

Введение в платформу Microsoft .NET

1. Введение в платформу Microsoft .NET.

История и этапы развития технологий программирования.

Технологией программирования называют совокупность методов и средств, используемых в процессе разработки программного обеспечения. Как любая другая технология, технология программирования представляет собой *набор технологических инструкций, включающих:*

- указание последовательности выполнения технологических операций;
- перечисление условий, при которых выполняется та или иная операция;
- описания самих операций, где для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т. п.

Причины возникновения платформы Microsoft .NET.

При разработке платформы .NET Framework учитывались следующие цели.

- Обеспечение согласованной объектно-ориентированной среды программирования для локального сохранения и выполнения объектного кода, для локального выполнения кода, распределенного в Интернете, либо для удаленного выполнения.
- Обеспечение среды выполнения кода, минимизирующей конфликты при развертывании программного обеспечения и управлении версиями.
- Обеспечение среды выполнения кода, гарантирующей безопасное выполнение кода, включая код, созданный неизвестным или не полностью доверенным сторонним изготовителем.
- Обеспечение среды выполнения кода, исключающей проблемы с производительностью сред выполнения сценариев или интерпретируемого кода.
- Обеспечение единых принципов разработки для разных типов приложений, таких как приложения Windows и веб-приложения.
- Взаимодействие на основе промышленных стандартов, которое гарантирует интеграцию кода платформы .NET Framework с любым другим кодом.

Сравнительный анализ преимуществ и недостатков платформы Microsoft .NET

Преимущества

- **Независимость от языка.** Благодаря CLR (Common Language Runtime, общезыковая исполняющая среда) все языки которые поддерживаются .Net компилируются на языке промежуточного уровня. Это позволяет внедрять библиотеки, которые написаны на других языках.
- **Среда разработки.** Visual Studio (VS) одна из лучших IDE (Integrated Development Environment, интегрированная среда разработки) на сегодняшний день. Удобная в использовании, большое количество настроек позволяют подстроить среду под себя.
- **Сила C#.** Объектно-ориентированный язык. Является основным при разработке .Net проектов.
- **Библиотеки.** Интеграция библиотек максимально упрощена благодаря Nuget менеджеру (инструмент для работы с библиотеками). Огромное количество библиотек для различных видов проектов.

Если бы не .NET, пользователям пришлось бы устанавливать среду исполнения для программ на каждом языке. То есть чтобы запустить приложение на Visual Basic, нужно скачать среду выполнения для Visual Basic. Если же программа написана на C#, то придётся скачивать среду и для неё.

Это быстро займет всё место на компьютере немного отличающимися копиями одних и тех же библиотек.

Для программистов это тоже важно, потому что даёт возможность развивать одну среду, которая используется сразу для четырёх языков. Иначе обычным разработчикам приходилось бы ждать, пока выйдет новая версия библиотек для их языка. Менее популярные языки, вроде F#, получали бы обновление намного позже C#.

Кроме основных языков есть также и другие, которые поддерживаются .NET. Среди них *COBOL*, *Fortran*, *Haskell* и даже *Java* .

На этих языках часто написаны старые (*legacy*) проекты, которые сложно перевести на новую технологию. .NET позволяет переписать часть программы на *COBOL* под стандарты .NET, а потом просто писать новые части на более современном языке, вроде *Visual Basic*.

Недостатки

- Недостатком в некоторых случаях может является существенное замедление выполнения программ. Это неудивительно, так как между исходным языком и машинным кодом вводится дополнительный уровень, MSIL. Однако промежуточное представление .NET с самого начала

проектировалось с прицелом на компиляцию времени исполнения (в отличие, например, от Java bytecode, который разрабатывался с прицелом на интерпретацию).

- **Дистрибутивы для работы приложений.** Для работы приложений .Net на Windows должны быть установлены специальные дистрибутивы. Каждую новую версию дистрибутивов .Net нужно устанавливать отдельно. Если зайти в менеджер приложений, можно удивиться сколько их установлено.

2. Базовые понятия платформы Microsoft .NET

Архитектура платформы Microsoft .NET.

Платформа .NET Framework состоит из общезыковой среды выполнения (среды CLR) и библиотеки классов .NET Framework. Основой платформы .NET Framework является среда CLR. Среду выполнения можно считать агентом, который управляет кодом во время выполнения и предоставляет основные службы, такие как управление памятью, управление потоками и удаленное взаимодействие. При этом средой накладываются условия строгой типизации и другие виды проверки точности кода, обеспечивающие безопасность и надежность. Фактически основной задачей среды выполнения является управление кодом. Код, который обращается к среде выполнения, называют управляемым кодом, а код, который не обращается к среде выполнения, называют неуправляемым кодом. Библиотека классов является комплексной объектно-ориентированной коллекцией повторно используемых типов, которые применяются для разработки приложений — начиная с обычных приложений, запускаемых из командной строки, и приложений с графическим интерфейсом (GUI) и заканчивая приложениями, использующими последние технологические возможности ASP.NET, такие как веб-формы и веб-службы XML.

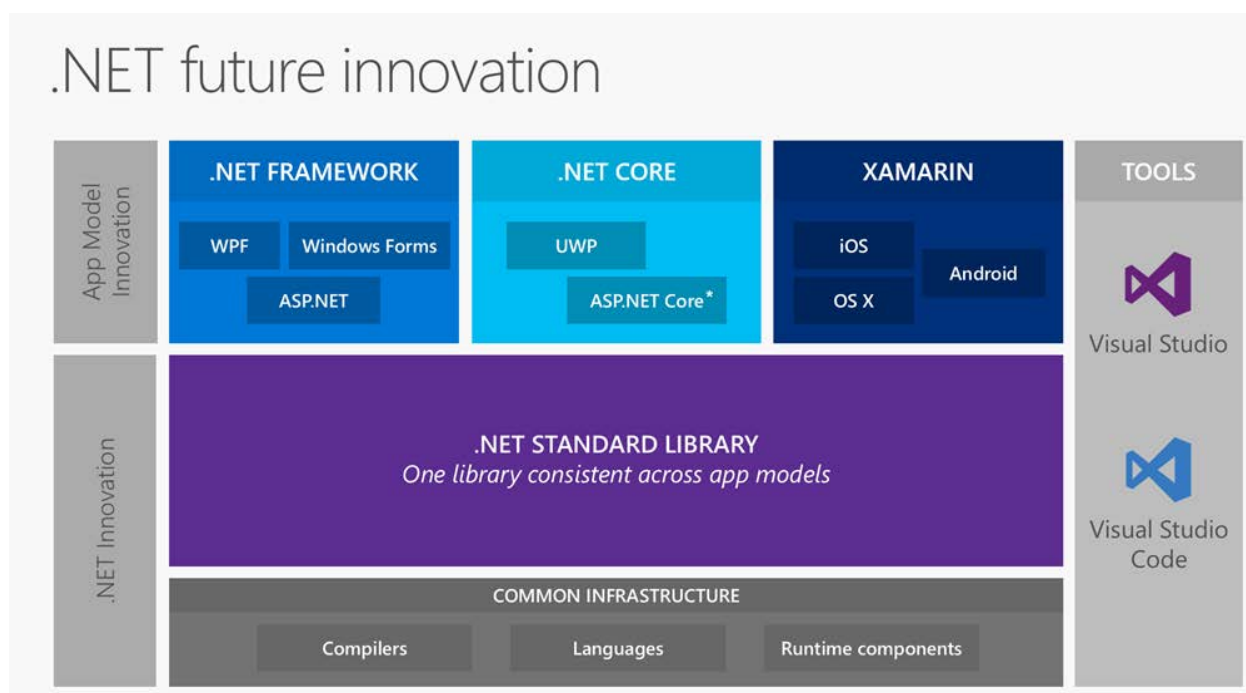
Платформа .NET Framework может размещаться неуправляемыми компонентами, которые загружают среду CLR в собственные процессы и запускают выполнение управляемого кода, создавая таким образом программную среду, позволяющую использовать средства как управляемого, так и неуправляемого выполнения. Платформа .NET Framework не только предоставляет несколько базовых сред выполнения, но также поддерживает разработку базовых сред выполнения независимыми производителями.

Например, ASP.NET размещает среду выполнения и обеспечивает масштабируемую среду для управляемого кода на стороне сервера. ASP.NET работает непосредственно со средой выполнения, чтобы обеспечить выполнение приложений ASP.NET и веб-служб XML.

Обозреватель Internet Explorer может служить примером неуправляемого приложения, размещающего среду выполнения (в виде расширений типов MIME). Размещение среды выполнения в обозревателе Internet Explorer позволяет внедрять управляемые компоненты или элементы управления Windows Forms в HTML-документы. Такое размещение среды позволяет выполнять управляемый мобильный код и пользоваться его существенными преимуществами, в частности выполнением в условиях неполного доверия и изолированным хранением файлов.

.NET Core

.NET Core — это открытая универсальная платформа разработки, которая поддерживается корпорацией Майкрософт и сообществом .NET на сайте [GitHub](#). Она является *кроссплатформенной*, поддерживает **Windows**, **Mac OS** и **Linux** и может использоваться на устройствах, в облаке, во внедренных системах и в сценариях **IoT** (Интернета вещей). В её основе лежат технологии *.NET Framework* и *Silverlight*. Она оптимизирована для мобильных и серверных рабочих нагрузок, поскольку обеспечивает поддержку самодостаточных развёртываний XCOPY.



.NET Core обладает следующими характеристиками:

- **Кроссплатформенность.** Поддержка операционных систем Windows, macOS и Linux.
- **Согласованность между архитектурами.** Одинаковое выполнение кода в различных архитектурах, включая x64, x86 и ARM.
- **Программы командной строки.** Удобные инструменты для локальной разработки и сценариев непрерывной интеграции.
- **Гибкая разработка.** Может включаться в приложение или устанавливаться параллельно (на уровне пользователя или системы). Возможность использования с контейнерами Docker.
- **Совместимость.** Платформа .NET Core совместима с .NET Framework, Xamarin и Mono благодаря .NET Standard.
- **Открытый код.** Платформа .NET Core имеет открытый код и распространяется по лицензиям MIT и Apache 2. .NET Core является проектом .NET Foundation.

- **Поддержка от Майкрософт.** Корпорация Майкрософт предоставляет поддержку .NET Core.

Языки

.NET Core позволяет создавать приложения и библиотеки на языках C#, Visual Basic и F#. Эти языки можно использовать в вашем любимом текстовом редакторе либо интегрированной среде разработки (IDE), включая следующие.

- Visual Studio
- Visual Studio Code
- Sublime Text
- Vim

API - интерфейсы

.NET Core предоставляет API-интерфейсы для множества сценариев, в некоторых из которых используются:

- Примитивные типы, например System.Boolean и System.Int32.
- Коллекции, такие как System.Collections.Generic.List<T> и System.Collections.Generic.Dictionary<TKey,TValue>.
- Служебные типы, такие как System.Net.Http.HttpClient и System.IO.FileStream.
- Типы данных, такие как System.Data.DataSet и DbSet.
- Высокопроизводительные типы, такие как System.Numerics.Vector и Pipelines.

Благодаря поддержке спецификации .NET Standard платформа .NET Core совместима с .NET Framework и API-интерфейсами Mono.

.NET Core состоит из перечисленных ниже компонентов.

- Среда выполнения .NET Core предоставляет систему типов, функции загрузки сборок, сборщик мусора, собственные функции взаимодействия и другие базовые службы. Библиотеки платформы .NET Core предоставляют примитивные типы данных, типы компоновки приложений и базовые служебные программы.
- Среда выполнения ASP.NET Core — это платформа для создания современных облачных приложений, подключенных к Интернету: веб-приложений, приложений Интернета вещей, серверной части мобильных решений и многого другого.
- .NET Core CLI и компиляторы языков (Roslyn и F#) реализуют возможности разработки .NET Core.
- Команда dotnet используется для запуска приложений .NET Core и CLI. Оно выбирает среду выполнения, размещает ее, предоставляет политику загрузки сборок и запускает приложения и инструменты.

Открытый код

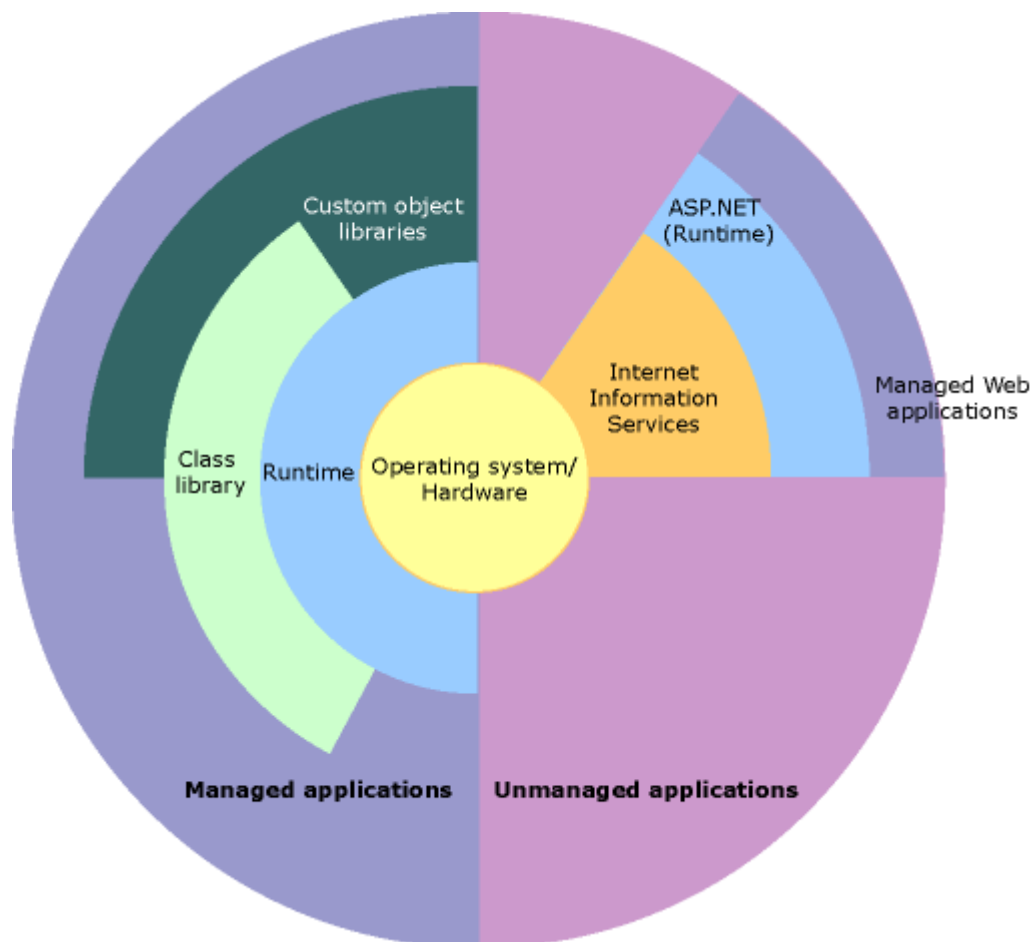
Платформа .NET Core имеет открытый исходный код (по лицензии MIT) и была передана корпорацией Майкрософт в фонд .NET Foundation в 2014 году. Теперь это один из самых активно развивающихся проектов .NET Foundation. Частные лица и организации могут использовать платформу в

личных, образовательных или коммерческих целях. Многие компании используют .NET Core в составе своих приложений, средств, новых платформ и услуг размещения. Некоторые из них вносят существенный вклад в проект .NET Core на портале GitHub и участвуют в управлении его развитием в рамках Координационного совета .NET Foundation.

Платформа .NET Core состоит из сочетания зависимых и независимых от платформы библиотек. Это можно проследить на ряде примеров.

- Библиотека CoreCLR зависит от платформы. Она построена на основе подсистем ОС, таких как диспетчер памяти и планировщик потоков.
- Библиотеки System.IO и System.Security.Cryptography.Algorithms зависят от платформ, так как API-интерфейсы хранения и шифрования данных различаются в каждой ОС.
- Библиотеки System.Collections и System.Linq не зависят от платформы, так как они создают структуры данных и работают с ними.

На следующем рисунке демонстрируется взаимосвязь среды CLR и библиотеки классов с пользовательскими приложениями и всей системой. На рисунке также показано, как управляемый код работает в пределах более широкой архитектуры.



Общезыковая среда исполнения CLR (common language runtime).

Среда CLR управляет памятью, выполнением потоков, выполнением кода, проверкой безопасности кода, компиляцией и другими системными службами. Эти средства являются внутренними для управляемого кода, который выполняется в среде CLR.

Среда выполнения также обеспечивает надежность кода, реализуя инфраструктуру строгой типизации и проверки кода, которую называют системой общих типов (CTS). Система общих типов обеспечивает самоописание всего управляемого кода. Различные языковые компиляторы корпорации Microsoft и независимых изготовителей создают управляемый код, удовлетворяющий системе общих типов. Это означает, что управляемый код может принимать другие управляемые типы и экземпляры, при этом обеспечивая правильность типов и строгую типизацию.

Кроме того, управляемая среда выполнения исключает многие часто возникающие проблемы с программным обеспечением. Например, среда выполнения автоматически управляет размещением объектов и ссылками на объекты, освобождая их, когда они больше не используются. Автоматическое управление памятью исключает две наиболее часто возникающие ошибки приложений: утечки памяти и недействительные ссылки на память.

Среда выполнения также повышает продуктивность разработчиков. Например, программисты могут писать приложения на привычном языке разработки, при этом используя все преимущества среды выполнения, библиотеки классов и компонентов, написанных другими разработчиками на других языках. Это доступно любому производителю компиляторов, обращающихся к среде выполнения. Языковые компиляторы, предназначенные для платформы .NET Framework, делают средства .NET Framework доступными для существующего кода, написанного на соответствующих языках, существенно облегчая процесс переноса существующих приложений.

Хотя среда выполнения разрабатывалась для будущего программного обеспечения, она также поддерживает сегодняшнее и вчерашнее программное обеспечение. Взаимодействие управляемого и неуправляемого кодов позволяет разработчикам использовать необходимые компоненты COM и библиотеки DLL.

Среда выполнения разработана для повышения производительности. Хотя общезыковая среда выполнения предоставляет многие стандартные службы времени выполнения, управляемый код никогда не интерпретируется. Средство компиляции по требованию (JIT) позволяет выполнять весь управляемый код на машинном языке компьютера, где он

запускается. Между тем диспетчер памяти устраняет возможность фрагментации памяти и увеличивает объем адресуемой памяти для дополнительного повышения производительности.

Наконец, среда выполнения может размещаться в высокопроизводительных серверных приложениях, таких как Microsoft SQL Server и службы IIS (Internet Information Services). Такая инфраструктура позволяет использовать управляемый код для написания собственной логики программ, пользуясь при этом высочайшей производительностью лучших производственных серверов, которые поддерживают размещение среды выполнения.

Существует ряд средств, которые поддерживаются .NET, но не поддерживаются C#, и, возможно, вас удивит, что есть также средства, поддерживаемые C# и не поддерживаемые .NET (например, некоторые случаи перегрузки операций). Однако поскольку язык C# предназначен для применения на платформе .NET, вам, как разработчику, важно иметь представление о .NET Framework, если вы хотите эффективно разрабатывать приложения на C#. Поэтому давайте заглянем "за кулисы" .NET.

Центральной частью каркаса .NET является его общезыковая исполняющая среда, известная как *Common Language Runtime (CLR)* или *.NET runtime*. Код, выполняемый под управлением CLR, часто называют *управляемым кодом*. С точки зрения программирования под термином **исполняющая среда** может пониматься коллекция внешних служб, которые требуются для выполнения скомпилированной единицы программного кода.

В составе .NET предлагается еще одна исполняющая среда. Главное отличие между исполняющей средой .NET и упомянутыми выше средами, состоит в том, что исполняющая среда .NET обеспечивает единый четко определенный уровень выполнения, который способны использовать все совместимые с .NET языки и платформы.

Однако перед тем как код сможет выполняться CLR, любой исходный текст (на C# или другом языке) должен быть скомпилирован. Компиляция в .NET состоит из двух шагов:

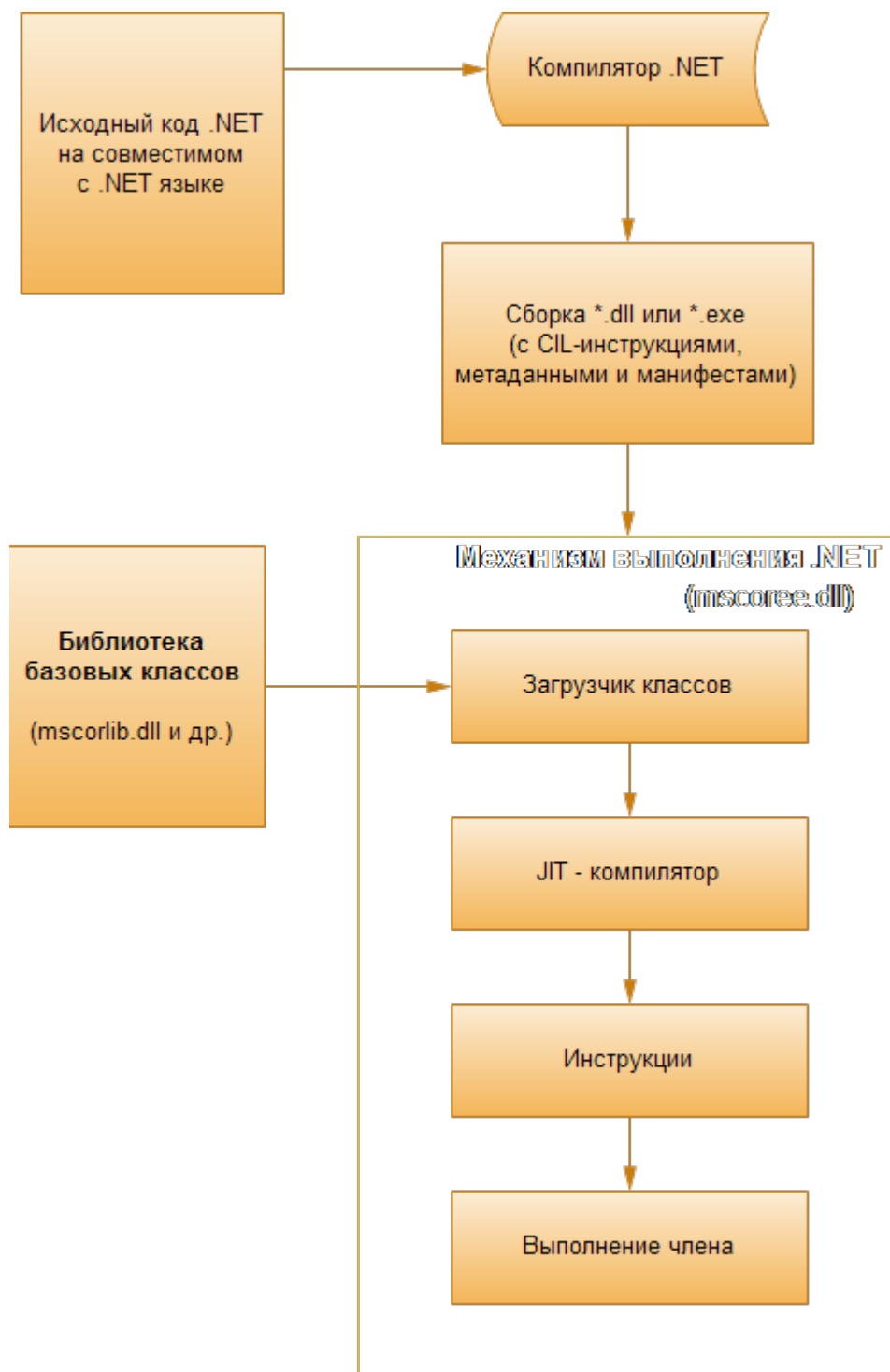
1. Компиляция исходного кода в Microsoft Intermediate Language (IL)
2. Компиляция IL в специфичный для платформы код с помощью CLR

Этот двухшаговый процесс компиляции очень важен, потому что наличие Microsoft Intermediate Language (IL) является ключом ко многим преимуществам .NET. Microsoft Intermediate Language (промежуточный язык Microsoft) разделяет с байт-кодом Java идею низкоуровневого языка с

простым синтаксисом (основанным на числовых, а не текстовых кодах), который может быть очень быстро транслирован в родной машинный код.

Основной механизм CLR физически имеет вид библиотеки под названием *mscorlib.dll* (и также называется общим механизмом выполнения исполняемого кода объектов — Common Object Runtime Execution Engine). При добавлении ссылки на сборку для ее использования загрузка библиотеки *mscorlib.dll* осуществляется автоматически и затем, в свою очередь, приводит к загрузке требуемой сборки в память. Механизм исполняющей среды отвечает за выполнение целого ряда задач. Сначала, что наиболее важно, он отвечает за определение места расположения сборки и обнаружение запрашиваемого типа в двоичном файле за счет считывания содержащихся там метаданных. Затем он размещает тип в памяти, преобразует CIL-код в соответствующие платформе инструкции, производит любые необходимые проверки на предмет безопасности и после этого, наконец, непосредственно выполняет сам запрашиваемый программный код.

Помимо загрузки пользовательских сборок и создания пользовательских типов, механизм CLR при необходимости будет взаимодействовать и с типами, содержащимися в библиотеках базовых классов .NET. Хотя вся библиотека базовых классов поделена на ряд отдельных сборок, главной среди них является сборка *mscorlib.dll*. В этой сборке содержится большое количество базовых типов, охватывающих широкий спектр типичных задач программирования, а также базовых типов данных, применяемых во всех языках .NET. При построении .NET-решений доступ к этой конкретной сборке будет предоставляться автоматически.



На данной схеме наглядно видно, как выглядят взаимоотношения между исходным кодом (предусматривающим использование типов из библиотеки базовых классов), компилятором .NET и механизмом выполнения .NET.

Использование байт-кода с четко определенным универсальным синтаксисом дает ряд существенных *преимуществ*:

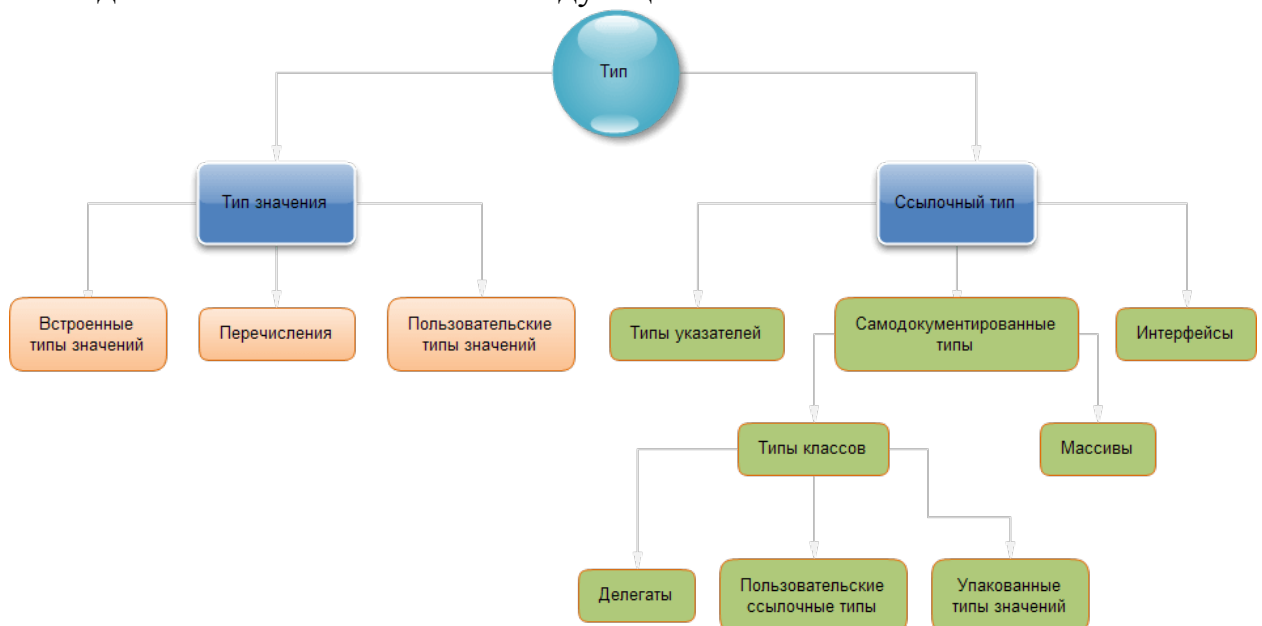
Независимость от платформы

Первым делом, это значит, что файл, содержащий инструкции байт-кода, может быть размещен на любой платформе; во время выполнения может быть легко проведена финальная стадия компиляции, что позволит выполнить код на конкретной платформе. Другими словами, компилируя в IL, вы получаете платформенную независимость .NET — во многом так же, как компиляция в байт-код Java обеспечивает независимость от платформы программам на Java.

Стандартная система типов CTS (common type system)

В каждой конкретной сборке может содержаться любое количество различающихся типов. В мире .NET "тип" представляет собой просто общий термин, который применяется для обозначения любого элемента из множества (класс, интерфейс, структура, перечисление, делегат). При построении решений с помощью любого языка .NET, скорее всего, придется взаимодействовать со многими из этих типов. Например, в сборке может содержаться один класс, реализующий определенное количество интерфейсов, метод одного из которых может принимать в качестве входного параметра перечисление, а возвращать структуру.

CTS (общая система типов) представляет собой формальную спецификацию, в которой описано то, как должны быть определены типы для того, чтобы они могли обслуживаться в CLR-среде. Внутренние детали CTS обычно интересуют только тех, кто занимается разработкой инструментов и/или компиляторов для платформы .NET. Т.е. CTS описывает не просто примитивные типы данных, а целую развитую иерархию типов, включающую хорошо определенные точки, в которых код может определять свои собственные типы. Иерархическая структура общей системы типов (CTS) отражает объектно-ориентированную методологию одиночного наследования IL и показана на следующей схеме:



Важно уметь работать на предпочитаемом ими языке с пятью типами из CTS. Краткий обзор этих типов приведен ниже:

Типы классов

В каждом совместимом с .NET языке поддерживается, как минимум, понятие *типа класса (class type)*, которое играет центральную роль в объектно-ориентированном программировании (ООП). Каждый класс может включать в себя любое количество членов (таких как конструкторы, свойства, методы и события) и точек данных (полей). В C# классы объявляются с помощью ключевого слова **class**.

```
class mySum
{
    public int Sum (int x, int y)
    { return x+y; }
}
```

Рассмотрим основные характеристики классов CTS:

Типы классов CTS

Характеристика классов	Описание
Запечатанные	<i>Запечатанные (sealed)</i> , или герметизированные, классы не могут выступать в роли базовых для других классов, т.е. не допускают наследования
Реализующие интерфейсы	<i>Интерфейсом (interface)</i> называется коллекция абстрактных членов, которые обеспечивают возможность взаимодействия между объектом и пользователем этого объекта. CTS позволяет реализовать в классе любое количество интерфейсов
Абстрактные или конкретные	Экземпляры <i>абстрактных (abstract)</i> классов не могут создаваться напрямую, и предназначены для определения общих аспектов поведения для производных типов. Экземпляры же <i>конкретных (concrete)</i> классов могут создаваться напрямую
Степень	Каждый класс должен конфигурироваться с <i>атрибутом</i>

видимости

видимости (visibility). По сути, этот атрибут указывает, должен ли класс быть доступным для использования внешним сборкам или только изнутри определяющей сборки

Типы интерфейсов

Интерфейсы представляют собой не более чем просто именованную коллекцию определений абстрактных членов, которые могут поддерживаться (т.е. реализоваться) в данном классе или структуре. В С# типы интерфейсов определяются с помощью ключевого слова **interface**, как показано ниже:

```
// Объявление интерфейса
public interface ICommandSource
{
    void CommandParameter();
}
```

Сами по себе интерфейсы мало чем полезны. Однако когда они реализуются в классах или структурах уникальным образом, они позволяют получать доступ к дополнительным функциональным возможностям за счет добавления просто ссылки на них в полиморфной форме.

Типы структур

Понятие структуры тоже сформулировано в CTS. Тем, кому приходилось работать с языком С, будет приятно узнать, что таким пользовательским типам удалось "выжить" в мире .NET (хотя на внутреннем уровне они и ведут себя несколько иначе). Попросту говоря, структура может считаться "облегченным" типом класса с основанной на использовании значений семантикой. В С# они создаются с помощью ключевого слова **struct**.

```
// Тип структуры в С#
struct Rectangle
{
    // В структурах могут содержаться поля, конструкторы и определяться
    методы

    public string recFill;

    public void MyBackground()
```

```
{
    Console.WriteLine("Фон элемента: "+recFill);
}
```

Типы перечислений

Перечисления (enumeration) представляют собой удобную программную конструкцию, которая позволяет группировать данные в пары "имя-значение". Вместо того чтобы использовать и отслеживать числовые значения для каждого варианта, в этом случае гораздо удобнее создать соответствующее перечисление с помощью ключевого слова **enum**:

Объявление перечисления, в котором день недели имеет свой номер, начиная с понедельника. Демонстрируется правило: каждой следующей константе присваивается значение, которое на 1 больше значения предшествующей инициализированной константы.

Объявление перечисления типа DayWeek:

```
enum DayWeek
{
    Sunday = 7, Monday = 1, Tuesday, Wednesday,
    Thursday, Friday, Saturday
}
```

В CTS необходимо, чтобы перечисляемые типы наследовались от общего базового класса **System.Enum**. В этом базовом классе присутствует ряд весьма интересных членов, которые позволяют извлекать, манипулировать и преобразовывать базовые пары "имя-значение" программным образом.

Члены типов

Теперь, когда было приведено краткое описание каждого из сформулированных в CTS типов, пришла пора рассказать о том, что большинство из этих типов способно принимать любое количество *членов (member)*. Формально в роли члена типа может выступать любой элемент из множества {конструктор, финализатор, статический конструктор, вложенный тип, операция, метод, свойство, индексатор, поле, поле только для чтения, константа, событие}.

В спецификации CTS описываются различные "характеристики", которые могут быть ассоциированы с любым членом. Например, каждый член может обладать характеристикой, отражающей его доступность (т.е. общедоступный, приватный или защищенный). Некоторые члены могут объявляться как абстрактные (для навязывания полиморфного поведения производным типам) или как виртуальные (для определения фиксированной, но допускающей переопределение реализации). Кроме того, почти все члены также могут делаться статическими членами (привязываться на уровне класса) или членами экземпляра (привязываться на уровне объекта).

Встроенные типы данных

И, наконец, последним, что следует знать о спецификации CTS, является то, что в ней содержится четко определенный набор фундаментальных типов данных. Хотя в каждом отдельно взятом языке для объявления того или иного встроенного типа данных из CTS обычно предусмотрено свое уникальное ключевое слово, все эти ключевые слова в конечном итоге соответствуют одному и тому же типу в сборке mscorlib.dll.

В следующей таблице показано, как ключевые типы данных из CTS представляются в C#:

Классы типов данных C#

Типы данных в CTS	Ключевое слово в C#
--------------------------	----------------------------

System.Byte	byte
System.SByte	sbyte
System.Int16	short
System.Int32	int
System.Int64	long
System.UInt16	ushort
System.UInt32	uint
System.UInt64	ulong
System.Single	float
System.Double	double
System.Object	object
System.Char	char
System.String	String

Стандартная языковая спецификация CLS (common language specification).

CLS (Common Language Specification — общая спецификация для языков программирования) как раз и представляет собой набор правил, которые во всех подробностях описывают минимальный и полный комплект функциональных возможностей, которые должен обязательно поддерживать каждый отдельно взятый .NET-компилятор для того, чтобы генерировать такой программный код, который мог бы обслуживаться CLR и к которому в то же время могли бы единообразным образом получать доступ все языки, ориентированные на платформу .NET. Во многих отношениях CLS может считаться просто подмножеством всех функциональных возможностей, определенных в CTS.

Самым главным в CLS является *правило 1*, гласящее, что *правила CLS касаются только тех частей типа, которые делаются доступными за пределами сборки, в которой они определены.*

Из этого правила можно (и нужно) сделать вывод о том, что все остальные правила в CLS не распространяются на логику, применяемую для построения внутренних рабочих деталей типа .NET. Единственными аспектами типа, которые должны соответствовать CLS, являются сами **определения членов** (т.е. соглашения об именовании, параметры и возвращаемые типы). В рамках логики реализации члена может применяться любое количество и не согласованных с CLS приемов, поскольку для внешнего мира это не будет играть никакой роли.

Библиотека классов FCL (BCL)

BCL обозначает библиотеку базовых классов, также известную как библиотека классов (CL). BCL является подмножеством библиотеки классов Framework (FCL). Библиотека классов - это коллекция многократно используемых типов, которые тесно интегрированы с CLR. Библиотека базовых классов предоставляет классы и типы, которые полезны при выполнении повседневных операций, например работа со строковыми и примитивными типами, подключением к базе данных, операциями IO.

Стоит отметить, что не все языки, поддерживающие платформу .NET, предоставляют или обязаны предоставлять одинаково полный доступ ко всем классам и всем возможностям BCL — это зависит от особенностей реализации конкретного компилятора и языка.

В отличие от многих других библиотек классов, например, MFC, ATL/WTL или SmartWin, библиотека BCL не является некоей «надстройкой» над функциями операционной системы или над каким-либо API. Библиотека BCL является органической частью самой платформы

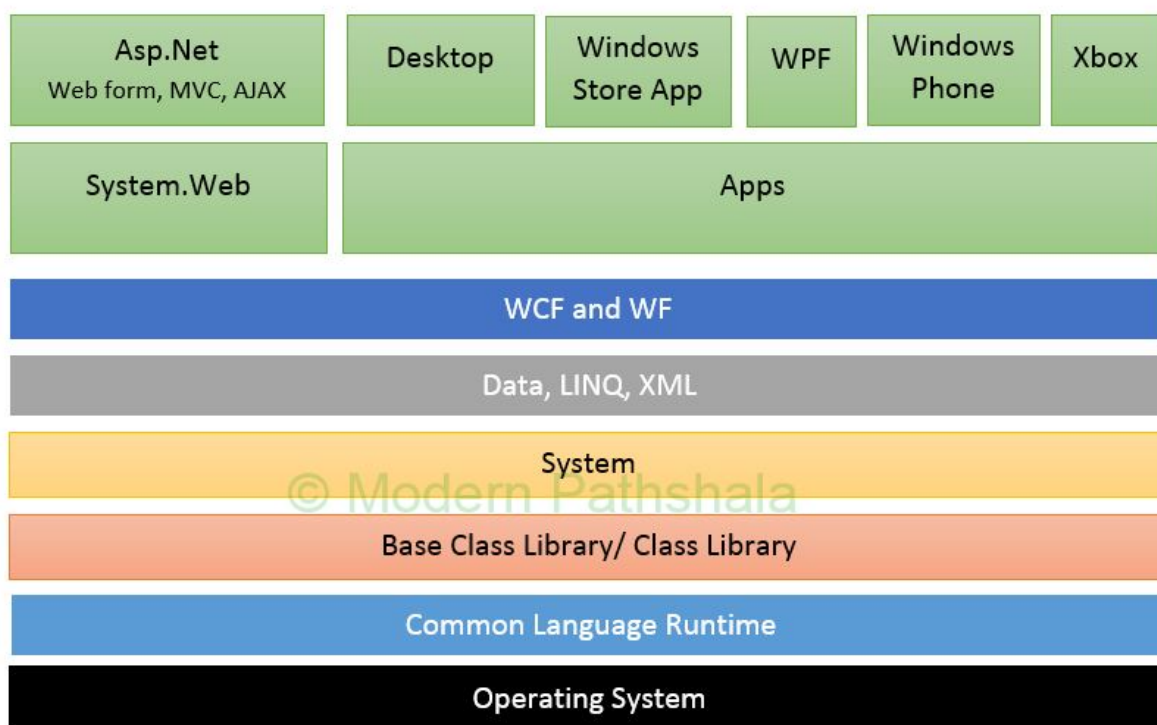
.NET Framework, её «родным» API. Её можно рассматривать как API виртуальной машины .NET.

BCL обновляется с каждой версией .NET Framework.

В то время как библиотека классов Framework содержит тысячи классов, используемых для создания различных типов приложений, и предоставляет все основные функциональные возможности и службы, которые необходимы приложению. FCL включает в себя классы и сервисы для поддержки различных приложений, например.

- Настольное приложение,
- Веб-приложение (ASP.Net, MVC, WCF),
- Мобильное приложение,
- Приложение Xbox,
- услуги Windows и т. д.

Framework Class Library



.NET Standard

.NET Standard — это официальная спецификация API .NET, которые должны быть доступны во всех реализациях .NET. .NET Standard создана для того, чтобы повысить согласованность в экосистеме .NET. ECMA 335 (спецификация общезыковой инфраструктуры (CLI)) продолжает обеспечивать единообразие для реализации .NET. Но так как в ECMA 335 определен лишь небольшой набор стандартных библиотек, спецификация .NET Standard охватывает более широкий спектр API-интерфейсов .NET. .NET Standard предоставляет следующие важные возможности:

- Определяет унифицированный набор API-интерфейсов BCL для реализации всеми реализациями .NET независимо от рабочей нагрузки.
- Позволяет разработчикам создавать переносимые библиотеки, которые могут использоваться в разных реализациях .NET, с помощью одного набора API-интерфейсов.
- Позволяет сократить или даже устранить условную компиляцию общего источника из-за API-интерфейсов .NET (только для API операционной системы).

Различные реализации .NET реализуют конкретные версии .NET Standard. Каждая версия реализации .NET ориентирована на использование максимальной поддерживаемой ею версии .NET Standard. Это также означает, что она поддерживает и предыдущие версии. Например, .NET Framework 4.6 реализует .NET Standard 1.3, то есть предоставляет все API, определенные в стандартах .NET Standard версий 1.0–1.3. Аналогичным образом .NET Framework 4.6.1 реализует .NET Standard 1.4, а .NET Core 1.0 — .NET Standard 1.6.

Поддержка реализации .NET

В следующей таблице перечислены **минимальные** версии платформы, которые поддерживают каждую версию .NET Standard. Это означает, что более поздние версии перечисленных платформ также поддерживают соответствующую версию .NET Standard. К примеру, .NET Core 2.2 поддерживает .NET Standard 2.0 и более ранних версий.

TABLE 1

.NET Standard	<u>1.0</u>	<u>1.1</u>	<u>1.2</u>	<u>1.3</u>	<u>1.4</u>	<u>1.5</u>	<u>1.6</u>	<u>2.0</u>	<u>2.1</u>
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework ¹	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1 ²	4.6.1 ²	4.6.1 ²	N/A ³
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5,4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
Универсальная платформа Windows	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	TBD
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	TBD

1 Перечисленные версии для .NET Framework применимы к пакету SDK для .NET Core 2.0 и инструментарию более поздних версий. В предыдущих версиях использовалось другое сопоставление для .NET Standard 1.5 и более поздних версий.

2. Перечисленные здесь версии соответствуют правилам, которые NuGet (система управления пакетами для платформ разработки Microsoft, в первую очередь библиотек .NET Framework. Управляется .NET Foundation.) использует для определения применимости указанной библиотеки .NET Standard. Хотя NuGet считает, что .NET Framework 4.6.1 поддерживает .NET Standard с версии 1.5 до 2.0, существует ряд проблем с использованием библиотек .NET Standard, созданных для этих версий из проектов .NET Framework 4.6.1. Для проектов .NET Framework, которым нужно использовать такие библиотеки, рекомендуем обновить проект и использовать в качестве целевой версию .NET Framework 4.7.2 или выше.

3 .NET Framework не поддерживает .NET Standard 2.1 или более поздних версий.

- Столбцы представляют версии .NET Standard..
- Строки представляют различные реализации .NET.
- Номер версии в каждой ячейке обозначает *минимальную* версию реализации, которая потребуется для работы с соответствующей версией .NET Standard.

Чтобы найти самую позднюю версию .NET Standard, на которую можно ориентироваться, выполните следующее:

1. Найдите строку, которая соответствует вашей реализации .NET.
2. Двигаясь по этой строке справа налево, найдите столбец, который содержит вашу версию платформы.
3. Заголовок столбца указывает версию .NET Standard, которую поддерживает ваша целевая платформа. Соответственно, поддерживаются и все более ранние версии .NET Standard. Более поздние версии .NET Standard также будет поддерживать текущую реализацию.
4. Повторите эту процедуру для всех платформ, с которыми вы будете работать. Если целевых платформ несколько, выберите из них самую младшую версию. Например, если вы хотите работать с .NET Framework 4.5 и .NET Core 1.0, наивысшей доступной версией .NET Standard будет .NET Standard 1.1.

Какую версию .NET Standard выбрать в качестве целевой

При выборе версии .NET Standard перед вами стоит такая дилемма:

- Чем выше версия, тем больше вам доступно интерфейсов API.
- Чем ниже версия, тем больше платформ ее поддерживают.

В общем случае рекомендуется выбирать *наименьшую* из возможных версий .NET Standard. После того, как вы определите номер наибольшей возможной версии .NET Standard, выполните следующие действия.

1. Выберите предыдущую версию .NET Standard, создайте и соберите проект для нее.

2. Если сборка проекта пройдет успешно, повторите шаг 1. В противном случае вернитесь к предыдущей, более высокой, версии, и используйте именно ее.

Но при выборе более ранних версий .NET Standard предоставляются различные возможности для поддержки зависимостей. Если проект предназначен для .NET Standard 1.x, мы рекомендуем вам *также* выбрать .NET Standard 2.0. Это упростит схему зависимостей для пользователей библиотеки, выполняющейся на совместимых платформах с .NET Standard 2.0, и сократит количество пакетов, которые необходимо скачать.

Правила управления версиями .NET Standard

Существует два основных правила управления версиями.

- Аддитивность. Все версии .NET Standard логически расширяются, то есть более поздние версии содержат все интерфейсы API предыдущих версий. Отсутствуют критические изменения между версиями.

- Неизменяемость. Версии .NET после выпуска закрепляются в определенном состоянии. Новые интерфейсы API сначала становятся доступными для конкретных реализаций .NET, например .NET Core. Если совет по утверждению .NET Standard решает, что новые интерфейсы API нужно сделать доступными для всех реализаций .NET, их добавляют в новую версию .NET Standard.

Пространства имён (Namespaces)

Пространство имен (namespace) в C# представляет собой некий контейнер для логического объединения именованных сущностей, таких как классы, интерфейсы, перечисления и тому подобное. Для чего нужны такие контейнеры? Ну во-первых, пространства имен позволяют логически группировать классы и другие сущности, а во-вторых, позволяют использовать одни и те же имена для сущностей в разных пространствах имен. Причем, вторая функция даже больше востребована, так как логическую группировку тех же классов, программист может игнорировать, особенно, если классов не так много, а вот создать два класса с одним именем уже нельзя, не даст компилятор!

При создании нового проекта в Visual Studio автоматически создается пространство имен. для этого используется ключевое слово **namespace**, после которого, через пробел, указывается название пространства имен. И всё, что объявляется внутри фигурных скобок, следующих на за названием, относится к пространству имен.

Как же пространства имен, позволяют использовать одни и те же имена для сущностей. Дело в том, что пространство имен, как бы задает префикс к имени любой сущности, в нем объявленной. Но мы этого не замечаем, если

работаем с этой сущностью в том пространстве имен, в котором она объявлена. Таким образом имя сущности, например класса, на самом деле становится как бы составным. Представим, что у нас есть класс «*SomeClass*», который объявлен в пространстве имен «*SomeNamespace*», тогда доступ к этому классу из метода «*Main*» был бы примерно таким:

```
//Главный метод программы (точка входа)
static void Main(string[] args)
{
    //Создание объекта класса SomeClass из пространства
имен SomeNamespace
    SomeNamespace.SomeClass tmpObj = new
SomeNamespace.SomeClass();
}
```

Пространство имен «*SomeNamespace*», как бы ограничивает область видимости для класса «*SomeClass*», и получить доступ, к этому классу мы можем только через пространство имен, его содержащее.

System

Наиболее важное пространство имён. Включает в себя все примитивные типы языка C#: «пустой» тип `Void`, знаковые и беззнаковые целочисленные типы (например, `Int32`), типы чисел с плавающей запятой одинарной и двойной точности (`Single`, `Double`), «финансовый» тип `Decimal`, логический тип `Boolean`, символьный и строковый типы `Char` и `String`, а также, например, тип `DateTime` и другие. Обеспечивает также необходимым набором инструментов для работы с консолью, математическими функциями, и базовыми классами для атрибутов, исключений и массивов.

Языки платформы Microsoft .NET.

Языки программирования .NET (Языки с поддержкой CLI или CLI-языки) — компьютерные языки программирования, используемые для создания библиотек и программ, удовлетворяющих требованиям Common Language Infrastructure. За исключением некоторых серьёзных оговорок, большинство CLI-языков целиком компилируется в Common Intermediate Language (CIL), промежуточный язык, который может быть оттранслирован непосредственно в машинный код при помощи виртуальной машины Common Language Runtime (CLR), являющейся частью Microsoft .NET Framework.

Во время выполнения программы в среде CLR её CIL-код компилируется и кэшируется на лету в машинный код, соответствующий архитектуре, на которой выполняется программа. Этот последний этап может быть принудительно сокращен, а кэширование может выполняться на

предыдущем этапе при помощи «опережающего» (англ. *ahead of time*) компилятора, такого как например, ngen.exe от Microsoft или ключа «-aot» в Mono.

Одной из основных идей Microsoft .NET является совместимость программных частей, написанных на разных языках. Например, служба, написанная на C++ для Microsoft .NET, может обратиться к методу класса из библиотеки, написанной на Delphi; на C# можно написать класс, наследованный от класса, написанного на Visual Basic .NET, а исключение, созданное методом, написанным на C#, может быть перехвачено и обработано в Delphi. Каждая библиотека (сборка) в .NET имеет сведения о своей версии, что позволяет устранить возможные конфликты между разными версиями сборок.

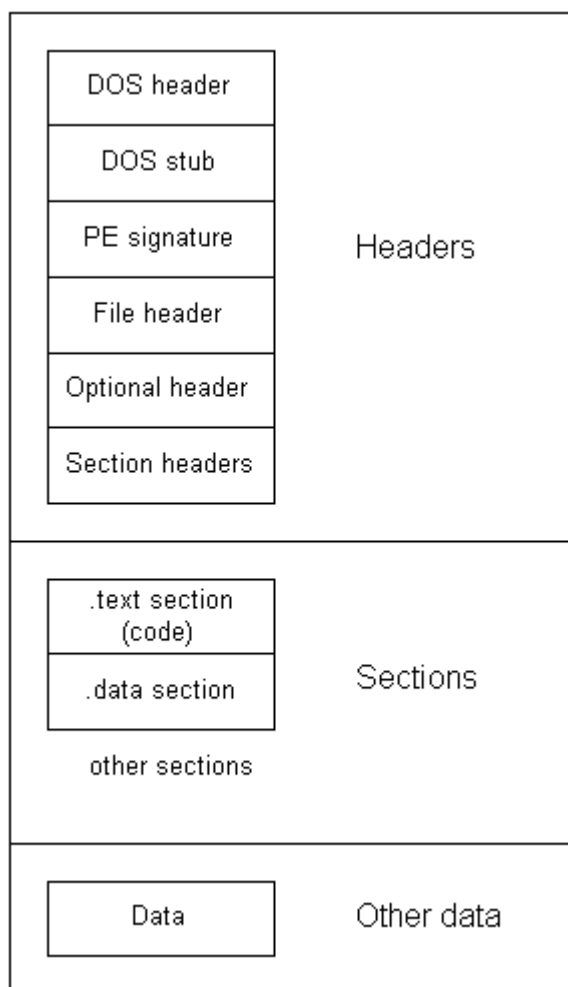
Языки, поставляемые вместе с Microsoft Visual Studio:

- C#
- Visual Basic .NET
- JScript .NET
- C++/CLI — новая версия Managed C++
- F# — член семейства языков программирования ML, включён в VS2010/VS2012/VS2015/VS2017

Схема компиляции и исполнения приложения платформы Microsoft .NET.

Создавать файлы с исходным кодом можно на любом языке, поддерживающем CLR, соответствующий компилятор проверяет синтаксис и анализирует исходный код. Независимо от компилятора результатом является управляемый модуль (managed module) – стандартный переносимый исполняемый (portable executable, PE) файл 32-разрядной (PE32) или 64-разрядной Windows (PE32+), требующий для своего выполнения CLR.

Формат PE представляет собой структуру данных, содержащую всю информацию, необходимую PE-загрузчику для отображения файла в память. Исполняемый код включает в себя ссылки для связывания динамически загружаемых библиотек, таблицы экспорта и импорта API-функций, данные для управления ресурсами и данные локальной памяти потока (TLS). В операционных системах семейства Windows NT формат PE используется для EXE, DLL, SYS (драйверов устройств) и других типов исполняемых файлов.



Файл PE состоит из нескольких заголовков и секций, которые указывают динамическому компоновщику, как отображать файл в память. Исполняемый образ состоит из нескольких различных областей (секций), каждая из которых требует различных прав доступа к памяти; таким образом, начало каждой секции должно быть выровнено по границе страницы. Например, обычно секция .text, которая содержит код программы, отображена как исполняемая/доступная только для чтения, а секция .data, содержащая глобальные переменные, отображена как неисполняемая/доступная для чтения и записи. Однако, чтобы не тратить впустую пространство на жёстком диске, различные секции на нём на границу страницы не выровнены. Часть работы динамического компоновщика состоит в том, чтобы отобразить каждую секцию в память отдельно и присвоить корректные права доступа получившимся областям согласно указаниям, содержащимся в заголовках.

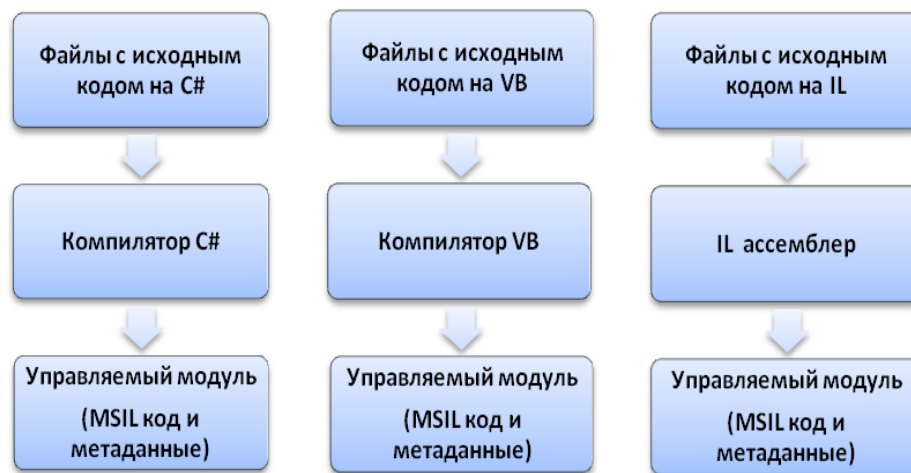


Рис. 1.

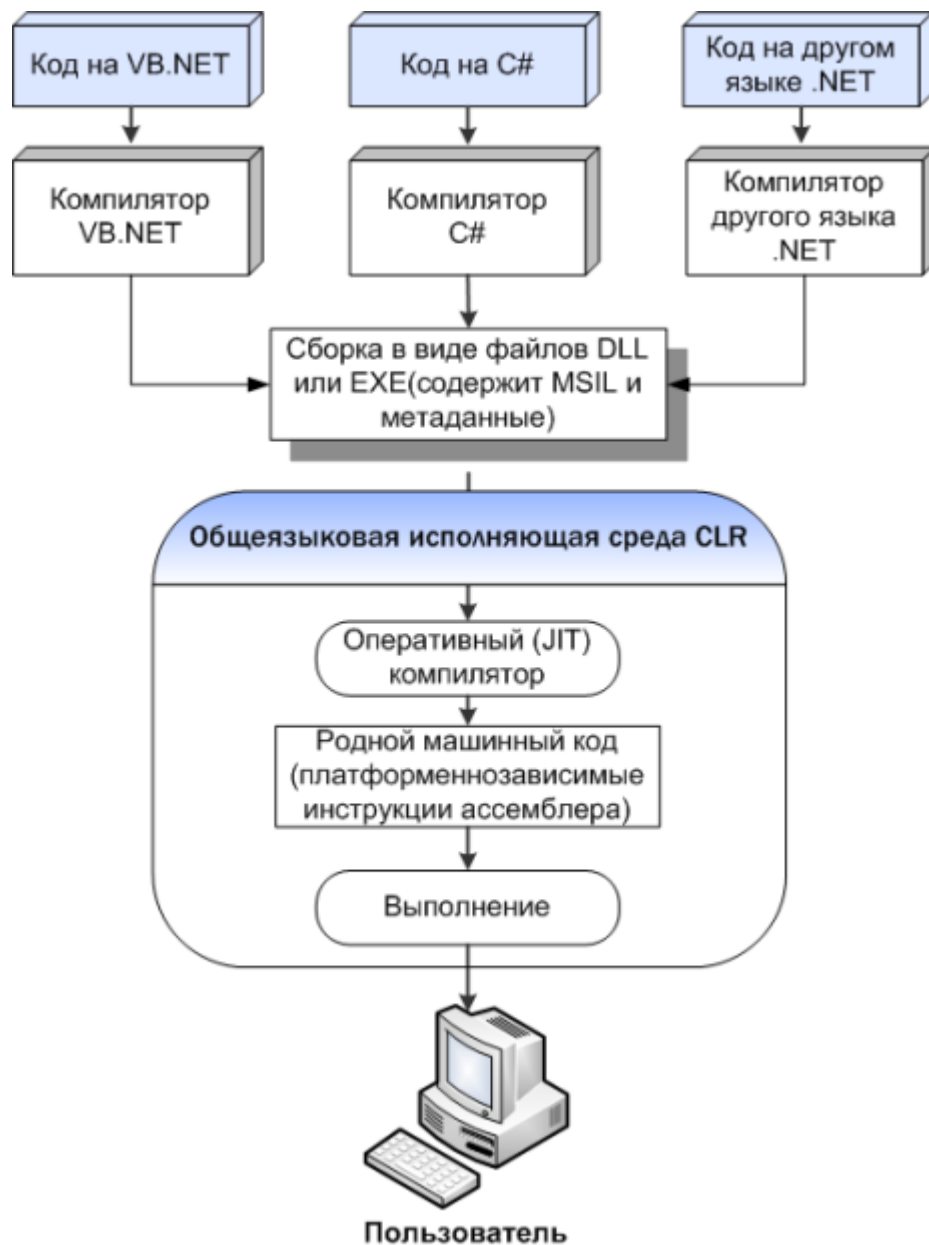
Повышение производительности

Хотя язык IL выше сравнивался с Java, все же IL на самом деле более гибкий, чем байт-код Java. Код IL всегда компилируется оперативно (Just-In-Time, ЛТ-компиляция), в то время как байт-код Java часто интерпретируется. Одним из недостатков Java было то, что во время выполнения программ процесс трансляции байт-кода Java в родной машинный код приводил к снижению производительности (за исключением самых последних версий, где Java компилируется оперативно (JIT) на некоторых платформах).

Вместо компиляции всего приложения за один проход (что может привести к задержкам при запуске), ЛТ-компилятор просто компилирует каждую порцию кода при ее вызове (т.е. оперативно). Если промежуточный код однажды скомпилирован, то результирующий машинный исполняемый код сохраняется до момента завершения работы приложения, поэтому его перекомпиляция при повторных обращениях к нему не требуется. В Microsoft аргументируют, что такой процесс более эффективен, чем компиляция всего приложения при запуске, поскольку высока вероятность того, что крупные фрагменты кода приложения на самом деле не будут выполняться при каждом запуске. При использовании ЛТ-компилятора такой код никогда не будет скомпилирован.

Это объясняет, почему можно рассчитывать на то, что выполнение управляемого кода IL будет почти настолько же быстрым, как и выполнение родного машинного кода. Однако это не объясняет того, почему Microsoft ожидает повышения производительности. Причина состоит в том, что поскольку финальная стадия компиляции происходит во время выполнения, ЛТ-компилятор на этот момент уже знает, на каком типе процессора будет запущена программа. А это значит, что он может оптимизировать финальный исполняемый код, используя инструкции конкретного машинного кода, предназначенные для конкретного процессора.

Традиционные компиляторы оптимизируют код, но они могут проводить лишь оптимизацию, не зависящую от конкретного процессора, на котором код будет выполняться. Это происходит потому, что традиционные компиляторы генерируют исполняемый код до того, как он поставляется пользователям. А потому компилятору не известно, на каком типе процессора они будут работать, за исключением самых общих характеристик вроде того, что это будет x86-совместимый процессор либо же процессор Alpha.



Многоуровневая компиляция в .NET Core

- Указывает, использует ли JIT-компилятор многоуровневую компиляцию. Многоуровневая компиляция обеспечивает переход методов по двум уровням.
 - Первый уровень создает код быстрее (быстрая JIT-компиляция) или загружает предварительно скомпилированный код (ReadyToRun).
 - Второй уровень создает оптимизированный код в фоновом режиме (JIT-компиляция с оптимизацией).
- В .NET Core 3.0 и более поздних версий многоуровневая компиляция включена по умолчанию.
- В .NET Core 2.1 и 2.2 многоуровневая компиляция отключена по умолчанию.

Язык MSIL (Microsoft Intermediate Language)

Промежуточный язык Microsoft очевидно играет фундаментальную роль в среде .NET. Теперь имеет смысл внимательнее рассмотреть основные характеристики CIL, поскольку логично, что любой язык, предназначенный для .NET, также должен поддерживать эти основные характеристики CIL.

Важнейшие свойства CIL могут быть сформулированы следующим образом:

- объектная ориентация и применение интерфейсов
- строгое различие между типами значений и типами ссылок
- строгая типизация данных
- обработка ошибок через использование исключений
- применение атрибутов

Давайте рассмотрим первые два свойства более подробно:

Поддержка объектной ориентации и интерфейсов

Независимость .NET от языка имеет некоторые практические ограничения. IL неизбежно должен воплощать некоторую определенную методологию программирования, а это означает, что исходный язык также должен быть совместим с этой методологией. Принцип, которым руководствовались в Microsoft при создании IL: классическое объектно-ориентированное программирование с реализацией одиночного наследования классов.

Так же, язык CIL имеет способность языкового взаимодействия, под которой подразумевается возможность для классов, написанных на одном языке, напрямую обращаться к классам, написанным на другом языке, т.е.:

- класс, написанный на одном языке, может быть унаследован от класса, реализованного на другом языке
- класс может содержать экземпляр другого класса, независимо от того, на каких языках написан каждый из них
- объект может напрямую вызывать методы другого объекта, написанного на другом языке
- объекты (или ссылки на объекты) могут передаваться между методами
- при вызове методов между языками можно шагать в отладчике по вызовам, даже если это означает необходимость перемещения между фрагментами исходного кода, написанными на разных языках

Различие типов значений и типов ссылок

Как и любой язык программирования, ПЛ предлагает множество предопределенных примитивных типов данных. Одна из особенностей ПЛ, однако, заключается в том, что он делает четкое различие между значениями и ссылочными типами. *Типы значений* — это те переменные, которые непосредственно хранят данные, в то время как *ссылочные типы* — это те переменные, которые просто хранят адрес, по которому соответствующие данные могут быть найдены.

В языке ПЛ также установлена своя спецификация относительно хранения данных: экземпляры ссылочных типов всегда хранятся в области памяти, известной как *managed heap*, в то время как типы значений обычно хранятся в *стеке* (хотя, если типы значений объявлены как поля внутри ссылочных типов, то они также будут сохранены в *managed heap*).

При работе со ссылочными типами необходимо учитывать следующие обстоятельства, относящиеся к производительности приложения: [2]

- память для ссылочных типов всегда выделяется из управляемой кучи; [2]
- каждый объект, размещаемый в куче, содержит дополнительные члены, подлежащие инициализации; [2] незанятые полезной информацией байты объекта обнуляются (это касается полей); [2]
- размещение объекта в управляемой куче со временем инициирует сборку мусора.

Если бы все типы были ссылочными, эффективность приложения резко упала бы.

В документации на .NET Framework можно сразу увидеть, какие типы относят к ссылочным, а какие — к значимым. Если тип называют классом (*class*), речь идет о ссылочном типе. Например, классы *System.Object*, *System.Exception*, *System.IO.FileStream* и *System.Random* — это ссылочные типы. В свою очередь, значимые типы в документации называются

структурами (structure) и перечислениями (enumeration). Например, структуры `System.Int32`, `System.Boolean`, `System.Decimal`, `System.TimeSpan` и перечисления `System.DayOfWeek`, `System.IO.FileAttributes` и `System.Drawing.FontStyle` являются значимыми типами. Все структуры являются прямыми потомками абстрактного типа `System.ValueType`, который, в свою очередь, является производным от типа `System.Object`. По умолчанию все значимые типы должны быть производными от `System.ValueType`. Все перечисления являются производными от типа `System.Enum`, производного от `System.ValueType`.

Следует объявлять тип как значимый, если верно хотя бы одно из следующих условий: [2]

- Размер экземпляров типа мал (примерно 16 байт или меньше). [2]
- Размер экземпляров типа велик (более 16 байт), но экземпляры не передаются в качестве параметров метода или не являются возвращаемыми из метода значениями.

Основное достоинство значимых типов в том, что они не размещаются в управляемой куче. Конечно, в сравнении со ссылочными типами у значимых типов есть недостатки. Важнейшие отличия между значимыми и ссылочными типами: [2]

- Объекты значимого типа существуют в двух формах: неупакованной (unboxed) и упакованной (boxed). Ссылочные типы бывают только в упакованной форме. [2]

- Значимые типы являются производными от `System.ValueType`. Этот тип имеет те же методы, что и `System.Object`. Однако `System.ValueType` переопределяет метод `Equals`, который возвращает `true`, если значения полей в обоих объектах совпадают. Кроме того, в `System.ValueType` переопределен метод `GetHashCode`, который создает хеш-код по алгоритму, учитывающему значения полей экземпляра объекта. Из-за проблем с производительностью в реализации по умолчанию, определяя собственные значимые типы значений, надо переопределить и написать свою реализацию методов `Equals` и `GetHashCode`.

- Поскольку в объявлении нового значимого или ссылочного типа нельзя указывать значимый тип в качестве базового класса, создавать в значимом типе новые виртуальные методы нельзя. Методы не могут быть абстрактными и неявно являются запечатанными (то есть их нельзя переопределить). [2]

- Переменные ссылочного типа содержат адреса объектов в куче. Когда переменная ссылочного типа создается, ей по умолчанию присваивается `null`, то есть в этот момент она не указывает на действительный объект. Попытка задействовать переменную с таким значением приведет к генерации исключения `NullReferenceException`. В то же время в переменной значимого типа всегда содержится некое значение соответствующего типа, а при инициализации всем членам этого типа присваивается 0. Поскольку переменная значимого типа не является указателем, при обращении к

значимому типу исключение `NullReferenceException` возникнуть не может. CLR поддерживает понятие значимого типа особого вида, допускающего присваивание `null` (`nullable types`).

- Когда переменной значимого типа присваивается другая переменная значимого типа, выполняется копирование всех ее полей. Когда переменной ссылочного типа присваивается переменная ссылочного типа, копируется только ее адрес. ☐

- Вследствие сказанного в предыдущем пункте несколько переменных ссылочного типа могут ссылаться на один объект в куче, благодаря чему, работая с одной переменной, можно изменить объект, на который ссылается другая переменная. В то же время каждая переменная значимого типа имеет собственную копию данных «объекта», поэтому операции с одной переменной значимого типа не влияют на другую переменную. ☐

- Так как неупакованные значимые типы не размещаются в куче, отведенная для них память освобождается сразу при возвращении управления методом, в котором описан экземпляр этого типа (в отличие от ожидания уборки мусора).

Для повышения производительности CLR дано право устанавливать порядок размещения полей типа. Например, CLR может выстроить поля таким образом, что ссылки на объекты окажутся в одной группе, а поля данных и свойства — выровненные и упакованные — в другой. Однако при описании типа можно указать, сохранить ли порядок полей данного типа, определенный программистом, или разрешить CLR выполнить эту работу. Для того чтобы сообщить CLR способ управления полями, укажите в описании класса или структуры атрибут `System.Runtime.InteropServices.StructLayoutAttribute`. Чтобы порядок полей устанавливался CLR, нужно передать конструктору атрибута параметр `LayoutKind.Auto`, чтобы сохранить установленный программистом порядок — параметр `LayoutKind.Sequential`, а параметр `LayoutKind.Explicit` позволяет разместить поля в памяти, явно задав смещения. Если в описании типа не применен атрибут `StructLayoutAttribute`, порядок полей выберет компилятор. Для ссылочных типов (классов) компилятор C# выбирает вариант `LayoutKind.Auto`, а для значимых типов (структур) — `LayoutKind.Sequential`. Очевидно, разработчики компилятора считают, что структуры обычно используются для взаимодействия с неуправляемым кодом, а значит, поля нужно расположить так, как определено разработчиком. Однако при создании значимого типа, не работающего совместно с неуправляемым кодом, скорее всего, поведение компилятора, предлагаемое по умолчанию, потребуется изменить, например:

```
using System;
using System.Runtime.InteropServices;
// Для повышения производительности разрешим CLR
// установить порядок полей для этого типа
[StructLayout(LayoutKind.Auto)]
```

```

internal struct SomeValType {
    private readonly Byte m_b;
    private readonly Int16 m_x;
    ...
}

```

Атрибут `StructLayoutAttribute` также позволяет явно задать смещение для всех полей, передав в конструктор `LayoutKind.Explicit`. Затем можно применить атрибут `System.Runtime.InteropServices.FieldOffsetAttribute` ко всем полям путем передачи конструктору этого атрибута значения типа `Int32`, определяющего смещение (в байтах) первого байта поля от начала экземпляра. Явное размещение обычно используется для имитации того, что в неуправляемом коде на C/C++ называлось объединением (`union`), то есть размещения нескольких полей с одного смещения в памяти, например:

```

using System;
using System.Runtime.InteropServices;
// Разработчик явно задает порядок полей в значимом типе
[StructLayout(LayoutKind.Explicit)]
internal struct SomeValType {
    [FieldOffset(0)]
    private readonly Byte m_b;
    // Поля m_b и m_x перекрываются
    [FieldOffset(0)]
    private readonly Int16 m_x; // в экземплярах этого класса
}

```

Понятия метаданных, манифеста, сборки.

Сборка (`assembly`) – это абстрактное понятие. Во-первых, это логическая группировка одного или нескольких управляемых модулей и файлов ресурсов. Во-вторых, это самая маленькая единица с точки зрения повторного использования, безопасности и управления версиями. Сборка может состоять из одного (однофайловая сборка) или нескольких (многофайловая сборка) файлов – все зависит от выбранных средств и компиляторов. В мире .NET сборка представляет собой то, что в других условиях называют компонентом. Некоторые управляемые модули и файлы ресурсов (или данных) создаются инструментальным средством, которое и формирует единственный PE-файл, представляющий логическую группировку файлов. При этом PE-файл содержит блок данных, называемый декларацией или манифестом (`manifest`). Манифест – один из наборов таблиц в метаданных. Эти таблицы описывают файлы, которые формируют сборку, общедоступные экспортируемые типы, реализованные в файлах сборки, а также относящиеся к сборке файлы ресурсов или данных. Если сборка является набором нескольких файлов, один из файлов сборки выбирают для хранения ее манифеста. По умолчанию компиляторы сами выполняют работу

по превращению созданного управляемого модуля в сборку, то есть компилятор С# создает управляемый модуль с манифестом, указывающим, что сборка состоит только из одного файла. Итак, в проектах, где есть только один управляемый модуль и нет файлов ресурсов (или данных), сборка и будет управляемым модулем, и не нужно прилагать дополнительных усилий по компоновке приложения. Если же надо сгруппировать набор файлов в сборку, потребуются дополнительные инструменты (вроде компоновщика сборок AL.exe) со своими параметрами командной строки (Рис. 2).

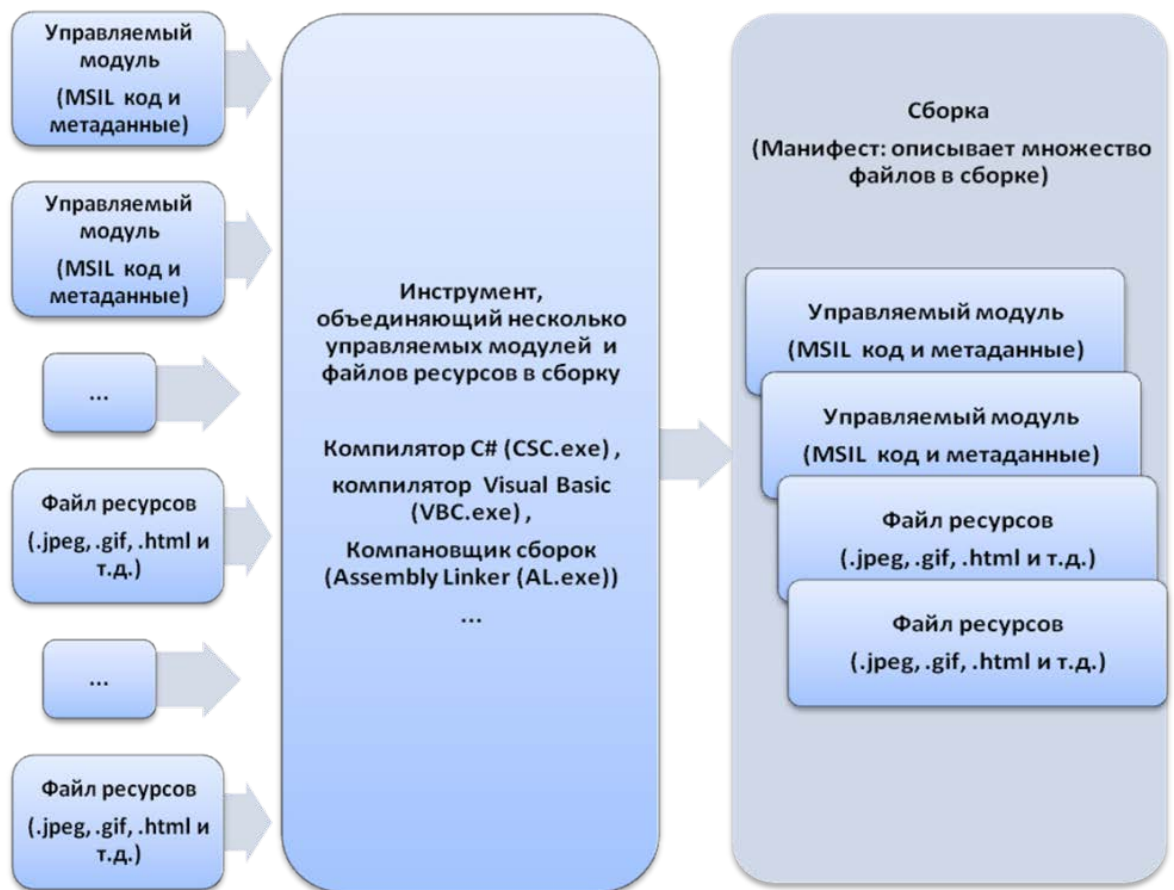


Рис. 2.

Сборка позволяет разделить логический и физический аспекты компонента, поддерживающего повторное использование, безопасность и управление версиями. Разбиение кода и ресурсов на разные файлы отдано полностью на откуп разработчика. Так, редко используемые типы и ресурсы можно вынести в отдельные файлы сборки. Отдельные файлы могут загружаться из интернета по мере надобности. Если файлы никогда не потребуются, они не будут загружаться, что сохранит место на жестком диске и ускорит установку. Сборки позволяют разбить на части процесс развертывания файлов и в то же время рассматривать все файлы как единый набор. Модули сборки также содержат сведения о других сборках, на которые они ссылаются (в том числе номера версий). Эти данные делают сборку самоописываемой (self-describing). Иначе говоря, CLR знает о сборке все, что нужно для ее выполнения. Не нужно размещать никакой

дополнительной информации ни в реестре, ни в службе каталогов Active Directory. Поэтому развертывать сборки гораздо проще, чем неуправляемые компоненты.

CLR работает со сборками – сначала всегда загружает файл с манифестом, а затем получает из манифеста имена остальных файлов сборки. Некоторые характеристики сборки стоит запомнить:

- в сборке определены повторно используемые типы;
- сборка помечена номером версии;
- со сборкой может быть связана информация безопасности.

Чаще всего сборка состоит из одного файла, но могут быть и сборки из нескольких файлов: PE-файлов с метаданными и файлов ресурсов, например .gif- или .jpg-файлов (Рис. 2). Сборка позволяет разграничить логическое и физическое понятия повторно используемых типов. Допустим, сборка состоит из нескольких типов. При этом типы, используемые чаще всех, можно поместить в один файл, а используемые реже — в другой. Если сборка развертывается путем загрузки через интернет, клиент может совсем не загружать файл с редко используемыми типами, если он никогда их не использует.

Платформа .NET разделяет сборки на локальные (или сборки со слабыми именами) и глобальные (или сборки с сильными именами). Локальные сборки размещаются в том же каталоге (или подкаталоге), что и клиентское приложение, в котором они используются. Локальные сборки обеспечивают простоту развертывания приложения (все его компоненты сосредоточены в одном месте) и изолированность компонентов. Имя локальной сборки – слабое имя – это имя файла сборки без расширения.

Хотя использование локальных сборок имеет свои преимущества, иногда необходимо сделать сборку общедоступной. До появления платформы .NET доминировал подход, при котором код общих библиотек помещался в системный каталог простым копированием файлов при установке. Такой подход привел к проблеме, известной как «ад DLL» (DLL Hell). Инсталлируемое приложение могло заменить общую библиотеку новой версией, при этом другие приложения, ориентированные на старую версию библиотеки, переставали работать. Для устранения «ада DLL» в платформе .NET используется специальное защищенное хранилище сборок (Global Assembly Cache, GAC).

Сборки, помещаемые в GAC, должны удовлетворять определенным условиям. Во-первых, такие глобальные сборки должны иметь цифровую подпись. Это исключает подмену сборок злоумышленниками. Во-вторых, для глобальных сборок отслеживаются версии и языковые культуры. Допустимой является ситуация, когда в GAC находятся разные версии одной и той же сборки, используемые разными приложениями.

Сборка, помещенная в GAC, получает сильное имя. Использование сильного имени является признаком, по которому среда исполнения понимает, что речь идет не о локальной сборке, а о сборке из GAC. Сильное имя включает: имя главного файла сборки (без расширения), версию сборки, указание о региональной принадлежности сборки и маркер открытого ключа сборки:

```
NameAssembly,Version=1.1.0.0,Culture=en,PublicKeyToken=874e23ab874e23ab
```

Номер версии сборки состоит из следующих компонент:

- Главного номера версии (Major version number)
- Второстепенного номера версии (Minor version number)
- Номера сборки (Build number)
- Номера версии (Revision number)

Часть Major является обязательной. Любая другая часть может быть опущена (в этом случае она полагается равной нулю). Часть Revision можно задать как *, тогда компилятор генерирует ее как количество секунд, прошедших с полуночи, деленное на два. Часть Build также можно задать как *. Тогда для нее будет использовано количество дней, прошедших с 1 февраля 2000 года.

Как правило, когда сборка представляется клиентам как часть приложения, следует убедиться, что она содержит информацию о версии, и что сборка подписана (signed). Управление версиями сборок важно, поскольку, в конечном счете, все строящиеся приложения будут иметь несколько выпусков (releases). Информация о версии сможет помочь определить, какие версии у клиентов уже есть, и позволит выполнять необходимые действия по обновлению приложений. Аналогичным образом информация о версии также может помочь при документировании и исправление ошибок. Подписание сборки важно, потому что оно гарантирует, что сборка не может быть легко изменена или заменена альтернативной реализацией от злоумышленников, и так как дает сборке строгое имя. Такая информация, как версия сборки и идентичность безопасности хранятся, как и метаданные, в манифесте сборки. Манифест также содержит метаданные, описывающие сферу сборки и любые ссылки на классы и ресурсы. Манифест, как правило, хранится в PE-файле.

Подписание сборки (Assembly Signing) является важным шагом, который должен включаться в свой процесс разработки, поскольку это обеспечивает следующие преимущества:

- Защищает сборки от модификаций.
- Позволяет включать подписанную сборку в глобальный кэш сборок (GAC), позволяя ее использовать другим приложениям.
- Гарантирует, что имя сборки является уникальным.

3. Введение в язык программирования C#.

Плюсы и минусы языка программирования C#

К многочисленным преимуществам этого языка относят:

- поддержку подавляющего большинства продуктов Microsoft;
- для небольших компаний и некоторых индивидуальных разработчиков бесплатными являются такие инструменты, Visual Studio, облако Azure, Windows Server, Parallels Desktop для Mac Pro и многие другие;
- большое количество “синтаксического сахара”, представляющего собой специальные конструкции, разработанные для понимания и написания кода. Они не имеют значения при компиляции;
- порог вхождения у языка C# низкий. Его синтаксис имеет много схожего с другими языками программирования, благодаря чему облегчается переход для программистов. Язык C# считается наиболее понятным и подходящим для новичков;
- после покупки Xamarin на C# можно писать программы и приложения для таких операционных систем, как iOS, Android, MacOS и Linux;
- имеется целое сообщество из опытных программистов;
- сегодня в любом регионе имеется много вакантных мест на должность C#-программиста.

Кроме многочисленных плюсов эта программа имеет некоторые недостатки. Среди них следует выделить:

- приоритетная ориентированность на Windows платформу;
- язык бесплатен только для небольших фирм, индивидуальных программистов, стартапов и учащихся . Крупной компании покупка лицензионной версии этого языка обойдется в круглую сумму;
- в языке осталась возможность использования оператора безусловного перехода.

Простейшая программа на языке программирования C#

Давайте создадим новое консольное приложение в среде Visual Studio 2010 и наберем следующий код:

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

```

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Форматируем шапку программы
            Console.BackgroundColor = ConsoleColor.Green;
            Console.ForegroundColor = ConsoleColor.Black;
            Console.WriteLine("*****");
            Console.WriteLine("**** Мой проект ****");
            Console.WriteLine("*****");
            // Основная программа
            Console.BackgroundColor = ConsoleColor.Black;
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine();
            Console.WriteLine("Hello, World!");

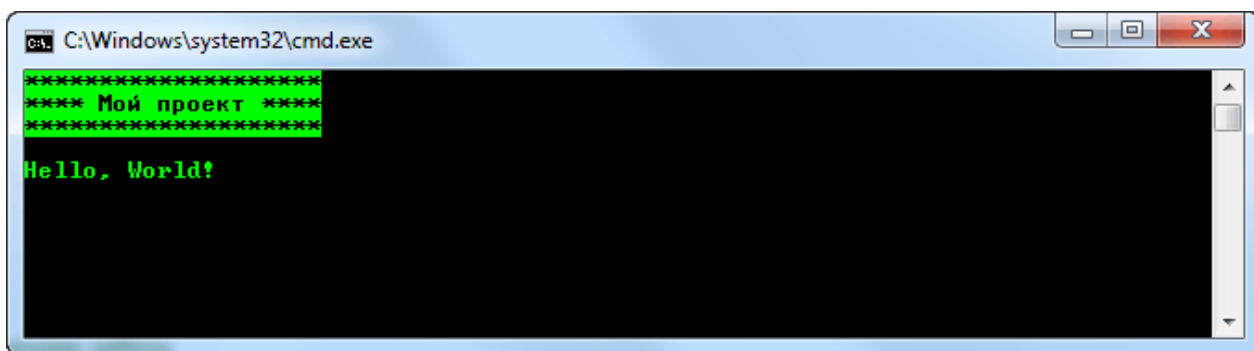
            // Ожидание нажатия клавиши Enter перед завершением работы
            Console.ReadLine();
        }
    }
}

```

В результате получилось определение типа класса, поддерживающее единственный метод по имени *Main()*. По умолчанию классу, в котором определяется метод *Main()*, в Visual Studio 2010 назначается имя *Program*; при желании это имя легко изменить. Класс, определяющий метод *Main()*, должен обязательно присутствовать в каждом исполняемом приложении на С# (будь то консольная программа, настольная программа для Windows или служба Windows), поскольку он применяется для обозначения точки входа в приложение.

Чтобы посмотреть результат работы данного приложения на этапе отладки, в программной среде Visual Studio 2010 необходимо нажать кнопку

"Начать отладку" или нажать <F5>. Результатом отладки этого приложения будет консольное окно следующего вида



проанализируем данную программу более подробно, начиная с ее имени.

В отличие от ряда других языков программирования, и в особенности Java, где имя файла программы имеет большое значение, имя программы на C# может быть произвольным. Visual Studio по умолчанию задает имя Program.cs, при этом в C# файл с исходным текстом этой программы можно было бы назвать как угодно. Например, его можно было назвать Sample.cs, Test.cs или даже X.cs.

Обратите внимание, что в файлах с исходным текстом программ на C# условно принято расширение **.cs**, и это условие вы должны соблюдать. Кроме того, многие программисты называют файлы с исходным текстом своих программ по имени основного класса, определенного в программе.

Рассмотрим первую строку программы:

```
using System;
```

Эта строка означает, что в программе используется пространство имен *System*. В C# *пространство имен* определяет область объявлений. Благодаря пространству имен одно множество имен отделяется от других. По существу, имена, объявляемые в одном пространстве имен, не вступают в конфликт с именами, объявляемыми в другом пространстве имен. В анализируемой программе используется пространство имен System, которое зарезервировано для элементов, связанных с библиотекой классов среды .NET Framework, применяемой в C#. **Ключевое слово using** просто констатирует тот факт, что в программе используются имена в заданном пространстве имен. (Попутно обратим внимание на весьма любопытную возможность создавать собственные пространства имен, что особенно полезно для работы над крупными проектами.)

Хочу обратить ваше внимание, что использование данной строки в программе не обязательно, т.к. в C# можно всегда полностью определить имя

с помощью пространства имен, к которому оно принадлежит. Например, строку

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

можно переписать следующим образом

```
System.Console.WriteLine("Hello, World!");
```

Однако указывать пространство имен `System` всякий раз, когда используется член этого пространства, — довольно утомительное занятие, и поэтому большинство программистов на `C#` вводят директиву `using System` в начале своих программ. Следует, однако, иметь в виду, что любое имя можно всегда определить, явно указав его пространство имен, если в этом есть необходимость.

С помощью **ключевого слова** `namespace` объявляется пространство имен, с которым должен быть ассоциирован класс. Весь код в последующих фигурных скобках рассматривается как принадлежащий этому пространству имен. Оператор `using` специфицирует пространство имен, которое должен просматривать компилятор в поисках классов, упомянутых в коде, но не определенных в текущем пространстве имен. Это служит тем же целям, что оператор `import` в `Java` и `using namespace` в `C++`.

Перейдем к следующей строке программы:

```
class Program
```

В этой строке **ключевое слово** `class` служит для объявления вновь определяемого класса. Класс является основной единицей инкапсуляции в `C#`, а `Program` — это имя класса. Определение класса начинается с открывающей фигурной скобки `"{"` и оканчивается закрывающей фигурной скобкой `"}"`. Элементы, заключенные в эти фигурные скобки, являются членами класса. Не вдаваясь пока что в подробности, достаточно сказать, что в `C#` большая часть действий, выполняемых в программе, происходит именно в классе.

Метод `Main()`, как уже говорилось, является точкой входа в программу. Формально класс, в котором определяется метод `Main()`, называется *объектом приложения*. Хотя в одном исполняемом приложении допускается иметь более одного такого объекта (это может быть удобно при проведении модульного тестирования), при этом обязательно необходимо информировать компилятор о том, какой из методов `Main()` должен использоваться в качестве входной точки. Для этого нужно либо указать опцию `main` в командной строке, либо выбрать соответствующий вариант в раскрывающемся списке на вкладке `Application` (Приложение) окна редактора свойств проекта в `Visual Studio 2010`.

Обратите внимание, что в сигнатуре метода `Main()` присутствует ключевое слово `static`. Область действия статических (`static`) членов охватывает уровень всего класса (а не уровень отдельного объекта) и потому они могут вызываться без предварительного создания нового экземпляра класса.

Внутри метода `Main()` используется несколько встроенных методов предопределенного класса **Console**, в частности *`BackgroundColor`* - задает цвет фона, *`ForegroundColor`* - задает цвет контента (в нашем случае текста), *`WriteLine()`* - выводит на экран строку и *`ReadLine()`* - считывает данные из консоли.

4. Рефлекторы и дотфускаторы.

Рефлектор (Reflection) - позволяет открыть уже от компилирование .NET приложения и посмотреть его исходный код

Утилита ildasm.exe (Intermediate Language Disassembler – дизассемблер IL), поставляемая в составе пакета .NET Framework 4 SDK, позволяет загружать любую сборку .NET и изучать ее содержимое, в том числе ассоциируемый с ней манифест, IL-код и метаданные типов (Рис. 3).

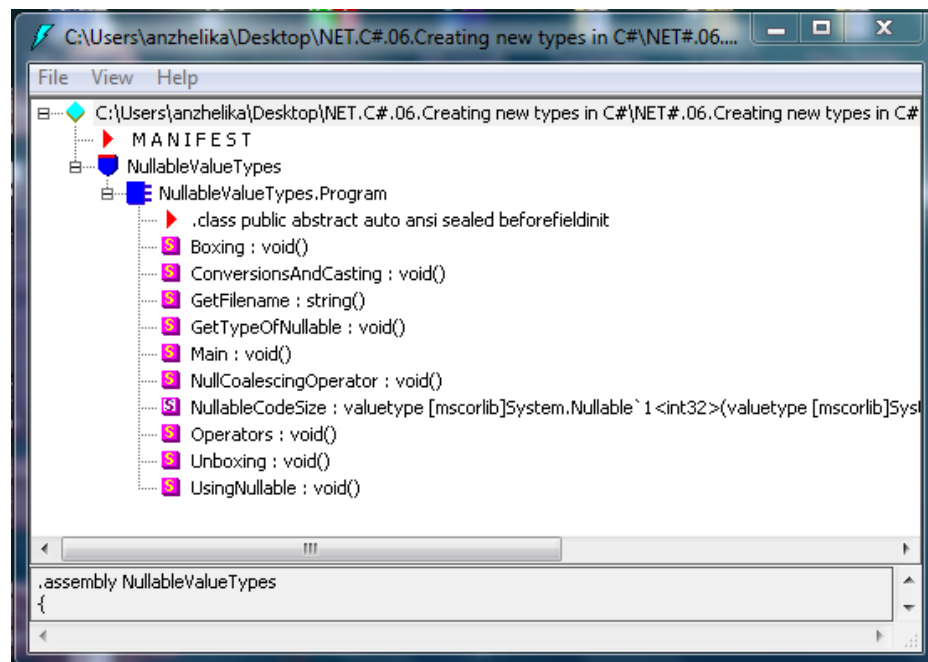
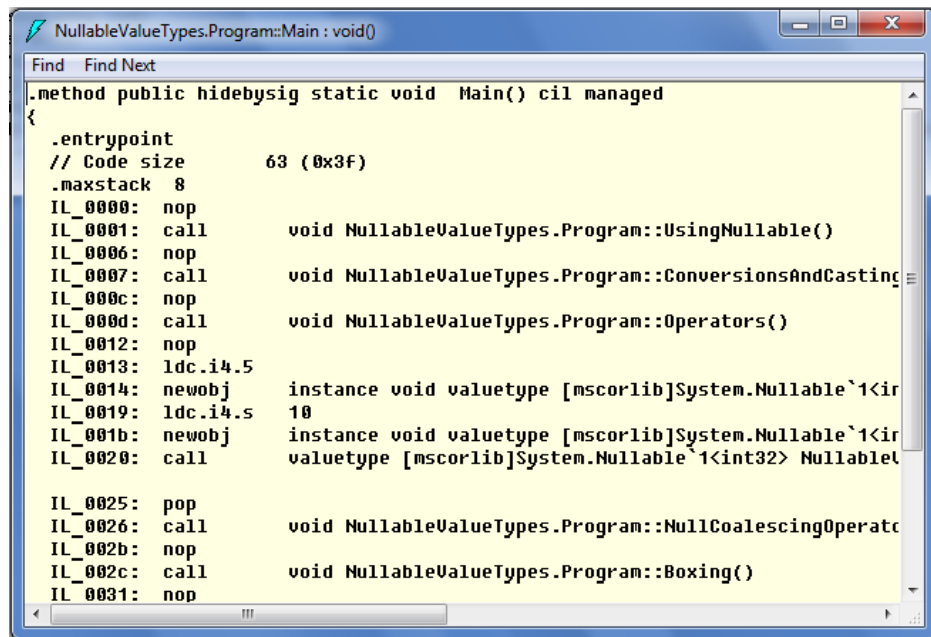


Рис. 3.

Помимо содержащихся в сборке пространств имен, типов и членов, утилита ildasm.exe позволяет просматривать IL-инструкции, которые лежат в основе каждого конкретного члена. Например, в результате двойного щелчка на методе открывается отдельное окно с IL-кодом, лежащим в основе этого метода (Рис. 4.).

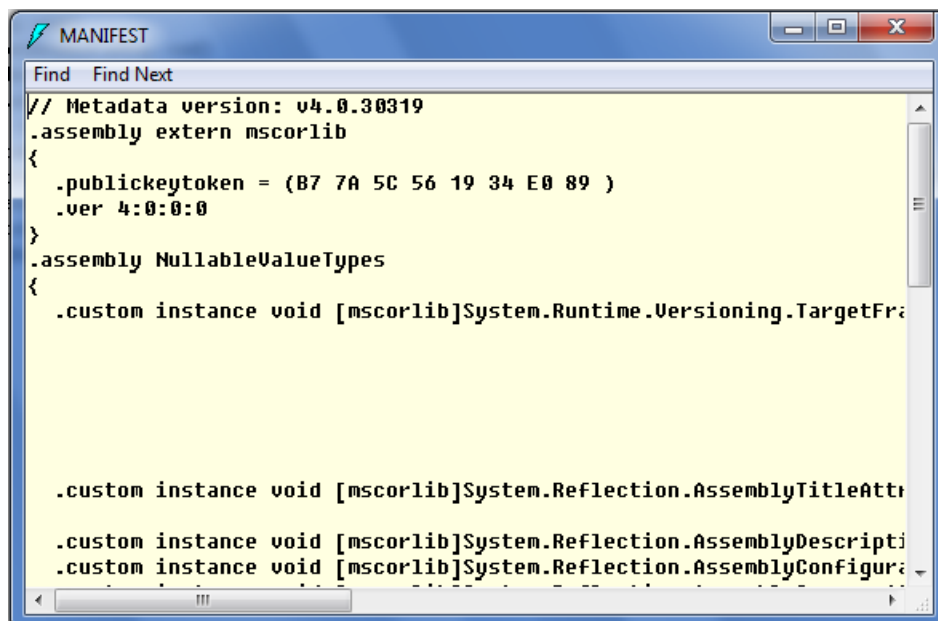
Помимо содержащихся в сборке пространств имен, типов и членов, утилита ildasm.exe позволяет просматривать IL-инструкции, которые лежат в основе каждого конкретного члена. Например, в результате двойного щелчка на методе открывается отдельное окно с IL-кодом, лежащим в основе этого метода (Рис. 4.).



```
NullableValueTypes.Program::Main : void()
Find Find Next
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size      63 (0x3f)
    .maxstack 8
    IL_0000: nop
    IL_0001: call        void NullableValueTypes.Program::UsingNullable()
    IL_0006: nop
    IL_0007: call        void NullableValueTypes.Program::ConversionsAndCasting()
    IL_000c: nop
    IL_000d: call        void NullableValueTypes.Program::Operators()
    IL_0012: nop
    IL_0013: ldc.i4.5
    IL_0014: newobj      instance void valuetype [mscorlib]System.Nullable`1<ir
    IL_0019: ldc.i4.5
    IL_001b: newobj      instance void valuetype [mscorlib]System.Nullable`1<ir
    IL_0020: call        valuetype [mscorlib]System.Nullable`1<int32> Nullable
    IL_0025: pop
    IL_0026: call        void NullableValueTypes.Program::NullCoalescingOperato
    IL_002b: nop
    IL_002c: call        void NullableValueTypes.Program::Boxing()
    IL_0031: nop
}
```

Рис. 4.

Для просмотра метаданных типов, которые содержатся в загруженной в текущий момент сборке, необходимо нажать комбинацию клавиш <Ctrl+M>. Чтобы просмотреть содержимое манифеста сборки, необходимо дважды щелкнуть на значке MANIFEST (Рис. 5.).



```
MANIFEST
Find Find Next
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
.assembly NullableValueTypes
{
    .custom instance void [mscorlib]System.Runtime.Versioning.TargetFra

    .custom instance void [mscorlib]System.Reflection.AssemblyTitleAttr

    .custom instance void [mscorlib]System.Reflection.AssemblyDescripti
    .custom instance void [mscorlib]System.Reflection.AssemblyConfigura
```

Рис. 5.

Хотя утилита `ildasm.exe` и применяется очень часто для просмотра деталей двоичного файла .NET, одним из ее недостатков является то, что она позволяет просматривать только лежащий в основе IL-код, но не реализацию сборки с использованием предпочитаемого управляемого языка. Существует множество других утилит для просмотра и декомпиляции объектов .NET, в том числе и популярная утилита Reflector¹ (Рис. 6).

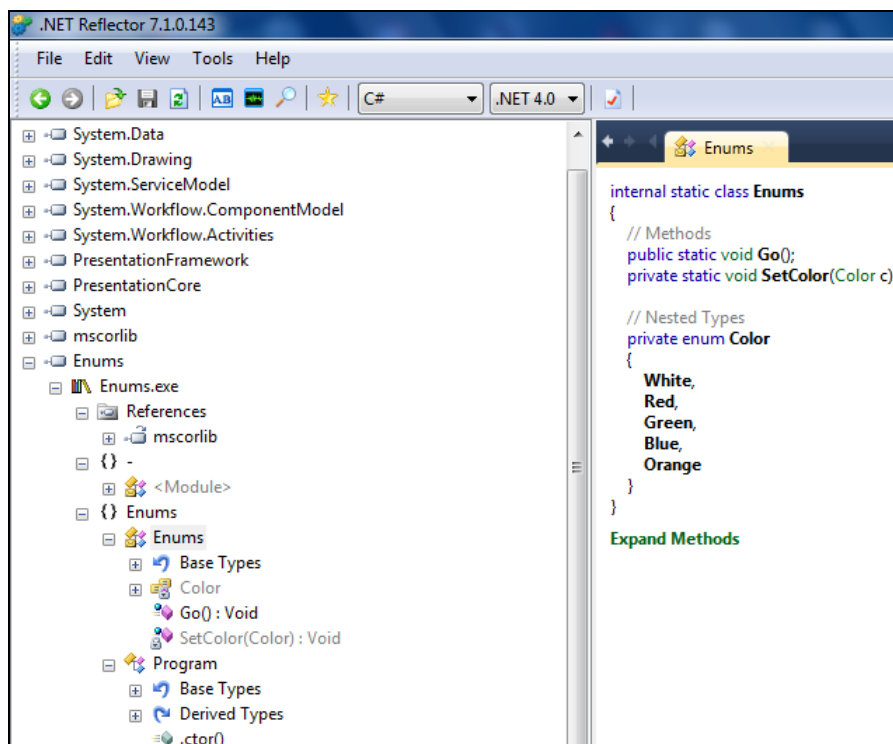


Рис. 6.

Чтобы защитить свое .NET приложения используют **Дотфускатор (dotfuscators)**.

Дотфускатор изменяет исходный код приложения таким образом что после этого его не возможно будет разобрать хотя и возможно.

Дотфускатор усложняет процесс обратной-инженерии, с одной стороны, чем защищает интеллектуальные права разработчика и его коммерческую тайну. С другой стороны, процесс запутывания усложняет работу по анализу средств безопасности приложения, что уменьшает вероятность взлома программного решения пиратам.

Общие выводы и рассуждения

- Бесплатные обфускаторы весьма слабые и пригодны только для простого переименования.
- Существуют весьма неплохие решения (control flow, MSIL encryption) стоимостью до \$500;

¹ .NET Reflector – платная утилита для Microsoft .NET, комбинирующая браузер классов, статический анализатор и декомпилятор, изначально написанная Lutz Roeder. 20 августа 2008 Red Gate Software объявили, что они берут ответственность за дальнейшую разработку программы. MSDN Magazine назвал ее одной из десяти «Must-Have» утилит для разработчиков, Scott Hanselman включил ее в свой «Big Ten Life and Work-Changing Utilities» (примерный перевод: большая десятка утилит, меняющих жизнь и работу).

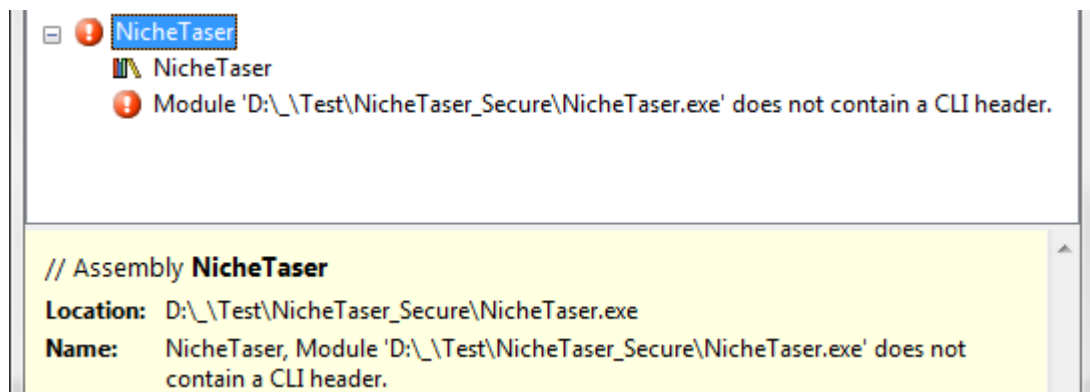
- Существуют решения, которые стоят около 5000, но к сожалению, для многих из них есть распаковщики. Некоторые из них взломаны.

Примеры: .NET Reactor, CodeVeil, CodeWall, dotNetProtector, VMWare ThinApp.

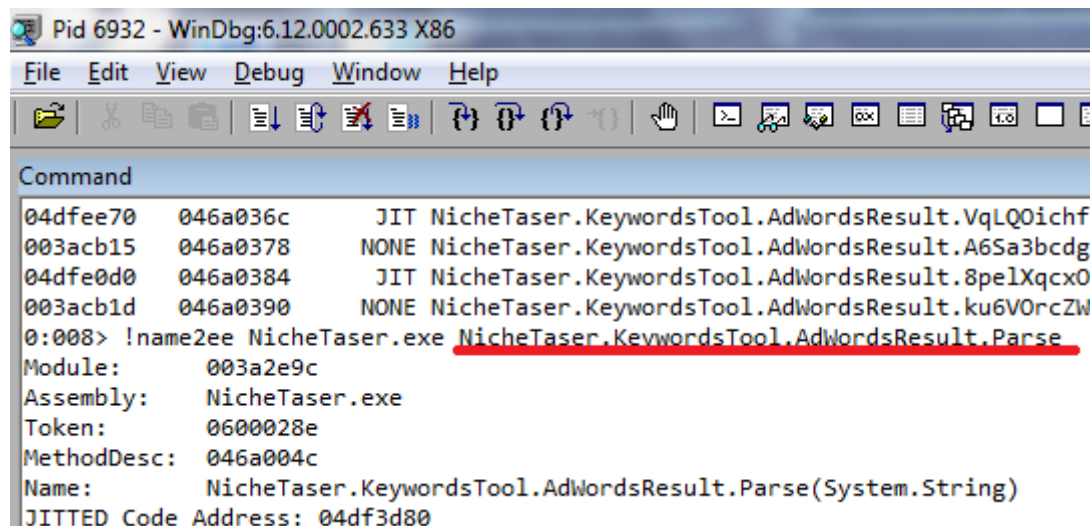
.NET Reactor

Шифрует код при помощи NecroBit (название их технологии). Код разобрать не получилось, WinDbg тоже ничего не нашёл.

Reflector:



Что-то извлечь при помощи WinDbg можно, но IL-код методов не отдаётся.



CodeWall, dotNetProtector

Продукты одного класса, подходят, если требуется защита кода от дальнейшего рефакторинга и излишнего любопытства.

dotNetProtector

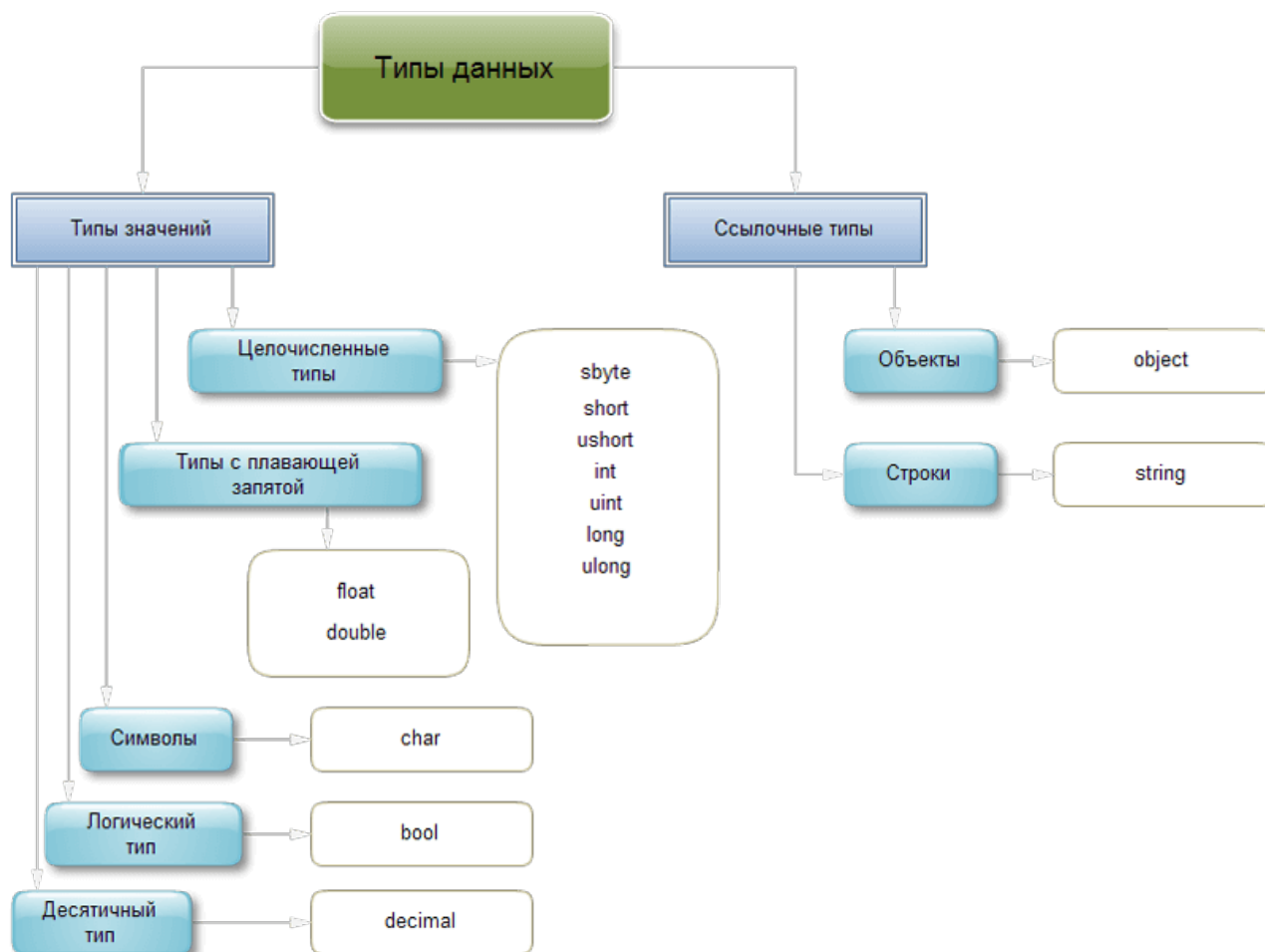
Сборка получается достаточно хорошая, но со сборкой появляются много DLL от этого обфускатора.

5. Типы данных

Типы данных имеют особенное значение в С#, поскольку это строго типизированный язык. Это означает, что все операции подвергаются строгому контролю со стороны компилятора на соответствие типов, причем недопустимые операции не компилируются. Следовательно, строгий контроль типов позволяет исключить ошибки и повысить надежность программ. Для обеспечения контроля типов все переменные, выражения и значения должны принадлежать к определенному типу. Такого понятия, как "бестиповая" переменная, в данном языке программирования вообще не существует. Более того, тип значения определяет те операции, которые разрешается выполнять над ним. Операция, разрешенная для одного типа данных, может оказаться недопустимой для другого.

В С# имеются две общие категории встроенных типов данных: **типы значений** и **ссылочные типы**. Они отличаются по содержимому переменной. Концептуально разница между ними состоит в том, что тип значения (value type) хранит данные непосредственно, в то время как ссылочный тип (reference type) хранит ссылку на значение.

Эти типы сохраняются в разных местах памяти: типы значений сохраняются в области, известной как стек, а ссылочные типы — в области, называемой управляемой кучей.



Целочисленные типы данных

В C# определены девять целочисленных типов: *char*, *byte*, *sbyte*, *short*, *ushort*, *int*, *uint*, *long* и *ulong*. Но тип *char* применяется, главным образом, для представления символов и поэтому рассматривается отдельно. Остальные восемь целочисленных типов предназначены для числовых расчетов. Ниже представлены их диапазон представления чисел и разрядность в битах:

Целочисленные типы C#

Тип	Тип CTS	Разрядность в битах	Диапазон
byte	System.Byte	8	0:255
sbyte	System.SByte	8	-128:127
short	System.Int16	16	-32768 : 32767
ushort	System.UInt16	16	0 : 65535

int	System.Int32	32	-2147483648 : 2147483647	
uint	System.UInt32	32	0 : 4294967295	
long	System.Int64	64	-9223372036854775808 9223372036854775807	:
ulong	System.UInt64	64	0 : 18446744073709551615	

Как следует из приведенной выше таблицы, в С# определены оба варианта различных целочисленных типов: со знаком и без знака. Целочисленные типы со знаком отличаются от аналогичных типов без знака способом интерпретации старшего разряда целого числа. Так, если в программе указано целочисленное значение со знаком, то компилятор С# сгенерирует код, в котором старший разряд целого числа используется в качестве флага знака. Число считается положительным, если флаг знака равен 0, и отрицательным, если он равен 1.

Отрицательные числа практически всегда представляются методом дополнения до двух, в соответствии с которым все двоичные разряды отрицательного числа сначала инвертируются, а затем к этому числу добавляется 1.

Вероятно, самым распространенным в программировании целочисленным типом является **тип int**. Переменные типа `int` нередко используются для управления циклами, индексирования массивов и математических расчетов общего назначения. Когда же требуется целочисленное значение с большим диапазоном представления чисел, чем у типа `int`, то для этой цели имеется целый ряд других целочисленных типов.

Так, если значение нужно сохранить без знака, то для него можно выбрать **тип uint**, для больших значений со знаком — **тип long**, а для больших значений без знака — **тип ulong**. В качестве примера ниже приведена программа, вычисляющая расстояние от Земли до Солнца в сантиметрах. Для хранения столь большого значения в ней используется переменная типа `long`:

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

```
namespace ConsoleApplication1
{
    class Program
```

```

{
    static void Main(string[] args)
    {
        long result;
        const long km = 149800000; // расстояние в км.

        result = km * 1000 * 100;
        Console.WriteLine(result);
        Console.ReadLine();
    }
}

```

Всем целочисленным переменным значения могут присваиваться в десятичной или шестнадцатеричной системе обозначений. В последнем случае требуется префикс 0x:

```

uint ui = 1234U;
long l = 1234L;
ulong ul = 1234UL;

```

U и L можно также указывать в нижнем регистре, хотя строчную L легко зрительно спутать с цифрой 1 (единица).

Типы данных для чисел с плавающей точкой.

Типы с плавающей точкой позволяют представлять числа с дробной частью. В С# имеются две разновидности типов данных с плавающей точкой: **float** и **double**. Они представляют числовые значения с одинарной и двойной точностью соответственно. Так, разрядность типа float составляет 32 бита, что приблизительно соответствует диапазону представления чисел от 5E-45 до 3,4E+38. А разрядность типа double составляет 64 бита, что приблизительно соответствует диапазону представления чисел от 5E-324 до 1,7E+308.

Тип данных float предназначен для меньших значений с плавающей точкой, для которых требуется меньшая точность. Тип данных double больше, чем float, и предлагает более высокую степень точности (15 разрядов).

Если нецелочисленное значение жестко кодируется в исходном тексте (например, 12.3), то обычно компилятор предполагает, что подразумевается значение типа double. Если значение необходимо специфицировать как float, потребуется добавить к нему символ F (или f):

```
float f = 12.3F;
```

Для представления чисел с плавающей точкой высокой точности предусмотрен также десятичный тип **decimal**, который предназначен для применения в финансовых расчетах. Этот тип имеет разрядность 128 бит для представления числовых значений в пределах от 1E-28 до 7,9E+28. Вам, вероятно, известно, что для обычных арифметических вычислений с плавающей точкой характерны ошибки округления десятичных значений. Эти ошибки исключаются при использовании типа decimal, который позволяет представить числа с точностью до 28 (а иногда и 29) десятичных разрядов. Благодаря тому что этот тип данных способен представлять десятичные значения без ошибок округления, он особенно удобен для расчетов, связанных с финансами:

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

```
namespace ConsoleApplication1
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            // *** Расчет стоимости капиталовложения с ***
            // *** фиксированной нормой прибыли***
            decimal money, percent;
            int i;
            const byte years = 15;

            money = 1000.0m;
            percent = 0.045m;

            for (i = 1; i <= years; i++)
            {
                money *= 1 + percent;
            }

            Console.WriteLine("Общий доход за {0} лет: {1} $$",years,money);
            Console.ReadLine();
        }
    }
}
```


Символьный тип данных

В С# символы представлены не 8-разрядным кодом, как во многих других языках программирования, например C++, а 16-разрядным кодом, который называется *юникодом (Unicode)*. В юникоде набор символов представлен настолько широко, что он охватывает символы практически из всех естественных языков на свете. Если для многих естественных языков, в том числе английского, французского и немецкого, характерны относительно небольшие алфавиты, то в ряде других языков, например китайском, употребляются довольно обширные наборы символов, которые нельзя представить 8-разрядным кодом. Для преодоления этого ограничения в С# определен тип **char**, представляющий 16-разрядные значения без знака в пределах от 0 до 65 535. При этом стандартный набор символов в 8-разрядном коде ASCII является подмножеством юникода в пределах от 0 до 127. Следовательно, символы в коде ASCII по-прежнему остаются действительными в С#.

Для того чтобы присвоить значение символьной переменной, достаточно заключить это значение (т.е. символ) в *одинарные кавычки*:

```
char ch;
```

```
ch = 'Z';
```

Несмотря на то что тип `char` определен в С# как целочисленный, его не следует путать со всеми остальными целочисленными типами. Дело в том, что в С# отсутствует автоматическое преобразование символьных значений в целочисленные и обратно. Например, следующий фрагмент кода содержит ошибку:

```
char ch;
```

```
ch = 8; // ошибка, не выйдет
```

Наравне с представлением `char` как символьных литералов, их можно представлять как 4-разрядные шестнадцатеричные значения Unicode (например, `"\u0041"`), целочисленные значения с приведением (например, `(char) 65`) или же шестнадцатеричные значения (например, `"\x0041"`). Кроме того, они могут быть представлены в виде управляющих последовательностей.

Другие типы данных.

Тип `bool` представляет два логических значения: "истина" и "ложь". Эти логические значения обозначаются в C# зарезервированными словами *true* и *false* соответственно. Следовательно, переменная или выражение типа `bool` будет принимать одно из этих логических значений. Кроме того, в C# не определено взаимное преобразование логических и целых значений. Например, 1 не преобразуется в значение `true`, а 0 — в значение `false`.

6. Литералы

В C# **литералами** называются постоянные значения, представленные в удобной для восприятия форме. Например, число 100 является литералом. Сами литералы и их назначение настолько понятны, что они применялись во всех предыдущих примерах программ без всяких пояснений. Но теперь настало время дать им формальное объяснение.

В C# литералы могут быть любого простого типа. Представление каждого литерала зависит от конкретного типа. Как пояснялось ранее, символьные литералы заключаются в одинарные кавычки. Например, 'a' и '%' являются символьными литералами.

Целочисленные литералы указываются в виде чисел без дробной части. Например, 10 и -100 — это целочисленные литералы. Для обозначения литералов с плавающей точкой требуется указывать десятичную точку и дробную часть числа. Например, 11.123 — это литерал с плавающей точкой. Для вещественных чисел с плавающей точкой в C# допускается также использовать экспоненциальное представление.

У литералов должен быть также конкретный тип, поскольку C# является строго типизированным языком. В этой связи возникает естественный вопрос: к какому типу следует отнести числовой литерал, например 2, 12 3987 или 0.23? К счастью, для ответа на этот вопрос в C# установлен ряд простых для соблюдения правил:

- Во-первых, у целочисленных литералов должен быть самый мелкий целочисленный тип, которым они могут быть представлены, начиная с типа `int`. Таким образом, у целочисленных литералов может быть один из следующих типов: `int`, `uint`, `long` или `ulong` в зависимости от значения литерала.
- Литералы с плавающей точкой относятся к типу `double`.
- Если вас не устраивает используемый по умолчанию тип литерала, вы можете явно указать другой его тип с помощью **суффикса**.

Так, для указания типа `long` к литералу присоединяется суффикс `l` или `L`. Например, 12 — это литерал типа `int`, а 12L — литерал типа `long`. Для указания целочисленного типа без знака к литералу присоединяется суффикс `u` или `U`. Следовательно, 100 — это литерал типа `int`, а 100U — литерал типа `uint`. А для указания длинного целочисленного типа без знака к литералу присоединяется суффикс `ul` или `UL`. Например, 984375UL — это литерал типа `ulong`.

Кроме того, для указания типа `float` к литералу присоединяется суффикс `F` или `f`. Например, 10.19F — это литерал типа `float`. Можете даже указать тип `double`, присоединив к литералу суффикс `d` или `D`, хотя это

излишне. Ведь, как упоминалось выше, по умолчанию литералы с плавающей точкой относятся к типу `double`.

И наконец, для указания типа `decimal` к литералу присоединяется суффикс `m` или `M`. Например, `9.95M` — это десятичный литерал типа `decimal`.

Несмотря на то что целочисленные литералы образуют по умолчанию значения типа `int`, `uint`, `long` или `ulong`, их можно присваивать переменным типа `byte`, `sbyte`, `short` или `ushort`, при условии, что присваиваемое значение может быть представлено целевым типом.

Шестнадцатеричные литералы

Вам, вероятно, известно, что в программировании иногда оказывается проще пользоваться системой счисления по основанию 16, чем по основанию 10. Система счисления по основанию 16 называется *шестнадцатеричной*. В ней используются числа от 0 до 9, а также буквы от A до F, которыми обозначаются десятичные числа 10, 11, 12, 13, 14 и 15. Например, десятичному числу 16 соответствует шестнадцатеричное число 10. Вследствие того что шестнадцатеричные числа применяются в программировании довольно часто, в C# разрешается указывать целочисленные литералы в шестнадцатеричном формате. Шестнадцатеричные литералы должны начинаться с **символов 0x**, т.е. нуля и последующей латинской буквы "икс". Ниже приведены некоторые примеры шестнадцатеричных литералов:

```
count = 0xFF; // равно 255 в десятичной системе
```

```
incr = 0x1a; // равно 26 в десятичной системе
```

Управляющие последовательности символов

Большинство печатаемых символов достаточно заключить в одинарные кавычки, но набор в текстовом редакторе некоторых символов, например возврата каретки, вызывает особые трудности. Кроме того, ряд других символов, в том числе одинарные и двойные кавычки, имеют специальное назначение в C#, поэтому их нельзя использовать непосредственно. По этим причинам в C# предусмотрены специальные **управляющие последовательности символов**:

Управляющие последовательности C#

Управляющая последовательность	Описание
-----------------------------------	----------

<code>\a</code>	Звуковой сигнал (звонок)
-----------------	--------------------------

<code>\b</code>	Возврат на одну позицию
<code>\f</code>	Перевод страницы (переход на новую страницу)
<code>\n</code>	Новая строка (перевод строки)
<code>\r</code>	Возврат каретки
<code>\t</code>	Горизонтальная табуляция
<code>\v</code>	Вертикальная табуляция
<code>\0</code>	Пустой символ
<code>\'</code>	Одинарная кавычка
<code>\"</code>	Двойная кавычка
<code>\\</code>	Обратная косая черта

Строковые литералы

В C# поддерживается еще один тип литералов — **строковый**. Строковый литерал представляет собой набор символов, заключенных в двойные кавычки. Например следующий фрагмент кода:

```
"This is text"
```

Помимо обычных символов, строковый литерал может содержать одну или несколько управляющих последовательностей символов, о которых речь шла выше. Также можно указать **буквальный строковый литерал**. Такой литерал начинается с символа `@`, после которого следует строка в кавычках. Содержимое строки в кавычках воспринимается без изменений и может быть расширено до двух и более строк. Это означает, что в буквальный строковый литерал можно включить символы новой строки, табуляции и прочие, не прибегая к управляющим последовательностям. Единственное исключение составляют двойные кавычки (`"`), для указания которых необходимо использовать двойные кавычки с обратным слэшем (`"\"`). Например:

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

```

using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Используем перенос строки
            Console.WriteLine("Первая строка\nВторая строка\nТретья строка\n");

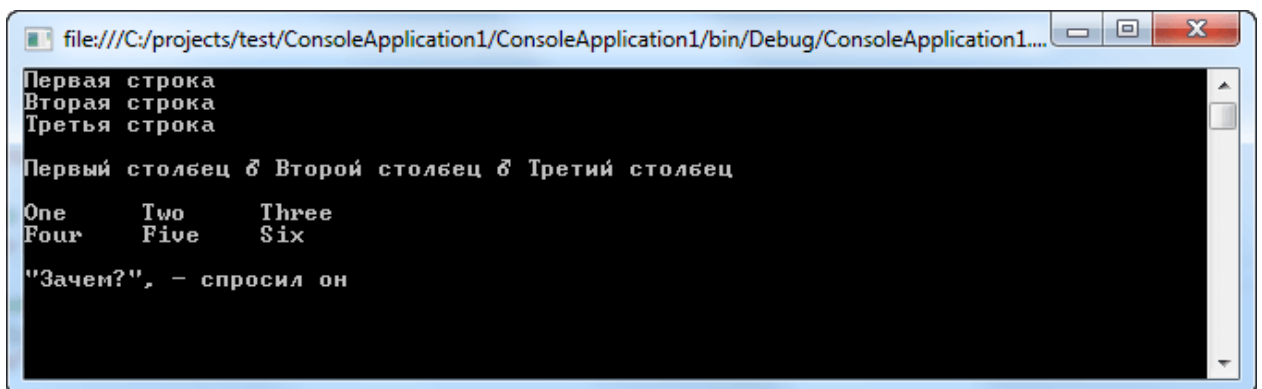
            // Используем вертикальную табуляцию
            Console.WriteLine("Первый столбец \v Второй столбец \v Третий столбец \n");

            // Используем горизонтальную табуляцию
            Console.WriteLine("One\tTwo\tThree");
            Console.WriteLine("Four\tFive\tSix\n");

            //Вставляем кавычки
            Console.WriteLine("\"Зачем?\", - спросил он");

            Console.ReadLine();
        }
    }
}

```



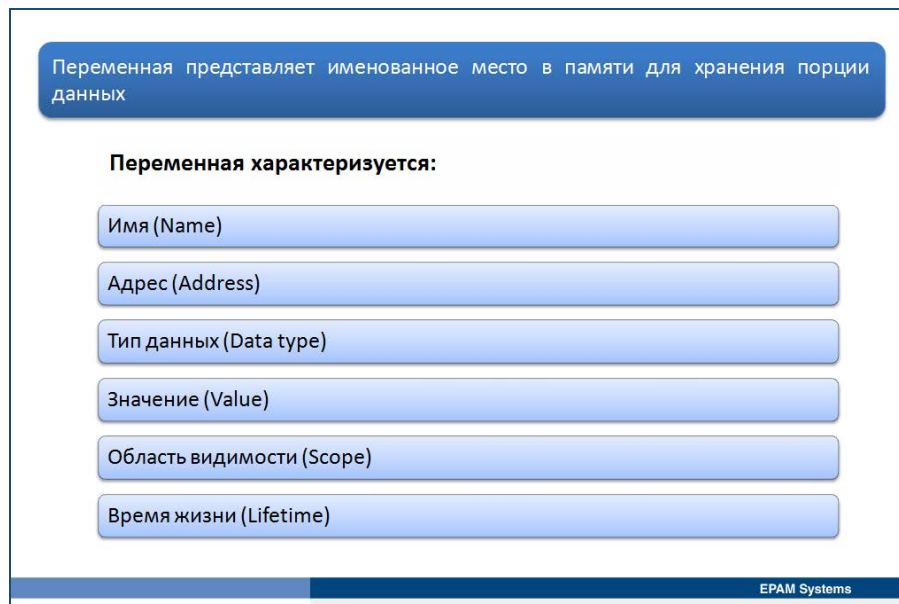
```

file:///C:/projects/test/ConsoleApplication1/ConsoleApplication1/bin/Debug/ConsoleApplication1....
Первая строка
Вторая строка
Третья строка
Первый столбец \v Второй столбец \v Третий столбец
One\tTwo\tThree
Four\tFive\tSix
"Зачем?", - спросил он

```

7. Переменные

Понятие переменной



Переменная представляет именованное место в памяти для хранения порции данных. Приложение может получить доступ к данным с помощью переменной, с которой они связаны. Часто при выполнении расчетов или передаче данных между пользователем, приложением и базой данных необходимо временно хранить значения. Например, можно получить несколько значений из базы данных, сравнить их, и выполнить различные операции над ними в зависимости от результатов сравнения. Переменные хранят значения, которые приложение может изменить во время его работы.

На переменную можно взглянуть с разных сторон. Переменная характеризуется:

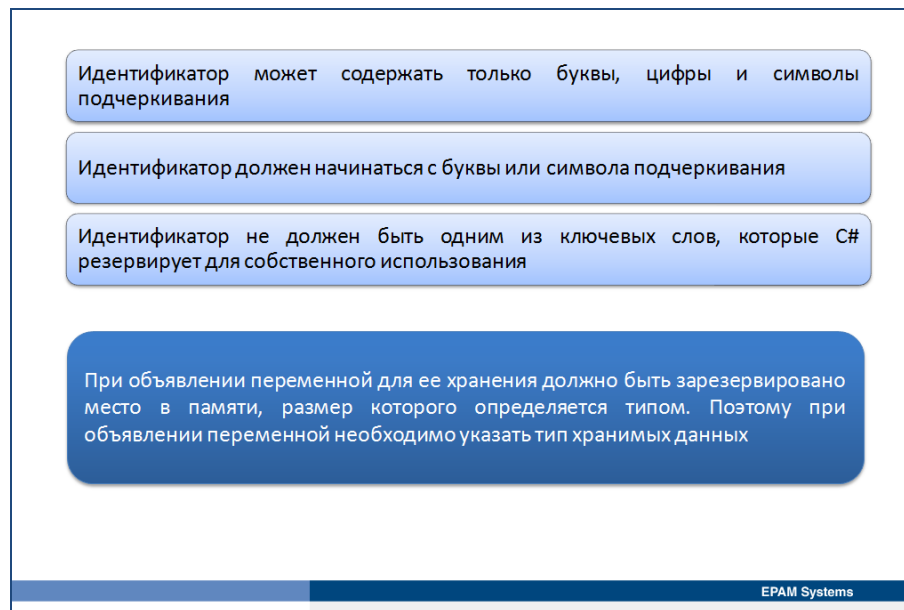
- *Имя (Name)*. Уникальный идентификатор, который ссылается на переменную в коде.
- *Адрес (Address)*. Ячейка памяти переменной.
- *Тип данных (Data type)*. Тип данных, которые может хранить переменная (определяет размер и представление данных в памяти).
- *Значение (Value)*. Значение по адресу переменной.
- *Область видимости (Scope)*. Определенные участки кода, которые могут получать доступ и использовать переменную.
- *Время жизни (Lifetime)*. Период времени, который переменная является действительной и доступной для использования.

Переменные можно использовать во многих случаях, в том числе:

- Как счетчик для циклов.

- В качестве временного хранилища для значения свойства.
- В качестве контейнера для хранения значения, возвращаемого из функции.

Правила именования переменных.



Прежде чем использовать переменную, ее нужно объявить, указав ее имя и характеристики. Имя переменной называется идентификатором. Существуют правила, касающиеся идентификаторов, которые можно использовать:

- Идентификатор может содержать только буквы, цифры и символы подчеркивания.
- Идентификатор должен начинаться с буквы или символа подчеркивания.
- Идентификатор не должен быть одним из ключевых слов, которые C# резервирует для собственного использования.

C# при именовании учитывает регистр, а, следовательно, идентификаторы `MyData` и `MYDATA`, определяют различные переменные, хотя это и не является хорошим стилем программирования.

Для именования переменных следует использовать осмысленные имена, поскольку это делает код более простым для понимания. При этом необходимо придерживаться соглашения об именовании.

При объявлении переменной для ее хранения должно быть зарезервировано место в памяти, размер которого определяется типом. Поэтому при объявлении переменной необходимо указать тип хранимых данных. В одном объявлении можно с помощью запятой указать несколько переменных, при этом все объявленные переменные будут иметь один и тот же тип. Синтаксис объявления переменных показан ниже.

```
DataType variableName;  
// or
```



```
DataType variableName1, variableName2;  
int variableName;
```

После объявления переменной можно присвоить значение для его дальнейшего использования в приложении с помощью оператора присваивания. Значение переменной можно изменить столько раз, сколько это необходимо. Операция присваивания «=» присваивает значение переменной.

```
variableName = value;
```

Значение правой части выражения присваивается переменной в левой части выражения. В следующем примере объявлена целочисленная переменная **price**, которой присваивается значение **10**.

```
int price = 10;
```

Далее существующей целочисленной переменной **price** присваивается число **20**.

```
price = 20;
```

Присвоить значение переменным можно при их объявлении (инициализация). В следующем примере кода показан синтаксис объявления переменной с одновременным присваиванием.

```
DataType variableName = value;
```

Тип выражения при присваивании должен соответствовать типу переменной, иначе программа не будет компилироваться. Например, код в следующем примере не будет компилироваться, потому что нельзя присвоить строковое значение целочисленной переменной.

```
int numberOfEmployees;  
numberOfEmployees = "Hello";
```

При объявлении переменной, пока ей не присвоено значение, она содержит случайное значение. Такое поведение являлось источником ошибок в программах C и C++, которые создавали переменную и могли случайно использовать ее в качестве источника информации, прежде чем присвоить ей значение. C# не позволяет использовать неинициализированную переменную. Необходимо присвоить значение переменной прежде, чем использовать его, в противном случае, программа не будет компилироваться.

При объявлении переменных, можно использовать ключевое слово `var` вместо указания явного типа данных, таких как `int` или `string`. Когда компилятор встречает ключевое слово `var`, он автоматически выводит лежащий в основе тип данных на основе первоначального значения, используемого для инициализации переменной. Поэтому, при объявлении необходимо инициализировать переменную, определив ее следующим образом

```
var price = 20;
```

В примере переменная `price` неявно типизированная переменная. Однако, ключевое слово `var` не означает, что переменной `price` можно позже присвоить значение другого типа. Тип переменной `price` фиксирован, так же, как если бы явно было объявлено, что это целочисленная переменная.

Неявную типизацию можно использовать для любых типов, включая массивы, обобщенные типы и пользовательские специальные типы.

С использованием ключевого слова `var` связаны различные ограничения. Самое первое и важное из них состоит в том, что неявная типизация применима только для локальных переменных в контексте какого-то метода или свойства. Применять ключевое слово `var` для определения возвращаемых значений, параметров или данных полей специального типа не допускается.

```
class ThisWillNeverCompile
{
    // var не может применяться для определения полей!
    private var myInt = 10;
    // var не может применяться для определения
    // возвращаемого значения или типа параметра!
    public var MyMethod(var x, var y) { }
```

Однако, неявно типизированную локальную переменную можно возвращать вызывающему методу, при условии, что возвращаемый тип этого метода совпадает с типом, лежащим в основе определенных с помощью `var` данных.

```
static int GetAnIntValue()
{
    var retVal = 9;
    return retVal;
}
```

Локальным переменным, объявленным с помощью ключевого слова `var`, не допускается присваивать в качестве начального значения `null`. Однако присваивание значения `null` локальной переменной с выведенным после начального присваивания типом вполне допустимо (если переменная относится к ссылочному типу).

```
var myObj = null;
// Ошибка! Присваивание null в качестве начального
// значения не допускается!

var myObj = (int?)null;
myObj = 78;

var myCar = new Car();
// Все в порядке, поскольку Car
// является переменной ссылочного типа!
myCar = null;
```

Значение неявно типизированной локальной переменной может быть присвоено другим переменным, причем как неявно, так и явно типизированным.

```
var myInt2 = 0;  
var anotherInt = myInt;  
string myString2 = "Wake up!";  
var myData = myString;
```

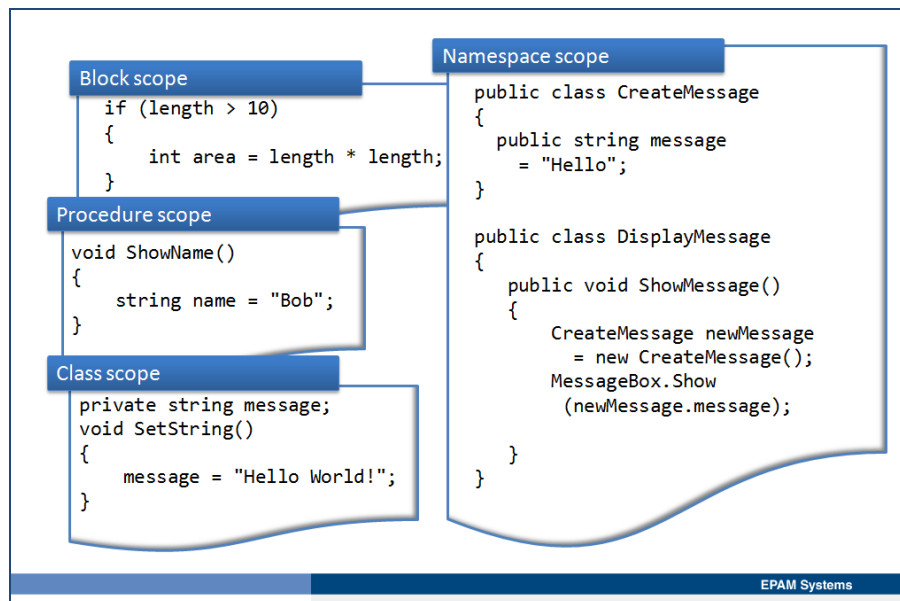
И, наконец, определять неявно типизированную локальную переменную как допускающую значение null с использованием лексемы «?» в C# нельзя.

Следует отметить, что неявная типизация локальных переменных приводит к получению строго типизированных данных. Таким образом, применение ключевого слова `var` в C# не приводит к созданию переменных, которые на протяжении своего существования в программе могут хранить значения разных типов (динамическая типизация²). Выведение типа позволяет языку C# оставаться строго типизированным и оказывает влияние только на объявление переменных во время компиляции. После этого данные трактуются как объявленные с выведенным типом; присваивание такой переменной значения другого типа будет приводить к возникновению ошибок на этапе компиляции.

Использование ключевого слова `var` для объявления локальных переменных просто так особой пользы не приносит, а в действительности может даже вызвать путаницу у тех, кто будет изучать данный код, поскольку лишает возможности быстро определить тип данных и, следовательно, понять, для чего предназначена переменная. Поэтому если точно известно, что переменная должна относиться к типу `int`, лучше сразу объявить ее с указанием этого типа. Неявно типизированные переменные полезны, когда не известно или трудно явно установить тип выражения, которое будет присвоено переменной. Например, в технологии LINQ применяются так называемые выражения запросов (`query expression`), которые позволяют получать динамически создаваемые результирующие наборы на основе формата запроса. В таких выражениях неявная типизация чрезвычайно полезна, так как в некоторых случаях явное указание типа просто не возможно.

² В .NET Framework 4 появилась возможность динамической типизации в C# с использованием нового ключевого слова `dynamic`.

Область видимости переменных



Область видимости переменной определяется частью программы, которая может получить доступ к этой переменной. При попытке сослаться на переменную вне ее области видимости, компилятор будет генерировать ошибку.

Переменные могут иметь один из следующих уровней области видимости:

- Блок (Block).
- Процедура (Procedure).
- Класс (Class).
- Пространство имен (Namespace).

Эти уровни охватывают диапазон от узкой (Block) до широкой (Namespace) областей видимости.

Область видимости Block. Блоком является набор операторов, заключенных в пределах инициирования и определения операторов объявления. Объявленная в блоке переменная, может быть использована только внутри этого блока. В следующем примере кода показано, как объявить локальную переменную `area` с областью видимости уровня блока.

```
if (length > 10)
{
    int area = length * length;
}
```

Область видимости Procedure. Переменные, которые объявлены в рамках процедуры не доступны вне пределов этой процедуры. Только процедура, которая содержит объявление переменной может ее использовать. При объявлении переменных в блоке или процедуре переменные известны как локальные. В следующем примере показано объявление локальной переменной с именем `name` области видимости уровня процедуры.

```

void ShowName()
{
    string name = "Bob";
    MessageBox.Show("Hello " + name);
}

```

Область видимости Class. Если необходимо, чтобы время жизни локальной переменной выходило за рамки жизни процедуры, следует объявить переменную на уровне класса. При объявлении переменных в классе или структуре, но вне процедуры, переменные известны как переменные класса. Определить область видимости для переменных класса можно используя модификаторы доступа. В следующем примере показано объявление **private** переменную `message` с областью видимости уровня класса.

```

...
private string message;
void SetString()
{
    message = "Hello World!";
}
void ShowString()
{
    MessageBox.Show(message);
}
...

```

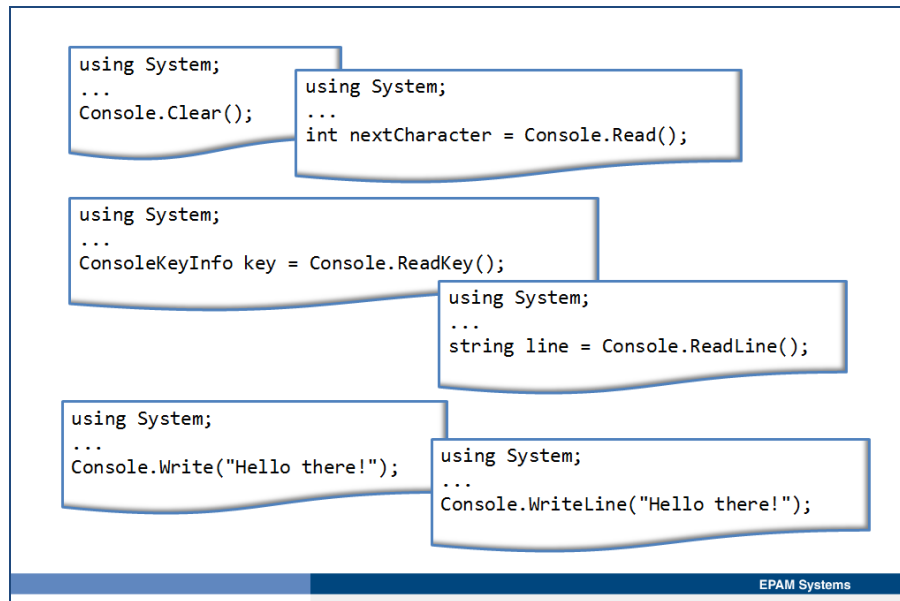
Область видимости Namespace. При объявлении переменных на уровне класса с помощью ключевого слова **public** переменные доступны для всех процедур в пределах всего пространства имен. В следующем примере показано, как объявить **public** переменную `message` в одном классе, но к которой можно получить доступ в другом классе.

```

public class CreateMessage
{
    public string message = "Hello";
}
public class DisplayMessage
{
    public void ShowMessage()
    {
        CreateMessage newMessage = new CreateMessage();
        MessageBox.Show(newMessage.message);
    }
}

```

8. Ввод, вывод в консольном приложении.



Пространство имен System предоставляет класс Console, содержащий методы, позволяющие добавлять основную консольную **I/O** функциональность приложению, а именно, осуществлять ввод и вывод данных. В следующей таблице описаны некоторые из ключевых методов, которые предоставляет класс Console.

Метод	Описание	Пример кода
Clear()	Очищает окно и буфер консоли от данных.	<pre>using System; ... Console.Clear();</pre>
Read()	Читает следующий символ из консоли.	<pre>using System; ... int nextCharacter = Console.Read();</pre>
ReadKey()	Читает следующий символ или клавишу из окна консоли.	<pre>using System; ... ConsoleKeyInfo key = Console.ReadKey();</pre>
ReadLine()	Считывает следующую строку символов из окна консоли.	<pre>using System; ... string line = Console.ReadLine();</pre>
Write()	Пишет текст в окне консоли.	<pre>using System; ... Console.Write("Hello there!");</pre>
WriteLine()	Пишет текст в следующую строку в окне консоли.	<pre>using System; ... Console.WriteLine("Hello there!");</pre>

9. Структурные и ссылочные типы

Как вам должно быть уже известно, классы относятся к ссылочным типам данных. Это означает, что объекты конкретного класса доступны по ссылке, в отличие от значений простых типов, доступных непосредственно. Но иногда прямой доступ к объектам как к значениям простых типов оказывается полезно иметь, например, ради повышения эффективности программы. Ведь каждый доступ к объектам (даже самым мелким) по ссылке связан с дополнительными издержками на расход вычислительных ресурсов и оперативной памяти.

Для разрешения подобных затруднений в C# предусмотрена **структура**, которая подобна классу, но относится к типу значения, а не к ссылочному типу данных. Т.е. структуры отличаются от классов тем, как они сохраняются в памяти и как к ним осуществляется доступ (классы — это ссылочные типы, размещаемые в куче, структуры — типы значений, размещаемые в стеке), а также некоторыми свойствами (например, структуры не поддерживают наследование). Из соображений производительности вы будете использовать структуры для небольших типов данных. Однако в отношении синтаксиса структуры очень похожи на классы.

Главное отличие состоит в том, что при их объявлении используется **ключевое слово `struct`** вместо `class`.

Назначение структур

В связи с изложенным выше возникает резонный вопрос: зачем в C# включена структура, если она обладает более скромными возможностями, чем класс? Ответ на этот вопрос заключается в повышении эффективности и производительности программ. Структуры относятся к типам значений, и поэтому ими можно оперировать непосредственно, а не по ссылке. Следовательно, для работы со структурой вообще не требуется переменная ссылочного типа, а это означает в ряде случаев существенную экономию оперативной памяти.

Более того, работа со структурой не приводит к ухудшению производительности, столь характерному для обращения к объекту класса. Ведь доступ к структуре осуществляется непосредственно, а к объектам — по ссылке, поскольку классы относятся к данным ссылочного типа. Косвенный характер доступа к объектам подразумевает дополнительные издержки вычислительных ресурсов на каждый такой доступ, тогда как обращение к структурам не влечет за собой подобные издержки. И вообще, если нужно просто сохранить группу связанных вместе данных, не требующих наследования и обращения по ссылке, то с точки зрения производительности для них лучше выбрать структуру.

То обстоятельство, что объекты классов доступны по ссылке, объясняет, почему классы называются *ссылочными типами*. Главное отличие типов значений от ссылочных типов заключается в том, что именно содержит переменная каждого из этих типов. Так, переменная типа значения содержит конкретное значение, а ссылочная переменная содержит не сам объект, а лишь ссылку на него.

Для объявления ссылочных типов используются следующие ключевые слова:

- `class`
- `interface`
- `delegate`

В С# также предусмотрены следующие встроенные ссылочные типы:

- `dynamic`
- `object`
- `string`

Класс представляет собой шаблон, по которому определяется форма объекта. В нем указываются данные и код, который будет оперировать этими данными. В С# используется спецификация класса для построения объектов, которые являются экземплярами класса. Следовательно, класс, по существу, представляет собой ряд схематических описаний способа построения объекта. При этом очень важно подчеркнуть, что класс является логической абстракцией. Физическое представление класса появится в оперативной памяти лишь после того, как будет создан объект этого класса.

Классы и структуры — это, по сути, шаблоны, по которым можно создавать объекты. Каждый объект содержит данные и методы, манипулирующие этими данными.

В С# предусмотрен специальный **класс object**, который неявно считается базовым классом для всех остальных классов и типов, включая и типы значений. Иными словами, все остальные типы являются производными от `object`. Это, в частности, означает, что переменная ссылочного типа `object` может ссылаться на объект любого другого типа. Кроме того, переменная типа `object` может ссылаться на любой массив, поскольку в С# массивы реализуются как объекты. Формально имя `object` считается в С# еще одним обозначением класса `System.Object`, входящего в библиотеку классов для среды .NET Framework.

Практическое значение этого в том, что помимо методов и свойств, которые вы определяете, также появляется доступ к множеству общедоступных и защищенных методов-членов, которые определены в классе `Object`. Эти методы присутствуют во всех определяемых классах.

С точки зрения регулярного программирования строковый **тип данных string** относится к числу самых важных в С#. Этот тип определяет и поддерживает символьные строки. В целом ряде других языков программирования строка представляет собой массив символов. А в С# строки являются объектами. Следовательно, тип `string` относится к числу ссылочных.

Тип **dynamic** указывает, что использование переменной и ссылок на ее члены обходит проверку типа во время компиляции. Такие операции разрешаются во время выполнения. Тип `dynamic` упрощает доступ к API COM, таким как API автоматизации Office, к динамическим API, таким как библиотеки IronPython, и к HTML-модели DOM.

Тип `dynamic` в большинстве случаев ведет себя как тип `object`. В частности, можно преобразовать любое выражение, отличное от `NULL`, в тип `dynamic`. Тип `dynamic` отличается от `object` тем, что операции, которые содержат выражения типа `dynamic`, не разрешаются или тип проверяется компилятором. Компилятор объединяет сведения об операции, которые впоследствии будут использоваться для оценки этой операции во время выполнения. В рамках этого процесса переменные типа `dynamic` компилируются в переменные типа `object`. Таким образом, тип `dynamic` существует только во время компиляции, но не во время выполнения.

Интерфейс (interface) представляет собой не более чем просто именованный набор абстрактных членов. Абстрактные методы являются чистым протоколом, поскольку не имеют никакой стандартной реализации. Конкретные члены, определяемые интерфейсом, зависят от того, какое поведение моделируется с его помощью. Это действительно так. Интерфейс выражает поведение, которое данный класс или структура может избрать для поддержки. Более того, каждый класс (или структура) может поддерживать столько интерфейсов, сколько необходимо, и, следовательно, тем самым поддерживать множество поведений.

В интерфейсе ни у одного из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации. В нем указывается только, что именно следует делать, но не как это делать. Как только интерфейс будет определен, он может быть реализован в любом количестве классов. Кроме того, в одном классе может быть реализовано любое количество интерфейсов.

Для реализации интерфейса в классе должны быть предоставлены тела (т.е. конкретные реализации) методов, описанных в этом интерфейсе. Каждому классу предоставляется полная свобода для определения деталей своей собственной реализации интерфейса. Следовательно, один и тот же интерфейс может быть реализован в двух классах по-разному. Тем не менее в каждом из них должен поддерживаться один и тот же набор методов данного интерфейса. А в том коде, где известен такой интерфейс, могут использоваться объекты любого из этих двух классов, поскольку интерфейс для всех этих объектов остается одинаковым. Благодаря поддержке интерфейсов в C# может быть в полной мере реализован главный принцип полиморфизма: *один интерфейс — множество методов*.

Делегат представляет собой объект, который может ссылаться на метод. Следовательно, когда создается делегат, то в итоге получается объект, содержащий ссылку на метод. Более того, метод можно вызывать по этой ссылке. Иными словами, делегат позволяет вызывать метод, на который он ссылается.

По сути, делегат — это безопасный в отношении типов объект, указывающий на другой метод (или, возможно, список методов) приложения, который может быть вызван позднее.

10. Преобразование типов

Явное преобразование (explicit conversion) или приведение (casting)

Требуется, чтобы был написан код для выполнения преобразования, которое, в противном случае, может привести к потере информации или ошибке

```
DataType variableName1 = (castDataType)variableName2;  
...  
string possibleInt = "1234";  
int count = Convert.ToInt32(possibleInt);  
...  
int number = 1234;  
string numberString = number.ToString();  
...  
int number = 0;  
string numberString = "1234";  
if (int.TryParse(numberString, out number))  
{// Conversion succeeded, number now equals 1234}
```

EPAM Systems

При разработке приложения, может понадобиться преобразовать данные из одного типа в другой. Преобразования необходимы, когда значение одного типа должно быть присоединено переменной другого типа. Например, может понадобиться преобразовать строковое значение «99», прочитанное из текстового файла в целое значение 99 для его хранения в целочисленной переменной. Процесс преобразования значения одного типа данных в другой называется преобразование (conversion) или приведение (casting).

В .NET Framework существует два типа преобразования:

- *Неявное преобразование (implicit conversion).* Неявное преобразование выполняется автоматически общезыковой средой выполнения (CLR) согласно операциям, которые гарантированно завершатся успешно без потери информации.
- *Явное преобразование (explicit conversion) или приведение (casting).* Приведение требует, чтобы был написан код для выполнения преобразования, которое, в противном случае, может привести к потере информации или ошибке.

Явное преобразование уменьшает возможность ошибки в коде и делает его более эффективным. В C# запрещены неявные преобразования, которые приводят к потере точности. Однако следует помнить, что и некоторые явные преобразования могут привести к непредсказуемым результатам.

Неявное преобразование. Неявное преобразование происходит при автоматическом преобразовании значения из одного типа данных в другой. Преобразование не требует никакого специального синтаксиса в исходном коде. C# позволяет только безопасные неявные преобразования, такие, как расширение числа. В следующем примере показано неявное преобразование из целочисленной переменной в тип long.

```
int a = 4;
long b;
b = a; // Implicit conversion of int to long
```

Такое преобразование всегда успешно и никогда не приведет к потере информации. Однако, обратное преобразование не верно, нельзя неявно преобразовать тип long в тип int, потому что это преобразование рискует потерей информации (значение типа long может выходить за пределы диапазона, который поддерживает тип int). В следующей таблице приведены неявные преобразования типов, которые поддерживаются в C#.

Из	В
sbyte	short, int, long, float, double, decimal;
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Явные преобразования. В C# можно использовать операцию приведения для выполнения явного преобразования. Приведение типов указывает на тип преобразования в круглых скобках.

```
DataType variableName1 = (castDataType)variableName2;
```

Таким способом можно выполнять только значимые преобразования, такие как преобразование long к int. Нельзя использовать приведение, если формат данных физически изменяется, например, если нужно конвертировать строку в целое число. Для выполнения таких преобразований, используются методы класса System.Convert. Одно из преимуществ подхода с применением класса System.Convert связано с тем, что он позволяет выполнять преобразования между типами данных нейтральным к языку образом, следовательно, все языки, ориентированные на CLR могут использовать этот класс. Этот класс можно использовать для облегчения преобразований, поскольку Microsoft IntelliSense помогает найти метод преобразования, который нужен. Класс System.Convert предоставляет методы, которые могут преобразовать данные базового типа в другой

базовый тип данных. Эти методы имеют имена, такие как `ToDouble`, `ToInt32`, `ToString` и так далее. В следующем примере тип `string` преобразуется к типу `int`.

```
string possibleInt = "1234";  
int count = Convert.ToInt32(possibleInt);
```

В дополнение к методу `Convert.ToString`, типы реализуют собственный метод `ToString`.

```
int number = 1234;  
string numberString = number.ToString();
```

Некоторые из встроенных типов данных в C# обеспечивают метод `TryParse`, который позволяет определить, является ли преобразование удачным, перед его выполнением.

```
int number = 0;  
string numberString = "1234";  
if (int.TryParse(numberString, out number))  
{  
    // Conversion succeeded, number now equals 1234  
}  
else  
{  
    // Conversion failed, number now equals 0  
}
```

11. Операторы

Арифметические операторы

Арифметические операторы C#

Оператор	Действие
----------	----------

+	Сложение
-	Вычитание, унарный минус
*	Умножение
/	Деление
%	Деление по модулю
--	Декремент
++	Инкремент

Операторы +, -, * и / действуют так, как предполагает их обозначение. Их можно применять к любому встроенному числовому типу данных.

Действие арифметических операторов не требует особых пояснений, за исключением следующих особых случаев. Прежде всего, не следует забывать, что когда оператор / применяется к целому числу, то любой остаток от деления отбрасывается; например, результат целочисленного деления 13/3 будет равен 4. Остаток от этого деления можно получить с помощью оператора деления по модулю (%), который иначе называется оператором вычисления остатка. Он дает остаток от целочисленного деления. Например, 13 % 3 равно 1. В C# оператор % можно применять как к целочисленным типам данных, так и к типам с плавающей точкой. Поэтому 13.0 % 3.0 также равно 1. В этом отношении C# отличается от языков C и C++, где операции деления по модулю разрешаются только для целочисленных типов данных. Давайте рассмотрим следующий пример:

```
using System;
```

```

using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            int num1, num2;
            float f1, f2;

            num1 = 13 / 3;
            num2 = 13 % 3;

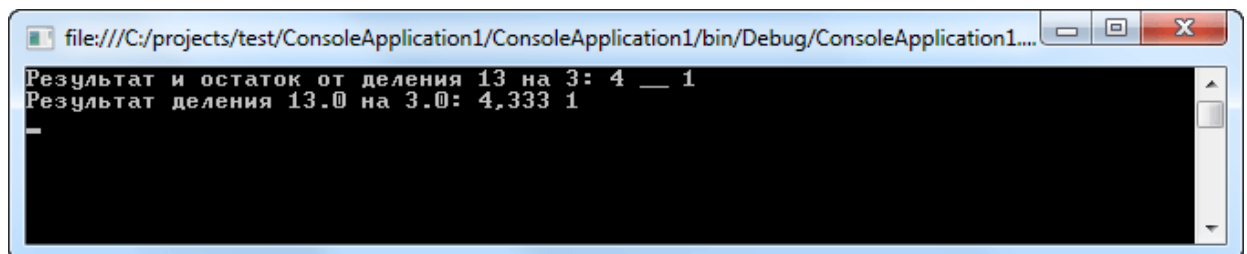
            f1 = 13.0f / 3.0f;
            f2 = 13.0f % 3.0f;

            Console.WriteLine("Результат и остаток от деления 13 на 3: {0} __ {1}", num1, num2);
            Console.WriteLine("Результат деления 13.0 на 3.0: {0:#####} {1}", f1, f2);

            Console.ReadLine();
        }
    }
}

```

Результат работы данной программы:



Операторы инкремента и декремента

Оператор инкремента (++) увеличивает свой операнд на 1, а оператор декремента (--) уменьшает операнд на 1. Следовательно, операторы:

`x++;`

`x--;`

равнозначны операторам:

`x = x + 1;`

`x = x - 1;`

Следует, однако, иметь в виду, что в инкрементной или декрементной форме значение переменной `x` вычисляется только один, а не два раза. В некоторых случаях это позволяет повысить эффективность выполнения программы.

Оба оператора инкремента и декремента можно указывать до операнда (в **префиксной форме**) или же после операнда (в **постфиксной форме**). Давайте разберем разницу записи операции инкремента или декремента на примере:

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

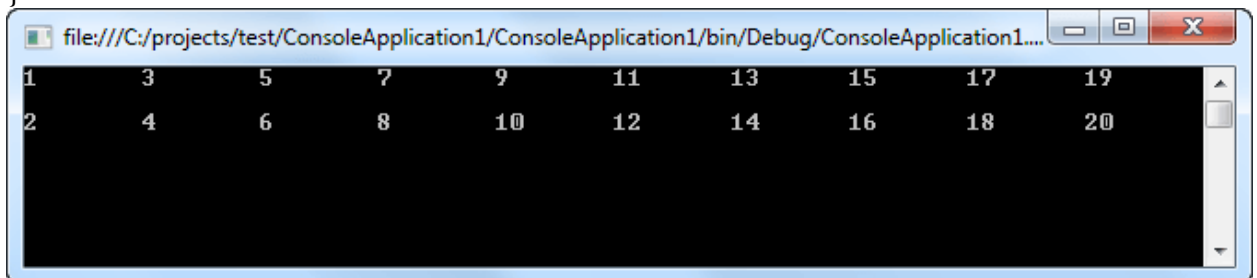
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            short d = 1;

            for (byte i = 0; i < 10; i++)
                Console.Write(i + d++ + "\t");

            Console.WriteLine();
            d = 1;

            for (byte i = 0; i < 10; i++)
                Console.Write(i + ++d + "\t");

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



Т.е. операция инкремента в префиксной форме происходит раньше, нежели в постфиксной форме, в результате чего числа из второго ряда получаются на единицу больше. Отмечу, что возможность управлять

моментом инкремента или декремента дает немало преимуществ при программировании.

Операторы отношений

В обозначениях **оператор отношения** и **логический оператор** термин *отношения* означает взаимосвязь, которая может существовать между двумя значениями, а термин *логический* — взаимосвязь между логическими значениями "истина" и "ложь". И поскольку операторы отношения дают истинные или ложные результаты, то они нередко применяются вместе с логическими операторами. Именно по этой причине они и рассматриваются совместно.

Ниже перечислены операторы отношения:

Операторы отношения C#

Оператор	Значение
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Логические операторы

К числу логических относятся операторы, приведенные ниже:

Логические операторы C#

Оператор	Значение
&	И
	ИЛИ
^	Исключающее ИЛИ

&&	Укороченное И
	Укороченное ИЛИ
!	НЕ

Результатом выполнения оператора отношения или логического оператора является логическое значение типа bool.

В целом, объекты можно сравнивать на равенство или неравенство, используя операторы отношения == и !=. А операторы сравнения <, >, <= или >= могут применяться только к тем типам данных, которые поддерживают отношение порядка. Следовательно, операторы отношения можно применять ко всем числовым типам данных. Но значения типа bool могут сравниваться только на равенство или неравенство, поскольку истинные (true) и ложные (false) значения не упорядочиваются. Например, сравнение true > false в C# не имеет смысла.

Рассмотрим пример программы, демонстрирующий применение операторов отношения и логических операторов:

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            short d = 10, f = 12;
            bool var1 = true, var2 = false;

            if (d < f)
                Console.WriteLine("d < f");
            if (d <= f)
                Console.WriteLine("d <= f");
            if (d != f)
                Console.WriteLine("d != f");

            // Следующее условие не выполнится
            if (d > f)
                Console.WriteLine("d > f");

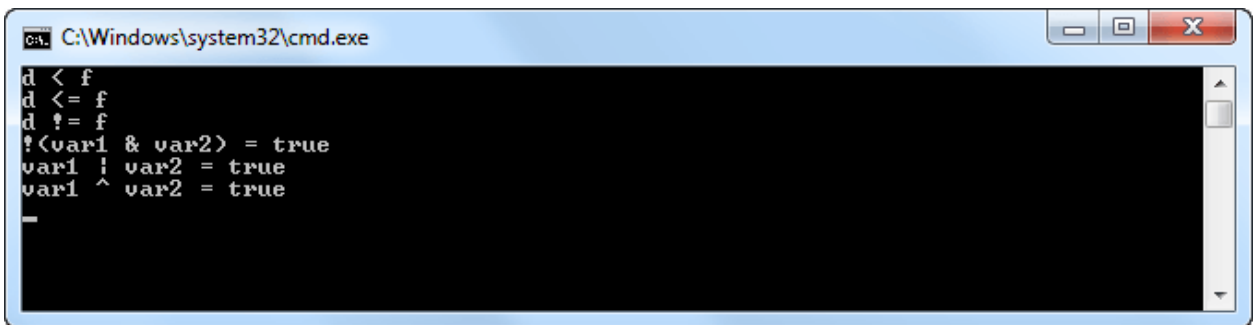
            // Сравнением переменные var1 и var2
            if (var1 & var2)
                Console.WriteLine("Данный текст не выведется");
        }
    }
}
```

```

        if (!(var1 & var2))
            Console.WriteLine("!(var1 & var2) = true");
        if (var1 | var2)
            Console.WriteLine("var1 | var2 = true");
        if (var1 ^ var2)
            Console.WriteLine("var1 ^ var2 = true");

        Console.ReadLine();
    }
}

```



Логические операторы в С# выполняют наиболее распространенные логические операции. Тем не менее существует ряд операций, выполняемых по правилам формальной логики. Эти логические операции могут быть построены с помощью логических операторов, поддерживаемых в С#. Следовательно, в С# предусмотрен такой набор логических операторов, которого достаточно для построения практически любой логической операции, в том числе импликации. **Импликация** — это двоичная операция, результатом которой является ложное значение только в том случае, если левый ее операнд имеет истинное значение, а правый — ложное. (Операция импликации отражает следующий принцип: *истина не может подразумевать ложь*.)

Операция импликации может быть построена на основе комбинации логических операторов ! и |:

$!p \mid q$

Укороченные логические операторы

В С# предусмотрены также специальные, *укороченные*, варианты логических операторов И и ИЛИ, предназначенные для получения более эффективного кода. Поясним это на следующих примерах логических операций. Если первый операнд логической операции И имеет ложное значение (false), то ее результат будет иметь ложное значение независимо от значения второго операнда. Если же первый операнд логической операции ИЛИ имеет истинное значение (true), то ее результат будет иметь истинное

значение независимо от значения второго операнда. Благодаря тому что значение второго операнда в этих операциях вычислять не нужно, *экономится время и повышается эффективность кода.*

Укороченная логическая операция И выполняется с помощью **оператора &&**, а укороченная логическая операция ИЛИ — с помощью **оператора ||**. Этим укороченным логическим операторам соответствуют обычные логические операторы & и |. Единственное отличие укороченного логического оператора от обычного заключается в том, что второй его операнд вычисляется только по мере необходимости.

Укороченные логические операторы иногда оказываются более эффективными, чем их обычные аналоги. Так зачем же нужны обычные логические операторы И и ИЛИ? Дело в том, что в некоторых случаях требуется вычислять оба операнда логической операции И либо ИЛИ из-за возникающих побочных эффектов. Пример:

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            short d = 12, f = 0, i = 0;
            bool b = true;

            // В данном случае используется укороченный оператор
            // и операции сравнения выполнятся в нормальном потоке
            if (f != 0 && (d % f) == 0)
                Console.WriteLine("{0} делится нацело на {1}", d, f);

            // В этом случае так же используется укороченный оператор
            // но при этом возникнет исключительная ситуация
            // т.к. первый оператор сравнения содержит деление на 0
            if ((d % f) == 0 && f != 0)
                Console.WriteLine("{0} делится нацело на {1}", d, f);

            // При использовании целостного оператора в любом
            // случае возникнет исключительная ситуация
            if (f != 0 & (d % f) == 0)
                Console.WriteLine("{0} делится нацело на {1}", d, f);

            /*** Практический пример использования обычных операторов ***/
            // При использовании обычного оператора, в данной конструкции
            // i будет инкрементироваться
            if (b | (++i < 10))
```

```

        Console.WriteLine("i равно {0}", i); // i = 1

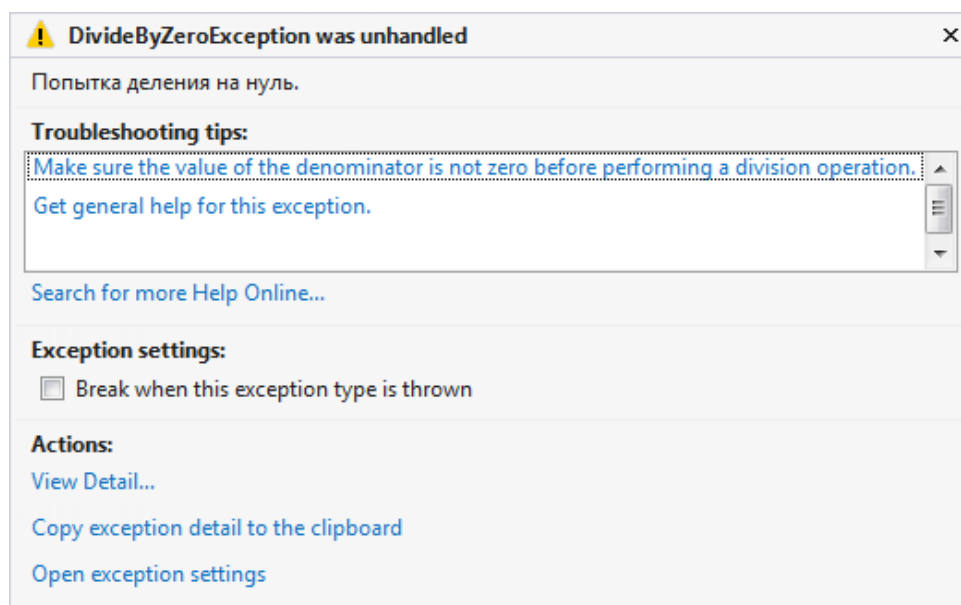
        i = 0;
        // При использовании укороченного оператора
        // значение i останется прежним
        if (b || (++i < 10))
            Console.WriteLine("i равно {0}", i); // i = 0

        Console.ReadLine();
    }

}
}

```

Стоит отметить, что при возникновении исключительной ситуации во время отладки кода, Visual Studio 2010 выводит сообщение следующего характера:



Битовые операторы

В C# предусмотрен ряд **поразрядных операторов**, расширяющих круг задач, для решения которых можно применять C#. Поразрядные операторы воздействуют на отдельные двоичные разряды (биты) своих операндов. Они определены только для целочисленных операндов, поэтому их нельзя применять к данным типа bool, float или double.

Эти операторы называются *поразрядными*, поскольку они служат для проверки, установки или сдвига двоичных разрядов, составляющих целое значение. Среди прочего поразрядные операторы применяются для решения самых разных задач программирования на уровне системы, включая, например, анализ информации состояния устройства. Все доступные в C# поразрядные операторы приведены ниже:

Поразрядные операторы C#

Оператор	Значение
----------	----------

&	Поразрядное И
	Поразрядное ИЛИ
^	Поразрядное исключающее ИЛИ
<<	Сдвиг влево
>>	Сдвиг вправо
~	Дополнение до 1 (унарный оператор НЕ)

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ обозначаются следующим образом: `&`, `|`, `^` и `~`. Они выполняют те же функции, что и их логические аналоги. Но в отличие от логических операторов, поразрядные операторы действуют на уровне отдельных двоичных разрядов.

С точки зрения наиболее распространенного применения поразрядную операцию И можно рассматривать как способ подавления отдельных двоичных разрядов. Это означает, что если какой-нибудь бит в любом из операндов равен 0, то соответствующий бит результата будет сброшен в 0. Поразрядный *оператор ИЛИ* может быть использован для установки отдельных двоичных разрядов. Если в 1 установлен какой-нибудь бит в любом из операндов этого оператора, то в 1 будет установлен и соответствующий бит в другом операнде. Поразрядный *оператор исключающее ИЛИ* устанавливает двоичный разряд операнда в том и только в том случае, если двоичные разряды сравниваемых операндов оказываются разными, как в приведенном ниже примере. Для понимания вышесказанного, разберите следующий пример:

1101 0011	1101 0011	1101 0011	
1001 1001	1001 1001	1001 1001	1101 0011
&		^	~
1001 0001	1101 1011	0100 1010	0010 1100

Давайте теперь рассмотрим пример программы, использующей поразрядные операторы:

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            chet(16);
            provChet(8);
            nechet(16);

            Console.ReadLine();
        }

        // Метод, преобразующий все нечетные числа в четные
        // в диапазоне [0, x] с помощью
        // поразрядного оператора &
        static void chet(int x)
        {
            int result;
            Console.WriteLine("Преобразованный диапазон чисел от 0 до {0}:\n",x);
            for (int i = 0; i <= x; i++)
            {
                // Сбрасываем младший разряд числа, чтобы
                // получить четное число
                result = i & 0xFFFE;
                Console.Write("{0}\t",result);
            }
        }

        // Метод, проверяющий является ли число четным
        static void provChet(int x)
        {
            Console.WriteLine("\n\nПроверка четности чисел в диапазоне от 1 до {0}\n",x);
            for (int i = 1; i <= x; i++)
            {
                if ((i & 1) == 0)
                    Console.WriteLine("Число {0} - является четным",i);
                else
                    Console.WriteLine("Число {0} - является нечетным",i);
            }
        }

        // Метод, преобразующий четные числа в нечетные
```

```

// с помощью поразрядного оператора |
static void nechet(int x)
{
    int result;
    Console.WriteLine("\nПреобразованный диапазон чисел от 0 до {0}:\n",x);
    for (int i = 0; i <= x; i++)
    {
        result = i | 1;
        Console.Write("{0}\t",result);
    }
}
}
}

```

```

Преобразованный диапазон чисел от 0 до 16:
0      0      2      2      4      4      6      6      8      8
10     10     12     12     14     14     16     -     -     -
Проверка четности чисел в диапазоне от 1 до 8
Число 1 - является нечетным
Число 2 - является четным
Число 3 - является нечетным
Число 4 - является четным
Число 5 - является нечетным
Число 6 - является четным
Число 7 - является нечетным
Число 8 - является четным
Преобразованный диапазон чисел от 0 до 16:
1      1      3      3      5      5      7      7      9      9
11     11     13     13     15     15     17     -     -     -

```

Операторы сдвига

В С# имеется возможность сдвигать двоичные разряды, составляющие целое значение, влево или вправо на заданную величину. Ниже приведена общая форма для этих операторов:

значение << число_битов

значение >> число_битов

где *число_битов* — это число двоичных разрядов, на которое сдвигается указанное значение.

При сдвиге влево все двоичные разряды в указываемом значении сдвигаются на одну позицию влево, а младший разряд сбрасывается в нуль. При сдвиге вправо все двоичные разряды в указываемом значении сдвигаются на одну позицию вправо. Если вправо сдвигается целое значение

без знака, то старший разряд сбрасывается в нуль. А если вправо сдвигается целое значение со знаком, то разряд знака сохраняется. Напомним, что для представления отрицательных чисел старший разряд целого числа устанавливается в 1. Так, если сдвигаемое значение является отрицательным, то при каждом сдвиге вправо старший разряд числа устанавливается в 1. А если сдвигаемое значение является положительным, то при каждом сдвиге вправо старший разряд числа сбрасывается в нуль.

При сдвиге влево и вправо крайние двоичные разряды теряются. Восстановить потерянные при сдвиге двоичные разряды *нельзя*, поскольку сдвиг в данном случае не является циклическим. Рассмотрим пример:

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

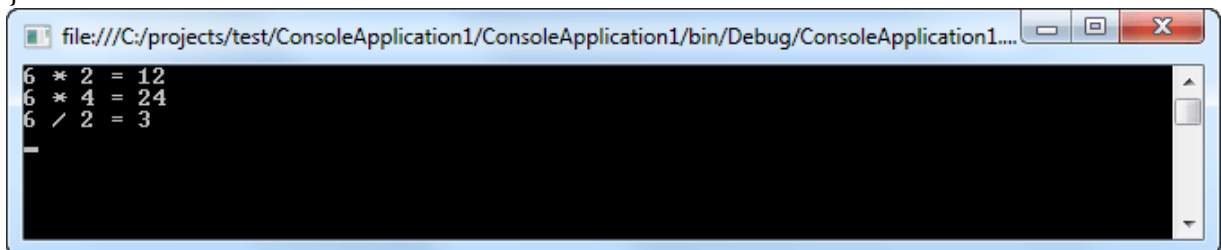
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            byte n = 6, result;

            // Умножить на 2
            result = (byte)(n << 1);
            Console.WriteLine("{0} * 2 = {1}",n,result);

            // Умножить на 4
            result = (byte)(n << 2);
            Console.WriteLine("{0} * 4 = {1}",n,result);

            // Разделить на 2
            result = (byte)(n >> 1);
            Console.WriteLine("{0} / 2 = {1}",n,result);

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



Оператор присваивания

Оператор присваивания обозначается одиночным знаком равенства (=). В С# оператор присваивания действует таким же образом, как и в других языках программирования. Ниже приведена его общая форма:

имя_переменной = выражение

Здесь *имя_переменной* должно быть совместимо с типом выражения. У оператора присваивания имеется одна интересная особенность, о которой вам будет полезно знать: он позволяет создавать цепочку операций присваивания. Рассмотрим следующий фрагмент кода:

```
int x, y, z;
```

```
x = y = z = 10; // присвоить значение 10 переменным x, y и z
```

В приведенном выше фрагменте кода одно и то же значение 10 задается для переменных x, y и z с помощью единственного оператора присваивания. Это значение присваивается сначала переменной z, затем переменной y и, наконец, переменной x. Такой способ присваивания "по цепочке" удобен для задания общего значения целой группе переменных.

Составные операторы присваивания

В С# предусмотрены специальные составные операторы присваивания, упрощающие программирование некоторых операций присваивания. Обратимся сначала к простому примеру:

```
x = x + 1;
```

```
// Можно переписать следующим образом
```

```
x += 1;
```

Пара операторов += указывает компилятору на то, что переменной x должно быть присвоено ее первоначальное значение, увеличенное на 1.

Для многих двоичных операций, т.е. операций, требующих наличия двух операндов, существуют отдельные составные операторы присваивания. Общая форма всех этих операторов имеет следующий вид:

имя переменной op = выражение

где *op* — арифметический или логический оператор, применяемый вместе с оператором присваивания. Ниже перечислены составные операторы присваивания для арифметических и логических операций:

Составные операторы присваивания C#

Оператор Аналог (выражение из вышеуказанного примера)

`+=` `x = x + 1;`

`-=` `x = x - 1;`

`*=` `x = x*1;`

`/=` `x = x/1;`

`%=` `x = x%1;`

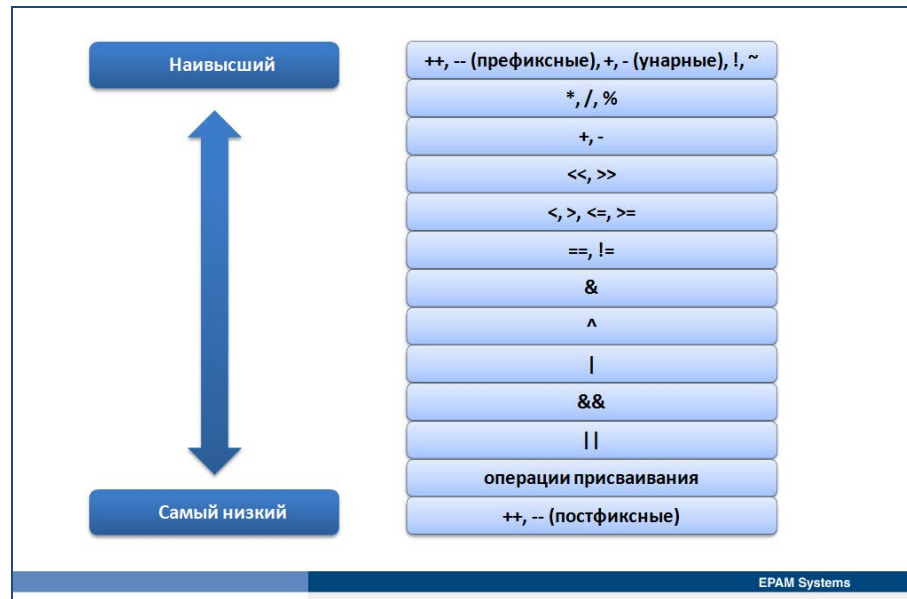
`|=` `x = x | 1;`

`^=` `x = x^1;`

Составные операторы присваивания записываются более кратко, чем их несоставные эквиваленты. Поэтому их иногда еще называют *укороченными операторами присваивания*.

У составных операторов присваивания имеются два главных преимущества. Во-первых, они более компактны, чем их "несокращенные" эквиваленты. И во-вторых, они дают более эффективный исполняемый код, поскольку левый операнд этих операторов вычисляется только один раз. Именно по этим причинам составные операторы присваивания чаще всего применяются в программах, профессионально написанных на C#.

Приоритет операторов



Выражение может содержать сложный ряд операций и операндов. Порядок, в котором операции обрабатываются и вычисляются зависит от самих операций. Не всегда существует простой лево-направленный поток выражений.

Операции, которые используются для создания выражения имеют приоритет, определяющий порядок, в котором они вычисляются. Некоторые операции имеют более высокий приоритет, чем остальные, а значит, выполняются в первую очередь. Например, в следующем примере, операция деления выполняется до операции сложения.

$a = b + 1 / 2;$

В таблице приведен приоритет операций от высокого уровня до самого низкого.

Приоритет	Операция
Наивысший	++, -- (префиксные), +, - (унарные), !, ~
	*, /, %
	+, -
	<<, >>
	<, >, <=, >=
	==, !=
	&
	^
	&&
	операции присваивания
	++, -- (постфиксные)
Самый низкий	

Кроме того, операции имеют конкретную ассоциативность, определяющую порядок, в котором они выполняются по отношению к другим операциям, имеющим такой же приоритет. При использовании операций с одинаковым приоритетом ассоциативность операций используется для определения порядка выполнения. Операции либо право-ассоциативны, либо лево-ассоциативны. Лево-ассоциативные операции выполняются слева направо, например, операция деления «/»:

$$a / 5 / b$$

Здесь, **a** делится на 5, а затем результат делится на **b**. Все бинарные операции лево-ассоциативны, кроме операции присваивания, которая право-ассоциативна:

$$a = b = c$$

Здесь значение **c** присваивается **b**, а затем значение **b** присваивается **a** (множественное присваивание). Кроме того, стоит отметить, что для многих операторов, ассоциативность не всегда важна.

$$a + 5 + b$$

В этом примере не существует никакой разницы для результата при обработке выражения слева направо или справа налево. Однако, поскольку операция сложения определяется как лево-ассоциативный, это может иметь значение в более сложных случаях, например, при перегрузке операций.

Для управления порядком выполнения и изменения приоритета в выражениях можно использовать круглые скобки. Любая часть выражения, которая заключена в скобки обрабатывается до частей выражения, которые находятся вне скобок:

$$a = (b + 1) / 2;$$

Здесь $(b + 1)$ – часть выражения, которая обрабатывается в первую очередь, и результат этой операции делится на **2**, чтобы определить значение, присвоенное **a**.

12. Условия

Условный оператор *if*

The diagram is enclosed in a blue border. It contains two main sections: 'Синтаксис' (Syntax) and 'Пример' (Example). The 'Синтаксис' section is a rounded rectangle with a blue header containing the word 'Синтаксис'. It lists two forms of the `if` statement: a single-line form `if ([condition]) [code to execute]` and a block form `if ([condition]) { [code to execute if condition is true] }`. The 'Пример' section is a rectangle with a blue header containing the word 'Пример'. It shows a code snippet: `if (a > 50) { // Add code to execute if a is greater than 50 here. }`. At the bottom right of the diagram, the text 'EPAM Systems' is visible.

```
Синтаксис

if ([condition]) [code to execute]

if ([condition])
{
    [code to execute if condition is true]
}

Пример

if (a > 50)
{
    // Add code to execute if a is greater than 50 here.
}
```

EPAM Systems

Сокращенная форма оператора **if** являются очень полезной, когда нужно выполнить код, основанный на условии. Синтаксис сокращенной формы оператора **if** имеет вид

```
if ([condition]) [code to execute]
```

В случае если значение выражения **([condition])** истинно, то выполняется код **[code to execute]**. Следует отметить, что условие **[condition]** всегда должно быть заключено в круглые скобки. Если код **[code to execute]** состоит из более чем одного оператора, то он помещается в операторные скобки. Это расширяет синтаксис использования оператора `if` следующим образом:

```
if ([condition])
{
    [code to execute if condition is true]
}
```

Такая форма оператора `if` используется часто, даже если выполняется только одна строка кода, поскольку делает код легче читаемым и расширяемым. Пример выполнения кода, когда переменная имеет значение больше 50, имеет следующий вид:

```
if (a > 50)
{
    // Add code to execute if a is greater than 50 here.
}
```

Условный оператор *if else*

The diagram illustrates the syntax and an example of the `if-else` operator. It is divided into two sections: **Синтаксис** (Syntax) and **Пример** (Example).

Синтаксис

```
if ([condition])
{
    [code to execute if condition is true]
}
else
{
    [code to execute if condition is false]
}
```

Пример

```
if (a > 50)
{
    // Add code to execute if a is greater than 50 here.
}
else
{
    // Add code to execute if a is less than or equal to 50 here.
}
```

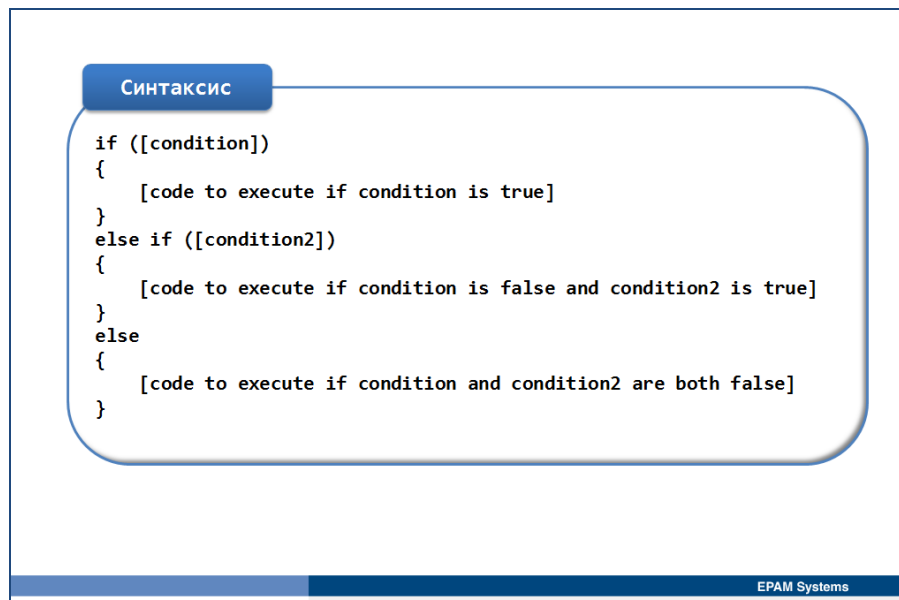
EPAM Systems

Для того, чтобы обеспечить выполнения дополнительного блока кода, когда условие **[condition]** имеет значение **false**, используется ключевое слово **else**, а синтаксис оператора **if** имеет вид

```
if ([condition])
{
    [code to execute if condition is true]
}
else
{
    [code to execute if condition is false]
}
```

Например,

```
if (a > 50)
{
    // Add code to execute if a is greater than 50 here.
}
else
{
    // Add code to execute if a is less than or equal to 50 here.
}
```



Несколько операторов if можно объединить для создания лесенки **if else if...** (multiple-outcome operator) следующим образом:

```
if ([condition])
{
    [code to execute if condition is true]
}
else if ([condition2])
{
    [code to execute if condition is false and condition2
is true]
}
else
{
    [code to execute if condition and condition2 are both
false]
}
```

Здесь важно отметить, что если условие **[condition]** истинно, первый блок кода выполняется, независимо от значения условия **[condition2]**. В этом случае оставшийся код пропускается, и условие **[condition2]** не вычисляется. Это улучшает производительность, поскольку не требует времени для вычисления каждого условия. Ускорить работу кода можно, гарантируя, что наиболее часто выполняющиеся условия будут проверяться в первую очередь. Следующий код показывает пример оператора if, использующего эту конструкцию.

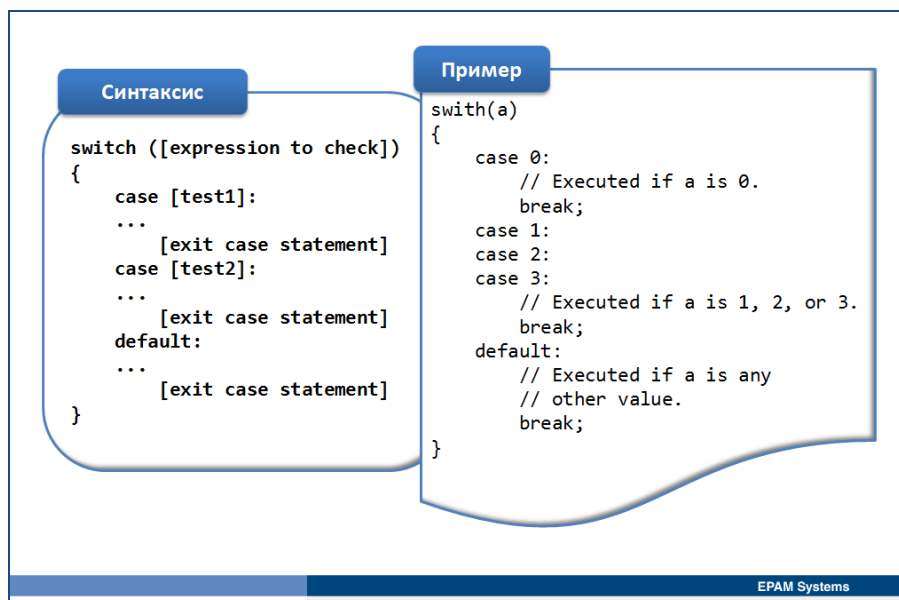
```
if (a > 50)
{
    // Add code to execute if a is greater than 50 here.
}
else if (a > 10)
{
    // Add code to execute if a is greater than 10 and
less than or
```

```

        // equal to 50 here.
    }
else
{
    // Add code to execute if a is less than or equal to
    50 here.
}

```

Условный оператор switch



Позволяет выполнить один из нескольких блоков кода в зависимости от значения переменной или выражения. Эти блоки кода обеспечивают очень простую легкую для чтения конструкцию и предлагают альтернативный подход к использованию оператора `if else if...`:

```

switch ([expression to check])
{
    case [test1]:
        ...
        [exit case statement]
    case [test2]:
        ...
        [exit case statement]
    default:
        ...
        [exit case statement]
}

```


В операторе `switch` необходимо в круглых скобках указать выражение для проверки **[expression to check]** и определить значения **[testX]** для сравнения с переменной. Сравнения выполняются по очереди, таким образом, если значение **[expression to check]** совпадет со значением **[test1]**, выполнится первый блок кода, если со значением **[test2]**, выполнится второй блок кода и так далее. Не существует никаких ограничений на количество сравнений, которые можно включить в оператор `switch`, кроме памяти компьютера. Если совпадений не обнаружено, выполняется блок кода **default**. Блок **default** не является обязательным. Тип значения, который возвращает тестируемое выражение **[expression to check]** должен быть целым числом (включая `char`), строкой или логическим значением, а значения, задаваемые операторами **case** должны соответствовать этому типу. Каждое сравнение (**[testX]**) является одним значением. Проверить нескольких значений можно с помощью нескольких последовательных операторов **case**.

```
switch(a)
{
    case 0:
        // Executed if a is 0.
        break;
    case 1:
    case 2:
    case 3:
        // Executed if a is 1, 2, or 3.
        break;
    default:
        // Executed if a is any other value.
        break;
}
```

Каждый блок кода в операторе `switch` должен заканчиваться оператором, который явно завершает конструкцию (**[exit case statement]**). Если опустить этот оператор, возникнет ошибка компиляции. В качестве таких операторов можно использовать:

- **break;** Оператор **break** завершает выполнение выбранного оператора.
- **goto case [testX];** Оператор **goto** передает управление на выполнение указанного блока кода в операторе **switch**.
- **return;** Оператор приводит к завершению оператора **switch** и содержащихся в нем методов. С помощью оператора можно передать возвращаемое значение.
- **throw;** Оператор **throw** генерирует исключительную ситуацию.

Рекомендуется по мере возможности использовать оператор **break**. Использование **goto case** или **return** может привести к коду, который трудно поддерживать. В следующем примере показано использование оператора **switch** для проверки значения строки.

```
switch(carColor.ToLower())
```

```

{
    case "red":
        // Red car
        break;
    case "blue":
        // Blue car
        break;
    default:
        // Unknown car
        break;
}

```

Оператор ?:

В некоторых простых случаях можно использовать тернарную операцию «?:» как альтернативу использованию полной формы оператора if. Синтаксис использования операции «?:» в этом случае имеет вид

```
Type result = [condition] ? [true expression] : [false expression]
```

Если выражение **[condition]** истинно, выполняется **[true expression]**, в противном случае выполняется **[false expression]**. В следующем примере показан пример использования тернарной операции «?:» для проверки значения строки и возвращения ответа.

```

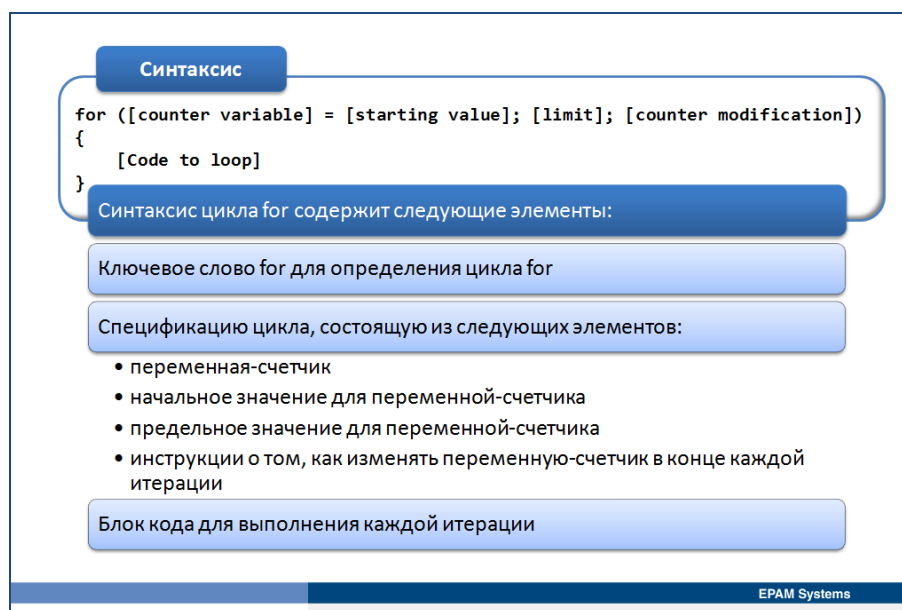
string carColor = "green";
string response = (carColor == "red") ? "You have a red car" : "You do not have a red car";

```

13. Циклы

Цикл for

Цикл for позволяет выполнять код заданное количество раз. Для этого в цикле определяется переменная-счетчик, значение которой изменяется на каждой итерации. Когда переменная-счетчик достигает предельного значения, определяемого вне цикла, цикл завершается. Код в теле цикла for может использовать значение переменной-счетчика. Например, можно использовать цикл for для обработки элементов массива. Можно также использовать вложенные циклы for с различными счетчиками для обработки многомерных массивов или проверки пикселей по указанным координатам.



Цикл for позволяет выполнять блок кода заданное количество раз, отслеживая число выполняемых итераций с помощью переменной-счетчика. Синтаксис цикла for содержит следующие элементы:

- Ключевое слово for для определения цикла for.
- Спецификацию цикла, состоящую из следующих элементов:
 - а. переменная-счетчик (это может быть переменная, которая уже определена или переменная, которая определяется как часть спецификации цикла);
 - б. начальное значение для переменной-счетчика;
 - с. предельное значение для переменной-счетчика;
 - д. инструкции о том, как изменять переменную-счетчик в конце каждой итерации;
- Блок кода для выполнения каждой итерации.

Цикл for имеет следующий синтаксис.

```
for ([counter variable] = [starting value]; [limit];  
[counter modification])  
{  
    [Code to loop]  
}
```

В таблице описаны цели заполнителей кода, представленного выше.

Заполнитель	Использование
[counter variable]	Идентификатор существующей числовой переменной или определение новой числовой переменной.
[starting value]	Число, присваиваемое переменной-счетчику на первой итерации.
[limit]	Условие, проверяемое в начале каждой итерации. Если условие оценивается как истинное, цикл продолжается. Если оно оценивается как ложное, цикл завершается.
[counter modification]	Операции для выполнения в конце каждой итерации.

Следующий пример кода показывает простой цикл for, выполняющий **10** итераций. Переменная-счетчик создается на первой итерации, устанавливается в 0 и увеличивается в конце каждой итерации на **1**. Когда значение **i** достигает **10**, цикл завершается (при **i** равно **10** цикл не работает).

```
for (int i = 0; i < 10; i++)  
{  
    // Code to loop, which can use i.  
}
```

Следующий код выполняется пять раз со значениями **i** для каждой итерации **0, 2, 4, 6** и **8**.

```
for (int i = 0; i < 10; i = +2)  
{  
    // Code to loop, which can use i.  
}
```

В приведенных примерах, управляющая переменная **i** создается как часть конструкции for. Область видимости этой переменной – тело цикла for. Когда цикл заканчивается, переменная **i** не доступна. Если необходимо изучить значение управляющей переменной за пределами цикла, нужно объявить переменную перед началом цикла и использовать ее, как показано в следующем примере кода.

```
int j;  
for (j = 0; j < 10; j++)  
{  
    // Code to loop, which can use j.  
}  
// j is also available here
```

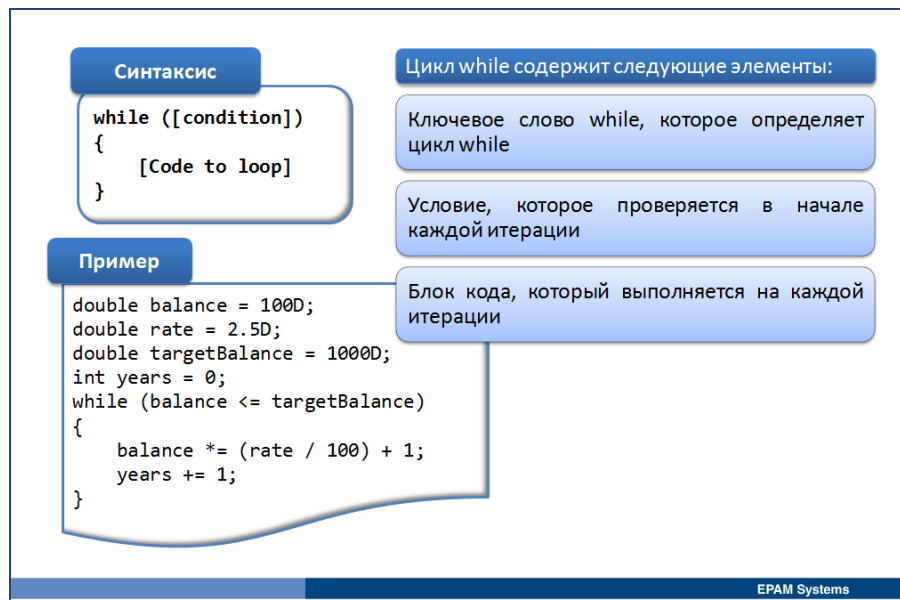
Можно использовать вложенные циклы `for`, каждый из которых определяет свою собственную переменную-счетчик. Эта идиома полезна при обработке многомерных массивов. В следующем примере показано, как использовать два вложенных цикла `for` для обработки в обратном порядке символов в массиве строк.

```
string[] strings = new string[]{"One", "Two", "Three",  
    "Four", "Five"};  
string result = "";  
for (int stringIndex = 0; stringIndex < strings.Length;  
    stringIndex++)  
{  
    for (int charIndex = strings[stringIndex].Length - 1;  
        charIndex >= 0; charIndex--)  
    {  
        result += strings[stringIndex][charIndex];  
    }  
}
```

После инициализации переменных внешний цикл `for` итерируется по значениями счетчика **0, 1, 2, 3, 4** (**strings.Length: 5**). Для каждого значения этого счетчика соответствующая строка в массиве используется для определения начального значения для счетчика внутреннего цикла. Внутренний цикл перебирает строку посимвольно, начиная с конца строки и продвигаясь в обратном порядке (управляющая переменная **charIndex** установлена в длину строки и уменьшается в конце каждой итерации. Цикл останавливается, когда **charIndex** меньше нуля. Тело внутреннего цикла извлекает символ, который индексируется **charIndex** от строки **stringIndex** в массиве. Следует отметить, что можно извлечь отдельные символы из строки, используя массивоподобный доступ с помощью индекса. Когда каждый символ в строке обработан, первая итерация внешнего цикла завершена, и внешний цикл начинает вторую итерацию. Этот процесс продолжается, пока не обработается каждый символ в каждой строке. Значением **result** при завершении работы кода является строка «**enOowTeerhTruoFeviF**».

Цикл while

Цикл `while` позволяет выполнять блок кода ноль или более раз. Цикл `while` не использует переменную-счетчик, хотя можно ее реализовать, определив вне цикла и манипулируя ею на каждой итерации. В начале каждой итерации цикла `while`, проверяется логическое условие. Если это условие истинно, выполняется тело цикла, если условие ложно, цикл завершается. Циклы `while` могут быть очень полезными, если заранее не известно, сколько раз нужно выполнять тело цикла.



Цикл **while** позволяет выполнять блок кода ноль или более раз. В начале каждой итерации цикл **while** вычисляет выражение. Если результат выражения истина, начинается следующая итерация, если оно ложно, цикл завершается. Цикл **while** содержит следующие элементы:

- Ключевое слово **while**, которое определяет цикл **while**.
- Условие, которое проверяется в начале каждой итерации.
- Блок кода, который выполняется на каждой итерации.

Цикл **while** имеет следующий синтаксис.

```
while ([condition])
{
    [Code to loop]
}
```

Условием **[condition]** может быть любое выражение, результатом которого является логическое значение. Каждый раз, когда начинается итерация, в том числе первая, выполняется выражение. Если выражение истинно, выполняется итерация, в противном случае, цикл завершается.

Условие вычисляется для каждой итерации, перед началом итерации. Условие не контролируется во время выполнения итерации, поэтому последняя итерация всегда завершена до завершения цикла.

Следующий пример кода показывает простой расчет, который можно использовать, чтобы определить, сколько лет потребуется банковскому счету для превышения указанного значения с указанной процентной ставкой.

```
double balance = 100D;
double rate = 2.5D;
double targetBalance = 1000D;
int years = 0;
while (balance <= targetBalance)
{
    balance *= (rate / 100) + 1;
```

```

    years += 1;
}

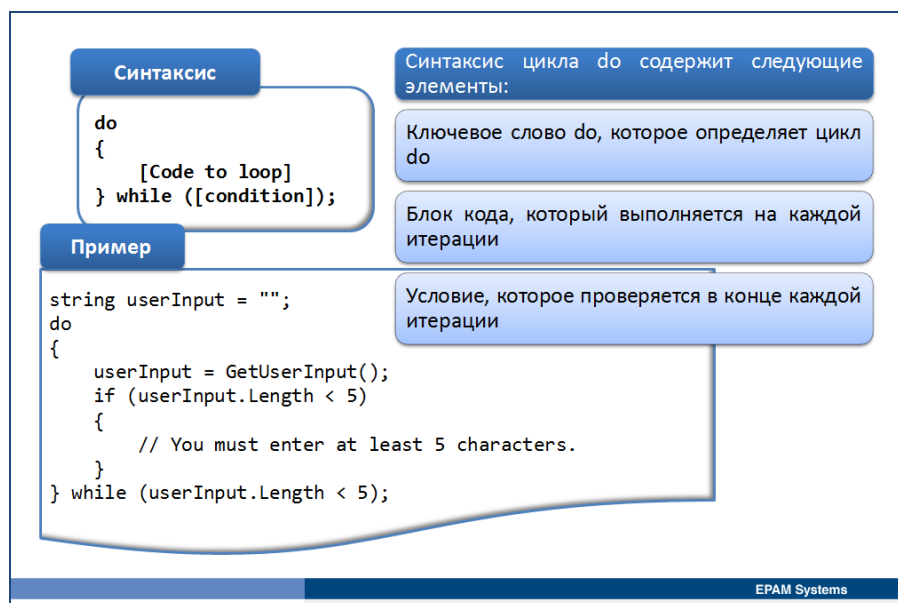
```

Здесь условие **balance <= targetBalance** проверяется перед каждой итерацией. Если значение переменной **balance** больше значения **targetBalance** перед началом цикла, итерации не будут выполняться и значение **years** останется равным значению **0**.

*При использовании цикла **while** код должен изменять булево условие, в противном случае цикл будет выполняться бесконечное число раз.*

*Цикл **do while***

Цикл работает аналогично циклу **while** за исключением одной детали. В цикле **do** условие проверяется в конце итерации, а не в ее начале. Это означает, что тело цикла **do** всегда выполняется по крайней мере один раз, в отличие от цикла **while**, который может не выполниться ни разу. Циклы **do** являются очень полезными, когда заранее не известно, сколько раз должен выполняться код и еще не определено условие. Циклы **do** можно использовать, например, для осуществления пользовательского ввода до тех пор, пока не будет предоставлена достоверная информация.



Цикл **do** позволяет выполнять блок кода один или несколько раз. В конце каждой итерации цикла **do** вычисляется логическое выражение. Если это выражение истинно, начинается другая итерация. Если оно ложно, цикл завершается. Синтаксис цикла **do** содержит следующие элементы:

- Ключевое слово **do**, которое определяет цикл **do**.
- Блок кода, который выполняется на каждой итерации.
- Условие, которое проверяется в конце каждой итерации.

Цикл **do** имеет следующий синтаксис:

```

do
{
    [Code to loop]
}

```

```
    } while ([condition]);
```

Как и в цикле `while` условием **[condition]** может быть любое выражение, результатом которого является логическое значение. Каждый раз при завершении итерации вычисляется выражение. Если выражение истинно, выполняется следующая итерация, в противном случае цикл завершается.

Типичным примером использования цикла `do` является анализ ввода пользователя. Следующий пример показывает код, который требует от пользователя ввести строку длиной по крайней мере пять символов.

```
string userInput = "";
do
{
    userInput = GetUserInput();
    if (userInput.Length < 5)
    {
        // You must enter at least 5 characters.
    }
} while (userInput.Length < 5);
```

Метод **GetUserInput** осуществляет пользовательский ввод и возвращает строку (код метода не приводится).

При использовании цикла `do` код должен изменить булево условие, в противном случае цикл будет выполняться бесконечное число раз.

Цикл `foreach`

Цикл `foreach` служит для циклического обращения к элементам коллекции, представляющей собой группу объектов. В C# определено несколько видов коллекций, каждая из которых является массивом. Ниже приведена общая форма оператора цикла `foreach`:

```
foreach (тип имя_переменной_цикла in коллекция)
    оператор;
```

Здесь *тип имя_переменной_цикла* обозначает тип и имя переменной управления циклом, которая получает значение следующего элемента коллекции на каждом шаге выполнения цикла `foreach`. А коллекция обозначает циклически опрашиваемую коллекцию, которая здесь и далее представляет собой массив. Следовательно, тип переменной цикла должен соответствовать типу элемента массива. Кроме того, тип может обозначаться ключевым словом `var`. В этом случае компилятор определяет тип переменной цикла, исходя из типа элемента массива. Это может оказаться полезным для работы с определенными запросами. Но, как правило, тип указывается явным образом.

Оператор цикла `foreach` действует следующим образом. Когда цикл начинается, первый элемент массива выбирается и присваивается переменной цикла. На каждом последующем шаге итерации выбирается следующий элемент массива, который сохраняется в переменной цикла. Цикл завершается, когда все элементы массива окажутся выбранными.

Цикл `foreach` позволяет проходить по каждому элементу коллекции (объект, представляющий список других объектов). Формально для того, чтобы нечто можно было рассматривать как коллекцию, это нечто должно поддерживать интерфейс `IEnumerable`. Примерами коллекций могут служить массивы `C#`, классы коллекций из пространства имен `System.Collection`, а также пользовательские классы коллекций.

Пример использования цикла `foreach`:

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Объявляем два массива
            int[] myArr = new int[5];
            int[,] myTwoArr = new int[5, 6];
            int sum = 0;

            Random ran = new Random();

            // Инициализируем массивы
            for (int i = 1; i <= 5; i++)
            {
                myArr[i-1] = ran.Next(1, 20);
                for (int j = 1; j <= 6; j++)
                    myTwoArr[i - 1, j - 1] = ran.Next(1, 30);
            }

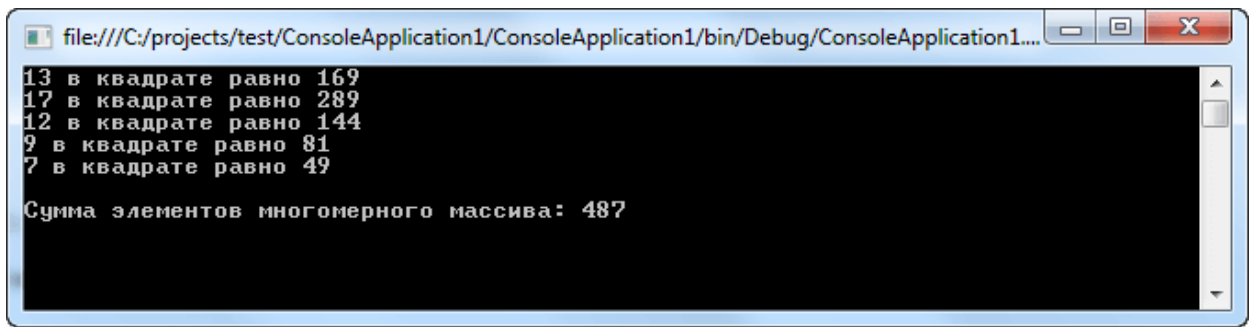
            // Вычисляем квадрат каждого элемента одномерного массива
            foreach (int fVar in myArr)
                Console.WriteLine("{0}          в          квадрате          равно
{1}", fVar, fVar*fVar);

            Console.WriteLine();
            // Вычислим сумму элементов многомерного массива
            foreach (int fTwoVar in myTwoArr)
                sum += fTwoVar;

            Console.WriteLine("Сумма элементов многомерного массива:
{0}", sum);

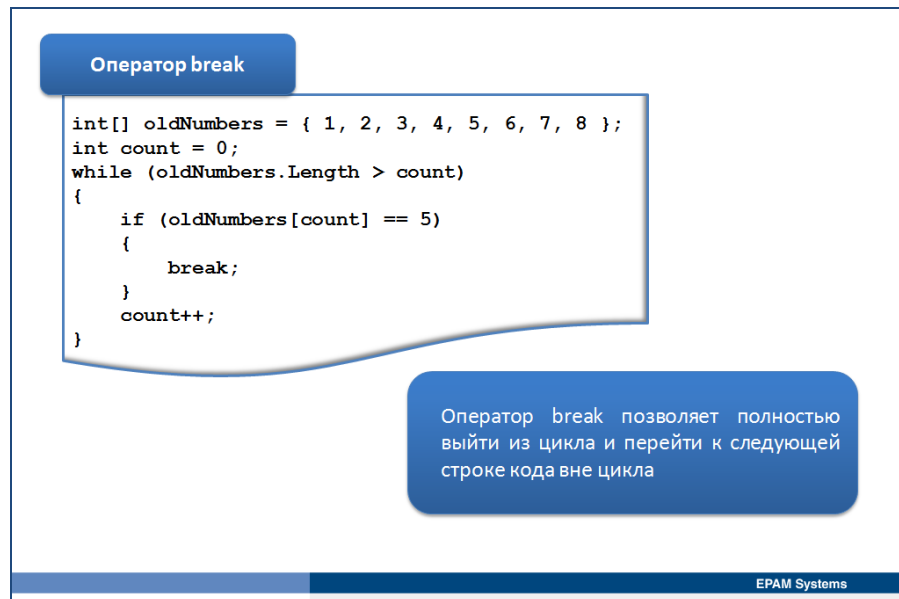
            Console.ReadLine();
        }
    }
}
```

Попробуйте запустить данный пример несколько раз и вы наглядно увидите, что элементы массива изменяются каждый раз (с помощью метода `Random.Next`), и соответственно опрашиваются в цикле `foreach`.



```
file:///C:/projects/test/ConsoleApplication1/ConsoleApplication1/bin/Debug/ConsoleApplication1....
13 в квадрате равно 169
17 в квадрате равно 289
12 в квадрате равно 144
9 в квадрате равно 81
7 в квадрате равно 49
Сумма элементов многомерного массива: 487
```

Инструкция *break*



Оператор break

```
int[] oldNumbers = { 1, 2, 3, 4, 5, 6, 7, 8 };
int count = 0;
while (oldNumbers.Length > count)
{
    if (oldNumbers[count] == 5)
    {
        break;
    }
    count++;
}
```

Оператор `break` позволяет полностью выйти из цикла и перейти к следующей строке кода вне цикла

EPAM Systems

При использовании операторов циклов `while`, `do` и `for` для изменения поведения цикла можно использовать операторы `break` и `continue`. Следует помнить, что эти операторы следует использовать с осторожностью, поскольку они могут привести к коду, который трудно понимать и поддерживать.

Оператор `break`. Оператор `break` позволяет полностью выйти из цикла и перейти к следующей строке кода вне цикла. Оператор `break` особенно полезен, если происходит итерирование по массиву в поисках записи для преждевременного (до выполнения условия окончания) выхода из цикла при ее нахождении. Не следует путать использование оператора `break` в цикле с его использованием в операторе **`switch`**. В следующем примере показан досрочный выход из цикла, когда в массиве найдено значение **5**.

```
int[] oldNumbers = { 1, 2, 3, 4, 5, 6, 7, 8 };
int count = 0;
while (oldNumbers.Length > count)
{
    if (oldNumbers[count] == 5)
    {
        break;
    }
}
```

```
        count++;  
    }
```

При использовании в циклах `while`, `do` и `for` оператор `break` ведет себя одинаково.

Инструкция `continue`

Оператор `continue` похож на оператор `break` за исключением того, что, вместо выхода из цикла полностью, он пропускает оставшийся код текущей итерации, проверяет условие, и, в случае его истинности, начинает следующую итерацию цикла. В следующем примере кода показано, как добавить дополнительную логику в цикл `while`, который не будет выполняться, пока не найдено значение 5.

```
int[] oldNumbers = { 1, 2, 3, 4, 5, 6, 7, 8 };  
int count = 0;  
while (oldNumbers.Length > count)  
{  
    if (oldNumbers[count] == 5)  
    {  
        continue;  
    }  
    // Code that won't be hit when the value 5 is found  
    count++;  
}
```

Оператор `continue` ведет себя одинаково при использовании в циклах `while` и `do`. Существует небольшое различие при его использовании в цикле `for` — в этом цикле оставшийся код текущей итерации пропускается, как и в других циклах, но модификатор переменной счетчика изменяется до проверки условия и начала следующей итерации.

Инструкция `goto`

Имеющийся в С# оператор **`goto`** представляет собой оператор безусловного перехода. Когда в программе встречается оператор `goto`, ее выполнение переходит непосредственно к тому месту, на которое указывает этот оператор. Он уже давно "вышел из употребления" в программировании, поскольку способствует созданию "макаронного" кода. Хотя в некоторых случаях он оказывается удобным и дает определенные преимущества, если используется благоразумно. Главный недостаток оператора `goto` с точки зрения программирования заключается в том, что он вносит в программу беспорядок и делает ее практически неудобочитаемой. Но иногда применение оператора `goto` может, скорее, прояснить, чем запутать ход выполнения программы.

Для выполнения оператора `goto` требуется *метка* — действительный в С# идентификатор с двоеточием. Метка должна находиться в том же методе, где и оператор `goto`, а также в пределах той же самой области действия.

Пример использования оператора `goto`:

```

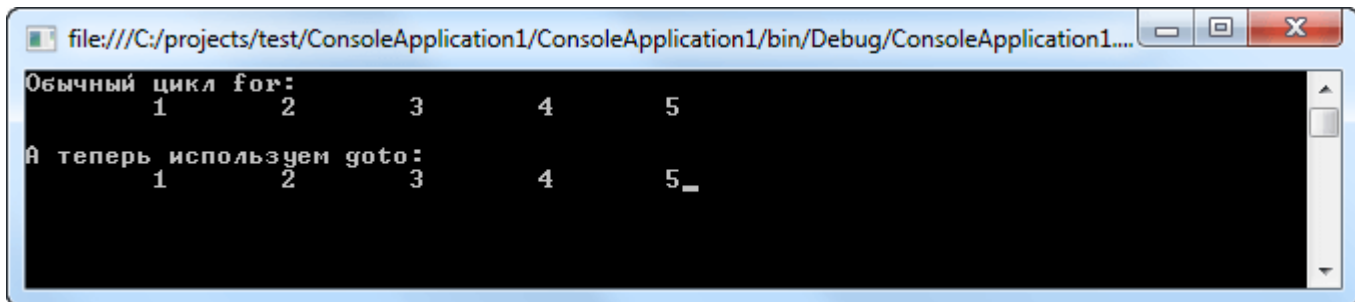
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Обычный цикл for выводящий числа от 1 до 5
            Console.WriteLine("Обычный цикл for:");
            for (int i = 1; i <= 5; i++)
                Console.Write("\t{0}", i);

            // Реализуем то же самое с помощью оператора goto
            Console.WriteLine("\n\nА теперь используем goto:");
            int j = 1;
link1:
            Console.Write("\t{0}", j);
            j++;
            if (j <= 5) goto link1;

            Console.ReadLine();
        }
    }
}

```



```

file:///C:/projects/test/ConsoleApplication1/ConsoleApplication1/bin/Debug/ConsoleApplication1....
Обычный цикл for:
1      2      3      4      5

А теперь используем goto:
1      2      3      4      5_

```

Репутация оператора `goto` такова, что в большинстве случаев его применение категорически осуждается. Вообще говоря, он, конечно, не вписывается в рамки хорошей практики объектно-ориентированного программирования.