static CPoint operator ++(CPoint s)
        {
            s.x++; s.y++; return s;
```

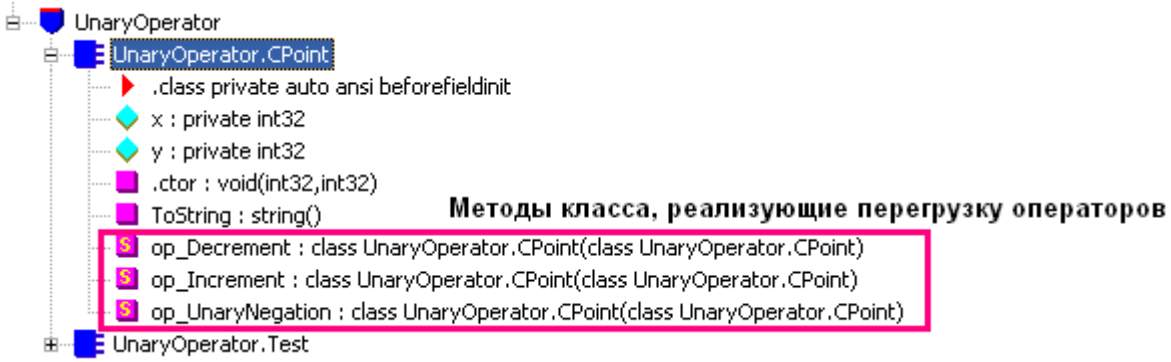
```
        }
        //перегрузка декремента
        public static CPoint operator --(CPoint s)
        {
            s.x--; s.y--; return s;
        }
        //перегрузка оператора -
        public static CPoint operator -(CPoint s)
        {
            CPoint p = new CPoint(s.x, s.y);
            p.x = -p.x; p.y = -p.y; return p;
        }

        public override string ToString()
        {
            return string.Format("X = {0} Y = {1}", x, y);
        }
    }

    class Test
    {
        static void Main()
        {
            CPoint p = new CPoint(10, 10);
            //префиксная и постфиксная формы выполняются одинаково
            Console.WriteLine(++p); //x=11, y=11
            CPoint p1 = new CPoint(10, 10);
            Console.WriteLine(p1++); //x=11, y=11
            Console.WriteLine(--p); //x=10, y=10
            Console.WriteLine(-p); //x=-10, y=-10
            //после выполнения оператора -
            //состояние исходного объекта не изменилось
            Console.WriteLine(p); //x=10, y=10
        }
    }
}
```

```
X = 11 Y = 11
X = 11 Y = 11
X = 10 Y = 10
X = -10 Y = -10
X = 10 Y = 10
```

В данном примере CPoint является ссылочным типом, поэтому изменения значений x и y, которые выполняются в перегруженных операторах инкремента и декремента, изменяют переданный в них объект. Оператор – (изменение знака) не должен изменять состояние переданного объекта, а должен возвращать новый объект с измененным знаком. Для этого в реализации этого метода создается новый объект CPoint, изменяется знак его координат и этот объект возвращается из метода.

Интересно отметить, что с C# нет возможности выполнить отдельно перегрузку постфиксной и префиксной форм операторов инкремента и декремента. Поэтому при вызове постфиксная и префиксная форма работают одинаково.

Как отмечалось ранее, перегрузка операторов выполняется путем создание специальных методов класса. Рассмотрим с помощью ildasm класс CPoint.



Легко установить следующее соответствие перегруженных операторов сгенерированным методам:

operatpr -- op_Decrement

operatpr ++ op_Increment

operatpr - op_UnryNegation

Из метаданных видно, что эти методы имеют флаг SpecialName.

```
Method #2 (06000002)
-------------------------------------------------------
      MethodName: op_Increment (06000002)
      Flags     : [Public] [Static] [HideBySig] [ReuseSlot] [SpecialName]  (00000896)
      RVA       : 0x0000206c
      ImplFlags : [IL] [Managed]  (00000000)
      CallCnvntn: [DEFAULT]
      ReturnType: Class UnaryOperator.CPoint
      1 Arguments
            Argument #1:  Class UnaryOperator.CPoint
      1 Parameters
            (1) ParamToken : (08000003) Name : s flags: [none] (00000000)
```

Если открыть ассемблерный код функции Main, можно увидеть, что вызову ++p соотвествует вызов:

```
call       class UnaryOperator.CPoint UnaryOperator.CPoint::op_Increment(class UnaryOperator.CPoint)
```

15. Перегрузка бинарных операторов

Для примера перегрузки бинарных операций дополним разработанный ранее класс CPoint новыми возможностями:

- выполнение преобразования сдвига – сложение точки с вектором, задающим смещение, в результате получается точка с новыми координатами

- выполнение преобразования масштабирование – умножение точки на коэффициент, задающий масштаб

- вычисление расстояния между двумя точками – выполняется путем вычитания соответствующих координат, в результате получается вектор

```
using System;

namespace BinaryOperator
{
    class CVector
    {
        public int x;
        public int y;
        public CVector(int x, int y)
        {
            this.x = x;
            this.y = y;
        }

        public override string ToString()
        {
            return string.Format("Vector: X = {0} Y = {1}", x, y);
        }
    }
    class CPoint
    {
        private int x;
        private int y;

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

        //перегрузка бинарного оператора +
        public static CPoint operator +(CPoint p, CVector v)
        {
            return new CPoint(p.x + v.x, p.y + v.y);
        }

        //перегрузка бинарного оператора *
        public static CPoint operator *(CPoint p, int a)
        {
            return new CPoint(p.x * a, p.y * a);
        }
```

```
        //перегрузка бинарного оператора -
        public static CVector operator -(CPoint p1, CPoint p2)
        {
            return new CVector(p1.x - p2.x, p1.y - p2.y);
        }

        public override string ToString()
        {
            return string.Format("Point: X = {0} Y = {1}", x, y);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            CPoint p1 = new CPoint(10,10);
            CPoint p2 = new CPoint(12, 20);
            CVector v = new CVector(10,20);
            Console.WriteLine("Точка p1: {0}",p1);
            Console.WriteLine("Сдвиг: {0}", p1 + v);
            Console.WriteLine("Масштабирование: {0}", p1 * 10);
            Console.WriteLine("Точка p2: {0}",p2);
            Console.WriteLine("Расстояние: {0}",p2 - p1);
        }
    }
}
```

```
Точка p1: Point: X = 10 Y = 10
Сдвиг: Point: X = 20 Y = 30
Масштабирование: Point: X = 100 Y = 100
Точка p2: Point: X = 12 Y = 20
Расстояние: Vector: X = 2 Y = 10
```

Как отмечалось ранее, выполненная перегрузка автоматически перегружает операторы +=, *=, -=. Например, можно записать:

p1+= v;

Для реализации этого действия будет использован перегруженный оператор +.

Однако перегруженные в примере операторы будут использоваться компилятором только если переменная типа CPoint находится слева от знака операнда. Т.е. выражение p * 10 откомпилируется нормально, а при перестановке сомножителей, т.е. в выражении 10 * p произойдет ошибка компиляции с сообщением: «Error 2 Operator '*' cannot be applied to operands of type 'int' and 'BinaryOperator.CPoint'». Для испраления этой ошибки следует перегрузить оператор * с другим порядком операндов:

```
public static CPoint operator *(int a, CPoint p)
        {
            return p * a;
        }
```

Эта перегруженная версия сводится к вызову оператора `operator * (CPoint p, int a)`

16. Перегрузка операторов отношений

Операции сравнения перегружаются парами: если перегружается операция ==, также должна перегружаться операция !=. Существуют следующие пары операторов сравнения:

- ✓ == и !=
- ✓ < и >
- ✓ <= и >=.

При перегрузке операторов отношения надо учитывать, что есть два способа проверки равенства:

- ✓ равенство ссылок (тождество)
- ✓ равенство значений

В классе Object определены следующие методы сравнения объектов:

public static bool ReferenceEquals(Object obj1, Object obj2)

public bool virtual Equals(Object obj)

Есть отличия в работе этих методов со значимыми и ссылочными типами.

Метод ReferenceEquals() проверяет, указывают ли две ссылки на один и тот же экземпляр класса; точнее - содержат ли две ссылки один и тот адрес памяти. Этот метод невозможно переопределить. Со значимыми типами ReferenceEquals() всегда возвращает false, т.к. при сравнении выполняется приведение к Object и упаковка, упакованные объекты располагаются по разным адресам.

Метод Equals() является виртуальным. Его реализация в Object выполняется проверку равенства ссылок, т.е. работает так же как и ReferenceEquals. Для значимых типов в базовом типе System.ValueType выполнена перегрузка метода Equals(), которая выполняет сравнение объектов путем сравнения всех полей (побитовое сравнение).

Пример использования операторов ReferenceEquals() и Equals() со ссылочными и значимыми типами:

```
using System;

namespace Equals_and_ReferenceEquals
{
 class  CPoint
  {
    private int x, y;
    public CPoint(int x, int y)
    {
        this.x = x; this.y = y;
    }
  }
 struct  SPoint
 {
    private int x, y;
    public SPoint(int x, int y)
    {
        this.x = x; this.y = y;
    }
 }
class Program
 {
    static void Main()
    {
        //Работа метода ReferenceEquals с сылочным и значимым типами
        //ссылочный тип
        CPoint p = new CPoint(0, 0);
        CPoint p1 = new CPoint(0, 0);
        CPoint p2 = p1;
        Console.WriteLine("ReferenceEquals(p, p1)= {0}",
                          ReferenceEquals(p, p1)); //false
        //хотя p,p1содержат одинаковые значения,
        //они указывают на разные адреса памяти
        Console.WriteLine("ReferenceEquals(p1, p2)= {0}",
                          ReferenceEquals(p1, p2));//true
        //p1 и p2 указывают на один и тот же адрем памяти

        //значимй тип
        SPoint p3 = new SPoint(0, 0);
        //при передаче в метод ReferenceEquals выполняется упаковка,
        //упакованные объекты располагаются по разным адресам
        Console.WriteLine("ReferenceEquals(p3, p3) =  {0}",
                          ReferenceEquals(p3, p3)); //false

        //Работа метода Equals с сылочным и значимым типами
        //ссылочный тип
        CPoint cp = new CPoint(0, 0);
        CPoint cp1 = new CPoint(0, 0);
        Console.WriteLine("Equals(cp, cp1) =  {0}", Equals(cp, cp1)); //false
        //выполняется сравнение адресов

        //значимй тип
        SPoint sp = new SPoint(0, 0);
        SPoint sp1 = new SPoint(0, 0);
        Console.WriteLine("Equals(sp, sp1) =  {0}", Equals(sp, sp1)); //true
        //выполняется сравнение значений полей
```

```
        }
    }
}
```

```
ReferenceEquals(p, p1)= False
ReferenceEquals(p1, p2)= True
ReferenceEquals(p3, p3) =  False
Equals(cp, cp1) =  False
Equals(sp, sp1) =  True
```

При создании собственного типа оператор Equals() можно перегрузить. Для ссылочных типов согласно рекомендации MSDN перегрузку следует выполнять, только если тип представляет собой неизменяемый объект. Например, для типа String, который содержит в себе неизменную строку, имеется перегруженный метод Equals() и оператор ==.

Поскольку в System.ValueType перегруженный метод Equals() выполняет побитовое сравнение, то в собственных значимых типах его можно не перегружать. Однако, в System.ValueType получение значений полей для сравнения в методе Equals() выполняется с помощью рефлексии, что приводит к снижению производительности. Поэтому при разработке значимого типа для увеличения быстродействия рекомендуется выполнить перегрузку метода Equals().

При перегрузке метода Equals() следует также перегружать метод GetHashCode(). Этот метод предназначен для получения целочисленного значения хеш - кода объекта. Причем, различным (т.е. не равным между собой) объектам должны соответствовать различные хеш – коды. Если перегрузку метода GetHashCode() не выполнить возникнет предупреждение компилятора.

Перегрузка оператора == обычно выполняется путем вызова метода Equals().

Если предполагается сравнивать экземпляры собственного типа для целей сортировки, рекомендуется унаследовать этот тип от интерфейсов System.IComparable и System.IComparable<T> и реализовать метод CompareTo(). В дальнейшем этот метод можно

вызывать из реализации Equals() и возвращать true, если CompareTo() возвращает 0.

Пример перегрузки операторов == и != для класса CPoint:

```
using System;

namespace ComparisonOperator
{
    using System;
    class CPoint
    {
        int x, y;
        public CPoint(int x, int y)
        {
            this.x = x; this.y = y;
        }
        //Перегрузка метода Equals
        public override bool Equals(object obj)
        {
            //если obj == null,
            //значит он != объекту, от имени которого вызывается этот метод
            if (obj == null) return false;
            CPoint p = obj as CPoint;
            //переданный объект не является ссылкой на CPoint
            if (p == null) return false;
            //проверяется равенство содержимого
            return ((x == p.x) && (y == p.y));
        }

        //При перегрузке Equals надо также перегрузить GetHashCode()
        public override int GetHashCode()
        {
            return x ^ y;//использование XOR для получения хеш кода
        }

        public static bool operator ==(CPoint p1, CPoint p2)
        {
            //проверка, что переменные ссылаются на один и тот же адрес
            //сравнение p1 == p2 приведет к бесконечной рекурсии
            if (ReferenceEquals(p1, p2)) return true;
            //приведение к object необходимо,
            //т.к. сравнение p1 == null приведет к бесконечной рекурсии
            if ((object)p1 == null) return false;
            return p1.Equals(p2);
        }

        public static bool operator !=(CPoint p1, CPoint p2)
        {
            return (!(p1 == p2));
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            CPoint cp = new CPoint(0, 0);
            CPoint cp1 = new CPoint(0, 0);
```

```
            CPoint cp2 = new CPoint(1, 1);
            Console.WriteLine("cp == cp1: {0}", cp == cp1); //true
            Console.WriteLine("cp == cp1: {0}", cp == cp2); //false
        }
    }
}
```

17. Перегрузка операторов true и false

При перегрузке операторов true и false разработчик задает критерий истинности для своего типа данных. После этого объекты типа напрямую можно использовать в структуре операторов if, do, while, for в качестве условных выражений.

Перегрузка выполняется по следующим правилам:

✓ оператор true должен возвращать значение true, если состояние объекта истинно и false в противном случае.

✓ оператор false должен возвращать значение true, если состояние объекта ложно и false в противном случае.

✓ операторы true и false надо перегружать в паре

При этом возможна ситуация, когда состояние не является не истинным ни ложным, т.е. оба оператора могут вернуть результат false.

В версиях до C# 2.0 отсутствовали значимые типы данных, допускающие присвоение null. Для создания пользовательских нулевых типов значений использовалась перегрузка операторов true и false. Наиболее удачно объяснение перегрузки этих операторов рассматривается в MSDN на примере класс BDNull. В этом классе может содержать одно из 3 значений:

- 1 (True)
- -1 (False)
- 0 (Null).

При перегрузке операторов true и false использовалась следующая таблица истинности:

Значение	Оператор True	Оператор False
1	true	False

-1	false	True
0	false	false

```
public struct DBBool
{
    //три возможных значения
    // Значение параметра может быть: - 1(false), 1 - true и 0 - null.
    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);

    sbyte value;

    DBBool(int value)
    {
        this.value = (sbyte)value;
    }

    public DBBool(DBBool b)
    {
        this.value = (sbyte)b.value;
    }

    // Возвращает true, если в операнде содержится True,
    // иначе возвращает false
    public static bool operator true(DBBool x)
    {
        return x.value > 0;
    }
    // Возвращает true, если в операнде содержится False,
    // иначе возвращает false
    public static bool operator false(DBBool x)
    {
        return x.value < 0;
    }
}
class Test
{
    static void Main()
    {
        DBBool b1 = new DBBool(DBBool.True);
        if (b1) Console.WriteLine("b1 is true");
           else Console.WriteLine("b1 is not true");
    }
}
```

Как видно из реализации, если в объекте класса содержится значение null, то оба оператора возвращают значение false.

Начиная со 2-ой версии, язык предоставляет встроенную поддержку для нулевых типов значений и по возможности следует использовать их вместо перегрузки операторов true и false.

18. Перегрузка логических операторов

Условные логические операторы && и || нельзя перегрузить, но они вычисляются с помощью & и |, допускающих перегрузку.

В класса DBBool для перегрузки логических операторов предлагается следующая таблица истинности:

Операнд 1	Операнд 2	&	|
True	True	True	True
False	True	False	True
Null	True	Null	True
True	False	False	True
False	False	False	False
Null	False	False	Null
True	Null	Null	True
False	Null	False	Null
Null	Null	Null	Null

```
using System;

public struct DBBool
{
    //содержимое класса DBBool из предыдущего примера

    // Оператор Логическое И. Возвращает:
    // False, если один из операндов False независимо от 2-ого операнда
    // Null, если один из операндов Null, а другой Null или True
    // True, если оба операнда True
    public static DBBool operator &(DBBool x, DBBool y)
    {
        return new DBBool(x.value < y.value ? x.value : y.value);
    }

     // Оператор Логическое ИЛИ. Возвращает:
    // True, если один из операндов True независимо от 2-ого операнда
    // Null, если один из операндов Null, а другой False или Null
    // False, если оба операнда False
    public static DBBool operator |(DBBool x, DBBool y)
    {
        return new DBBool(x.value > y.value ? x.value : y.value);
    }

    // Оператор Логической инверсии. Возвращает:
```

```
    // False, если операнд содержит True
    // True, если операнд содержит False
    // Null, если операнд содержит Null
    public static DBBool operator !(DBBool x)
    {
        return new DBBool(-x.value);
    }

    public override string ToString()
    {
        if (value > 0) return "DBBool.True";
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}

class Test
{
    static void Main()
    {
        DBBool bTrue = new DBBool(DBBool.True);
        DBBool bNull = new DBBool(DBBool.Null);
        DBBool bFalse = new DBBool(DBBool.False);

        Console.WriteLine("bTrue && bNull is {0}", bTrue && bNull);
        Console.WriteLine("bTrue && bFalse is {0}", bTrue && bFalse);
        Console.WriteLine("bTrue && bTrue is {0}", bTrue && bTrue);
        Console.WriteLine();

        Console.WriteLine("bTrue || bNull is {0}", bTrue || bNull);
        Console.WriteLine("bFalse || bFalse is {0}", bFalse || bFalse);
        Console.WriteLine("bTrue || bFalse is {0}", bTrue || bFalse);
        Console.WriteLine();

        Console.WriteLine("!bTrue is {0}", !bTrue);
        Console.WriteLine("!bFalse is {0}", !bFalse);
        Console.WriteLine("!bNull is {0}", !bNull);
    }
}
```

```
bTrue && bNull is DBBool.Null
bTrue && bFalse is DBBool.False
bTrue && bTrue is DBBool.True

bTrue || bNull is DBBool.True
bFalse || bFalse is DBBool.False
bTrue || bFalse is DBBool.True

!bTrue is DBBool.False
!bFalse is DBBool.True
!bNull is DBBool.Null
```

19. Перегрузка операторов преобразования

В собственных типах можно определить операторы, которые будут использоваться для выполнения приведения.

Приведение может быть 2 типов:

- ✓ из произвольного типа в собственный тип

- ✓ из собственного типа в произвольный тип

В отличие от C++ конструктор с 1-им параметром не используется для преобразования произвольного типа в собственный тип.

Для ссылочных и значимых типов приведение выполняется одинаково.

Приведение может выполняться явным и неявным образом. Явное приведение типов требуется, если возможна потеря данных в результате приведения. Например:

- при преобразовании int в short, потому что размер short недостаточен для сохранения значения int.
- при преобразовании типов данных со знаком в беззнаковые может быть получен неверный результат, если переменная со знаком содержит отрицательное значение.
- при конвертировании типов с плавающей точкой в целые дробная часть теряется.
- при конвертировании типа, допускающего null-значения, в тип, не допускающий null, если исходная переменная содержит null, генерируется исключение.

Если потери данные в результате приведения не происходит приведение можно выполнять как неявное.

Операция приведения должна быть помечена либо как implicit, либо как explicit, чтобы указать, как ее предполагается использовать:

- ✓ implicit задает неявное преобразование, его можно использовать, если преобразование всегда безопасно независимо от значения переменной, которая преобразуется
- ✓ explicit задает явное преобразование, его следует использовать, если возможна потеря данных или возникновение исключения

Объявление оператора преобразования в классе:

public static { implicit |explicit} operator <целевой тип> (исходный тип)

Пример:

- приведение CPoint к типу int с потерей точности, к типу double без потери точности. В качестве результата возвращается расстояние от точки до начала координат.

- приведение типа int к CPoint без потери точности, типа double к CPoint с потерей точности. В качестве результата возвращается точка, содержащее заданное значение в качестве значений координат.

```
using System;

namespace CastOperator
{
    class CPoint
    {
        int x, y;
        public CPoint(int x, int y)
        {
            this.x = x;
            this.y = y;
        }
        //может быть потеря точности, преобразование должно быть явным
        public static explicit operator int(CPoint p)
        {
            return (int)Math.Sqrt(p.x * p.x + p.y * p.y);
            //можно и так: return (int)(double)p;
        }
        //преобразование без потери точности, может быть неявным
        public static implicit operator double(CPoint p)
        {
            return Math.Sqrt(p.x * p.x + p.y * p.y);
        }
        //переданное значение сохраняется в x и y координате,
        //преобразование без потери точности, может быть неявным
        public static implicit operator CPoint(int a)
        {
            return new CPoint(a, a);
        }
        //преобразование с потерей точности, должно быть явным
        public static explicit operator CPoint(double a)
        {
            return new CPoint((int)a, (int)a);
        }
        public override string ToString()
        {
            return string.Format("X = {0} Y = {1}", x, y);
        }
    }

  class Test
  {
     static void Main()
     {
       CPoint p = new CPoint(2, 2);
       int a = (int)p; //выполнение явного преобразования CPoint в int
       double d = p;   //выполнение неявного преобразования CPoint в double
       Console.WriteLine("p as int: {0}", a); //2
```

```
            Console.WriteLine("p as double: {0:0.0000}", d);   //2.8284
            p = 5; //выполнение неявного преобразования int в CPoint
            Console.WriteLine("p: {0}", p); //x = 5 y = 5
            p = (CPoint)2.5; //выполнение явного преобразования double в CPoint
            Console.WriteLine("p: {0}", p); //x = 2 y = 2
        }
    }
}
```

```
p as int: 2
p as double: 2,8284
p: X = 5 Y = 5
p: X = 2 Y = 2
```

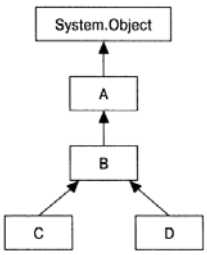
Приведение между классами

Имеется возможность выполнять приведение между экземплярами разных собственных структур или классов. Однако при этом существуют следующие ограничения:

✓ нельзя определить приведение между классами, если один из них является наследником другого;

✓ приведение может быть определено только в одном из типов: либо в исходном типе, либо в типе назначения.

Например, имеется следующая иерархия классов:



Единственное допустимое приведение типов – это приведения между классами C и D, потому что эти классы не наследуют друг друга. Код таких приведений может выглядеть следующим образом:

public static explicit operator D(C value) {…}

public static explicit operator C(D value) {…}

Эти операторы могут быть внутри определения класса C или же внутрь определения класса D. Если приведение определено внутри

одного класса, то нельзя определить такое же приведение внутри другого.

20. Что такое свойства

Думаю, неплохой подводкой к данной теме будет напоминание одного из основных принципов объектно-ориентированного программирования – инкапсуляции. Этот принцип позиционирует защиту данных класса от несанкционированного доступа вызывающей стороной (пользователем).

Для регулирования уровня доступа к данным классов в C# используются модификаторы доступа: private, protected, public. Поля класса, помеченные модификатором public, имеют самый высокий уровень доступности – через объект класса. Модификатор protected используется для открытия доступа к полям-членам из унаследованных классов, но при этом доступ через объект остаётся невозможным. Модификатор private применяется для жёсткого ограничения доступа к данным класса отовсюду, кроме как из методов этого же класса.

В подавляющем большинстве случаев данные классов помечаются модификатором private, и тогда доступ к ним можно получить только при помощи методов. Ранее (вы могли наблюдать это в C++) практиковалось назначение в классе двух методов – одного для определения значения приватного поля класса, второго – для считывания этого значения. Если изобразить эту схему, пользуясь языком C#, это будет выглядеть так:

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

namespace ConsoleApplication1
{
    class Class
    {
        private int num;
        //метод для установления значения переменной num
```

```
            public void Set(int num)
            {
                this.num = num;
            }
            //метод для считывания значения переменной num
            public int Get()
            {
                return num;
            }
        }
        class Tester
        {
            static void Main(string[] args)
            {
                Class obj1 = new Class();
                obj1.Set(int.Parse(Console.ReadLine()));
                Console.WriteLine(obj1.Get());
            }
        }
    }
```

Но C# предлагает более гибкий способ – это применение так называемых свойств (properties). Свойства относят к группе функций – членов класса. Они представляют собой метод, который обеспечивает как возможность присваивания значений приватным данным классов, так и считывание их значений, по необходимости. То есть свойства интегрируют возможности двух разных по назначению групп методов. “Два метода в одном флаконе”.

У тех, кто только начинает вникать в тонкости программирования, часто возникает вопрос: а зачем вообще делать данные-члены класса приватными и писать отдельные методы для доступа к ним, если можно просто делать все данные открытыми, и тогда код станет более облегчённым, и доступ к данным становится более простым. Если Вы в числе этих людей, то советую ещё раз перечитать материалы по основным концепциям ООП. А, в свою очередь, напомню, что неограниченный доступ к данным – палка о двух концах. Вспомните глобальные переменные в Си. Данные можно просто-напросто испортить. Поэтому, по возможности, необходимо проверять правильность задаваемых полям классов значений. Проверка может

быть самой разнообразной: формат задаваемых значений, целесообразность, соответствие определённым ограничениям (например, принадлежность заданному диапазону). Соответствующая проверка данных должна осуществляться как можно раньше, и при необходимости должна последовать незамедлительная реакция по устранению всех ”неправильностей”. Если данные открыты, то при многократном обращении к ним, переприсваивании значений, можно забыть произвести необходимую проверку данных, и данные могут быть испорчены. Я даже могу сказать, что они наверняка будут испорчены, если не Вами, то невнимательным пользователем точно.

Когда поля приватные, а для задания значений полям у нас есть специальный метод, нам достаточно произвести всю необходимую проверку данных внутри этого метода, и тогда при каждом обращении к этому методу задаваемые значения будут автоматически “просеиваться”.

Надеюсь, сейчас вопрос удобства использования специальных методов для доступа к закрытым полям класса можно считать решённым. Тогда вернёмся к специализированным методам C# - свойствам.

21. Синтаксис объявления свойств.

Мы сказали, что свойства объединяют возможности считывания и присваивания значений полям класса. Это достигается при помощи так называемых методов-аксессоров get и set, которые, в свою очередь помещаются внутрь блока кода, по синтаксису напоминающего функцию, но без параметров.

```
public int Var
{
    get { return var; }
    set { var = value; }
}
```

Метод – аксессор set позволяет присваивать значение переменной класса, а метод - аксессор get – считывать это значение.

В целом всё свойство имеет модификатор доступа public, причём указывать его надо явно, далее идёт тип возвращаемого значения – в данном случае int. Само свойство рекомендуется именовать так же, как и поле, со значением которого оно работает, но в верхнем регистре. Это облегчает читаемость кода, делая идентификаторы более прозрачными. Но это всего лишь рекомендация, а не жёсткое правило.

Свойство не имеет параметров, т.е. параметры не указываются вообще (не путать с функциями с пустым списком параметров).

Аксессор get включает оператор return, с помощью которого значение поля класса может быть считанным.

Аксессор set имеет своеобразную запись, которая может быть не сразу понятна.

```
set { var = value; }
```

Здесь присутствует специальное слово value, которое ставится справа от знака присваивания. Заметьте, именно “специальное” слово. Оно не является ключевым в широком смысле. Скорее можно назвать его контекстным ключевым словом. То есть за пределами свойства оно может использоваться как обычный идентификатор, и при этом не будет возникать конфликта имён. Это слово обозначает любое значение соответствующего типа, присваиваемое переменной класса.

Именно в пределах данного аксессора может быть размещён весь код, нацеленный на проверку входных значений.

Например:

```
set
{
  if(value <=120 && value >= 0)
     num = value;
}
```

Здесь задаваемое значение проверяется на принадлежность диапазону от 0 до 120 включительно. Если всё благополучно, значение будет присвоено, иначе оно останется прежним.

Ещё один момент, на который следует обратить внимание – это модификатор доступа для каждого из аксессоров. Как видите, он явно не указан. По умолчанию модификаторы доступа для обоих аксессоров public, т.е. соответствует уровню доступа всего свойства в целом. Явно модификаторы доступа для аксесоров указываются только в ком случае, если они ниже по уровню, чем всё свойство. Причём явно указан может быть модификатор только для одного из аксессоров (тот, который ниже по уровню доступа).

Например:

```
public int Num
{
          get { return num; }
protected set { num = value; }
}
```

Для обоих аксессоров одновременно нельзя указывать модификаторы доступа. Т.е. если мы добавим модификатор public перед get в данном примере, то компилятор выдаст ошибку. Второй аксессор должен иметь модификатор доступа уровня свойства. Другими словами, необходимо задавать модификатор доступа только к наименее доступному аксессору свойства, а второй автоматически получает уровень доступа всего свойства – т.е. public.

Свойство может быть статическим. Это нужно в том случае, если свойство нужно для задания или считывания значения статического приватного поля класса.

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

```
using System.Text;

namespace ConsoleApplication1
{
    class Class
    {
        private static int num = 5;

        public static int Num
        {
             get { return num; }
             set { num = value; }
        }
    }
    class Tester
    {
        static void Main(string[] args)
        {
            Class.Num = 10;
            Console.WriteLine(Class.Num);
        }
    }
}
```

Также свойство может иметь только один аксессор, вместо двух. Это нужно в том случае, если нельзя менять значение приватного поля класса, т.е. это вариация на тему readonly полей.

Выглядит это так:

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

namespace ConsoleApplication1
{
    class Class
    {
        private readonly int num = 10;

        public int Num
        {
             get { return num; }
        }
    }
    class Tester
    {
        static void Main(string[] args)
        {
```

```
            Class obj1 = new Class();
            Console.WriteLine(obj1.Num);
        }
    }
}
```

В данном примере в классе есть одно приватное поле только для чтения. Поэтому в свойстве определён только один аксессор get для считывания значения поля класса. Если в это же свойство добавить аксессор set, компилятор выдаст ошибку:

A readonly field cannot be assigned to (except in a constructor or a variable initializer).

Того же эффекта можно добиться с помощью исключения аксессора set из свойства даже в том случае, если само поле класса не помечено как readonly. То есть можно сделать поле readonly, не указывая явно этот модификатор. Достаточно убрать из свойства аксессор set.

Нельзя разделять свойство на 2 части для каждого из аксесоров, т.е. следующая запись является недопустимой:

```
class Class
{
    private int num = 10;

    public int Num
    {
        get { return num; }
    }

     //Error
    public int Num
    {
        set { num = value; }
    }

}
```

На подобное безобразие компилятор возмутится:

The type 'ConsoleApplication1.Class' already contains a definition for 'Num'

и подчеркнёт вторую “часть” свойства.

Свойства назначаются для каждого поля класса, независимо от их типа. При этом нужно не забывать указывать соответствующий тип возвращаемого значения для каждого из свойств.

22. Примеры использования свойств.

Рассмотрим следующий пример. В нём отображён класс Employee с четырьмя приватными полями, обозначающими имя, фамилию, возраст и зарплату сотрудника. Класс также включает два перегруженных конструктора (с параметрам и без параметров). Для каждого из полей класса предусмотрено открытое свойство с двумя аксессорами. В свойствах осуществляется дополнительная проверка задаваемых значений. Имя и фамилия приводятся к верхнему регистру, возраст проверяется на принадлежность интервалу допустимых значений, зарплат не может быть отрицательной величиной. Перегруженный метод ToString() позволяет распечатать состояние объекта.

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

namespace ConsoleApplication1
{
    class Emoloyee
    {
        private string firstName;
        private string lastName;
        private int age;
        private float wage;

        public Emoloyee()
        {
        }
```

```
        public Emoloyee(string first, string last, int age, float
wage)
        {
            this.FirstName = first;
            this.LastName = last;
            this.Age = age;
            this.Wage = wage;
        }

        public string FirstName
        {
            get { return firstName != null ? firstName : "Not set"; }
            set { firstName = value.ToUpper(); }
        }

        public string LastName
        {
            get { return lastName != null ? lastName : "Not set"; ; }
            set { lastName = value.ToUpper(); }
        }

        public int Age
        {
            get { return age; }
            set { age = (value>100 || value<1) ? 0 : value; }
        }
        public float Wage
        {
            get { return wage; }
            set { wage = value<0 ? 0 : value; }
        }
        public override string ToString()
        {
            return string.Format("First name: {0}\nLast name:
{1}\nAge:
                    {2}\nWage: {3}\n",
                  this.FirstName, this.LastName, this.Age, this.Wage);
        }
    }
    class Tester
    {
        static void Main(string[] args)
        {
           Emoloyee emp1 = new Emoloyee("Oleg", "Sikorsky", 29,
4800F);
            Emoloyee emp2 = new Emoloyee();
            emp2.FirstName = "Daniel";
            //Last name не установлено
            //попытка присвоить невозможный возраст
            emp2.Age = 120;
            //попытка задать зарплату со знаком минус
            emp2.Wage = -1000;
            Emoloyee emp3 = new Emoloyee("Natali","Borisova",29,
2500F);
```

```
            Console.WriteLine(emp1.ToString());
            Console.WriteLine(emp2.ToString());
            Console.WriteLine(emp3.ToString());
        }
}
```

```
На выходе:
First name: OLEG
Last name: SIKORSKI
Age: 29
Wage: 4800

First name: DANIEL
Last name: Not set
Age: 0
Wage: 0

First name: NATALI
Last name: BORISOVA
Age: 29
Wage: 2500
```

Можем также просмотреть определение свойств данного класса на промежуточном языке. Для этого воспользуемся утилитой MSIL Disassembler.

```
.property instance string FirstName()
{
  .get instance string EmployeeValidation.Emoloyee::get_FirstName()
  .set instance void EmployeeValidation.Emoloyee::set_FirstName(string)
} // end of property Emoloyee::FirstName

.property instance string LastName()
{
  .set instance void EmployeeValidation.Emoloyee::set_LastName(string)
  .get instance string EmployeeValidation.Emoloyee::get_LastName()
} // end of property Emoloyee::LastName

.property instance int32 Age()
{
  .set instance void EmployeeValidation.Emoloyee::set_Age(int32)
  .get instance int32 EmployeeValidation.Emoloyee::get_Age()
} // end of property Emoloyee::Age
```

23. Понятие индексатора

А теперь ещё одно любопытное редство языка C#, представляющее собой одновременно способ перегрузки оператора [] (но без участия

ключевого слова operator) и разновидностью свойства (или как его ещё называют, свойтвом с парамертами). Индексаторы применяются для обращения к одним объектам так, словно они являются элементами класса пользовательской коллекции, а с другими – словно они и есть эта коллекция.

Объявление индексатора подобно свойству, но с той разницей, что индексаторы безымянные (вместо имени используется ссылка this) и что индексаторы включают параметры индексирования.

Синтаксис объявления индексатора следующий:

Тип this[тип аргумента] {get; set;}

Тип – это тип объектов коллекции. This - это ссылка на объект, в котором появляется индексатор. Тип аргумента представляет индекс объекта в коллекции, причём этот индекс необязательно целочисленный, как мы привыкли, он может быть любого типа.

У каждого индексатора должен быть минимум один параметр, но их может быть и больше (напоминает многомерные массивы).

То, что для индексаторов используется синтаксис со ссылкой this, подчёркивает, что нельзя использовать их иначе, как на экземплярном уровне.

24. Создание одномерного индексатора

Рассмотрим пример создания и применения индексатора. Предположим, есть некий магазин, занимающийся реализацией ноутбуков. Эта ситуация отображается при помощи двух классов: класса Shop, изображающего магазин, и класса Laptop, изображающего его продукцию. Дабы не перегружать пример лишней информацией, снабдим класс Laptop только двумя полями: vendor – для отображения имени фирмы-производителя, а также price – для отображения цены ноутбука. Наш класс будет включать соответствующие открытые свойства Vendor и Price, конструктор с двумя параметрами, а также переопределённый

метод ToString() для отображения информации по конкретной единице товара. В качестве единственного поля класса Shop выступает ссылка на массив объектов Laptop. В конструкторе с одним параметром задаётся количество элементов массива и выделяется память для их хранения. Далее нам нужно сделать возможным обращение к элементам этого массива через экземпляр класса Shop, пользуясь синтаксисом массива так, словно класс Shop и есть массив элементов типа Laptop.

Для этого мы добавляем в класс Shop индексатор.

```
public Laptop this[int pos]
 {
     get
     {
         if (pos >= LaptopArr.Length || pos < 0)
             throw new IndexOutOfRangeException();
         else
             return (Laptop)LaptopArr[pos];
     }
     set
     {
         LaptopArr[pos] = (Laptop)value;
     }
 }
```

Здесь в аксессоре get мы предусматриваем выход за пределы массива, и при этом генерируется исключение `IndexOutOfRangeException.`

Для проверки работы наших классов создаётся отдельный класс Tester, содержащий точку входа. В нём создаётся экземпляр класса Shop, причём в конструкторе мы задаём количество элементов, которые в нём можно разместить.

```
Shop sh = new Shop(3);
```

Далее мы заполняем этот массив объектами Laptop.

```
sh[0] = new Laptop("Samsung", 5200);
sh[1] = new Laptop("Asus", 4700);
sh[2] = new Laptop("LG", 4300);
```

И, наконец, выводим на экран данные по каждому объекту Laptop, пользуясь синтаксисом массива.

```
for (int i = 0; i < 3; i++)
```

```
Console.WriteLine(sh[i].ToString());
```

Заметим, что в цикле ограничивающим значением в условии является явно заданное число 3. Дело в том, что индексатор позволяет нам пользоваться лишь синтаксисом индексирования массива, но других функциональных возможностей массива не предоставляет. Будь это стандартный массив, мы бы оформили условие иначе:

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

Другими словами, мы бы воспользовались свойством массива Length.

Для того, чтобы подобная функциональная возможность появилась и в нашем примере, необходимо добавить в класс Shop дополнительное свойство Length.

```
public int Length
 {
     get { return LaptopArr.Length; }
 }
```

Теперь рассмотрим код программы целиком.

```
using System;
using System.Collections;
namespace NS
{
    public class Laptop
    {
        private string vendor;
        private double price;


        public string Vendor
        {
            get { return vendor; }
            set { vendor = value; }
        }
        public double Price
        {
            get { return price; }
            set { price = value; }

        }
        public Laptop(string v, double p)
        {
            vendor = v;
            price = p;
```

```
        }

        public override string ToString()
        {
            return vendor + " " + price.ToString();
        }
    }

    public class Shop
    {
        private Laptop[] LaptopArr;

        public Shop(int size)
        {
            LaptopArr = new Laptop[size];
        }
        public int Length
        {
            get { return LaptopArr.Length; }
        }

        public Laptop this[int pos]
        {
            get
            {
                if (pos >= LaptopArr.Length || pos < 0)
                    throw new IndexOutOfRangeException();
                else
                    return (Laptop)LaptopArr[pos];
            }
            set
            {
                LaptopArr[pos] = (Laptop)value;
            }
        }
    }
    public class Tester
    {
        public static void Main()
        {
            Shop sh = new Shop(3);
            sh[0] = new Laptop("Samsung", 5200);
            sh[1] = new Laptop("Asus", 4700);
            sh[2] = new Laptop("LG", 4300);

            try
            {
                for (int i = 0; i < sh.Length; i++)
                    Console.WriteLine(sh[i].ToString());
                Console.WriteLine();
            }
            catch(System.NullReferenceException)
            {
            }
        }
```

```
        }
    }
```

На выходе получается:

```
Samsung 5200
Asus 4700
LG 4300
```

Казалось бы, всё успешно работает. Однако уже небольшое изменение в коде, заключающееся в применении другого цикла для выведения информации об объектах приведёт к ошибке на этапе компиляции.

Error 1 foreach statement cannot operate on variables of type 'NS.Shop' because 'NS.Shop' does not contain a public definition for 'GetEnumerator'

Для устранения этой проблемы необходимо применить наследование интерфейса `IEnumerable` для класса Shop и явно определить его метод `GetEnumerator()`:

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

Тема интерфейсов будет рассматриваться подробно в следующих уроках, а пока просто скажем, что для возможности применения цикла foreach коллекция должна содержать данный метод, а его реализация есть в интерфейсе `IEnumerable.` Для более детальной информации об `IEnumerable,` можно обратиться к MSDN.

После внесённых поправок наш код выглядит следующим образом:

```
using System;
using System.Collections;

namespace NS
{
    public class Laptop
    {
        private string vendor;
        private double price;


        public string Vendor
        {
```

```
            get { return vendor; }
            set { vendor = value; }
        }

        public double Price
        {
            get { return price; }
            set { price = value; }

        }

        public Laptop(string v, double p)
        {
            vendor = v;
            price = p;
        }

        public override string ToString()
        {
            return vendor + " " + price.ToString();
        }
    }


    public class Shop : IEnumerable
    {
        private Laptop[] LaptopArr;

        public Shop(int size)
        {
            LaptopArr = new Laptop[size];
        }

        public int Length
        {
            get { return LaptopArr.Length; }
        }

        public Laptop this[int pos]
        {
            get
            {
                if (pos >= LaptopArr.Length || pos < 0)
                    throw new IndexOutOfRangeException();
                else
                    return (Laptop)LaptopArr[pos];
            }
            set
            {
                LaptopArr[pos] = (Laptop)value;
            }
        }

        #region IEnumerable Members
```

```
            IEnumerator IEnumerable.GetEnumerator()
            {
                return LaptopArr.GetEnumerator();
            }

            #endregion
        }
        public class Tester
        {
            public static void Main()
            {
                Shop sh = new Shop(3);
                sh[0] = new Laptop("Samsung", 5200);
                sh[1] = new Laptop("Asus", 4700);
                sh[2] = new Laptop("LG", 4300);

                try
                {
                    foreach (Laptop LT in sh)
                        Console.WriteLine(LT.ToString());

                }
                catch (System.NullReferenceException)
                {
                }
                Console.WriteLine();
            }
        }
    }
```

25. Создание многомерных индексаторов

В C# есть возможность создавать не только одномерные, но и многомерные индексаторы. Это возможно, если класс-контейнер содержит в качестве поля массив с более чем одним измерением. Для демонстрации такой возможности приведём схематичный пример, не перегруженный дополнительными проверками.

```
using System;
using System.Collections;
namespace NS
{
    public class A
    {
        private int[,] arr;
```

```
            private int rows, cols;

            public int Rows
            {
                get { return rows; }
            }

            public int Cols
            {
                get { return cols; }
            }

             public A(int rows, int cols)
             {
                 this.rows = rows;
                 this.cols = cols;
                 arr = new int[rows, cols];
             }

            public int this[int r, int c]
            {
                get { return arr[r, c]; }
                set { arr[r, c] = value; }

            }
        }
        public class Tester
        {
            static void Main()
            {
                A obj = new A(2, 3);


                for (int i = 0; i < obj.Rows; i++)
                {
                    for (int j = 0; j < obj.Cols; j++)
                    {
                        obj[i, j] = i + j;
                        Console.Write(obj[i, j].ToString());
                    }
                    Console.WriteLine();
                }
            }
        }
    }
```

26. Перегрузка индексаторов

Как мы отмечали ранее, тип может поддерживать различные перегрузки индексаторов при условии, что они отличаются сигнатурой. Это значит, что тип параметра индексатора может быть не только целочисленным, но вообще любым. Например, можно добавить в наш класс индексатор для поиска по имени производителя (vendor). Для этого понадобится дополнительный метод Find для поиска индекса элемента с заданным именем.

```
private int Find(string name)
{
    for (int i = 0; i < LaptopArr.Length; i++)
        if (LaptopArr[i].Vendor == name)
            return i;
    return -1;
}
```

А в индексаторе мы вызываем этот метод :

```
public Laptop this[string name]
{
    get
    {
        if (name.Length == 0)
            //return null;
            throw new System.Exception("Недопустимый индекс");
        return this[Find(name)];
    }
}
```

этот метод

Аналогичным путём мы добавляем в наш класс индексатор для поиска по цене (price). Теперь у нас есть 3 перегрузки индексатора, но они между собой не конфликтуют, покольку сигнатуры у них разные по типу. Вот что получилось в конечном счёте:

```
using System;
using System.Collections;
namespace NS
{
    public class Laptop
    {
        private string vendor;
```

```
        private double price;


        public string Vendor
        {
            get { return vendor; }
            set { vendor = value; }
        }
        public double Price
        {
            get { return price; }
            set { price = value; }

        }
        public Laptop(string v, double p)
        {
            vendor = v;
            price = p;
        }

        public override string ToString()
        {
            return vendor + " " + price.ToString();
        }
    }

    public class Shop : IEnumerable
    {
        private Laptop[] LaptopArr;

        public Shop(int size)
        {
            LaptopArr = new Laptop[size];
        }


        public int Length
        {
            get { return LaptopArr.Length; }
        }

        public Laptop this[int pos]
        {
            get
            {
                if (pos >= LaptopArr.Length || pos < 0)
                    throw new IndexOutOfRangeException();
                else
                    return (Laptop)LaptopArr[pos];
            }
            set
            {
                LaptopArr[pos] = (Laptop)value;
            }
        }
```

```
        private int Find(string name)
        {
            for (int i = 0; i < LaptopArr.Length; i++)
                if (LaptopArr[i].Vendor == name)
                    return i;
            return -1;
        }

        public Laptop this[string name]
        {
            get
            {
                if (name.Length == 0)
                    //return null;
                    throw new System.Exception("Недопустимый
                     индекс");
                return this[Find(name)];
            }
        }

        public int FindByPrice(double price)
        {
            for (int i = 0; i < LaptopArr.Length; i++)
                if (LaptopArr[i].Price == price)
                    return i;
            return -1;
        }

        public Laptop this[double price]
        {
            get
            {
                if (price == 0)
                    throw new System.Exception("Недопустимый
                    индекс");
                return this[FindByPrice(price)];
            }
        }

        #region IEnumerable Members

        IEnumerator IEnumerable.GetEnumerator()
        {
            return LaptopArr.GetEnumerator();
        }

        #endregion
    }
    public class Tester
    {
        public static void Main()
        {
            Shop sh = new Shop(3);
```

```
                    sh[0] = new Laptop("Samsung", 5200);
                    sh[1] = new Laptop("Asus", 4700);
                    sh[2] = new Laptop("LG", 4300);


                    try
                    {
                        foreach (Laptop lt in sh)
                            Console.WriteLine(lt.ToString());
                        Console.WriteLine();
                        Console.WriteLine(sh["LG"]);
                        Console.WriteLine();
                        Console.WriteLine(sh[4700.0]);

                    }
                    catch (IndexOutOfRangeException ex)
                    {
                        Console.WriteLine(ex.Message);
                    }

                }
            }
        }
```

Домашнее задание

1. Построить диаграмму классов и разместить на ней все классы исключений, определенные в MSCorLib.dll

2. Разработать собственный структурный тип данных для хранения целочисленных коэффициентов A и B линейного уравнения A×X + B×Y = 0. В классе реализовать статический метод Parse, которые принимает строку со значениями коэффициентов, разделенных запятой или пробелом. В случае передачи в метод строки недопустимого формата генерируется исключение FormatException.

3. Разработать метод для решения системы 2 линейных уравнений:

A1×X + B1×Y = 0

A2×X + B2×Y = 0

Метод с помощью выходных параметров должен возвращать найденное решение или генерирует исключение ArgumentOutOfRangeException, если решение не существует.

4. Разработайте приложение “7 чудес света”, где каждое чудо будет представлено отдельным классом. Создайте дополнительный класс, содержащий точку входа. Распределите приложение по файлам проекта и с помощью пространства имён обеспечьте возможность взаимодействия классов.

5. Разработать приложение, в котором бы сравнивалось население трёх столиц из разных стран. Причём страна бы обозначалась пространством имён, а город - классом в данном пространстве.

6. Реализовать класс для хранения комплексного числа. Выполнить в нем перегрузку всех необходимых операторов для успешной компиляции следующего фрагмента кода:

```
Complex z = new Complex(1,1);
Complex z1;
z1 = z - (z * z * z - 1) / (3 * z * z);
Console.WriteLine("z1 =  {0}", z1);
```

Краткая справка по комплексным числам (из Викепедии)

Любое комплексное число может быть представлено как формальная сумма $x + iy$, где x и y — вещественные числа, i — мнимая единица, то есть число, удовлетворяющее уравнению $i^2 = -1$

Действия над комплексными числами

- Сравнение

  $a + bi = c + di$ означает, что $a = c$ и $b = d$ (два комплексных числа равны между собой тогда и только тогда, когда равны их действительные и мнимые части).

- Сложение

  $(a + bi) + (c + di) = (a + c) + (b + d)i$

- Вычитание

  $(a + bi) - (c + di) = (a - c) + (b - d)i$

- Умножение

  $(a + bi)(c + di) = ac + bci + adi + bdi^2 = (ac - bd) + (bc + ad)i$

- Деление

$$\frac{(a + bi)}{(c + di)} = \left(\frac{ac + bd}{c^2 + d^2}\right) + \left(\frac{bc - ad}{c^2 + d^2}\right) i$$

7. Разработать класс Fraction, представляющий простую дробь. В классе предусмотреть два поля: числитель и знаменатель дроби. Выполнить перегрузку следующих операторов: +,-,*,/,==,!=,<,>, true и false.

Арифметические действия и сравнение выполняется в соответствии с правилами работы с дробями. Оператор true возвращает true если дробь правильная (числитель меньше знаменателя), оператор false возвращает true если дробь неправильная (числитель больше знаменателя).

Выполнить перегрузку операторов, необходимых для успешной компиляции следующего фрагмента кода:

```
Fraction f = new  Fraction(3,4);
int a = 10;
Fraction f1 =  f  * a;
Fraction f2 =  a * f;
double d = 1.5;
Fraction f3 = f  + d;
```