Программирование на языке C#.

Содержание.

Глава 1. Обзор платформы MS.NET.

Microsoft®.NET Framework содержит все инструменты и технологии для построения web-приложений. Отличается независимостью от языка программирования, непротиворечивой программной моделью через все уровни приложения, предоставляет хорошие условия для взаимодействию и легкое обновление со старых технологий. Платформа MS.NET полностью поддерживает технологии интернета, HTTP, XML и e-mail.

C # - новый язык, определенно разработанный для построения .NET приложений. Как разработчикам, вам будет интересно узнать связи и свойства определяющие основу платформы MS.NET.

Платформа MS.NET включает в себя несколько узловых понятий.

- MS.NET Framework – основан на общеязыковой среде исполнения. Выполняет программный код независимо от языка.
- MS.Net My Services – сервисы, приложения которые могут обращаться напрямую друг к другу через интернет и использовать методы друг друга.
- Серверы предприятий – хранят данные, обеспечивают масштабируемость, надежность, управление.
- Visual Studio – инструмент разработки в среде MS.NET

Далее рассмотрим подробнее MS.NET Framework. Необходимая платформа для запуска MS.NET Framework– Microsoft Windows. При запуске приложения в Windows ему доступны все сервисы Windows, такие как IIS, WMI, Component services, Message Queuing. Платформа MS.NET обеспечивает доступ к ним через классы библиотеки MS.NET. Состоит из следующих элементов:

Общеязыковая среда выполнения – упрощает разработку приложений поддерживает различные языки, обеспечивает безопасное выполнение кода, контролирует ресурсы.

Библиотека классов – содержит огромное количество классов, с возможностью расширения под нужды разработчика.

ADO.NET – обеспечивает поддержку работы с базами данных и XML

ASP.NET – инструмент для разработки web-приложений.

XML Web сервисы – программируемые web компоненты, которые доступны различным приложениям

User Interface – пользовательский интерфейс, поддерживает три типа web, windows form и консольные приложения.

Платформа MS.NET предоставляет следующие преимущества:

- Поддержка всех web стандарты такие как HTML, XML, SOAP, XSLT, XML Path Language и другие. Создаёт и поддерживает XML Web сервисы.
- Одинаковая среда для всех языков программирования.
- Легко использовать при разработке, все классы собраны в иерархические пространства имен, поддержка общей системы типов.
- Легко расширяемые классы – иерархия классов не скрыта от разработчика, он может включать в любое её место свои новые унаследованные классы.

Общеязыковая среда выполнения.

Общеязыковая среда выполнения упрощает разработку приложений, обеспечивает правильное и безопасное выполнение, поддерживает мультиязычность и следит за ресурсами. Эта среда также называется управляемой средой, в которой автоматически работают такие сервисы, как сборщик мусора и сервис безопасности. Далее опишем остальные элементы:

- Загрузчики классов – управляет метаданными, загружает и располагает классы в памяти.
- Перевод из языка MSIL в машинные коды при выполнении (Just-in-time).
- Сборщик мусора – следит за неиспользуемыми объектами и при недостатке памяти очищает их.
- Ядро отладки – позволяет отлаживать и трассировать код.

- Проверка типов – недопускает опасных преобразований типов
- Обработка исключений – структурированная система классов исключений интегрированная с Windows.
- Поддержка COM-объектов
- Поддержка базовых классов интегрированных в среду

Библиотека классов .Net Framework увеличивает мощность среды выполнения и обеспечивает высоко-уровненевые сервисы необходимые при программировании. Библиотека классов разбита на иерархию пространств имен. Представим некоторые из них.

Пространство имен System – содержит фундаментальные классы и определяет обычно используемые типы данных, события, интерфейсы, атрибуты и исключения. Также классы преобразования данных, манипуляций с параметрами

System.Collections – содержит списки, хэш-таблицы и другие виды группировки данных.

ADO.NET – следующее поколение технологии ADO, обеспечивают расширенную поддержку отсоединенной программной модели БД. Кроме того поддерживает работу с XML. Классы для работы с ADO.NET находятся в System.Data, с XML – в пространстве имен System.XML.

ASP.NET – программная структура, основанная на среде выполнения, работающей на сервере, которая позволяет создавать мощные Web приложения. Классы расположены в System.Web. Обеспечивается поддержка XML Web сервисов, необходимых для распределенной разработки.

Пространство имен System.Windows.Forms используется для создания интерфейса клиента Большое количество функций ранее доступных только в API теперь есть в распоряжении разработчика. Пространство имен System.Drawing обеспечивает доступ к графике GDI+

Языки MS.NET Framework.

MS.NET Framework поддерживает множество языков, всего порядка двадцати, рассмотрим некоторые из них:

C# - разработан для платформы .NET, это первый современный объектно-ориентированный из семьи C и C++. Несколько из главных его свойств – это классы, интерфейсы, делегаты, пространства имен, свойтсва, индексаторы и т.д. Нет необходимости в заголовочном файле.

Управляемое расширение C++ - минимальное расширение C++, обеспечен доступ к .NET Framework, включая сборщик мусора, единственное наследование и т.д.

Visual Basic .NET улучшен по сравнению с предыдущей версией, добавлены наследование, конструкторы, полиморфизм, перегрузка конструкторов и т.д.

JScript.NET – полностью переписан для поддержки среды .NET, включает поддержку классов, наследования, типов и компиляции.

Visual J#.NET – инструмент разработки для Java-программистов, желающих создавать приложения и сервисы в среде .NET. Имеется возможность автоматически обновлять существующие проекты Visual J++ 6.0 в решения Visual Studio .NET.

Сторонние языки – некоторые языки программирования также поддерживаются платформой .NET, такие как APL, COBOL, Pascal, Oberon, Perl, Python и SmallTalk.

Вопросы к разделу.

1. Что такое платформа MS.NET.
2. Какие узловые технологии в платформе .NET
3. Список компонент MS.NET Framework.
4. Что такое общеязыковая среда выполнения.
5. Что такое управляемая среда.

Глава 2. Обзор языка C#.

В этом разделе будет изучена основная структура программ на C#. Рассмотрен класс Console для выполнения простых операций ввода и вывода. Будут даны советы по обработке исключений и документации кода.

Структура программы на C#

При изучении любого языка, первая программа, которую пишут обычно Hello, World. Рассмотрим как она выглядит в версии C#.

```
 Using System;
Class Hello
{
    public static void Main()
    {
          Console.WriteLine(“Hello, World”);
    }
}
```

Код примера содержит все необходимые элементы программы на C# и легко проверяем. При выполнении на экране появится надпись:

```
  Hello, World
```

В C# приложение это коллекция одного или нескольких классов, структур и других типов. Класс определяется как набор данных и методов работающих с ними.

При рассмотрении кода приложения Hello, World, видим что есть единственный класс названный Hello. После имени класса открываются фигурные скобки ({). Всё что находится до соответствующей закрывающейся скобки (}) часть класса.

Можно распределить один класс приложения C# на несколько файлов. Также можно поместить несколько классов в один файл.

Каждое приложение должно начинаться с какого-то оператора. В C# при запуске приложения начинает выполняться метод Main. Несмотря на то, что в приложении может быть много классов, точка входа всегда одна. Может быть несколько методов Main, но запускаться будет только один, он выбирается при компиляции. Синтаксис Main важен, если вы создали проект в Visual Studio, то он сгенерируется автоматически. При завершении метода Main приложение заканчивает работу.

Как часть MS.NET Framework, C# содержит много различных классов, которые упорядоченные в пространствах имен. Пространство имен – это набор связанных классов. Пространство имен может содержать вложенные пространства имен. Основное пространство имен System. К объектам пространств имен можно обращать при помощи полного имени используя префиксы. Например пространство System, содержит класс Console, который выполняет несколько методов, включая WriteLine

```
System.Console.WriteLine(“Hello, World”);
```

Директива using позволяет обращаться к членам пространства имен напрямую без использования полного имени.

```
using System;
Console.WriteLine(“Hello, World”);
```

Основные операции ввода/вывода

В этом разделе рассматрим класс Console и его методы ввода и вывода. Класс Console обеспечивает приложению C# доступ к стандартным поток ввода, вывода и ошибок. Стандартный поток ввода – клавиатура. Стандартный поток вывода и ошибок – экран. Эти потоки могут быть перенаправлены в файлы.

Write и WriteLine методы вывода информации на консоль. Они перегружаемы, то есть могут выводить как строки, так и числа.

```
Console.WriteLine(99);
Console.WriteLine("Hello, World");
```

Можно использовать форматирующую строку, в которой маркерами отмечены параметры после строки

```
Console.WriteLine("The sum of {0} and {1} is {2}", 100, 130, 100+130);
```

Можно использовать левое и правое выравнивание, задавать ширину вывода.

```
Console.WriteLine("\"Левое выравнивание в поле ширины 10: {0, -10}\"", 99);
Console.WriteLine("\"Правое выравнивание в поле ширины 10: {0,10}\"", 99);
```

Будет выведено:

“Левое выравнивание в поле ширины 10: 99 ”

“Правое выравнивание в поле ширины 10: 99”

Есть настройки для вывода числовых форматов:

```
Console.WriteLine("Валюта - {0:C} {1:C4}", 88.8, -888.8);
Console.WriteLine("Целое число - {0:D5}", 88);
Console.WriteLine("Экспонента - {0:E}", 888.8);
Console.WriteLine("Фиксированная точка - {0:F3}",
888.8888);
Console.WriteLine("Общий формат- {0:G}", 888.8888);
Console.WriteLine("Числовой формат - {0:N}", 8888888.8);
Console.WriteLine("Шестнадцатиричный - {0:X4}", 88);
```

Соответственно отобразят:

Валюта - $88.80 ($888.8000)

Целое число - 00088

Экспонента - 8.888000E+002

Фиксированная точка - 888.889

Общий формат - 888.8888

Числовой формат - 8,888,888.80

Шестнадцатеричный формат – 0058

Рекомендации по оформлению кода.

Главная рекомендация – комментирование кода. Комментирование программы позволяет разработчикам, не участвовавшим при написании, разобраться как работает приложение. Используйте полные и значимые имена для именования компонент. Хорошие комментарии объясняют не что написано в программе, а отвечают на вопрос, почему именно так. Если в вашей организации есть стандарты комментирования, придерживайтесь их.

C# поддерживает несколько вариантов комментирования кода: однострочные(//), многострочные комментарии(/* */) и XML-документация(///). В XML-документации можно использовать различные преопределенные теги. Для генерации файла используется параметр при компиляции кода из командной строки.

csc myprogram.cs /doc:mycomments.xml

Надежная программа на C# обязана обрабатывать непредвиденное. Независимо от того сколько различных ошибок вы предусмотрели, всегда что-то может пойти не так. Когда происходит ошибка при выполнении приложения, операционная системы выбрасывает исключение. Используя конструкцию try-catch можно ловить эти исключения. Если при выполнении программы в блоке try произошло исключение, управление передаётся блоку catch. Если вы не обрабатываете исключения, то исключение вызовет окошко операционной

системы с предложением отладить программу при помощи Just-in-Time Debugging.

Перед запуском приложения C# необходимо его откомпилировать. Компилятор преобразует исходный код в машинные коды, которые понятны компьютеру. Компилятор C# можно вызывать как из комнадной строки, так и из Visual Studio. Пример вызова компилятора из командной строки:

csc Hello.cs /debug+ /out:Greet.exe

Если компилятор находит ошибки, он сообщает об этом, строку ошибки и номер символа. Если ошибок нет и приложение откомпилировалось, можно его запускать. Или при помощи Visual Studio, или в командной строке по имени файла с расширением exe.

Вопросы к разделу.

1. Откуда начинается выполнение в приложении C#.
2. Когда приложение заканчивает работу.
3. Сколько классов может содержать приложение C#.
4. Сколько методов Main может содержать приложение.
5. Как прочитать данные введенные пользователем с клавиатуры.
6. В каком пространстве имен находится класс Console.
7. Что произойдёт при необработанном в приложении исключении.

Лабораторная работа.

Задания на создание C# программ, компилирование и отладку, использование отладчика Visual Studio, обработку исключительных ситуаций. Время, необходимое на выполнение задания 60 мин.

Упражнение 2.1 Написать программу, которая спрашивает имя пользователя, и затем приветствует пользователя по имени. (Создать консольное приложение.)

- Откомпилировать и запустить программу с помощью Visual Studio.
- Откомпилировать и запустить программу с командной строки.
- В среде Visual Studio использовать пошаговую отладку. Посмотреть, как изменяется текущее значение переменной.

Упражнение 2.2 Написать программу, которой на вход подается два целых числа, на выходе – результат деления одного числа на другое. Предусмотреть обработку исключительной ситуации, возникающей при делении числа на ноль.

Домашнее задание 2.1 Прочитать букву с экрана и вывести на печать следующую за ней букву в алфавитном порядке.

Домашнее задание 2.2 Написать программу, которая решает квадратное уравнение. Входные данные – коэффициенты уравнения, выходные – найденные корни.

Глава 3. Использование структурных переменных.

Все приложения работают с теми или иными данными. Как C# разработчикам, вам необходимо понимать как хранятся и обрабатываются данные в приложении. Обычно данные хранятся в переменных, до использования переменной её необходимо объявить. При объявлении переменной вы резервируете под неё некоторое количество памяти в зависимости от типа переменной и даёте ей имя. После объявления переменной можно присваивать ей значения.

В этом разделе рассматривается как использовать структурные переменные в C#. Как именовать переменные в соответствии со стандартами, как присваивать значения и как переводить существующие переменные из одного типа в другой.

Общая система типов.(Common Type System)

При объявлении переменной, для неё выделяется область в памяти, в зависимости от типа переменной. Только после объявления переменной ей можно присвоить значение.

Каждая переменная имеет тип данных, который определяет какие значения хранятся в ней. C# язык безопасных типов, т.е. компилятор гарантирует что значение хранящееся в переменной всегда соответствующего типа.

CTS интегрированная часть общеязыковой среды выполнения. Эта модель определяет правила, которыми руководствуется среда выполнения при объявлении, использовании и управления типами. CTS обеспечивает структуру, необходимую для многоязыковой интеграции, безопасности типов и высокой эффективности выполнения кода.

Рассмотрим два типа переменных: структурные и ссылочные.
Структурные типы данных напрямую содержат данные. Каждый переменная содержит свою копию данных, т.е. при операции над одной переменной невозможно изменить другую. Структурные типы данных включают в себя встроенные и пользовательские типы. Разница между ними в C# минимальна,

так как они используются одинаково. Все структурные типы напрямую содержат данные и не могут быть null.

Ссылочные типы данных содержат ссылки на данные. Две переменные могут указывать на один и тот же объект, т.е. при изменении одной ссылочной переменной, можно изменить другую.

Все базовые типы данных содержаться в пространстве имен System. Все типы наследуют от System.Object. Все структурные типы наследуют от System.ValueType.

Встроенные типы объявляются при помощи зарезервированных слов, кроме того можно объявить при помощи типа структуры из пространства имен System

sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Правила именования переменных.

При объявлении переменных необходимо выбирать значимые и соответствующие имена. Дадим несколько советов по именованию переменных.

При именовании переменных необходимо соблюдать следующие правила, если не соблюдать получим ошибку при компиляции:

- можно использовать только буквы, нижнее подчеркивание и цифры

- имя переменной начинается только с буквы или подчеркивания
- после первого символа можно использовать и цифры
- нельзя использовать зарезервированные слова

Рекомендации по именованию:

- избегать имена только из заглавных букв
- избегать начинать с подчеркивания
- избегать аббревиатур
- Именование PascalCasing – каждое слово начинается с большой буквы, используется для классов, методов, свойств, перечислений, интерфейсов, констант, пространств имен.
- Именование camelCasing - каждое слово кроме первого начинается с большой буквы, используется для переменных, полей и параметров.

Использование встроенных типов данных

Для использования переменной необходимо выбрать ей имя, назначить тип и присвоить начальное значение.

Переменные которые объявлены в методах называются локальными. В C# нельзя использовать неинициализированные переменные, выдаётся ошибка при компиляции. Переменной можно присвоить значение соответствующего типа.

Добавление значения переменной осуществляется очень просто

```
int itemCount;
itemCount = 2;
itemCount = itemCount + 40;
```

Сокращенный вид

```
var += expression;          // var = var + expression
var -= expression;          // var = var - expression
var *= expression;          // var = var * expression
var /= expression;          // var = var / expression
var %= expression;          // var = var % expression
```

Выражения состоят из операций и операндов.

Общие операции:

Операции присваивания – присваивают значение правого операнда левому

= *= /= %= += -= <<= >>= &= ^= |=

Операции сравнения – сравнивают два значения.

== !=

Логические – производит побитовые операции над операндами

<< >> & ^ |

Условные – выполняют одно выражение из двух в зависимости от булевого условия.

&& || ?:

Инкремент и декремент – увеличивает/уменьшает значение на единицу.

++ --

Арифметические – выполняют стандартные арифметические операции.

+ - * / %

Операции инкремент и декремент имеют два варианта префиксный и постфиксный. Соответственно, сначала выполняется увеличение значения, а затем выполняется выражение и наоборот, сначала выражение, и только после него изменяется значение операнда.

Пользовательские типы данных

В приложениях у вас есть возможность создавать свои типы данных: структуры и перечисления.

Перечисления полезны, когда переменные могут принимать значения только из определенного набора. По умолчанию нумерация элементов начинается с нуля.

enum Color { Red, Green, Blue }

Цвет Red будет иметь значение 0, Green - 1, а Blue будет 2.

Использование переменных перечислимого типа.

```
Color colorPalette;       // объявление переменной
colorPalette = Color.Red; // установка значения
```

или

```
colorPalette = (Color)0;   //явное преобразование из int
```

Структуры можно использовать для создания объектов ведущих себя как встроенные типы. Они группируют данные различного типа.

```
public struct Employee
{
        public string firstName;
        public int age;
}
```

Для доступа к элементам структур используется следующая конструкция:

```
Employee companyEmployee; //объявление переменной
companyEmployee.firstName = “Joe”; //установка значений
companyEmployee.age = 23;
```

Преобразование типов.

Различают два типа преобразования типов: явное и неявное. Неявное преобразование выполняется всегда, но можно потерять точность. Для явного преобразования используется операция приведения типов.

Неявное преобразование:

```
using System;
```

```
class Test;
{
        static void Main()
        {
                int intValue = 123;
                long longValue = intValue;
                Console.WriteLine("(long) {0} = {1}", intValue, longValue)
        }
}
```

Явное проебразование:

```
using System;
class Test;
{
        static void Main()
        {
                long longValue = Int64.MaxValue;
                int intValue = (int) longValue;
                Console.WriteLine("(int) {0} = {1}", longValue, intValue)
        }
}
```

Эта программа выдаст ошибку из-за переполнения, на экране получим (int) 9223372036854775807 = -1

Вопросы к разделу.

1. Что такое общая система типов?
2. Может ли структурная переменная иметь значение null?
3. Можно ли не инициализировать переменные в C#? Почему?
4. Можно ли потерять данные в результате неявного преобразования?

Лабораторная работа.

Задания на создание пользовательских типов данных, объявление и использование переменных. Время, необходимое на выполнение задания 35 мин.

Упражнение 3.1 Создать перечислимый тип данных отображающий виды банковского счета (текущий и сберегательный). Создать переменную типа перечисления, присвоить ей значение и вывести это значение на печать.

Упражнение 3.2 Создать структуру данных, которая хранит информацию о банковском счете – его номер, тип и баланс. Создать переменную такого типа, заполнить структуру значениями и напечатать результат.

Домашнее задание 3.1 Создать перечислимый тип ВУЗ{КГУ, КАИ, КХТИ}. Создать структуру работник с двумя полями: имя, ВУЗ. Заполнить структуру данными и распечатать.

Глава 4. Операторы и исключения.

Одним из главных умений, требуемых для использования языка программирования, является умение работать с операторами, которые формируют логику программы. В этом разделе рассказывается о том, как использовать некоторые операторы в C#, кроме этого описывается обработка исключений. В частности, показывается как выбрасывать исключения, как ловить их, и как описывать блок try-catch, чтобы ни одно исключение не вышла за рамки приложения.

Операторы в C#.

Программа состоит из последовательности операторов. При выполнении они запускаются по очереди. При разработке приложения необходимо группировать операторы, в C# они группируются в блоки при помощи фигурных скобок. Каждый блок задают свою область видимости, в которой существуют локальные переменные

```
{
        // блок операторов
}
```

Операторы используемые в программах можно сгруппировать в три группы: условные операторы, операторы цикла и операторы перехода.

Условные операторы.

Операторы if и switch – операторы выбора, они вычисляют значение условия и на основе этого выбора выполняют операторы.

Неявное преобразование из int в bool потенциальный путь к ошибкам, поэтому в C# такое действие запрещено. В условии обязательно должно быть булево выражение.

```
int x; ...
if (x) ...    // Должно быть x != 0 в C#
if (x = 0) ... // Должно быть x == 0 в C#
```

Можно использовать каскадную последовательность if, для этого используем конструкцию else if.

Оператор switch элегантный механизм для обработки сложных условий. Он состоит из нескольких блоков case, каждый из которых определяется своей константой. Для группировки констант, пишем каждую из них в собственном case, выполняющийся оператор после всей группы. Операторы выполняются до оператора break.

```
enum MonthName { January, February, ..., December }
MonthName current;
int monthDays; ...
switch (current) {
case MonthName.February :
  monthDays = 28;
  break;
case MonthName.April :
case MonthName.June :
case MonthName.September :
case MonthName.November :
  monthDays = 30;
  break;
default :
  monthDays = 31;
    break;
}
```

Если константы повторяются, получим ошибку при компиляции:

```
 switch (trumps) {
case Suit.Clubs :
case Suit.Clubs : // Ошибка: повтор метки
  ...
default :
default : // Ошибка: повтор метки
    ...
}
```

Операторы цикла.

Операторы цикла выполняют операции до тех пор пока условие верно. Операторы циклов while, do - while, for и foreach.

Цикл while – простейший оператор цикла. C# не поддерживает неявное преобразование в boolean, поэтому в условии обязательно.

Отличие цикла do – while, в том что условие проверяется после выполнения операции.

Цикл for включает в себя первоначальную инициализацию счетчика, условие выполнения и обновление счетчика. Однако в нём может отсутствовать любая часть for – инициализация, условие и обновление.

```
for (;;) {
   Console.WriteLine("Help ");
   ...
}
```

Цикл foreach цикл для выборки всех элементов коллекции. Необходимо выбрать имя и тип переменной, которая будет идти по коллекции. Это переменная в теле цикла будет доступна только для чтения.

```
foreach(int number in numbers)
        Console.WriteLine(number);
```

Операторы перехода.

Операторы goto, break и continue – операторы перехода. Они используются для передачи управления из одной части программы в другую.

Goto простейший оператор, управление передается в точку программу отмеченную меткой. Метка определяется двоеточием после идентификатора.

```
if(number % 2 == 0) goto Even;
Console.WriteLine(“odd”);
Goto End;
Even:
Console.WiteLine(“even”);
```

End:;

Break и continue используются в циклах, первый для выхода из цикла, второй для перехода к следующей итерации.

Обработка исключений.

В этом разделе изучается как ловить и обрабатывать исключения.

Традиционно обработка ошибок производилась в теле программы, это затрудняло понятие логики самого приложения, запутывало код. Сообщения о ошибках представляют собой число, которое не несло смысловой нагрузки.

В C# все исключения это объекты унаследованные от класса Exception. Названия классов раскрывают вид исключения. Объекты могут содержать дополнительную информацию, например FileNotFoundException может хранить имя файла.

Логика программы помещается в блок try, если при выполнении выскакивает исключение, то управление передается блоку catch.

Есть несколько вариантов блока catch:

- общий catch { ... } или равносильно catch (System.Exception) { ... }
- по виду исключения

catch (OutOfMemoryException caught) { ... }, где caught объект класса исключения.

Если присутствует несколько блоков catch, то они не должны повторятся и направление от более частных классов к общим, блок для System.Exception может быть только последним.

Оператор throw выбрасывает вручную исключение. Мы можем выбрасывать только объекты класса исключения, то есть для этого его надо создать.

```
if (minute < 1 || minute >= 60) {
  string fault = minute + " это неправильные минуты ";
  throw new InvalidTimeException(fault);
  // !!Не проходит!!
}
```

Блок finally выполняется обязательно вне зависимости произошло исключение или нет. Он используется в двух случаях для избегания повтора операторов и для освобождения ресурсов.

По умолчанию проверка переполнения стека арифметическими операциями в C# выключен. Для включения можно создать блок checked, или при компиляции указать опцию

csc /checked+ example.cs

отключить

csc /checked- example.cs

Блок unchecked явно выключает проверку.

Рекомендации по обработке исключений:

- Включайте строки описания

```
string description = String.Format("{0}({1}): newline in string constant",filename, linenumber);
throw new SyntaxException(description);
```

- Выбрасывать исключения более специфичного вида, например FileNotFoundException, лучше чем более общий IOException.
- Не позволять исключениям выходить из Main, то есть всегда должна существовать такая конструкция

```
static void Main( )
{
   try {
      ...
   }
   catch (Exception caught) {
      ...
   }
}
```

Вопросы к разделу.

1. Написать условный оператор, который проверяет находится ли переменная целого типа с именем hour в пределах интервала от 0 до 23, если нет обнулить переменную.
2. Описать цикл do-while, в теле которого происходит чтение целой переменной hour с клавиатуры, до тех пор пока она не окажется в интервале от 0 до 23.
3. Описать цикл for для предыдущей задачи, но оставив для ввода максимум пять попыток. Не использовать break и continue.
4. Написать предыдущую задачу используя break.
5. Написать оператор выбрасывающий исключение типа ArgumentOutOfRangeException, если переменная percent меньше 0 или больше 100.

Лабораторная работа.

Задания на операторы if, switch, for, while, foreach, обработку исключений. Время, необходимое на выполнение задания 45 мин.

Упражнение 4.1 Написать программу, которая читает с экрана число от 1 до 365 (номер дня в году), переводит этот число в месяц и день месяца. Например, число 40 соответствует 9 февраля (високосный год не учитывать).

Упражнение 4.2 Добавить к задаче из предыдущего упражнения проверку числа введенного пользователем. Если число меньше 1 или больше 365, программа должна вырабатывать исключение, и выдавать на экран сообщение.

Домашнее задание 4.1 Изменить программу из упражнений 4.1 и 4.2 так, чтобы она учитывала год (високосный или нет). Год вводится с экрана. Год високосный если выполняются одновременно два условия: 1) год делится на 4; 2) не делится на 100 и не делится на 400.

Глава 5. Методы и параметры.

При разработке большинства приложений, мы разделяем их на функциональные модули. Это центральный принцип программирования, так маленькие разделы кода легче для понимания, разработки и отладки. Кроме того это позволяет снова использовать участки кода для других приложений.

В C# структура приложения состоит из классов, которые содержат именованные блоки кода, называемые методами. Метод – это член класса, который может осуществлять действия или вычислять значения.

Использование методов.

Разделение программной логики на отдельные функциональные модули центральный принцип программирования. Так как небольшие фрагменты программы легко понятны и легко используемы. В C# программа состоит из классов, поэтому такие блоки называются методами. Метод может выполнять действие или вычислять значение.

Метод это набор операторов, которые выполнятся вместе и имеют имя. В C# все методы принадлежат какому-либо классу. Для создания метода необходимо задать его имя, определить список параметров и тело метода.

Для вызова метода используется имя метода, если у метода есть параметры, то необходимо их определить. Для вызова метода другого класса необходимо чтобы он был public, вызов осуществляется по имени класса и метода.

```
using System;
class NestExample
{
        static void Method1
        {
                Console.WriteLine(“Method1”);
        }
        static void Method2
```

```
        {
                Method1();
                Console.WriteLine(“Method2”);
                Method1();
        }
        static void Main()
        {
        Method2();
        Method1();
        }
}
```

В результате получим:

Method1

Method2

Method1

Method1

Оператор return останавливает выполнение метода и передаёт управление вызвавшему методу. Если метод не void, то необходимо вернуть значение соответствующего типа.

Каждый метод имеет набор своих локальных переменных, они видны только в нем и при завершении работы метода уничтожаются. Для того чтобы переменные были видны в нескольких методах класса, необходимо объявить их полями вне метода, но внутри класса.

Для не void методов необходимо возвращать значение, каждый путь выполнения метода должен заканчиваться оператором return. Для void методов оператор return не обязателен.

Использование параметров.

Параметры позволяют передавать информацию из одного методы в другой. При объявлении метода можно задать список его параметров, если список пустой значит метод не имеет параметров.

```
static void MethodWithParameters(int n, string y)
{
   // ...
}
```

При вызове метода необходимо задать значения его параметрам.

```
MethodWithParameters(2, "Hello, world");
```

Или

```
int p = 7;
string s = "Test message";
MethodWithParameters(p, s);
```

В C# существуют три варианта передачи параметров: по значению, по ссылке и выходные параметры. В основном информация передаются для работы метода, то есть в метод, реже обратно. При передаче параметров по значению изменение значения параметра в методе не влияет на значение в вызвавшем методе.

```
static void AddOne(int x)
{
        x++;
}
static void Main()
{
        int k = 6;
        AddOne(k);
        Console.WriteLine(k); // Выведет на экран 6, не 7
}
```

При передаче по ссылке передается ссылка на переменную, поэтому все действия. производимые с параметром, оказывают влияние на значение

переменной в вызывающем методе. Для каждого параметра по ссылке необходимо указывать ключевое слово ref. До вызова метода необходимо обязательно инициализировать переменную.

```
static void AddOne(ref int x)
{
        x++;
}
static void Main()
{
        int k = 6;
        AddOne(k);
        Console.WriteLine(k); // Выведет на экран 7
}
```

Выходные параметры похожи на параметры по ссылке, единственное исключение их можно не инициализировать до вызова метода.

```
static void OutDemo(out int p)
{
        // …
}
```

C# позволяет использовать механизм передачи списка параметров изменяемой длины. Для этого используется ключевое слово params. Правила использования, только один список, помещается в конец списка параметров, объявляет список параметров как массив конкретного типа.

```
static long AddList(params long[] v)
{
        long total;
        long i;
        for(i = 0, total = 0; i < v.Length; i++)
            total += v[i];
        return total;
}
```

При вызове можно использовать два пути: вызывать как список отдельных элементов или как массив.

```
static void Main( )
{
   long x;
   x = AddList(63, 21, 84); // Список
   x = AddList(new long[ ]{ 63, 21, 84 }); // Массив
}
```

Для выбора вида параметров необходимо учитывать два аспекта: механизм передачи и эффективность. По эффективности передача по значению лучше, чем передачи по ссылке.

Когда метод вызывает себя, то это называется рекурсией. Если это происходит при участии другого метода, то это будет неявной рекурсией. Рекурсия часто используется для упрощения логики программы.

Перегрузка методов.

Имя метода не может совпадать с именем любого другого элемента класса, но может совпадать с другим методом – это называется перегрузка метода. Перегружаемые методы – это методы с одинаковыми именами в одном классе.

```
Class OverloadingExample
{
        static int Add(int a, intb)
        {
                return a + b;
        }
        static int Add(int a, intb, int c)
        {
                return a + b + c;
        }
        static void Main()
```

```
        {
            Console.WriteLine(Add(1,2) + Add(1,2,3));
        }
}
```

Нельзя использовать одно имя для метода и переменной, константы или перечисления.

```
class BadMethodNames
{
    static int k;
    static void k( )
    {
        // ...
    }
}
```

Компилятор использует сигнатуру метода для различения методов в классе. В сигнатуру входят следующие элементы: имя метода, типы параметров, модификаторы параметров. В сигнатуру не входят имена параметров и возвращаемый тип метода.

Следующие три метода имеют различную сигнатуру

```
static int LastErrorCode( )
{
}
static int LastErrorCode(int n)
{
}
static int LastErrorCode(int n, int p)
{
}
```

Следующие три метода имеют одинаковую сигнатуру

```
static int LastErrorCode(int n)
{
}
static string LastErrorCode(int n)
```

```
{
}
static int LastErrorCode(int x)
{
}
```

Перегружаемые методы полезны, если есть одинаковые методы различающиеся только списком параметров.

```
class GreetDemo
{
   static void Greet( )
   {
    Console.WriteLine("Hello");
   }
   static void Greet(string Name)
   {
    Console.WriteLine("Hello " + Name);
   }
   static void Main( )
   {
    Greet( );
    Greet("Alex");
   }
}
```

Кроме того они полезны при добавлении новой функциональности. Допустим в предыдущем классе, мы захотим приветствовать пользователи в соответствии со временем дня.

```
class GreetDemo
{
   enum TimeOfDay { Morning, Afternoon, Evening }

   static void Greet( )
   {
```

```
        Console.WriteLine("Hello");
    }
    static void Greet(string Name)
    {
        Console.WriteLine("Hello " + Name);
    }
    static void Greet(string Name, TimeOfDay td)
    {
        string Message = "";

        switch(td)
        {
        case TimeOfDay.Morning:
            Message="Good morning";
            break;
        case TimeOfDay.Afternoon:
            Message="Good afternoon";
            break;
        case TimeOfDay.Evening:
            Message="Good evening";
            break;
        }
        Console.WriteLine(Message + " " + Name);
    }
    static void Main( )
    {
        Greet( );
        Greet("Alex");
        Greet("Sandra", TimeOfDay.Morning);
    }
}
```

Вопросы к разделу.

1. Объясните что такое методы и почему они важны?
2. Опишите три возможных пути передачи параметров и соответствующие ключевые слова C#.
3. Когда создаются и уничтожаются локальные переменные?
4. Какое ключевое слово должно быть добавлено к определению метода, чтобы его можно было вызывать из другого метода?
5. Что входит в сигнатуру метода?

Лабораторная работа.

Задания на методы с параметрами и без, различные механизмы передачи параметров. Время, необходимое на выполнение задания 60 мин.

Упражнение 5.1 Написать метод, возвращающий наибольшее из двух чисел. Входные параметры метода – два целых числа. Протестировать метод.

Упражнение 5.2 Написать метод, который меняет местами значения двух передаваемых параметров. Параметры передавать по ссылке. Протестировать метод.

Упражнение 5.3 Написать метод вычисления факториала числа, результат вычислений передавать в выходном параметре. Если метод отработал успешно, то вернуть значение true; если в процессе вычисления возникло переполнение, то вернуть значение false. Для отслеживания переполнения значения использовать блок checked.

Упражнение 5.4 Написать рекурсивный метод вычисления факториала числа.

Домашнее задание 5.1 Написать метод, который вычисляет НОД двух натуральных чисел (алгоритм Евклида). Написать метод с тем же именем, который вычисляет НОД трех натуральных чисел.

Домашнее задание 5.2 Написать рекурсивный метод, вычисляющий значение n-го числа ряда Фибоначчи. Ряд Фибоначчи – последовательность натуральных чисел 1, 1, 2, 3, 5, 8, 13… Для таких чисел верно соотношение $F_k = F_{k-1} + F_{k-2}$.

Глава 6. Массивы.

Массивы хороший инструмент группировки данных. Для работы с C# необходимо понимать как они работают. Существуют два пути группировки связанных данных структуры и массивы. Структуры группируют данные различных типов, массивы это последовательность данных одного типа. Массивы разрешают произвольный доступ к любому элементу, причём он одинаково быстр.

Синтаксис массивов в C#

```
type[ ]  name;    // правильно
type   name[ ];   // не правильно в C#
type[4]    name;   // также неправильно C#
```

Для объявления двумерных массивов, используются пустые индексы через запятую.

```
int[,] grid;
```

Для доступа к элементам массива используются индексы в квадратных скобках. Нумерация начинается с нуля. Если размерность массива больше одного, то индексы перечисляются через запятую.

```
long[] row;                    int[,] grid;
…                              …
row[3];                        grid[1,2];
```

В C# индексы массива автоматически проверяются на удовлетворение размерности, при выходе за интервал выдаётся исключение IndexOutOfRangeException. Для проверки размерности можно использовать свойство массива Length и метод GetLength.

Сравнение массивов с коллекциями. Коллекции гибче массивов, могут хранить различные элементы, имеют переменную длину. Но в связи с этим работа с коллекциями медленнее.

```
ArrayList flexible = new ArrayList( );
flexible.Add("one"); // Добавили строку...
flexible.Add(99);    // Добавили int
```

Создание коллекции только для чтения

```
ArrayList flexible = new ArrayList( ); ...
ArrayList noWrite = ArrayList.ReadOnly(flexible);
noWrite[0] = 42;    // Исключение при выполнении
```

Объявление массива не создает его, так как массив – это ссылочный тип. При объявлении массива можно не знать его размерность. При создании это необходимо. Память для массива выделяется последовательно.

```
long[] row = new long[4];
int[,] grid = new int[2,3];
```

Можно использовать список инициализации при создании массива. Необходимо объявить все элементы, в качестве элементов можно использовать выражения. Те же правила для многомерных массивов, необходимо определить все строки и в каждой строке все элементы.

```
int[,] data = new int[2,3]{
        {2,3,4},
        {5,6,7}  }
```

Размерность массива можно задавать как константами, так и вычисляемыми значениями. Единственное ограничение при вычисляемой длине нельзя использовать список инициализации.

```
string s = Console.ReadLine();
int size = int.Parse(s);
long[] row = new long[size];
```

При копировании переменной массива, не создается новый массив, создается новая ссылка на тот же массив.

```
long [] row = new long[4];
long [] copy = row;
row[0]++; // изменит copy[0]
```

Рассмотрим некоторые свойства и методы класса System.Array.

- Свойство Rank - размерность массива.
- Свойство Length - общая длина массива, для многомерных массивов количество всех ячеек.

System.Array – класс от которого неявно наследуют все массива в C#.

- Sort – сортировка массива, поддержка интерфейса IComparable
- Clear – очистка массива, все элементы устанавливаются в NULL.
- Clone – создаёт копию массива, копирует все элементы. Этот метод не следит за значениями в элементах. Если там ссылки на объекты,то он просто скопируются, новых объектов создано не будет.
- GetLength – по номеру размерности получаем длину массива в этой размерности.
- IndexOf – ищет значение в массиве, если нашел возвращает индекс первого вхождения, иначе -1.

При возвращении массива из метода его размеры не задаются, скобки остаются пустыми. Если задать, получим ошибку при компиляции. Также с многомерными массивами.

```
static int[,] CreateArray( ) {
  string s1 = System.Console.ReadLine( );
  int rows = int.Parse(s1);
  string s2 = System.Console.ReadLine( );
  int cols = int.Parse(s2);
 return new int[rows,cols];
}
```

При передаче массива в метод в качестве параметра, новый массив не создается, передается ссылка на тот же массив. Все действия с массивом в методе сохранятся для вызывающего метода. Если хотите избежать этого, необходимо передавать копию массива при помощи метода Array.Copy

При вызове приложения из командной строки можно использовать строку параметров. Например:

C:\> pkzip –add –rec –path=relative c:\code *.cs

Тогда если pkzip написан на C#, получим массив

```
string[ ] args = {
 "-add",
 "-rec",
 "-path=relative",
 "c:\\code",
 "*.cs"
};
```

Этот массив мы получаем при запуске метода Main.

```
class  PKZip
{
        static void Main(string[] args)
        {
        }
}
```

Для обработки массивов можно использовать foreach. Тогда не нужен счетчик, проверка длины массива и обращение к элементу.

Две следующие контсрукции эквивалентны

```
for (int i = 0; i < args.Length; i++) {
 System.Console.WriteLine(args[i]);
}

foreach (string arg in args) {
```

```
 System.Console.WriteLine(arg);
}
```

Также можно использовать для многомерных массивов

```
int[,] numbers = { {0,1,2}, {3,4,5} };
foreach (int number in numbers) {
 System.Console.WriteLine(number);
}
```

Вопросы к разделу.

1. Объявите одномерный массив *evens* из целых чисел, и инициализируете их первыми пятью четными числами.
2. Написать оператор, который объявляет целую переменную *crowd* и инициализирует её значением второго элемента массива *evens.*
3. Написать два оператора: один объявляет одномерный массив *copy* из целых чисел, второй присваивает *copy even.*
4. Описать статичные метод Method, который возвращает двумерный массив целых чисел и не имеет параметров. В методе присутствует только один return, возвращающий только созданный массив размерности 3 на 5 заполненный числами 42.
5. Написать статичный метод Parametr, который ничего не возвращает и получает в качестве параметра двумерный массив. Метод выводит на экран обе размерности полученного массива.
6. Написать foreach, который проходит по одномерному массиву строк names и выводит каждую строку на экран.

Лабораторная работа.

Задания на массивы, передачу массива в качестве аргумента методу Main. Время, необходимое на выполнение задания 60 мин.

Упражнение 6.1 Написать программу, которая вычисляет число гласных и согласных букв в файле. Имя файла передавать как аргумент в функцию Main. Содержимое текстового файла заносится в массив символов. Количество гласных и согласных букв определяется проходом по массиву. Предусмотреть метод, входным параметром которого является массив символов. Метод вычисляет количество гласных и согласных букв.

Упражнение 6.2 Написать программу, реализующую умножению двух матриц. В программе предусмотреть два метода: метод печати матрицы, метод умножения матриц (на вход две матрицы, возвращаемое значение – матрица).

Домашнее задание 6.1 Написать программу, вычисляющую среднюю температуру за год из набора, введенного пользователем. Значения температур за каждый месяц необходимо сохранить в одномерный массив. Написать функцию, которая по данному массиву вычисляет среднюю температуру и возвращает среднее значение.

Усложнение: Создать двумерный массив temperature[12,30], в котором будет храниться температура для каждого дня месяца (предполагается, что в каждом месяце 30 дней). Сгенерировать значения температур случайным образом. Для каждого месяца распечатать среднюю температуру. Для этого написать метод, который по массиву temperature [12,30] для каждого месяца вычисляет среднюю температуру в нем, и в качестве результата возвращает массив средних температур. Полученный массив средних температур отсортировать по возрастанию.

Глава 7. Основы объектно-ориентированного программирования.

C# объектно-ориентированный язык. В этом разделе изучаются терминология и концепции ООП.

Классы и объекты

Ключевое слово в ООП класс. Все языки программирования могут обращаться с общими данными и методами. Эта возможность помогает избежать дублирования. Главная концепция программирования не писать один и тот же код дважды. Программы без дублирования лучше и понятнее, так как кода меньше. ООП переводит эту концепцию на новый уровень, он позволяет описывать классы (множества объектов), которые делают общими и структуру, и поведение. Классы не ограничиваются описанием конкретных объектов, они также могут описывать и абстрактные вещи.

Объект – это конкретный представитель класса. Его определяет три характеристики: уникальность, поведение и состояние. *Уникальность* это характеристика определяющая отличие одного объекта от другого. *Поведение* определяет то, чем объект может быть полезен, что он может делать. Поведение объекта классифицирует его. Объекты разных классов отличаются своим поведением. *Состояние* описывает внутреннюю работу объекта то, что обеспечивает его поведение. Хорошо спроектированный объект оставляет своё состояние недоступным. Нас не интересует, как он это делает, нам важно то, что он умеет это делать.

Сравним структуры и классы. Хотя структуры и могут иметь методы, но желательно избегать этого. Однако в некоторых структурах необходимы операторы. Операторы – стилизованные методы, но они не добавляют поведение, а обеспечивают более краткий синтаксис. Например

Структурные типы – нижний уровень программы, это элементы из которых строятся более сложные элементы. Переменные структурных типов свободно копируются и используются как поля и атрибуты объектов.

Ссылочные типы – верхний уровень программы, они состоят из мелких элементов. Ссылочные типы в основном не могут быть скопированы.

Абстракция – это тактика очистки объекта от всех несущественных деталей, оставляя только существенную минимальную форму. Абстракция важный принцип программирования. Хорошо спроектированный класс содержит минимальный набор методов, полностью описывающий его поведение. Хорошая абстракция позволяет делать сложные вещи простыми.

Инкапсуляция данных

Традиционное процедурное программирование содержит много данных и много процедур. Любая функция имеет доступ к любым данным. Когда программы становятся большими, это создаёт много проблем при изменениях необходимо следить за всей программой. Другая проблема - это хранение данных отдельно от функций. ООП это движение к решению этих проблем. Первый и наиболее важный шаг от процедурного программирования к ООП это объединение данных и методов.

Данные и функции объединены в одну сущность, эта сущность ограничивает капсулу. Получаем две области снаружи этой капсулы и внутри неё. Элементы, которые доступны извне называются public, те которые доступны только внутри класса private. C# не ограничивает области видимости, любой элемент может быть как public, так и private.

Две причины использования инкапсуляции - это контроль использования и минимизация воздействий при изменениях. Вы можете использовать инкапсуляцию данных и определить поведение для того, чтобы с объектом работали по нами заданным правилам. Вторая причина вытекает из первой, если данные закрыты от внешнего использования, то их изменение не влияет на использование объекта извне.

Большинство данных внутри объектов описывают информацию об индивидуальном объекте. Данные внутри объекта обычно private и доступны только из методов класса.

Иногда необязательно хранить информацию внутри каждого объекта. То есть информация одинакова для всех объектов данного класса. Для решения этой проблемы используются статичные поля, которые принадлежат не конкретному объект, а всему классу.

Статичные методы инкапсулируют статичные данные. Статичные методы существуют на уровне класса, в них нельзя использовать оператор this. Но может обращаться к полям класса, если получит объект класса как параметр

```
class Time
{
   public static void Reset(Time t)
   {
      t.hour = 0;    // OK
      t.minute = 0;  // OK
      hour = 0;       // ошибка при компиляции
      minute = 0;      // ошибка при компиляции
   }
   private int hour, minute;
}
```

Теперь вернемся к программе Hello world, рассмотрим её со стороны объектно-ориентированного программирования. Ответим на два вопроса: как при выполнении вызывается класс и почему метод Main статичный?

Если в файле два класса с методом Main, то точка входа определяется при компиляции

```
// TwoEntries.cs
using System;
class EntranceOne
{
   public static void Main( )
   {
      Console.Write("EntranceOne.Main( )");
   }
}
```

```
class EntranceTwo
{
   public static void Main( )
   {
      Console.Write("EntranceTwo.Main( )");
   }
}
// Конец файла

c:\> csc /main:EntranceOne TwoEntries.cs
c:\> twoentries.exe
EntranceOne.Main( )
c:\> csc /main:EntranceTwo TwoEntries.cs
c:\> twoentries.exe
EntranceTwo.Main( )
c:\>
```

Если в файле нет классов с методом Main, то из него нельзя скомпилировать запускаемый файл, только библиотеку dll

```
// NoEntrance.cs
using System;
class NoEntrance
{
   public static void NotMain( )
   {
      Console.Write("NoEntrance.NotMain( )");
   }
}
// Конец файла

c:\> csc /target:library NoEntrance.cs
c:\> dir
...
NoEntrance.dll
...
```

Почему метод Main статичный? Так как для вызова нестатичного метода необходимо создать объект, а при вызове Main программа только начинает работу, и никаких объектов ещё нет.

Определение простых классов. Ключевое слово class, определить поля как в структурах, определить методы внутри класса, установить модификаторы доступа для всех полей и методов класса. Модификатор public означает, что “доступ неограничен.”, private - “доступ ограничен типом которому принадлежит.” Если пропустить модификатор, то по умолчанию поле или метод будут private.

```
class BankAccount
{
        public void Deposit(decimal amount)
        {
                balance += amount;
        }
        private decimal balance;
}
```

При объявлении объекта класса, объект не создаётся, необходимо использовать оператор new, при этом все поля инициализируются нулями. При использовании объекта без создания ошибка при компиляции.

```
Time now = new Time();
```

Ключевое слово this неявно указывает на объект вызвавший метод. Например, в следующем примере полю name нельзя присвоить значение, компилятор будет считать его параметром

```
class BankAccount
{
   public void SetName(string name)
   {
      name = name;
   }
   private string name;
```

}

Необходимо использовать this.name

В C# пять различных видов типов

 class struct interface enum delegate

Любой из них можно положить в класс. То есть в классе могут содержаться другие классы. Вложенные классы должны помечаться модификатором доступа, использование вложенных классов переводит имена из глобальной области видимости и пространства имен.

Public вложенные классы не имеют ограничений по использованию, полное имя класса можно использовать в любом месте программы. Private вложенный класс виден только из класса его содержащего. Класс без модификатора по умолчанию private.

Наследование и полиморфизм

Наследование это связи на уровне классов. Новый класс может наследовать существующий класс. Наследование это мощная связь, так как наследуемый класс наследует все элементы базового класса. От базового класса может наследовать любое количество классов. Изменение базового класса, автоматические изменяет все классы потомки.

Классы потомки могут быть одновременно и базовым классам для других классов. Группа классов связанных наследованием формирует структуру называемую иерархией классов. При движении по иерархии вверх переходим к более общим классам. При движении вниз – более специализированным классам.

Простое наследование, когда у класса есть только один прямой базовый класс.

Множественной наследование, когда у класса есть несколько прямых базовых классов. Множественное наследование создаёт предпосылки к ошибочному использованию наследования. Поэтому C#, как и большинство современных языков программирования, запрещает множественное

наследование. Напомним, что наследование особенно множественное позволяет рассматривать один объект с разных точек зрения.

Полиморфизм с литературной точки зрения означает много форм или много обликов. Концепция, что один и тот же метод определенный в базовом классе, может быть по-разному определен в разных классах потомках. Появляется новая проблема, как работать методу у объекта базового класса. Есть возможность не определять метод в базовом классе, то есть тело метода отсутствует. Такие методы называют *операциями.*

В типичной иерархии классов операции объявляются в базовом классе, а определяются различными путями в различных классах потомках. Базовый класс представляет имя метода в иерархии. В случае, когда метод не определен в базовом классе, то нельзя создавать объекты этого класса, так для этого объекта будет не определен этот метод. Такие классы называются *абстрактными.*

Абстрактные классы и интерфейсы похожи, так как не могут иметь объектов. Отличие между ними в том, что абстрактный класс может содержать определения методов, *интерфейсы* содержат только операции (имена методов). То есть интерфейсы абстрактнее абстрактных классов.

Когда вы вызываете метод напрямую из объекта, не из операции базового класса, то метод связывается с вызовом при компиляции – *раннее связывание*. При вызове метода не напрямую через объект, а через операцию базового типа, он вызывается при выполнении программы – *позднее связывание.* Гибкость позднего связывания обеспечивается физической и логической ценой. Выполняется дольше, чем раннее связывание. При позднем связывании классы потомки могут заменять базовые классы. Операции могут быть вызваны из интерфейса, классы потомки обеспечат правильное выполнение.

Вопросы к разделу.

1. Объясните концепцию абстракции, и почему она важна для программной инженерии?
2. Два принципа инкапсуляции.

3. Опишите наследование в контексте ООП.
4. Что такое полиморфизм? Как он связан с ранним и поздним связыванием?
5. Опишите разницу между интерфейсами, абстрактными классами и конкретными классами.

Лабораторная работа.

Задания на классы. Время, необходимое на выполнение задания 45 мин.

Упражнение 7.1 Создать класс счет в банке с закрытыми полями: номер счета, баланс, тип банковского счета (использовать перечислимый тип из упр. 3.1). Предусмотреть методы для доступа к данным – заполнения и чтения. Создать объект класса, заполнить его поля и вывести информацию об объекте класса на печать.

Упражнение 7.2 Изменить класс счет в банке из упражнения 7.1 таким образом, чтобы номер счета генерировался сам и был уникальным. Для этого надо создать в классе статическую переменную и метод, который увеличивает значение этого переменной.

Упражнение 7.3 Добавить в класс счет в банке два метода: снять со счета и положить на счет. Метод снять со счета проверяет, возможно ли снять запрашиваемую сумму, и в случае положительного результата изменяет баланс.

Домашнее задание 7.1 Реализовать класс для описания здания (уникальный номер здания, высота, этажность, количество квартир, подъездов). Поля сделать закрытыми, предусмотреть методы для заполнения полей и получения значений полей для печати. Добавить методы вычисления высоты этажа, количества квартир в подъезде, количества квартир на этаже и т.д. Предусмотреть возможность, чтобы уникальный номер здания генерировался программно. Для этого в классе предусмотреть статическое поле, которое бы хранило последний использованный номер здания, и предусмотреть метод, который увеличивал бы значение этого поля.

Глава 8. Использование ссылочных переменных

Ссылочные типы важная часть языка C#. Они дают возможность писать сложные и мощные приложения более эффективно.

Структурные типы – простейшие типы в C# такие как int и char, содержат информацию в переменных. Ссылочные типы в переменных не содержат информации, а только ссылки. Данные хранятся в других ячейках памяти.

Объявление, создание и очистка ссылочных переменных. Объявляются ссылочные переменные так же, как и структурные, но после объявления необходимо выделить для них память при помощи оператора new.

```
coordinate c1 = new coordinate();
c1.x = 6.12;
c1.y = 4.2;
```

В C# определено специальное значение называемое null. При помощи него можно очистить ссылочные переменные, присвоением null.

Вы можете обращаться к элементам объекта через ссылочные переменные, если только ссылка правильна, то есть указывает на объект соответствующего класса. Такие ситуации в некоторых случаях может находить компилятор, а в остальных надо ловить такие исключения и обрабатывать их.

```
coordinate c1;
c1.x = 6.12; // ошибка при компиляции, переменная не создана


try {
   c1.x = 45;
}
catch (NullReferenceException) {
   Console.WriteLine("значение c1 равно null ");
}
```

Операторы равенства и неравенства не сравнивают объекты, а только ссылки.

```
coordinate c1= new coordinate( );
coordinate c2= new coordinate( );
```

```
c1.x = 1.0;
c1.y = 2.0;
c2.x = 1.0;
c2.y = 2.0;
if (c1 == c2)
   Console.WriteLine("Одинаковые");
else
   Console.WriteLine("Разные");
```

Две ссылочные переменные могут указывать на один и тот же объект.

```
coordinate c1 = new coordinate( );
coordinate c2;
c1.x = 2.3;
c1.y = 7.6;
c2 = c1;
c1.x = 99; // Меняем значение c1
Console.WriteLine(c1.x + " , " + c1.y);
Console.WriteLine(c2.x + " , " + c2.y);
```

Получим одинаковые значения.

Ссылочные переменные можно передавать в и из методов. Также как и структурные типы, ссылочные параметры можно передавать тремя путями. При передаче по значению, передается копия ссылки, но она будет указывать на тот же объект, то есть изменение объекта в методе изменит его в вызывающем методе. Но изменение самой ссылки не изменит ссылки на объект в вызывающем методе.

```
static void PassCoordinateByValue(coordinate c)
{
   c = new coordinate( ); //c больше не ссылка на loc
   c.x = c.y  = 22.22;
}
coordinate loc = new coordinate( );
PassCoordinateByValue(loc);
```

```
Console.WriteLine(loc.x + " , " + loc.y);
```

При передаче по ссылке, ссылочная переменная помечается ref, новая переменная не создается, то есть все изменения сохраняются.

```
static void PassCoordinateByRef(ref coordinate c)
{
   c = new coordinate( );
   c.x = c.y  = 33.33;
}
coordinate loc = new coordinate( );
PassCoordinateByRef(ref loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

Передача ссылочной переменной как выходной параметр, помечается out. Ничем не отличается от ref для ссылочных переменных.

Класс object

Все классы в MS.NET Framework наследуют от класса System.Object. Класс System.Object базовый для всех ссылочных типов в C#. Когда вы объявляете класс без явного указания базового класса, неявно он наследует от object. Так как все класса наследуют от System.Object, то у всех классов есть методы этого класса. Это следующие методы:

ToString – строковое представление текущего объекта

Equals – определяет совпадает ли текущий объект с параметром.

GetType – получает информацию об объекте во время выполнения программы

Finalize – вызывается когда объект становится недоступным.

Мы можем получить информацию об объекте используя механизм называемый рефлексией. Пространство имен System.Reflection содержит классы и интерфейсы, которые обеспечивают получение типов, полей и методов классов.

Класс System.Type обеспечивает методы для получения информации об объявлении типа, такие как конструкторы, методы, поля, свойства и события.

Оператор typeof по имени типа возвращает информацию о типе.

```
using System;
using System.Reflection;
Type t = typeof(string); // Получить информацию о типе
MethodInfo[ ] mi = t.GetMethods( );
foreach (MethodInfo m in mi)  {
   Console.WriteLine("Method: {0}", m);
}
```

Пространство имен System.IO содержит классы для работы с вводом и выводом.

```
using System;
using System.IO; // Use IO namespace
// ...
StreamReader reader = new StreamReader("infile.txt");
  //Текст читается из файла
StreamWriter writer = new StreamWriter("outfile.txt");
  // Текст выводится в файл
string line;
while ((line = reader.ReadLine( )) != null)
{
   writer.WriteLine(line);
}

reader.Close( );
writer.Close( );
```

Преобразование ссылочных типов

C# поддерживает как явное, так и неявное преобразование типов. Неявное преобразование когда переменной одного типа вы присваиваете значение другого типа. При неявном преобразовании присвоение произойдёт всегда, только можно потерять информацию. При явном преобразовании

используется оператор cast. Если при выполнении преобразования возникают проблемы, выдается исключение.

```
try {
   a = checked((int) b);
}
catch (Exception) {
   Console.WriteLine("Проблема в преобразованиит");
}
```

Все преобразования в C# осуществляются при помощи класса System.Convert

Возможно преобразовывать объекты по иерархии классов, в объект класса потомка или в объект базового класса.

Преобразование к родительскому типу.

```
Animal a;
Bird b = new Bird(...);
a = b;
```

также можно преобразовать явно

```
a = (Animal) b;
```

Преобразование к типу потомка

```
Bird b = (Bird) a; // ОК
 b = a; // Ошибка при компиляции

try {
   b = (Bird) a;
}
catch (InvalidCastException) {
   Console.WriteLine("Это не птица");
}
```

Кроме обработки исключений есть другие механизмы работы с преобразованиями типов. Например оператор is.

```
Bird b;
if (a is Bird)
```

```
        b = (Bird) a; // безопасно
    else
        Console.WriteLine(“Это не птица”);
```

Кроме того для преобразований есть ещё оператор as.

```
Bird b = a as Bird; // преобразование
if( b == null)
        Console.WriteLine(“Это не птица”);
```

Все классы наследуют от object, то есть любая ссылочная переменная может быть преобразована в object.

```
try {
   b = (Bird) ox;
}
catch (InvalidCastException) {
   Console.WriteLine("Нельзя преобразовать в Bird");
}
b = ox as Bird;
if (b == null)
   Console.WriteLine("Нельзя преобразовать в Bird");
```

Оператор is всегда будет возвращать true

```
if (a is object)
```

Мы можем выполнять преобразования типов, работая с интерфейсами. Преобразование ссылки к интерфейсу

```
IHashCodeProvider hcp;
hcp = (IHashCodeProvider) x;
```

При помощи оператора is можно узнать определяет ли класс интерфейс

```
if (x is IHashCodeProvider) ...
```

Также можно использовать оператор as, вместо оператора преобразования

```
IHashCodeProvider hcp;
hcp = x as IHashCodeProvider;
```

Пример

```
interface IVisual
{
   void Paint( );
}

Rectangle r = new Rectangle( );
r.Move( );     // ОК
r.Paint( );    // ОК
IVisual v = (IVisual) r;
v.Move( );     // ошибка
v.Paint( );    // ОК
```

C# может переводить структурные типы в ссылочные и наоборот.

```
static void Show(object o)
{
   Console.WriteLine(o.ToString( ));
}
Show(42);
```

Или явное преобразование

```
object o = (object) 42; // Box
Console.WriteLine(o.ToString( ));
```

Вопросы к разделу.

1. Объясните как распределяется память для переменных ссылочного типа?
2. Каким значение присваивают ссылочной переменной. Чтобы показать, что она не указывает на объект? Что произойдет, если обратиться к ней как к объекту?
3. Базовый класс для всех классов C#.

4. Объясните разницу между операцией преобразования типа и оператором *as*.

Лабораторная работа.

Задания на использование переменных ссылочного типа, передачу их в качестве параметров методам, преобразование типов данных. Время, необходимое на выполнение задания 75 мин.

Упражнение 8.1 В класс банковский счет, созданный в упражнениях 7.1-7.3 добавить метод, который переводит деньги с одного счета на другой. У метода два параметра: ссылка на объект класса банковский счет откуда снимаются деньги, второй параметр – сумма.

Упражнение 8.2 Реализовать метод, который в качестве входного параметра принимает строку string, возвращает строку типа string, буквы в которой идут в обратном порядке. Протестировать метод.

Упражнение 8.3 Написать программу, которая спрашивает у пользователя имя файла. Если такого файла не существует, то программа выдает пользователю сообщение и заканчивает работу, иначе в выходной файл записывается содержимое исходного файла, но заглавными буквами.

Упражнение 8.4 Реализовать метод, который проверяет реализует ли входной параметр метода интерфейс System.IFormattable. Использовать оператор is и as. (Интерфейс IFormattable обеспечивает функциональные возможности форматирования значения объекта в строковое представление.)

Домашнее задание 8.1 Работа со строками. Дан текстовый файл, содержащий ФИО и e-mail адрес. Разделителем между ФИО и адресом электронной почты является символ #:

Иванов Иван Иванович # iviviv@mail.ru

Петров Петр Петрович # petr@mail.ru

Сформировать новый файл, содержащий список адресов электронной почты.

Предусмотреть метод, выделяющий из строки адрес почты. Методу в

качестве параметра передается символьная строка s, e-mail возвращается в той же строке s:

public void SearchMail (ref string s).

Домашнее задание 8.2 Список песен. В методе Main создать список из четырех песен. В цикле вывести информацию о каждой песне. Сравнить между собой первую и вторую песню в списке.

Песня представляет собой класс с методами для заполнения каждого из полей, методом вывода данных о песне на печать, методом, который сравнивает между собой два объекта:

```
class Song{
        string name; //название песни
        string author; //автор песни
        Song prev; //связь с предыдущей песней в списке

        //метод для заполнения поля name
        //метод для заполнения поля author
        //метод для заполнения поля prev
        //метод для печати названия песни и ее исполнителя
        public string Title(){… /*возвращ название+исполнитель*/ …}
//метод, который сравнивает между собой два объекта-песни:
public bool override Equals(object d){…}
}
```

Глава 9. Создание и удаление объектов

В этом разделе мы изучим, что происходит при создании объекта, как конструкторы инициализируют объекты и как используются деструкторы для уничтожения объектов. Узнаем, что происходит с объектом после уничтожения и как работает сборщик мусора.

Использование конструкторов.

Конструкторы это специальные методы, которые используются для инициализации объектов при создании. Если конструктор не описан в классе, то запускается конструктор по умолчанию.

Процесс создания объекта проходит в два этапа, но записывается всё в одно выражение. И хотя это два разных действия, в программе C# их нельзя разделить, так это обеспечивает, что память будет заполнена соответствующими значениями. Первый шаг – это выделение памяти, за это отвечает оператор new. Второй шаг – инициализация объекта при помощи конструктора

При создании объекта создаёт конструктор по умолчанию, если вы не создали свой. Иногда конструктор по умолчанию не подходит для инициализации объекта, тогда можно определить свой конструктор. Есть несколько причин, когда не подходит конструктор по умолчанию: не подходит доступ public, инициализация нулями неправильна, невидимый код тяжело понимать.

Все поля которые не определены в конструкторе инициализируются нулями. Если конструктор отработал, то с объектом можно работать, если произошла ошибка в конструкторе, то объект не создался.

Конструкторы как и любые другие методы можно перегрузить.

```
class Overload
{
```

```
    public Overload( ) { this.data = -1; }
    public Overload(int x) { this.data = x; }
    private int data;
}
class Use
{
    static void Main( )
    {
        Overload o1 = new Overload( );
        Overload o2 = new Overload(42);
    }
}
```

Если вы определили свой конструктор, то компилятор не создаст конструктор по умолчанию, его придется прописать самостоятельно.

При определении конструкторов можно использовать конструкцию, называемую списком инициализации. Определение одного конструктора при помощи вызова другого перегруженного конструктора.

```
class Date
{
        public Date() : this(1970, 1, 1){}
        public Date(int year, int month, int day){}
}
```

Ограничения – нельзя вызывать конструктор со списком инициализации из другого метода

```
class Point
{
```

```
   public Point(int x, int y) { ... }
   public void Init( ) : this(0, 0) { }   // Ошибка при компиляции
}
```

нельзя вызывать самого себя

```
class Point
{              // Ошибка при компиляции
   public Point(int x, int y) : this(x, y) { }
}
```

нельзя использовать ключевое слово this в списке инициализации.

```
class Point
{    // Ошибка при компиляции
   public Point( ) : this(X(this), Y(this)) { }
   public Point(int x, int y) { ... }
   private static int X(Point p) { ... }
   private static int Y(Point p) { ... }
}
```

При определении конструкторов необходимо определить константы и поля только для чтения. Поля не могут быть переприсвоены, поэтому они называются полями только для чтения. Есть три варианта инициализации полей только для чтения:

- Нулём неявно
- Присвоением в конструкторе, что разрешено
- Присвоением при объявлении поля в классе

```
class SourceFile
{
   public SourceFile( )   { }
   private readonly ArrayList lines = new ArrayList( );
```

```
}
```

Синтаксис конструкторов одинаков и для структур. Отличие в том, что для структур обязательно необходимо инициализировать все поля.

В некоторых приложениях полезны private конструкторы, например для процедурных функций.

```
class Math
{
   public static double Cos(double x) { ... }
   public static double Sin(double x) { ... }
   private Math( ) { ... }
}
class LessCumbersome
{
   static void Main( )
   {
    double answer = Math.Cos(42.0);
    }
}
```

Достоинства статичных методов простота и быстрота, не надо создавать объект. Статичный конструктор вызывается при загрузке класса в память.

```
class Example
{
   static Example( ) { ... }
}
```

Уничтожение объектов.

В вашем приложении вам необходимо знать, что случается когда объект выходит из области видимости или уничтожается. Процесс уничтожения объекта в C# состоит из двух шагов: деинициализация объекта и возвращение памяти в управляемую кучу.

Различия во времени жизни переменных структурных типов и объектов. У переменных структурного типа обычно короткое время жизни, ограниченное блоком в котором они объявлены. Также их отличает детерминированное создание и уничтожение. Для объектов время жизни дольше и время уничтожение недетерминированное.

В C# нельзя вручную удалить объект, так как эта возможность вызывала много различных ошибок в других языках таких как: забывание удаления объектов, попытка уничтожения объекта дважды, удаление активного объекта.

Сборщик мусора удаляет объекты за вас. Он гарантирует, что объекты будут удалены, будут удалены только один раз, будут удаляться только недостижимые объекты.

```
class Example
{
    void Method(int limit)
    {
        for(int i = 0; i < limit; i++){
            Example eg = new Example();
            ….
        }
        // eg за пределами блока, существует ли он ещё?
    }
}
```

Вы уже знаете, что удаление объекта двухшаговый процесс. На первом шаге память очищается, на втором возвращается в кучу. Второй шаг одинаков для всех классов, а вот первый индивидуален для каждого класса. Для описания первого шага определяется деструктор или метод Finalize. Вы можете описать деструктор для определения очистки объекта. В C# нельзя вручную вызвать деструктор или метод Finalize.

В C# мы не знаем когда будет уничтожен объект, только знаем что это произойдёт после того, как он станет недостижимым. Порядок и время вызова деструкторов неопределенны. Желательно избегать деструкторов, так как он использует ресурсы сборщика мусора. Если в классе нет неуправляемых частей, то сборщик мусора сам уничтожит объекты и деструктор не нужен.

Память от удаленного объекта освободится когда сборщик мусора уничтожит объект, однако кроме памяти объект может содержать другие ресурсы, которые лучше освободить побыстрее, подключение к БД, открытые файловые потоки. Для этого используется метод Dispose, для его использования необходимо, чтобы класс определял интерфейс IDisposable, и описать метод Dispose, убедится что метод не вызывается дважды.

Оператор using определяет область видимости объект, в конце этого блока для объекта вызывается метод Dispose. Семантически эквивалентно

```
Resource r1 = new Resource( );
try {
 r1.Test( );
}
finally {
  if (r1 != null) ((IDisposable)r1).Dispose( );
}
```

Вопросы к разделу.

1. Объявите класс Date с public конструктором получающим три целых параметра *year, month* и *day*.
2. Для класса Date из предыдущего вопроса сгенерируется ли конструктор по умолчанию? А если бы Date был структурой?
3. Какой метод вызывает сборщик мусора даже, если память ещё не переполнена?
4. В чем смысл использования оператора *using*?

Лабораторная работа.

Задания на создание конструкторов, деструкторов, обращение к сборщику мусора. Время, необходимое на выполнение задания 75 мин.

Упражнение 9.1 В классе банковский счет, созданном в предыдущих упражнениях, удалить методы заполнения полей. Вместо этих методов создать конструкторы. Переопределить конструктор по умолчанию, создать конструктор для заполнения поля баланс, конструктор для заполнения поля тип банковского счета, конструктор для заполнения баланса и типа банковского счета. Каждый конструктор должен вызывать метод, генерирующий номер счета.

Упражнение 9.2 Создать новый класс BankTransaction, который будет хранить информацию о всех банковских операциях. При изменении баланса счета создается новый объект класса BankTransaction, который содержит текущую дату и время, добавленную или снятую со счета сумму. Поля класса должны быть только для чтения (readonly). Конструктору класса передается один параметр – сумма.

В классе банковский счет добавить закрытое поле типа System.Collections.Queue, которое будет хранить объекты класса BankTransaction для данного банковского счета; изменить методы снятия со

счета и добавления на счет так, чтобы в них создавался объект класса BankTransaction и каждый объект добавлялся в переменную типа System.Collections.Queue.

Упражнение 9.3 В классе банковский счет создать метод Dispose, который данные о проводках из очереди запишет в файл. Не забудьте внутри метода Dispose вызвать метод GC.SuppressFinalize, который сообщает системе, что она не должна вызывать метод завершения для указанного объекта.

Домашнее задание 9.1 В класс Song (из домашнего задания 8.2) добавить следующие конструкторы:

1) параметры конструктора – название и автор песни, указатель на предыдущую песню инициализировать null.

2) параметры конструктора – название, автор песни, предыдущая песня.

В методе Main создать объект mySong. Возникнет ли ошибка при инициализации объекта mySong следующим образом: Song mySong = new Song(); ?

Исправьте ошибку, создав необходимый конструктор.

Глава 10. Наследование в C#

Этот разделе мы представим как работает механизм наследования в C#. Расскажем о типах методов virtual, override, new, опишем ненаследуемые классы, интерфейсы и абстрактные классы.

Наследование - это свойство объектно-ориентированной системы наследовать данные и функциональность базового класса. Кроме того класс потомок может замещать методы родительского класса. Вы можете наследовать от класса, только если он спроектирован для наследования. Если новый класс наследует от класса, непредназначенного для наследования, то в дальнейшем при изменении базового класса, ваш класс может оказаться нерабочим.

Наследование от класса называется *расширением* базового класса. Класс потомок наследует все элементы базового класса, кроме конструктора и деструктора. Все public элементы базового класса остаются неявно public в потомке. Private элементы, хоть и наследуются, но доступны только для объектов базового класса. Класс потомок не может быть более доступным, чем базовый класс.

```
class Example
{
   private class NestedBase { }
   public class NestedDerived: NestedBase { } // Ошибка
}
```

Доступ *protected* ограничивает доступ потомками класса. Класс имеет доступ ко всем protected полям всех своих родительских классов. При передачи объекта в качестве параметра доступ будет запрещен

```
class Token
{
        protected string name;
```

```
}
class CommentToken:Token
{
        public string Name()
        {
                return name;   // Доступ разрешен
        }

}

class CommentToken: Token
{
   void Fails(Token t)
   {
      Console.WriteLine(t.name); // Ошибка при компиляции
   }
}
```

Вызов конструктора базового класса

```
C(...): base( ) {...}
```

Ключевое слово base обращается к базовому классу. Для конструктора по умолчанию вызов базового конструктора производится по умолчанию, то есть его можно не прописывать. Для конструкторов не по умолчанию необходимо явно вызывать конструктор базового класса.

```
class Token
{
```

```
    protected Token(string name) { ... }
}
class CommentToken: Token
{
    public CommentToken(string name) { ... }
}
```

Получим ошибку при компиляции, так как будет вызываться конструктор по умолчанию, но он отсутствует в классе Token. Для правильной работы необходимо записать конструктор в следующем виде:

```
    public CommentToken(string name) : base(name) { ... }
```

Если конструктор базового класса private, то мы не можем создать конструктор класса потомка.

```
class NonDerivable
{
        private NonDerivable(){  }
}
class Impossible: NonDerivable
{
        public Impossible() {   }   // Ошибка при компиляции
}
```

Вы можете переопределять методы базового класса, только если они спроектированы для этого. Методы *virtual* можно полиморфно переопределять в классах потомках. Не виртуальный метод имеет только одно определение, одинаковое для всех потомков. Когда вы объявляете виртуальный метод, он обязательно должен содержать тело метода. Нельзя объявлять виртуальными статичные и private методы. Статичные методы не могут быть виртуальными, так как полиморфизм – это свойство относящееся к объектам, не к классам.

Методы *override* переопределяют родительский метод. Определяем

виртуальный метод в базовом классе, и в классах-потомках мы можем переопределить этот метод. Переопределенный метод, также как и виртуальный должен содержать тело метода.

```
class Token
{
        public virtual string Name() {…}
}
class CommentToken : Token
{
        public override string Name() {…}
}
```

Можно переопределять только абсолютно идентичные методы, переопределяемый метод должен быть virtual или override. Нельзя объявлять переопределенный метод виртуальным, статичным или private.

Можно спрятать наследуемый метод в иерархии классов, заменив его новым идентичным методом при помощи new. Метод родительского класса не будет наследоваться потомком, и заменится на новый идентичный.

```
class Token
{
        public int LineNumber() {…}
}
class CommentToken : Token
{
        new public int LineNumber() {…}
}
```

Ключевое слово new прячет как виртуальные, так и невиртуальные методы, исправляет конфликт имен, прячет методы с одинаковой сигнатурой.

Создание гибкой иерархии классов достаточно непростая задача. Однако, многие классы спроектированы работать в одиночку и не предназначены для наследования. Но любой программист может создать класс потомок от этого класса, и .Net Framework вынужден поверять, нет ли у данного класса потомков, то есть тратить на это ресурсы. Для решения этой проблемы можно объявить класс ненаследуемым, при помощи ключевого слова *sealed*.

```
namespace System
{
        public sealed class String
        {
        }
}
```

Использование интерфейсов.

Интерфейс – синтаксический и семантический контракт обязательный для всех классов потомков. Интерфейс говорит, что он умеет делать, классы определяют, как они это делают. Интерфейс представляет собой класс без какого-либо кода. Все интерфейсы по умолчанию public, однако, модификатор доступа не используется.

```
interface IToken
{
   public int LineNumber( ); // Ошибка при компиляции
}
```

C# позволяет наследовать от одного класса и множества интерфейсов. Интерфейс может наследовать от многих интерфейсов

```
interface IToken { ... }
interface IVisitable { ... }
interface IVisitableToken: IVisitable, IToken { ... }
```

```
class Token: IVisitableToken { ... }
```

Класс может быть более доступным, чем интерфейс

```
class Example
{
   private interface INested { }
   public class Nested: INested { } // Разрешено
}
```

Класс должен определить все методы всех интерфейсов определяемых как напрямую, так и косвенно. Метод интерфейса, определяемый классом должен быть идентичен, то есть должны совпадать параметр доступа, имя, возвращаемое значение и список параметров.

При определения метода интерфейса *явно,* необходимо указать полное имя метода, метод не может быть виртуальным, отсутствует модификатор доступа. При вызове нет прямого доступа, только через интерфейс.

```
class Token: IToken, IVisitable
{
   string IToken.Name( )
   {
      ...
   }
   private void Example( )
   {
      Name( );              // Ошибка при компиляции

      ((IToken)this).Name( ); // Правильно
   }
   ...
}
```

Для чего может понадобиться явное определение методов интерфейса.

- Позволяет исключить определение интерфейса из класса, если он не интересен пользователям класса.

- Позволяет классу обеспечивать несколько определений различных методов интерфейсов одинаковой сигнатуры.

```
interface IArtist
{
        void Draw();
}
interface ICowboy
{
        void Draw();
}
class ArtisticCowboy: IArtist, ICowboy
{
        void IArtist.Draw() {…}
        void ICowbowt.Draw() {…}
}
```

Использование абстрактных классов.

Абстрактные классы осуществляют частичное определение класса, от них наследуют конкретные классы. Правила создания абстрактного класса совпадают с обычным классом, отличия только в том, что можно объявлять абстрактные методы и нельзя создавать объекты абстрактного класса. Абстрактный класс может расширять неабстрактный. Все методы интерфейса, определяемого абстрактным классом, должны быть определены.

И абстрактные классы, и интерфейсы предназначены для наследования. Однако класс может наследовать только от одного абстрактного класса, поэтому вы должны быть более настороже при наследовании от него, чем при наследовании от интерфейса. Проверьте, что абстрактный класс удовлетворяет связи “is a” с вашим классом.

Только абстрактные классы могут иметь абстрактные методы. У абстрактного метода отсутствует тело метода. При определении абстрактных методов помните: Абстрактные методы – виртуальные, переопределенные абстрактные методы в дальнейшем будут override.

Абстрактные методы могут переопределять virtual и override методы

```
class Token
{
   public virtual string Name( ) { ... }
}
abstract class Force: Token
{
   public abstract override string Name( );
}
```

Вопросы к разделу.

1. Создать класс *Widget* в котором объявлено два метода. Оба метода ничего не возвращают и не имеют параметров. Первый метод *First* виртуальный, второй *Second* не виртуальный. Создать класс расширяющий наш *FancyWidget*, переопределить *First* и спрятать *Second*.
2. Создать интерфейс *IWidget*, который объявляет два метода. Оба метода ничего не возвращают и не имеют параметров. Первый метод назвать *First,* другой *Second*. Создать класс *Widget* определяющий интерфейс *IWidget,* определить метод *First* как виртуальный, и определить явно *Second*.
3. Создать абстрактный класс *Widget*, который объявляет protected абстрактный метод *First*, который ничего не возвращает и не имеет параметров. Создать класс, расширяющий наш абстрактный *FancyWidget*, определить *First*.

Лабораторная работа.

Задания на наследование, определение и использование интерфейсов, абстрактных классов, виртуальные методы. Время, необходимое на выполнение задания 60 мин.

Упражнение 10.1 . Создать интерфейс ICipher, который определяет методы поддержки шифрования строк. В интерфейсе объявляются два метода encode() и decode(), которые используются для шифрования и дешифрования строк, соответственно.

Создать класс ACipher, реализующий интерфейс ICipher. Класс шифрует строку посредством сдвига каждого символа на одну «алфавитную» позицию выше. Например, в результате такого сдвига буква А становится буквой Б.

Создать класс BCipher, реализующий интерфейс ICipher. Класс шифрует строку, выполняя замену каждой буквы, стоящей в алфавите на i-й позиции, на букву того же регистра, расположенную в алфавите на i-й позиции с конца алфавита. Например, буква В заменяется на букву Э.

Написать программу, демонстрирующую функционирование классов.

Домашнее задание 10.1 Создать класс Figure для работы с геометрическими фигурами. В качестве полей класса задаются цвет фигуры, состояние «видимое/невидимое». Реализовать операции: передвижение геометрической фигуры по горизонтали, по вертикали, изменение цвета, опрос состояния (видимый/невидимый). Метод вывода на экран должен выводить состояние всех полей объекта.

Создать класс Point (точка) как потомок геометрической фигуры. Создать класс Circle (окружность) как потомок точки. В класс Circle добавить метод, который вычисляет площадь окружности. Создать класс Rectangle (прямоугольник) как потомок точки, реализовать метод вычисления площади прямоугольника.

Точка, окружность, прямоугольник должны поддерживать методы передвижения по горизонтали и вертикали, изменения цвета.

Подумать, какие методы можно объявить в интерфейсе, нужно ли объявлять абстрактный класс, какие методы и поля будут в абстрактном классе, какие методы будут виртуальными, какие перегруженными.

Глава 11. Аггрегации и пространства имен

В этом разделе мы переходим от программирования на низком уровне небольших классов к более высокому уровню. Создание классов важная часть ООП, но в проектах необходимо работать со структурами большими, чем индивидуальные классы.

Использование internal классов, методов и данных.

Модификаторы доступа определяют возможность доступа к элементам класса, таким как методы и свойства:

- Public элементы доступны отовсюду из области видимости
- Protected – доступны изнутри класса и из всех потомков, извне доступа нет.
- Private – доступны только внутри класса
- Internal – доступны отовсюду из одной сборки Microsoft .NET, для других сборок доступ запрещён.
- protected internal – доступ классам потомкам и всем классам из одной сборки.

Создание хорошо-спроектированных объектно-ориентированных программ нелегкая задача, создание больших хорошо-спроектированных объектно-ориентированных программ и вовсе тяжелая работа. Полезный совет, создавать один объект для выполнения одного действия, позволяет создавать маленькие классы, легко понятные и легко используемые. Однако тогда получаем другую проблему огромное количество классов. Системы сложны, если они сложны для понимания. Один большой класс сложней для понимания, чем несколько маленьких, поэтому разбиение одного большого класса на несколько маленьких облегчает программирование.

Сила ООП в создании связей между объектами. Объекты создаются из других объектов, получаются большие сущности. Однако возникают потенциальные проблемы с модификаторами доступа private, public и protected.

Public – слишком открытый доступ, public – наоборот слишком закрытый, protected – открыт только для потомков. Для создания промежуточного типа доступа был создан внутренний (internal) доступ. Типы доступа public и private логические, то есть не зависят от физического размещения класса. Для internal классов или элементов размещение определяет область доступа. Если класс находится в исполняемом файле, то он доступен для всех классов файла. Если класс относится к конкретной сборке, то доступ разрешен только из данной сборки. В исполняемом файле может быть несколько сборок, тогда доступ к нашему классу из другой сборки будет запрещен. То есть для каждой сборки одного исполняемого файла свой внутренний доступ.

Внутренний доступ можно сравнить с механизмом дружественности в C++. Если класс A дружествен B, то методы имеют доступ к private полям A. Однако у дружественных классов есть следующие ограничения:

- закрытость – если X хочет получить доступ к элементам Y, то не может объявить сам, только если изменить класс Y и прописать в нем дружественность к X
- отсутствие рефлексивности – если X дружественен Y, это не означает автоматически, что Y дружественен X.

Внутренний доступ открыт – при компиляции класса в сборку (или исполняемый класс) он получает доступ ко всем internal классам и элементам.

Когда класс определен как internal, доступ к нему имеют все классы из сборки. Классы и элементы protected internal доступны всей сборке, а также всем потомкам текущего класса. Вне классов и пространств имен нельзя объявлять классы protected и private, эти типы доступа можно использовать только внутри какого-либо класса. При объявлении типа в глобальной области видимости или в пространстве имен по умолчанию он принимает доступ internal. Внутри какого-либо класса – по умолчанию будет private.

Использование агрегаций.

Далее мы рассмотрим, как использовать агрегации для группировки объектов вместе в иерархию объектов. Агрегации определяют связь целое/часть между объектами, не классами. Сложные объекты состоят из объектов поменьше. В свою очередь маленькие объекты являются частью больших. Например, объект машина состоит из следующих объектов шасси, мотор и четыре колеса. Агрегации и механизм наследования оба обеспечивают создание новых больших классов из маленьких, но делают это разными путями. Главное различие, что агрегации работают с объектами, наследование с классами. В агрегациях связь между целым и частью слабая, то есть при изменении метода класса части методы класса целого не обязательно изменять. При наследовании, изменение родительского класса меняется потомок. Динамическая гибкость агрегации, то есть в объект целого можно добавлять и убирать объекты части без каких-либо ограничений. В главном объекте сохраняются ссылки на части, при необходимости они могут изменятся, можно создавать новые, удалять старые. В механизме наследованию всё статично, как объявили, что класс потомок родительского, так далее он и будет таковым.

Часто программисты только пришедшие в ООП спрашивают, как создавать виртуальные конструкторы. Ответ – никак, конструктор базового класса не наследуется, поэтому и не может быть виртуальным. Вместо того, чтобы пытаться создать виртуальный конструктор, вы можете использовать ручную делегацию через объектную иерархию:

```
class Product
{
   public void Use( ) { ... }
   ...
   internal Product( ) { ... }
}

class Factory
{
   public Product CreateProduct( )
   {
```

```
        return new Product( );
    }
    ...
}
```

В этом примере метод CreateProduct является так называемым методом фабрикой. Метод класса Factory, который создаёт объект класса Product. Так же можно создавать методы уничтожающие объекты.

```
class Factory
{
    public Product CreateProduct( ) { ... }
    public void DestroyProduct(Product toDestroy) { ... }
    ...
}
```

Пространства имен

Теперь снова вернемся к областям видимости.

```
public class Bank
{
        public class Account
        {
                public void Deposit(decimal amount)
                {
                        balance += amount;
                }
                private decimal balance;
        }
        public Account OpenAccount( ) { ... }
}
```

В этом примере мы видим четыре области видимости:

- глобальная – в ней располагается класс Bank
- область класса Bank – в ней класс Account и метод OpenAccount
- область класса Account – здесь метод Deposit и поле balance

- последняя область - тело метода Deposit.

Когда объект или метод находится не в той же области видимости, тогда вызвать его можно только при помощи полного пути. Обычно это необходимо при завершении области видимости, но иногда имя просто скрыто.

```
class Top
{
        public void M(){…}
}
class Bottom
{
        new public void M()
        {
                M();            // рекурсия
                base.M();       //обращение к скрытому методу
        }
}
```

Тоже самое и для полей

```
public struct Point
{
        public Point(int x, int y)
        {
                this.x = x;
                this.y = y;
        }
        private int x,y;
}
```

Теперь рассмотрим проекты, в которых используются огромное количество классов. Весьма возможен случай, что в одной области видимости окажутся два класса с одинаковыми именами. Решение – использовать пространства имен (Namespaces). Пространства имен разделяют большой проект на несколько подсистем, над каждым из которых может работать своя группа разработчиков.

Пространства имен – открытая область видимости, то есть объявив в одном файле, его также можно дополнить в другом файле. Таким образом, получаем два важных свойства пространств имен:

- несколько исходных файлов
- расширяемость – новый класс можно добавить без каких-либо ограничений

Пространства имен могут быть вложенными друг в друга

```
namespace Outer
{
    namespace Inner
    {
        class Widget{…}
    }
}
```

В C# можно записать предыдущий класс короче:

```
namespace Outer.Inner
{
    class Widget{…}
}
```

При использовании класса внутри пространства имен можно вызывать его при помощи его имени, однако если находимся в другом пространстве имен необходимо использовать полное имя класса. В него входят все пространства имен, в которых находится класс, перечисленные через точку, и имя класса. Для того чтобы не писать большое количество раз полное имя класса можно использовать директиву using, и снова обращаться к классу по его короткому имени.

```
namespace VendorA.VendoreB
{
    public class Widget {…}
}
// новый файл
using VendoreA.SuiteB;
```

```
class Application
{
        static void Main()
        {
                Widget w = new Widget();
        }
}
```

Использовать директиву using можно только вначале какого-либо пространства имен до объявления элементов. При попытке использования using в глобальной области видимости получим ошибку при компиляции. Директива using разрешает доступ по имени к классам только данного пространства имен, то есть доступ к классам вложенных пространств имен в данный по короткому имени закрыт. Кроме того, если мы подключим при помощи директивы два пространства имен с несколькими классами с одним именем, то при вызове этого класса получим ошибку компиляции.

```
namespace Test
{
        Using VendoreA;   // содержит класс Widget
        Using VendoreB;   // также содержит класс Widget
        // Здесь ошибки нет

        class Application
        {
                static void Main()
                {
                        Widget w = new Widget(); // ошибка при компиляции
                }
        }
}
```

Ошибка возникает, так как компилятор не знает, какой класс нам нужен. Надо явно прописать необходимое пространство имен, например

```
VendoreA.Widget w = VendoreA.Widget();
```

Кроме того директиву using можно использовать для объявления коротких имён классов напрямую.

using Widget = VendoreA.SuiteB.Widget

И тогда можно сразу обращаться по короткому имени к данному классу. Все ограничения на использование директивы остаются. Директиву using можно использовать в обоих её возможностях одновременно, единственное что надо помнить, что действие директивы начинается с первого элемента пространства имен в которой она объявлена. То есть друг на друга директивы не действуют, и следующая конструкция даст ошибку:

```
using System;
using TheConsole = Console; //ошибка, хотя Console класс из System
```

Правильно будет:

```
using System;
using TheConsole = System.Console;
```

Модули и сборки

В C# присутствует возможность компилировать исходные файлы .cs в управляемые модули. Для этого необходимо вызвать компилятор в командной строке:

```
csc /target:module Bank.cs
```

Управляемый модуль – это откомпилированный в MSIL исходный файл в котором хранятся метаданные достаточные для своего описания. При выполнении программы управляемый модуль преобразовывается в машинный код. При компиляции исходный файл компилируется в управляемый модуль с расширением .netmodule.

Выполняемый файл может запускать модули только в составе какой-либо сборки. В сборке физически размещается группа взаимодействующих классов. Классы, находящиеся в одной сборке, имеют доступ к internal

элементам друг друга. Для классов из других сборок доступ к этим элементам закрыт.

Сборка – это многократно-используемый, безопасный, и самоописываемый модуль содержащий типы и ресурсы с поддержкой версий; это - первичный стандартный блок приложения .NET. Сборка состоит из двух логических частей:

- наборов типов и ресурсов, которые формируют некоторый логический модуль из функциональных возможностей,
- метаданные, которые описывают, как эти элементы связаны и от чего зависят их правильная работа.

Метаданные, который описывают сборку, называют манифестом. Следующая информация зафиксирована в манифесте сборки:

- *Уникальность.* Уникальность сборки включает в себя простое текстовое название, номер версии и дополнительный открытый ключ, который гарантирует уникальность названия и защищает от нежелательного использования.
- *Содержание*. Сборка содержат типы и ресурсы. Манифест перечисляет названия всех типов и ресурсов, которые видимы извне сборки, и информация о том, где они могут быть найдены в трансляции.
- *Ссылки*. Каждая сборка явно описывает другие сборки, от которых она зависит.

В простейшем случае, сборка состоит из одного файла, который содержит код, ресурсы, метаданные и манифест. В более общем случае, сборка состоит из нескольких файлов, тогда сборка существует как автономный файл или содержится в одном из файлов PE, которые содержат типы и ресурсы.

Соответствующие команды компилятора:

csc /target:library /out:Bank.dll Bank.cs Account.cs

Для сборки из нескольких файлов:

csc /t:library /addmodule:Account.netmodule
/out:Bank.dll Bank.cs

Каждая сборка имеет свой уникальный, сравнимый номер версии. Две сборки, которые отличаются только номером, для исполняющей среды различны. Номер версии состоит из четырех частей:

старшая версия . младшая версия . номер компиляции . редакция

Например, номер 1.5.12540.0 означает, что старшая версия 1, младшая 5, номер компиляции 12540 и редакция 0.

В заключение, сравним пространства имен и сборки. Пространства имен – логический механизм компиляции, он обеспечивает структуру имен сущностей исходного кода. При выполнении программы пространства имен не рассматриваются. Сборки – это физический механизм выполнения, обеспечивающий структуру компонент при исполнении программы. Можно разместить классы одного пространства имен в разных сборках, также в одной сборке могут быть классы разных пространств имен, то есть, никакой связи между этими структурами нет. Элементы пространств имен могут хранится в различных исходных файлах. Элементы сборок не обязательно должны находится в одном файле. Как мы видели, в сборку можно подключать внешние модули. Тогда в сборке записывается именованная ссылка на управляемый модуль.

Вопросы к разделу.

1. Допустим у нас есть два cs файла. Файл alpha.cs содержит класс *Alpha* с internal методом *Method.* Файл beta.cs содержит класс *Beta* с internal методом с тем же названием *Method.* Может ли Alpha.Method вызвать Beta.Method, и наоборот?
2. Аггрегации это связь объектов или классов?
3. Откомпилируется ли следующий код без ошибок?

```
namespace Outer.Inner
{
        class Wibble{}
}
namespace Test
{
```

```
    using Outer.Inner;
    class SpecialWible:Inner.Wible{}
}
```

4. Может ли исполняемое приложение прямо ссылаться на управляемый модуль?

Лабораторная работа.

Модификатор доступа internal. Задание на агрегацию, пространство имен. Время, необходимое на выполнение задания 60 мин.

Упражнение 11.1 Создать новый класс, который будет являться фабрикой объектов класса банковский счет. Изменить модификатор доступа у конструкторов класса банковский счет на internal. Добавить в фабричный класс перегруженные методы создания счета CreateAccount, которые бы вызывали конструктор класса банковский счет и возвращали номер созданного счета. Использовать хеш-таблицу для хранения всех объектов банковских счетов в фабричном классе. В фабричном классе предусмотреть метод закрытия счета, который удаляет счет из хеш-таблицы (методу в качестве параметра передается номер банковского счета).

Использовать утилиту ILDASM для просмотра структуры классов.

Упражнение 11.2

Разбить созданные классы, связанные с банковским счетом, и тестовый пример в разные исходные файлы. Разместить классы в одно пространство имен и создать сборку. Подключить сборку к проекту и откомпилировать тестовый пример со сборкой. Получить исполняемый файл, проверить с помощью утилиты ILDASM, что тестовый пример ссылается на сборку и не содержит в себе классов, связанный с банковским счетом.

Домашнее задание 11.1 Для реализованного класса из домашнего задания 7.1 создать новый класс Creator, который будет являться фабрикой объектов класса здания.

Для этого изменить модификатор доступа к конструкторам класса, в

новый созданный класс добавить перегруженные фабричные методы CreateBuild для создания объектов класса здания. В классе Creator все методы сделать статическими, конструктор класса сделать закрытым. Для хранения объектов класса здания в классе Creator использовать хеш-таблицу. Предусмотреть возможность удаления объекта здания по его уникальному номеру из хеш-таблицы.

Создать тестовый пример, работающий с созданными классами.

Домашнее задание 11.2 Разбить созданные классы (здания, фабричный класс) и тестовый пример в разные исходные файлы. Разместить классы в одном пространстве имен. Создать сборку (DLL), включающие эти классы. Подключить сборку к проекту и откомпилировать тестовый пример со сборкой. Получить исполняемый файл, проверить с помощью утилиты ILDASM, что тестовый пример ссылается на сборку и не содержит в себе классов здания и Creator.

Глава 12. Операторы, делегаты и события.

В этой главе затрагиваются три области полезной функциональности: операторы, делегаты и события.

Операторы – это простейшие компоненты языка. Мы используем операторы для выполнения действий над переменными, для их сравнения.

Делегаты определяют контракт между объектом вызывающим функцию и объектом определяющим вызываемую функцию.

События обеспечивают возможность оповещать своих клиентов о произошедших изменениях.

Операторы

В этом разделе речь будет идти об операторах. Операторы отличаются от методов, их цель облечь выражения в удобную и понятную форму. Конечно, в языке можно обойтись без операторов, используя вместо них хорошо-определенные методы. Например:

```
myIntVar1 = Int.Add( myIntVar2, Int.Add
                (Int.Add ( myIntVar3, myIntVar4 ) , 33);
```

При записи с оператором:

```
myIntVar1 = myIntVar2 + myIntVar3 + myIntVar4 + 33;
```

Операторы определяются в C#, как обычные методы, но спроектированы для более простого использования. Компилятор и среда исполнения автоматически переводят выражения с операторами в соответствующую серию вызовов методов.

C# предлагает большое количество предопределенных операторов. Некоторые из них определяются не как символы, а как обычные идентифи-

каторы, но функциональность для типов данных и классов поддерживаемых .Net полностью определена. Для различных типов один и тот же оператор может реализовывать различные действия. Это свойство называется перегрузкой операторов.

Так операторы задумывались для упрощения выражений, то определять их желательно только, если они обеспечивают это. Операторы можно перегружать когда класс или структура хранят какие-либо данные, для которых действие этого оператора понятно. То есть, например, для класса Employee семантически не понятно действие оператора инкремента(emp++).

Все операторы public static методы и их имена задаются по шаблону operator**op,** где **op** – знак оператора. Для опертора сложения operator+. Список параметров и их типы должны быть определены. Операторы возвращают объект класса.

```
public static Time operator+(Time t1, Time t2)
{
    int newHours = t1.hours + t2.hours;
    int newMinutes = t1.minutes + t2.minutes;
    return new Time(newHours, newMinutes);
}
```

Перегрузка операторов сравнения производится попарно: < и >, <= и =>, == и !=. Если перегрузили один оператор из пары, то обязательно надо определить и второй. Кроме того если перегрузили операторы равенства и неравенства, то необходимо определить виртуальный метод Equals наследуемый от класса Object. Это нужно для того чтобы при сравнении объектов с помощью метода Equals не получить результат противоположный сравнению с ==. Для той же цели перегружается ещё один метод наследуемый от Object – GetHashCode, то есть для равных объектов хеш-код также должен быть одинаков.

Для логических операторов && и || нет прямой перегрузки, для этого используются побитовые операторы. То есть перегрузив операторы &, |, true и false, можно определить логическое И и логическое ИЛИ. Пусть x и y

переменные типа T, тогда логические операторы определяются следующим образом:

x && y - будет выражаться через T.false(x) ? x : T.&(x,y), что означает, что если x в терминах класса T является ложью, то результат равен x, иначе в результате получим побитовое И x и y в терминах класса T.

x || y - выражается через T.true(x) ? x : T.|(x,y), то есть если x в терминах класса T является истинной, то результат равен x, иначе получим побитовое ИЛИ x и y в терминах класса T.

Далее рассмотрим операторы преобразования типов. Есть два варианта преобразования: явное и неявное, приведем несколько примеров:

explicit operator Time (int minutes) – переводит int в Time, является явным так как не любой int можно перевести во время, при отрицательных числах получим исключение.

explicit operator Time (float minutes) – аналогично

implicit operator int (Time t1) – переводит время в целые числа, а эта операция безопасна, можно преобразовывать неявно.

explicit operator float (Time t1) – опасность возникает при преобразовании в вещественные числа значения могут отличаться друг от друга.

implicit operator string (Time t1) – тут все безопасно, но если для класса мы перегружаем преобразование в строку, то необходимо аналогично перегрузить метод ToString(), унаследованный от класса Object.

В заключении добавим, что можно перегружать операторы несколько раз в зависимости от параметров:

public static Time operator+ (Time t1, int hours) {…}

public static Time operator+ (Time t1, float hours) {…}

Создание и использование делегатов.

Делегаты позволяют писать код, динамически изменяющийся при вызове метода. Это гибкое свойство позволяющее методу изменять действия в зависимости от вызова.

Для наглядного примера использования делегатов рассмотрим пример атомной электростанции. Допустим, ядерный реактор электростанции всё время нагревается, необходимо чтобы она не поднималась выше критической. У нас есть возможность в любой момент измерить температуру в реакторе и несколько охлаждающих насосов. Приложение должно запускать время от времени насосы, чтобы температура не достигала критической.

Рассмотрим возможные решения: программа управляющая насосом может также следить за температурой и запускать насос при необходимости. Второй вариант, компонент, следящий за температурой в реакторе, может при необходимости вызывать включение насоса. У обоих вариантов есть недостатки: в первом случае, необходимо определить, с какой периодичностью измерять температуру. Слишком частое измерение может мешать работе самого насоса, при редком измерении можно пропустить быстрое повышение. Во втором случае, программа ядра электростанции должна уметь включать и выключать насосы, причем эта задача может быть достаточно сложной, особенно если насос несколько и они разного типа.

Попробуем написать код для решения нашей задачи. Пусть есть два насоса: электрический и пневматический, у которых определен метод, включающий их.

```
public class ElectricPumpDriver
{
        public void StartElectricPump() {…}
}
public class PneumaticPumpDriver
{
```

```
        public void SwitchOn() {…}
    }
```

Компонента следящая за температурой в реакторе включает насосы. Следующий код описывает основной класс этой компоненты CoreTempMonitor. Он создает несколько насосов и хранит их в коллекции ArrayList. Метод SwitchOnAllPumps проходит по всем насосам и в зависимости от типа насоса вызывает соответствующий метод включающий их.

```
public class CoreTempMonitor
{
    public void Add(object pump)
    {
        pumps.Add(pump);
    }
    public void SwitchOnAllPumps()
    {
        foreach(object pump in pumps)
            if (pump is ElectricPumpDriver)
               ((ElectricPumpDriver)pump.StartElectricPump();
            if (pump is PneumaticDriver)
               ((PneumaticPumpDriver)pump.SwitchOn();
    }
    …
    private ArrayList pumps = new ArrayList();
}
public class ExampleOfUse
{
```

```
        public static void Main()
        {
                CoreTempMonitor ctm = new CoreTempMonitor();
                ElectricPumpDriver ed1 = new ElectricPumpDriver();
                ctm.Add(ed1);
                PneumaticPumpDriver pd1 = new PneumaticPumpDriver();
                ctm.Add(pd1);
                ctm.SwitchOnAllPumps();
}
```

У полученного решения возникают сразу несколько недостатков. Если добавится новый вид насосов, придётся переписывать метод SwitchOnAllPumps. А это означает, проводить заново тестирование всего приложения, что потребует многих затрат. В решении этой проблемы нам помогут делегаты.

Делегат содержит ссылку на метод, а не имя метода. То есть при вызове делегата, вы не знает имя метода который начнет работу. Вызов делегата вызывает метод на который есть ссылка в делегате. В нашей задаче, вместо использования коллекции для хранения насосов, в делегате можно хранить ссылки на методы включающие их.

Делегаты похожи на интерфейсы. Он определяет обязательства между вызывающим и исполняющим методом. Определение метода может быть связано с именем делегата, и компонента может вызвать метод при помощи этого имени. Главное требование к определяемым методам, они долдны иметь одинаковую сигнатуру и возвращаемое значение. Для использования делегатов необходимо объявить его и затем создать элемент.

При объявлении делегата определяем какой список параметров и возвращаемое значение должны обеспечить методы. В нашем случае:

```
public delegate void StartPumpCallback();
```

После объявления делегата, создаем его и добавляем ссылки на вызываемые методы.

```
void Example()
{
    ElectricPumpDriver ed1 = new ElectricPumpDriver();
    StartPumpCallback callback;
    Callback = new StartPumpCallback(ed1.StartElectricPump);
}
```

Делегат – это переменная, которая вызывает метод. Мы вызываем его также как и метод, исключая только то, что он заменяет имя метода. Посмотрим как поменялась наше приложение:

```
public delegate void StartPumpCallback();
public class CoreTempMonitor2
{
    public void Add(StartPumpCallback callback)
    {
        callbacks.Add(callback);
    }
    public void SwitchOnAllPumps()
    {
        foreach(StartPumpCallback callback in callbacks)
            callback();
    }
    private ArrayList callbacks = new ArrayList();
}
public class ExampleOfUse
{
    public static void Main()
    {
```

```
        CoreTempMonitor2 ctm = new CoreTempMonitor2();
        ElectricPumpDriver ed1 = new ElectricPumpDriver();
        ctm.Add(new StartPumpCallback(ed1.StartElectricPump);
        PneumaticPumpDriver pd1 = new PneumaticPumpDriver();
        ctm.Add(new StartPumpCallback(pd1.SwitchOn);
        ctm.SwitchOnAllPumps();
}
```

Определение и использование событий.

В предыдущем разделе мы научились использовать делегаты для решения задачи включения различных насосов в одинаковом стиле. Однако, реактор всё равно должен при достижении определенной температуры сам вызывать насосы. Помочь в этом может механизм событий. События позволяют объекту оповещать другие объекты, что произошли изменения. Остальные объекты могут зарегистрироваться оповещение о каком-либо событии, и они будут получать уведомлении при наступлении этого события.

События позволяют регистрировать интерес одного объекта в изменении второго. Другими словами события позволяют объекту отмечать, что необходимо сообщить ему при изменении некоторого объекта. Издатель – объект, за состояние которого следят. Подписчик – объект который следит за событием, он будет извещен если произойдёт изменение издателя. Событие может иметь ноль или больше подписчиков.

События в C# используют делегаты для вызова методов объектов подписчиков. События поддерживают групповую передачу, то есть одно событие может вызывать несколько делегатов. Однако, в этом случае нельзя быть точно уверенным что все делегаты вызовутся, так как если один из делегатов вызовет исключение, остальные могут остановится и вполне рабочий делегат может не отработать.

Для определения события издатель объявляет делегата и связывает его с событием.

```
public delegate void StartPumpCallback;

private event StartPumpCallback CoreOverheating;
```

Подписчики определяют метод, который будет вызываться при событии. Если событие еще не создано, то подписчик определяет делегата ссылающегося нам метод при создании события. Если событие уже существует, пописчик добавляет делегат вызывающий метод при событии.

```
ElectricPumpDriver ed1 = new ElectricPumpDriver();

PneumaticPumpDriver pd1 = new PneumaticPumpDriver();

…

CoreOverheating = new StartPumpCallback(ed1.StartElectricPump);

CoreOverheating += new StartPumpCallback(pd1.SwitchOn);
```

При использовании делегатов для событий возникает одно ограничение: делегат обязательно должен не возвращать значений, то есть быть типа void. Для уведомления подписчиков необходимо вызывать событие.

```
public void SwitchOnAllPumps()
{
        if(CoreOverheating != null)
                CoreOverheating();
}
```

Если событие происходит, то вызываются все делегаты. Заметим, что сначала надо проверить, а есть ли хоть один подписчик, так при отсутствии подписчиков вызов делегата даст исключение.

Кроме того, что произошло событие, подписчикам бывает интересно при каких условиях оно произошло. Для этого события могут передавать параметры. Для передачи параметров событиями в C# выделен отдельный класс System.EventsArg. Допустим, в нашем примере насосам необходимо знать температуру ядра для правильной регулировки мощности насоса. Для этого

создаем класс параметров:

```
public class CoreOverheatingEventArgs : EventArgs
{
    private readonly int temperature;
    public CoreOverheatingEventArgs(int temperature)
    {
        this.temperature = temperature;
    }
    public int GetTemperature()
    {
        return temperature;
    }
}
```

Объект может быть подписан на несколько от разных издателей и иногда необходимо знать у кого произошло событие. Для этого всегда первым параметром передается ссылка на объект, который вызвал событие sender.

```
public class ElectricPumpDriver
{
    …
    public void StartElectricPump(object sender,
                                  CoreOverheatingEventArgs args);
    {
        //узнаем температуру
        int currentTemperature = args.GetTemperarure();
        // в зависимости от температуры различная мощность насосов
        …
    }
```

```
}
public class PneumaticPumpDriver
{
        …
        public void SwitchOn(object sender, CoreOverheatingEventArgs args);
        {
                int currentTemperature = args.GetTemperarure();
                …
        }
}
public delegate void StartPumpCallback(object sender,
                                        CoreOverheatingEventArgs args);
```

Вопросы по разделу.

1. Может ли арифметическое присваивание (+=, -=, *=, /= и %=) быть перегружено?
2. При каких обстоятельствах оператор преобразования типа должен быть явным?
3. Как вызывается явное преобразование типа?
4. Что такое делегат?
5. Как можно подписаться на событие?
6. Каким образом вызывается метод подписавшийся на событие? Он всегда работает пока ждёт события?

Лабораторная работа.

Задания на определение операторов сложения, умножения, вычитания, деления, равенства, переопределение методов Equals(), ToString(),

GetHashCode(); публикацию событий, передачу параметров событиям. Время, необходимое на выполнение задания 60 мин.

Упражнение 12.1 Для класса банковский счет переопределить операторы == и != для сравнения информации в двух счетах. Переопределить метод Equals аналогично оператору ==, не забыть переопределить метод GetHashCode(). Переопределить метод ToString() для печати информации о счете. Протестировать функционирование переопределенных методов и операторов на простом примере.

Упражнение 12.2 Создать класс рациональных чисел. В классе два поля – числитель и знаменатель. Предусмотреть конструктор. Определить операторы ==, != (метод Equals()), <, >, <=, >=, +, – , ++, --. Переопределить метод ToString() для вывода дроби.

Если останется время, то определить операторы преобразования типов между типом дробь, float, int. Определить операторы *, /, %.

Домашнее задание 12.1 На перегрузку операторов. Описать класс комплексных чисел. Реализовать операцию сложения, умножения, вычитания, проверку на равенство двух комплексных чисел. Переопределить метод ToString() для комплексного числа. Протестировать на простом примере.

Домашнее задание 12.2 Написать делегат, с помощью которого реализуется сортировка книг.

Книга представляет собой класс с полями Название, Автор, Издательство и конструктором.

Создать класс, являющийся контейнером для множества книг (массив книг). В этом классе предусмотреть метод сортировки книг. Критерий сортировки определяется экземпляром делегата, который передается методу в качестве параметра.

Каждый экземпляр делегата должен сравнивать книги по какому-то одному полю: названию, автору, издательству.

Глава 13. Свойства и индексаторы.

Вы можете работать с именованными атрибутами класса такими как поля и свойства. Поля определяются как члены переменные с private доступом. В C# свойства проявляются пользователю как поля, но они используют значения методов get и set.

C# обеспечивает поддержку индексаторов позволяющих обращаться к объекту как к массиву.

Использование свойств

Свойства обеспечивают инкапсуляцию данных в классе. C# добавляет свойства как первоклассный элемент языка. Если мы считаем свойство в качестве поля, то это не отвлекает нас от решения основных задач приложении. Например, сравним два варианта записи: используя свойства и без них.

```
o.SetValue(o.GetValue() + 1);

o.Value++;
```

Для чтения и записи значения свойства мы используем запись в идее поля. Компилятор сам переводит эту запись в вид методов get и set. При обращении к состоянию объекта через свойства мы теряем эффективность, чем при прямом обращении к полям. Однако, если свойство содержит небольшой код и не является виртуальным, то исполняющая среда заменяет вызов метода доступа, на актуальный код метода доступа. Этот процесс называется inlining, и он позволяет работать со свойствами также эффективно, как и с полями.

Свойства – это методы класса, которые обеспечивают доступ к полям класса. Объявление свойства состоит из типа, имени и по-крайней мере одного из двух методов set и get. Они не имеют параметров и не обязательно определять оба из них.

Метод get возвращает значение свойства

```
public string Caption
{
        get{return caption;}
```

```
}
```

Метод get неявно вызывается при использовании свойства в контексте чтения

```
Button myButton;
string cap = myButton.Caption; // вызывает myButton.Caption.get
```

Свойства вызываются без скобок. Чтение свойства не должно менять данные объекта.

Метод set изменяет значение свойства.

```
public string Caption
{
        set { caption = value;}
}
```

Он неявно вызывается при обращении к полю в контексте записи:

```
myButton.Caption = “OK”; // вызывается myButton.Caption.set
```

Переменная value создается и инициализируется компилятором автоматически. Метод set не может возвращать значения, он синтаксически идентичен одиночному присваиванию.

Сравним свойства и поля. Свойства – это логические поля. У них похожий синтаксис создания и использования, но в отличии от полей свойства не хранят значения и у них нет адреса. И они не могут использоваться в качестве ref и out параметров.

При сравнении свойств с методами увидим, что они оба содержат код, используются для реализации деталей. Могут быть виртуальными, абстрактными и перегруженными. Разница синтаксическая – в свойствах не используются скобки, семантическая – свойства не могут быть void и иметь произвольное количество параметров.

При использовании свойств можно определить необходимый тип.

Свойства для чтения/записи – если присутствуют оба метода get и set. Свойства только для чтения, если есть только get, и только для записи, если только set. Кроме того свойства могут быть статичными, связанные с статичными полями.

Индексаторы.

Индексаторы – это элементы позволяющие индексировать объекты как в массивах. Если свойства позволяют обращаться к данным объекта как к полям, то индексаторы позволяют обращаться к членам класса как к массиву.

Пусть объект состоит из нескольких подэлементов (например, listbox состоит из нескольких строк). Индексаторы позволяют обращаться к подэлементам, как в массивах.

```
class StringList
{
    private string[] list;
    public string this[int index]
    {
        get{ return list[index];}
        set{ list[index] = value;}
    }
}
```

Индексатор – это свойство с именем this и квадратными скобками и индексом в них. Для использования индексатора, обращаемся к объекту:

```
StringList myList = new StringList();
myList[3] = “ok”;          // индексатор записывает
string myString = myList[3];    // индексатор читает
```

Сравним индексаторы с массивами: похожи по написанию, разница – можно использовать не только целые индексы, можно перегружать для различных типов индексов. Индексаторы не хранят значения и поэтому не могут быть переданы как ref и out параметры.

Сравним индексаторы со свойствами: оба имеют методы-аксессоры get и set, оба не имеют адреса и не могут быть void. Разница в том, что индексаторы можно перегружать и они не могут быть статичными.

При объявлении индексаторов необходимо определить по крайней мере один параметр, определить значения для всех параметров и не использовать модификаторы ref и out. Можно определить несколько параметров для одного индексатора:

```
class MultipleParameters
{
        public string this[int one, int two]
        {
                get{…}
                set{…}
        }
}
…
MultipleParameters mp = new MultipleParameters();
string s = mp[2,3];
…
```

Вопросы по разделу.

1. Объявите класс Font содержащий свойство только для чтения Name типа string.
2. Объявите класс DialogBox который содержит свойство Caption типа

string.

3. Объявите класс MutableString содержащий индексатор типа char с одним параметром целого типа.
4. Объявите класс Graph содержащий индексатор только для чтения типа double с одним параметром типа Point.

Лабораторная работа.

Задания на свойства, создание и использование индексаторов. Время, необходимое на выполнение задания 30 мин.

Упражнение 13.1 Из класса банковский счет удалить методы, возвращающие значения полей номер счета и тип счета, заменить эти методы на свойства только для чтения. Добавить свойство для чтения и записи типа string для отображения поля держатель банковского счета.

Изменить класс BankTransaction, созданный для хранений финансовых операций со счетом, - заменить методы доступа к данным на свойства для чтения.

Упражнение 13.2 Добавить индексатор в класс банковский счет для получения доступа к любому объекту BankTransaction.

Домашнее задание 13.1 В классе здания из домашнего задания 7.1 все методы для заполнения и получения значений полей заменить на свойства. Написать тестовый пример.

Домашнее задание 13.2 Создать класс для нескольких зданий. Поле класса – массив на 10 зданий. В классе создать индексатор, возвращающий здание по его номеру.

Глава 14. Атрибуты.

Атрибуты – простая техника для добавления метаданных в классы. Они полезны при отладке приложений. В этом разделе изучим синтаксис атрибутов, рассмотрим предопределенные и пользовательские атрибуты. Узнаем как классы могут обращаться к атрибутам и использовать их при работе.

Среда .Net поддерживает атрибуты для возможности расширить язык C#. Атрибуты – декларативный теги, которые используются для передачи информации времени выполнения о поведении элементов, таких как классы, нумераторы и сборки. Можно считать, что атрибуты – это аннотации, который можно хранить и использовать.

В большинстве случаев вы пишите код, который отыскивает значения атрибута во время исполнения и меняет поведение в зависимости от них. В самой простой форме атрибуты используются для документации кода. Атрибуты можно связать с большим количеством элементов исходного кода. Информация о атрибутах хранится в метаданных элемента связанного с ним.

Атрибуты могут быть связаны с разными программными элементами: сборками, модулями, классами, структурами, полями, методами, свойствами и т.д. Описываются следующим образом:

```
[атрибут (позиционные_параметры, именованные_параметры=значение, … )] элемент
```

Определяем имя атрибута, затем можно определить параметры атрибута. Параметры различаются на именованные и позиционные. Позиционные пишутся первыми, они являются обязательными, именованные после них, они опциональны.

```
using System.ComponentModel;
…
[DefaultEvent(“ShowResult”)]
public class Calculator : System.Windows.Form.UserControl
{ … }
```

Одному элементу можно прикрепить несколько атрибутов, они записываются или в различных скобочках, или в одной через запятую.

Возможности предопределенных атрибутов в .Net охватывают широкий диапазон областей от взаимодействия с COM до совместимости с визуальными инструментами. Рассмотрим некоторые из них.

Условные атрибуты – их можно использовать как помощь при отладки кода. Эти атрибуты вызывают условную компиляцию вызовов метода, в зависимости от значения атрибута, который определен. Это позволяет вызывать методы, которые, например, отображают значения переменных, во время отладки кода. После отладки можете изменить атрибут и перекомпилировать код, не изменяя код программы.

```
using System.Diagnostics;
class MyClass
{
        [Conditional(“DEBUGGING”)]
        public static void MyMethod() { … }
}
```

Где идентификатор DEBUGGING определяется следующим образом:

```
#define DEBUGGING
Class AnotherClass
{
        public static void Test()
        {
                MyClass.MyMethod();
        }
}
```

Эта запись означает, компиляция зависит от значения идентификатора,

заданного программистом. При установке значения метод отрабатывает, при неопределенном значении метод не вызывается. Можно объявлять идентификатор при помощи директивы #define или прямо в командной строке при компиляции приложения.

Ограничения на методы с условными директивами: должен возвращать void, не может быть override и не должен быть методом определяющим интерфейс.

Атрибуты DLLImport используются для вызова неуправляемого кода в программы C#. *Неуправляемый код* – это термин для кода разработанного вне среды .NET(например, стандартный C откомпилированный в файл dll). При помощи атрибута DLLImport можно вызывать неуправляемый расположенный в dll-файлах из управляемой среды C#.

При помощи этого атрибута можно вызывать extern методы из неуправляемых dll. При вызове этого метода, общая среда выполнения определяет DLL, загружает её в память вашего процесса, преобразует параметры, если это необходимо, и передает управление адресу начала неуправляемого кода.

```
using System.Runtime.InteropService;
…
public class MyClass()
{
        [DLLImport(“MyDll.dll”, EntryPoint=”MyFunction”)]
        public static extern int MyFunction(string param1);
        …
        int result = MyFunction(“Привет из неуправляемого кода”);
        …
}
```

Также как и в C++ разработчики, работающие в среде Microsoft,

сталкиваются с технологиями COM+. Важное свойство COM+ разработка компонент поддерживающих участие в распределенных транзакциях, которые могут быть разделены на несколько баз данных, машин и компонент. Написать код гарантирующий корректное принятие транзакций в распределенной среде сложная задача. Однако, если вы используете COM+, он берет на себя заботу об управлении интеграции транзакции системы и координации событий сети. В этом случае, остается только определить, что ваш компонент включен, при принятии транзакции приложением. Для этой проверки можно использовать атрибут *Transaction* для класса реализующего этот компонент. Это один из атрибутов .NET, который среда выполнения интерпретирует автоматически

```
using System.EnterpriseServices;
…
[Transaction(TransactionOption.Required)]
public class MyTransactionalComponent
{
        …
}
```

Пользовательские атрибуты.

Если для вашей ситуации вы не нашли ни одного предопределенного атрибута .NET, можно создать свой атрибут. Также как и предопределенные атрибуты, они связаны с каким-то элементом программы, хранятся в их метаданных и обеспечены механизмом получения значения атрибута.

Перед созданием пользовательского атрибута необходимо определить, к какому элементу он будет привязан. Если элементов несколько используется оператор | для их связи:

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Constructor)]
public class MyAttribute: System.Attribute
```

```
{
        …
}
```

После определения области атрибута, необходимо определить все остальную информацию, хранящуюся в атрибуте. Класс атрибута должен явно или неявно наследовать от System.Attribute, имя класса должно заканчиваться на Attribute – это не обязательно, но правило хорошего тона. Все классы атрибутов должны иметь конструктор. Если у атрибута есть какой-либо позиционный параметр, то конструктор должен принимать этот параметр.

Пользовательский атрибут должен определять только один конструктор для принудительной информации. Для опциональных параметров, можно перегружать конструктор. То есть можно использовать именованные параметры для обеспечения опциональных данных. Класс атрибута может содержать свойства для своих опциональных данных. Например, атрибут DeveloperInfo содержит позиционный строковый параметр и именованный Date.

```
public class DeveloperInfoAttribute:System.Attribute
{
        public DeveloperInfoAttribute(string developer)
        {   …   }
        public string Date
        {
                get { … }
                set { … }
        }
}

[DeveloperInfoAttribute(“Bert”, Date = “08-28-2009” )]
public class MyClass
```

```
{
        …
}
```

При компиляции при встрече атрибута будет происходить следующая последовательность действий:

1. Поиск класса атрибута.
2. Проверка области атрибута.
3. Проверка конструктора.
4. Создание объекта.
5. Проверка именованных параметров.
6. Установка значений именованных параметров.
7. Сохранение состояния объекта атрибута в метаданных связанного элемента.

Как мы знаем, к одному элементу могут быть прикреплены несколько атрибутов. Причем атрибуты могут быть одного типа, для этого необходимо при объявлении класса атрибута установить параметр:

```
[AttributeUsage(AttributeTergets.Class, AllowMultiple = true)]
```

Для получения информации хранящейся в метаданных среда .NET Framework использует механизм называющийся рефлексией. Пространство имен System.Reflection содержит классы для извлечения метаданных. Класс MemberInfo – очень полезен для получения информации об атрибутах. Для заполнения массива MemberInfo используется метод GetMembers объекта System.Type.

```
System.Reflection.MemberInfo[] memberInfoArray;
memberInfoArray = typeof(MyClass).GetMembers();
```

Для получения информации о пользовательских атрибутах есть специальный метод GetCustomAttributes объекта MemberInfo.

```
System.Reflection.MemberInfo typeInfo;
typeInfo = typeof(MyClass);
object[] attrs = typeInfo.GetCustomAttributes(false);
```

После получения массива атрибутов, можно проверить их значения.

```
foreach(Attribute atr in attrs){
    if(atr is DeveloperInfoAttribute){
        DeveloperInfoAttribute dia = (DeveloperInfoAttribute) atr;
        Console.WriteLine("{0} {1}", dia.Developer, dia.Date);
    }
}
```

Если нет ни одного атрибута для класса, то метод возвращает null. Кроме того можно поверить наличие атрибута при помощи метода IsDefined.

```
Type devInfoAttrType = typeof(DeveloperInfoAttribute);
if(typeInfo.IsDefined(devInfoAttrType, false))
    object[] attrs = typeInfo.GetCustomAttributes(devInfoAttrType,false);
```

Вопросы по разделу.

1. Можно ли отметить один объект класса используя атрибут?
2. Где хранятся значения атрибутов?
3. Какой механизм используется для определения значения атрибиты при выполнении приложения?
4. Объявить класс атрибута CodeTestAttribute принадлежащий только

классам. Позиционные параметры отсутствуют, есть два именованных Reviewed и HasTestSuite. Эти параметры должны быть булевыми и определяться свойствами для чтения и записи.

5. Объявить класс Widget и при помощи атрибута из предыдущего задания отметить, что класс Reviewed, но не HasTestSuite.
6. Предположим класс Widget имеет метод LogBug. Можно ли его отметить нашим атрибутом?

Лабораторная работа.

Задания на использование условного атрибута(ConditionalAttribute), создание пользовательского атрибута. Время, необходимое на выполнение задания 45 мин.

Упражнение 14.1 Использование предопределенного условного атрибута для условного выполнения кода (указывает компиляторам, что при отсутствии символа условной компиляции, вызов метода или атрибут следует игнорировать).

В классе банковский счет добавить метод DumpToScreen, который отображает детали банковского счета. Для выполнения этого метода использовать условный атрибут, зависящий от символа условной компиляции DEBUG_ACCOUNT. Протестировать метод DumpToScreen.

Упражнение 14.2 Создать пользовательский атрибут DeveloperInfoAttribute. Этот атрибут позволяет хранить в метаданных класса имя разработчика и, дополнительно, дату разработки класса. Атрибут должен позволять многократное использование. Использовать этот атрибут для записи имени разработчика класса рациональные числа (упражнение 12.2).

Домашнее задание 14.1 Создать пользовательский атрибут для класса из домашнего задания 13.1. Атрибут позволяет хранить в метаданных класса имя разработчика и название организации. Протестировать.