Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А.А. Волосевич

ЯЗЫК C#
И ПЛАТФОРМА .NET

Учебно-методическое пособие
по курсу «Избранные главы информатики»
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Минск 2006

Библиотека БГУИР

УДК 681. 3. 062
ББК 32. 973. 26-018. 1 я 73
В 68

Рецензент:
заведующий отделом информатизации Института математики НАН Беларуси,
канд. физ.-мат. наук А.А. Сенько

Волосевич А.А.
В 68 Язык C# и платформа .NET: Учебно-метод. пособие по курсу «Избранные главы информатики» для студ. спец. I-31 03 04 «Информатика» всех форм обуч. / А.А. Волосевич. – Мн.: БГУИР, 2006. – 60 с.: ил.
ISBN 985-488-018-4

В пособие включены базовые сведения об архитектуре платформы Microsoft .NET. Рассматривается C# – новый язык программирования, разработанный специально для среды .NET. Описаны все основные аспекты языка: типы данных, операторы, управляющие инструкции, классы, интерфейсы, делегаты, индексаторы, события. Приводятся примеры приложений.

Пособие может быть рекомендовано студентам и магистрантам технических специальностей для изучения синтаксиса языка C# и основ программирования для Microsoft .NET Framework.

УДК 681. 3. 062
ББК 32. 973. 26-018. 1 я 73

ISBN 985-488-018-4

© Волосевич А.А., 2006
© БГУИР, 2006

Библиотека БГУИР

СОДЕРЖАНИЕ

Введение 4
1. Платформа .NET – обзор архитектуры 5
2. Язык C# – общие концепции синтаксиса. 6
3. Система типов языка C# 8
4. Преобразования типов 10
5. Идентификаторы, ключевые слова и литералы 13
6. Объявление переменных, полей и констант 14
7. Выражения и операции 16
8. Операторы языка C# 18
9. Объявление и вызов методов 21
10. Массивы в C# 23
11. Работа с символами и строками в C# 27
12. Синтаксис объявления класса, поля и методы класса 31
13. Свойства и индексаторы 33
14. Конструкторы класса и жизненный цикл объекта 36
15. Наследование классов 40
16. Перегрузка операторов 42
17. Делегаты 45
18. События. 47
19. Интерфейсы 50
20. Структуры и перечисления 53
21. Пространства имен 55
22. Генерация и обработка исключительных ситуаций 55
Литература 59

Библиотека БГУИР

ВВЕДЕНИЕ

В середине 2000 года корпорация Microsoft представила новую модель для создания приложений, основой которой является платформа .NET[1]. Платформа .NET образует каркас, который включает технологии разработки Windows-приложений, Web-приложений и Web-сервисов, технологии доступа к данным и межпрограммного взаимодействия. В состав платформы входит обширная библиотека классов. Основным инструментом для разработки является интегрированная среда MS Visual Studio.

Платформа .NET позволяет с легкостью создавать и интегрировать приложения, написанные на различных языках программирования. Специально для .NET был разработан язык программирования C#. Этот язык сочетает простой синтаксис, похожий на синтаксис языков C++ и Java, и полную поддержку всех современных объектно-ориентированных концепций и подходов. В качестве ориентира при разработке языка было выбрано безопасное программирование, нацеленное на создание надежного, простого в сопровождении кода.

Цель данного учебного пособия – описать синтаксис языка программирования C#. В пособии особо выделяются и рассматриваются аспекты языка C#, связанные со спецификой платформы .NET. Хотелось бы подчеркнуть, что синтаксис языка C# рассматривается по спецификации первой версии данного языка. В ноябре 2005 года корпорация Microsoft выпустила новую, вторую версию платформы .NET. Многочисленные изменения этой версии затронули и язык C#. Относительная новизна второй версии платформы, а также малый объем учебного пособия не позволили в данной работе рассмотреть эти изменения.

Пособие содержит фрагменты кода и небольшие программы, иллюстрирующие теоретический материал. Данных примеров достаточно для начала самостоятельного программирования на C#. Примеры могут служить основой при написании лабораторных работ, связанных с объектно-ориентированным программированием с использованием C#.

Библиотека БГУИР

[1] Произносится как «дот-нэт».

1. ПЛАТФОРМА .NET – ОБЗОР АРХИТЕКТУРЫ

Задача *платформы .NET* (*.NET Framework*) – предоставить программистам эффективную и гибкую среду разработки традиционных и Web-приложений. Одна из наиболее важных особенностей .NET Framework – способность обеспечить совместную работу кода, написанного на различных языках программирования. На рис. 1 показана структура платформы .NET на самом высоком уровне.

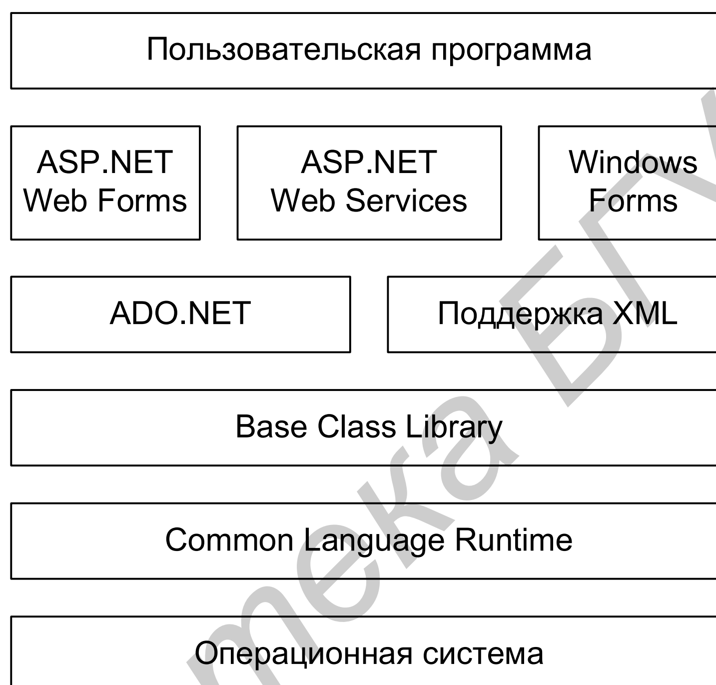

Рис. 1. Общая структура .NET Framework

Базой платформы является *общеязыковая среда исполнения* (*Common Language Runtime*, *CLR*). CLR является «прослойкой» между операционной системой и кодом приложений для .NET Framework. Такой код называется *управляемым* (*managed code*). Более подробно роль CLR обсуждается далее.

В состав платформы .NET входит библиотека классов *Framework Class Library* (*FCL*). Элементом этой библиотеки является базовый набор классов *Base Class Library* (*BCL*). В BCL входят классы для работы со строками, коллекциями данных, поддержки многопоточности и множество других классов. Частью FCL являются компоненты, поддерживающие различные технологии обработки данных и организации взаимодействия с пользователем. Это классы для работы с XML, базами данных (ADO.NET), создания Windows-приложений и Web-приложений (ASP.NET).

В стандартную поставку .NET Framework включены компиляторы для платформы. Это компиляторы языков C#, Visual Basic.NET, J#. Благодаря открытым спецификациям компиляторы для .NET предлагаются различными сто-

Библиотека БГУИР

ронними производителями (не Microsoft). На данный момент количество компиляторов измеряется десятками.

Рассмотрим подробнее компоненты и роль CLR. Любой компилятор для .NET позволяет получить из исходного текста программы двоичный исполняемый файл или библиотеку кода. Однако эти файлы по своей структуре и содержанию не имеют ничего общего с традиционными исполняемыми файлами операционной системы. Двоичные файлы для платформы .NET называются *сборками* (*assembly*). Сборка состоит из следующих частей:

1. *Манифест* (*manifest*) – описание сборки: версия, ограничения безопасности, список внешних сборок и файлов, необходимых для работы данной сборки.

2. *Метаданные* – специальное описание всех пользовательских типов данных, размещенных в сборке.

3. *Код на промежуточном языке Microsoft Intermediate Language* (*MSIL* или просто *IL*). Данный код является независимым от операционной системы и типа процессора, на котором будет выполняться приложение. В процессе работы приложения он компилируется в машинно-зависимый код специальным компилятором (*Just-in-Time compiler*, *JIT compiler*).

Основная задача CLR – это манипулирование сборками: загрузка сборок, трансляция кода IL в машинно-зависимый код, создание окружения для выполнения сборок. Важной функцией CLR является управление размещением памяти при работе приложение и выполнение *автоматической сборки мусора*, то есть фонового освобождения неиспользуемой памяти. Кроме этого, CLR реализует в приложениях для .NET верификацию типов, управление политиками безопасности при доступе к коду и некоторые другие функции.

Кроме упомянутых элементов, выделим еще две части платформы .NET:

- *Система типов данных* (*Common Type System*, *CTS*) – базовые, не зависящие от языка программирования примитивные типы, которыми может манипулировать CLR.
- *Набор правил* для языка программирования (*Common Language Specification*, *CLS*), соблюдение которых обеспечивает создание на разных языках программ, легко взаимодействующих между собой.

В заключение хотелось бы подчеркнуть, что любой компилятор для .NET является верхним элементом архитектуры. Библиотека классов FCL, имена ее элементов не зависят от языка программирования. Специфичным элементом языка остается только синтаксис, но не работа с внешними классами. Это упрощает межъязыковое взаимодействие, перевод текста программы с одного языка на другой. С другой стороны, тесная связь с CLR неизбежно находит свое отражение в синтаксических элементах языка программирования.

2. ЯЗЫК C# – ОБЩИЕ КОНЦЕПЦИИ СИНТАКСИСА

Ключевыми структурными понятиями в языке C# являются *программы*, *пространства имен*, *типы*, *элементы типов* и *сборки*. Программа на языке C#

Библиотека БГУИР

размещается в одном или нескольких текстовых файлах, стандартное расширение которых – `.cs`. В программе объявляются пользовательские типы, которые состоят из элементов. Примерами пользовательских типов являются классы и структуры, а примером элемента типа может служить метод класса. Типы могут быть логически сгруппированы в пространства имен. При компиляции программы получается сборка, представляющая собой файл с расширением `.exe` или `.dll`.

Исходный текст программы на языке C# содержит *операторы* и *комментарии*. Основными видами операторов в C# являются следующие.

- *Оператор-выражение*. Под выражением может пониматься вызов метода, присваивание, а также допустимые комбинации операндов и операций. Оператор-выражение завершается символом `;` (точка с запятой).
- *Операторы управления* ходом выполнения программы, такие как оператор условного перехода или операторы циклов.
- *Блок операторов*. Блок – это набор операторов, обрамленных фигурными скобками – `{` и `}`. Блоки используют там, где синтаксис языка требует одного оператора.
- *Операторы объявлений* пользовательских типов, элементов типов и локальных переменных и констант.

Программа может содержать комментарии, игнорируемые при компиляции. Различают следующие виды комментариев:

1. *Строчный комментарий* – это комментарий, начинающийся с последовательности `//` и продолжающийся до конца строки.

2. *Блочный комментарий* – все символы, заключенные между `/*` и `*/`.

3. *Комментарии для документации* – напоминают строчные комментарии, но начинаются с последовательности `///` и могут содержать специальные XML-тэги.

В языке C# различаются строчные и прописные символы при записи идентификаторов и ключевых слов. Количество пробелов в начале строки, в конце строки и между элементами строки значения не имеет. Это позволяет улучшить структуру исходного текста программы.

Программа «Hello, World» традиционно используется для первого знакомства с языком программирования. Вот пример этой программы на языке C#.

```
using System;
class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

Дадим некоторые пояснения. Программа представляет собой описание пользовательского типа – класса `Hello`. Любая исполняемая программа на C#

Библиотека БГУИР

должна иметь специальную *точку входа*, с которой начинается выполнение приложения. Такой точкой входа является статический метод `Main()`, объявленный в некотором классе программы (в данном случае – в классе `Hello`). Метод `Main()` содержит вызов метода `WriteLine()` класса `Console` из пространства имен `System`. Ключевое слово `using` служит для подключения пространства имен `System`, содержащего базовые классы. Использование `using` позволяет вместо полного имени класса `System.Console` записать короткое имя `Console`.

Если программа содержится в файле `hello.cs`, то она может быть скомпилирована при помощи компилятора командной строки `csc.exe`.

```
csc hello.cs
```

После компиляции будет получена сборка `hello.exe`.

В заключение параграфа заметим, что большинство примеров в данном пособии представляет собой простые консольные приложения. В таких приложениях для вывода информации используются методы `WriteLine()` и `Write()` класса `Console`. Ввод данных осуществляется функцией `Console.ReadLine()`. Функция возвращает введенную строку, которая обычно преобразуется в значение требуемого типа.

3. СИСТЕМА ТИПОВ ЯЗЫКА C#

Основой C# является развитая система типов. Проведем ее классификацию. С точки зрения размещения переменных в памяти все типы можно разделить на *структурные типы* и *ссылочные типы*. Переменная структурного типа содержит непосредственно данные и размещается в стеке. Переменная ссылочного типа, далее называемая *объектом*, содержит ссылку на данные, которые размещены в управляемой динамической памяти. Структурными типами являются *примитивные типы*, *перечисления* и *структуры*. Ссылочные типы – это *классы*, *интерфейсы*, *массивы* и *делегаты*.

Числовые типы составляют подмножество примитивных типов. Информация о числовых типах содержится в табл. 1.

Таблица 1

Числовые типы языка C#

Категория	**Размер (бит)**	**Имя типа**	**Диапазон/Точность**
Знаковые целые	8	`sbyte`	–128...127
	16	`short`	–32 768...32 767
	32	`int`	–2 147 483 648...2 147 483 647
	64	`long`	-9 223 372 036 854 775 808...9 223 372 036 854 775 807
Беззнаковые целые	8	`byte`	0...255
	16	`ushort`	0...65535
	32	`uint`	0...4294967295
	64	`ulong`	0...18446744073709551615

Библиотека БГУИР

Вещественные	32	`float`	Точность: от 1.5×10^{-45} до 3.4×10^{38}, 7 цифр
	64	`double`	Точность: от 5.0×10^{-324} до 1.7×10^{308}, 15 цифр
	128	`decimal`	Точность: от 1.0×10^{-28} до 7.9×10^{28}, 28 цифр

Отметим, что типы `sbyte`, `ushort`, `uint`, `ulong` не соответствуют Common Language Specification. Это означает, что данные типы не следует использовать в интерфейсах многоязыковых приложений и библиотек. Тип `decimal` удобен для проведения финансовых вычислений.

Примитивный тип `bool` служит для представления булевых значений. Переменные данного типа могут принимать значения `true` или `false`.

При работе с символами и строками в C# используется кодировка Unicode. Тип `char` представляет символ в 16-битной Unicode-кодировке, тип `string` – это последовательность Unicode-символов. Заметим, что хотя тип `string` относится к примитивным, переменная этого типа хранит адрес строки в динамической памяти.

Имя примитивного типа в языке C# является синонимом соответствующего типа Framework Class Library. Например, типу `int` в C# соответствует тип `System.Int32`, типу `float` – тип `System.Single` и т. д.

Перечисления, структуры, классы, интерфейсы, массивы и делегаты составляют множество *пользовательских типов*. Элементы пользовательских типов должны быть описаны программистом при помощи особых синтаксических конструкций. Можно утверждать, что любая программа на языке C# представляет собой набор определенных пользовательских типов. Опишем функциональность, которой обладают пользовательские типы.

1. Класс – тип, поддерживающий всю функциональность объектно-ориентированного программирования, включая наследование и полиморфизм.
2. Структура – тип, обеспечивающий всю функциональность ООП, кроме наследования. Структура в C# очень похожа на класс, за исключением метода размещения в памяти и отсутствия поддержки наследования.
3. Интерфейс – абстрактный тип, реализуемый классами и структурами для обеспечения оговоренной функциональности.
4. Массив – пользовательский тип для представления упорядоченного набора значений некоторых (примитивных или пользовательских) типов.
5. Перечисление – тип, содержащий в качестве членов именованные целочисленные константы.
6. Делегат – пользовательский тип для представления ссылок на методы.

В .NET Framework сглажено различие между типами и классами. А именно, любой тип можно воспринимать как класс, который может быть связан с другими типами (классами) отношением наследования. Это позволяет рассматривать все типы .NET Framework (и языка C#) в виде *иерархии классов*. При этом существует базовый тип `System.Object` (в C# – `object`), являющийся общим предком всех типов. Все структурные типы наследуются от класса `System.ValueType`.

Библиотека БГУИР

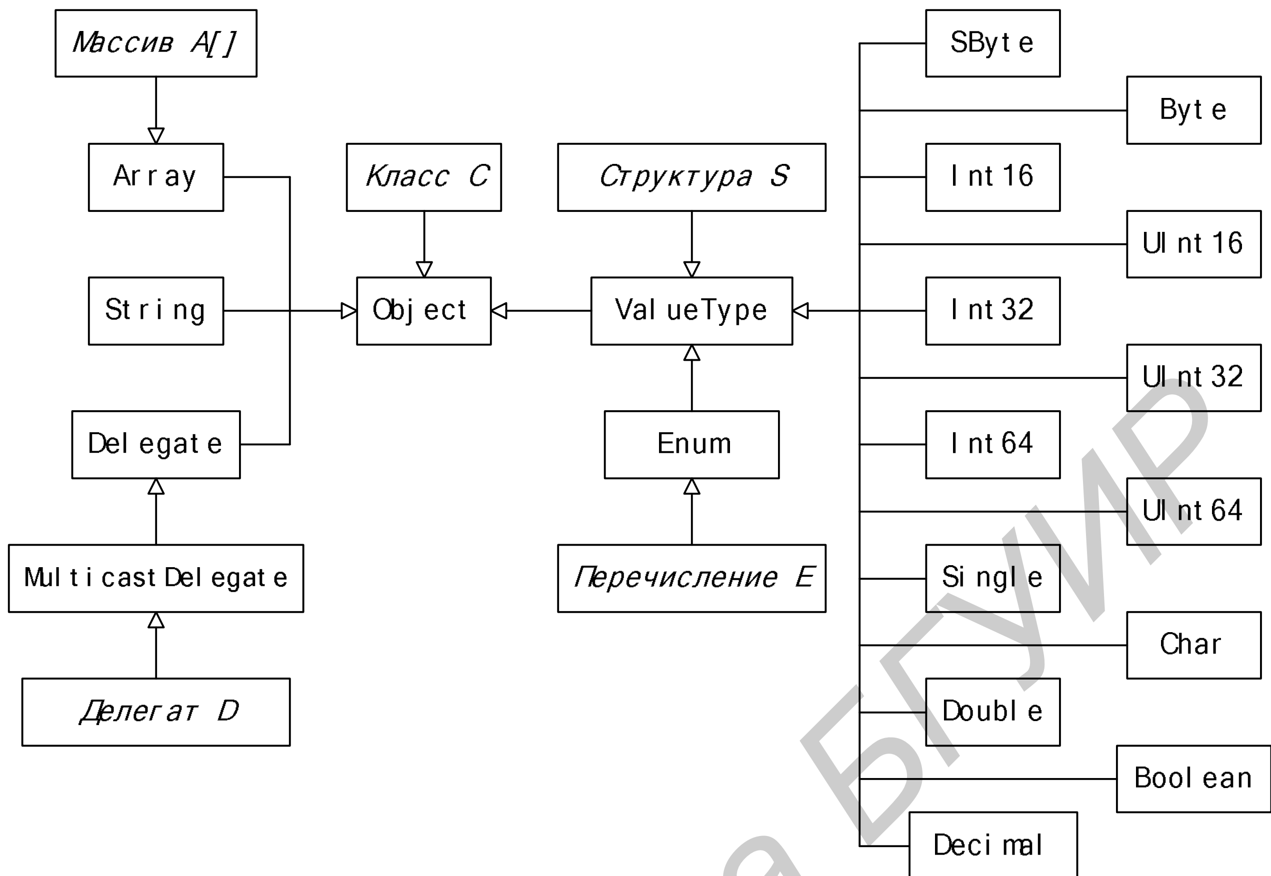

Рис. 2. Иерархия типов .NET Framework

В C# допускается рассмотрение значений структурных типов как переменных типа `object`. Преобразование в объект называется *операцией упаковки* (*boxing*), обратное преобразование – *операцией распаковки* (*unboxing*). При упаковке в динамической памяти создается объект, содержащий значение структурного типа. При распаковке проверяется фактический тип объекта, и значение из динамической памяти переписывается в соответствующую переменную в стеке. Операция распаковки требует явного указания целевого типа.

```
int i = 123;
object o = i;              // Упаковка
int j = (int)o;            // Распаковка
```

Возможность автоматического преобразование каждого типа в тип `object` позволяет создавать универсальные классы, работающие с любыми типами. Например, класс `ArrayList` может содержать коллекцию пользовательских объектов, или целых чисел или строк.

4. ПРЕОБРАЗОВАНИЯ ТИПОВ

Если при вычислении выражения операнды имеют разные типы, то возникает необходимость приведения их к одному типу. Такая необходимость возникает и тогда, когда операнды имеют один тип, но он несогласован с типом операции. Например, при выполнении сложения операнды типа `byte` должны быть приведены к типу `int`, поскольку сложение не определено над байтами. При

Библиотека БГУИР

выполнении присваивания x=e тип источника e и тип цели x должны быть согласованы. Аналогично, при вызове метода также должны быть согласованы типы фактического и формального аргументов.

Рассмотрим преобразования при работе с числовыми типами. Заметим, что преобразование типов бывает неявным и явным. *Неявное преобразование* (*implicit conversion*) выполняется автоматически. При выполнении данного преобразования никогда не происходит потеря точности или переполнение, так как множество значений целевого типа включает множества значений приводимого типа. Для числовых типов неявное преобразование типа A в тип B возможно, если на схеме 3 существует путь из A в B.

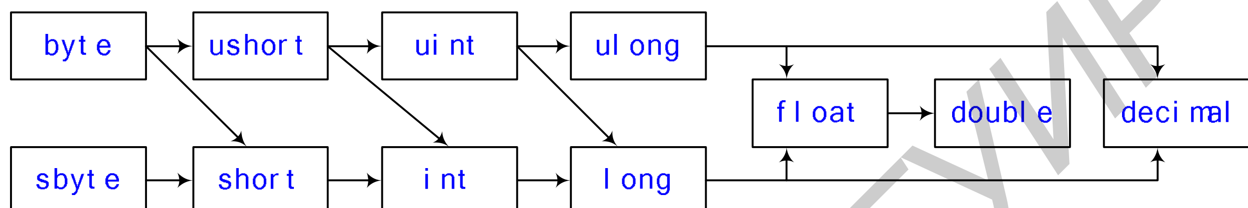

Рис. 3. Схема неявного преобразования числовых типов.

Для *явного преобразования* (*explicit conversion*) требуется применять *оператор приведения* в форме (<целевой тип>)<выражение>. При выполнении явного преобразования ответственность за его корректность возлагается на программиста.

```
int k = 100;
byte i;                  //тип byte «меньше» типа int
i = (byte) k;            //требуется явное преобразование типов
```

Для более гибкого контроля значений, получаемых при работе с числовыми выражениями, в языке C# предусмотрено использование контролируемого и неконтролируемого контекстов. *Контролируемый контекст* объявляется в форме checked <программный блок>, либо как оператор checked(<выражение>). Если при преобразовании типов выражение в контролируемом контексте получает значение, выходящие за пределы целевого типа, то генерируется либо ошибка компиляции (для константных выражений), либо ошибка времени выполнения (для выражений с переменными).

При использовании *неконтролируемого контекста* выход за пределы целевого типа ведет к автоматическому «урезанию» результата либо путем отбрасывания бит (целые типы), либо путем округления (вещественные типы). *Неконтролируемый контекст* объявляется в форме unchecked <программный блок>, либо как оператор unchecked(<выражение>).

Рассмотрим несколько примеров использования контекстов:

```
int i = 1000000;
int k = 1000000;
int n = i * k;
```

Библиотека БГУИР

В данном примере n получит значение -727379968, то есть произойдет отбрасывание «лишних» бит числа, получившегося в результате умножения. Используем неконтролируемый контекст:

```
int i = 1000000;
int k = 1000000;
int n = unchecked(i * k);
```

Значение n осталось прежним (-727379968), таким образом, неконтролируемый контекст применяется по умолчанию.

Теперь проведем вычисления в контролируемом контексте:

```
int i = 1000000;
int k = 1000000;
int n = checked(i * k);
```

При выполнении последнего оператора произойдет генерация исключения System.OverflowException – переполнение.

Важным классом преобразований являются преобразования в строковый тип и наоборот. Преобразования в строковый тип всегда определены, поскольку все типы являются потомками базового класса object, а, следовательно, обладают методом ToString() этого класса. Для встроенных типов определена подходящая реализация этого метода. В частности, для всех числовых типов метод ToString() возвращает строку, задающую соответствующее значение типа. Метод ToString() можно вызывать явно, но, если явный вызов не указан, то он будет вызываться неявно, всякий раз, когда требуется преобразование к строковому типу. Преобразования из строкового типа в другие типы должны всегда выполняться явно при помощи методов встроенных или пользовательских классов. Например, класс System.Int32 обладает методом Parse(), позволяющим преобразовать строку в целое число.

```
System.Console.WriteLine("Input your age");
string s = System.Console.ReadLine();
int Age = System.Int32.Parse(s);
```

Тип char преобразуется в типы sbyte, short, byte явно, а в остальные числовые типы – неявно. Заметим, что преобразование любого числового типа в тип char может быть выполнено, но только в явной форме.

Преобразования пользовательских ссылочных типов выполняются при присваивании и вызове методов. При этом действует стандартное правило ООП: объект типа-потомка преобразуется в объект типа-предка автоматически. Все прочие преобразования должны быть выполнены при помощи оператора приведения. Ответственность за их правильность возлагается на программиста.

В пространстве имен System содержится класс Convert, методы которого поддерживают общий способ выполнения преобразований между типами. Класс Convert содержит набор статических методов вида To<Type>(), где Type – имя встроенного типа CLR (ToBoolean(),ToUInt64() и т. д.). Все методы To<Type>() класса Convert перегружены и каждый из них имеет, как правило,

более десятка реализаций с аргументами разного типа. Так что фактически эти методы задают все возможные преобразования между всеми встроенными типами языка C#. Кроме методов, задающих преобразования типов, в классе `Convert` имеются и другие методы, например, задающие преобразования символов Unicode в однобайтную кодировку ASCII.

5. ИДЕНТИФИКАТОРЫ, КЛЮЧЕВЫЕ СЛОВА И ЛИТЕРАЛЫ

Идентификатор – это пользовательское имя для переменной, константы, метода или типа. В C# идентификатор – это произвольная последовательность букв, цифр и символов подчеркивания, начинающаяся с буквы, символа подчеркивания, либо с символа `@`. Идентификатор должен быть уникальным внутри области использования. Он не может совпадать с ключевым словом языка, за исключением того случая, когда используется специальный *префикс* `@`. Примеры допустимых идентификаторов: `Temp`, `_Variable`, `@class` (используется префикс `@`, `class` – ключевое слово).

Далее представлен список всех ключевых слов языка 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   while
```

В C# *литерал* – это последовательность символов, которая может интерпретироваться как значение одного из примитивных типов. Так как язык C# является языком со строгой типизацией, иногда необходимо явно указать, к какому типу относится последовательность символов, определяющая данные.

Рассмотрим правила записи некоторых литералов. В языке C# два булевых литерала: `true` и `false`. Целочисленные литералы могут быть записаны в десятичной или шестнадцатеричной форме. Признаком шестнадцатеричного литерала является префикс `0x`. Конкретный тип целочисленного литерала определяется следующим образом:

- Если литерал не имеет суффикса, то его тип – это первый из типов `int`, `uint`, `long`, `ulong`, который способен вместить значение литерала.
- Если литерал имеет суффикс `U` или `u`, его тип – это первый из типов `uint`, `ulong`, который способен вместить значение литерала.

Библиотека БГУИР

- Если литерал имеет суффикс `L` или `l`, то его тип – это первый из типов `long`, `ulong`, который способен вместить значение литерала.
- Если литерал имеет суффикс `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU` или `lu`, то его тип – `ulong`.

Если в числе с десятичной точкой не указан суффикс, то подразумевается тип `double`. Суффикс `f` (или `F`) используется для указания на тип `float`, суффикс `d` (или `D`) используется для явного указания на тип `double`, суффикс `m` (или `M`) определяет литерал типа `decimal`. Число с плавающей точкой может быть записано в научном формате: `3.5E-6`, `-7E10`, `.6E+7`.

Символьный литерал обычно записывают как единичный символ в кавычках (`'a'`). Однако таким образом нельзя представить символы `'` и `\`. Альтернативным способом записи символьного литерала является использование шестнадцатеричного значения кода Unicode, заключенного в одинарные кавычки (`'\x005C'` или `'\u005C'` – это символ `\`). Кроме этого, для представления некоторых специальных символов используются следующие пары:

- `\'` – одинарная кавычка;
- `\"` – двойная кавычка;
- `\\` – обратный слеш;
- `\0` – пустой символ (null);
- `\a` – оповещение;
- `\b` – забой;
- `\f` – новая страница;
- `\n` – новая строка;
- `\r` – возврат каретки;
- `\t` – горизонтальная табуляция;
- `\v` – ввертикальная табуляция.

Для строковых литералов в языке C# существуют две формы. Обычно строковый литерал записывается как последовательность символов в двойных кавычках. Среди символов строки могут быть и управляющие последовательности (`"This is \t tabbed string"`). *Дословная форма* (*verbatim form*) строкового литерала – это запись строки в кавычках с использованием префикса `@` (`@"There is \t no tab"`). В этом случае управляющие последовательности воспринимаются как обычные пары символов.

6. ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ, ПОЛЕЙ И КОНСТАНТ

Объявление переменных в языке C# может использоваться в двух контекстах. В первом контексте объявление используется на уровне метода и описывает *локальную переменную*. Во втором контексте оно используется на уровне пользовательского типа (класса или структуры). В этом случае правильнее было бы говорить об объявлении *поля типа*.

Для объявления переменных и полей в C# используется оператор следующего формата:

Библиотека БГУИР

```
<тип> <имя переменной или поля> [= <начальное значение>];
```

Здесь <имя переменной или поля> – допустимый идентификатор, <тип> – тип переменной, <начальное значение> – литерал или выражение, соответствующие типу переменной и задающие начальное значение. Если начальное значение переменной не задано, то поля получают значение 0 для числовых типов, false для типа bool, символ с кодом 0 для типа char. Любой ссылочный тип, включая строки и массивы, получает специальное значение null. Начальное значение может быть задано как для полей пользовательских типов, так и для локальных переменных методов. Однако локальные переменные методов не могут использоваться без инициализации (она может быть выполнена как в момент объявления, так и позднее).

Если необходимо объявить несколько переменных или полей одного типа, то идентификаторы переменных можно перечислить через запятую после имени типа. При этом для каждой переменной можно выполнить начальную инициализацию.

```
int a;           //Простейший вариант объявления
int a = 20;      //Объявление с инициализацией
int a, b, c;     //Объявление нескольких однотипных переменных
int a = 20, b = 10;    //Объявление и инициализация переменных
```

Локальная переменная метода может быть объявлена в программном блоке. В этом случае *время жизни* переменной ограничено блоком:

```
{
    int i = 10;
    Console.WriteLine(i);
}
// ошибка компиляции – переменная i не доступна!!!
Console.WriteLine(i);
```

Если программные блоки вложены друг в друга, то внутренний блок не может содержать объявлений переменных, идентификаторы которых совпадают с переменными внешнего блока:

```
{
    int i = 10;
    {
        //ошибка компиляции – i существует во внешнем блоке
        int i = 20;
    }
}
```

Как и другие языки программирования, C# позволяет описать в пользовательском типе или в теле метода константы. Синтаксис объявления константы следующий:

```
const <тип константы> <имя константы> = <значение константы>;
```

Библиотека БГУИР

Тип константы – это любой примитивный тип (за исключением `object`), `<значение константы>` может быть литералом или результатом действий с другими константами. Примеры объявления констант:

```
const double Pi = 3.1415926;
const double Pi2 = Pi + 2;
```

Для полей пользовательских типов возможно применение модификатора `readonly`, который фактически превращает их в константу. Однако в отличие от констант, тип такого поля может быть любым:

```
public readonly int Age;
```

Для полей классов, которые объявлены с модификатором `readonly`, начальное значение может устанавливать только конструктор (но оно может быть указано и при объявлении поля). Использование модификатора `readonly` для локальных переменных запрещено.

При написании программ часто возникает необходимость разграничить доступ к пользовательским типам или их элементам. Например, скрыть поля класса, ограничить доступ к методу, описанному в некотором классе и т. п. Для этих целей применяют *модификаторы доступа*, указываемые непосредственно перед объявлением типа или элемента типа. Возможно использование следующих модификаторов:

- `private`. Элемент с данным модификатором доступен только в том типе, в котором определен. Например, поле доступно только в содержащем его классе.
- `public`. Элемент доступен без ограничений как в той сборке, где описан, так и в других сборках, к которым подключается сборка с элементом.
- `internal`. Элемент доступен без ограничений, но только в той сборке, где описан.
- `protected`. Элемент с данным модификатором видим только в типе, в котором определен, и в наследниках данного типа (даже если эти наследники расположены в других сборках). Данный модификатор может применяться только в типах, поддерживающих наследование, то есть в классах.
- `protected internal`. Комбинация модификаторов `protected` и `internal`. Элемент виден в содержащей его сборке без ограничений, а вне сборки – только в наследниках типа.

Для локальных переменных методов и программных блоков модификаторы доступа не используются.

Библиотека БГУИР

7. ВЫРАЖЕНИЯ И ОПЕРАЦИИ

Любое выражение в языке C# состоит из операндов и операций. В табл. 2 представлен список операций языка C#, в котором они расположены по убыванию приоритета.

Библиотека БГУИР

Таблица 2

Операции языка C#

Категория	**Выражение**	**Описание**
Первичные	`x.m`	Доступ к элементу типа
	`x(...)`	Вызов методов и делегатов
	`x[...]`	Доступ к элементу массива и индексатора
	`x++`	Постинкремент
	`x--`	Постдекремент
	`new T(...)`	Создание объекта или делегата
	`new T[...]`	Создание массива
	`typeof(T)`	Получение для типа `T` объекта `System.Type`
	`checked(x)`	Вычисление в контролируемом контексте
	`unchecked(x)`	Вычисление в неконтролируемом контексте
Унарные	`+x`	Идентичность
	`-x`	Отрицание
	`!x`	Логическое отрицание
	`~x`	Битовое отрицание
	`++x`	Пре-инкремент
	`--x`	Пре-декремент
	`(T)x`	Явное преобразование `x` к типу `T`
Умножение	`x * y`	Умножение
	`x / y`	Деление
	`x % y`	Вычисление остатка
Сложение	`x + y`	Сложение, конкатенация строк
	`x - y`	Вычитание
Сдвиг	`x << y`	Битовый сдвиг влево
	`x >> y`	Битовый сдвиг вправо
Отношение и проверка типов	`x < y`	Меньше
	`x > y`	Больше
	`x <= y`	Меньше или равно
	`x >= y`	Больше или равно
	`x is T`	Возвращает `true`, если тип `x` это `T`
	`x as T`	Возвращает `x`, приведенный к типу `T`, или `null`
Равенство	`x == y`	Равно
	`x != y`	Не равно
Логическое AND	`x & y`	Целочисленное битовое AND, логическое AND
Логическое XOR	`x ^ y`	Целочисленное битовое XOR, логическое XOR
Логическое OR	`x | y`	Целочисленное битовое OR, логическое OR
Сокращенное AND	`x && y`	Вычисляется `y`, только если `x = true`
Сокращенное OR	`x || y`	Вычисляется `y`, только если `x = false`
Условие	`x ? y : z`	Если `x = true`, вычисляется `y`, иначе `z`
Присваивание	`x = y`	Присваивание
	`x` *op=* `y`	Составное присваивание, поддерживаются `*= /= %= += -= <<= >>= &= ^= |=`

Правила работы с операциями в C# в основном совпадают с аналогичными правилами в языке C++. Тип результата арифметических операций – это

Библиотека БГУИР

«больший» из типов операндов. Таким образом, `5/2 = 2` (так как операнды целые, то и результат – целый тип), а `5/2d = 2.5`. Составное присваивание неявно включает приведение к типу переменной в левой части. Деление на `0` для вещественных типов не вызывает ошибку – результатом являются специальные значения `infinity` или `NaN` (то есть «бесконечность» при делении на ноль и «не число», если ноль делится на ноль).

8. ОПЕРАТОРЫ ЯЗЫКА C#

В языке C# описания типов, методов, свойств, синтаксические конструкции операторов ветвления и циклов образуют в тексте программные блоки. *Программный блок* – это последовательность операторов (возможно пустая), заключенная в фигурные скобки `{` и `}`.

Рассмотрим операторы языка C# для управления ходом выполнения программы. Оператор `break` используется для выхода из блоков операторов `switch`, `while`, `do`, `for` или `foreach`. Оператор `break` выполняет переход на оператор за блоком.

Оператор `continue` применяется для запуска новой итерации циклов `while`, `do`, `for` или `foreach`. Оператор располагается в теле цикла. Если циклы вложены, то запускается новая итерация того цикла, в котором непосредственно располагается `continue`.

Оператор `goto` передает управление на помеченный оператор. Обычно данный оператор употребляется в форме `goto <метка>`, где `<метка>` – это допустимый идентификатор. Метка должна предшествовать помеченному оператору и заканчиваться двоеточием, отдельно описывать метки не требуется:

```
goto label;
. . .
label:
A = 100;
```

Оператор `goto` и помеченный оператор должны располагаться в одном программном блоке. Возможно использование команды `goto` в одной из следующих форм:

```
goto case <константа>;
goto default;
```

Данные варианты обсуждаются при рассмотрении оператора `switch`.

Оператор условного перехода в языке C# имеет следующий формат:

```
if (<условие>)
    <блок1>
[else
    <блок2>]
```

Здесь `<условие>` – это некоторое булево выражение. Ветвь `else` является необязательной.

Библиотека БГУИР

Оператор выбора switch выполняет одну из групп инструкций в зависимости от значения тестируемого выражения. Синтаксис оператора switch:

```
switch (<выражение>)
{
    case <константное выражение>:
       <оператор 1>
       . . .
       <оператор n>
       <оператор перехода>
    case <константное выражение 2>:
       <оператор 1>
       . . .
       <оператор n>
       <оператор перехода>
    . . .
    [default:
       <оператор 1>
       . . .
       <оператор n>
       <оператор перехода>]
}
```

Тестируемое <выражение> должно иметь целый числовой тип, символьный или строковый тип. При совпадении тестируемого и константного выражений выполняется соответствующая ветвь case. Если совпадения не обнаружено, то выполняется секция default (если она есть). <оператор перехода> – это один из следующих операторов: break, goto, return. Оператор goto используется с указанием либо ветви default (goto default), либо определенной ветви case (goto case <константное выражение>).

Приведем пример использования оператора switch:

```
Console.WriteLine("Input number");
int n = Int32.Parse(Console.ReadLine());
switch (n)
{
    case 0:
        Console.WriteLine("Null");
        break;
    case 1:
        Console.WriteLine("One");
        goto case 0;
    case 2:
        Console.WriteLine("Two");
        goto default;
    case 3:
        Console.WriteLine("Three");
        return;
    default:
        Console.WriteLine("I do not know");
        break;
```

Библиотека БГУИР

```
}
```

Хотя после case может быть указано только одно константное выражение, при необходимости несколько ветвей case можно сгруппировать следующим образом:

```
switch (n)
{
    case 0:
    case 1:
    case 2:
        . . .
}
```

C# представляет разнообразный набор операторов организации циклов. Для циклов с определенным числом итераций используется оператор for. Его синтаксис:

```
for ([<инициализатор>]; [<условие>]; [<итератор>]) <блок>
```

<инициализатор> задает начальное значение счетчика (или счетчиков) цикла. В инициализаторе может использоваться существующая переменная для счетчика или объявляться новая переменная, время жизни которой будет ограничено циклом. Цикл выполняется, пока булево <условие> истинно, а <итератор> определяет изменение счетчика цикла.

Простейший пример использования цикла for:

```
for (int i = 0; i < 10; i++)  // i доступна только в цикле for
    Console.WriteLine(i);     // вывод чисел от 0 до 9
```

В инициализаторе можно объявить и задать начальные значения для нескольких счетчиков одного типа. В этом случае итератор может представлять собой последовательность из нескольких операторов, разделенных запятой:

```
// цикл выполнится 5 раз, на последней итерации i=4, j=6
for (int i = 0, j = 10; i < j; i++, j--)
    Console.WriteLine("i = {0}, j = {1}", i, j);
```

Если число итераций цикла заранее не известно, можно использовать цикл while или цикл do/while. Данные циклы имеют схожий синтаксис объявления:

```
while (<условие>) <блок>

do
    <блок>
while (<условие>);
```

В обоих циклах тело цикла выполняется, пока булево <условие> истинно. В цикле while условие проверяется в начале очередной итерации, а в цикле do/while – в конце. Таким образом, цикл do/while всегда выполнится по крайней мере один раз. Обратите внимание, <условие> должно присутствовать обязательно. Для организации бесконечных циклов на месте условия можно использовать литерал true:

Библиотека БГУИР

```
while (true) Console.WriteLine("Бесконечный цикл!");
```

Для перебора элементов массивов и коллекций в языке C# существует специальный цикл `foreach`:

```
foreach (<тип> <идентификатор> in <коллекция>) <блок>
```

В заголовке цикла объявляется переменная, которая будет последовательно принимать значения элементов коллекции. При этом присваивание переменной новых значений не отражается на элементах коллекции. Для выполнения цикла `foreach` над коллекцией необходимо, чтобы коллекция реализовывала интерфейс `IEnumerable`.

9. ОБЪЯВЛЕНИЕ И ВЫЗОВ МЕТОДОВ

Методы в языке C# являются неотъемлемой частью описания таких пользовательских типов как класс и структура. В C# не существует глобальных методов – любой метод должен быть членом класса или структуры.

Рассмотрим «облегченный» синтаксис описания метода (не используются некоторые модификаторы, в частности, модификаторы доступа):

```
<тип> <имя метода>([<список аргументов>]) <тело метода>
```

`<тип>` – это тип значения, которое возвращает метод. Допустимо использование любого примитивного или пользовательского типа. В C# формально не существует процедур – любой метод является функцией, возвращающей значение. Для «процедуры» в качестве типа указывается специальное ключевое слово `void`. `<имя метода>` – любой допустимый идентификатор, уникальный в описываемом контексте. После имени метода следует пара круглых скобок, в которых указывается список формальных параметров метода (если он не пуст).

Список формальных параметров метода – это набор элементов, разделенных запятыми. Каждый элемент списка имеет следующий формат:

```
[<модификатор>] <тип> <имя формального параметра>
```

Существуют четыре вида параметров, которые специфицируются модификатором:

1. *Параметры-значения* – объявляются без модификатора;
2. *Параметры, передаваемые по ссылке* – используют модификатор `ref`;
3. *Выходные параметры* – объявляются с модификатором `out`;
4. *Параметры-списки* – применяется модификатор `params`.

Параметры, передаваемые по ссылке и по значению, ведут себя аналогично тому, как это происходит в других языках программирования. *Выходные параметры* подобны ссылочным – при работе с ними в теле метода не создается копия фактического параметра. Компилятор отслеживает, чтобы в теле метода выходным параметрам обязательно было присвоено какое-либо значение.

Параметры-списки позволяют передать в метод любое количество фактических параметров. Метод может иметь не более одного параметра-списка, который обязательно должен быть последним в списке формальных параметров.

Библиотека БГУИР

Тип параметра-списка объявляется как тип-массив, и работа с таким параметром происходит в методе как с массивом. Каждый фактический параметр из передаваемого в метод списка ведет себя как параметр, переданный по значению.

Для выхода из метода служит оператор `return`. Если тип возвращаемого методом значения не `void`, то после `return` обязательно указывается возвращаемое методом значение (тип этого значения и тип метода должны совпадать). Кроме этого, инструкция `return` должна встретиться в таком методе во всех ветвях кода по крайней мере один раз.

Рассмотрим несколько примеров объявления методов.

1. Простейшее объявление метода-процедуры без параметров:

```
void SayHello() {
    Console.WriteLine("Hello!");
}
```

2. Метод-функция без аргументов, возвращающая целое значение:

```
int SayInt() {
    Console.WriteLine("Hello!");
    return 5;
}
```

3. Функция `Add()` выполняет сложение двух аргументов, передаваемых как параметры-значения:

```
int Add(int a, int b) {
    return a + b;
}
```

4. Функция `ReturnTwo()` возвращает `10` как результат своей работы, кроме этого значение параметра `a` устанавливается равным `100`:

```
int ReturnTwo(out int a) {
    a = 100;
    return 10;
}
```

4. Метод `PrintList()` использует параметр-список:

```
void PrintList(params int[] List) {
    foreach(int i in List)
        Console.WriteLine(i);
}
```

Метод `PrintList()` можно вызвать несколькими способами. Можно передать методу произвольное количество аргументов целого типа или массив целых значений:

```
//передаем два аргумента
PrintList(10,20);
//а теперь передаем четыре аргумента
PrintList(1, 2, 3, 4);
//создаем и передаем массив целых чисел
PrintList(new int[] {10, 20, 30, 40});
```

Библиотека БГУИР

```
//а можем вообще ничего не передавать
PrintList();
```

При вызове методов на месте формальных параметров помещаются фактические, совпадающие с формальными по типу или приводимые к этому типу. Если при описании параметра использовались модификаторы ref или out, то они должны быть указаны и при вызове. Кроме этого, фактические параметры с такими модификаторами должны быть представлены переменными, а не литералами или выражениями.

Значение, возвращаемое методом-функцией, может использоваться в выражениях, а может быть проигнорировано:

```
SayInt(); //хотя это функция, результат ее работы игнорируется
```

C# позволяет использовать *перегрузку методов* в пользовательских типах. Перегруженные методы имеют одинаковое имя, но разную сигнатуру. *Сигнатура* – это набор из модификаторов и типов списка формальных параметров. В языке C# считается, что сигнатура включает модификаторы ref и out, но не включает модификатор params:

```
//Данный код не компилируется - методы Foo различить нельзя!
void Foo(params int[] a) {. . .}
void Foo(int[] a) {. . .}

//Следующий фрагмент кода корректен
void Foo(out int a) {. . .}
void Foo(ref int a) {. . .}
```

Тип метода также не является частью сигнатуры. Специальных ключевых слов для определения перегруженных методов указывать не требуется. При вызове одного из перегруженных методов компилятор выбирает подходящий метод по сигнатуре.

10. МАССИВЫ В C#

Объявление массива в языке C# схоже с объявлением переменной, но после указания типа размещается пара квадратных скобок – признак массива:

```
int[] Data;
```

Массив является ссылочным типом, поэтому перед началом работы любой массив должен быть создан в памяти. Для этого используется ключевое слово new, после которого указывается тип массива и в квадратных скобках – количество элементов массива.

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

Созданный массив автоматически заполняется значениями по умолчанию своего базового типа. Создание массива можно совместить с его объявлением:

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

Библиотека БГУИР

Для доступа к элементу массива указывается имя массива и индекс в квадратных скобках: `Data[0] = 10`.

Элементы массива нумеруются с нуля, в C# не предусмотрено синтаксических конструкций для указания особого значения нижней границы массива.

В языке C# существует способ инициализации массива значениями при создании. Для этого используется список элементов в фигурных скобках. Инициализация может быть выполнена в *развернутой* и *короткой форме*, которые эквивалентны:

```
int[] Data = new int[10] {1, 2, 3, 5, 7, 11, 13, 17, 19, 23};
int[] Data = {1, 2, 3, 5, 7, 11, 13, 17, 19, 23};
```

Если инициализация выполняется в развернутой форме, то количество элементов в фигурных скобках должно быть равно числу, указанному в квадратных скобках.

При необходимости можно объявить массивы, имеющие несколько размерностей. Для этого в квадратных скобках после типа массива помещают запятые, «разделяющие» размерности:

```
// объявлен двумерный массив D
int[,] D;
// создаем массив D так:
D = new int[10,2];

// объявим трехмерный Cube и создадим его
int[,,] Cube = new int[3,2,5];
// установим элемент массива Cube:
Cube[1,1,0] = 1000;

// объявим маленький двумерный массив и инициализируем его
int[,] C = new int[2,4] {
                         {1, 2, 3, 4},
                         {10, 20, 30, 40}
                        };

// то же самое, немного короче:
int[,] C = {{1, 2, 3, 4}, {10, 20, 30, 40}};
```

В приведенных примерах объявлялись массивы из нескольких размерностей. Такие массивы всегда являются прямоугольными. Можно объявить *массив массивов*, используя следующий синтаксис:

```
int[][] Table;             // Table – массив одномерных массивов
Table = new int[3][];      // в Table будет 3 одномерных массива
Table[0] = new int[2];     // в первом будет 2 элемента
Table[1] = new int[20];    // во втором – 20 элементов
Table[2] = new int[12];    // в третьем – 12 элементов
// а вот так мы работаем с элементами массива Table:
Table[1][3] = 1000;
// совместим объявление и инициализацию массива массивов
int[][] T = {
```

Библиотека БГУИР

```
                new int[2] {10, 20},
                new int[3] {1, 2, 3}
               };
```

При работе с массивами можно использовать цикл foreach, перебирающий все элементы. В следующем фрагменте производится суммирование элементов массива Data:

```
int[] Data = {1,3,5,7,9};
int Sum = 0;
foreach(int element in Data)
    Sum += element;
```

В цикле foreach возможно перемещение по массиву в одном направлении – от начала к концу, при этом попытки присвоить значение элементу массива игнорируются.

В качестве примера работы с массивами рассмотрим программу, выполняющую сортировку массива целых чисел.

```
using System;
class MainClass
{
    public static void Main()
    {
        Console.Write("Введите число элементов: ");
        int Size = Int32.Parse(Console.ReadLine());
        int[] M = new int[Size];

        for (int i = 0; i < Size; i++) {
            Console.Write("Введите {0} элемент массива: ", i);
            M[i] = Int32.Parse(Console.ReadLine());
        }

        Console.WriteLine("Исходный массив:");
        foreach(int i in M) Console.Write("{0,6}", i);
        Console.WriteLine();

        for(int i = 0; i < Size-1; i++)
            for(int j = i+1; j < Size; j++) {
                if (M[i] > M[j]) {
                    int dop = M[i];
                    M[i] = M[j];
                    M[j] = dop;
                }
            }
        Console.WriteLine("Отсортированный массив:");
        foreach(int i in M) Console.Write("{0,6}", i);
    }
}
```

Библиотека БГУИР

Все массивы в .NET Framework могут рассматриваться как классы, являющиеся потомками класса `System.Array`. В табл. 3 описаны основные методы и свойства класса `System.Array`.

Таблица 3

Элементы класса `System.Array`

Имя элемента	Описание
1	2
`Rank`	Свойство только для чтения, возвращает размерность массива
`Length`	Свойство только для чтения, возвращает число элементов массива
`GetLength()`	Метод возвращает число элементов в указанном измерении
`GetLowerBound()`	Метод возвращает нижнюю границу для указанного измерения
`GetUpperBound()`	Метод возвращает верхнюю границу для указанного измерения
`GetValue()`	Метод возвращает значение элемента с указанными индексами
`SetValue()`	Метод устанавливает значение элемента с указанными индексами (значение – первый аргумент)
`Sort()`	Статический метод, который сортирует массив, переданный в качестве параметра. Тип элемента массива должен реализовывать интерфейс `IComparable`
`BinarySearch()`	Статический метод поиска элемента в отсортированном массиве. Тип элемента массива должен реализовывать интерфейс `IComparable`
`IndexOf()`	Статический метод, возвращает индекс первого вхождения своего аргумента в одномерный массив или `-1`, если элемента в массиве нет
`LastIndexOf()`	Статический метод. Возвращает индекс последнего вхождения своего аргумента в одномерный массив или `-1`, если элемента в массиве нет
`Reverse()`	Статический метод, меняет порядок элементов в одномерном массиве или его части на противоположный
`Copy()`	Статический метод. Копирует раздел одного массива в другой массив, выполняя приведение типов
`Clear()`	Статический метод. Устанавливает для диапазона элементов массива значение по умолчанию для типов элементов
`CreateInstance()`	Статический метод. Динамически создает экземпляр массива любого типа, размерности и длины

Теперь рассмотрим примеры использования описанных методов и свойств. В примерах выводимые данные записаны как комментарии. Вначале – использование нескольких простых элементов `System.Array`:

```
int[,] M = {{1, 3, 5}, {10, 20, 30}};
Console.WriteLine(M.Rank);                      // 2
Console.WriteLine(M.Length);                    // 6
Console.WriteLine(M.GetLowerBound(0));          // 0
Console.WriteLine(M.GetUpperBound(1));          // 2
```

Продемонстрируем сортировку и поиск в одномерном массиве:

Библиотека БГУИР

```
int[] M = {1, -3, 5, 10, 2, 5, 30};
Console.WriteLine(Array.IndexOf(M, 5)); );          //2
Console.WriteLine(Array.LastIndexOf(M, 5));         //5

Array.Reverse(M);
foreach(int a in M)
    Console.WriteLine(a);    //30, 5, 2, 10, 5, -3, 1

Array.Sort(M);
foreach(int a in M)
    Console.WriteLine(a);    //-3, 1, 2, 5, 5, 10, 30

Console.WriteLine(Array.BinarySearch(M, 10));      //5
```

Опишем процесс динамического создания массива. Данный способ позволяет задать для массивов произвольные нижние и верхние границы. Допустим, нам необходим двумерный массив из элементов decimal, первая размерность которого представляет годы в диапазоне от 1995 до 2004, а вторая – кварталы в диапазоне от 1 до 4. Следующий код осуществляет создание массива и обращение к элементу массива:

```
//Назначение этих массивов понятно из их названий
int[] LowerBounds = {1995, 1};
int[] Lengths = {10, 4};

//"Заготовка" для будущего массива
decimal[,] Target;
Target = (decimal[,])Array.CreateInstance(typeof(decimal),
                                         Lengths,LowerBounds);

//Пример обращения к элементу
Target[2000, 1] = 10.3M;
```

Допустимо было написать код для создания массива без приведения типов. Однако в этом случае для установки и чтения элементов необходимо было бы использовать методы SetValue() и GetValue():

```
Array Target;
Target = Array.CreateInstance(typeof(decimal), Lengths,
                              LowerBounds);
Target.SetValue(10.3M, 2000, 1);
Console.WriteLine(Target.GetValue(2000, 1));
```

Работа с элементами массива, созданного при помощи CreateInstance(), происходит медленнее, чем работа с «обычным» массивом.

11. РАБОТА С СИМВОЛАМИ И СТРОКАМИ В C#

Для представления отдельных символов в языке C# применяется тип char, основанный на структуре System.Char и использующий двухбайтную кодировку Unicode представления символов. Статические методы структуры System.Char представлены в табл. 4.

Библиотека БГУИР

Таблица 4

Статические методы `System.Char`

Имя метода	Описание
1	2
`GetNumericValue()`	Возвращает численное значение символа, если он является цифрой, и `-1` в противном случае
`GetUnicodeCategory()`	Метод возвращает Unicode-категорию символа
`IsControl()`	Возвращает `true`, если символ является управляющим
`IsDigit()`	Возвращает `true`, если символ является десятичной цифрой
`IsLetter()`	Возвращает `true`, если символ является буквой
`IsLetterOrDigit()`	Возвращает `true`, если символ является буквой или цифрой
`IsLower()`	Возвращает `true`, если символ задан в нижнем регистре
`IsNumber()`	Возвращает `true`, если символ является десятичной или шестнадцатиричной цифрой
`IsPunctuation()`	Возвращает `true`, если символ является знаком препинания
`IsSeparator()`	Возвращает `true`, если символ является разделителем
`IsSurrogate()`	Некоторые символы Unicode представляются двумя 16-битными «суррогатными» символами. Метод возвращает `true`, если символ является суррогатным
`IsUpper()`	Возвращает `true`, если символ задан в верхнем регистре
`IsWhiteSpace()`	Возвращает `true`, если символ является «белым пробелом». К белым пробелам, помимо пробела, относятся и другие символы, например, символ конца строки и символ перевода каретки
`Parse()`	Преобразует строку в символ. Естественно, строка должна состоять из одного символа, иначе возникнет ошибка
`ToLower()`	Приводит символ к нижнему регистру
`ToUpper()`	Приводит символ к верхнему регистру

Большинство статических методов перегружены. Они могут применяться как к отдельному символу, так и к строке, для которой указывается номер символа для применения метода.

Из экземплярных методов `System.Char` стоит отметить метод `CompareTo()`, позволяющий проводить сравнение символов. Он отличается от метода `Equals()` тем, что для несовпадающих символов выдает «расстояние» между символами в соответствии с их упорядоченностью в кодировке Unicode.

Основным типом при работе со строками в C# является тип `string`, задающий строки переменной длины. Тип `string` относится к ссылочным типам. Объекты класса `string` объявляются как все прочие объекты простых типов – с явным или неявным вызовом конструктора класса. Чаще всего, при объявлении строковой переменной конструктор явно не вызывается, а инициализация задается строковым литералом. Но у класса `string` достаточно много конструкторов. Они позволяют сконструировать строку из:

Библиотека БГУИР

- символа, повторенного заданное число раз;
- массива символов `char[]`;
- части массива символов.

Над строками определены следующие операции: присваивание (=), операции проверки эквивалентности (== и `!=`), *конкатенация* или сцепление строк (+), взятие индекса (`[]`).

Операция присваивания строк имеет важную особенность. Поскольку `string` – это ссылочный тип, то в результате присваивания создается ссылка на константную строку, хранимую в динамической памяти. С одной и той же строковой константой в динамической памяти может быть связано несколько переменных. Но когда одна из переменных получает новое значение, она связывается с новым константным объектом в динамической памяти. Остальные переменные сохраняют свои связи. Для программиста это означает, что семантика присваивания строк аналогична семантике присваивания структурных типов.

В отличие от других ссылочных типов операции, проверяющие эквивалентность строк, сравнивают значения строк, а не ссылки. Эти операции выполняются как над структурными типами.

Возможность взятия индекса при работе со строками отражает тот факт, что строку можно рассматривать как массив и получать каждый ее символ. Внимание: символ строки доступен только для чтения, но не для записи.

В языке C# существует понятие *неизменяемый класс* (*immutable class*). Для такого класса невозможно изменить значение объекта при вызове его методов. К неизменяемым классам относится и класс `System.String`. Ни один из методов этого класса не меняет значения существующих объектов. Конечно, некоторые из методов создают новые значения и возвращают в качестве результата новые строки. Рассмотрим статические методы и свойства класса `System.String`.

Таблица 5

Статические элементы класса `System.String`

Имя элемента	**Описание**
`Empty`	Возвращается пустая строка. Свойство со статусом `readonly`
`Compare()`	Сравнение двух строк. Реализации метода позволяют сравнивать как строки, так и подстроки. При этом можно учитывать регистр, особенности форматирования дат, чисел и т.д.
`CompareOrdinal()`	Сравнение двух строк. Реализации метода позволяют сравнивать как строки, так и подстроки. Сравниваются коды символов
`Concat()`	Конкатенация строк, метод допускает сцепление произвольного числа строк
`Copy()`	Создается копия строки
`Format()`	Выполняет форматирование строки в соответствии с заданными спецификациями формата
`Join()`	Конкатенация массива строк в единую строку. При конкатенации между элементами массива вставляются разделители. Операция, заданная методом `Join()`, является обратной к операции, заданной экземплярным методом `Split()`

Библиотека БГУИР

Из описанных статических методов подробно рассмотрим метод Join() и «парный» ему экземплярный метод Split(). Метод Split() позволяет осуществить разбор текста на элементы. Статический метод Join() выполняет обратную операцию, собирая строку из элементов.

Метод Split() перегружен. Наиболее часто используемая реализация этого метода имеет следующий синтаксис:

```
public string[] Split(params char[])
```

На вход методу Split() передается один или несколько символов, интерпретируемых как разделители. Объект string, вызвавший метод, разделяется на подстроки, ограниченные этими разделителями. Из этих подстрок создается массив, возвращаемый в качестве результата метода.

Синтаксис статического метода Join() таков:

```
public static string Join(string delimiters, string[] items)
```

В качестве результата метод возвращает строку, полученную конкатенацией элементов массива items, между которыми вставляется строка разделителей delimiters. Как правило, строка delimiters состоит из одного символа.

В следующем примере строка разбивается на отдельные слова, затем производится обратная сборка текста:

```
string txt = "А это пшеница, "
             + "которая в темном чулане хранится, "
             + "в доме, который построил Джек!";
Console.WriteLine(txt);

string[] SimpleSentences, Words;

// делим сложное предложение на простые
SimpleSentences = txt.Split(',');
for(int i = 0;i < SimpleSentences.Length; i++)
    Console.WriteLine(SimpleSentences[i]);

// собираем сложное предложение
string txtjoin = string.Join(",",SimpleSentences);

// делим сложное предложение на слова
Words = txt.Split(',', ' ');
for(int i = 0;i < Words.Length; i++)
    Console.WriteLine("Words[{0}]= {1}",i, Words[i]);
```

Сводка экземплярных методов класса System.String, приведенная в таблице 6, дает достаточно полную картину широких возможностей, имеющихся при работе со строками в C#. Следует помнить, что класс string является неизменяемым. Поэтому Replace(), Insert() и другие методы представляют собой функции, возвращающие новую строку в качестве результата.

Библиотека БГУИР

Таблица 6

Экземплярные методы класса `System.String`

Имя метода	Описание
`Insert()`	Вставляет подстроку в заданную позицию
`Remove()`	Удаляет подстроку в заданной позиции
`Replace()`	Заменяет подстроку в заданной позиции на новую подстроку
`Substring()`	Выделяет подстроку в заданной позиции
`IndexOf()`, `IndexOfAny()`, `LastIndexOf()`, `LastIndexOfAny()`	Определяются индексы первого и последнего вхождения заданной подстроки или любого символа из заданного набора
`StartsWith()`, `EndsWith()`	Возвращается `true` или `false`, в зависимости от того, начинается или заканчивается строка заданной подстрокой
`PadLeft()`, `PadRight()`	Выполняет набивку нужным числом пробелов в начале и в конце строки
`Trim()`, `TrimStart()`, `TrimEnd()`	Удаляются пробелы в начале и в конце строки, или только с одного ее конца
`ToCharArray()`	Преобразование строки в массив символов

В пространсве имен `System.Text` содержится класс `StringBuilder`. Этот класс также предназначен для работы со строками, но он принадлежит к изменяемым классам. Если в программе планируется активно изменять и анализировать строки, рекомендуется использовать именно объекты `StringBuilder`. Это позволит получить существенный (в разы) выйгрыш в производительности.

12. СИНТАКСИС ОБЪЯВЛЕНИЯ КЛАССА, ПОЛЯ И МЕТОДЫ КЛАССА

Класс является основным пользовательским типом в языке C#. Синтаксис объявления класса:

```
class <имя класса>
    [<члены класса>]
```

Здесь `<имя класса>` – любой уникальный идентификатор, `<члены класса>` объединены в программный блок. Допустимы следующие члены класса.

1. Поле. Поля класса описываются как обычные переменные, возможно с указанием модификатора доступа. Если для поля не указан модификатор доступа, то по умолчанию подразумевается модификатор `private`. Полям класса можно придавать начальные значения.

```
class CSomeClass {
    int Field1;
    private int Field2 = 10;
    public string Field3;
    . . .
}
```

2. Константа. Объявление константы обычно используется для того, чтобы сделать текст программы более читабельным. Модификатор доступа к константам по умолчанию – `private`. Если объявлена открытая (`public` или `in-`

Библиотека БГУИР

ternal) константа, то для ее использования вне класса можно указывать как имя объекта, так и имя класса.

3. Метод. Методы описывают функциональность класса. Код методов записывается непосредственно в теле класса. Модификатором доступа для методов по умолчанию является private.

4. Свойство. Свойства класса призваны предоставить защищенный доступ к полям. Подробнее синтаксис и применение свойств обсуждаются ниже.

5. Индексатор. Индексатор – это свойство-коллекция, отдельный элемент которого доступен по индексу.

6. Конструктор. Задача конструктора – начальная инициализация объекта или класса.

7. Деструктор. Деструктор класса служит для уничтожения объектов класса. Так как язык C# является языком с автоматической сборкой мусора, в явном вызове деструкторов нет необходимости. Обычно они содержат некий *завершающий код* для объекта.

8. Событие. События представляют собой механизм рассылки уведомлений различным объектам.

9. Операция. Язык C# допускает перегрузку некоторых операций для объектов класса.

10. Вложенный пользовательский тип. Описание класса может содержать описание другого пользовательского типа – класса, структуры, интерфейса, делегата. Обычно вложенные типы выполняют вспомогательные функции и явно вне основного типа не используются.

При описании класса допустимо указать для него следующие модификаторы доступа – public или internal (применяется по умолчанию). Если класс является элементом другого пользовательского типа, то его можно объявить с любым модификатором доступа. Заметим, что если класс объявлен с модификатором internal, то его public-элементы не видны за пределами сборки.

Переменная класса – *объект* – объявляется как обычная переменная:

```
<имя класса> <имя объекта>;
```

Так как класс – ссылочный тип, то объекты должны быть инициализированы до непосредственного использования. Для инициализации объекта используется оператор new, совмещенный с вызовом конструктора класса. Если конструктор не описывался, используется предопределенный конструктор без параметров с именем класса:

```
<имя объекта> = new <имя класса>();
```

Инициализацию объекта можно совместить с его объявлением:

```
<имя класса> <имя объекта> = new <имя класса>();
```

Доступ к членам класса через объект осуществляется по синтаксису `<имя объекта>.<имя члена>`.

Приведем пример описания класса, который содержит два поля:

Библиотека БГУИР

```
class CPet {
    public int Age;
    public string Name;
}
```

Проиллюстрируем описание и использование объектов класса CPet:

```
CPet Dog;                  //Просто объявление
CPet Cat = new CPet();     //Объявление с инициализацией
Dog = new CPet();          //Инициализация объекта
Dog.Age = 10;              //Доступ к полям
Cat.Name = "123Y";
```

Добавим в класс CPet методы. Заметим, что для устранения конфликта имен «имя члена класса = имя параметра метода» возможно использование ключевого слова this – это ссылка на текущий объект класса:

```
class CPet {
    public int Age;
    public string Name;
    void SetAge(int Age) {
        this.Age = Age;
    }
    string GetName() {
        return Name;
    }
}
```

Поля и методы, которые рассматривались в предыдущих примерах, были связаны с объектом класса. Подобные (связанные с объектом) элементы класса называются *экземплярными*. *Статические* поля, методы и свойства предназначены для работы с классом, а не с объектом. Статические поля хранят информацию, общую для всех объектов, статические методы работают со статическими полями. Для того чтобы объявить статический член класса, используется ключевое слово static:

```
class CAccount {
    public static double Tax = 0.1;
    public static double getTax() {
        return Tax * 100;
    }
}
```

Для вызова статических элементов требуется использовать имя класса:

```
CAccount.Tax = 0.3;
Console.WriteLine(CAccount.getTax());
```

В качестве одно из примеров использования статических элементов класса опишем класс Singleton. Особенностью этого класса является то, что в приложении можно создать только один объект данного класса.

```
class Singleton {
    static Singleton Instance;
```

Библиотека БГУИР

```
        public string Info;
        private Singleton() {}
        public static Singleton Create() {
            if (Instance == null) Instance = new Singleton();
            return Instance;
        }
    }
```

Так как в классе `Singleton` конструктор объявлен как закрытый, то единственный способ создать объект этого класса – вызвать функцию `Create()`. Логика работы этой функции организована так, что `Create()` всегда возвращает ссылку на один и тот же объект. Обратите внимание: поле `Instance` хранит ссылку на объект и описано как статическое. Это сделано для того, чтобы с ним можно было работать в статическом методе `Create()`. Далее приводится пример кода, использующего класс `Singleton`:

```
Singleton A = Singleton.Create();
Singleton B = Singleton.Create();
A.Info = "Information";
Console.WriteLine(B.Info);
```

Объекты `A` и `B` представляют собой одну сущность, то есть на консоль выведется строка `"Information"`.

13. СВОЙСТВА И ИНДЕКСАТОРЫ

Свойства класса призваны предоставить защищенный доступ к полям. Как и в большинстве объектно-ориентированных языков, в C# непосредственная работа с полями не приветствуется. Поля класса обычно объявляются как `private`-элементы, а для доступа к ним используются свойства.

Рассмотрим синтаксис описания свойства:

```
<тип свойства> <имя свойства> {
    get {<блок кода>}
    set {<блок кода>}
}
```

Как видно, синтаксис описания свойства напоминает синтаксис описания обычного поля. Тип свойства обычно совпадает с типом того поля, для обслуживания которого свойство создается. У свойства присутствует специальный блок, содержащий методы для доступа к свойству. Данный блок состоит из get-части и set-части. Одна из частей может отсутствовать, так получается *свойство только для чтения* или *свойство только для записи*. Get-часть отвечает за возвращаемое свойством значение и работает как функция (обязательно наличие в блоке кода get-части оператора `return`). Set-часть работает как метод-процедура, устанавливающий значение свойства. Считается, что параметр, передаваемый в set-часть, имеет специальное имя `value`.

Добавим свойства в класс `CPet`, закрыв для использования поля:

```
class CPet {
    private int age;
```

Библиотека БГУИР

```
    private string name;

    public int Age {
        get {
            return age;
        }
        set {
            // проверка корректности значения
            age = value < 0 ? age = 0 : age = value;
        }
    }

    public string Name {
        get {
            return "My name is " + name;
        }
        set { name = value; }
    }
}
```

Свойства транслируются при компиляции в вызовы методов. В скомпилированный код класса добавляются методы со специальными именами `get_Name` и `set_Name`, где *`Name`* – это имя свойства. Побочным эффектом данного преобразования является тот факт, что пользовательские методы с данными именами допустимы в классе, только если они имеют сигнатуру, отличающуюся от методов, соответствующих свойству.

В языках программирования VB.NET и Object Pascal наряду с обычными свойствами существуют свойства-массивы. Роль свойств-массивов в C# выполняют *индексаторы*. При помощи индексаторов осуществляется доступ к коллекции данных, содержащихся в объекте класса, с использованием привычного синтаксиса для доступа к элементам массивы – пары квадратных скобок.

Объявление индексатора напоминает объявление свойства:

```
<тип индексатора> this[<аргументы>] { <get и set блоки> }
```

Аргументы индексатора служат для описания типа и имен индексов, применяемых для доступа к данным объекта. Обычно используется индексы целого типа, хотя это и не является обязательным. Аргументы индексатора доступны в блоках `get` и `set`. Если индексатор имеет более одного аргумента, то аргументы в описании индексатора перечисляются через запятую.

Рассмотрим пример класса, содержащего индексаторы. Пусть данный класс описывает студента с набором оценок:

```
class Student {
    private int[] marks = new int[5];
    public string Name;

    public int this[int i] {
        get {
            if ((i >= 1) && (i <= 5)) return marks[i-1];
```


Библиотека БГУИР

```
                else return 0;
            }
            set {
                if ((i >= 1) && (i <= 5) && (value <= 10))
                    marks[i-1] = value;
            }
        }
    }
```

Данный класс и индексатор можно использовать следующим образом:

```
Student Ivan = new Student();
Ivan[1] = 8;
Ivan[3] = 4;
for(int i = 1; i <= 5; i++)
    Console.WriteLine(Ivan[i]);
```

Индексаторы всегда работают как *свойства по умолчанию*. Это значит, что в одном классе нельзя объявить два индексатора, у которых совпадают типы аргументов. Однако можно объявить в одном классе индексаторы, у которых аргументы имеют разный тип или количество аргументов различается.

Если свойства транслировались компилятором в методы со специальными именами get_*Name* и set_*Name*, то индексаторы транслируются в методы с именами get_Item и set_Item. Изменить имена методов для индексаторов можно, используя специальный атрибут:

```
class Student {
    . . .
     // методы будут называться get_ Mark и set_ Mark
    [System.Runtime.CompilerServices.IndexerName("Mark")]
    public int this[int i] {. . .}
}
```

14. КОНСТРУКТОРЫ КЛАССА И ЖИЗНЕННЫЙ ЦИКЛ ОБЪЕКТА

Конструкторы используются для начальной инициализации объектов. Конструкторы похожи на методы, но в отличие от методов конструкторы не наследуются, и вызов конструктора в виде <имя объекта>.<имя конструктора> после создания объекта запрещен. Различают несколько видов конструкторов – конструкторы по умолчанию, пользовательские конструкторы, статические конструкторы.

Конструктор по умолчанию автоматически создается компилятором, если программист не описал в классе собственный конструктор. Конструктор по умолчанию – это всегда конструктор без параметров. Можно считать, что конструктор по умолчанию содержит код начальной инициализации полей.

```
//Класс CPet не содержит конструктора
class CPet {
    public int Age;
    public string Name = "I'm a pet";
}
. . .
```

Библиотека БГУИР

```
CPet Dog = new CPet();        //Вызов конструктора по умолчанию
Console.WriteLine(Dog.Age);   //Выводит 0
Console.WriteLine(Dog.Name);  //Выводит I'm a pet
```

Пользовательский конструктор описывается в классе как метод с именем, совпадающим с именем класса. Тип возвращаемого значения для конструктора не указывается (отсутствует любое указание на тип, даже `void`). Пользовательский конструктор может получать параметры, необходимые для инициализации объекта. Класс может содержать несколько пользовательских конструкторов, однако они обязаны различаться сигнатурой.

Опишем два пользовательских конструктора в классе `CPet`:

```
class CPet {
    public int Age;
    public string Name = "I'm a pet";
    public CPet() {
        Age = 0;
        Name = "CPet";
    }
    public CPet(int x, string y) {
        Age = x;
        Name = y;
    }
}
```

При вызове пользовательского конструктора его фактические параметры указываются после имени конструктора в скобках. Если конструкторов у класса несколько, подходящий выбирается по сигнатуре:

```
CPet Cat = new CPet();              //Вызов первого конструктора
CPet Dog = new CPet(5, "Buddy");    //Вызов второго конструктора
```

Пользовательские конструкторы могут применяться для начальной инициализации `readonly`-полей. Напомним, что такие поля ведут себя как константы, однако могут иметь произвольный тип. Таким образом, `readonly`-поля доступны для записи, но только в конструкторе.

Обратите внимание: если в классе определен хотя бы один пользовательский конструктор, конструктор по умолчанию уже не создается. Если мы изменим класс `CPet`, оставив там только пользовательский конструктор с параметрами, то строка `CPet Cat = new CPet()` будет вызывать ошибку компиляции. Код, который выполняет присваивание полям значений по умолчанию, добавляется компилятором автоматически в начало любого пользовательского конструктора.

Конструктор класса может вызывать другой конструктор того же класса, но только в начале своей работы. Для этого при описании конструктора используется синтаксис, аналогичный приведенному в следующем примере:

```
public CPet() : this(10, "CPet") { . . . }
```

Библиотека БГУИР

Статические конструкторы используются для начальной инициализации статических полей класса. Статический конструктор объявляется с модификатором `static` и без параметров. Область видимости у статических конструкторов не указывается.

```
class CAccount {
    public static double Tax;
    static CAccount(){
        Tax = 0.1;
    }
}
```

Статические конструкторы вызываются не программистом, а общеязыковой средой исполнения CLR в следующих случаях:

- перед созданием первого объекта класса или при первом обращении к элементу класса, не унаследованному от предка;
- в любое время перед первым обращением к статическому полю, не унаследованному от предка.

В теле статического конструктора возможна работа только со статическими полями и методами класса, статические конструкторы не могут вызывать экземплярные конструкторы класса.

Как уже отмечалось выше, все пользовательские типы в языке C# можно разделить на ссылочные и структурные. Переменные структурных типов создаются средой исполнения CLR в стеке. *Время жизни* (*lifetime*) переменных структурного типа обычно ограничено тем блоком кода, в котором они объявляются. Например, если переменная, соответствующая пользовательской структуре, объявлена в неком методе, то после выхода из метода память, занимаемая переменной, автоматически освободится.

Переменные ссылочного типа (объекты) размещаются в динамической памяти – «куче». Среда исполнения платформы .NET использует *управляемую кучу* (*managed heap*). Объясним значение этого понятия. Если при работе программы превышен некий порог расходования памяти, CLR запускает процесс, называемый *сборка мусора*. Среда исполнения отслеживает все используемые объекты и определяет реально занимаемую этими объектами память. После этого вся оставшаяся память освобождается, то есть помечается как свободная для использования. Освобождая память, CLR заново размещает «уцелевшие» объекты в куче, чтобы уменьшить ее фрагментацию. Ключевой особенностью сборки мусора является то, что она осуществляется средой исполнения автоматически и независимо от основного потока выполнения приложения.

Обсудим автоматическую сборку мусора с точки зрения программиста, разрабатывающего некий класс. С одной стороны, такой подход имеет свои преимущества. В частности, практически исключаются случайные утечки памяти, которые могут вызвать «забытые» объекты. С другой стороны, объект может захватывать некоторые особо ценные ресурсы (например, подключения к базе данных), которые требуется освободить сразу после того, как объект пе-

Библиотека БГУИР

рестает использоваться. В этой ситуации выходом является написание некого особого метода, который содержит код освобождения ресурсов.

Но как *гарантировать* освобождение ресурсов, даже если ссылка на объект была случайно утеряна? Класс `System.Object` содержит виртуальный метод `Finalize()` без параметров. Если пользовательский класс при работе резервирует некие ресурсы, он может переопределить метод `Finalize()` для их освобождения. Объекты классов, имеющих реализацию `Finalize()`, при сборке мусора обрабатываются особо. Когда CLR распознаёт, что уничтожаемый объект имеет собственную реализацию метода `Finalize()`, она откладывает уничтожение объекта. Через некоторое время в отдельном программном потоке происходит вызов метода `Finalize()` и фактическое уничтожение объекта.

Язык C# не позволяет явно переопределить в собственном классе метод `Finalize()`. Вместо переопределения `Finalize()` в классе описывается специальный метод, синтаксис которого напоминает синтаксис *деструктора* языка C++. Имя метода образовывается по правилу `~<имя класса>`, метод не имеет параметров и модификаторов доступа. Считается, что модификатор доступа «деструктора» – `protected`, следовательно, явно вызвать его у объекта нельзя.

Рассмотрим пример класса с «деструктором»:

```
class ClassWithDestructor {
    public string name;
    public ClassWithDestructor(string name) {
        this.name = name;
    }
    public void doSomething() {
        Console.WriteLine("I'm working...");
    }
    //Это деструктор
    ~ClassWithDestructor() {
        Console.WriteLine("Bye!");
    }
}
```

Приведем пример программы, использующей описанный класс:

```
class MainClass {
    public static void Main() {
        ClassWithDestructor A = new ClassWithDestructor("A");
        A.doSomething();
        // Сборка мусора запустится перед
        // завершением приложения
    }
}
```

Данная программа выводит следующие строки:

```
I'm working...
Bye!
```

Проблема с использованием метода-«деструктора» заключается в том, что момент вызова этого метода сложно отследить. Программист может описать в

Библиотека БГУИР

классе некий метод, который *следует* вызывать «вручную», когда объект больше не нужен. Для унификации данного решения платформа .NET предлагает интерфейс `IDisposable`, содержащий единственный метод `Dispose()`, куда помещается завершающий код работы с объектом. Класс, объекты которого требуется освобождать «вручную», реализовывает интерфейс `IDisposable`.

Изменим приведенный выше класс, реализовав в нем интерфейс `IDisposable`:

```
class ClassWithDestructor : IDisposable {
    public string name;
    public ClassWithDestructor(string name) {
        this.name = name;
    }
    public void doSomething() {
        Console.WriteLine("I'm working...");
    }
    // Реализуем метод "освобождения"
    public void Dispose() {
        Console.WriteLine("Dispose called for " + name);
    }

    ~ClassWithDestructor() {
        // Деструктор вызывает Dispose "на всякий случай"
        Dispose();
        Console.WriteLine("Bye!");
    }
}
```

C# имеет специальную *обрамляющую конструкцию* `using`, которая *гарантирует* вызов метода `Dispose()` для объектов, использующихся в своем блоке. Синтаксис данной конструкции:

```
using (<имя объекта или объявление и создание объектов>)
    <программный блок>
```

Изменим программу с классом `ClassWithDestructor`, поместив туда обрамляющую конструкцию `using`:

```
class MainClass {
    public static void Main() {
        using(ClassWithDestructor A =
                                   new ClassWithDestructor("A")) {
            A.doSomething();
            // компилятор поместит сюда вызов A.Dispose()
        }
    }
}
```

Что выведет на консоль данная программа? Ниже представлены выводимые строки с комментариями:

```
I'm working...           – это работа метода A.doSomething()
Dispose called for A     – вызывается Dispose() в конце using
```

Библиотека БГУИР

```
Dispose called for A         – эта и следующая строка являются
Bye!                         - результатом работы деструктора
```

Сборщик мусора представлен классом `System.GC`. Метод `Collect()` данного класса вызывает принудительную сборку мусора в программе и может быть вызван программистом. Не рекомендуется пользоваться методом `Collect()` часто, так как сборка мусор требует расхода ресурсов.

15. НАСЛЕДОВАНИЕ КЛАССОВ

Язык C# полностью поддерживает объектно-ориентированную концепцию наследования. Чтобы указать, что один класс является наследником другого, используется следующий синтаксис:

```
class <имя наследника> : <имя базового класса> {<тело класса>}
```

Наследник обладает всеми полями, методами и свойствами предка, однако элементы предка с модификатором `private` не доступны в наследнике. Конструкторы класса-предка не переносятся в класс-наследник.

При наследовании нельзя расширить область видимости класса: `internal`–класс может наследоваться от `public`–класса, но не наоборот.

Для обращения к методам непосредственного предка класс-наследник может использовать ключевое слово `base` в форме `base.<имя метода базового класса>`. Если конструктор наследника должен вызвать конструктор предка, то для этого также используется `base`:

```
<конструктор наследника>([<параметры>]): base([<параметры_2>])
```

Для конструкторов производного класса справедливо следующее замечание: конструктор должен вначале совершить вызов другого конструктора своего или базового класса. Если вызов конструктора базового класса отсутствует, компилятор автоматически подставляет в заголовок конструктора вызов `:base()`. Если в базовом классе нет конструктора без параметров, происходит ошибка компиляции.

Наследование от двух и более классов в C# запрещено.

Для классов можно указать два модификатора, связанных с наследованием. Модификатор `sealed` задает класс, от которого запрещено наследование. Модификатор `abstract` задает *абстрактный класс*, у которого обязательно должны быть наследники. Объект абстрактного класса создать нельзя, хотя статические члены такого класса можно вызвать, используя имя класса. Модификаторы наследования указываются непосредственно перед ключевым словом `class`:

```
sealed class FinishedClass { }
abstract class AbstractClass { }
```

Класс-наследник может дополнять базовый класс новыми методами, а также замещать методы базового класса. Для замещения достаточно указать в новом классе метод с прежним именем и, возможно, с новой сигнатурой:

```
class CPet {
```

Библиотека БГУИР

```
    public void Speak() {
        Console.WriteLine("I'm a pet");
    }
}

class CDog : CPet {
    public void Speak() {
        Console.WriteLine("I'm a dog");
    }
}
. . .
CPet Pet = new CPet();
CDog Dog = new CDog();
Pet.Speak();
Dog.Speak();
```

При компиляции данного фрагмента будет получено предупреждающее сообщение о том, что метод `CDog.Speak()` закрывает метод базового класса `CPet.Speak()`. Чтобы подчеркнуть, что метод класса-наследника замещает метод базового класса, используется ключевое слово `new`:

```
class CDog : CPet
{
    new public void Speak() { //Компиляция без предупреждений
        Console.WriteLine("I'm a dog");
    }
}
```

Ключевое слово `new` может размещаться как до, так и после модификаторов доступа для метода. Данное ключевое слово применимо и к полям класса.

Замещение методов класса не является полиморфным по умолчанию. Следующий фрагмент кода печатает две одинаковые строки:

```
CPet Pet, Dog;
Pet = new CPet();
Dog = new CDog();        // Допустимо по правилам присваивания
Pet.Speak();             // Печатает "I'm a pet"
Dog.Speak();             // Так же печатает "I'm a pet"
```

Для организации полиморфного вызова методов применяется пара ключевых слов `virtual` и `override`: `virtual` указывается для метода базового класса, который мы хотим сделать полиморфным, `override` – для методов производных классов. Эти методы должны совпадать по имени и сигнатуре с перекрываемым методом класса-предка.

```
class CPet {
    public virtual void Speak() {
        Console.WriteLine("I'm a pet");
    }
}

class CDog : CPet {
```

Библиотека БГУИР

```
        public override void Speak() {
            Console.WriteLine("I'm a dog");
        }
    }
    . . .
    CPet Pet, Dog;
    Pet = new CPet();
    Dog = new CDog();
    Pet.Speak();            // Печатает "I'm a pet"
    Dog.Speak();            // Теперь печатает "I'm a dog"
```

Если на некоторой стадии построения иерархии классов требуется запретить дальнейшее переопределение виртуального метода в производных классах, этот метод помечается ключевым словом `sealed`:

```
class CDog : CPet {
    public sealed override void Speak() { . . . }
```

Для методов абстрактных классов (классов с модификатором `abstract`) возможно задать модификатор `abstract`, который говорит о том, что метод не реализуется в классе, а должен обязательно переопределяться в наследнике.

```
abstract class AbstractClass
{
    //Реализации метода в классе нет
    public abstract void AbstractMethod();
}
```

Отметим, что наряду с виртуальными методами в C# можно описать виртуальные свойства (свойство транслируется в методы). Статические элементы класса не могут быть виртуальными.

16. ПЕРЕГРУЗКА ОПЕРАЦИЙ

Язык C# позволяет организовать для объектов пользовательского класса или структуры *перегрузку операций*. Могут быть перегружены унарные операции +, -, !, ~, ++, --, `true`, `false` и бинарные операции +, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=.

При перегрузке бинарной операции автоматически перегружается соответствующая операция с присваиванием (например, при перегрузке операции + перегрузится и операция +=). Некоторые операции могут быть перегружены только парами: == и !=, > и <, >= и <=, `true` и `false`.

Для перегрузки операций используется специальный статический метод, имя которого образовано из ключевого слова `operator` и знака операции. Количество формальных параметров метода зависит от типа операции: унарная операция требует одного параметра, бинарная – двух. Метод обязательно должен иметь модификатор доступа `public`.

Рассмотрим перегрузку операций на примере. Определим класс для представления комплексных чисел с перегруженной операцией сложения:

Библиотека БГУИР

```
class Complex {
    public double Re;
    public double Im;
    public Complex(double Re, double Im) {
        this.Re = Re;
        this.Im = Im;
    }
    public override string ToString() {
        return String.Format("Re = {0} Im = {1}", Re, Im);
    }

    public static Complex operator + (Complex A, Complex B) {
        return new Complex(A.Re + B.Re, A.Im + B.Im);
    }
}
```

Для объектов класса `Complex` возможно использование следующего кода:

```
Complex A = new Complex(10.0, 20.0);
Complex B = new Complex(-5.0, 10.0);
Console.WriteLine(A);          // Выводит Re = 10.0 Im = 20.0
Console.WriteLine(B);          // Выводит Re = -5.0 Im = 10.0
Console.WriteLine(A + B);      // Выводит Re =  5.0 Im = 30.0
```

Параметры метода перегрузки должны быть параметрами, передаваемыми по значению. Тип формальных параметров и тип возвращаемого значения метода перегрузки обычно совпадает с описываемым типом, хотя это и не обязательное условие. Более того, класс или структура могут содержать версии одной операции с разным типом формальных параметров. Однако не допускается существование двух версий метода перегрузки операции, различающихся только типом возвращаемого значения. Также класс не может содержать перегруженной операции, у которой ни один из формальных параметров не имеет типа класса.

Внесем некоторые изменения в класс `Complex`:

```
class Complex {
. . .
    public static Complex operator + (Complex A, Complex B) {
        return new Complex(A.Re + B.Re, A.Im + B.Im);
    }
    public static Complex operator + (Complex A, double B) {
        return new Complex(A.Re + B, A.Im + B);
    }
}
```

Новая перегруженная операция сложения позволяет прибавлять к комплексному числу вещественное число.

Любой класс может перегрузить операции `true` и `false`. Операции перегружаются парой, тип возвращаемого значения операций – булевский. Если в классе выполнена подобная перегрузка, объекты класса могут использоваться

Библиотека БГУИР

как условия в операторах условного перехода или циклов. При вычислении условий используется перегруженная версия операции true.

Рассмотрим следующий пример. Пусть в классе Complex перегружены операции true и false:

```
class Complex {
. . .
    public static bool operator true (Complex A) {
        return (A.Re > 0) || (A.Im > 0);
    }
    public static bool operator false (Complex A) {
        return (A.Re == 0) && (A.Im == 0);
    }
}
```

Теперь возможно написать такой код (обратите внимание на оператор if):

```
Complex A = new Complex(10.0, 20.0);
Complex B = new Complex(0, 0);
if (B)
    Console.WriteLine("Number is not zero");
else
    Console.WriteLine("Number is 0.0 + 0.0i");
```

Кроме перечисленных операций, любой класс может перегрузить операции для неявного и явного приведения типов. При этом используется следующий синтаксис:

```
public static implicit operator <целевой тип>(<привод. тип> <имя>)
public static explicit operator <целевой тип>(<привод. тип> <имя>)
```

Ключевое слово implicit используется при перегрузке неявного приведения типов, а ключевое слово explicit – при перегрузке операции явного приведения. Либо <целевой тип>, либо <приводимый тип> должены совпадать с типом того класса, где выполняется перегрузка операций.

Поместим две перегруженных операции приведения в класс Complex:

```
class Complex {
. . .
    public static implicit operator Complex (double a) {
        return new Complex(a, 0);
    }
    public static explicit operator double (Complex A) {
        return Math.Sqrt(A.Re * A.Re + A.Im * A.Im);
    }
}
```

Вот пример кода, использующего преобразование типов:

```
Complex A = new Complex(3.0, 4.0);
double x;
//Выполняем явное приведение типов
```

Библиотека БГУИР

```
x = (double) A;
Console.WriteLine(x);        //Выводит 5

double y = 10;
//Выполняем неявное приведение типов
A = y;
Console.WriteLine(A);        //Выводит Re = 10 Im = 0
```

17. ДЕЛЕГАТЫ

Делегат в языке C# исполняет роль указателя на метод. Делегат объявляется с использованием ключевого слова `delegate`. При этом указывается имя делегата и сигнатура инкапсулируемого метода. Модификаторы доступа при необходимости указываются перед ключевым словом `delegate`:

```
delegate double Function(double x);
public delegate void IntegerSub(int i);
```

Делегат – самостоятельный пользовательский тип, он может быть как вложен в другой пользовательский тип (класс, структуру), так и объявлен отдельно. Так как делегат – это пользовательский тип, то нельзя объявить два или более делегатов с одинаковыми именами, но разной сигнатурой.

После объявления делегата можно объявить переменные этого типа:

```
Function Y;
IntegerSub SomeSub;
```

Переменные делегата инициализируются конкретными адресами методов при использовании *конструктора делегата* с одним параметром – именем метода (или именем другого делегата). Если делегат инициализируется статическим методом, требуется указать имя класса и имя метода, для инициализации экземплярным методом указывается объект и имя метода. При этом метод должен обладать подходящей сигнатурой:

```
Y1 = new Function(ClassName.MyStaticFunction);
Y1 = new Function(Obj1.MyInstanceFunction);
Y2 = new Function(Y1);
```

После того как делегат инициализирован, инкапсулированный в нем метод вызывается, указывая параметры метода непосредственно после имени переменной-делегата:

```
Y1(0.5);
```

Приведем пример использования делегатов. Опишем класс, содержащий метод вывода массива целых чисел.

```
class ArrayPrint {
    public static void print(int[] A, PrintMethod P) {
        foreach(int element in A)
            P(element);
    }
}
```

Библиотека БГУИР

PrintMethod является делегатом, который определяет способ печати отдельного числа. Он описан следующим образом:

```
delegate void PrintMethod(int x);
```

Теперь можно написать класс, который работает с классом ArrayPrint и делегатом PrintMethod:

```
class MainClass {
    public static void ConsolePrint(int i) {
        Console.WriteLine(i);
    }

    public void FormatPrint(int i) {
        Console.WriteLine("Element is {0}", i);
    }

    public static void Main() {
      int[] A = {1, 20, 30};

      PrintMethod D = new PrintMethod(MainClass.ConsolePrint);
      ArrayPrint.print(A, D);

      MainClass C = new MainClass();
      D = new PrintMethod(C.FormatPrint);
      ArrayPrint.print(A, D);
    }
}
```

В результате работы данной программы на экран будут выведены следующие строки:

```
1
20
30
Element is 1
Element is 20
Element is 30
```

Обратите внимание, что в данном примере переменная D инициализировалась статическим методом, а затем методом объекта. Делегат не делает различий между экземплярными и статическими методами класса.

Ключевой особенностью делегатов является то, что они могут инкапсулировать не один метод, а несколько. Подобные делегаты называются *групповыми делегатами*. При вызове группового делегата срабатывает вся цепочка инкапсулированных в нем методов.

Групповой делегат объявляется таким же образом, как и обычный. Затем создается несколько объектов делегата, и все они связывается с некоторыми методами. После этого используются перегруженные версии операций + или += класса System.Delegate для объединения делегатов в один групповой делегат. Для объединения можно использовать статический метод Sys-

Библиотека БГУИР

tem.Delegate.Combine(), который получает в качестве параметров два объекта делегата (или массив объектов-делегатов) и возвращает групповой делегат, являющийся объединением параметров.

Модифицируем код из предыдущего примера следующим образом:

```
class MainClass {
. . .
    public static void Main() {
        int[] A = {1, 20, 30};
        PrintMethod first, second, result;
        first = new PrintMethod(MainClass.ConsolePrint);
        MainClass C = new MainClass();
        second = new PrintMethod(C.FormatPrint);
        result = first + second;
        ArrayPrint.print(A, result);
    }
}
```

Теперь результат работы программы выглядит следующим образом:

```
1
Element is 1
20
Element is 20
30
Element is 30
```

Если требуется удалить некий метод из цепочки группового делегата, то используются перегруженные операции - или -= (или метод System.Delegate.Remove()). Если из цепочки удаляют последний метод, результатом будет значение null. Следующий код удаляет метод first из цепочки группового делегата result:

```
result -= first;
```

Любой пользовательский делегат можно рассматривать как класс-наследник класса System.MulticastDelegate, который, в свою очередь, наследуется от класса System.Delegate. Именно на уровне класса System.Delegate определены перегрузки операций + и –, использовавшихся для создания групповых делегатов. Полезным также может оказаться экземплярный метод GetInvocationList(). Он возвращает массив объектов, составляющих цепочку вызова группового делегата.

18. СОБЫТИЯ

События представляют собой способ описания связи одного объекта с другими по действиям. Родственным концепции событий является понятие функции обратного вызова.

Работу с событиями можно условно разделить на три этапа:

- объявление события (*publishing*);
- регистрация получателя события (*subscribing*);

Библиотека БГУИР

• генерация события (*raising*).

Событие можно объявить в пределах класса, структуры или интерфейса. При объявлении события требуется указать делегат, описывающий процедуру обработки события. Синтаксис объявления события следующий:

```
event <имя делегата> <имя события>;
```

Ключевое слово event указывает на объявление события. Объявление события может предваряться модификаторами доступа.

Приведем пример класса с объявлением события:

```
//Объявление делегата для события
delegate void Proc(int val);

class CEventClass {
    int data;
    //Объявление события
    event Proc OnWrongData;
    . . .
}
```

Фактически, события являются полями типа делегатов. Объявление события транслируется компилятором в следующий набор объявлений в классе:

a. в классе объявляется private-поле с именем <имя события> и типом <имя делегата>;

b. в классе объявляются два метода с именами add_<имя события> и remove_<имя события> для добавления и удаления обработчиков события.

Методы для обслуживания события содержат код, добавляющий (add_*) или удаляющий (remove_*) процедуру обработки события в цепочку группового делегата, связанного с событием.

Для генерации события в требуемом месте кода помещается вызов в формате <имя события>(<фактические аргументы>). Предварительно можно проверить, назначен ли обработчик события. Генерация события может происходить в одном из методов того же класса, в котором объявлено событие. Генерировать в одном классе события других классов нельзя.

Приведем пример класса, содержащего объявление и генерацию события. Данный класс будет включать метод с целым параметром, устанавливающий значение поля класса. Если значение параметра отрицательно, генерируется событие, определенное в классе:

```
delegate void Proc(int val);

class CExampleClass {
    int field;

    public event Proc onErrorEvent;

    public void setField(int i){
        field = i;
```

Библиотека БГУИР

```
            if(i < 0) {
                if(onErrorEvent != null) onErrorEvent(i);
            }
        }
    }
```

Рассмотрим этап регистрации получателя события. Для того чтобы отреагировать на событие, его надо ассоциировать с *обработчиком события*. Обработчиком события может быть метод-процедура, совпадающий по типу с типом события (делегатом). Назначение и удаление обработчиков события выполняется при помощи перегруженных версий операторов += и -=. При этом в левой части указывается имя события, в правой части – объект делегата, созданного на основе требуемого метода-обработчика.

Используем предыдущий класс `CExampleClass` и продемонстрируем назначение и удаление обработчиков событий:

```
class MainClass {
    public static void firstReaction(int i) {
        Console.WriteLine("{0} is a bad value!", i);
    }

    public static void secondReaction(int i) {
        Console.WriteLine("Are you stupid?");
    }

    public static void Main() {
        CExampleClass c = new CExampleClass();
        c.setField(200);
        c.setField(-200);   // Нет обработчиков, нет и реакции
        // Если бы при генерации события в CExampleClass
        // отсутствовала проверка на null, то предыдущая
        // строка вызвала бы исключительную ситуацию

        // Назначаем обработчик
        c.onErrorEvent += new Proc(firstReaction);

        // Теперь будет вывод "-10 is a bad value!"
        c.setField(-10);

        // Назначаем еще один обработчик
        c.onErrorEvent += new Proc(secondReaction);

        // Вывод: "-10 is a bad value!" и "Are you stupid?"
        c.setField(-10);
    }
}
```

Выше было указано, что методы добавления и удаления обработчиков события генерируются в классе автоматически. Если программиста по каким-либо причинам не устраивает подобный подход, он может описать собственную

Библиотека БГУИР

реализацию данных методов. Для этого при объявлении события указывается блок, содержащий секции `add` и `remove`:

```
event <имя делегата> <имя события> {
    add { }
    remove { }
};
```

Кроме этого, при наличии собственного кода для добавления/удаления обработчиков, требуется *явно* объявить поле-делегат для хранения списка методов обработки.

Исправим класс `CExampleClass`, использовав для события `onErrorEvent` секции `add` и `remove`:

```
class CExampleClass {
    int field;
    //Данное поле будет содержать список обработчиков
    private Proc handlerList;

    public event Proc onErrorEvent {
        add {
            Console.WriteLine("Handler added");
            // Обработчик поступает как неявный параметр value
            // Обратите внимание на приведение типов!
            handlerList += (Proc) value;
        }
        remove {
            Console.WriteLine("Handler removed");
            handlerList -= (Proc) value;
        }
    }

    public void setField(int i){
        field = i;
        if (i < 0) {
            // Проверяем на null не событие, а скрытое поле
            if (handlerList != null) handlerList(i);
        }
    }
}
```

В заключение отметим, что считается стандартным такой подход, при котором сигнатура делегата, отвечающего за обработку события, содержит параметр `sender` (типа `object`), указывающий на источник события, и объект класса `System.EventArgs` (или класса, производного от `System.EventArgs`). Задача второго параметра – инкапсулировать параметры обработчика события.

19. ИНТЕРФЕЙСЫ

В языке C# запрещено множественное наследование классов. Тем не менее, в C# существует концепция, позволяющая имитировать множественное наследование. Эта концепция *интерфейсов*. Интерфейс представляет собой набор

Библиотека БГУИР

объявлений свойств, индексаторов, методов и событий. Класс или структура могут *реализовывать* определенный интерфейс. В этом случае они берут на себя обязанность предоставить полную реализацию элементов интерфейса (хотя бы пустыми методами). Можно сказать так: интерфейс – это контракт, пункты которого суть свойства, индексаторы, методы и события. Если пользовательский тип реализует интерфейс, он берет на себя обязательство выполнить этот контракт.

Объявление интерфейса схоже с объявлением класса. Для объявления интерфейса используется ключевое слово interface. Интерфейс содержит только заголовки методов, свойств и событий:

```
interface IBird {
    // Метод
    void Fly();

    // Свойство
    double Speed { get; set;}
}
```

Обратите внимание – в определении элементов интерфейса отсутствуют модификаторы уровня доступа. Считается, что все элементы интерфейса имеют public уровень доступа. Более точно, следующие модификаторы не могут использоваться при объявлении членов интерфейса: abstract, public, protected, internal, private, virtual, override, static. Для свойства, объявленного в интерфейсе, указываются только ключевые слова get и (или) set.

Если класс собирается реализовать интерфейс IBird, то он обязуется содержать метод-процедуру без параметров и свойство, доступное и для чтения и для записи, имеющее тип double.

Чтобы показать, что класс реализовывает некий интерфейс, используется синтаксис <имя класса> : <имя реализовываемого интерфейса> при записи заголовка класса. Если класс является производным от некоторого базового класса, то имя базового класса указывается перед именем реализовываемого интерфейса: <имя класса> : <имя базового класса>, <имя интерфейса>. В простейшем случае, чтобы указать, что некий член класса соответствует элементу интерфейса, у них должны совпадать имена:

```
class CFalcon : IBird {
    private double FS;
    public void DoSomeThing() {
        Console.WriteLine("Falcon Flys");
    }
    public void Fly() {
        Console.WriteLine("Falcon Flys");
    }
    public double Speed {
        get { return FS; }
        set { FS = value; }
    }
```

Библиотека БГУИР

```
}
```

При реализации в классе некоторого интерфейса запрещается использование модификатора `static` для соответствующих элементов класса, так как элементы интерфейса должны принадлежать конкретному объекту, а не классу в целом. Для элементов класса, реализующих интерфейс, обязательным является использование модификатора доступа `public`.

В программе допускается использование переменной интерфейсного типа. Такой переменной можно присвоить значение объекта любого класса, реализующего интерфейс. Однако через такую переменную можно вызывать только члены соответствующего интерфейса:

```
// Объявим переменную интерфейсного типа
IBird Bird;

// Инициализация объектом подходящего класса
Bird = new CFalcon();
Bird.Fly();                // Фактически вызывается CFalcon.Fly()

// Строка вызовет ошибку компиляции! В IBird нет такого метода
Bird.DoSomeThing();
```

Если необходимо проверить, поддерживает ли объект `Obj` некоего класса интерфейс `Inter`, то можно воспользоваться операцией `is`:

```
//Результа равен true, если Obj реализует Inter
if (Obj is Inter) . . .
```

Один класс может реализовывать несколько интерфейсов, при этом имена интерфейсов перечисляются после имени класса через запятую:

```
interface ISwimable {
    void Swim();
}

class CDuck : IBird, ISwimable {
    public void Fly() {
        Console.WriteLine("Duck Flies");
    }
    public void Swim() {
        Console.WriteLine("Duck Swims");
    }
    public double Speed {
        get { return 0.0;}
        set {  }
    }
}
```

Если класс реализует несколько интерфейсов, которые имеют элементы с совпадающими именами, или имя одного из членов класса совпадает с именем элемента интерфейса, то при записи члена класса требуется указать имя в виде

Библиотека БГУИР

<имя интерфейса>.<имя члена>. Указание модификаторов доступа при этом запрещается.

Подобно классам, интерфейсы могут наследоваться от других интерфейсов. При этом, в отличие от классов, наследование интерфейсов может быть множественным.

20. СТРУКТУРЫ И ПЕРЕЧИСЛЕНИЯ

Структура – это пользовательский тип, поддерживающий всю функциональность класса, кроме наследования. Тип структуры, определенный в языке C#, в простейшем случае позволяет инкапсулировать несколько полей различных типов. Но элементами структуры в C# могут быть не только поля, а и методы, свойства, события, константы. Структуры, как и классы, могут реализовывать интерфейсы. Синтаксис определения структуры следующий:

```
struct <имя структуры> {
    <элементы структуры>
}
```

При описании полей структуры следует учитывать, что они не могут инициализироваться при объявлении. Как и класс, структура может содержать конструкторы. Однако в структуре можно объявить только пользовательский конструктор с параметрами.

Рассмотрим пример структуры для представления комплексных чисел:

```
struct Complex {
    public double Re, Im;
    public Complex(double X, double Y) {
        Re = X;
        Im = Y;
    }
    public Complex Add(Complex Z) {
        return new Complex(this.Re + Z.Re, this.Im + Z.Im);
    }
}
```

Переменные структур определяются как обычные переменные примитивных типов. Однако следует иметь в виду, что поля таких переменных будут не инициализированны. При определении переменной можно вызвать конструктор без параметров – тогда поля получат значения по умолчанию своих типов. Также допустим вызов пользовательского конструктора:

```
// Поля Z1 не инициализированы, их надо установить
Complex Z1;

// Поля Z2 инициализированы значениями 0.0
Complex Z2 = new Complex();

// Поля Z3 инициализированы значениями 2.0, 3.0
Complex Z3 = new Complex(2.0, 3.0);
```

Библиотека БГУИР

Доступ к элементам структуры осуществляется так же, как к элементам объекта класса:

```
Z1.Re = 10.0;
Z1.Im = 5.0;
Z2 = Z3.Add(Z1);
```

Напомним, что переменные структур размещаются в стеке приложения. Структурные переменные можно присваивать друг другу, при этом выполняется копирование данных структуры на уровне полей.

Перечисление – это тип, содержащий в качестве элементов именованные целочисленные константы. Рассмотрим синтаксис определения перечисления:

```
enum <имя перечисления> [: <тип перечисления>] {
    <элемент перечисления 1> [= <значение элемента>],
    . . .
    <элемент перечисления N> [= <значение элемента>]
}
```

Перечисление может предваряться модификатором доступа. Если задан тип перечисления, то он определяет тип каждого элемента перечисления. Типами перечислений могут быть только `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` или `ulong`. По умолчанию принимается тип `int`. Для элементов перечисления область видимости указать нельзя. Значением элемента перечисления должна быть целочисленная константа. Если для какого-либо элемента перечисления значение опущено, то в случае, если это первый элемент, он принимает значение 0, иначе элемент принимает значение на единицу большее предыдущего элемента. Заданные значения элементов перечисления могут повторяться.

Приведем примеры определения перечислений:

```
enum Seasons {
    Winter,
    Spring,
    Summer,
    Autumn
}

public enum ErrorCodes : byte {
    First = 1,
    Second = 2,
    Fourth = 4
}
```

После описания перечисления можно объявить переменную соответствующего типа:

```
Seasons S;
ErrorCodes EC;
```

Переменной типа перечисления можно присваивать значения, как и обычной переменной:

```
S = Seasons.Spring;
```


Библиотека БГУИР

```
Console.WriteLine(S);   // Выводит на печать Spring
```

Перечисления фактически являются наследниками типа System.Enum. При компиляции проводится простая подстановка соответствующих значений для элементов перечислений.

21. ПРОСТРАНСТВА ИМЕН

Пространства имен служат для логической группировки пользовательских типов. Применение пространств имен обосновано в крупных программных проектах для снижения риска конфликта имен и улучшения структуры библиотек кода.

Синтаксис описания пространства имен следующий:

```
namespace <имя пространства имен> {
    [<компоненты пространства имен>]
}
```

Компонентами пространства имен могут быть классы, делегаты, перечисления, структуры и другие пространства имен. Само пространство имен может быть вложено только в другое пространство имен.

Если в разных местах программы (возможно, в разных входных файлах) определены несколько пространств имен с одинаковыми именами, компилятор собирает компоненты из этих пространств в общее пространство имен. Для этого только необходимо, чтобы одноименные пространства имен находились на одном уровне вложенности в иерархии пространств имен.

Для доступа к компонентам пространства имен используется синтаксис `<имя пространства имен>.<имя компонента>`. Для компилируемых входных файлов имя пространства имен по умолчанию (если в файле нет обрамляющего пространства имен) можно задать специальной опцией компилятора.

Для использования в программе некоего пространства имен служит команда using. Ее синтаксис следующий:

```
using <имя пространства имен>;
```

или

```
using [<имя псевдонима> =] <имя пространства>[.<имя типа>];
```

Импортирование пространства имен позволяет сократить трудозатраты программиста при наборе текстов программ. *Псевдоним*, используемый при импортировании, это обычно короткий идентификатор для ссылки на пространство имен (или элемент из пространства имен) в тексте программы. Импортировать можно пространства имен из текущего проекта, а также из подключенных к проекту сборок.

22. ГЕНЕРАЦИЯ И ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Опишем возможности по обработке и генерации исключительных ситуаций в языке C#.

Библиотека БГУИР

Рассмотрим синтаксис генерации исключительной ситуации. Для генерации исключительной ситуации используется команда `throw` со следующим синтаксисом:

```
throw <объект класса исключительной ситуации>;
```

Обратите внимание: объект, указанный после `throw`, должен обязательно быть объектом класса исключительной ситуации. Таким классом является класс `System.Exception` и все его наследники.

Рассмотрим пример программы с генерацией исключительной ситуации:

```
using System;
class CExample {
    private int fX;
    public void setFx(int x) {
        if (x > 0)
            fX = x;
        else
            // Объект исключит. ситуации создается "на месте"
            throw new Exception();
    }
}
class MainClass {
    public static void Main() {
        CExample A = new CExample();
        A.setFx(-3); // ИС генерируется, но не обрабатывается!
    }
}
```

Так как в данном примере исключительная ситуация генерируется, но никак не обрабатывается, при работе приложения появится стандартное окно с сообщением об ошибке.

Класс `System.Exception` является базовым классом для представления исключительных ситуаций. Основными членами данного класса является свойство только для чтения `Message`, содержащее строку с описанием ошибки, и перегруженный конструктор с одним параметром-строкой, записываемой в свойство `Message`. Естественно, библиотека классов .NET Framework содержит большое число разнообразных классов, порожденных от `System.Exception` и описывающих конкретные исключительные ситуации.

Пользователь может создать собственный класс для представления информации об исключительной ситуации. Единственным условием является прямое или косвенное наследование этого класса от класса `System.Exception`.

Модифицируем пример с генерацией исключительной ситуации, описав для исключительной ситуации собственный класс:

```
class MyException : Exception {
    public int info;
}
class CExample {
    private int fX;
```

Библиотека БГУИР

```
        public void setFx(int x) {
            if (x > 0)
                fX = x;
            else {
                MyException E = new MyException();
                E.info = x;
                throw E;
            }
        }
    }
```

Опишем возможности по обработке исключительных ситуаций. Для перехвата исключительных ситуаций служит блок `try` – `catch` – `finally`. Синтаксис блока следующий:

```
try {
    [<команды, способные вызвать исключительную ситуацию>]
}
[<один или несколько блоков catch>]
[finally {
<операторы из секции завершения> }]
```

Операторы из части `finally` (если она присутствует) выполняются всегда, вне зависимости от того, произошла исключительная ситуация или нет. Если один из операторов, расположенных в блоке `try`, вызвал исключительную ситуацию, управление немедленно передается на блоки `catch`. Синтаксис отдельного блока `catch` следующий:

```
catch [(<тип ИС> [<идентификатор объекта ИС>])] {
    <команды обработки исключительной ситуации>
}
```

`<идентификатор объекта ИС>` – это некая временная переменная, которая может использоваться для извлечения информации из объекта исключительной ситуации. Отдельно описывать эту переменную нет необходимости.

Модифицируем программу, описанную выше, добавив в нее блок перехвата ошибки:

```
class MainClass
{
  public static void Main()
  {
    CExample A = new CExample();
    try {
      Console.WriteLine("Эта строка печатается");
      A.setFx(-3);
      Console.WriteLine("Строка не печатается, если ошибка ");
    }
    catch (MyException ex) {
      Console.WriteLine("Ошибка при параметре {0}", ex.Info);
    }
    finally {
```

Библиотека БГУИР

```
        Console.WriteLine("Строка печатается - блок finally");
      }
    }
  }
```

Если используется несколько блоков `catch`, то обработка исключительных ситуаций должна вестись по принципу «от частного – к общему», так как после выполнения одного блока `catch` управление передается на часть `finally` (при отсутствии `finally` – на оператор после `try` – `catch`). Компилятор C# не позволяет разместить блоки `catch` так, чтобы предыдущий блок перехватывал исключительные ситуации, предназначенные последующим блокам:

```
try {
. . .
}
//Ошибка компиляции, так как MyException - наследник Exception
catch (Exception ex) {
    Console.WriteLine("Общий перехват");
}
catch (MyException ex) {
    Console.WriteLine("Эта строка не печатается никогда!");
}
```

Запись блока `catch` в форме `catch (Exception) {    }` позволяет перехватывать все исключительные ситуации, генерируемые CLR. Если записать блок `catch` в форме `catch {     }`, то такой блок будет обрабатывать любые исключительные ситуации, в том числе и не связанные с исполняющей средой.

Библиотека БГУИР

ЛИТЕРАТУРА

1. Ватсон К. и др. C# для профессионалов (в двух томах). : Пер. с англ. – М.: Лори, 2005.

2. Ватсон К. C#. Программист – программисту. : Пер. с англ. – М.: Лори, 2005.

3. Дубовцев А. В. Microsoft .NET в подлиннике. – СПб.: БХВ-Петербург, 2004.

4. Либерти Дж. Программирование на C#. : Пер. с англ. – М.: Символ-Плюс, 2003.

5. Рихтер Дж. Программирование на платформе Microsoft .NET Framework. : Пер. с англ. – СПб.: Питер, 2005.

6. Троелсен Э. C# и платформа .NET. Библиотека программиста. : Пер. с англ. – СПб.: Питер, 2004.

7. ШИЛДТ Г. ПОЛНЫЙ СПРАВОЧНИК ПО C#. : ПЕР. С АНГЛ. – М.: ИЗДАТЕЛЬСКИЙ ДОМ «ВИЛЬЯМС», 2004.

8. Hoang Lam, Thuan L. Thai .NET Framework Essentials – O'Reilly, 2003.

9. Jeff Prosise. Programming Microsoft .NET (core reference) – Microsoft Press, 2002.

Библиотека БГУИР

СВ. ПЛАН 2006, ПОЗ. 62

Учебное издание

Волосевич Алексей Александрович

ЯЗЫК C#
И ПЛАТФОРМА .NET

Учебно-методическое пособие
по курсу «Избранные главы информатики»
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Ответственный за выпуск А.А. Волосевич

Подписано в печать 23.06.2006. Формат 60x84 1/16. Бумага офсетная.
Гарнитура «Таймс». Печать ризографическая Усл. печ. л. 3,6.
Уч.-изд. л. 3,3. Тираж 150 экз. Заказ 361.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
ЛИ № 02330/0056964 от 01.04.2004. ЛП № 0233/0131666 от 30.04.2004.
220013, Минск, П.Бровки, 6

Библиотека БГУИР