МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО
ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Воронежский государственный технический
университет»

А. М. Нужный Н. И. Гребенникова
В. Ф. Барабанов О. Б. Кремер

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА C# В СРЕДЕ VISUAL STUDIO

Учебное пособие

Воронеж 2019

УДК 681.3.06(07)
ББК 32.973-018я7

Рецензенты:

Зав. кафедрой «Вычислительная техника и информационные системы» ФГБОУ ВО «Воронежский государственный Лесотехнический университет имени Г.Ф. Морозова» д.т.н., профессор В.К. Зольников
ФГБОУ ВО «Воронежский государственный технический университет» д.т.н., профессор А.М. Литвиненко

Нужный, А.М.

Р177 Разработка приложений на C# в среде Visual Studio: учебное пособие, ч.1 [Электронный ресурс]. - Электрон, текстовые и граф. данные (11,8 Мб) / А. М. Нужный, Н. И. Гребенникова, В. Ф. Барабанов, О. Б. Кремер. - Воронеж: ФГБОУ ВО «Воронежский государственный технический университет», 2019. - 1 электрон. опт. диск (CD-ROM): цв. - Систем. требования: ПК 500 и выше; 256 Мб ОЗУ; Windows 7; SVGA с разрешением 1024x768;. Adobe Acrobat; CD-ROM дисковод; мышь. - Загл. с экрана.

ISBN 978-5-77-31-0771-2

В пособии рассматриваются основные приемы разработки приложений в среде Visual Studio на языке C#.

Издание предназначено для студентов, обучающихся по направлению 09.03.01 «Информатика и вычислительная техника» (профили «Вычислительные машины, комплексы, системы и сети», «Системы автоматизированного проектирования», «Системы автоматизированного проектирования в машиностроении») при изучении дисциплины «Разработка приложений в Visual Studio».

Ил. 8. Табл. 5. Библиогр.: 6 назв.

УДК 681.3.06(07)
ББК 32.973-018я7

Издается по решению учебно-методического совета Воронежского государственного технического университета

ISBN 978-5-77-31-0771-2

© Нужный А. М., Гребенникова Н. И., Барабанов В. Ф., Кремер О. Б., 2019
© ФГБОУ ВО «Воронежский государственный технический университет», 2019

ВВЕДЕНИЕ

Для эффективного использования технологий разработки программного обеспечения, интегрированных в состав среды Microsoft Visual Studio, разработчик должен уверенно владеть навыками программирования на одном или нескольких языках, поддерживающих платформу .NET Framework.

Большинство языков платформы .NET предполагает использование объектно-ориентированной парадигмы программирования (ООП).

Основным отличием ООП от процедурного программирования является то, что при решении задачи выполняется ее декомпозиция не на процедуры (алгоритмы), а на объекты предметной области, способные выполнять некоторые действия в зависимости от происходящих событий.

Данное учебное пособие позволяет изучить основные аспекты ООП применительно к языку C#.

В первой части пособия рассмотрены основные аспекты организации платформы .NET, знакомство с которыми позволяет понять особенности разработки для этой платформе, оценить преимущества и недостатки управляемого кода.

Во второй части подробно рассмотрены организация и конструкции языка программирования C#, приведены примеры простейших программ.

Третья часть пособия посвящена рассмотрению вопросов объектно-ориентированной методологии в C#.

Подробное описание основных механизмов ООП сопровождается детальными примерами программ на C#.

Все приведенные в пособии иллюстрации – авторские.

1. ПЛАТФОРМА .NET. СРЕДА РАЗРАБОТКИ VISUAL STUDIO

С момента выхода в 2002 году платформы Microsoft .NET, интегрированная среда разработки программного обеспечения Microsoft Visual Studio (MsVS, VS) тесно связана с разработкой программного обеспечения на этой платформе.

В состав VS входит редактор исходного кода, поддерживающий технологию IntelliSense (технология автодополнения Microsoft), встроенный отладчик, редактор форм визуального проектирования графического интерфейса приложения, веб-редактор, дизайнер классов и дизайнер схемы базы данных, а также другие инструменты.

Так как платформа .NET, является наиболее востребованной при программировании для ОС Windows, рассмотрим ее подробнее.

1.1. Платформа .NET Framework

До появления платформы .NET при разработке для ОС Windows наиболее используемой платформой была технология COM (Component Object Model — модель компонентных объектов). Ее популярность объясняется возможностью создания в ней универсальных библиотек, используемых в различных языках программирования. При этом COM отличалась сложной внутренней организацией и усложненной моделью развертывания. К тому же технология COM ориентирована только на использование в среде Windows.

Поэтому программная платформа .NET Framework оказалась весьма востребованной. Ниже приведены основные характеристики .NET:

- кросплатформенность;

- поддержка существующего кода (возможность совместного использования написанных ранее двоичных

компонент COM с более новыми программными компонентами .NET);

- поддержка многочисленных языков программирования (C#, Visual Basic, F# и т.д.);

- наличие общеязыкового исполняющего механизма, обладающего хорошо определенным набором типов, универсальным для всех .NET языков;

- языковая интеграция, обеспечиваемая такими механизмами, как межъязыковое наследование, межъязыковая обработка исключений и межъязыковая отладкой кода;

- наличие обширной библиотеки базовых классов, обеспечивающей простоту низкоуровневых обращений к API-интерфейсам;

- упрощенная модель развертывания, характеризующаяся тем, что в отличии от COM, библиотеки .NET не регистрируются в системном реестре, что позволяет сосуществовать на одном и том же компьютере нескольким версиям одной и той же сборки *.dll.

1.2. Основные модули платформы .NET (CLR, CTS и CLS)

С точки зрения программиста, .NET представляет собой исполняющую среду и обширную библиотеку базовых классов.

Общеязыковая исполняющая среда (Common Language Runtime - CLR) обеспечивает автоматическое обнаружение, загрузку и управление объектами .NET, управляет памятью, координирует потоки и осуществляет проверки, обеспечивающие безопасность.

Общая система типов (Common type System - CTS) содержит описания всех возможных программных конструкций и типов данных, поддерживаемых исполняющей средой и определяет их взаимодействие.

Поскольку невозможно обеспечить поддержку каждым языком, совместимым с .NET, поддержку всего функционала, определенного CTS, существует **общеязыковая спецификация (Common Language Specification - CLS)**, описывающая подмножество типов, поддерживаемых всеми языками .NET.

Помимо вышеперечисленных блоков .NET предоставляет общую для всех языков программирования библиотеку базовых классов, инкапсулирующую такие примитивы, как файловый ввод-вывод, потоки, средства визуализации графики и механизмы взаимодействия с периферийными устройствами, а также поддержку многочисленных служб операционных систем.

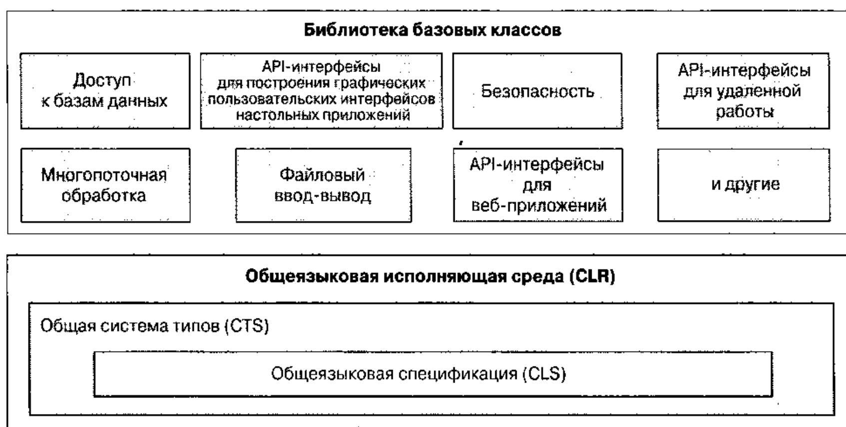


Рис. 1. Отношения между CLR, CTS, CLS и библиотеками базовых классов

Управляемый код. Сборки .NET. Манифест сборки

Основным аспектом, который следует знать, приступая к разработке на платформе .NET, является то, что все языки, предлагаемые Visual Studio для разработки .NET –

приложений (изначально - C#, Visual Basic, C++/CLI, JavaScript и F#), порождают только **управляемый код.**

Это обозначает, что такой код может выполняться **только в рамках исполняющей среды .NET – CLR.**

Какой бы язык .NET не был выбран для программирования, важно понимать, что, хотя двоичные модули .NET имеют то же самое файловое расширение, как и неуправляемые двоичные компоненты Windows (*.dll или *.exe), внутренне они устроены совершенно по-другому.

Двоичные модули .NET содержат независимые от платформы инструкции на промежуточном языке (Intermediate Language — **IL**, он же Common Intermediate Language – **CIL**, он же Microsoft Intermediate Language - **MSIL**) и метаданные типов.

CIL - «высокоуровневый ассемблер» виртуальной машины .NET. Все компиляторы, поддерживающие платформу .NET, транслируют код с языков высокого уровня платформы .NET на язык CIL, а затем CLR, в момент исполнения программы преобразует байт-код CIL в Native-код, т.е. в собственный код операционной системы.

Такой подход несет в себе ряд ограничений (к примеру, использовать C# для построения COM-сервера или неуправляемого приложения C/C++ не допускается), но при этом облегчает создание кросплатформенных приложений, а также делает возможной языковую интеграцию.

Исполняемый файл *.exe или библиотека *.dll, созданный с помощью .NET-компилятора, называется **сборкой (assembly).**

В сборке содержит CIL-код, схожий концептуально с байт-кодом Java, поскольку так же не компилируется в инструкции, специфичные для платформы, до самого момента исполнения. Компиляция выполняется, обычно, когда на блок инструкций CIL, содержащий, например, реализацию метода, производится ссылка при его использовании исполняющей средой .NET.

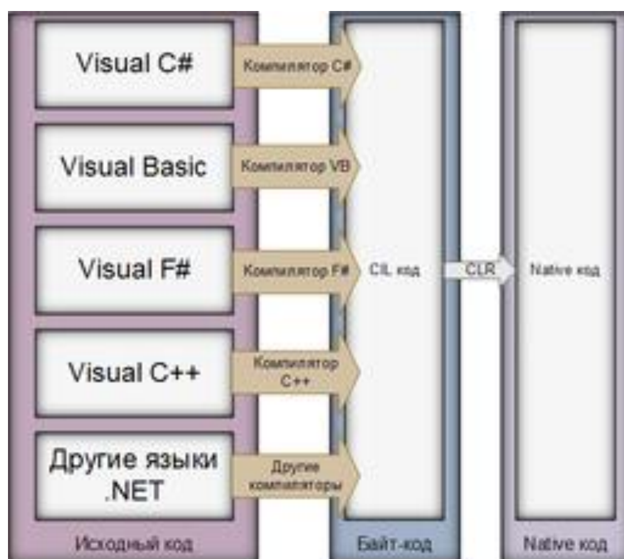


Рис. 2. Управляемый код .NET

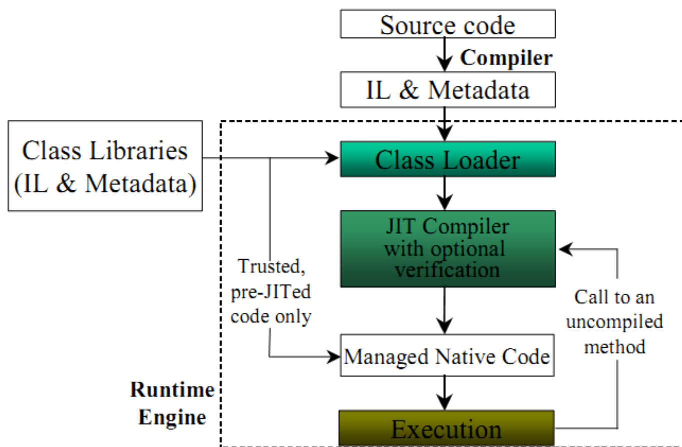
Компиляцию CIL-кода в инструкции ОС осуществляет оперативный (Just-in-time – JIT) компилятор, называемый иногда Jitter-ом.

Помимо CIL-кода, в сборках содержатся также метаданные, детально описывающие характеристики каждого “типа” внутри двоичного модуля.

С учетом вышесказанного, общая сема архитектуры .NET может быть представлена на рис. 3.

Помимо метаданных типов, сборка .NET содержит также метаданные самоописания, называемые **манифестом**.

Манифест содержит перечисление всех внешних сборок, необходимых для корректного функционирования текущей сборки, версию сборки, информацию об авторских правах и т.д. Манифест, как и метаданные типов, генерируется компилятором.



В табл. 1 приведена структура метаданных манифеста.

Рис. 3. Общая схема архитектуры .NET

Таблица 1

Структура метаданных манифеста

Имя таблицы метаданных манифеста	Содержимое таблицы
AssemblyDef	Для сборки содержит единственную запись, включающую имя сборки версию, номер редакции, региональные стандарты, флаги, хэш алгоритм и открытый ключ издателя.
FileDef	Содержит по одной записи для каждого Portable Executable (PE)-файла и файла ресурсов, входящих в состав сборки. Хранит имя, хэш и флаги. Если в сборке один файл, таблица FileDef пуста.

Окончание табл.1

ManifestResourceDef	Содержит по одной записи для каждого ресурса, из состава сборки. Хранит имя ресурса, флаги (public / private), индекс для таблицы FileDef. Если ресурс не является отдельным файлом (например, JPEG- или GIF-файлом), он хранится в виде потока в составе PE-файла. В случае встроенного ресурса запись также содержит смещение, указывающее начало потока ресурса в PE-файле
ExportedTypesDef	Содержит записи для всех открытых типов, экспортируемых всеми PE-модулями сборки. Хранит имя типа, индекс для таблицы FileDef (указывающий файл сборки, в котором реализован этот тип), а также индекс для таблицы TypeDef.

Подводя итог вышесказанному можно сделать следующие выводы о достоинствах и недостатках платформы .NET.

Достоинства платформы .NET:

- наличие цельной объектно-ориентированной модели программирования, существенно упрощающей разработку программ;
- возможность разработки многоплатформенных приложений;
- наличие механизмов автоматического управления ресурсами;
- упрощенное, по сравнению с COM, развертывание приложений;
- полный отказ от использования реестра;

- использование безопасных типов, и повышение безопасности приложений;
- наличие единой модели обработки ошибок;
- эффективные механизмы обеспечения межъязыкового взаимодействия;
- возможность проведения межъязыковой отладки за счет использования единой среды разработки;
- расширение возможностей повторного использования кода.

Недостатки платформы .NET:

- замедление выполнения программ из-за компиляции CIL-кода при выполнении;
- необходимость изменения стандартов большинства языков программирования, поддерживающих .NET.

2. C#. НАЗНАЧЕНИЕ, ОРГАНИЗАЦИЯ, КОНСТРУКЦИИ ЯЗЫКА

C# - язык программирования с использованием платформы .NET, позволяющий выполнять быструю разработку крупномасштабных приложений.

В то время, как C++ рекомендуют использовать для разработки наиболее критичных участков кода, где **важно быстродействие**, C# хорош для написания интерфейсов (в том числе и web), разработки различных сервисов и служб, где важна **скорость** разработки.

C# позволяет быстрее разрабатывать бизнес-приложения. Однако приложения будут работать под управлением среды .NetFramework (Mono для Linux).

Большинство типичных потребностей бизнеса вполне можно реализовать с помощью C# гораздо быстрее, чем на C++.

2.1. Типы данных, их классификация

Как отмечалось выше, для обеспечения языкового взаимодействия платформа .NET использует **общую систему типов (CTS),** содержащую описание типов данных и программных конструкций, поддерживаемых исполняющей средой.

В CTS **тип** — это просто общий термин, используемый для ссылки на члены из следующего набора:

- класс;
- интерфейс;
- структура;
- перечисление;
- делегат.

Типы классов CTS

Класс – одно из основных понятий ООП. Класс может включать любое количество **членов-данных** (поля,

константы, *события*) и **членов-функций** (методы, свойства, конструкторы и пр.).

В C# классы объявляются с помощью ключевого слова class.

Листинг 1. Пример.

```
// Тип класса C# с одним методом.
class Calc
{
	public int Add(int x, int y)
	{
		return x + y;
	}
}
```

Типы интерфейсов CTS

Интерфейсы —именованные коллекции определений абстрактных членов, которые могут поддерживаться в заданном классе или структуре. Определяются с помощью ключевого слова **interface.** Имена всех интерфейсов .NET начинаются с прописной буквы I. как показано в листинге 2.

Листинг 2.

```
// Тип интерфейса в C# обычно объявляется
// открытым, чтобы позволить типам в других
// сборках реализовать его поведение.
public interface IDraw
{
void Draw();
}
```

Реализованный классом или структурой отдельный интерфейс предоставляет возможность получать доступ к определенной функциональности, используя ссылку на этот интерфейс.

Типы структур CTS

Структура – тип, определяемый пользователем, наилучшим образом подходящий для моделирования

геометрических и математических данных. Создается в C# с применением ключевого слова **struct**.

Листинг 3. Пример.

```
// Тип структуры в C#.
struct Point
{
        // Могут содержать поля
        public int xPos, yPos;
        // Могут содержать конструкторы
        public Point(int x, int y)
        { xPos = x; yPos = y;}
        // Могут определять методы
        public void PrintPosition()
        {
        Console.WriteLine("({0}, {1})", xPos, yPos);
        }
}
```

Типы перечислений CTS

Перечисление - программная конструкция, позволяющая группировать пары “имя-значение”. Создаются при помощи ключевого слова **enum:**

Листинг 4.

```
// Тип перечисления C#.
enum Days_Value
{
        Monday = 1,
        Tuesday = 2,
        Wednesday = 3,
        Thursday=4,
        Friday=5,
        Saturday=6,
        Sunday=7
}
```

Типы делегатов CTS

Делегаты - аналог безопасных к типам указателей на функции в стиле С. Делегат .NET представляет собой класс, порожденный от System.MulticastDelegate, а не просто

указатель на низкоуровневый адрес в памяти. Объявляются с помощью ключевого слова **delegate**.

Листинг 5.

```
delegate int BinaryOp(int x, int y); // может "указывать" на любой метод, возвращающий значение int и принимающий два значения int.
```

Члены типов CTS

Большинство рассмотренных типов способно поддерживать любое количество членов. Формально **член типа ограничен набором (конструктор, финализатор, статический конструктор, вложенный тип, операция, метод, свойство, индексатор, поле, поле только для чтения, константа, событие).**

2.2. Классификация типов CTS

Все типы CTS можно разделить на следующие категории:

- примитивные типы, типы-значения и ссылочные типы;
- объектные и интерфейсные типы.

Как следует из рисунка, два основных вида данных в системе типов .NET — это **типы-значения (value types)** и **ссылочные типы (reference types)**.

Тип-значение - последовательность битов в стеке программы, в которых хранится значение. Объем памяти для значимых объектов определяется при компиляции.

Ссылочный тип - ссылка на адрес в динамически распределяемой памяти (heap - куча).

Основная разница между типом-значением и ссылочным типом в том, что два разных объекта **ссылочного тип**а, представляющие один и тот же класс, могут содержать абсолютно одинаковые данные, но при этом не будут равными, так как указывают на разные участки памяти.

В то время любые две целочисленные переменные будут считаться равными, если содержат одинаковое число.

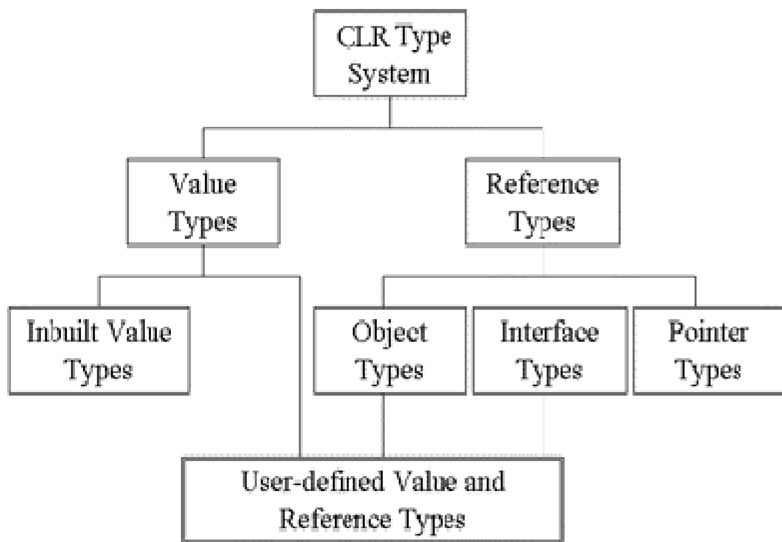


Рис. 4. Классификация типов CTS

2.3. Встроенные (примитивные) типы

Наиболее часто употребляемые типы данных прописаны в компиляторе напрямую, что позволяет обращаться с ними по упрощенной схеме:

string str = "Привет!";,

это сокращенная форма записи, подразумевающая следующее присваивание:

System.String str = new System.String();
str = "Привет!";

Практически все примитивные типы являются **типами-значениями** и память под них выделяется в стеке программного потока. Большинство примитивных типов проецируется на типы данных, существующие в базовой библиотеке классов .NET.

В табл. 2 перечислены примитивные типы C#.

Таблица 2

Примитивные типы C#

Название типа	Ключе-вое слово	Тип .NET	Диапазон значений	Описание	Размер, битов
Логический	bool	Boolean	true, false		
Целые	sbyte	SByte	От –128 до 127	Со знаком	8
	byte	Byte	От 0 до 255	Без знака	8
	short	Int16	От –32768 до 32767	Со знаком	16
	ushort	UInt16	От 0 до 65535	Без знака	16
	int	Int32	От –2 10^{9} до 2 10^{9}	Со знаком	32
	uint	UInt32	От 0 до 4 10^{9}	Без знака	32
	long	Int64	От –9 10^{18} до 9 10^{18}	Со знаком	64
Целый	ulong	UInt64	От 0 до 18 10^{18}	Без знака	64
Символь-ный	char	Char	От U+0000 до U+ffff	Unicode-символ	16
Веществен-ные	float	Single	От 1.5 10^{-45} до 3.4 10^{38}	7 цифр	32
	double	Double	От 5.010^{-324} до 1.7 10^{308}	15–16 цифр	64
Финансо-вый	decimal	Decimal	От 1.0 10^{-28} до 7.9 10^{28}	28–29 цифр	128
Строковый	string	String	Длина ограничена объемом доступной памяти	Строка из Unicode-симво-лов	

Продолжение табл.2

Название типа	Ключе-вое слово	Тип .NET	Диапазон значений	Описание	Размер, битов
object	object	Object	Можно хранить все, что угодно	Всеоб-щий предок	

Тип object

Последний тип, указанный в таблице (**object**), является базовым (родительским) для всех остальных типов. Поэтому переменная типа object может ссылаться на объект любого другого типа.

Тип object в C# считается обозначением класса System.Object и имеет ряд методов, некоторые из которых приведены ниже:

ToString()- возвращает символьное описание объекта, для которого вызывается. Автоматически вызывается при консольном выводе объекта с помощью метода WriteLine().

Equals (object) - определяет, ссылочную эквивалентность двух объектов.

Finalize()-вызывается при сборке мусора для очистки ресурсов, занятых ссылочным объектом.

GetType() - возвращает экземпляр класса, унаследованный от System.Type. Этот объект может предоставить большой объем информации о классе, членом которого является исследуемый объект.

2.4. Преобразование типов

В программировании часто происходит преобразование величин из одного типа в другой. Обычно это происходит при выполнении различных операций. Арифметические операции, например, определены для типов, int и больше. Значит, при использовании в арифметической операции только операторов типов sbyte, byte, short и ushort, перед выполнением операции все операнды будут преобразованы

в int и результат любой арифметической операции будет иметь тип не менее int.

Такое преобразование называется **неявным** и возможно только в случае приведения более коротких типов к более длинным для сохранения значимости и точности.

На рис. 5 указаны варианты неявных арифметических преобразований типов.

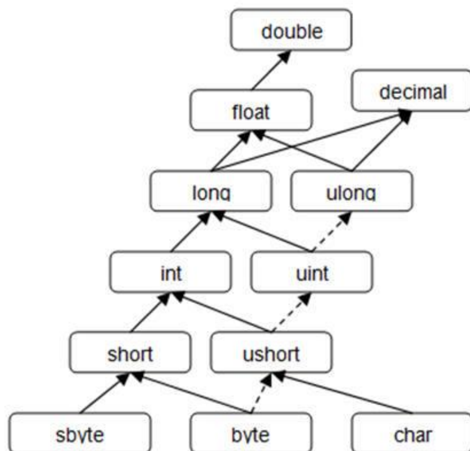


Рис. 5. Неявные арифметические преобразования типов

При необходимости можно выполнить ***явное*** преобразование типа с помощью операции:

(тип)x.

Кроме этого, в пространстве имен **System** имеется класс **Convert**, используемый для преобразования типов данных, например:

byte sum = Convert.ToByte(var1 + var2).

Важно! Следует учитывать, что при некорректном преобразовании типов возможна потеря данных, связанная с возникновением **условий переполнения**

(overflow) или **потерей значимости (underflow).** При этом по умолчанию сообщение об ошибке не выдается.

Контроль за такими ситуациями может быть осуществлен с использованием ключевого слова **checked,** позволяющего генерировать исключение «**OverflowException».**

Листинг 6. Пример.

```
static void Main(string[] args)
      {
         byte var1 = 250;
         byte var2 = 150;
         try
         {
            byte sum = checked(byte)(var1 + var2);
            Console.WriteLine("Сумма: {0}", sum);
            Console.ReadLine();
         }
         catch (OverflowException ex)
         {
            Console.WriteLine(ex.Message);
            Console.ReadLine();
         }
      }
```

2.5. Типы-значения

Типы-значения — это не только примитивные типы, но и некоторые другие типы **CTS**, например, структуры (struct), перечисления (enum).

Важно! Тип-значение содержит само значение переменной, а не ссылку на него. Положительной стороной такого подхода является экономия памяти наряду с улучшением производительности, поскольку отпадает потребность в разыменовывании указателя и создании новой копии объекта в "куче".

Минусом является то, что типы-значения не могут наследовать от других типов и от них также нельзя ничего унаследовать – они являются "запечатанными" - **sealed**.

Важно помнить, что при присваивании типов-значений происходит копирование значения, поэтому типы-значения редко используются, если их надо часто передавать в качестве параметров.

2.6. Ссылочные типы

Ссылочные типы – это указатель на какую-либо структуру. К числу ссылочных типов относятся **объекты (object)**, **классы (class), интерфейсы (interface), делегаты (delegate), а также строки и массивы**. Место для хранения переменных ссылочных типов всегда выделяется в «куче» (heap) – динамически распределяемой памяти, объем которой для хранения объекта определяется во время выполнения программы.

При обращении к значению ссылочного типа происходит разыменовывание указателя, поэтому они считаются более тяжеловесными, чем типы-значения.

При создании ссылочные типы инициализируются значением null.

Важно! При присваивании ссылочных типов происходит копирование адреса, а не значения переменной, поэтому изменение значений одной переменной может повлиять на другие, указывающие на тот же объект!

2.7. Упаковка и распаковка

Часто возникает необходимость взаимного преобразования значимых и ссылочных типов. Для этого используются операции **упаковки (boxing)** и **распаковки (unboxing)**.

В приведенном ниже примере создается значимая неупакованная переменная v, затем производится занесение ее упакованного значения в ссылочный объект o. В последнем операторе команда «(Int32) o» производит распаковку «o» с преобразованием к целому типу. Но, поскольку здесь

используется операция конкатенации (объединения) строк, а строки представляют собой ссылочный тип, значимая величина «o» опять преобразуется к ссылочному типу. Поэтому правильнее здесь обойтись без распаковки.

Листинг 7. Пример упаковки и распаковки.

```
public static void Main()
{
        Int32 v = 5; // создание неупакованной переменной
                // значимого типа
        Object o = v; // o ссылается на упакованную версию v
        v = 123; // изменение неупакованной переменной
        Console.WriteLine (v + ", " + (Int32) o);
        // выводится "123, 5"
}
```

2.8. Ключевое слово var. Неявная типизация

Начиная с версии Visual C# 3.0, появилась возможность неявной типизации локальных переменных с использованием ключевого слова **var**.

Листинг 8. Пример.

```
var i = 10; // Неявная типизация.
int i = 10; // Явная типизация.
```

В этом случае тип переменной определяется компилятором из правой части оператора инициализации. Тип может быть значимым, ссылочным, анонимным, пользовательским, либо типом, определяемым библиотекой классов .NET Framework.

2.9. Операции C#

Прежде чем начать разговор об операторах C#, вспомним несколько понятий, относящихся к этой теме.

Программный код и прочие тексты при создании программ пишутся с использованием **алфавита** языка программирования.

Алфавит C# включает:

- буквы (латинские и национальных алфавитов) и символ подчеркивания (_), употребляемый наряду с буквами;
- цифры;
- специальные символы, например +, *, { и &;
- пробельные символы (пробел и символы табуляции);
- символы перевода строки.

Алфавит C# использует кодировку символов Unicode.

Из символов составляются более крупные строительные блоки: **лексемы, директивы препроцессора и комментарии.**

Лексема (token) — это минимальная единица языка, имеющая самостоятельный смысл. Существуют следующие ***виды лексем***:

- имена (идентификаторы);
- ключевые слова;
- знаки операций;
- разделители;
- литералы (константы).

Идентификаторы служат для того чтобы обращаться к программным объектам и различать их. Характеристики:

- в идентификаторе могут использоваться буквы, цифры (не могут стоять в начале) и символ подчеркивания;
- прописные и строчные буквы различаются (C#-регистрозависимый язык);
- пробелы в идентификаторах не допустимы, а употребление букв национальных алфавитов – допустимо;
- в идентификаторах можно применять escape-последовательности Unicode (представление символа в виде шестнадцатеричного кода префиксом \u, например, \u00F2);
- идентификаторы не должны совпадать с ключевыми словами.

Ключевые слова - это зарезервированные идентификаторы, которые имеют специальное значение для компилятора, табл. 3.

Таблица 3

Перечень ключевых слов C#

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

Знак операции — это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются.. Символы, составляющие знак операций, могут быть как специальными, например, &&, | и <, так и буквенными, такими как as или new.

Литералами, или константами, называют неизменяемые величины. В C# бывают логические, целые, вещественные,

Управляющей последовательностью, или простой **escape-последовательностью,** называют определенный символ, предваряемый обратной косой чертой, табл. 4.

Таблица 4

Управляющие последовательности в C#

Вид	Наименование
\a	Звуковой сигнал
\b	Возврат на шаг
\f	Перевод страницы (формата)
\n	Перевод строки
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\\	Обратная косая черта
\'	Апостроф
\"	Кавычка
\0	Нуль-символ

Комментарии предназначены для записи пояснений к программе и формирования документации. В C# есть два вида комментариев:

//однострочный комментарий;

/*многострочный

комментарий */.

2.10. Операторы C#

Язык C# предоставляет большой набор операторов, которые представляют собой символы, определяющие операции, которые необходимо выполнить с выражением. Операции над целыми типами, такие как ==, **!**=, <, >, <=, >=, **binary** +, **binary** **-**, ^, **&**, |, ~, ++, **--** и **sizeof(),** обычно разрешены в перечислениях.

Кроме того, многие операторы могут перегружаться (переопределяться) пользователем. Таким образом, их

значение при применении к пользовательскому типу меняется.

Табл. 5 содержит операторы C#, сгруппированные в порядке убывания приоритета. Операторы одной группы имеют одинаковый приоритет.

Таблица 5

Операторы C#

Обозначение оператора	Описание
Основные	
x.y	Доступ к членам
x?.y	Условный доступ к членам
f(x)	Вызов метода и делегата
a[x]	Доступ к массиву и индексатору
a?[x]	Условный доступ к массиву и индексатору
x++	Постфиксный инкремент
x--	Постфиксный декремент
new T(...)	Создание объекта и делегата
new T(...) {...}	Создание объекта с инициализатором.
new {...}	Анонимный инициализатор объекта.
new T[...]	Создание массива
typeof(T)	Получение объекта System.Type для T
checked(x)	Вычисление выражения в проверенном контексте
unchecked(x)	Вычисление выражения в непроверенном контексте
default (T)	Получение значения по умолчанию для типа T
Унарные операторы	
+x	идентификации
-x	Отрицание
!x	Логическое отрицание
~x	Поразрядное отрицание
++x	Префиксный инкремент
--x	Префиксный декремент
(T)x	Явное преобразование x в тип T

Продолжение табл.5

Обозначение оператора	Описание
Мультипликативные операторы	
*	Умножение
/	Деление
%	Остаток
Аддитивные операторы	
x + y	Сложение, объединение строк, объединение делегатов
x - y	Вычитание, удаление делегатов
Операторы сдвига	
x <<y	Сдвиг влево
x >> y	Сдвиг вправо
Относительные операторы и операторы типов	
x < y	Меньше
x > y	Больше
x <= y	Меньше или равно
x >= y	Больше или равно
x is T	Возвращает значение true, если x относится к типу T, в противном случае возвращает значение false
x as T	Возвращает x типа T или значение NULL, если x не относится к типу T
Операторы равенства	
x == y	Равно
x != y	Не равно
Логические, условные операторы и NULL-операторы	
x & y	Логическое И. Поразрядное И для операндов целочисленного типа, логическое И для операндов логического типа
x ^ y	Логическое исключающее ИЛИ. Поразрядное исключающее ИЛИ для операндов целочисленного типа, логическое исключающее ИЛИ для операндов логического типа

Окончание табл.5

<table>
<tr><td>Обозначение оператора</td><td>Описание</td></tr>
<tr><td>x | y</td><td>Логическое ИЛИ. Поразрядное ИЛИ для операндов целочисленного типа, логическое ИЛИ для операндов логического типа</td></tr>
<tr><td>x && y</td><td>Условное И. Равно y, только если x имеет значение true</td></tr>
<tr><td>x || y</td><td>Условное ИЛИ. Равно y, только если x имеет значение false</td></tr>
<tr><td>x ?? Y</td><td>Объединение со значением NULL. Равно y, если x имеет значение NULL, в противном случае равно x</td></tr>
<tr><td>x ? y : z</td><td>Условие. Равно y, если x имеет значение true, и z, если x имеет значение false</td></tr>
<tr><td colspan="2">Операторы присваивания и анонимные операторы</td></tr>
<tr><td>=</td><td>Назначение</td></tr>
<tr><td>x op= y</td><td>Составное присваивание. Поддерживает следующие
операторы: +=, =, *=, /=, %=, &=, |=, ^=, <<=, >>=.</td></tr>
<tr><td>(T x) => y</td><td>Анонимная функция (лямбда-выражение)</td></tr>
</table>

Подробное описание всех операторов C# можно посмотреть в MSDN по ссылке:
https://msdn.microsoft.com/ru-ru/library/6a71f45d(v=vs.120).aspx

Если в выражении стоит несколько операторов с одинаковым приоритетом, то порядок их применения определяется на основе **ассоциативности**.

Ассоциативность большинства бинарных операторов – левая, значит, операции с равным приоритетом выполняются слева направо. Операторы присваивания имеют правую ассоциативность. Порядок, определяемый приоритетом и ассоциативностью операторов, можно изменить с помощью скобок.

2.11. Управляющие конструкции языка

2.11.1. Условные конструкции C#

Условные конструкции являются одним из основных компонентов большинства языков программирования, так как позволяют выбрать путь выполнения программы в зависимости от каких-либо условий. В C# используются 3 условные конструкции.

1. **if …else**.

Листинг 9. Пример.

```
int num1 = 8;
int num2 = 6;
if(num1 > num2)
{
   Console.WriteLine($"Число {num1} больше числа {num2}");
}
else if (num1 < num2)
{
   Console.WriteLine($"Число {num1} меньше числа {num2}");
}
else
{
   Console.WriteLine("Число num1 равно числу num2");
}
```

2. **switch..case.**

Листинг 10. Пример.

```
Console.WriteLine("Нажмите Y или N");
string selection = Console.ReadLine();
switch (selection)
{
        case "Y":
        Console.WriteLine("Вы нажали букву Y");
        break;
        case "N":
        Console.WriteLine("Вы нажали букву N");
        break;
        default:
        Console.WriteLine("Вы нажали неизвестную букву");
        break;
}
```

3. Тернарная операция

[первый операнд - условие] ? [второй операнд] : [третий операнд].

Листинг 11. Пример.

```
int x=3;
int y=2;
Console.WriteLine("Нажмите + или -");
string selection = Console.ReadLine();
int z = selection=="+"? (x+y) : (x-y);
Console.WriteLine(z);
```

2.11.2. Циклы в C#

Цикл – управляющая конструкция, позволяющая выполнять определенный набор действий несколько раз.

В C# определены следующие циклы:

- for;
- foreach;
- while;
- do...while;

Листинг 12. Пример цикла for.

```
for (int i = 0; i < 9; i++)
{
   Console.WriteLine($"Квадрат числа {i} равен {i*i}");
}
```

В этом примере знак $ в последней строке идентифицирует строку как интерполированную, т.е. строка может содержать интерполированные выражения, которые при выводе будут заменяться строковыми выражениями.

Цикл foreach аналогичен циклу for, но в качестве счетчика использует элементы экземпляра любого типа, включающего открытый метод GetEnumerator или реализующего интерфейс System.Collections.IEnumerable.

Цикл do – сначала выполняет код цикла, потом проверяет условие. Гарантирует выполнение кода цикла хотя бы один раз.

Листинг 13. Пример.

```
int i = 6;
do
{
    Console.WriteLine(i);
    i--;
}
while (i > 0);
```

Цикл while – выполняется только после успешной проверки истинности условия.

Листинг 14. Пример.

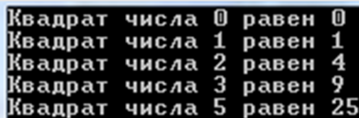
```
int i = 6;
while (i > 0)
{
    Console.WriteLine(i);
    i--;
}
```

Оператор **break** позволяет прервать выполнение любого из вышеперечисленных циклов.

Оператор **continue** позволяет пропустить итерацию.

Листинг 15. Пример.

```
for (int i = 0; i<9;i++)
{
    if (i == 4)
        continue;
    if (i == 6)
        break;
    Console.WriteLine($"Квадрат числа {i} равен {i * i}");     }
```



```
Квадрат числа 0 равен 0
Квадрат числа 1 равен 1
Квадрат числа 2 равен 4
Квадрат числа 3 равен 9
Квадрат числа 5 равен 25
```

2.12. Массивы

Массив – набор однотипных данных. Объект массив относится к классу **System.Array** и обладает всеми его методами.

Объявление массива похоже на объявление переменной, но после указания типа ставятся квадратные скобки.

Листинг 16.

```
int[] arr1;
arr1= new int[3];
arr1[0]=1;
arr1[1]=2;
arr1[2]=3;
```

После объявления массива следует определить его размер, после чего массив можно заполнять элементами.

Ниже приведены альтернативные способы определения массивов.

Листинг 17.

```
int[] arr2 = new int[4] { 1, 2, 3, 5 };
int[] arr3 = new int[] { 1, 2, 3, 5 };
int[] arr4 = new[] { 1, 2, 3, 5 };
int[] arr5 = { 1, 2, 3, 5 };
```

Если значения элементов массива явно не определены, то они заполняются значением, определенным для выбранного типа по умолчанию (для int это 0).

Многомерные массивы

C# поддерживает также массивы с размерностью больше единицы. При описании многомерного массива его размерности разделяются запятыми.

Ниже представлены варианты создания двумерных массивов.

Листинг 18.

```
int[,] nums1;
int[,] nums2 = new int[2, 3];
int[,] nums3 = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };
int[,] nums4 = new int[,] { { 0, 1, 2 }, { 3, 4, 5 } };
int[,] nums5 = new [,]{ { 0, 1, 2 }, { 3, 4, 5 } };
int[,] nums6 = { { 0, 1, 2 }, { 3, 4, 5 } };
```

Как можно видеть выше, при задании двумерного массива в размерности ставится одна запятая. В случае трехмерного массива запятых станет две.

Листинг 19.

```
int[,,] nums3 = new int[2, 3, 4];
```

Для определения размерности массива следует использовать метод GetUpperBound(dimension), возвращающий индекс последнего элемента в выбранной размерности. А используя этот метод совместно с методом Length, возвращающим количество элементов массива, несложно определить количество рядов и столбцов двумерного массива.

Листинг 20.

```
int[,] arr1 = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };

int rows =arr1.GetUpperBound(0) + 1; //количество строк
int columns = arr1.Length / rows;         //количество столбцов
```

Помимо этого, массивы поддерживают ряд **статических методов** (статические методы могут вызываться с указанием класса в целом, а не отдельного объекта):

Array.Clear() очищает массив, устанавливая для всех его элементов значение по умолчанию;

Array.Copy() копирует часть одного массива в другой массив;

Array.Exists() проверяет, содержит ли массив определенный элемент;

Array.Find() находит элемент, который удовлетворяет определенному условию;

Array.FindAll() находит все элементы, которые удовлетворяют определенному условию;

Array.IndexOf() возвращает индекс элемента;

Array.Resize() изменяет размер одномерного массива;

Array.Reverse() располагает элементы массива в обратном порядке;

Array.Sort() сортирует элементы одномерного массива.

Листинг 21. Примеры использования.

```
int[] numbers = { -4, -3, -2, -1,0, 1, 2, 3, 4 };

// расположим в обратном порядке
Array.Reverse(numbers);

// уменьшим массив до 4 элементов
Array.Resize(ref numbers, 4);

foreach(int number in numbers)
{
   Console.Write($"{number} \t");
}
Результат: 4       3       2       1
```

Неровные массивы

Помимо **одномерных** и **многомерных** массивов в C# возможно создание **неровных или зубчатых** массивов, которые представляют собой массив массивов.

Описание **неровного массива** выглядит следующим образом.

Листинг 22.

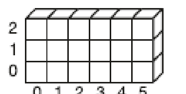
```
int[][] nums = new int[3][];
nums[0] = new int[2] { 1, 2 };          // выделяем память для
                                        //первого подмассива
nums[1] = new int[3] { 1, 2, 3 };       // выделяем память для
                                        //второго подмассива
nums[2] = new int[5] { 1, 2, 3, 4, 5 }; // выделяем память для
                                        //третьего подмассива
```

Почему такие массивы называются неровными, видно на приведенной ниже графической интерпретации трех видов массивов.

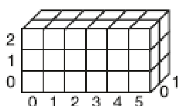
К неровным массивам также применимы перечисленные методы.

Одномерный массив

Многомерные массивы

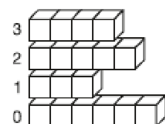


Двухмерный массив
int[3,6]



Трехмерный массив
int[3,6,2]

Зубчатый массив



Зубчатый массив
int[4][]

Рис. 6. Три вида массивов

2.13. Строки. Работа со строками

Строковые значения в C# представлены типом string, являющимся псевдонимом для класса **System.String**. Строки представляют собой объекты, содержащие набор символов Unicode размером до 2 Гб, что соответствует 1 миллиарду символов.

Листинг 23. Способы создания строк.

```
string s1 = "hello"; //создание переменной и присвоение знач.
string s2 = null; // т.к. строка это объект, может хранить null

//создание строк с использованием конструктора
string s3 = new String('a', 6); // результатом будет строка
                                //"aaaaaa"
string s4 = new String(new char[]{'w', 'o', 'r', 'l', 'd'});
```

К строке можно обращаться как к массиву символов и получить по индексу любой из ее символов.

Листинг 24. Пример.

```
string s1 = "hello";
char ch1 = s1[1]; // символ 'e'
Console.WriteLine(ch1);
Console.WriteLine(s1.Length);
```

Свойство **Length** возвращает длину строки.

Класс String поддерживает большое количество методов. Рассмотрим некоторые из них.

Concat – статический метод для объединения строк. Работает аналогично оператору «+».

Листинг 25.Пример.

```
string s1 = "hello";
string s2 = "world";
string s3 = s1 + " " + s2; // результат: строка "hello world"
string s4 = String.Concat(s3, "!!!"); // результат: строка "hello
                                   //world!!!"
 Console.WriteLine(s4);
```

Compare – статический метод для сравнения двух строк. Принимает две строки и возвращает число. Если первая строка по алфавиту стоит выше второй, то возвращается число меньше нуля. В противном случае возвращается число больше нуля. Если строки равны, то возвращается число 0.

Листинг 26. Пример.

```
string s1 = "Арбуз";
string s2 = "Дыня";

int result = String.Compare(s1, s2);
Console.WriteLine(result);
//результат: -1
```

Contains - определяет, содержится ли подстрока в строке.

Листинг 27. Пример.

```
string s1 = "Арбуз";
string s2 = "бу";
bool result = s1.Contains(s2);
Console.WriteLine(result);
//результат: True
```

StartWith/EndsWith – методы, определяющие, совпадает ли начало/конец строки с подстрокой.

Листинг 28. Пример.

```
string path = @"C:\ExampleDir";
string[] files = Directory.GetFiles(path);
for (int i = 0; i < files.Length; i++)
{
   if(files[i].EndsWith(".exe"))
      File.Delete(files[i]);
}
```

В примере производится поиск и удаление всех исполняемых («.exe») файлов в директории. Специальный символ @ - буквальный идентификатор, позволяет избежать интерпретации символа обратный слеш как начала escape-последовательности.

IndexOf/LastIndexOf - находят индекс первого/последнего вхождения символа или подстроки в строке.

Insert - вставляет подстроку в строку.

Листинг 29. Пример.

```
string text = "Разработка в Studio";
string subString = "Visual ";

text = text.Insert(11, subString);
Console.WriteLine(text);
//Результат: Разработка в Visual Studio
```

Join – **статический метод**, создает строку, соединяя элементы массива строк.

Листинг 30. Пример.

```
string[] arr = { "один", "два", "три" };
Console.WriteLine(String.Join(",", arr));
//Результат: один,два,три
```

Replace – заменяет символ или подстроку в строке другим символом или подстрокой.

Листинг 31. Пример.

```
string text = "пасмурный день";
text = text.Replace("пасмурный ", "солнечный");
Console.WriteLine(text);
text = text.Replace("е", "");
Console.WriteLine(text);
```

Split – позволяет разделить строку на массив подстрок.

Листинг 32. Пример.

```
string text = "один два три четыре";
string[] words = text.Split(new char[] { ' ' });
foreach (string s in words)
{
   Console.WriteLine(s);
}
```

Substring – извлечение подстроки с указанной позиции, указанной длины.

Листинг 33. Пример.

```
string text = "один два три четыре";
text = text.Substring(text.IndexOf(' ')+1, (text.Length- text.IndexOf(' ')-1));
Console.WriteLine(text);
//Результат: два три четыре
```

ToLower/ToUpper – функции переводят все символы строки в верхний/нижний регистр.

Trim - функция удаляет начальные и конечные пробелы из строки. Имеет частичные аналоги TrimStart и TrimEnd.

При работе со строками следует помнить, что в C# строки являются объектами, т.е. ссылочным типом. При этом

объекты типа **string** не подлежат изменению. Т.е., однажды созданная строка не редактируется. Использование методов редактирования строк приводит к созданию в управляемой куче нового строчного объекта, при этом ссылка типа string переопределяется на него. Старый объект становится не используемым и удаляется сборщиком мусора.

Можно говорить о недостаточной эффективности такого подхода, так как любая модификация объекта типа **string** приводит к необходимости его копирования в новые ячейки памяти.

Класс **StingBuilder**, присутствующий в пространстве имен System.Text позволяет повысить эффективность работы со строками.

Класс **System.String** выделяет для хранения объекта ровно столько памяти, сколько необходимо для хранения строки существующей длины. Поэтому модификация строки приведет к копированию ее в новые ячейки памяти.

Отличие класса **StingBuilder** в том, что он позволяет выделить для хранения объекта блок памяти «с запасом», указанным программистом, и любые модификации строки происходят внутри блока памяти, выделенного экземпляру **StringBuilder.**

В дополнение к избыточной памяти, выделяемой изначально, StringBuilder имеет свойство *удваивать* свою емкость, когда происходит переполнение.

Класс **StringBuilder** имеет следующие свойства:

- Length - длина строки, содержащейся в объекте в данный момент;

- Capacity - максимальная длина строки, которая может поместиться в выделенную для объекта память,

и поддерживает следующие методы:

- Append() - добавляет строку к текущей строке;

- AppendFormat() - добавляет строку, сформированную в соответствии со спецификатором формата;

- Insert() - вставляет подстроку в строку;

- Remove() - удаляет символ из текущей строки;

- Replace() - заменяет все вхождения символа другим символом или вхождения подстроки другой подстрокой;

- ToString() - возвращает текущую строку в виде объекта System.String.

Листинг 34. Пример использования класса StringBuilder.

```
StringBuilder sb1 = new StringBuilder("Это объект StringBuilder ", 50);
        Console.WriteLine(sb1);
        Console.WriteLine($"Длина строки в sb1 {sb1.Length}");
        Console.WriteLine($"Максимальная допустимая длина  sb1 
{sb1.Capacity}");
        Console.WriteLine($"Можно добавить символов в sb1 
{sb1.Capacity-sb1.Length}");
        sb1.AppendFormat("после использования AppendFormat.");
        Console.WriteLine(sb1);
        Console.WriteLine($"Длина строки в sb1 {sb1.Length}");
        Console.WriteLine($"Максимальная допустимая длина  sb1 
{sb1.Capacity}");
        Console.WriteLine($"Можно добавить символов в sb1 {sb1.Capacity 
- sb1.Length}");
        Console.ReadLine();
```

```
Это объект StringBuilder
Длина строки в sb1 25
Максимальная допустимая длина  sb1 50
Можно добавить символов в sb1 25
Это объект StringBuilder после использования AppendFormat.
Длина строки в sb1 58
Максимальная допустимая длина  sb1 100
Можно добавить символов в sb1 42
```

Как видно из результатов работы программы, после превышения первоначальной длины (50 символов), длина объекта была удвоена.

3. ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ МЕТОДОЛОГИЯ В C#

C# является полноценным объектно-ориентированным языком. По определению Гради Буча (американский специалист в области программной инженерии, один из основоположников объектно-ориентированной методологии разработки программ): "Объектно-ориентированное программирование (ООП) – это методология программирования, которая основана на **представлении программы в виде совокупности объектов**, каждый из которых является реализацией определенного класса (типа особого вида), а классы образуют иерархию на принципах наследуемости".

В объектно-ориентированной декомпозиции **разделение задачи идет не по алгоритмам, а по объектам**, использующимся в процессе решения задачи. Каждый выделяемый **объект** предметной области **отвечает за выполнение** некоторых **действий**, **зависящих от полученных сообщений и параметров самого объекта**.

Все языки ООП должны предоставлять разработчикам следующие механизмы:

Инкапсуляция – механизм, позволяющий объединить код и данные, которыми он манипулирует, что исключает вмешательство извне и неправильное использование данных. Элементом, поддерживающим **инкапсуляцию** является **объект,** а инструментом реализации инкапсуляции - **класс**.

Наследование — способность языка строить новые определения классов на основе определений существующих классов с наследованием основной функциональности в производных подклассах.

Полиморфизм - способность языка трактовать связанные объекты в сходной манере. В частности, этот принцип ООП позволяет базовому классу определять набор

членов (формально называемый **полиморфным интерфейсом**), которые доступны всем наследникам.

Ниже будут рассмотрены аспекты реализации механизмов ООП в C#.

3.1. Инкапсуляция. Классы. Содержимое классов

Класс – основное средство реализации механизма инкапсуляции в C#.

Кл**асс** представляет собой **шаблон**, определяющий форму **объекта**, и может содержать элементы двух типов:

- данные-члены;

- функции-члены.

Данные-члены содержат собственно данные, а **функции-члены** содержат код, оперирующий этими данными.

Рассмотрим подробнее элементы (члены) класса.

Данные- члены:

Поля (field) – переменные, ассоциированные с классом.

Константы – неизменяемые члены класса, могут быть ассоциированы с классом так же как переменные. Объявляются с помощью ключевого слова **const**.

События – определяют уведомления, которые может генерировать класс. Клиент, использующий класс может определять код, реагирующий на события и называемый **обработчик события.**

Функции-члены.

Методы - функции, определенные для данного класса.

Свойства – функции, определяющие характеристики класса. Клиент имеет к свойствам доступ, как к полям класса.

Конструкторы – функции, автоматически вызываемые при инициализации объекта. Имя конструктора совпадает с именем класса.

Финализаторы – функции, вызываемые средой CLR, для уничтожения объекта, когда он становится не нужным. Имя финализатора совпадает с именем класса, но начинается с «~».

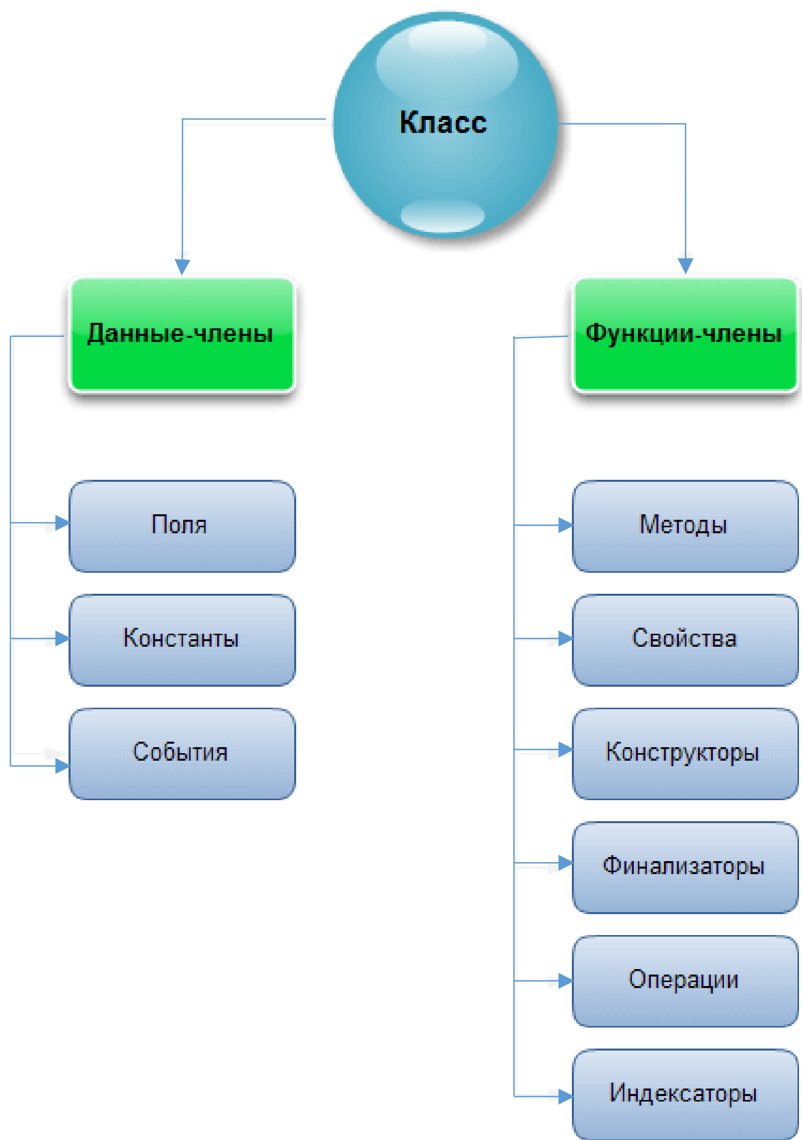


Рис. 7. Структура класса

Операции – простейшие действия, типа «+», «-», переопределенные (перегруженные) пользователем для работы с экземплярами класса.

Индексаторы – обеспечивают доступ к объектам как к элементам массива.

Листинг 35. Форма для создания простого метода.

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

        // Объявление методов.
        доступ возращаемый_тип метод1 (параметры)
        {
                // тело метода
        }
        доступ возращаемый_тип метод2 (параметры)
        {
                // тело метода
        }
        //. . .
        доступ возращаемый_тип методN (параметры)
        {
                // тело метода
        }
}
```

Рассмотрим пример создания и использования простого класса.

Листинг 36.

```
class Tovar //создание класса
    {
        //поля класса
        public string product;
        public int amount;
        public float price;
        //метод класса
        public void ViewProduct()
        {
            Console.WriteLine("Наименование товара:" + product);
            Console.WriteLine("Количество   товара:" + amount);
            Console.WriteLine("Цена         товара:" + price);
            Console.WriteLine("Стоимость    товара:" + amount*price);
            Console.ReadLine();
        }
    }
    static void Main(string[] args)
    {
        Tovar t1 =new Tovar(); //создание экземпляра класса
        t1.product = "Картошка"; //заполнение полей
        t1.amount = 500;
        t1.price = 30;
        t1.ViewProduct(); //вызов метода
    }
}
```

```
Наименование товара:Картошка
Количество   товара:500
Цена         товара:30
Стоимость    товара:15000
```

Рассмотрим наиболее важные вопросы создания экземпляров класса.

Методы

Привычные для процедурных языков программирования процедуры и функции в C# существуют в виде методов класса.

Метод – именованный блок кода, осуществляющий некоторую функциональность. По умолчанию, консольная

программа на C# должна содержать метод **Main,** являющийся точкой входа в приложение.

Листинг 37. Пример.

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

Общее определение метода выглядит следующим образом.

Листинг 38. Пример.

```
[модификаторы] тип_возвращаемого_значения название_метода
([параметры])
{
   // тело метода
}
```

Модификаторы доступа позволяют задать допустимую область видимости для метода или иного члена класса.

При определении метода обычно используются следующие модификаторы доступа:

- public - метод, доступен из **любого места в коде**, а также из других программ и сборок.

- private - закрытый метод, доступен только из кода в том же классе или контексте.

- protected - доступен из любого места в **текущем классе или в производных классах**, которые могут располагаться в других сборках.

- internal - доступны из любого **места кода в той же сборке**, однако он недоступен для других программ и сборок (как в случае с модификатором public).

Помимо перечисленных модификаторов доступа, могут создаваться статические (**static**) методы, которые относятся ко всему классу и для обращения к ним не обязательно создавать экземпляр класса (см. выше статические методы класса string).

Модификаторы и параметры в определении метода могут отсутствовать.

Использование параметров. Ключевые слова ref и out

При вызове метода ему могут передаваться один или несколько параметров. Параметры объявляются как переменные, например.

Листинг 39. Пример.

```
public double summa (double a, double b)
    {
      double sum1 = a + b;
      return sum1;
    }
```

Параметры, как и прочие переменные, могут быть значимого или ссылочного типа. Если параметр передается по ссылке, то метод получает саму переменную и ее изменения в теле метода сохранятся после его завершения. При передаче значимого параметра, метод получает копию переменной и все изменения, выполненные в теле метода будут утеряны по его окончании. Так, если передать методу в качестве параметров массив и целое число, выполнить изменение параметров в теле функции, то изменения массива сохранятся, а целое останется неизменным. Строки (string), хотя и являются ссылочным типом, ведут себя как значимый тип, т.е. изменения в строке не сохранятся.

Однако, если при описании метода, перед типом параметра поставить ключевое слово **ref**, то при вызове метода в качестве параметра будет передаваться ссылка, вне зависимости, значимый или ссылочный тип имеет параметр. В этом случае изменение параметра внутри метода приведет к изменению его значения вне метода.

Аналогично работает ключевое слово **out.** Разница между **ref и out** в том, что **ref** предполагает использование только инициализированных переменных, а **out** позволяет передавать ссылку на неинициализированную переменную.

Если определены значения параметров по умолчанию, то они могут не передаваться при вызове метода.

Листинг 40. Пример.

```
static int mySum(int a, int b = 5, int c = 10)
```

Вызов этой функции можно осуществлять без передачи параметров b и c. Они будут приняты равными 5 и 10 соответственно.

Использование **именованных параметров**, когда при передаче фактических параметров вызываемому методу, параметры именуются, позволяет не соблюдать порядок их следования, заданный в описании метода. Для вышеприведенного примера вызов метода может выглядеть следующим образом.

Листинг 41. Вызов метода.

```
int sum1=mySum(a: 3, c: 6, b: 2);
```

Тип возвращаемого значения должен соответствовать типу переменной, указанной в операторе **return.** Ключевое слово **void,** используемое в качестве типа возвращаемого значения при описании метода**,** указывает на то, что метод ничего не возвращает. В этом случае оператор **return** используется без переменной и производит выход из метода.

В C# возможно использование ключевого слова **this** для обеспечения доступа к текущему экземпляру класса. Это может понадобиться во избежание неоднозначности контекста в том случае, например, когда имя входящего параметра совпадает с именем поля.

Конструкторы

Конструктор – метод, имеющий имя, совпадающее с именем класса и выполняющий инициализацию объекта при его создании.

Листинг 42.

```
class MyClass
   {
      public string Name;
      public byte Age;

      // Создаем параметрический конструктор
      public MyClass(string s, byte b)
      {
         Name = s;
         Age = b;
      }
   }
static void Main(string[] args)
      {
         //создание экземпляра с использованием    //конструктора
           MyClass ex1  = new MyClass("Alexandr", 26);
      }
```

Деструкторы

Деструктор – метод, вызываемый непосредственно перед уничтожением неиспользуемого объекта системой, происходящего при «уборке мусора». Имя деструктора совпадает с именем класса, но начинается со знака «~».

Индексаторы

C# предоставляет возможность создания специальных классов, которые могут быть индексированы подобно массивам при помощи **индексатора.**

Индексаторы могут быть одно- или многомерными.

Листинг 43. Общая форма одномерного индексатора

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

Свойства

Свойство – как правило, сочетает в себе поле и метод доступа к нему. Состоит из имени и аксессоров (методов доступа) get и set.

Листинг 44.

```
тип имя {
  get
  {
     // код аксессора для чтения из поля
  }

  set
  {
     // код аксессора для записи в поле
  }
}
```

После того, как свойство определено, любое обращение к нему по имени приводит к вызову соответствующего аксессора. Аксессор **set** принимает неявный параметр **value,** содержащий значение, присваиваемое свойству.

Свойства не предназначены для создания полей, а лишь обеспечивают доступ к существующим полям.

3.2. Доступ к членам класса

Инкапсуляция, помимо того, что связывает данные с кодом, еще и предоставляет возможность **управлять доступом** к членам класса.

Контроль доступа к членам класса является одной из основных возможностей ООП, т.к. позволяет исключить неверное использование объектов.

Для управления доступом в C# используются, как отмечалось выше, четыре модификатора доступа (**public, private, protected и internal**).

При организации **открытого** или **закрытого доступа** необходимо придерживаться следующих принципов:

- члены, используемые только внутри класса, должны быть закрытыми;
- данные экземпляра, имеющие ограниченный перечень значений, должны быть закрытыми;
- если изменение члена оказывает влияние на другие члены объекта, он должен быть закрытым;
- методы, изменяющие значения закрытых членов, должны быть отрытыми.

Класс object

Все классы C#, включая классы библиотеки .NET, классы созданные программистами на платформе .NET, имеют один базовый класс — класс **object**. Это дает возможность переменной ссылочного типа **object** ссылаться на объект любого типа. На практике это значит, что для объекта любого класса программист, помимо созданных им методов и свойств, может использовать также методы и свойства класса **object:**

ToString() - возвращает символьную строку, с описанием объекта;

Equals() – позволяет определить ссылочную эквивалетентность объектов;

GetType() – позволяет получить детальные данные о классе, членом которого является ваш объект, включая базовый тип, методы, свойства и т.п.

Clone() - создает копию объекта и возвращает ссылку на эту копию.

3.3. Наследование

Наследование – следующий за инкапсуляцией главный принцип ООП. Наследование позволяет создать общий класс, определяющий особенности множества взаимосвязанных элементов, и на базе этого **(базового)** класса создавать новые

(производные) классы, определяющие индивидуальные характеристики, подмножеств исходного множества. Такой подход удобен для создания иерархических классификаций.

В C# наследование реализовано в виде возможности объявления нового класса внутри существующего.

Листинг 45. Пример.

```
class myProducts //создание базового класса Товары
{
   //определение полей и методов базового класса
   public string name;          //наименование
   public string manufacturer;  //производитель
   public float price;          //цена
   public void PrWrite()
   {
      Console.WriteLine("Наименование : " + name);
      Console.WriteLine("Производитель: " +
manufacturer);
   }

}
// Объявляем класс, унаследованный от класса Товары
class myLamp : myProducts        //класс Лампы
{
   public string lbase;         //цоколь
```

```
      public int power;          //мощность

         // Поля класса myProducts доступны через
         //конструктор наследуемого класса
         public myLamp(string name, string manufacturer, float
         price, string plbase, int ppower)
         {
                  this.name = name;
                  this.manufacturer = manufacturer;
                  this.price = price;
                  this.lbase = plbase;
                  this.power = ppower;
         |
         }
         public void LmpWrite()
         {

                  Console.WriteLine("Цоколь      : " + lbase);
                  Console.WriteLine("Мощность    : " + power);
                  Console.WriteLine("Цена        : " + price);
                  Console.ReadLine();
         }
   }
class Program
{
   static void Main(string[] args)
         {
         myLamp lamp1 = new myLamp("Лампа
         накаливания", "ООО ЭП",123.50f, "E27",75);
         lamp1.PrWrite();
         lamp1.LmpWrite();
         }
}
```

В данном примере создается класс «**myProducts**», для описания объектов Товары, имеющий поля Наименование,

Производитель и Цена, а также метод, позволяющий выводить эти поля на экран. Затем создается подкласс Лампы, имеющий поля, характерные только для ламп: цоколь и мощность, а также метод для вывода этих полей на экран.

Создается экземпляр класса Лампы и осуществляется вывод всех его полей на экран. Можно увидеть, что объект класса лампы имеет одинаковый доступ к полям и методам классов Товары и Лампы.

Однако, если поле или метод базового класса имеет модификатор доступа **private**, то доступ к нему из производного класса будет запрещен. Производный класс имеет доступ к членам базового класса, определенных как **public**, **internal**, **protected** и **protected internal**.

В примере выше конструктор определен только для производного класса. Если в базовом классе имеется конструктор, то конструктор производного класса определяется с использованием ключевого слова **base**.

Листинг 46.Пример.

```
public ClassA(int point, int x, int y, int z)
        : base(x, y, z)
```

В данном примере параметры x, y, z, определены в конструкторе базового класса.

3.4. Полиморфизм

Полиморфизм является третьим столпом ООП, наряду с инкапсуляцией и наследованием.

Владея основными понятиями ООП можно сказать, что полиформизм позволяет организовать различные способы представления и алгоритмы работы для одноименных членов базового класса и производных подклассов. В конечном итоге это позволяет сделать повторное использование кода более эффективным, а сам код более универсальным и легко читаемым.

Для реализации полиформизма в C# используются такие механизмы как сокрытие имен, виртуальные и абстрактные методы, переопределение методов, интерфейсы.

3.5. Наследование и сокрытие имен

Если в производном классе определяется член с именем, имеющимся в базовом классе, компилятор выдаст предупреждение, но программа будет скомпилирована. При этом, в производном классе будет использоваться член, определенный в нем. Этот механизм дает возможность скрывать члены базового класса, но при этом в описании одноименного члена класса, перед его именем следует указать ключевое слово **new**, как это делается при создании объекта.

Если член базового класса сокрыт вышеуказанным образом, но к нему необходимо обратиться, это возможно с использованием ключевого слова **base**, обеспечивающего доступ к элементам базового класса также, как **this** обеспечивает доступ к элементам текущего.

Виртуальные, свойства и индексаторы

Принцип полиморфизма предоставляет подклассу производить переопределение методов, свойств и индексаторов, определенных в базовом классе. Для этого используются ключевые слова **virtual, и override**.

Для того, чтобы метод (свойство, индексатор) мог быть переопределен в подклассе, при его создании следует объявить его как виртуальный, с использованием ключевого слова **virtual.**

Листинг 47. Пример.

```
public virtual string FontInfo(string str)
```

В подклассе такой метод можно переопределить, используя ключевое слово **override.**

Листинг 48. Пример.

```
public override string FontInfo(string str)
```

На основе этой возможности реализован один из самых эффективных в C# принципов - ***динамическая диспетчеризация методов***, заключающийся в определении вызываемого метода во время выполнения, а не компиляции. Это делает возможным реализацию в C# **динамического полиморфизма.**

Рассмотрим пример использования виртуального метода.

Листинг 49. Пример.

```
class Animal  // объявляем класс животное
{
    public string NickName;              // поле - кличка
    public Animal(string NickName) // конструктор
    { this.NickName = NickName; }
    public virtual void  service()       //виртуальный метод
    { Console.WriteLine(NickName + " это животное."); }
}
class Dog : Animal // объявляем класс собака как
                          //наследник класса животное
{
    //конструктор класса Собака
    public Dog(string NickName) : base(NickName) { }
    //переопределение виртуального метода service
    public override void service()
    { Console.WriteLine(NickName + " охраняет дом"); }
}
class Cat : Animal // объявляем класс кошка как наследник
класса животное
{
    public Cat(string NickName) : base(NickName) { }// задаем
прозвище в конструкторе
```

```
        //переопределение виртуального метода service
        public override void service()
        { Console.WriteLine(NickName + " ловит мышей"); }
    }
    class Cow : Animal // объявляем класс корова как наследник
класса животное
    {
        public Cow(string NickName) : base(NickName) { }//
задаем прозвище в конструкторе
        //переопределение виртуального метода service
        public override void service()
        { Console.WriteLine(NickName + " даёт молоко"); }
    }
    class Program
    {
        static void Main(string[] args)
        {

            //создаем список объектов Животное
            List<Animal> animals = new List<Animal>();
            //добавляем новые экземпляры подклассов к списку
            animals.Add(new Dog("Барбос"));
            animals.Add(new Cat("Барсик"));
            animals.Add(new Dog("Полкан"));
            animals.Add(new Cow("Буренка"));

            //вызываем метод service для каждого экземпл.списка
            foreach (Animal animal in animals)
            {
                animal.service();
            }
            Console.ReadKey();
        }
```

```
}
```

```
Барбос охраняет дом
Барсик ловит мышей
Полкан охраняет дом
Буренка даёт молоко
```

Рис. 8. Результат работы программы

3.6. Абстрактные методы

Если **виртуальный** (virtual) метод – метод который **МОЖЕТ** быть переопределен в подклассе, то **абстрактный** (**abstract**) метод **ДОЛЖЕН** быть переопределен в подклассе.

Абстрактный метод в базовом классе **не имеет тела** и описывается в следующем виде.

Листинг 50.

```
abstract тип имя(список_параметров)
```

Модификатор **abstract** не может применяться статических методах (**static**). Абстрактными могут быть также индексаторы и свойства.

Класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный, и для этого перед его объявлением **class** указывается модификатор **abstract**.

При наследовании абстрактного класса в подклассе ДОЛЖНЫ быть реализованы все абстрактные методы базового класса.

При переопределении абстрактного метода используется ключевое слово **override**, как и в случае с виртуальным классом.

3.7. Интерфейсы. Множественное наследование

Интерфейс – набор описаний абстрактных членов, тела которых будут созданы в классе, реализующем интерфейс.

В интерфейсе содержатся только **сигнатуры-** имена и типы параметров таких членов как: **методы, свойства, события и индексаторы.**

Интерфейс **не может** содержать: **конструкторы, поля, константы, статические члены.**

Интерфейс объявляется за пределами класса, т.к. является структурной единицей уровня класса. Имя интерфейса должно начинаться с префикса «I».

Листинг 51. Пример.

```
interface ISomeInterface
{
  // тело интерфейса
  string SomeProperty { get; set; } // сигнатура свойства
  void SomeMethod(int a); // метод
}
```

В сигнатурах членов интерфейса модификатор доступа не указывается.

При создании класса, реализующего интерфейс, имя интерфейса указывается после имени класса и двоеточия. При этом, класс, реализующий интерфейс, **должен реализовать все члены** интерфейса.

Листинг 52. Пример.

```
class SomeClass : ISomeInterface
{
  public string SomeProperty
  {
    get
    {
      // тело get аксессора
    }
    set
    {
      // тело set аксессора
    }
  }
```

```
interface IGeometrical // объявление интерфейса
{
  void GetPerimeter();
  void GetArea ();
}
class Rectangle : IGeometrical //реализация интерфейса
{
  public void GetPerimeter()
  {
   Console.WriteLine("(a+b)*2");
  }

  public void GetArea()
  {
   Console.WriteLine("a*b");
  }
}
class Circle : IGeometrical //реализация интерфейса
{
  public void GetPerimeter()
  {
   Console.WriteLine("2*pi*r");
  }

  public void GetArea()
  {
   Console.WriteLine("pi*r^2");
  }
}
class Program
{
  static void Main(string[] args)
  {
   List<IGeometrical> figures = new List<IGeometrical>();
   figures.Add(new Rectangle());
```

Листинг 53. Пример использования интерфейса

```
figures.Add(new Circle());
foreach (IGeometrical f in figures)
{
 f.GetPerimeter();
 f.GetArea();
 if (f is IGeometrical) //проверка поддержки интерфейса
     Console.WriteLine("Объект f поддерживает
     интерфейс IGeometrical");
}
Console.ReadLine();
}
}
```

Для проверки, поддерживает ли объект какой-либо интерфейс, используется ключевое слово **is** (см. выше).

Как можно отметить, по идеологии использование интерфейсов очень напоминает использование абстрактных классов. В обоих случаях родитель содержит набор абстрактных членов, которые реализуются в потомке (производном классе).

Основной особенностью интерфейсов является то, что они, в отличие от классов, позволяют реализацию **множественного наследования**. Это значит, что класс может реализовать сразу несколько интерфейсов, но не может быть наследником нескольких классов, не имеющих одного базового класса.

Если класс реализует несколько интерфейсов, они разделяются запятыми.

Листинг 54. Пример.

```
interface IDrawable //первый интерфейс
{
  void Draw();
}

interface IGeometrical //второй интерфейс
{
  void GetPerimeter();
  void GetArea ();
}
class Rectangle : IGeometrical, IDrawable //реализация двух
                                          //интерфейсов
{
  public void GetPerimeter()
  {
    Console.WriteLine("(a+b)*2");
  }

  public void GetArea()
  {
    Console.WriteLine("a*b");
  }

  public void Draw()
  {
    Console.WriteLine("Rectangle");
  }
}
```

Помимо этого, интерфейс может наследовать другой интерфейс или несколько интерфейсов, что позволяет расширить функциональность существующего интерфейса без внесения изменений в уже существующий код.

Если при реализации нескольких интерфейсов или при множественном наследовании интерфейсов возникает

конфликт имен (т.е. имеются члены с идентичными именами), может использоваться **явная реализация члена** интерфейса – т.е. указание имени члена через точку после имени интерфейса.

3.8.Делегаты. События

Делегаты – еще один базовый тип CTS (**Common type System**) наравне с классами, интерфейсами, списками, перечислениями.

Делегат – объект, создаваемый, чтобы ссылаться на метод (методы).

Делегат содержит следующие информационные блоки:

- адрес метода;
- аргументы (параметры) метода;
- возвращаемое значение метода.

Листинг 55.Форма объявления делегата.

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

Делегат может служить для вызова ЛЮБОГО метода, имеющего список параметров и возвращаемый тип, такие же, как в описании делегата.

Делегат может ссылаться как на статические методы, так и на методы экземпляров (в этом случае требуется ссылка на объект).

Одним из важных свойств делегата является возможность создания списка методов, вызываемых автоматически при вызове делегата (**групповая адресация**).

Листинг 56. Пример.

```
namespace ConsoleApplication6
{
   // Создадим делегат
   delegate int IntOperation(int  i, int j);
   class Program
   {
      // Создадим ряд методов
```

```
static int Sum(int x, int y)
{
   Console.WriteLine("Сумма: " + (x + y));
   return x + y;
}
static int Prz(int x, int y)
{
   Console.WriteLine("Произведение: " + (x * y));
   return x * y;
}
static int Del(int x, int y)
{
   Console.WriteLine("Частное: " + (x / y));
   return x / y;
}
static void Main()
{
   // Создаем делегаты разными способами
   IntOperation op1, op_group;
   op1 = Sum;
   IntOperation op2 = new IntOperation(Prz);
   IntOperation op3 = new IntOperation(Del);

   //вызываем делегаты
   int result = op1(10, 5);
   result = op2(10, 5);
   result = op3(10, 5);
   Console.ReadLine();
   //создаем делегат с групповой адресацией
   op_group = op1;
   op_group += op2;
   op_group += op3;
   result = op_group(10, 5);
   Console.WriteLine("Значение result:" + result);
   Console.ReadLine();
```

```
        }
    }
}
```

```
Сумма: 15
Произведение: 50
Частное: 2

Сумма: 15
Произведение: 50
Частное: 2
Значение result:2
```

Цепочки вызовов имеют особое значение для обработки событий.

Как видно из примера выше, переменная, принимающая значение от делегата, имеет значение, равное результату выполнения последнего метода в цепочке.

Повысить гибкость использования делегатов позволяют свойства **ковариантности и контрвариантности.**

Ковариантность – позволяет использовать в делегате метод, возвращающий тип, **производный** от класса возвращаемого значения делегата.

Контрвариантность – позволяет использовать в делегате метод, возвращающий тип, **базовый** для класса возвращаемого значения делегата.

Эти свойства также широко используются при обработке событий.

Обобщенные делегаты Action<T> и Func<T>, Predicate

Это, так называемые, **обобщенные** делегаты, которые позволяют избежать создания новых типов делегатов для разных параметров.

Делегат **Action<T>** предназначен для методов с возвращаемым типом **void** и позволяет передавать от 0 до 16 разных типов параметров.

Делегат **Func<T>** аналогичен **Action<T>,** но предназначен для методов, имеющих возвращаемый параметр.

Листинг 57. Пример.

```
TResult Func<out TResult> () // без вх. параметров
TResult Func<in T, out TResult>(T arg) //с 1 вх. параметром
TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2) // с
2 вх. параметрами

TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1,
T2 arg2, T3 arg3, T4 arg4) // с 4 вх. Параметрами.
```

Эти делегаты, как правило, передаются методу в качестве параметра и предусматривают выполнение некоторых действий в ответ на произошедшие действия (события).

Листинг 58. Пример.

```
static void Main(string[] args)
{
   Action<int, int> op; //создали обобщенный делегат op
   op = Add;              //передали делегату метод Add
   Operation(10, 6, op); //делегат- параметр
   op = Substract;        //передали делегату метод Substract
   Operation(10, 6, op); //делегат- параметр

   Console.Read();
}

static void Operation(int x1, int x2, Action<int, int> op)
{
   if (x1 > x2)
      op(x1, x2);
}

static void Add(int x1, int x2)
{
   Console.WriteLine("Сумма чисел: " + (x1 + x2));
}

static void Substract(int x1, int x2)
{
   Console.WriteLine("Разность чисел: " + (x1 - x2));
}
```

Делегат **Predicate<T>**, используется для сравнения объекта T определенному условию. В качестве выходного результата возвращается значение **true**, если условие соблюдено, и **false,** если не соблюдено.

Листинг 59. Пример.

```
Predicate<int> isPositive = delegate (int x) { return x > 0; };

Console.WriteLine(isPositive(20));
Console.WriteLine(isPositive(-20));
```

3.9. Анонимные функции. Лямбда-выражения

Если метод создается только для вызова в делегате, то можно не создавать метод, а использовать **анонимный метод** – безымянный блок кода, передаваемый конструктору экземпляра делегата.

Лямбда-выражение- улучшенная, с точки зрения синтаксиса, альтернатива анонимного метода.

Лямбда-выражение может использоваться везде, где есть параметр типа делегата.

Любое **лямбда-выражение** содержит оператор **«=>» - переходит или становится.**

Слева от оператора => указывается набор входных параметров, а справа- тело **лямбда-выражения.**

Листинг 60. Пример.

```
class Program
{
    // Создадим делегат для проверки ввода пароля при
    // регистрации
    delegate bool BoolPassword(string s1, string s2);

  static void Main()
  {
    Console.Write("Введите пароль: ");
    string password1 = Console.ReadLine();
    Console.Write("Повторите пароль: ");
    string password2 = Console.ReadLine();

    // Используем лямбда выражение
    BoolPassword bp = (s1, s2) => s1 == s2;

    if (bp(password1, password2))
      Console.WriteLine("Регистрация удалась!");
    else
      Console.WriteLine("Регистрация провалилась.
Пароли не совпадают");

    Console.ReadLine();
  }
}
```

Выше приведен пример простого лямбда-выражения.

При использовании **блочного лямбда-выражения вышеприведенный пример может выглядеть следующим образом.**

Листинг 61.

```
class Program
  {
    // Создадим делегат для проверки ввода пароля при
    // регистрации
    delegate bool BoolPassword(string s1, string s2);

    static void Main()
    {
      Console.Write("Введите пароль: ");
      string password1 = Console.ReadLine();
      Console.Write("Повторите пароль: ");
      string password2 = Console.ReadLine();

      // Используем лямбда выражение
      BoolPassword bp = (s1, s2) =>
      {
        if (s1 == s2)
        {
          Console.WriteLine("Регистрация удалась!");
          Console.ReadLine();
          return true;
        }
        else
        {
          Console.WriteLine("Регистрация провалилась.
Пароли не совпадают");
          Console.ReadLine();
           return false;
        }

      };
      bp(password1, password2); //вызываем делегат
  }
```

3.10. События

В ООП, **событие** – это автоматически формируемое объектом уведомление, что над ним выполнены некоторые действия. При этом, внешние объекты, проявляющие интересы к событию, регистрируют для события методы, называемые **обработчиками события.**

Лучшей иллюстрацией событий являются события визуальных элементов форм. Например, при нажатии левой кнопки мыши в области экрана, занятой визуальным элементом «кнопка», формируется событие Click, которое должно вызвать метод, зарегистрированный обработчиком события.

Другой пример: обработчик события может отсылать SMS на телефон абонента, при попытке входа в личный кабинет.

Событие – член класса, в определении которого используется ключевое слова **evetn**.

Листинг 62. Пример.

```
event делегат_события имя_события;
```

События тесно связаны с делегатами и осуществляют с их помощью механизм публикации.

События, как и делегаты, поддерживают **групповую адресацию**, что позволяет формировать набор обработчиков событий.

Листинг 63. Пример.

```
delegate void UI (); //объявили делегат события

class MyEvent
{
   // Объявляем событие
   public event UI UserEvent;

   // Используем метод для запуска события
   public void OnUserEvent()
   {
      UserEvent();
   }
}

class UserInfo
{
   string uiName, uiFamily;
   int uiAge;

   public UserInfo(string Name, string Family, int Age)
   {
      this.Name = Name;
      this.Family = Family;
      this.Age = Age;
   }

      public string Name { set { uiName = value; } get { return
uiName; } }
      public string Family { set { uiFamily = value; } get { return
uiFamily; } }
      public int Age { set { uiAge = value; } get { return uiAge; }
}

      // Обработчик события
      public void UserInfoHandler()
```

```
        public void UserInfoHandler()
        {
            Console.WriteLine("Событие вызвано!\n");
            Console.WriteLine("Имя: {0}\nФамилия:
{1}\nВозраст: {2}",Name,Family,Age);
        }
    }

    class Program
    {
        static void Main()
        {
            MyEvent evt = new MyEvent();
            UserInfo user1 = new UserInfo(Name: "Alex", Family:
"Erohin", Age: 26);

            // Добавляем обработчик события
            evt.UserEvent += user1.UserInfoHandler;

            // Запустим событие
            evt.OnUserEvent();

            Console.ReadLine();
        }
    }
```

Для добавления или удаления обработчиков событий используются аксессоры add и remove.

Листинг 64. Использование аксессоров add и remove.

```
event делегат_события имя_события {
  add {
  // Код добавления события в цепочку событий
  }
  remove {
  // Код удаления события из цепочки событий
  }
}
```

Вызов этих аксессоров осуществляется при добавлении или удалении обработчиков события с помощью операторов += (см. выше) и -=.

C# не накладывает ограничений на формат создаваемых событий. Однако для обеспечения совместимости с .NET Framework, следует придерживаться требования, что обработчик должен иметь 2 параметра.

Листинг 65. Пример обработчика.

```
void обработчик(object отправитель, EventArgs ea)
{
  //...
}
```

Здесь object – ссылка на объект, формирующий событие, а ea – параметр типа EventArgs. Класс EventArgs не содержит полей и используется в качестве базового для создания структур, содержащих необходимые данные для передачи обработчику.

3.11. Структуры

Структуры – один из 5 базовых типов CTS (**Common type System**) наравне с классами, интерфейсами, делегатами, перечислениями.

По функциональным возможностям структуры полностью идентичны классам, но имеют от них одно очень важной отличие: **структуры** представляют собой **значимый** тип, а **классы – ссылочный**. Это обозначает, что экземпляры классов – объекты, хранятся в куче и для доступа к ним всегда требуется распаковка и упаковка. Экземпляры структур представляют собой значения, хранятся в стеке программы, и при обращении не требуют распаковки, что может дать прирост производительности.

При этом структуры не наследуются, и при присвоении другим структурам копируются.

В связи с вышесказанным, можно сделать вывод, что структуры следует применять для хранения небольших типов

данных, доступ к которым нужно сделать максимально быстрым.

Работа со структурами схожа с работой с классами, но при описании структур используется ключевое слово **struct.** Как и классы, структуры могут содержать поля, методы, конструкторы, индексаторы, свойства, события. Для создания нового экземпляра структуры используется ключевое слово **new**.

Листинг 66. Пример использования структур.

```
struct User  //определение структуры
  {
     //поля структуры
     public string name;
     public int age;
     //конструктор
     public User(string name, int age)
     {
        this.name = name;
        this.age = age;
     }
     //метод
     public void DisplayInfo()
     {
        Console.WriteLine($"Name: {name}  Age: {age}");
     }
  }

  class Program
  {
     static void Main(string[] args)
     {
        //создание экземпляра структуры
        User tom = new User("Tom", 34);
        tom.DisplayInfo();

        User bob = new User();
        bob.DisplayInfo();

        Console.ReadKey();
     }
  }
```

3.12. Обобщения. Краткие сведения

Термин обобщение в ООП обозначает, по сути, **параметризованный тип**. Т.е. тип данных в классе, структуре, интерфейсе, методе или делегате не определен заранее, определяется только при их инициализации. Такой подход позволяет, например, создать единый класс, пригодный для обработки данных любого типа. Инструмент поддержки обобщений появился во второй версии платформы .NET вместе с пространством имен System.Collections.Generic, связанным с поддержкой коллекций. По сути **обобщение** – это развитие функциональности универсального типа **object**.

Для того, чтобы переменные **обобщенных типо**в легко отличались от необобщенных, приняты следующие соглашения:

- имена обобщенных типов должны начинаться с буквы Т;

- универсальный обобщенный тип, который может быть заменен любым классом, обозначается именем Т;

- если к обобщенному типу предъявляются какие-либо требования, то следует давать осмысленные имена, например: TEventArgs, TInput, TOutput.

Листинг 67. Пример создания обобщенного класса

```
namespace ConsoleApplication1
{
   // Создадим обобщенный класс имеющий параметр типа
T
   class MyObj<T>
   {
      T obj;
      public MyObj(T obj) //конструктор класса
      {
         this.obj  = obj;
      }
      public void objectType() //метод
      {
         Console.WriteLine("Тип объекта: " + typeof(T));
      }
   }

   class Program
   {
      static void Main()
      {
         // Создадим экземпляр обобщенного класса типа int
         MyObj<int> obj1 = new MyObj<int>(25);
         obj1.objectType();
         // Создадим экземпляр обобщ. класса типа string
         MyObj<string> obj2 = new
MyObj<string>("Строка");
         obj2.objectType();
         Console.ReadLine();
      }
   }
}
```

```
Тип объекта: System.Int32
Тип объекта: System.String
```

Часто есть необходимость ограничить перечень типов, передаваемых в качестве параметра. В этом случае используется т.н. **ограниченный тип**. Для создания ограниченного типа используется ключевое слово **where**.

Для инициализации обобщенной переменной рекомендуется использовать оператор **default,** возвращающий значение по умолчанию соответствующего типа. Так, ссылочному типу будет присвоено значение null, числовому 0 и т.д.

3.13. Коллекции. Обзор

В C# существует значительный набор **коллекций**, значительная часть которых содержится в пространствах имен **System.Collections** (простые необобщенные классы коллекций), **System.Collections.Generic** (обобщенные или типизированные классы коллекций) и **System.Collections.Specialized** (специальные классы коллекций).

Функционал коллекций значительно превосходит функционал массивов, т.к. коллекции могут хранить данные различных типов, размер коллекции может изменяться, и, помимо этого, некоторые коллекции реализуют такие употребимые структуры данных как стек, словарь, очередь.

Рассмотрим некоторые виды коллекций:

Необобщенные коллекции – реализуют такие структуры как динамический массив, стек, очередь, словари. Всегда оперируют данными типа object. Могут хранить данные любого типа, разнотипные данные. Классы необобщенных коллекций:

- **ArrayList**: класс простой коллекции объектов. Реализует интерфейсы IList, ICollection, IEnumerable
- **BitArray**: класс коллекции, содержащей массив битовых значений. Реализует интерфейсы ICollection, IEnumerable

- **Hashtable**: класс коллекции, представляющей хэш-таблицу и храняющий набор пар "ключ-значение"

- **Queue**: класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел"). Реализует интерфейсы ICollection, IEnumerable

- **SortedList**: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы ICollection, IDictionary, IEnumerable

- **Stack**: класс стека

Специальные коллекции оперируют данными конкретного типа особым образом. Например, имеются коллекции, поддерживающие специальные методу работы со строками.

Поразрядная коллекция – BitArray – поддерживает поразрядные операции над отдельными двоичными разрядами.

Обобщенные коллекции- типизированные коллекции, могут хранить только данные, совместимые по типу с коллекцией. Классы обобщенных коллекций:

- **List<T>**: класс, представляющий последовательный список однотипных объектов. Реализует интерфейсы IList<T>, ICollection<T>, IEnumerable<T>

- **Dictionary<TKey, TValue>**: класс коллекции, хранящей наборы пар "ключ-значение". Реализует интерфейсы ICollection<T>, IEnumerable<T>, IDictionary<TKey, TValue>

- **LinkedList<T>**: класс двухсвязанного списка. Реализует интерфейсы ICollection<T> и IEnumerable<T>

- **Queue<T>**: класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел"). Реализует интерфейсы ICollection, IEnumerable<T>

- **SortedSet<T>**: класс отсортированной коллекции однотипных объектов. Реализует интерфейсы ICollection<T>, ISet<T>, IEnumerable<T>

- **SortedList<TKey, TValue>**: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы ICollection<T>, IEnumerable<T>, IDictionary<TKey, TValue>
- **SortedDictionary<TKey, TValue>**: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. В общем похож на класс SortedList<TKey, TValue>, основные отличия состоят лишь в использовании памяти и в скорости вставки и удаления

Stack<T>: класс стека однотипных объектов. Реализует интерфейсы ICollection<T> и IEnumerable<T>

Параллельные коллекции - поддерживают многопоточный доступ к коллекции.

Рассмотрим пример работы с необобщенной и обобщенной коллекциями.

Листинг 68. Пример.

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

namespace Collections
{
    class Program
    {
        static void Main(string[] args)
        {
         // необобщенная коллекция ArrayList
        ArrayList objectList = new ArrayList() { 1, 2, "string", 'c', 2.0f };

            object obj = 45.8;

            objectList.Add(obj);
            objectList.Add("string2");
            objectList.RemoveAt(0); // удаление первого элемента
            foreach (object o in objectList)
            {
                Console.WriteLine(o);
            }
            Console.WriteLine("Общее число элементов коллекции: {0}",
objectList.Count);
            // обобщенная коллекция List
            List<string> countries = new List<string>() { "Россия", "США",
"Великобритания", "Китай" };
            countries.Add("Франция");
            countries.RemoveAt(1); // удаление второго элемента
            foreach (string s in countries)
            {
                Console.WriteLine(s);
            }

            Console.ReadLine();
        }
    }
}
```

3.14. Основы обработки исключений

Принята следующая классификация ошибок, возникающих при работе приложений:

- **программные ошибки (bugs)** – ошибки, допущенные программистом и не выявленные на этапе компиляции;

- **пользовательские ошибки (user errors)** – ошибки, возникающие по вине пользователя (например, ввод не правильных данных);

- **исключения (excaptions)** – не прогнозируемые ошибки времени исполнения, например, отсутствие места на диске при записи файла, удаление или разрушение базы данных и т.д.

Все перечисленные классы ошибок принято называть исключениями. Возникновение любого вида исключения приводит, как правило, к аварийному завершению программы. Чтобы избежать этого используются механизмы обработки исключений.

В .NET Framework предопределено значительное количество классов исключений, для обработки которых в C# используются программные конструкции трех типов:

- блоки **try** - выделяют блок кода, который потенциально может привести к возникновению ошибочных ситуаций;

- блоки **catch** – блок кода, обрабатывающий ошибочные ситуации, происходящие в коде блока try. Здесь же выполняется протоколирования ошибок.

- блоки **finally** – блок кода, очищающий ресурсы и выполняющий другие действия, которые обычно нужно выполнить в конце блоков **try** или **catch.** Важно понимать, что этот блок выполняется независимо от того, сгенерировано исключение или нет.

Листинг 69.

```
namespace ConsoleApplication1
{
   class Program
   {
      static int MyDel(int x, int y)
      {
         return x / y;
      }

      static void Main()
      {
         try
         {
            Console.Write("Введите x: ");
            int x = int.Parse(Console.ReadLine());
            Console.Write("Введите y: ");
            int y = int.Parse(Console.ReadLine());

            int result = MyDel(x, y);
            Console.WriteLine("Результат: " + result);
         }
         // Обрабатываем исключение возникающее при делении на ноль
         catch (DivideByZeroException)
         {
            Console.WriteLine("Деление на 0 detected!!!\n");
            Main();
         }
         // Обрабатываем исключение при неккоректном вводе числа в
консоль
         catch (FormatException)
         {
            Console.WriteLine("Это НЕ число!!!\n");
            Main();
         }

         Console.ReadLine();
      }
   }
}
```

Как видно из примера выше, существуют специализированные типы исключений, предназначенные для обработки ошибок определенного типа. Вот некоторые из них:

- **DivideByZeroException**: генерируется при делении на ноль;
- **ArgumentOutOfRangeException**: генерируется, если значение аргумента находится вне диапазона допустимых значений;
- **ArgumentException**: генерируется, если в метод для параметра передается некорректное значение;
- **IndexOutOfRangeException**: генерируется, если индекс элемента массива или коллекции находится вне диапазона допустимых значений;
- **InvalidCastException**: генерируется при попытке произвести недопустимые преобразования типов;
- **NullReferenceException**: генерируется при попытке обращения к объекту, который равен null (то есть по сути не определен).

Однако код может генерировать самые различные исключения, и перечислять их все в блоках **catch** не имеет смысла. Поэтому целесообразно в качестве последнего блока поставить **catch** для базового класса **Exception**, что позволит обработать все не перечисленные виды исключений:

Листинг 70.

```
static void Main(string[] args)
{
   try
   {
      object obj = "you";
      int num = (int)obj;    // InvalidCastException
      Console.WriteLine($"Результат: {num}");
   }
   catch (DivideByZeroException)
   {
      Console.WriteLine("Возникло исключение DivideByZeroException");
   }
   catch (IndexOutOfRangeException)
   {
      Console.WriteLine("Возникло исключение IndexOutOfRangeException");
   }
   catch (Exception ex)
   {
      Console.WriteLine($"Исключение: {ex.Message}");
          Console.WriteLine($"Метод: {ex.TargetSite}");
         Console.WriteLine($"Трассировка стека: {ex.StackTrace}");
   }
   Console.Read();
}
```

```
Исключение: Заданное приведение является недопустимым.
Метод: Void Main(System.String[])
Трассировка стека:    в ConsoleApplication1.Program.Main(String[] args) в C:\Use
rs\NAM\Documents\Visual Studio 2015\Projects\ConsoleApplication6\ConsoleApplicat
ion6\Program.cs:строка 30
```

ЗАКЛЮЧЕНИЕ

Владение средствами разработки для платформы .NET Framework весьма востребовано на сегодняшнем рынке разработки программ. Это объясняется тем, что вышеупомянутая технология позволяет выполнять разработку как настольных, так и Web-приложений в весьма сжатые сроки.

C# - язык программирования, реализующий все возможности платформы .NET, и предназначенный для выполнения быстрой разработки крупномасштабных приложений.

В отличие от C++, идеального для разработки участков кода, критичных к быстродействию, C# хорошо подходит для написания интерфейсов, разработки различных сервисов и служб, где важна скорость разработки.

Естественно, что для овладения всеми технологиями разработки программного обеспечения, ориентированными на использование платформы .NET, знания одного языка программирования C# будет недостаточно. Однако уже сегодня использование, например, фреймворка Xamarin делает возможной разработку на C# кроссплатформенных приложений для мобильных платформ.

В связи с тем, что C# считается наиболее простым и удобным из объектно-ориентированных языков семейства C, с уверенностью можно сказать, что последовательное овладение материалом, приведенным в данном учебном пособии, позволяет сформировать базовые навыки, необходимые для разработки программного обеспечения с использованием всего перечня технологий, представляемого средой Visual Studio.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 [Текст]: пер. с англ. / Э. Троелсен. - 6-е изд.— М. : ООО “И.Д. Вильямс”, 2013. — 1312 с.

2. Шарп, Д. Microsoft Visual C#. Подробное руководство [Текст] / Д. Шарп. - 8-е изд. - СПб.: Питер, 2017. — 848 с.

3. Кариев, Ч.А. Технология Microsoft ADO.NET [Текст] / Ч.А., Кариев. - 2-е изд., испр.- М.: Национальный открытый университет «ИНТУИТ», 2016. – 666 с.

4. Сайт о программировании: https://metanit.com/sharp/

5. Уроки по C# и платформе .NET Framework: http://professorweb.ru/

6. «Руководство по программированию на C#» https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 3
1. ПЛАТФОРМА .NET. СРЕДА РАЗРАБОТКИ VISUAL STUDIO 4
1.1. Платформа .NET Framework 4
1.2. Основные модули платформы .NET (CLR, CTS и CLS) 5
2. C#. НАЗНАЧЕНИЕ, ОРГАНИЗАЦИЯ, КОНСТРУКЦИИ ЯЗЫКА 12
2.1. Типы данных, их классификация 12
2.2. Классификация типов CTS 15
2.3. Встроенные (примитивные) типы 16
2.4. Преобразование типов 18
2.5. Типы-значения 20
2.6. Ссылочные типы 21
2.7. Упаковка и распаковка 21
2.8. Ключевое слово var. Неявная типизация 22
2.9. Операции C# 22
2.10. Операторы C# 25
2.11. Управляющие конструкции языка 29
2.12. Массивы 32
2.13. Строки. Работа со строками 35
3. ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ МЕТОДОЛОГИЯ В C# 41
3.1. Инкапсуляция. Классы. Содержимое классов 42
3.2. Доступ к членам класса 50
3.3. Наследование 51
3.4. Полиморфизм 54
3.5. Наследование и сокрытие имен 55
3.6. Абстрактные методы 58
3.7. Интерфейсы. Множественное наследование 59
3.8. Делегаты. События 63

3.9. Анонимные функции. Лямбда-выражения 68
3.10. События 70
3.11. Структуры 73
3.12. Обобщения. Краткие сведения 75
3.13. Коллекции. Обзор 77
3.14. Основы обработки исключений 81
ЗАКЛЮЧЕНИЕ 85
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 86
ОГЛАВЛЕНИЕ 87

Учебное издание

Нужный Александр Михайлович
Гребенникова Наталия Ивановна
Барабанов Владимир Федорович
Кремер Ольга Борисовна

**РАЗРАБОТКА ПРИЛОЖЕНИЙ НА C#
В СРЕДЕ VISUAL STUDIO**

Учебное пособие

Компьютерная верстка Н.И. Гребенниковой

Подписано к изданию xx.xx.xxxx.
Объем данных 11,8 Мб.

ФГБОУ ВО «Воронежский государственный технический университет»

394026 Воронеж, Московский просп., 14