Поволжский государственный университет
телекоммуникаций и информатики

Акчурин Э.А.

Программирование на языке C#
в Microsoft Visual Studio .Net или SharpDevelop

Учебное пособие для студентов направления
«Информатика и вычислительная техника»

Самара 2010

Факультет информационных систем и технологий

Кафедра «Информатика и вычислительная техника»

Автор - д.т.н., профессор Акчурин Э.А.

Другие материалы по дисциплине Вы найдете на сайте

www.ivt.psati.ru

Оглавление

1. Введение

1.1. Основные сведения по языку C#

Язык C# появился на свет в июне 2000 в результате работы группы программистов Microsoft, которую возглавляет датчанин Андерс Хейлсберг (Anders Hejlsberg). Этот человек известен как автор одного из первых компилируемых языков программирования для персональных компьютеров IBM - Turbo Pascal. Кроме того, во время работы в корпорации Borland он прославился созданием интегрированной среды Delphi. В 2000 году он получил награду популярного журнала Dr. Dobb's Journal за создание Turbo Pascal, Delphi и C#. Другой известной фигурой в команде разработчиков C# является Эрик Гуннерсон (Eric Gunnerson), автор первого популярного учебника «Введение в C#».

Андерс Хейлсберг, Эрик Гуннерсон - язык C#

В настоящее время над языком C# работает группа программистов: Джеффри Рихтер, Кристиан Нейгел, Билл Ивьен, Джей Глин, Карли Уотсон, Морган Скиннер, Эндрю Троелсен, Трей Нэш.

Символ # в названии языка можно интерпретировать, как две пары плюсов ++;++, намекающие на новый шаг в развитии языка по сравнению с C++.

C# часто называют «C шарпом» (от англ. sharp) из-за схожести символа # с диезом (музыкальный символ).

Язык программирования C# был разработан в качестве эффективного, надежного и простого в использовании средства. В настоящее время C# является частью системы Visual Studio.NET, предназначенной для удобной работы с

платформой .NET Framework и создания нового способа написания надежного программного обеспечения для высокопроизводительных серверов, компактных мобильных устройств и многого другого.

Существует бесплатная версия Visual Studio .NET под названием Visual Studio Express Edition, в состав которой входит большинство компонент полной версии, включая Visual C#. Последние версии созданы в 2008 и 2010 году. Обе являются локализованными, содержат интерфейс и справку на русском языке.

C# — это современный компонентно-ориентированный язык с рядом возможностей, общих для других языков программирования платформы .NET Framework. В C# существует около 80 ключевых слов, большинство из которых известно всем, кто работал с C, C++, Java или Visual Basic. Имеются различия в синтаксисе, но они обычно незначительные.

Благодаря среде редактирования Visual C#, использующей технологию IntelliSense для автоматического выполнения большей части сложных действий, изучение C# не представляет трудностей. Редактор C# автоматически сохраняет код в надлежащем виде, по мере необходимости предлагает методы и свойства и выделяет ошибки при вводе данных.

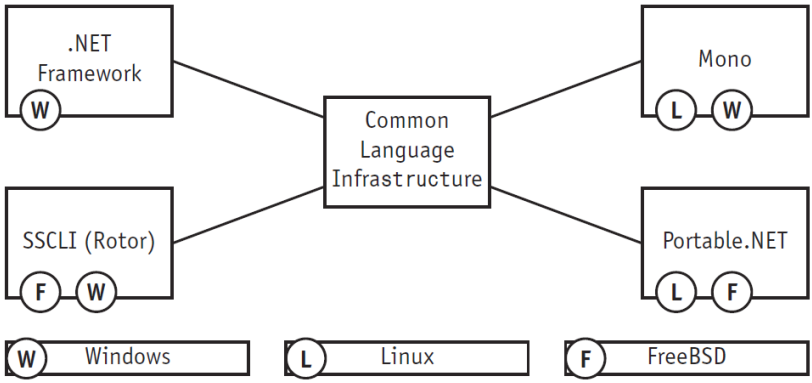
Список литературы

1. Троелсен Э. Язык программирования C# 2008 и платформа .NET 3.5, 4-е изд. : Пер. с англ. - М. : "Вильямс", 2010. 1344 с.
2. Нэш Т. C# 2010. Ускоренный курс для профессионалов. Пер. с англ. - М: “Вильямс»,.2010, 592с.
3. Макки А. Введение в .NET 4.0 и Visual Studio 2010 для профессионалов. Пер. с англ. - М.: "Вильямс", 2010. 412с.
4. Нейгел К. и др. C# 2008 и платформа .NET 3.5 для профессионалов. / Пер. с англ. - М.: "Вильямс", 2009. 1392с.
5. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. Пер. с англ. - М.: «Русская Редакция» ; СПб. : Питер , 2007. 656 стр.
6. Lidin S. Expert .NET 2.0 IL Assembler. Apress; 2006, 530с.
7. Макаров А. и др. CIL и системное программирование в Microsoft.NET: – М. : Интернет-УИТ, 2006. 328 с..
8. Климов Л. C#. Советы программистам. - СПб.: БХВ-Петербург, 2008. 544 с: ил. + CD-ROM.

1.2. Общие сведения о .Net Framework

.Net –одна из возможных реализаций так называемой общей инфраструктуры языков (Common Language Infrastructure, сокращенно CLI), спецификация которой разработана корпорацией Microsoft.

Можно, руководствуясь этой спецификацией, разработать собственную реализацию CLI. В настоящее время ведутся по крайней мере два посвященных этому проекта. Это платформа Mono, создаваемая компанией Ximian, и разрабатываемый в рамках GNU проект Portable.Net. Кроме того, Microsoft распространяет в исходных текстах еще одну свою реализацию CLI, работающую как в Windows, так и под управлением FreeBSD. Эта реализация называется Shared Source CLI (иногда можно услышать другое название – Rotor).



Чтобы понять, как работает .Net, необходимо изучить спецификацию CLI. Это ее составные части:

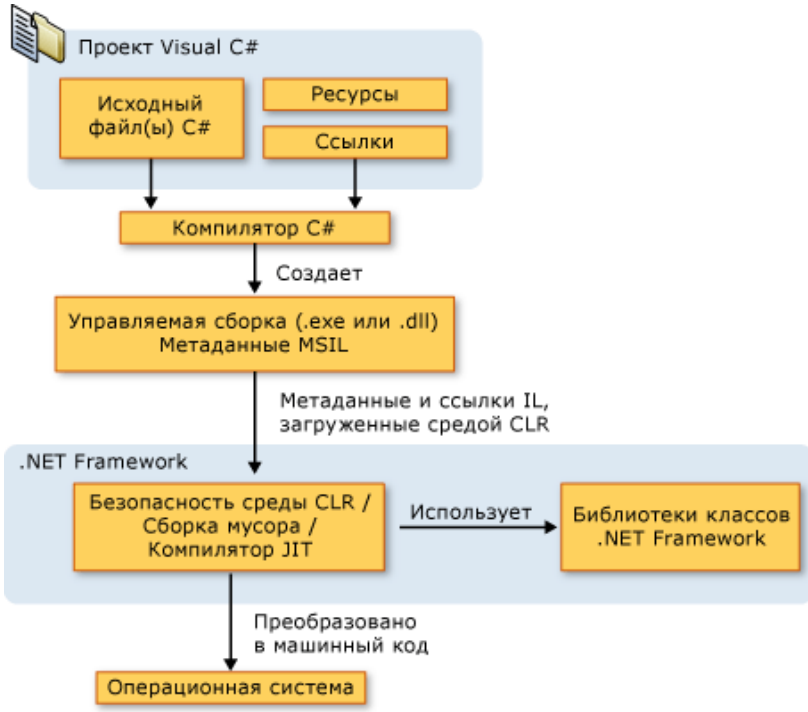
- **Общая система типов** (Common Type System, сокращенно CTS). Охватывает большую часть типов, встречающихся в распространенных языках программирования.
- **Виртуальная система исполнения** (Virtual Execution System, сокращенно VES). Отвечает за загрузку и выполнение программ, написанных для CLI.
- **Система метаданных** (Metadata System). Предназначена для описания типов, хранится в независимом от конкретного языка программирования виде, используется для передачи типовой информации между различными метаинструментами, а также между этими инструментами и VES.
- **Общий промежуточный язык** (Common Intermediate Language, сокращенно CIL) – независимый от платформы объектно-ориентированный байт-код, выступающий в роли целевого языка для любого поддерживающего CLI компилятора. Программа создается на одном из поддерживаемых в .NET языков программирования, компилируется в код CIL, из которого собирается исполняемый файл. Для каждого языка используется свой компилятор.

- **Общая спецификация языков** (Common Language Specification, сокращенно CLS). Это соглашение между разработчиками языков программирования и разработчиками библиотек классов, в котором определено подмножество CTS и набор правил. Если разработчики языка реализуют хотя бы определенное в этом соглашении подмножество CTS и при этом действуют в соответствии с указанными правилами, то пользователь языка получает возможность использовать любую соответствующую спецификации CLS библиотеку.

1.3. C# и платформа .Net Framework

В отличие от традиционного кода на C и C++, код C# не компилируется непосредственно в машинный язык. Компилятор C# преобразует исходный код C# в код на промежуточном языке IL (MIcrosoft Intermediate Language). Файлы IL (или сокращенно IL) называются сборками. Файлы IL создаются всеми языками на основе общеязыковой среды выполнения CLR (CLR – Common Language Runtime). Среда CLR поддерживает языки Visual C#, Visual C++, Visual J# и Visual Basic. Созданные файлы IL в большинстве случаев практически идентичны во многих языках, что упрощает сочетание различных программных компонентов, написанных на разных языках.

На следующей схеме показано преобразование написанного кода C# в исполняемое приложение.



Файлы CIL отображаются в виде стандартных файлов EXE или DLL, однако вместо выполнения непосредственно на платформе Windows, они выполняются средой CLR. При необходимости CLR компилирует программу IL в машинный код. Данный процесс называется JIT-компиляцией (JIT – Just in Time). Затем происходит прямое выполнение этого кода. В исполняемый код включаются ссылки на подпрограммы, содержащиеся в сборках (иначе, динамически подключаемых библиотеках DLL) платформы .NET Framework. Чтобы они работали, на компьютере **должна** быть установлена платформа .NET Framework.

За счет создания промежуточного, независимого от оборудования кода, который не преобразуется в машинный код вплоть до последнего момента, повышается надежность, безопасность и переносимость. Большая часть этого процесса скрыта от программистов: программы C# компилируются, выполняются и распространяются так же, как и другие. Пока на компьютере установлена платформа .NET Framework, программа C# будет выполняться подобно любому другому приложению.

Файлы CIL имеют расширение .exe. Однако фактически это исходники. Если их нужно разворачивать на компьютере без платформы .NET Framework, то тре-

буется формировать настоящий исполняемый файл, используя специальные процедуры.

Возможности платформы .NET Framework упорядочены по пространствам имен, каждое из которых обычно содержит несколько классов. Например, пространство имен System.IO содержит много классов для чтения и записи файлов, а пространство имен System.Text включает классы для обработки строковых данных. Просмотрите справочную документацию по библиотеке классов .NET Framework, чтобы получить представление о различных пространствах имен и их содержимом.

1.4. Средства разработки для .NET Framework

В настоящее время для создания программного обеспечения (ПО) для платформы .NET Framework используются ИСР - интегрированные среды разработки (IDE – Integrated Development Environment), в которых поддерживается технология быстрой разработки.

Для работы с .NET Framework в операционных системах Microsoft доступно несколько ИСР.

Visual Studio .Net. Платная ИСР от Microsoft. Включает набор языков программирования, выбирается желаемый язык. Интерфейс и справка на русском. Visual Studio .Net 4.0 включает базовые средства:

- Visual Basic .Net – язык Visual Basic. Совершенно новая версия языка, по функционалу совпадающая с C#.
- Visual C# .Net – язык C#.
- Visul C++ .Net – язык C++.
- Visual F# .Net – язык F#, язык функционального программирования.
- Visual JScript# .Net – язык JScript#.
- Visual Web Developer – разработка Web приложений.

Visual Studio .Net Express. Бесплатная ИСР от Microsoft. Включает ограниченный набор языков программирования. Для каждого языка ИСР устанавливается автономно. Интерфейс и справка на русском языке. . Visual Studio .Net Express Edition включает:

- Visual Basic .Net – язык Visual Basic.
- Visul C++ .Net – язык C++.
- Visual C# .Net – язык C#.
- Visual Web Developer – разработка Web приложений.

SharpDevelop. Бесплатная ИС от компании SharpDevelop P, в которой выбирается желаемый язык. Включает много языков программирования для выбора.

Интерфейс русский, справка на английском языке. SharpDevelop позволяет программировать на: Boo, C#, C++, F#, Python, Ruby, Visual Basic.

В настоящее время наиболее популярен язык Visual C#.

1.5. Среда CLR

Программы для платформы .NET распространяются в виде так называемых **сборок** (assemblies). Каждая сборка представляет собой совокупность метаданных, описывающих типы, и CIL-кода.

1.5.1. Понятие сборки (assembly)

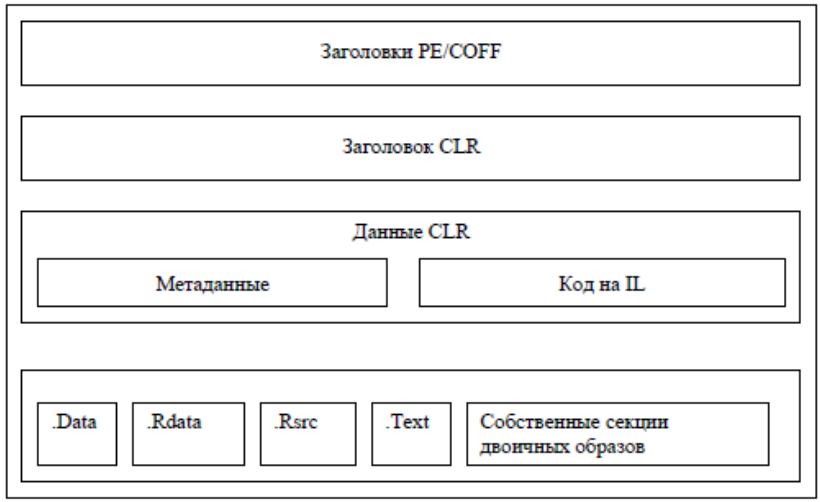
Основной задачей программиста является написание исходного текста программы на одном из языков программирования. Хороший программист, разрабатывая новую программу, не пишет весь код заново. Он старается использовать уже готовые программные коды (библиотеки), написанные как им самим, так и другими разработчиками. Если рассматривать эти библиотеки, как строительные блоки, то программист из них, как из кирпичей, строит здание – новую программу.

В процессе развития технологии программирования было несколько вариантов реализации подхода к повторному использованию кода. На сегодня это динамически подгружаемые библиотеки (DLL). DLL – это неуправляемые PE-файлы (PE – portable executable). Это значит, что компьютер, работающий под управлением Windows, способен загрузить этот файл и выполнить код, содержащийся в нем.

Данный подход используется уже несколько лет и кроме достоинств в нем есть ряд недостатков. В частности для использования DLL подгружается, но не выгружается, когда перестает использоваться. В качестве решения проблем DLL-библиотек в .NET предложен новый подход, в соответствии с которым на замену DLL-библиотекам пришло понятие сборок.

Сборка – это единица повторного использования кода, в которой поддерживается система управления версиями и заложена система управления безопасности программного обеспечения. Сборка подключается только на время исполнения кода. Файл сборки называется управляемым.

Сборка наряду с программным кодом CIL содержит метаданные и данные (ресурсы), необходимые при исполнении сборки для генерации бинарного файла. В общем виде структура сборки:



- Заголовок CLR – содержит информацию, указывающую, что сборка является исполняемым файлом .NET,
- Данные CLR – определяют, как будет выполняться программа. Данные включают метаданные и код программы на CIL. Служебная информация (метаданные) получила название манифест.

1.5.2. JIT-компиляция

Ключевой особенностью выполнения программ в среде .NET является JIT-компиляция. Аббревиатура JIT расшифровывается как Just-In-Time, и термин JIT-компиляция можно перевести как «точно во время», или «на лету». JIT-компиляция заключается в том, что CIL-код, находящийся в запускаемой сборке, тут же компилируется в машинный код, на который затем передается управление.

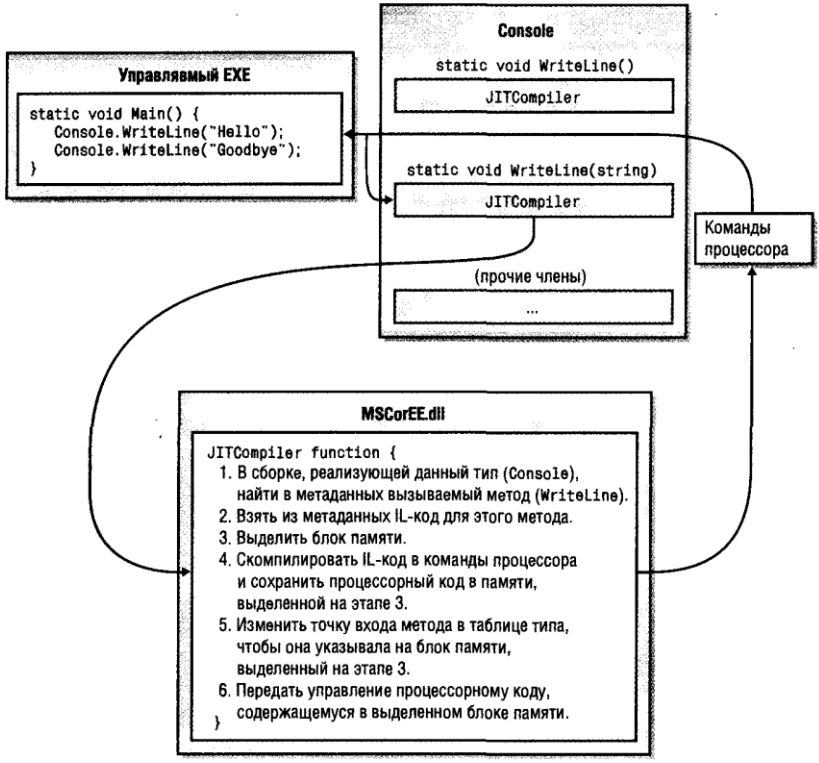
Рассмотрим порядок исполнения на примере простого управляемого EXE файла с кодом на языке C# с одним модулем Main, который предусматривает вывод в консоль последовательно двух фраз.

При исполнении кода в ИСР:

- Компилятор C# транслирует код в CIL. Результат в файле не сохраняется, а помещается в оперативную память по определенному адресу.
- Среда CLR получает этот адрес и анализирует содержащийся там CIL код. В нем CLR находит все типы, на которые ссылается код Main. Модуль Main() ссылается на единственный тип - Console, и CLR выделяет

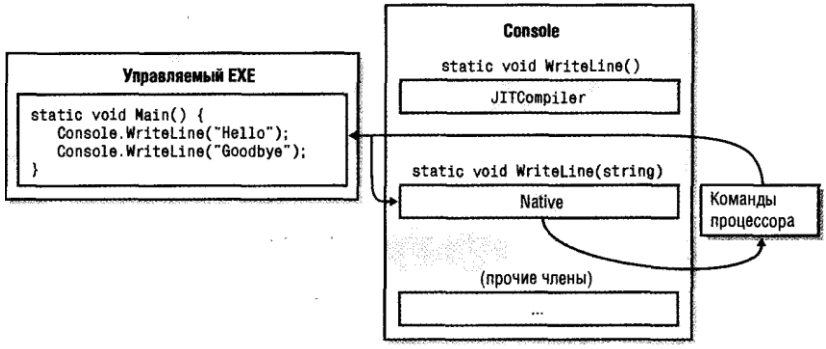
память под него. Эта структура содержит записи для каждого метода, определенного в типе Console. Каждая запись содержит адрес, по которому можно найти реализацию метода. При инициализации этой структуры CLR заносит в каждую запись адрес внутренней, недокументированной функции, содержащейся в самой CLR - JITCompiler.

- Далее последовательно компилируются методы в модуле.



- Когда Main первый раз обращается к WriteLine, вызывается функция JIT-Compiler. Функции JITCompiler известен вызываемый метод и тип, в котором он определен. JITCompiler ищет в метаданных соответствующей сборки IL-код вызываемого метода.
- Затем JITCompiler проверяет и компилирует CIL-код в собственные машинные команды, которые сохраняются в динамически выделенном блоке памяти.

- После этого JITCompiler возвращается к внутренней структуре данных типа и заменяет адрес вызываемого метода адресом блока памяти, содержащего готовые машинные команды.
- В завершение JITCompiler передает управление коду в этом блоке памяти. Этот код — реализация метода WriteLine (той его версии, что принимает параметр String). Из этого метода управление возвращается в Main, который продолжает выполнение в обычном порядке.
- Затем Main обращается к WriteLine вторично. К этому моменту код WriteLine уже проверен и скомпилирован, так что обращение к блоку памяти производится, минуя вызов JITCompiler. Отработав, метод WriteLine возвращает управление Main. Рисунок показывает, как выглядит ситуация при повторном обращении к WriteLine.



Снижение производительности наблюдается только при первом вызове метода. Все последующие обращения выполняются «на полной скорости»: повторная верификация и компиляция не производятся. JITCompiler хранит машинные команды в динамической памяти. Это значит, что скомпилированный код уничтожается по завершении работы приложения. Так что, если потом снова вызвать приложение или если одновременно запустить второй его экземпляр (в другом процессе ОС), JIT-компилятор заново будет компилировать CIL-код в машинные команды.

Для большинства приложений снижение производительности, связанное с работой JIT-компилятора, незначительно. Большинство приложений раз за разом обращается к одним и тем же методам. На производительности это скажется только раз. К тому же скорее всего больше времени занимает выполнение самого метода, а не обращение к нему.

Программы на CIL переводятся в исполняемый бинарный код реального процессора лишь непосредственно перед исполнением метода. Используется динамическая компиляция «на лету». При этом выполняется довольно слож-

ный типовой анализ программы и проверки условий корректности кода. Полученный код метода сохраняется в оперативной памяти и исполняется. Операция повторяется для каждого вызываемого метода. После завершения получается бинарный файл, сохраняемый на диске, который может затем исполняться без повторной компиляции.

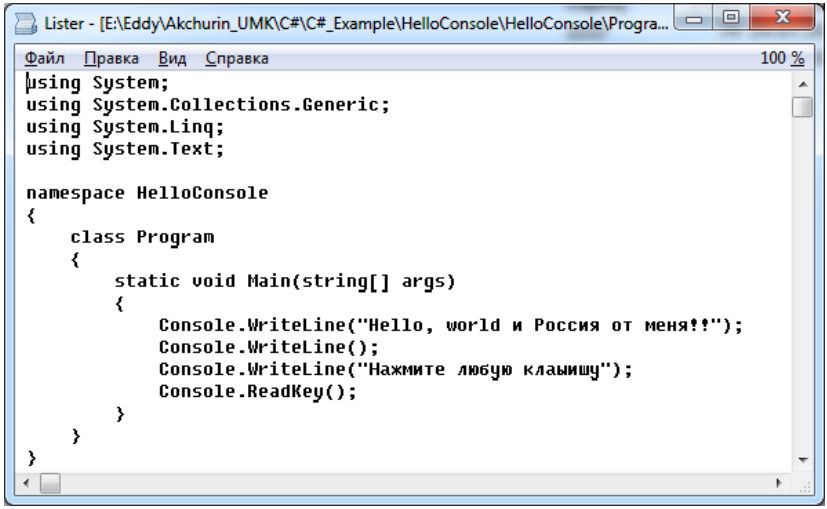
Внимание. Бинарный файл имеет формат PE (управляемый), может исполняться только на машине с Net.Framerwotk.

Таким образом, осуществляется интерпретация кода с компилированием последовательно вызываемых методов. Лучше употреблять вместо компиляции термин - трансляция.

Трансляция осуществляется автоматически. Программист может ничего не знать о командах CIL и JIT компиляторе.

1.5.3. Просмотр метаданных

Для демонстрации примера вида и структуры метаданных используем программу на C# HelloConsole:

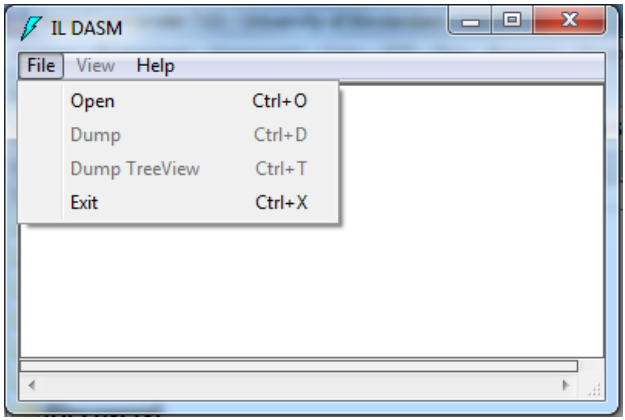


При компиляции программы в CIL создается сборка, сохраняемая на диске как управляемый файл HelloConsole.exe. Из сборки при исполнении генерируется исполняемый бинарный файл, который исполняется на-лету.

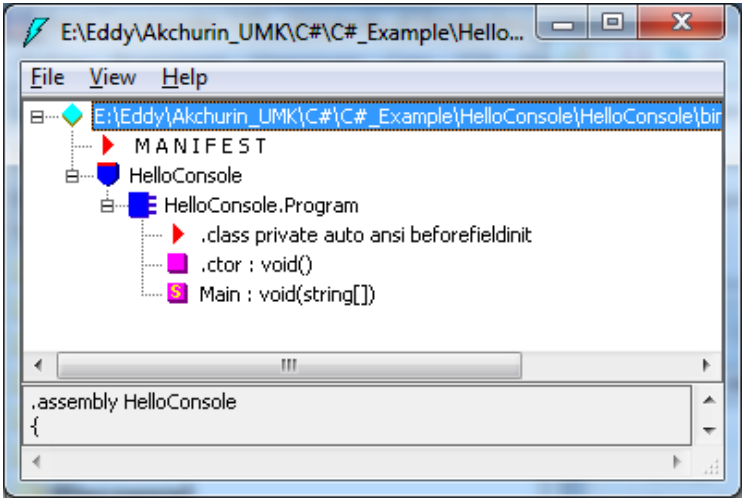
Внимание. Сам бинарный файл на диске не сохраняется.

Для просмотра (при желании) метаданных получившейся сборки необходима утилита **ildasm.exe** (Intermediate Language Disassembler - дизассемблер промежуточного языка). Она из файла сборки HelloConsole.exe. формирует сборку в виде сохраняемого файла. Для запуска утилиты нужна и динамическая библиотека fusion.dll. Если их нет, то их нужно скачать из Интернета.

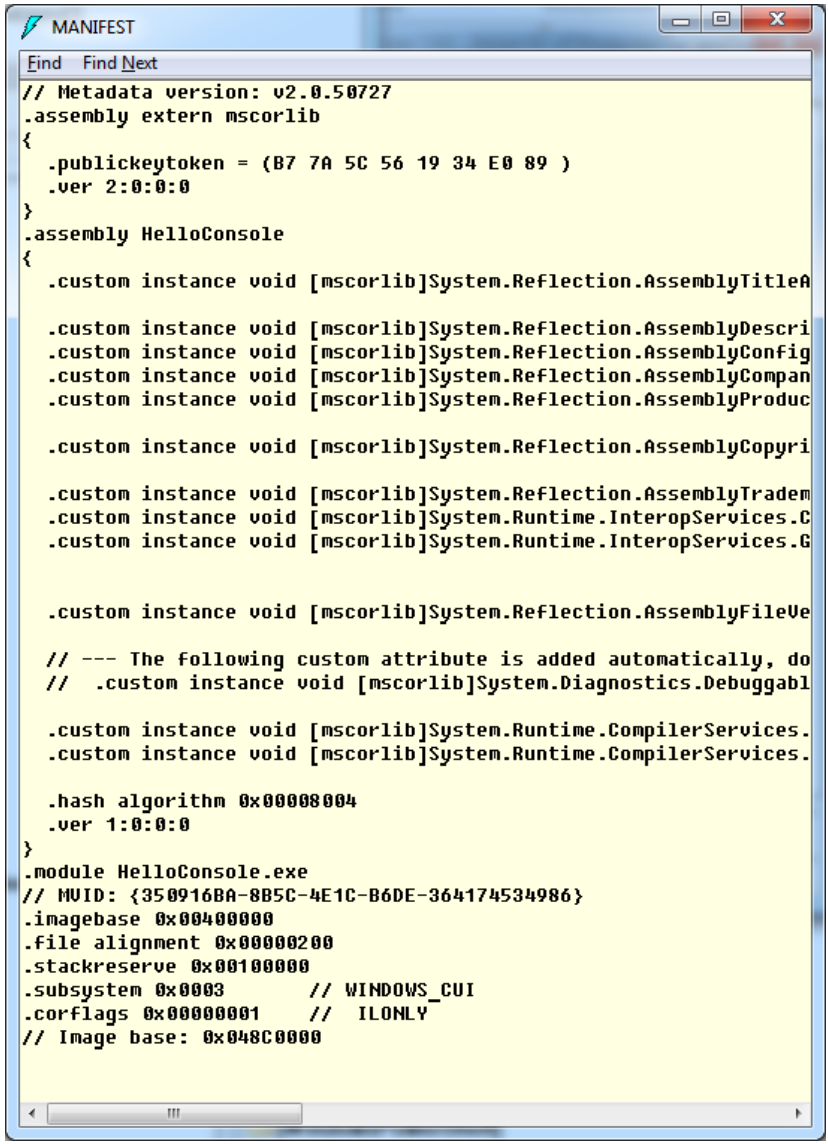
Для просмотра метаданных выполните утилиту ildasm. В окне программы выберите команду Open открытия файла



В открывшемся диалоговом окне найдите нужный файл сборки HelloConsole. Вы увидите содержимое сборки, из которой порождается бинарный файл. Сборка включает манифест и сведения о программе HelloConsole.



Для демонстрации манифеста необходимо выполнить двойной щелчок левой кнопкой мыши на разделе MANIFEST. Представление манифеста имеет следующий вид (отображена только левая часть окна).

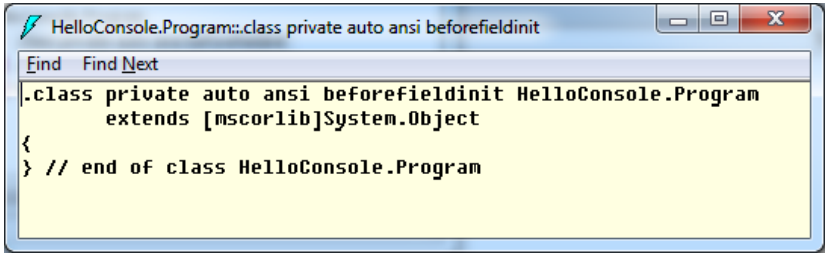


Манифест содержит полную системную информацию о сборке. Эта информация наиболее активно используется при разработке сложных динамически загружаемых приложений - для начала краткое описание содержимого манифеста сборки:

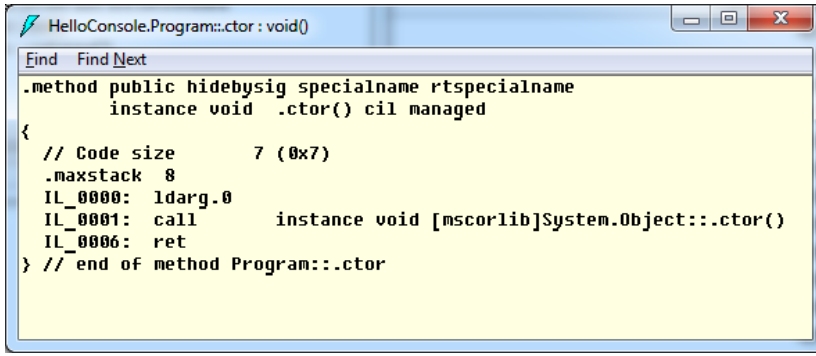
- Секция .assembly extern mscorlib, в ней описываются зависимости от внешних сборок, используемых в данной программе. Здесь для каждой сборки указывается версия и контрольная сумма. Эти данные берутся из сборок при компиляции программы, что гарантирует во время работы приложения использование именно тех сборок, которые использовались при компиляции и тестировании,
- Секция .assembly, но уже без модификатора extern. С этой директивы и начинается описание сборки.
- Секция .hash algorithm определяет функцию, по которой будет вычисляться хэш-код. Он нужен для подтверждения правильности сборки
- Секция .ver описывает версию сборки.
- Затем в манифесте располагаются описания имени самого программного модуля, подсистемы исполнения, информация о выравнивании секций и еще некоторые данные.

Папка HelloConsole содержит коды самой программы на CIL. Они распределены по вложенным папкам:

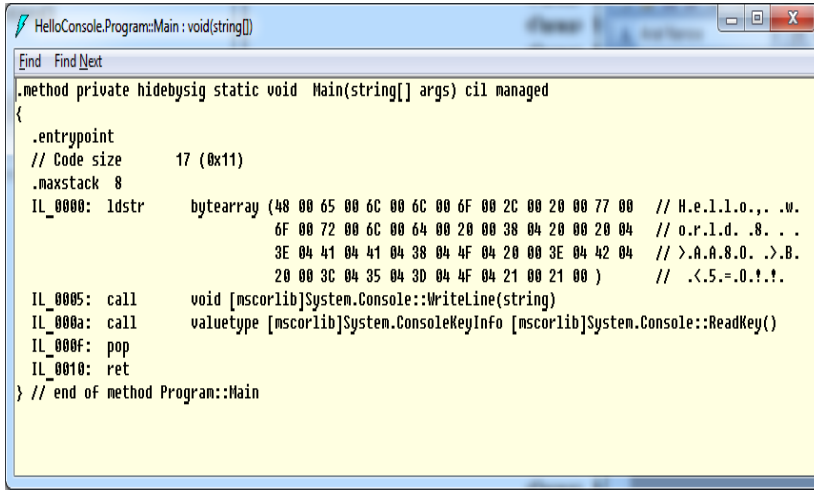
- Классы



- Методы



- Главная программа Main



Внимание. Писать программу на языке CIL вряд ли целесообразно. Пусть с ним работает CLR.

1.5.4. Сборка мусора

Одни из самых неприятных ошибок, которые портят жизнь программисту, это, безусловно, ошибки, связанные с управлением памятью. В таких языках, как C и C++, в которых управление памятью целиком возложено на программиста, львиная доля времени, затрачиваемого на отладку программы, приходится на борьбу с подобными ошибками.

Давайте перечислим типичные ошибки при управлении памятью (некоторые из них особенно усугубляются в том случае, если в программе существуют несколько указателей на один и тот же блок памяти):

- Преждевременное освобождение памяти (premature free). Эта беда случается, если мы пытаемся использовать объект, память для которого была уже освобождена. Указатели на такие объекты называются висящими (dangling pointers), а обращение по этим указателям дает непредсказуемый результат.
- Двойное освобождение (double free). Иногда бывает важно не перестараться и не освободить ненужный объект дважды.
- 3Утечки памяти (memory leaks). Когда мы постоянно выделяем новые блоки памяти, но забываем освобождать блоки, ставшие ненужными, память в конце концов заканчивается.
- Фрагментация адресного пространства (external fragmentation). При интенсивном выделении и освобождении памяти может возникнуть ситуация, когда непрерывный блок памяти определенного размера не может быть выделен, хотя суммарный объем свободной памяти вполне достаточен. Это происходит, если используемые блоки памяти чередуются со свободными блоками и размер любого из свободных блоков меньше, чем нам нужно.

Проблема особенно критична в серверных приложениях, работающих в течение длительного времени.

В программах, работающих в среде .NET, все вышеперечисленные ошибки никогда не возникают, потому что эти программы используют реализованное в CLR автоматическое управление памятью, а именно – сборщик мусора.

Работа сборщика мусора заключается в освобождении памяти, занятой ненужными объектами. При этом сборщик мусора также умеет «двигать» объекты в памяти, тем самым устраняя фрагментацию адресного пространства.

Все эти чудеса, которые творит сборщик мусора, возможны исключительно благодаря тому, что во время выполнения программы известны типы всех используемых в ней объектов. Другими словами, данные, с которыми работает программа, находятся под полным контролем среды выполнения и называются, соответственно, управляемыми данными (managed data).

2. Обзор ИСР

2.1. ИСР Visual C# Express

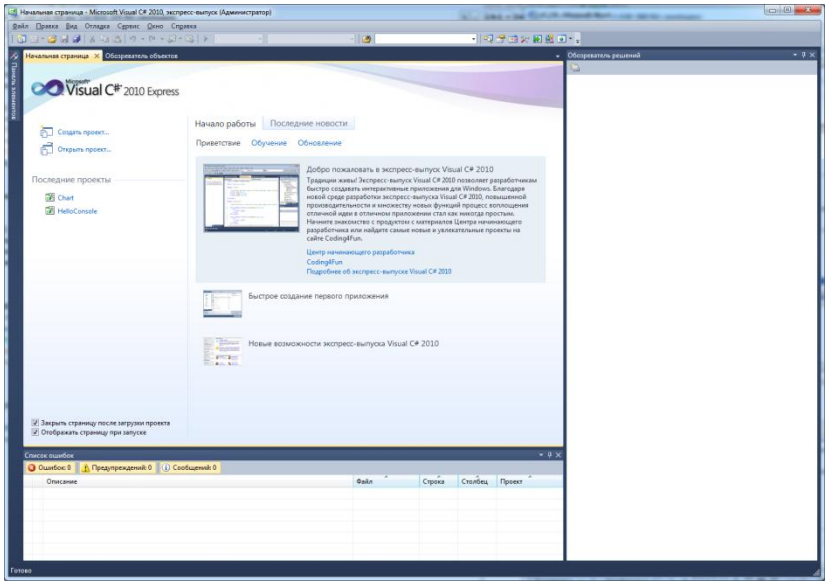
2.1.1. Стартовая страница

ИСР для создания проекта в .Net на языке C# позволяет создавать решения, включающие набор проектов.

Внимание. В русифицированной версии ИСР разделитель целой и дробной части числа:

- при работе с консолью – запятая.
- при наборе в редакторе кода – точка.

При запуске ИСР отображается стартовая страница.



Окно содержит встроенные окна. В центре главного окна на вкладках размещаются основные окна, а на периферии служебные окна.

В центре могут размещаться основные окна (на вкладках, если их несколько):

- Начальная страница.
- Дизайнеры.
- Редакторы кода.

Слева размещается панель инструментов с компонентами.

Справа размещаются:

- Обозреватель решений.
- Окно классов.
- Свойства.

Внизу размещается окно «Список ошибок».

Вид представления каждого окна можно изменить. Для этого вызвать выпадающее меню, в котором перечислены возможные решения. Это можно сделать двумя способами - щелчком правой кнопкой мыши по строке заголовка окна или стрелкой в строке заголовка. Меню содержит представления:

- Плавающая область. Окно автономное, может перемещаться произвольно. Опция позволяет делать снимок только этого окна.
- Закрепить. Окно докируется в основное окно и не может перемещаться автономно.
- Закрепить как вкладку. Окно размещается на вкладке в другом окне. Например, окно классов размещается на вкладке вместе с вкладкой Обозревателя решений.
- Автоматически скрывать. Окно отображается в виде узкой вертикальной полоски заголовка для экономии места. Это действие можно выполнить кнопкой в строке заголовка.
- Скрыть. Окно скрывается.

Окно «Начальная страница» содержит вложенные поля:

- Создать проект
- Открыть проект
- Последние проекты.
- Последние новости (есть при подключенном Интернете)..
- Начало работы.
- Приветствие.
- Обучение.
- Обновление.

В стартовом режиме меню содержит следующие команды

Файл	
Правка	
Вид	
Отладка	
Сервис	
Окно	

Справка	

Пункт «Файл содержит команды:

Создать проект…	Новый
Открыть проект…	Существующий
Открыть файл…	Существующий
Закрыть	
Закрыть решение	
Сохранить выбранные элементы	Под старым именем
Сохранить выбранные элементы, как…	Под новым именем
Сохранить все	
Экспорт шаблона…	
Параметры страницы…	
Печать…	
Последние файлы ►	Из предъявляемого списка
Последние проекты и решения ►	Из предъявляемого списка
Выход	

Пункт «Правка» содержит команды:

Отменить	
Вернуть	
Вырезать	
Копировать	
Вставить	
Выделить все	
Поиск и замена ►	Из предъявляемого списка
Перейти…	
Закладки ►	Из предъявляемого списка

Пункт «Вид» содержит команды для выбора отображаемых окон ИСР:

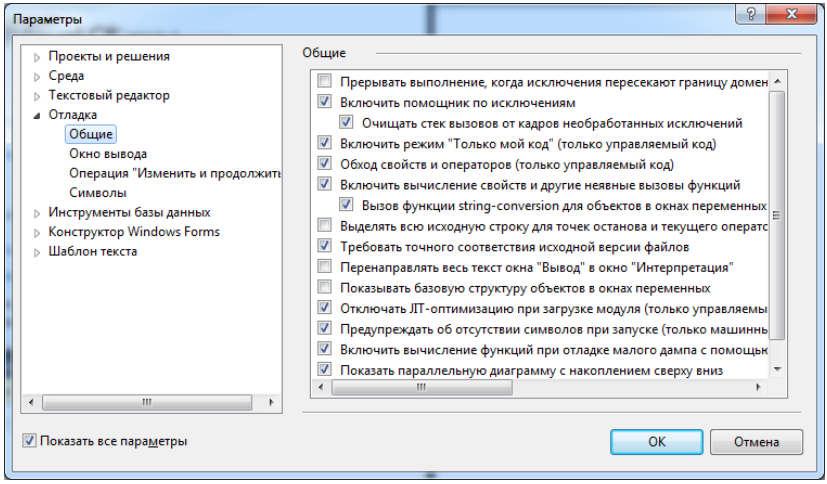
Классы	
Список ошибок	
Вывод	
Начальная страница	
Обозреватель решений	
Список задач	
Панель элементов	
Другие окна ►	Из предъявляемого списка
Панель инструментов ►	Из предъявляемого списка
Во весь экран	

Назад	
Вперед	
Окно свойств	
Окна свойств	

Пункт «Отладка» содержит команды:

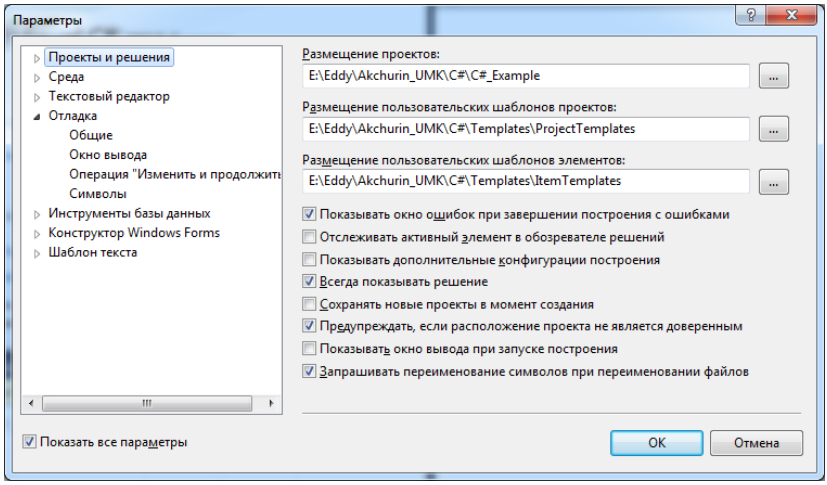
Параметры настройки…	

Команда вызывает окно «Параметры», в котором можно изменять параметры. Эта команда доступна и в пункте «Сервис». Окно имеет иерархическую структуру с множеством тематических вкладок.

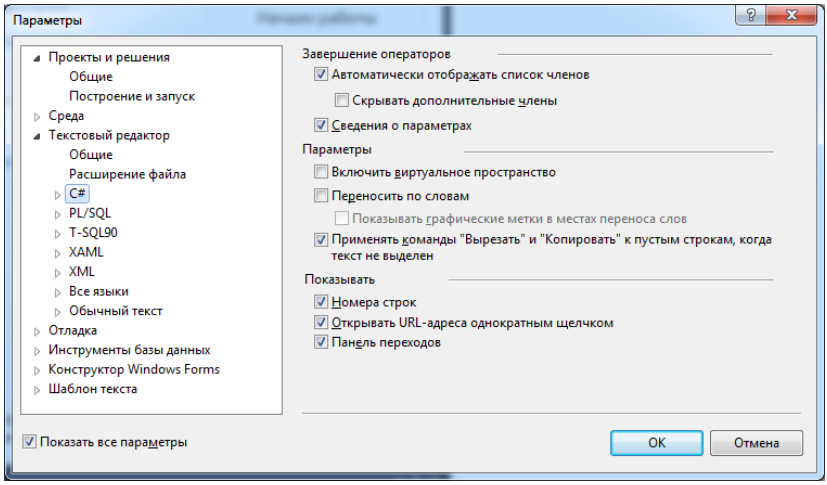


Целесообразно выполнить коррекцию параметров, выбираемых по умолчанию. Большая часть параметров не требует изменений.

Во вкладке «Проекты и решения => Общие» выбрать места размещения проектов, пользовательских шаблонов проектов, пользовательских шаблонов элементов.



Во вкладке «Текстовый редактор => C#» установить флаг отображения номеров строк.

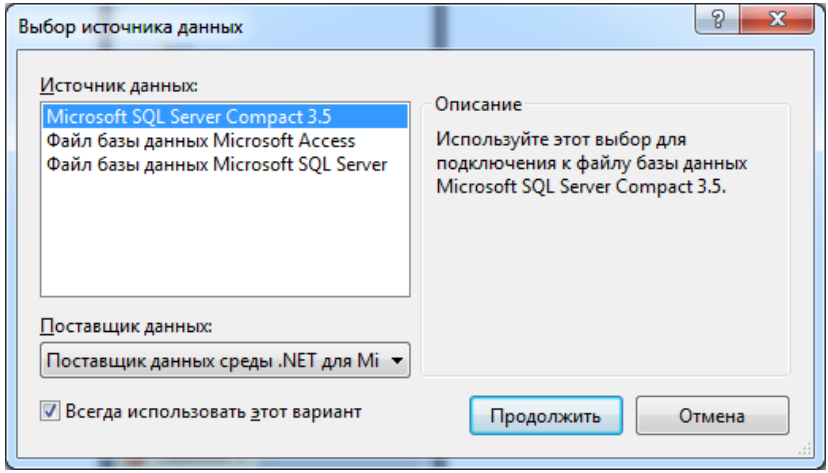


Пункт «Сервис» содержит команды:

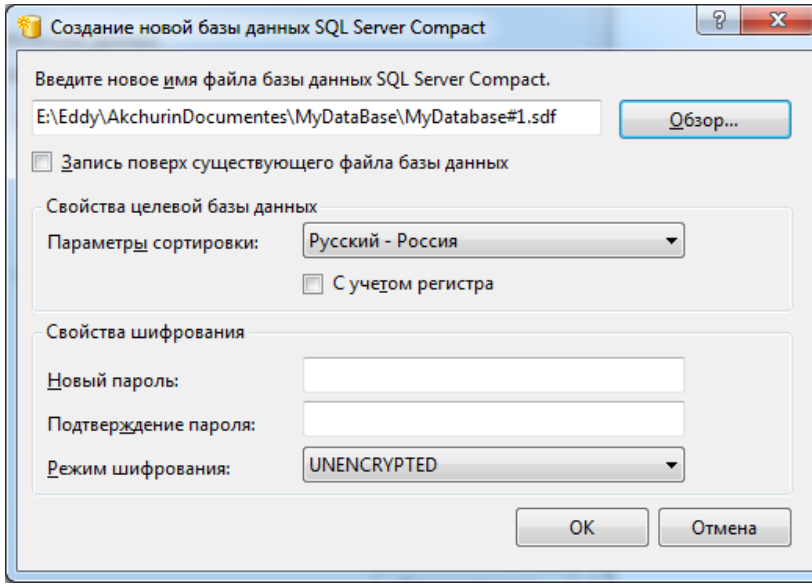
Подключиться к базе данных…	
Диспетчер фрагментов кода…	Вызов сохраненных кодов
Выбрать элементы панели элементов…	Обновление списка по умолчанию.
Диспетчер расширений…	Добавить, удалить или изменить.
Внешние инструменты…	Добавить, удалить или изменить.

Параметры ►	Из предъявляемого списка
Настройка…	Настройка панели инструментов.
Параметры…	Настройка компонент ИСР

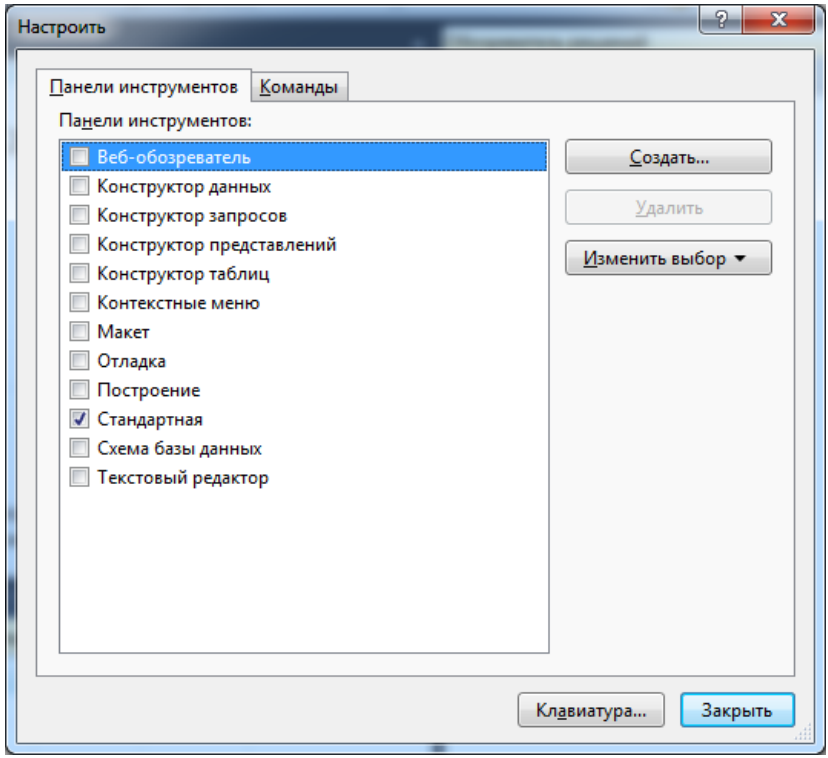
Команда «Подключиться к базе данных…» вызывает окно выбора источника данных. В нем можно выбрать источник.



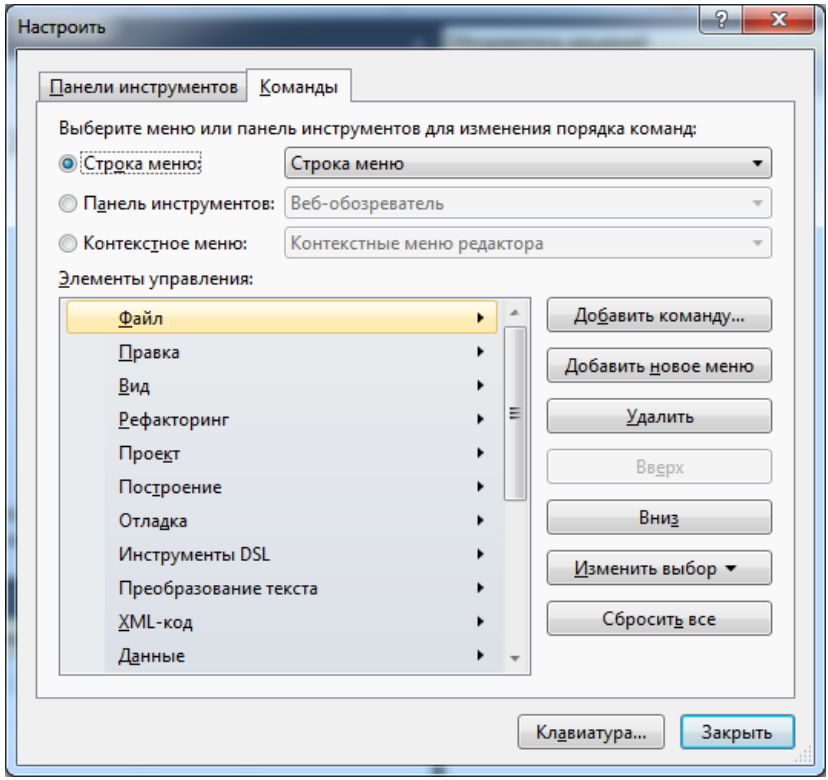
Команда «Продолжить» вызывает окно создания базы данных.



Команда «Настройка» вызывает диалоговое»Настроить», в котором имеются вкладки «Панели инструментов» и «Команды». В первой вкладке выбираются отображаемые панели инструментов.



Во второй вкладке можно изменить состав любой возможной панели инструментов.



Пункт «Окно» содержит команды:

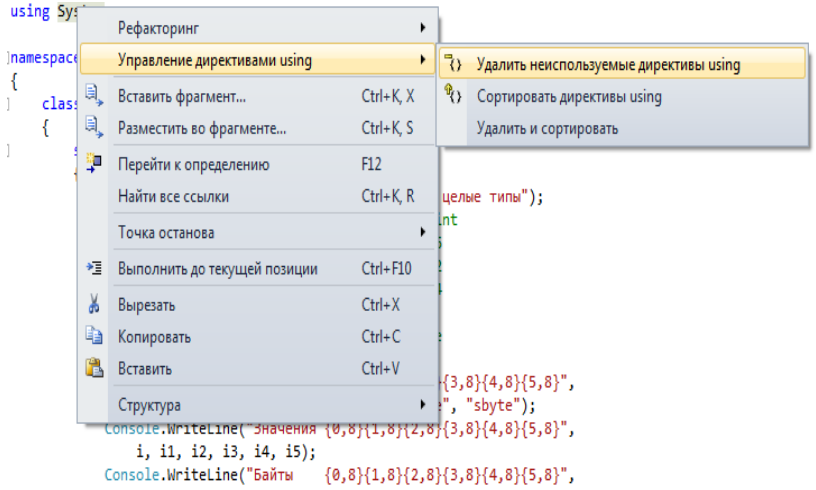
Создать окно	
Разделить	
Плавающая область	Окно автономное, может перемещаться произвольно. Опция позволяет делать снимок только этого окна.
Закрепить	Окно докируется в основное окно и не может перемещаться автономно.
Закрепить как вкладку	Окно размещается на вкладке в другом окне.
Автоматически скрывать	Окно скрывается, отображается узкой полоской.
Скрыть	Окно скрывается
Автоматически скрывать все	
Создать группу горизонтальных вкладок	
Создать группу вертикальных	

вкладок	
Закрыть все документы	
Сброс макета окон	
Начальная страница	
Обозреватель объектов	
Окна…	

Пункт «Справка» содержит команды:

Просмотр справки	Доступ к инструкциям по C#
Управление параметрами справки	
Форумы MSDN	Доступ в Интернет
Сообщить об ошибке	Разработчикам C#
Примеры	
Параметры отзывов пользователей	
Зарегистрировать продукт	
Проверить наличие обновлений	
Техническая поддержка	
Заявление о конфиденциальности…	
О программе	

Внимание. ИСР для каждого нового проекта использует шаблон, в который нужно добавить функциональность. ИСР создает перечень доступных пространств имен директивами using по умолчанию. Часть из них не используются. Их можно удалить. Щелчок правой кнопки по коду программы вызывает выпадающее меню, в котором нужно выбрать показанное.



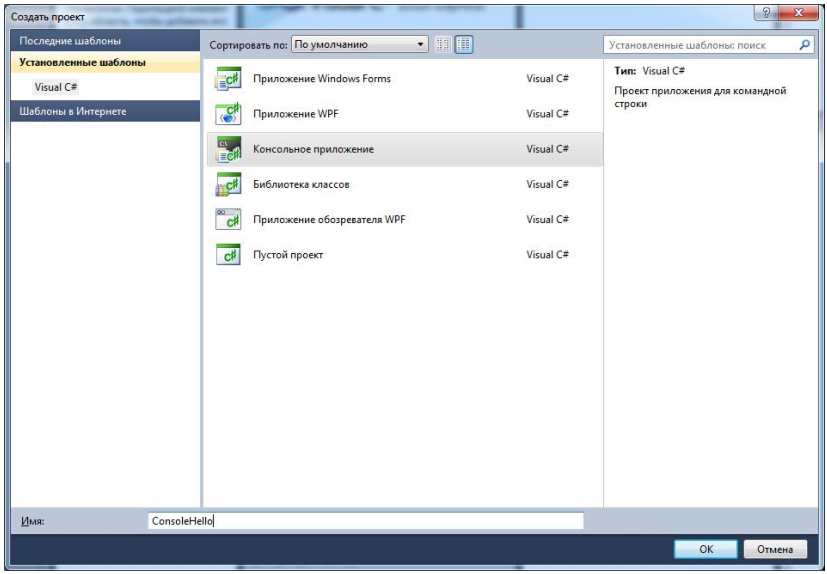
Далее нужно выбрать, что делать. Создаем новый проект.

2.1.2. Консольное приложение

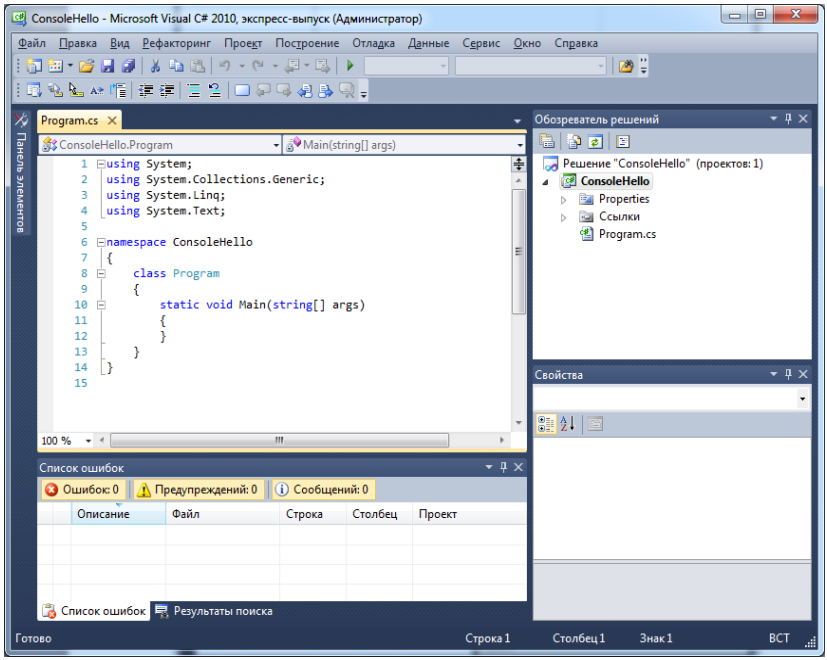
Для иллюстрации создаем проект консольного приложения «ConsoleHello», который будет входить в решение «ConsoleHello».

Консольные приложения предназначены для работы с командной строкой, в которой используется символьный интерфейс.

При запуске ИСР отображается стартовая страница. В ней нужно исполнить команду «**Файл => Создать проект**». Отображается окно выбора типа проекта. В нем нужно выбрать «Консольное приложение» и задать имя проекта – ConsoleHello.



Проект создается, отображаются его компоненты.



В режиме создания проекта меню ИСР меняется. Теперь оно содержит пункты:

Файл	
Правка	
Вид	
Проект	
Построение	
Отладка	
Данные	
Сервис	
Окно	
Справка	

Новый пункты «Проект» содержит команды:

Добавить форму Windows…	Из предъявляемого списка
Добавить пользовательский элемент управления…	Из предъявляемого списка
Добавить класс…	Из предъявляемого списка
Добавить новый элемент…	Из предъявляемого списка

Существующий элемент…	Из ресурсов проекта
Создать папку	Новую
Показать все файлы	Проекта
Добавить ссылку…	Из предъявляемого списка
Добавить ссылку на службу…	Из предъявляемого списка
Назначить запускаемым проектом	Из множества проектов в решении
Обновить элементы проекта в панели элементов	
Свойства ConsoleHello	

Новый пункт «Построение» содержит команды:

Построить решение	Компиляция
Перестроить решение	Компиляция с оптимизацией
Опубликовать ConsoleHello	Вызывается мастер создания папки с исполняемым файлом в другом месте.

Пункт «Отладка» теперь содержит команды:

Окна ►	Выбор окон вывода: Вывод, Интерпретация
Начать отладку	
Запуск без отладки	
Исключение	
Шаг с заходом	С заходом в подпрограммы
Шаг с обходом	Подпрограмма за шаг
Точка останова	
Очистить все подсказки по данным	
Экспорт подсказок по данным	
Экспорт подсказок по данным	
Точка останова	

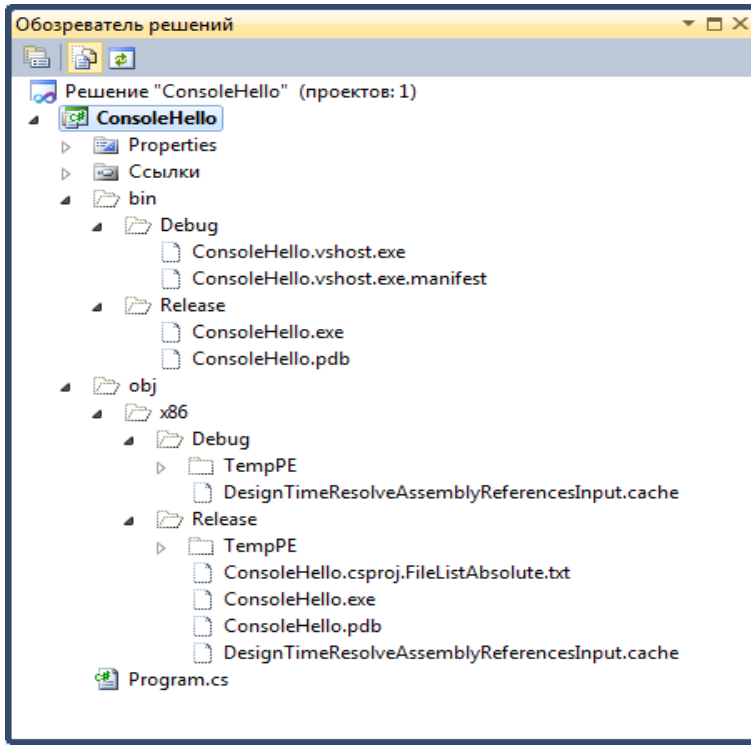
Новый пункт «Данные» содержит команды:

Показать источники данных…	Из предъявляемого списка
Добавить новый источник данных…	

Теперь сохраняем все командой **Файл => Сохранить все**. По умолчанию создается решение с именем главного проекта. В папке решения размещаются:

- Вложенная папка проекта.

- Файл ConsoleHello.sln. Текстовое описание решения.
- Файл ConsoleHello.suo. Генератор решения.



Решение создается в структуре папок. В папке **Решение “ConsoleHello”** размещается папка проекта **ConsoleHello, которая включает:**

- Папку Properties, свойства проекта. Файл AssemblyInfo.cs содержит общие сведения о сборке.
- Папка Ссылки содержит ссылки на используемые в сборке пространства имен.
- Папка bin содержит финальные бинарные файлы.
- Папка obj содержит промежуточные файлы, из которых при компоновке получаются бинарные файлы. Консольное приложение делается под процессор с форматом x86. Поэтому имеется вложенная папка x86. Если объектный файл один, то он будет совпадать с бинарным. Папка TempPE служит для хранения временных PE файлов.
- Program.cs – исходник на C#.

В каждой из папок bin и obj есть вложенные папки **Debug** и **Release.** При разработке приложения используются два режима:

- Построить решение. При компиляции в файл включается отладочная информация, оптимизация компилятора отключается. Это может увеличить размеры файлов. Формируемые файлы помешаются в папки Debug.
- Перестроить решение. Создание финального продукта. При компиляции в файл не включается отладочная информация, оптимизация компилятора включается. Это может уменьшить размеры файлов. Формируемые файлы помешаются в папки Release.

При сохранении проекта создаются файлы:

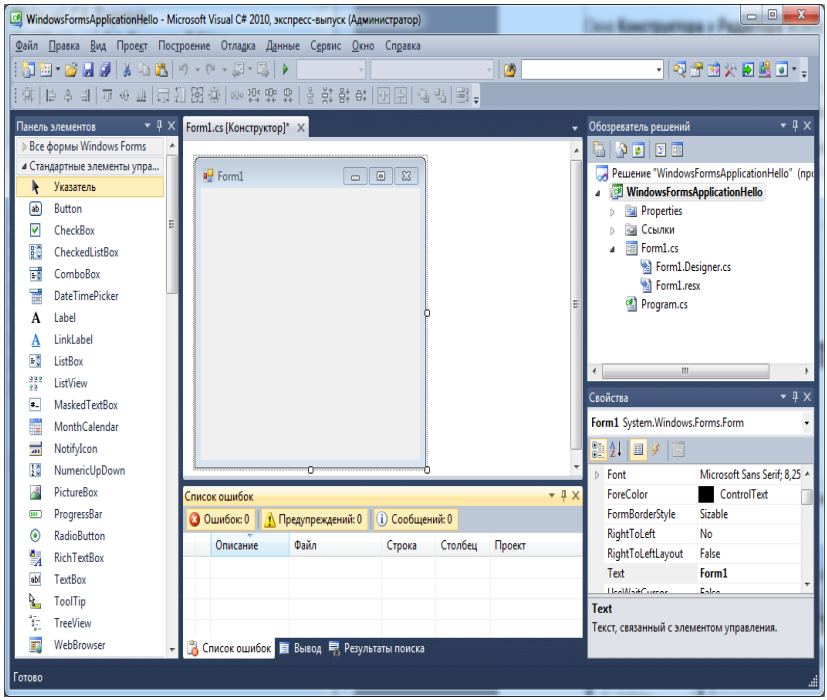
Program.cs	Программа на C#
ConsoleHello.exe	Исполняемый PE файл
ConsoleHello.pdb	База данных, описывающая его
ConsoleHello.vshost.exe	Вызов компилятора
ConsoleHello.vshost.exe.manifest	Манифест компилятора
ConsoleHello.csproj.FileListAbsolute.txt	Список файлов
DesignTimeResolveAssemblyReferencesInput.cache	Кэш проекта

Если выпускаемое приложение будет использоваться без поддержки платформы .Net, то нужно сформировать исполняемый файл для Win32, используя процедуру опубликования. Она использует встроенного мастера, который проводит вас через эту процедуру. В специальную папку published загружаются манифест файла IL с его описанием и setup.exe - инсталлятор Windows. При запуске инсталлятора в этой папке по манифесту формируется исполняемый файл под ОС компьютера развертывания приложения. Соотношение его размеров с размерами файла IL зависит от ОС и аппаратных средств.

2.1.3. Приложения Windows

В приложении Windows с графическим пользовательским интерфейсом большая часть действий после запуска происходит в ответ на действия пользователя. Все поля, хранящие сведения о состоянии приложения, находятся в основном классе Form, имеющем по умолчанию имя Form1.

При создании нового проекта из вкладки **Последние проекты** или командой **Файл => Создать проект** вызывается окно выбора проекта с набором шаблонов и полем имени проекта. В нем выбираем **Приложение Windows Forms**. Проекту назначаем имя Windows FormsApplicationHello.



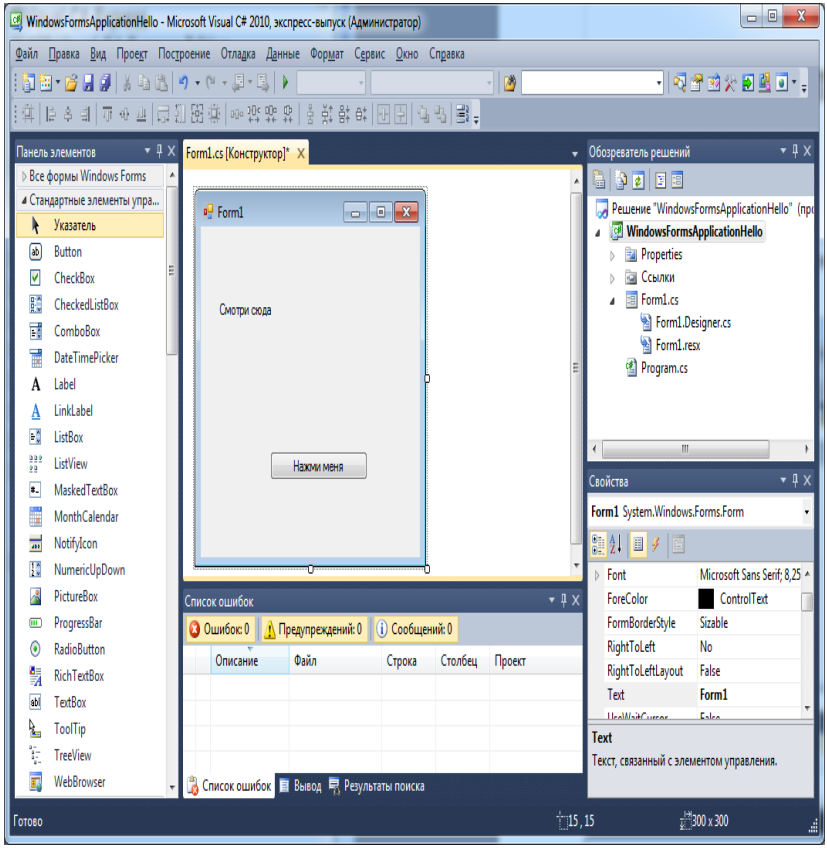
В центре в закладке **Form1.cs[Конструктор**] отображается окно **Конструктора** формы. Окно **Панель элементов** заполняется элементами для выбора. Окна **Обозреватель решений** содержит описание решения. **Конструктор** формы отображается по автоматически создаваемому коду (при желании его можно посмотреть двойным щелчком по **Form1.cs => Program.cs** в **Обозревателе решений**). **Редактор** кода модуля формы отображается командой **Перейти к коду**, которая находится в меню, выпадающем при щелчке по форме в конструкторе правой кнопкой мыши. Редактор отображается в закладке с именем **Form1.cs**.

Большая часть кода в **Редакторе** ИСР сделала автоматически. Нужно добавить функциональность.

Окна **Конструктора** и **Редактора** можно переключать кнопками в заголовках их закладок.

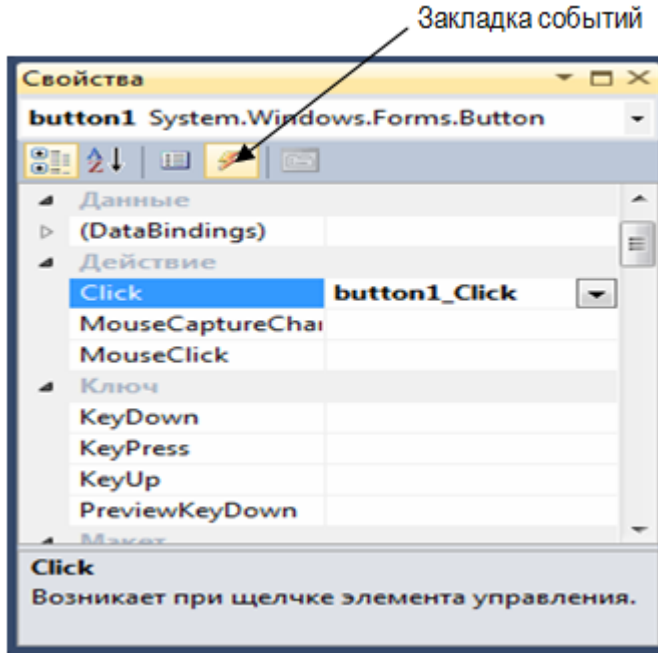
Теперь приступаем к проектированию в **Конструкторе**. Из окна Панели элементов перетаскиваем в форму объекты

- button1 – кнопка для запуска обработчика события. Выделяем объект, в окне свойств отображаются свойства кнопки. Свойству Text присваиваем значение - Нажми меня.
- label1 – метка, поле для отображения сообщения. Свойству Text присваиваем значение – Смотри сюда.



Для создания обработчика события щелчка по кнопке дважды щелкаем по кнопке в форме. Автоматически отображается окно **Редактора**, в котором в код добавлен шаблон обработчика события button1_Click, но без функциональности. Курсор устанавливается в место ввода кода, который будет задавать функциональность проекта.

Чтобы обработчик события срабатывал, нужно в окне свойств кнопки button1 в закладке событий выбрать реакцию на щелчок по кнопке (button1_Click) из списка:



В шаблон кода, начиная с позиции курсора, нужно ввести инструкции. В примере свойству Text объекта label1 нужно присвоить строку “Hello, world and Россия от меня!!”. Чтобы исключить повторный доступ к кнопке, сделаем ее после вывода текста невидимой. Для этого вводим код

```
label1.Text= "Hello, world и Россия от меня!!";
button1.Visible = false;
```

Итоговый код:

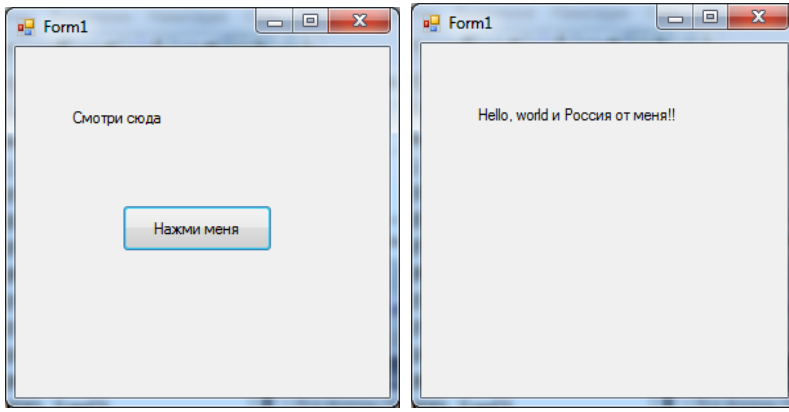
Form1.cs

WindowsFormsApplication1.Form1 | button1_Click(object sender, EventArgs e)

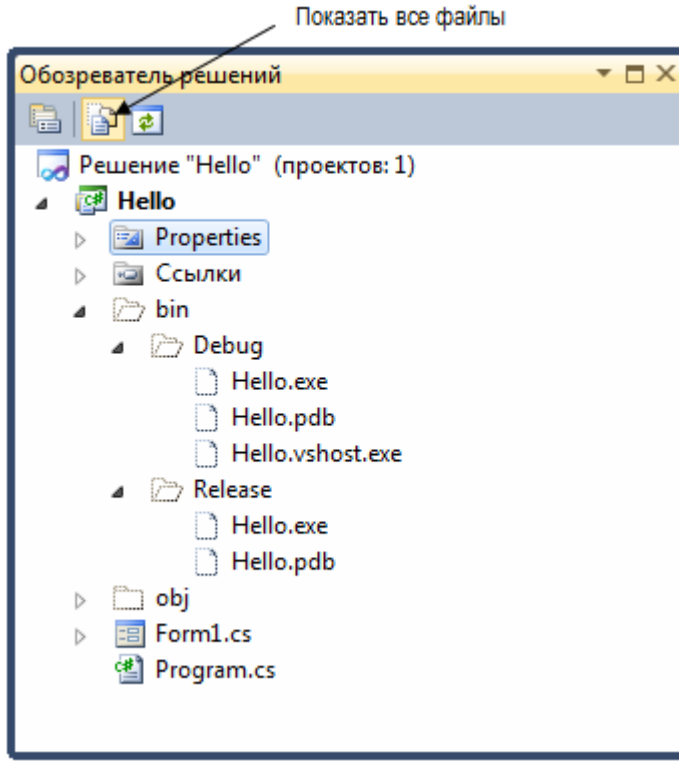
```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Drawing;
6  using System.Linq;
7  using System.Text;
8  using System.Windows.Forms;
9
10 namespace WindowsFormsApplication1
11 {
12     public partial class Form1 : Form
13     {
14         public Form1()
15         {
16             InitializeComponent();
17         }
18
19         private void button1_Click(object sender, EventArgs e)
20         {
21             label1.Text = "Hello, world и Россия от меня!!";
22             button1.Visible = false;
23         }
24     }
25 }
26
```

100 %

Проект готов. проверим его командой **Отладка => Запуск без отладки**. Получим окна приложения: слева стартовое, справа финальное.



Проект готов, сохраняем его командой **Файл => Сохранить все**. В папке Hello. В результате проект сохраняется в структуре папок (чтобы увидеть все файлы, нужно активизировать кнопку, показанную на рисунке):



Решение включает:

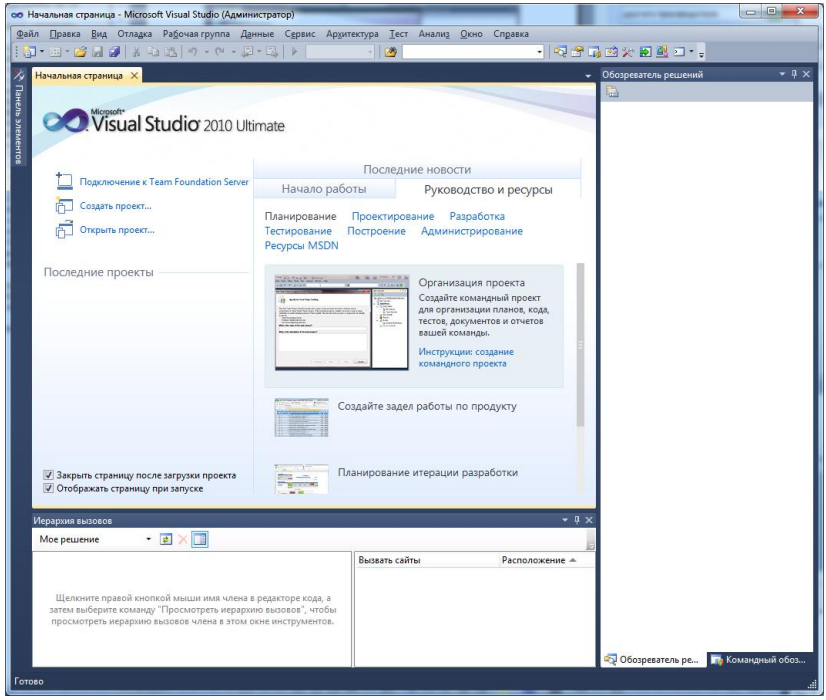
Решение Hello	Решение
Hello	Проект
Properties	Свойства
Ссылки	Ссылки
bin	Двоичные файлы
Debug	Файлы отладки
Hello.exe	Управляемый исполняемый файл
Hello.pdb	База данных для JIT компилятора
Hello.vshost.exe	Служебный файл
Release	Файлы выпуска
obj	Объектные файлы

Исполняемые (bin) и объектные (obj) файлы образуются при компиляции (построении). Возможны два режима:

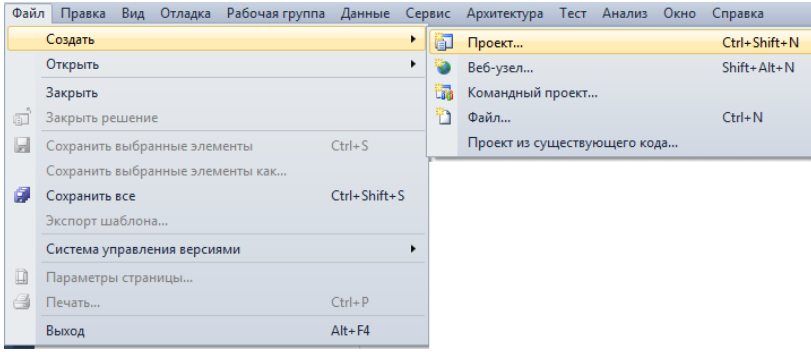
- Команда **Построение=>Построить решение**. Построение в режиме отладки, в компонуемые файлы включаются символы отладки и режим оптимизации исключается. Это может увеличить размеры файлов. Файлы размещаются в папках Debug.
- Команда **Построение=>Перестроить решение**. Построение отлаженного проекта, когда в компонуемые файлы символы отладки не включаются и компилятор использует режим оптимизации кода (например, исключает не использованные переменные). Это может уменьшить размеры файлов. Файлы размещаются в папках Release.

2.2. ИСР Visual Studio .Net

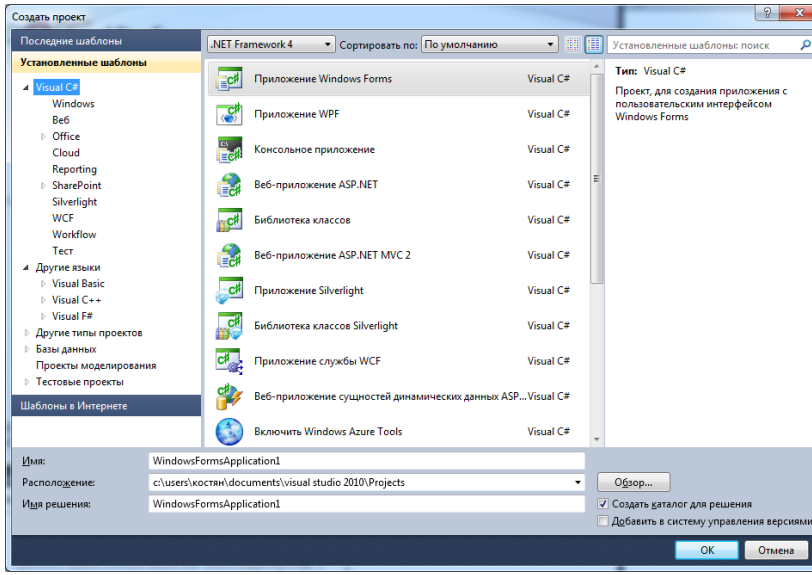
При запуске ИСР отображается стартовая страница.



Для создания нового проекта исполняется команда Файл=>Создать=>Проект



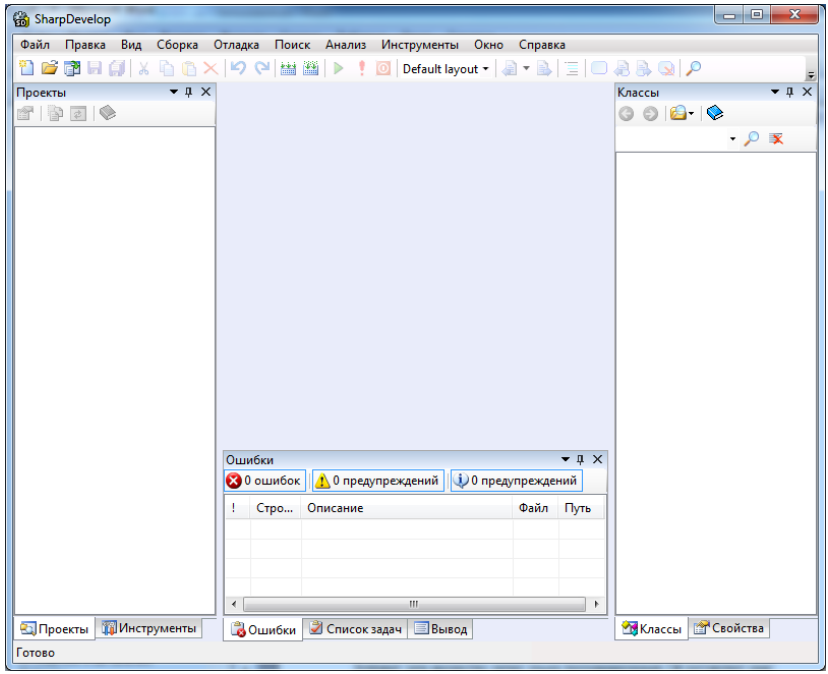
Отображается окно выбора языка программирования и типа проекта.



Далее как в других ИСР.

2.3. ИСР SharpDevelop

При запуске ИСР отображается стартовая страница.

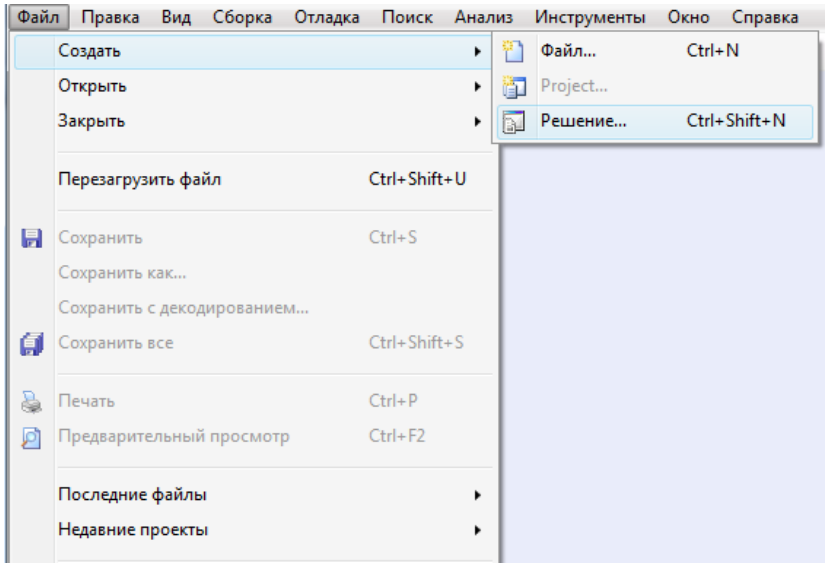


Окно содержит встроенные окна. В центре главного окна на вкладках размещаются основные окна, а на периферии служебные окна.

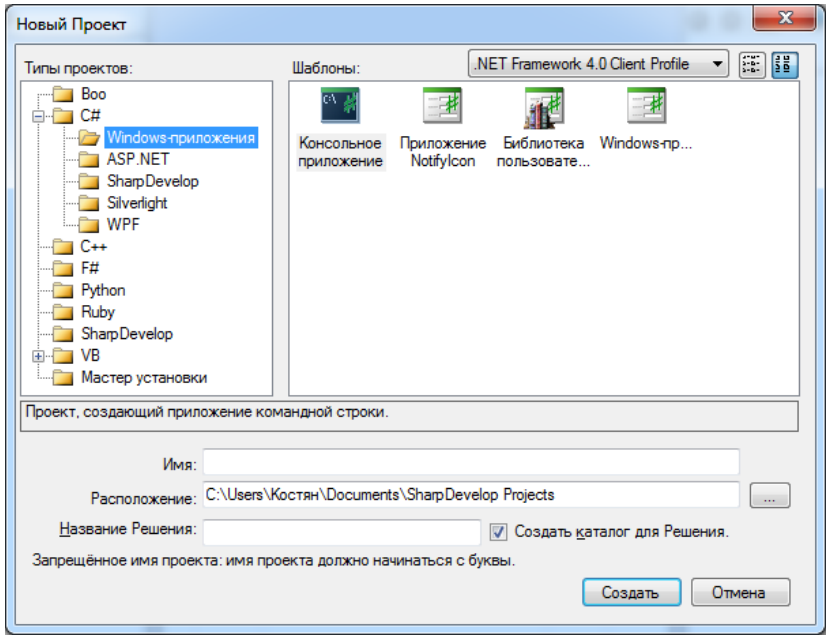
В центре могут размещаться основные окна (на вкладках, если их несколько):

- Начальная страница.
- Дизайнеры.
- Редакторы кода.
- Ощибки

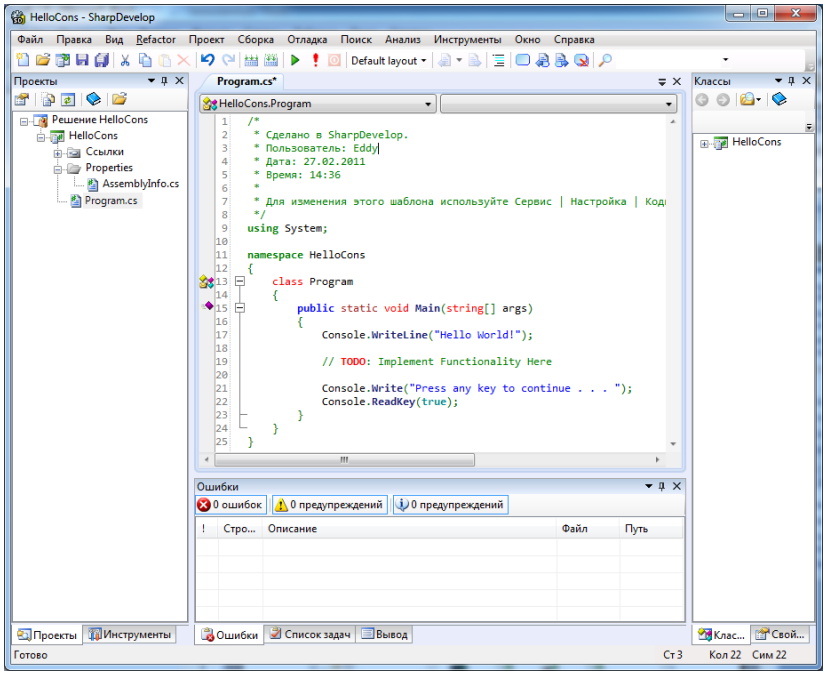
Для создания нового проекта исполняется команда Файл=>Создать=>Решение



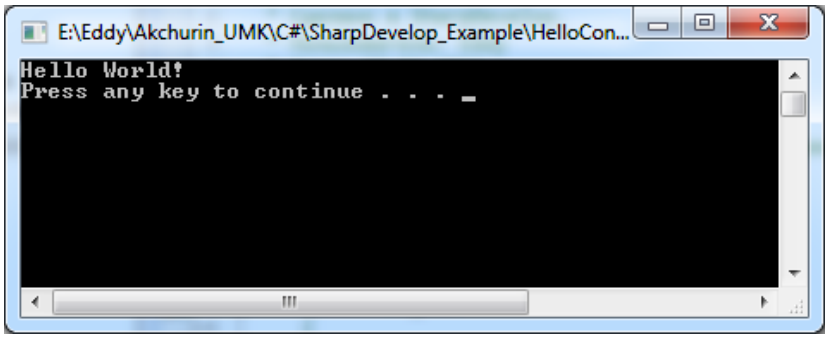
Отображается окно выбора языка программирования и типа проекта.



При выборе языкаC#, Windows приложения и шаблона Консольное приложение отображается окно проекта, в котором вставлен шаблон кода вывода в консоль фразы..



При запуске отображается консоль.



3. Основы языка C#

3.1. Алфавит

Алфавит (или множество литер) языка программирования C# составляют символы таблицы кодов unicod. Алфавит C# включает:

- строчные и прописные буквы латинского алфавита;
- строчные и прописные буквы национального алфавита;
- цифры от 0 до 9;
- символ «_»;
- набор специальных символов: " { }, 1 [ ] + — %/ \; ' : ? < > = ! & # ~ *-
- прочие символы.

Алфавит C# служит для построения слов, которые в C++ называются лексемами. Различают типы лексем:

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

Почти все типы лексем (кроме ключевых слов и идентификаторов) имеют собственные правила словообразования, включая собственные подмножества алфавита.
Лексемы обособляются разделителями. Этой же цели служит множество пробельных символов, к числу которых относятся пробел, табуляция, символ новой строки и комментарии.

3.2. Комментарии

Комментарии – это фрагменты кода, которые игнорируются компилятором. В C# определены комментарии:

- // - комментарий до конца строки.
- /* - символы начала многострочного комментария.
- */ - символы конца многострочного комментария.

3.3. Идентификаторы

В языке C# используется кодировка unicod. Это означает:

- **Чувствительность к регистру**, M и m - это разные переменные.
- Допустимо использовать для идентификаторов символы **кириллицы**.

На имена накладываются ограничения.

- Первый символ – буква.
- Следующие символы – буквы, цифры, символ подчеркивания.

В языке C# для многословных имен **не принято** применять символы подчеркивания. Рекомендуется разделять слова, используя в начале слова заглавные буквы. Возможны два стиля имен:

- Pascal, с заглавной буквы начинается каждое слово идентификатора, включая первое. Например, ЭтоМойИдентификатор.
- Camel, первая буква строчная, но имя может содержать заглавные буквы (визуально это горбы, camel = верблюд). Например, этоМойИдентификатор.

В типах .NET Framework используется иерархическая схема именования с точкой. При таком подходе связанные типы группируются в пространства имен, что упрощает их поиск и создание ссылок.

Первая часть полного имени — до крайней правой точки — это имя пространства имен. Последняя часть имени — это имя типа. Например,

System.Collections.ArrayList

представляет собой тип ArrayList, который принадлежит пространству имен System.Collections. Типы в System.Collections можно использовать для работы с коллекциями объектов.

3.4. Ключевые слова

Ключевые слова — это предварительно определенные зарезервированные идентификаторы, имеющие специальные значения для компилятора.

Ключевые слова используются для инструкций (команд) C#.

Их нельзя использовать в программе в качестве идентификаторов, если только они не содержат префикс @.

Например, @if является допустимым идентификатором, но if таковым не является, поскольку if — это ключевое слово. Идентификаторы с символом @ применять не рекомендуется.

В ИСР предусмотрены встроенные средства контроля правильности идентификаторов. ИСР не примет неправильный идентификатор, вы получите сообщение о допущенной ошибке.

Перечень ключевых слов:

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

3.5. Переменные и константы

Переменная представляет числовое или строковое значение или объект класса. Значение, хранящееся в переменной, может измениться, однако имя остается прежним.

Переменная представляет собой один тип поля. Переменная может быть объявлена в любом месте кода. При объявлении нужно указать тип переменной, задавать ее значение не обязательно.

Следующий код является простым примером объявления целочисленной переменной, присвоения ей значения и последующего присвоения нового значения.

```
int x = 1;  // x получает значение 1
x = 2;      // x получает значение 2
```

В C# переменные объявляются с определенным типом данных и именем. Тип указывает, помимо всего прочего, точный объем памяти, который следует выделить для хранения значения при выполнении приложения. При преобразовании переменной из одного типа в другой язык C# следует определенным правилам.

Константа является другим типом поля. Она хранит значение, присваиваемое по завершении компиляции программы, и никогда после этого не изменяется. Константы объявляются помощью ключевого слова const; их использование способствует повышению удобочитаемости кода

```
const int speedLimit = 55;
const double pi = 3.14159265358979323846264338327950;
```

4. Структура программы на C#

Чтобы понять принципы работы программы C#, давайте рассмотрим традиционную программу "Hello World!" и разберем каждую строку ее кода на C#.

Для упорядочения и оформления кода в языке C# используются классы. В действительности весь выполняемый код C# должен содержаться в классе, что справедливо и для короткой программы типа "Hello World!". Ниже приведен код программы, отображающей в окне консоли сообщение "Hello World!".

```
// A Hello World! program in C#
using System;
namespace HelloWorld
{
   class Hello
   {
      static void Main()
      {
         Console.WriteLine("Hello World!");
         Console.WriteLine("Нажмите любую клавишу");
         Console.ReadKey();
      }
   }
}
```

4.1. Пространства имен

Пространства имен представляют собой способ организации различных типов, присутствующих в программах C#. Их можно сравнить с папкой в компьютерной файловой системе. Подобно папкам, пространства имен определяют для классов уникальные полные имена. Программа C# содержит одно или несколько пространств имен, каждое из которых либо определено программистом, либо определено как часть написанной ранее библиотеки классов.

Например, пространство имен System содержит класс Console, который включает методы для чтения и записи в окне консоли.

При написании класса вне объявления пространства имен компилятор предоставит ему заданное по умолчанию пространство имен.

Для использования метода WriteLine, определенного в классе Console, который содержится в пространстве имен System, без предварительного определения пространства имен следует использовать строку кода

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

Необходимость помнить, что всем методам, содержащимся в Console, должно предшествовать System. Это быстро становится утомительным, поэтому в начало исходного файла C# целесообразно вставить директиву using, задающую пространство имен

using System;

Директива устанавливает, что предполагается пространство имен System и впоследствии можно написать Console.WriteLine("Hello, World!");

Если вставить директиву using, задающую пространство имен System.Console

using System.Console;

то предполагается пространство имен System.Console и впоследствии можно написать WriteLine("Hello, World!");

4.2. Main() и аргументы командной строки

Метод **Main** является точкой входа консольного приложения C# или приложения Windows (для библиотек и служб не требуется метод Main в качестве точки входа). При запуске приложения метод Main является первым вызываемым методом.

В программе C# возможна только одна точка входа.

```
class TestClass
{
   static void Main(string[] args)
   {
         // Здесь команды
   }
}
```

- static – ключевое слово, определяет способ выделения памяти под экземпляр.
- void – ключевое слово, определяет, что метод не возвращает значений. Главной программе некуда возвращать значения.
- (string[] args) – аргументы, передаваемые программе. Если программе передаются аргументы, то они передаются в виде массива с указанием типа и имени. Например, string[] – тип массива строк, arg – имя этого массива. Альтернативный вариант - простое перечисление пар (тип - значение) с разделением запятыми.
- Если программе не передаются аргументы, то можно просто Main().

5. Операторы

Синтаксис операторов в C# сходен с синтаксисом других языков программирования в стиле языка C. Операторы используются для выполнения вычислений, назначения значений, проверки на равенство и неравенство и т. д.

Язык C# предоставляет большой набор операторов, которые представляют собой символы, определяющие операции, которые необходимо выполнить с выражением.

Операторы в выражениях исполняются с приоритетами. Высший приоритет имеют основные операторы, далее мультипликативные, затем аддитивные.

5.1. Основные операторы

Оператор	Действие
x.y	Оператор "точка" используется для доступа к членам класса. Формат - Класс.Член.
(x)	Круглые скобки (…) используются для указания порядка выполнения операций в выражении. Наивысший приоритет – операции в самых внутренних скобках.
A[x]	Квадратные скобки [ ]. Используются для доступа к элементу массива, его индекс необходимо заключить в скобки. Для многомерных массивов индексы разделяются запятыми.
new	Используется для создания экземпляра класса. Class1 Имя = new Class1();
Typeof(имя типа)	Используется для получения типа объекта. System.Type type = typeof(имя типа);
checked	Ключевое слово. Используется для явного включения проверки переполнения при выполнении арифметических операций и преобразований с данными целого типа.
unchecked	Ключевое слово. Используется для подавления проверки переполнения при выполнении арифметических операций и преобразований с данными целого типа. Если в непроверяемом контексте результатом выполнения выражения является значение, выходящее за допустимые пределы значений конечного типа, то результат усекается.
->	Оператор -> объединяет разыменование указателя и доступ к члену класса.

5.2. Унарные операторы

Оператор	Действие
+ X	Унарный плюс. Это знак числа X. Он использован по умолчанию.
++X	Префиксный унарный плюс. Увеличение X на 1 перед использованием.
X ++	Постфиксный унарный плюс. Уменьшение X на 1 после использования.
- X	Унарный минус. Это знак числа X.
-- X	Префиксный унарный минус. Уменьшение X на 1 перед использованием.
X --	Постфиксный унарный минус. Уменьшение X на 1 после использования.
! X	Логическое отрицание. Унарный оператор, который выполняет над операндом X операцию НЕ. Он задан для типа bool и меняет значение операнда true на false, или наоборот.
~X	Поразрядное дополнение. Инвертирование каждого бита целого X.
(T) x	Явное преобразование x в тип T
& X	Возвращает адрес X.
sizeof(X)	Размер в байтах для X.

5.3. Аддитивные операторы

Оператор	Действие
X + Y	Сложение. Для числовых типов он вычисляет сумму X + Y. Для строкового типа он объединяет X и Y.
X - Y	Вычитание. Для числовых типов вычисляет разность X - Y.
X | Y	Логическое сложение (ИЛИ - OR). Вычисляет X и Y независимо от значения X.
X || Y	Условное логическое сложение (ИЛИ - OR). Вычисляет Y в зависимости от X. Если X предопределяет результат, то Y не вычисляется.
X ^ Y	Сложение по модулю 2 (Исключающее ИЛИ - XOR). Вычисляет X и Y независимо от значения X.

5.4. Мультипликативные операторы

Оператор	Действие

X * Y	Умножение. Вычисляет произведение двух операндов.
X / Y	Деление. Делит X на Y. При делении целых чисел результат всегда является целочисленным. Остаток отбрасывается.
X % Y	Остаток. Вычисляет остаток после деления X на Y.
X & Y	Логическое умножение (И - AND). Вычисляет X и Y независимо от X.
X && Y	Условное логическое умножение (И - AND). Вычисляет Y в зависимости от X. Если X предопределяет результат, то Y не вычисляется.

5.5. Операторы сдвига

Применяются для целых чисел.

Оператор	Действие
X >> Y	Сдвиг вправо. Сдвигает биты X вправо на число бит, заданное Y (целое число). Если тип X — целое со знаком, то сдвиг арифметический (пустым старшим разрядам задается знаковый бит). Если тип X — целое без знака, сдвиг логический (старшие разряды заполняются нулями).
X << Y	Сдвиг влево. Сдвигает биты X влево на число бит, заданное Y (целое число). Освобождающиеся разряды заполняются нулями. Если тип X — целое со знаком, сдвиг арифметический (знаковый бит не трогается). Если тип X — целое без знака, сдвиг логический.

5.6. Операторы отношений

Оператор	Действие
X == Y	Равно. Возвращает значение true, если X равно Y, в противном случае возвращается значение false.
X != Y	Не равно. Возвращает значение true, если X не равно Y, в противном случае возвращается значение false.
X < Y	Меньше. Возвращает значение true, если X меньше Y, в противном случае возвращается значение false.
X > Y	Больше. Возвращает значение true, если X больше Y, в противном случае

	возвращается значение false.
X <= Y	Меньше или равно. Возвращает значение true, если X меньше или равно Y, в противном случае возвращается значение false.
X >= Y	Больше или равно. Возвращает значение true, если X больше или равно Y , в противном случае возвращается значение false.
X ? Y : Z	Условный выбор (синоним – тернарный оператор). Если X = true, то выбирается Y, в противном случае Z.
X is Y	Проверяет совместимость X с Y по типу. Если X может быть приведен к типу Y, не вызывая исключение, то возвращается true, в противном случае возвращается значение false.
X as Y	Возвращает X типа Y или нуль, если X не относится к типу Y,

5.7. Операторы присваивания

Они задают новое значение переменной. Присваивание бывает простое и сложное. При простом присваивании оператор состоит из одного символа (=). Синтаксис оператора:

ИмяПеременной = выражение;

Присваиваемое значение должно иметь тип, совпадающий с типом переменной, или допускающий неявное преобразование. В противном случае можно использовать явное преобразование, используя синтаксис:

ИмяПеременной = (тип переменной для размещения результата) = выражение;

Оператор сложного присваивания состоит из нескольких знаков без разделителей. Правый символ - знак простого присвоения, слева дополнительные символы, указывающие на тип дополнительной операции, выполняемой перед присваиванием.

Оператор	Действие
X = Y	X = Y
X += Y	X = X + Y
X -= Y	X = X - Y
X *= Y	X = X * Y
X /= Y	X = X / Y
X %= Y	X = X % Y
X &= Y	X = X & Y
X |= Y	X = X | Y
X ^= Y	X = X ^ Y
X <<= Y	X = X << Y

X >>= Y	X = X >> Y

5.8. Арифметическое переполнение

Арифметические операторы могут выдавать результаты, находящиеся за пределами интервала возможных величин используемого числового типа.

В случае переполнения целочисленной арифметической операции либо вызывается исключение OverflowException, либо отбрасываются старшие двоичные разряды результата.

В случае деления целого числа на ноль вызывается исключение DivideByZeroException.

Для чисел с плавающей запятой при переполнении арифметической операции или делении на ноль исключение никогда не вызывается, потому что типы чисел с плавающей запятой основаны на стандарте IEEE 754 и включают правила для представления бесконечности и нечисловых значений (NaN).

5.9. Математические операции

Для выполнения более сложных математических операций, например в тригонометрии, используется класс платформ Math. В примере используются методы Sin (вычисление синуса) и Sqrt (вычисление квадратного корня) и константа PI (системная константа с большим количеством знаков).

```
using System;
double d = Math.Sin(Math.PI/2);
double e = Math.Sqrt(144);
```

Класс System.Math включает поля и методы.

Поля класса

Вызов	Действие
Math.E	Значение свойства E примерно равно 2,718.
Math.LN10	Значение свойства LN10 примерно равно 2,302.
Math.LN2	Значение свойства LN2 примерно равно 0,693.
Math.LOG10E	Свойство LOG10E (константа) приблизительно равно 0,434.
Math.LOG2E	Значение свойства LOG2E (константа) приблизительно равно 1,442.
Math.SQRT1_2	Свойство SQRT1_2 (константа) приблизительно равно 0,707.
Math.SQRT2	Свойство SQRT2 (константа) приблизительно равно 1,414.

Math.PI	Свойство PI является константой, приблизительно равной 3,14159.

Методы класса. Имена с заглавной буквы.

Вызов	Функция
Abs(x)	Абсолютное значение
Acos(x)	Обратный косинус
Asin(x)	Обратный синус
Atan(x)	Обратный тангенс
Atan2(x,y)	4-квадрантный арктангенс. Atan2(x,y) = Atan(x/y)
BigMul(x,y)	Умножает два 32-битовых числа.
Ceiling(x)	Округление вверх
Cos(x)	Косинус
Cosh(x)	Косинус гиперболический
DivRem(x,y)	Остаток от x/y, числа целые
Exp(x)	Экспонента = e^x
Floor(x)	Округление вниз
IEEERemainder(x,y)	Остаток от x/y, числа вещественные
Log(x)	Натуральный логарифм
Log(x,y)	Логарифм от x по основанию y
Log10(x)	Логарифм от x по основанию 10
Max(x,y)	Максимальное из двух
Min(x,y)	Минимальное из двух
Pow(x,y)	Возводит x в любую степень y
Round(x)	Округление до ближайшего целого
Sign(x)	Знак числа
Sin(x)	Синус
Sinh(x)	Синус гиперболичгский
Sqrt(x)	Квадратный корень
Tan	Тангенс
Tanh	Тангенс гиперболичгский
Truncate(x)	Отсечение дробной части

5.10. Литералы

Литерал представляет собой постоянное значение, у которого нет имени. Например, 5 и "Hello World" являются литералами.

6. Типы

Язык C# является строго типизированным языком. Каждая переменная и константа имеет тип, как и каждое выражение, результатом вычисления которого является значение.

Библиотека классов платформы .NET Framework определяет типы:

- Типы значения. Это встроенные числовые типы (или примитивные типы). Они содержат данные.
- Ссылочные типы. Это сложные типы, определяемые классами, например, коллекции и массивы объектов и даты. Они содержат ссылки на объекты в куче, где размещаются данные.
- Типы указателей. Могут использоваться только в небезопасном режиме. Они могут привести к сбоям и

Существует возможность преобразовать тип значения в ссылочный тип и обратно в тип значения с помощью упаковки-преобразования и распаковки-преобразования. За исключением упакованного типа значения преобразовать ссылочный тип в тип значения невозможно.

Типичная программа C# использует типы из библиотеки классов, а также пользовательские типы, моделирующие принципы, относящиеся к проблемной области программы.

К сведениям, хранимым в типе, может относиться следующее:

- Место для хранения переменной типа.
- Максимальное и минимальное значения, которые могут быть представлены.
- Содержащиеся члены (методы, поля, события и т. д.).
- Базовый тип, которому он наследует.
- Расположение, в котором будет выделена память для переменных во время выполнения.
- Разрешенные виды операций.

Компилятор использует сведения о типе, чтобы убедиться, что все операции, выполняемые в коде, являются строго типизированными. Например, при объявлении переменной численного типа **int**, компилятор позволяет использовать переменную и операции вычитания. При попытке выполнить эти же операции с переменной логического типа **bool** компилятор вызовет ошибку.

Компилятор внедряет сведения о типе в исполняемый файл в качестве метаданных. Среда CLR использует эти метаданные во время выполнения для

дальнейшего обеспечения безопасности типа при выделении и освобождении памяти.

6.1. Классы

6.1.1. Описание

Класс является чертежом для пользовательского типа данных. Определив класс, его можно использовать, загрузив в память. Класс, загруженный в память, называется объектом или экземпляром класса. Экземпляр класса создается с помощью ключевого слова **new**.

Подобно тому, как на основе одного чертежа можно построить несколько зданий, можно создать любое количество объектов одного класса. Очень часто используются массивы или списки, содержащие множество объектов одного класса. Каждый экземпляр класса занимает отдельную область памяти; значения его полей (исключая статические поля) также являются независимыми.

Классы объявляются с помощью ключевого слова class. Класс может содержать:

- Методы.
- Свойства.
- Поля.
- События.
- Делегаты.
- Вложенные классы.

Члены класса могут иметь модификаторы:

- Ключевое слово **public** является модификатором доступа для типов и членов типов. Общий (public) доступ является уровнем доступа с максимальными правами. Ограничений доступа к общим членам не существует.
- Ключевое слово **private** является модификатором доступа к члену. Закрытый (private) доступ является уровнем доступа с минимальными правами. Доступ к закрытым членам можно получить только внутри тела класса или структуры, в которой они объявлены.
- Ключевое слово **protected** является модификатором доступа к члену. Доступ к члену с модификатором protected (защищенный) возможен внутри класса и из производных экземпляров класса.
- Модификатор **static** используется для объявления статического члена, принадлежащего собственно типу, а не конкретному объекту. Модифика-

тор static можно использовать с классами, полями, методами, свойствами, операторами, событиями и конструкторами..

- При использовании в качестве возвращаемого типа метода ключевое слово **void** обозначает, что этот метод не возвращает какого-либо значения. Ключевое слово void не может входить в список параметров метода.

6.1.2. Структуры

Структура в C# аналогична классу, однако в структурах отсутствуют некоторые возможности, например наследование. Кроме того, поскольку структура является типом значения, ее можно создать быстрее, чем класс. При наличии непрерывного цикла, где создается большое количество новых структур данных, вместо класса рекомендуется использовать структуру. Структуры используются для инкапсуляции групп полей данных, таких как координаты точки в сетке или размеры прямоугольника.

6.1.3. Инкапсуляция

Класс представляет характеристики объекта и выполняемые им действия. Действия обычно специфические для объектов. Их желательно сделать недоступными для других объектов. Помещение методов в класс «прячет» метод для других объектов Например, чтобы представить животное как класс C#, необходимо задать ему размер, скорость и силу, представленными в виде чисел, а также некоторые функции, например MoveLeft(), MoveRight(), SpeedUp(), Stop() и так далее, в которых можно написать код для выполнения "животным" этих действий. В C# класс животного может выглядеть следующим образом.

```
public class :Животное
{
   private int Размер;
   private float Скорость;
   private int Сила;

      public void MoveLeft()  // Метод сместиться влево
      {
          // Код здесь...
      }
   // Другие методы здесь...
}
```

Методы, свойства и события, а также все остальные внутренние переменные и константы (поля) называются членами класса.

Группировка членов в классы имеет не только логический смысл, она также позволяет скрывать данные и функции, которые должны быть недоступны для другого кода. Этот принцип называют инкапсуляцией. При просмотре библиотек классов платформы .NET Framework будут видны только открытые члены этих классов. Возможно, каждый класс имеет закрытые члены, используемые внутренне этим классом или связанными классами, которые не предназначены для использования приложениями. Создавая собственные классы, нужно решить, какие члены будут открытыми, а какие — закрытыми.

6.1.4. Наследование

Класс может наследовать от другого класса. Это означает, что он включает все члены — открытые и закрытые — исходного класса, а также дополнительные определяемые им члены. Исходный класс называется базовым классом, а новый класс — производным классом. Производный класс создается для представления особых возможностей базового класса.

Наследование ускоряет программирование. Для нового класса нужно программировать только дополнения, которые он имеет.

Синтаксис наследования

КлассПотомок : КлассПредок;

Например, можно определить класс Кошка, который наследует от класса Животное. Класс Кошка может выполнять все то же, что и класс Животное и дополнительно одно уникальное действие. Код C# выглядит следующим образом.

```
public class Кошка : Животное
{
    public void Дополнительный()
    {
        // Здесь команды
    }
}
```

Нотация Кошка : Животное означает, что класс Кошка наследует от класса Животное и что Кошка также имеет метод MoveLeft и три закрытые переменные: Размер, Скорость и Сила. Если затем определяется следующий потмок - класс СиамскаяКошка, который наследует от класса Кошка, то он будет содержать все члены класса Кошка, а также все члены класса Животное.

6.1.5. Полиморфиэм

Полиморфизм – возможность использования методов с одинаковым именем, но с разным содержанием.

Полиморфизмом дает возможность производного класса изменять или переопределять методы, которые он наследует от базового класса. Эта функция используется, если в методе, который имеет отличия либо не определен в базовом классе, нужно выполнить какие-то особые действия.

Перегрузкой называется создание разных методов с одним именем для разных объектов. Компилятору известно, какой метод следует использовать, поскольку во время каждого создания объекта предоставляется список аргументов (если таковые имеются). Перегрузка может сделать код более гибким и удобочитаемым.

Например, поскольку метод Животное.MoveLeft должен быть общим, чтобы подходить для всех животных, он является, возможно, очень простым, как например "изменение положения так, чтобы голова животного была в направлении X". Однако для класса Кошка этого может быть недостаточно. Может потребоваться указать, как Кошка двигает лапами и хвостом при поворотах. И если класс Рыба или класс Птица уже определен, возможно, потребуется переопределить метод MoveLeft разными способами для каждого из классов.

Поскольку можно настроить поведение метода MoveLeft для конкретного класса, в коде, создающем класс и вызывающем его методы, отсутствует отдельный метод для каждого животного. Пока объект наследует от Животное, вызывающий код может вызывать лишь метод MoveLeft и собственную версию метода объекта.

6.1.6. Конструкторы

В каждом классе существует конструктор — метод с тем же именем, что и у класса. Конструктор вызывается при создании объекта на базе определения класса. Обычно конструкторы задают начальные значения переменных, определенных в классе. Конструкторы не используются, если начальным значением для числовых типов данных будет ноль, "false" для логических типов данных или null для ссылочных типов, поскольку эти типы данных инициализируются автоматически.

6.1.7. Деструкторы

Если вы работали с C++, то, возможно, вы уже располагаете сведениями о деструкторах. В связи с наличием в C# системы автоматического сбора мусора

маловероятно, что вам когда-либо придется применять деструктор, если только класс не использует неуправляемые ресурсы.

В C# допускается только одиночное наследование. Другими словами, класс может наследовать реализацию только от одного базового класса. Однако класс может реализовать несколько интерфейсов. Через это реализуется множественное наследование.

6.2. Интерфейсы

Через него реализуется множественное наследование. Интерфейс содержит:

- Методы
- Свойства
- Индексаторы
- События

Интерфейс способен наследовать от одного или нескольких базовых интерфейсов.

Интерфейсы описывают группу связанных функциональных возможностей, которые могут принадлежать к любому классу или структуре. Интерфейсы могут содержать методы, свойства, события, индексаторы или любое сочетание этих перечисленных типов членов. Интерфейсы не могут содержать поля. Члены интерфейсов автоматически являются открытыми.

Синтаксис интерфейса:

```
interface Имя
{
    ПодписьМетода;
    ПодписьСвойства;
}
```

Интерфейс может наследоваться классом или структурой. Когда говорят, что класс или структура наследует интерфейс, это означает, что класс или структура предоставляет реализацию для всех членов, определяемых интерфейсом. Сам интерфейс не предоставляет функциональных возможностей, которые класс или структура могут наследовать таким же образом, каким могут наследоваться функциональные возможности базового класса. Однако если базовый класс реализует интерфейс, производный класс наследует эту реализацию.

Классы и структуры могут быть унаследованы от интерфейсов таким же образом, как классы могут быть унаследованы от базового класса или структуры, но есть два исключения:

- Класс или структура может наследовать несколько интерфейсов.
- Когда класс или структура наследует интерфейс, наследуются только имена и подписи методов, поскольку сам интерфейс не содержит реализаций.

Интерфейс используется для создания класса, в котором применяются методы разных классов. **Множественное наследование е C# запрещено**. Интерфейс задает контракт, определяющий отношение типа "может" или "имеет".

У интерфейсов могут быть свойства, методы и события, являющиеся абстрактными членами. Это значит, что хотя в интерфейсе и объявляются члены и их сигнатуры, за определение функций этих членов отвечает тип, реализующий данный интерфейс. Любой класс или структура, реализующие интерфейс, должны содержать определения абстрактных членов, объявленных в этом интерфейсе. Интерфейс может потребовать реализации одного или нескольких других интерфейсов в любом реализующем классе или структуре.

К интерфейсам применяются следующие ограничения.

- Интерфейс может быть объявлен с любой доступностью, однако, члены интерфейса всегда должны иметь доступность уровня **public**.
- К членам интерфейса или к самому интерфейсу не могут быть присоединены никакие разрешения безопасности.
- В интерфейсах нельзя определять конструкторы.
- В интерфейсах нельзя определять поля.
- Все абстрактные свойства, методы и события, определенные в интерфейсе, должны быть членами экземпляра; они не могут быть статическими членами.
- Поскольку член с одинаковой подписью может быть объявлен в нескольких интерфейсах, и эти члены могут иметь отдельные реализации, каждый язык должен предоставлять правила сопоставления реализации и интерфейса, для которого необходим данный член.

6.3. Делегаты

Делегат — это тип, который определяет подпись метода и его можно связать с любым методом с совместимой подписью. Метод можно запустить (или вызвать) с помощью делегата. Делегаты используются для передачи методов в качестве аргументов к другим методам. Обработчики событий - это методы, вызываемые с помощью делегатов.

Объявление типа делегата аналогично сигнатуре метода. Оно имеет возвращаемое значение и некоторое число параметров какого-либо типа. Делегат используется для объявления ссылочного типа, который может быть исполь-

зован для инкапсуляции именованного или анонимного метода. Делегаты аналогичны используемым в языке C++ указателям на функции, но являются типобезопасными и безопасными.

Создание и использование делегатов. Во многих случаях, таких как методы обратного вызова, делегат представляет только один метод, и единственное, что нужно сделать — создать и вызвать делегат.

Делегат определяется так

public delegate void Имя(ИмяОбъекта);

Так как в C# все является классом, то и делегат в момент компиляции превращается в класс, наследуемый от system.Multicastdelegate.

6.4. Типы значений

Переменные, основанные на типах значений, содержат непосредственно значения. При присвоении переменной одного типа значений другому создается копия присваиваемого значения. В этом заключается отличие от переменных ссылочного типа, при присвоении которых копируются ссылки на объекты, но не сами объекты.

Все типы значений являются неявными производными от System .ValueType .

Типы значений состоят из двух основных категорий:

- Структура **struct**. Это тип значения, который используется для инкапсуляции небольших групп связанных переменных, например координат точки.
- Перечисление **enum**. Оно состоит из набора именованных констант, который называется списком перечислителя. По умолчанию первому перечислителю задан номер 0, а номер каждого последующего увеличивается на 1. Пример: enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};

Структуры делятся на следующие категории:

- Числовые типы. Они могут быть целочисленные, вещественные - с плавающей запятой, десятичные.
- Логический **bool.** Ключевое слово bool является псевдонимом свойства System .Boolean . Оно используется для объявления переменных для хранения логических значений, true и false.
- Пользовательские структуры.

6.5. Ссылочные типы

Переменные ссылочных типов, называемые объектами, сохраняют ссылки на фактические данные. В данном разделе описаны следующие ключевые слова, используемые для объявления ссылочных типов:

- Класс **class**.
- Интерфейс **interface.**
- Делегат **delegate**.
- Динамический **dynamic**.
- Объект **object**.
- Строка **string**.

6.6. Тип dynamic

Тип **dynamic** позволяет пропускать проверки типов во время компиляции операции, в которых он применяется . Вместо этого эти операции разрешаются во время выполнения. Тип упрощает доступ к API автоматизации Office и к динамическим API.

6.7. Тип object

Тип **object** представляет собой псевдоним для Object в платформе .NET Framework. В унифицированной системе типов C# все типы, предопределенные и пользовательские, ссылочные типы и типы значений, наследуют непосредственно или косвенно от Object. Переменным типа object можно назначать значения любых типов. Когда переменная типа значения преобразуется в объект, говорят, что она упаковывается. Когда переменная типа object преобразуется в тип значения, говорят, что она распаковывается.

6.8. Тип string

Строка **string**. Тип string представляет последовательность из нуля или более символов в кодировке Юникод. string – это псевдоним для String в платформе .NET Framework.

Строка является объектом типа **string**, значением которого является текст. Тип данных string (все буквы строчные) является псевдонимом класса **String**. Внутренне объект типа string хранится в коллекции объектов Char, каждый из которых представляет один символ Юникода в UTF-16.

Строка C# представляет собой группу одного или нескольких знаков, заключенных в **двойные кавычки** и объявленных с помощью ключевого слова string,

string Приветствие = "Hello, World!";

Строковые объекты являются неизменяемыми, после создания их нельзя изменить.

6.9. Встроенные базовые типы

C# имеет встроенные типы для представления значений целых чисел, вещественных чисел, логических выражений, текстовых символов, десятичных значений и других данных.

В таблице перечислены базовые типы, предоставляемые в .NET Framework, кратко описывается каждый тип и указывается соответствующий тип в C#.

Описание	.Net	C#
8-разрядное целое число без знака.	Byte	byte
8-разрядное целое число со знаком.	SByte	sbyte
16-разрядное целое число со знаком	Int16	short
32-разрядное целое число со знаком	Int32	int
64-разрядное целое число со знаком	Int64	long
16-разрядное целое число без знака	UInt16	ushort
32-разрядное целое число без знака	UInt32	uint
64-разрядное целое число без знака	UInt64	ulong
32-разрядное с плавающей точкой с обычной точностью	Single	float
64-разрядное с плавающей точкой с двойной точностью	Double	double
Логическое значение (true или false)	Boolean	bool
Символ Юникода (16-разрядный)	Char	char
128-разрядное десятичное целое	Decimal	decimal
Корень иерархии объектов	Object	object
Строка символов Юникода фиксированной длины	String	string

В дополнение к базовым типам данных пространство имен System содержит более 100 классов — от классов для обработки исключений до классов, которые работают с основными механизмами среды выполнения, такими как домены приложений и сборщик мусора.

6.10. Типы чисел

6.10.1. Типы целых чисел

В таблице представлены размеры и диапазоны целых типов, которые составляют подмножество простых типов.

Тип	Диапазон	Размер
sbyte	-128 … 127	8-разрядное целое число со знаком

byte	0 … 255	8-разрядное целое число без знака
char	U+0000 … U+ffff	16-разрядный символ Юникода
short	-32 768 … 32 767	16-разрядное целое число со знаком
ushort	0 … 65 535	16-разрядное целое число без знака
int	-2 147 483 648 … 2 147 483 647	32-разрядное целое число со знаком
uint	0 … 4 294 967 295	32-разрядное целое число без знака
long	-9 223 372 036 854 775 808 … 9 223 372 036 854 775 807	64-разрядное целое число со знаком
ulong	0 … 18 446 744 073 709 551 615	64-разрядное целое число без знака
decimal	$-7{,}9*10^{28}$ … $7{,}9*10^{28}$ Для финансовых расчетов	128-разрядное целое число со знаком

6.10.2. Типы чисел с плавающей запятой

Стандарт IEEE 754-1985 определяет

- Представление нормализованных положительные и отрицательные числа с плавающей точкой.
- Представление нормализованных положительные и отрицательные числа с плавающей точкой.
- Представление нулевых чисел,
- Представление специальной величины - бесконечность (Infiniti),
- Представление специальной величины "Не число" (NaN – No a Number),

IEEE 753-1985 определяет 4 формата представления чисел с плавающей запятой:

- с одинарной точностью (single-precision) 32 бита,
- с двойной точностью (double-precision) 64 бита,
- с одинарной расширенной точностью (single-extended precision) >=43 бит (редко используемый),
- с двойной расширенной точностью (double-extended precision) >= 79 бит (обычно используют 80 бит).

Основное применение в технике и программирование получили форматы 32 и 64 бита. Например, в C# используют типы данных single и double.

Основные понятия в представлении чисел с плавающей точкой.

Возьмем, к примеру, десятичное число 155.625. Представим это число в нормализованном экспоненциальном виде: $1.55625*10^2 = 1.55625e+2$.

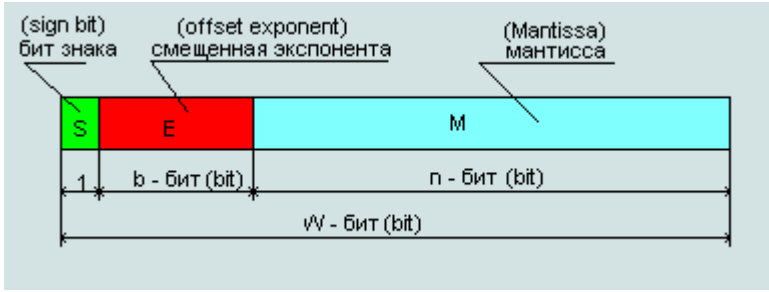
Число 1.55625e+2 состоит из двух частей: мантиссы M=1.55625 и экспоненты e =+2.

Нормализованные числа Если мантисса находится в диапазоне 1<=M<2 то число считается нормализованным.

Денормализованные числа. Это числа, мантиссы которых лежат в диапазоне 0.1 <= M <1 Денормализованные числа находятся ближе к нулю, чем нормализованные. Денормализованные числа как бы разбивают минимальный разряд нормализованного числа на некоторое подмножество. Сделано так потому, что в технической практике чаще встречаются величины близкие к нулю.

Экспонента представлена основанием системы исчисления (в данном случае 10) и порядком (в данном случае +2). Порядок экспоненты может иметь отрицательное значение, например число 0,0155625=1,55625e-2.

Формальное представление нормализованных чисел в формате IEEE 754.



- S - бит знака числа (0 - положительное число; 1 - отрицательное число).
- E - смещенная экспонента (смещение для устранения знака порядка).
  Порядок = $E - 2^{(b-1)} + 1$, где
  E- экспонента двоичного нормализованного числа с плавающей точкой,
  $2^{(b-1)}-1$ - смещение экспоненты.
- M - остаток мантиссы двоичного нормализованного числа с плавающей точкой (Первый бит мантиссы всегда 1, поэтому он скрывается).

Формула вычисления десятичных чисел с плавающей точкой из чисел, представленных в стандарте IEEE754:

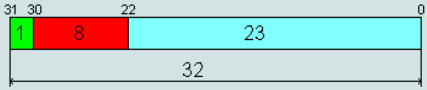
Формула расчета нормализованных чисел:

$$F = (-1)^S \ 2^{(E-2^{(b-1)}+1)}(1+M/2^n)$$

Формула расчета денормализованных чисел:

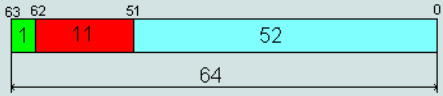
$$F = (-1)^S \ 2^{(E-2^{(b-1)}+2)} M/2^n$$

Формат числа одинарной точности (single-precision) 32 бита



$$F = (-1)^S 2^{(E-127)}(1+M/2^{23})$$

Формат числа двойной точности (double-precision) 64 бита



$$F = (-1)^S \ 2^{(E-1023)}(1+M/2^{52})$$

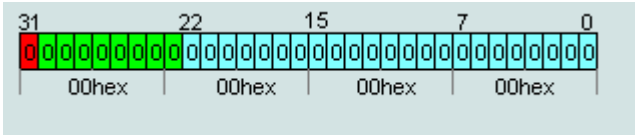
Исключения чисел формата IEEE754. Если применить формулу нормализованных чисел для вычисления минимального и максимального числа в формате single, представленного в IEEE754, то получим следующие результаты:

- 00 00 00 00 hex= 5,87747175411144e-39 (минимальное положительное).
- 80 00 00 00 hex=-5,87747175411144e-39 (минимальное отрицательное).
- 7f ff ff ff hex= 6,80564693277058e+38 (максимальное положительное).
- ff ff ff ff hex=-6,80564693277058e+38 (максимальное отрицательное).

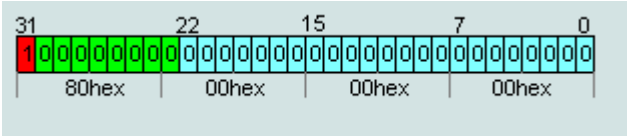
Отсюда видно, что **невозможно** представить число нуль в заданном формате.

Поэтому из стандарта сделаны исключения, и формула не применяется в следующих случаях:

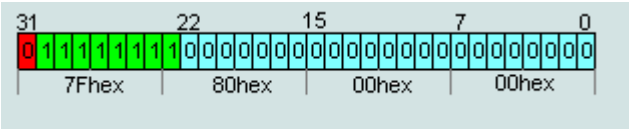
- Число IEEE754=00 00 00 00hex считается числом +0.



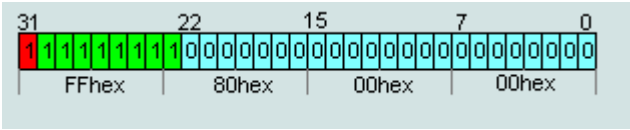
- Число IEEE754=80 00 00 00hex считается числом -0.



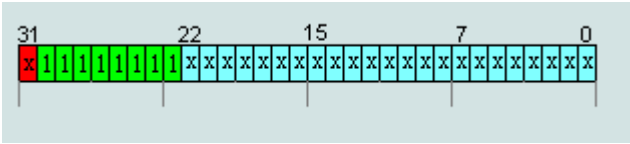
- Числа, которые выходят за границы диапазона представления чисел считаются бесконечными.
  Число IEEE754=7F 80 00 00hex считается числом +∞.



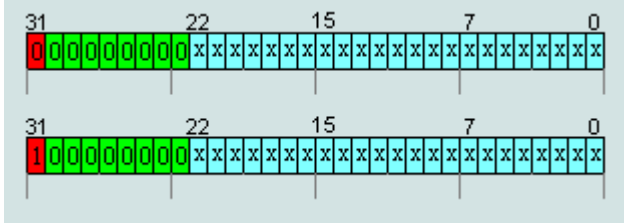
  Число IEEE754=FF 80 00 00hex считается числом -∞.



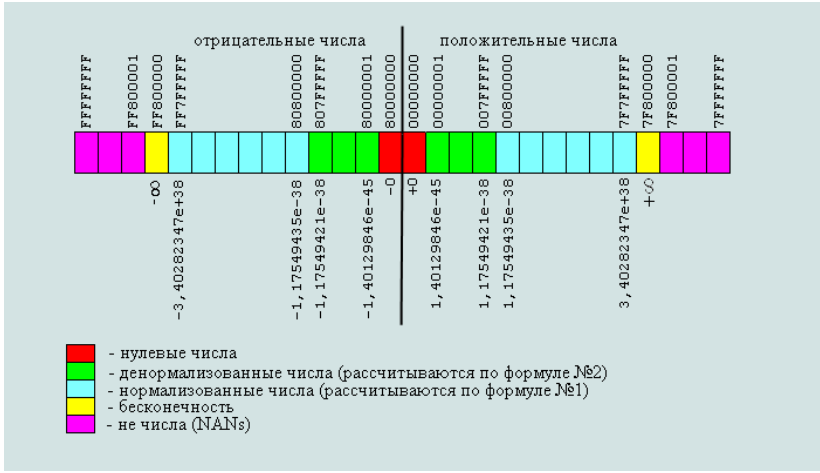
- NAN (No a Number) – не число.
  Числа IEEE754=FF (1xxx)X XX XXhex не считается числами, кроме +∞.
  Числа IEEE754=7F (1xxx)X XX XXhex не считается числами, кроме -∞.
  Числа, представленные в битах с 0...22, могут быть любыми кроме 0.



- Числа IEEE754=(x000) (0000) (0xxx)X XX XXhex считаются денормализованными числами, за исключением чисел -0 и +0



Полный диапазон чисел в формате 32 бит по стандарту IEEE754:



В таблице представлены приблизительные диапазоны типов с плавающей запятой.

Тип	Диапазон	Точность
float	±1,5e−45 … ±3,4e38	7 знаков
double	±5,0e−324 … ±1,7e308	15 -16 знаков

6.10.3. Значения типов по умолчанию

В приведенной ниже таблице показаны значения по умолчанию типов значений, возвращаемые конструкторами по умолчанию.

Помните, что в C# не допускается использование неинициализированных переменных.

Тип	Значение по умолчанию
bool	false
byte	0
char	'\0' – код с номером 0
decimal	0,0M
double	0,0D
float	0,0F
int	0
long	0L
sbyte	0
short	0
uint	0
ulong	0
ushort	0

6.10.4. Преобразования типов

Все вычисления происходят с использованием типа double. Другие типы чисел могут применяться для уменьшения занимаемой памяти. При их использовании перед вычислением они преобразуются в тип double. Различают преобразования:

- Неявные преобразования используются для совместимых типов. Значения источника полностью отображаются приемником. Например, при преобразовании int в double (int – подмножество double). Такое преобразование выполняется автоматически, его не надо заказывать.
- Явные преобразования используются для несовместимых типов. Например, при преобразовании double в int (int – подмножество double). Значения источника не полностью отображаются приемником. Если типы несовместимы, но ошибка допустима, то преобразование возможно, но его нужно явно заказать: для этого перед преобразуемым выражением добавляется префикс идентификации конечного типа (в круглых скобках).

Например,

```
double db=12.94;
int i = (int) db;
```

6.10.5. Стандартное форматирование чисел

Все числа выводятся в консоль в виде строк символов. Потому перед выводом применяется форматирование.

Все числовые типы, типы даты и времени, а также перечисления в составе платформы .NET Framework поддерживают предопределенный набор описателей формата. Для чисел поддерживаются национальные стандарты. Например, для России разделитель целой и дробной части числа - запятая.

Строки формата также можно использовать для определения разнообразных строковых представлений прикладных пользовательских типов данных.

Метод вывода в консоль содержит в двойных кавычках строку вывода, в которой содержатся выводимые символы и описатели форматов вывода не строковых данных, вставляемые в строку в нужных местах. После строки через запятые перечисляются имена выводимых данных. Количество описателей формата и не строковых данных одинаково, нумерация с нуля.

Описатель формата помещается в фигурные скобки. Описатель формата имеет структуру:

{<Номер вывода>,<Число позиций>:<Буква>d}

Если указывается только номер вывода, то выводимые нестроковые данные форматируются по умолчанию

Пример 1, в котором выводится значение x (номер вывода 0), под число отводится 8 позиций, используется денежный формат (буква C).

Console.WriteLine("{0,8:C}", x);

Пример 2. Выводятся значения x, y (номера вывода 0 и 1), под числа отводится 8 позиций, используется денежный формат (буква C).

Console.WriteLine("{0,8:C} {1,8:C}", x, y);

Пример 3. Выводятся значения x, y (номера вывода 0 и 1), формат по умолчанию.

Console.WriteLine("x = {0} y = {1}", x, y);

Имеется набор стандартных форматов. Синтаксис записи формата:

<Буква описания формата>d.

Описатель формата - это алфавитный символ, определяющий строковое представление объекта, к которому он применяется. Также строка формата может содержать необязательный описатель точности d, определяющий, сколько цифр отображается в результирующей строке. Если спецификаторы пропускаются, то используются их значения по умолчанию. Предусмотрены форматы.

Буква	Формат	Примеры	d по

			умолч.
G g	Общий	-123.456 -> -123.456	Факт
F f	Фиксированная запятая	1234.567 -> 1234.57	2
N n	Число	1234.567 -> 1,234.57 Запятая разделяет группы	2
E e	Экспоненциальный (научный)	1052.0329112756 -> 1.052033E+003	6
D d	Десятичный	-1234 ("D6") -> -001234	Мин.
C c	Валюта	123.456 -> $123.46 Используются символы валют	2
P p	Проценты	1 -> 100.00 %	2
R r	Приемо-передача		
X x	16 - ричный	255 ("X") -> FF -1 ("x") -> ff	
Другая	Неизвестный описатель		

Описание форматов:

- Формат G – общий. Результат - наиболее компактная запись из двух вариантов: экспоненциального и с фиксированной запятой. Поддерживается всеми числовыми типами данных. Описатель точности - количество значащих цифр. Описатель точности по умолчанию определяется численным типом.
- Формат G - фиксированная запятая. Результат - цифры целой и дробной частей с необязательным отрицательным знаком. Поддерживается всеми числовыми типами данных. Описатель точности - количество цифр дробной части.
- Формат N – число. Результат - цифры целой и дробной частей, разделители групп и разделитель целой и дробной частей с необязательным отрицательным знаком. Поддерживается всеми числовыми типами данных. Описатель точности - желаемое число знаков дробной части.
- Формат E - экспоненциальный (научный). Результат - экспоненциальная нотация. Поддерживается всеми числовыми типами данных. Описатель точности =-количество цифр дробной части. Описатель точности по умолчанию 6.

- Формат D – десятичный. Результат - целочисленные цифры с необязательным отрицательным знаком. Поддерживается только целочисленными типами данных. Описатель точности - минимальное число цифр. Описатель точности по умолчанию - минимальное необходимое число цифр.
- Формат C – валюта. Результат - значение валюты. Если есть символ то он выводится. Поддерживается всеми числовыми типами данных. Описатель точности - количество цифр дробной части.
- Формат P – проценты. Результат - число, умноженное на 100 и отображаемое с символом процента. Поддерживается всеми числовыми типами данных. Описатель точности - желаемое число знаков дробной части.
- Формат R – прием-передача. Результат - строка, дающая при обратном преобразовании идентичное число. Поддерживается Single, Double.
- Формат X – шестнадцатеричный. Результат - шестнадцатеричная строка. Поддерживается только целочисленными типами данных. Описатель точности - число цифр в результирующей строке.

6.10.6. Нестандартное форматирование чисел

В нестандартных форматных строках число форматируется по шаблону, состоящему из специальных символов, который помещается в описатель формата, заключаемый в фигурные скобки. Число символов определяет число позиций для вывода.

- Символ ноль 0. Обозначает позицию, в которой выводится цифра или 0. Если в разряде числа, помеченном символом 0, находится цифра, то она включается в результат, заменяя 0.
- Символ #. Обозначает позицию, в которой выводится цифра или пробел. Если в разряде числа, помеченном этим символом, находится цифра, то она включается в результат, в противном случае в этой позиции будет пробел.
- Символ точка. Определяет позицию десятичного разделителя целой и дробной части числа. Возможно использование в этом статусе символа запятая.
- Символ запятая. Используется для разделения групп разрядов. В этой позиции число перед форматированием делится на 1000.
- Символ % (процент). Означает, что число должно выводиться в процентах. В этой позиции число перед форматированием делится на 100.
- Символы E, e. Используются для форматирования с научной (экспоненциальной) записью. Если после символов 0 или # сразу следуют символы E+0, e+0, то число отображается в научном формате.

- Символ точка с запятой. Используется для разделения секций с положительными, нулевыми и отрицательными числами. Первые две секции могут объединяться. В каждой секции задается свой формат
- Символ \ (обратная косая черта). Ставится перед символами, которые не подвергаются форматированию.

Если целая часть не умещается в заданных позициях, то осуществляется расширение целой части. Точность вывода целой части гарантируется.

Если дробная часть не умещается в заданных позициях, то в форматах G и R осуществляется расширение дробной части. В остальных используется ее округление.

В денежном формате C добавляется символ р. Если число целое, то дробные разряды заполняются нулями. Целая часть отображается с разделением на денежные группы по три символа. Если d пропускается, по умолчанию d=2.

В процентном формате P число умножается на 100, добавляется символ %. Целая часть отображается с разделением на денежные группы по три символа.

6.11. Тип char - символы

Тип char – это один символ в коде unicod UTF-16. Символ помещается в **одиночные кавычки**. Синтаксис объявления:

char Имя = ‘ Символ ’;

Символ может быть:
- Буква кодировки ANSI.
- Escape-знак. Применяется для ввода управляющих символов кодировки ANSI. Например, \n (новая строка), \t (табуляция).
- Символ unicod - \u +xxxx, где xxxx – 16 ричные символы.
  char МойСимвол = ‘A ‘; // символ A
  char ЯпонСимвол = ‘\u30ad ‘; // символ из японской азбуки Катакана.

Определены следующие Escape-последовательности строк

Escape-посл.	Имя символа	Кодировка Юникода
\'	Одинарная кавычка	0x0027
\"	Двойная кавычка	0x0022
\\	Обратная косая черта	0x005C
\0	Null	0x0000
\a	ALERT	0x0007

\b	BACKSPACE	0x0008
\n	Новая строка	0x000A
\r	Возврат каретки	0x000D
\t	Горизонтальная табуляция	0x0009
\u	Escape-последовательность unicod	\u0041 = ‘A’
\v	Вертикальная табуляция	0x000B
\x	Escape-последовательность unicod аналогична "\u", за исключением строк с переменной длиной.	\x0041 = ‘A’

6.12. Тип enum - перечисление

Перечисление состоит из набора именованных констант, который называется списком перечислителя. По умолчанию первому перечислителю задан номер 0, а номер каждого последующего увеличивается на 1.

Пример: enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};

6.13. Тип DateTime

Поддерживает представление календарного времени, как совокупность даты и времени суток.

Значения измеряются в 100-наносекундных единицах, называемых тактами, и точная дата представляется числом тактов с 00:00 1 января 0001 года н. э. по календарю GregorianCalendar. Например, значение тактов, равное 31241376000000000L, представляет пятницу 1 января 0100 года 00:00:00. Значение DateTime всегда выражается в контексте явно определенного или заданного по умолчанию календаря.

Можно создать новый объект DateTime, используя один из способов:

- Путем вызова конструктора DateTime, который позволяет указать определенные элементы значения даты и времени. В следующей инструкции показан вызов конструктора DateTime для создания даты с определеными годом, месяцем, днем, часом, минутой и секундой.
  DateTime date1 = new DateTime(2008, 5, 1, 8, 30, 52);
- Создать пустой объект DateTime и затем присвоить значения его свойствам.

Внешний вид значения DateTime — это результат операции форматирования. Форматирование — это процесс преобразования значения в его строковое представление.

Внешний вид значений даты и времени зависит от таких факторов, как язык и региональные параметры, международные стандарты, программные требования и личные предпочтения.

Структура DateTime обеспечивает большую гибкость при форматировании значений даты и времени с помощью метода ToString(). Он по умолчанию возвращает строковое представление значений даты и времени, используя формат краткой записи даты и длинной записи времени, предусмотренный в языке и региональных параметрах.

Структура DateTime содержит 64-битовое поле, состоящее из закрытого поля Kind, сцепленного с полем Ticks. Поле Ticks содержит число тактов. Поле Kind является 2-битовым полем, указывающим, какое время представляет структура DateTime: местное, скоординированное всеобщее (UTC) или время в незаданном часовом поясе. Поле Kind используется при выполнении преобразования значения времени между часовыми поясами.

6.13.1. Свойства

Для объекта DateTime определены свойства:

Свойство	Описание
Date	Дата, Время = 0:00:00
Day	День месяца
DayOfWeek	День недели (имя)
DayOfYear	День года
Hour	Час суток
Kind	Стандарт представления
Minute	Минуты
Month	Месяц (номер)
Now	Местное время
Second	Секунды
Tics	Число тактов в дате
TimeOfDay	Время дня
Today	Текущая дата
UtcNow	Дата + Время по Гринвичу
Year	Год
MaxValue	Максимальное время, только для чтения
MaxValue	Минимальное время, только для чтения

6.13.2. Методы

Для объекта DateTime определены методы. Основные из них:

Метод	Описание
Add(TimeSpan)	Добавить интервал TimeSpan
AddDays(double)	Добавить дни
AddHours(double)	Добавить часы
AddMilliseconds(double)	Добавить миллисекунды
AddMinutes(double)	Добавить минуты
AddMounths(int)	Добавить месяцы
AddSecondes(double)	Добавить секунды
AddTics(long)	Добавить такты
AddYears(int)	Добавить годы
Compare(DateTime, DateTime)	Сравнить время
DateTime(y,m,d,h,mn,s,ms)	Задать (г,мес,день,час,мин,сек,мсек)
DateTime.GetDateTimeFormats()	Преобразовать в формат
Subtract(TimeSpan)	Вычесть интервал TimeSpan
Subtract(DateTime)	Вычесть аргумент
ToString()	Преобразует в строку
ToLongDateString()	Длинная дата (месяц - слово)
ToShortDateString()	Короткая дата, без времени
ToLongTimeString()	Длинное время, полное
ToShortTimeString()	Короткое время, без секунд

6.13.3. Пример

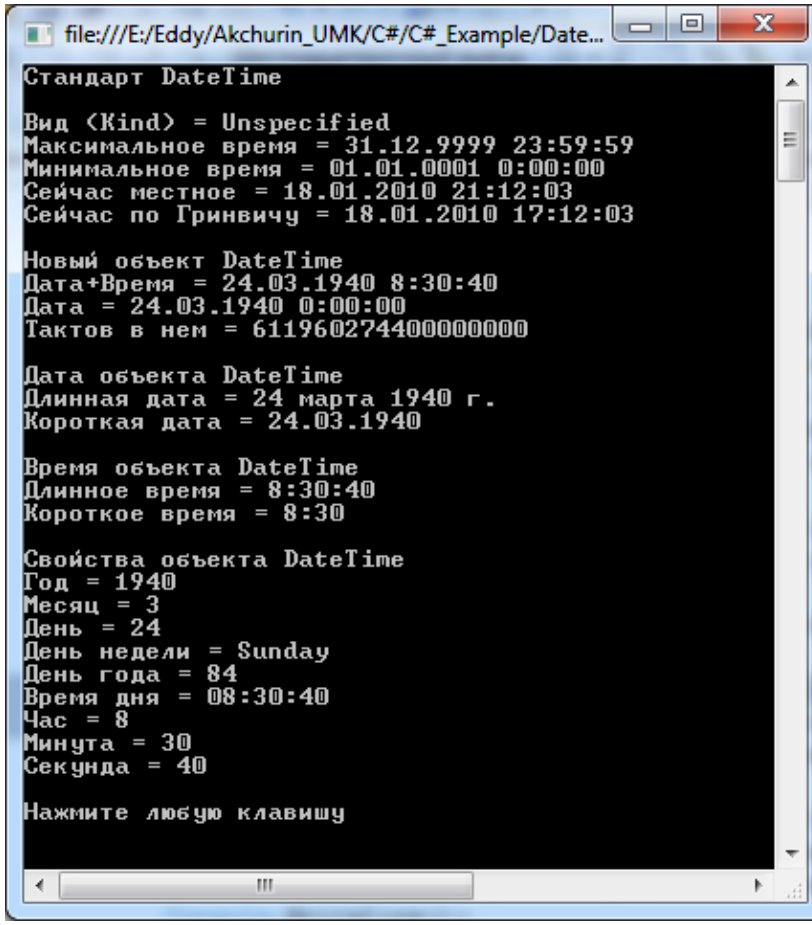
В примере создаются два объекта DateTime - date1, date2. В первом задается произвольная дата (например, момент рождения). Во втором фиксируется текущий момент.

Затем осуществляются операции с использованием свойств и методов этих объектов.

```
using System;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            DateTime date1 = new DateTime(1940,3,24,8,30,40);
            DateTime date2 = DateTime.Now;
            Console.WriteLine("Стандарт DateTime");
            Console.WriteLine();
            Console.WriteLine("Вид (Kind) = " + date1.Kind);
```

```
            Console.WriteLine("Максимальное время = " + DateTime.MaxValue);
            Console.WriteLine("Минимальное время = " + DateTime.MinValue);
            Console.WriteLine("Сейчас местное = " + date2);
            Console.WriteLine("Сейчас по Гринвичу = " + DateTime.UtcNow);
            Console.WriteLine();
            Console.WriteLine("Новый объект DateTime");
            Console.WriteLine("Дата+Время = " + date1);
            Console.WriteLine("Дата = "+ date1.Date);
            Console.WriteLine("Тактов в нем = " + date1.Ticks);
            Console.WriteLine();
            Console.WriteLine("Дата объекта DateTime");
            Console.WriteLine("Длинная дата = " + date1.ToLongDateString());
            Console.WriteLine("Короткая дата = " + date1.ToShortDateString());
            Console.WriteLine();
            Console.WriteLine("Время объекта DateTime");
            Console.WriteLine("Длинное время = " + date1.ToLongTimeString());
            Console.WriteLine("Короткое время = "+ date1.ToShortTimeString());
            Console.WriteLine();
            Console.WriteLine("Свойства объекта DateTime");
            Console.WriteLine("Год = "+date1.Year);
            Console.WriteLine("Месяц = "+date1.Month);
            Console.WriteLine("День = "+date1.Day);
            Console.WriteLine("День недели = " + date1.DayOfWeek);
            Console.WriteLine("День года = " + date1.DayOfYear);
            Console.WriteLine("Время дня = " + date1.TimeOfDay);
            Console.WriteLine("Час = " + date1.Hour);
            Console.WriteLine("Минута = " + date1.Minute);
            Console.WriteLine("Секунда = " + date1.Second);
            Console.WriteLine();
            Console.WriteLine("Нажмите любую клавишу");
            Console.ReadKey();
        }
    }
}
```

При прогоне получаем:



Фрагменты консоли по заголовкам:

- Стандарт DateTime. Максимальное и минимальное значения. Местные данные. Данные по Гринвичу.
- Новый объект DateTime. Дата + Время. Только дата. Число тактов.
- Дата объекта DateTime. Длинная и короткая даты.
- Время объекта DateTime. Длинное и короткое время.
- Свойства объекта DateTime. Год, месяц, день, день недели, день года, только время дня, час, минута, секунда.

6.14. Задание типов в объявлениях переменных

При объявлении переменной или константы в программе необходимо задать ее тип. Синтаксис объявления:

тип Имя = Значение;

В следующем примере показаны некоторые объявления переменных

```
float Температура = 40;            // переменная Температура типа float
string Имя = “ЭтоЯ” ;              // переменная Имя типа string
char ПерваяБуква = 'C';            // символ
int[] Массив = { 0, 1, 2, 3, 4, 5 }; // массив
```

После объявления переменной она не может быть повторно объявлена с новым типом, и ей нельзя присвоить значение, несовместимое с ее объявленным типом. Например, нельзя объявить переменную типа int и затем присвоить ему логическое значение true.

Однако значения могут быть преобразованы в другие типы, например, при их присвоении новым переменным или при передаче в качестве аргументов метода. Преобразование типов, которое не приводит к потере данных, автоматически выполняется компилятором. Оно называется **неявным**. Для преобразования, которое может привести к потере данных, необходимо в исходном коде указать тип преобразования. Такое преобразование называется **явным**. Оно может привести к ошибке.

7. Инструкции, введение

Код приложений в C# состоит из инструкций (команд) с ключевыми словами и выражениями с операторами.

В инструкцию включают объявление переменных, присвоение значений, вызов методов, ветвление на один или другой блок кода, в зависимости от заданного условия. Порядок выполнения инструкций в программе называется потоком управления или потоком выполнения.

Инструкция - строка кода, которая **заканчивается точкой с запятой**.

Блочная инструкция - набор инструкций **в фигурные скобки { }**.Может содержать вложенные блоки.

Пример. В следующем коде показаны примеры однострочных инструкций и блок многострочной инструкции:

static void Main()

```
{
    Int[]  МассивРадиусы = {1 2 3 4 5};     // Массив радиусов
    foreach (int Номер in МассивРадиусы)
        {
            double ДлинаОкружности = 2*Math. PI * МассивРадиусы);
            Номер++;
        }
}
```

7.1. Выражения

Выражение — это строка кода, которая определяет значение. Выражение включают в состав инструкции. Пример инструкции с простым выражением:

```
моеЗначение = 100;
```

Данная инструкция выполняет присвоение значения 100 переменной моеЗначение. Поскольку моеЗначение = 100 - это выражение, оно может использоваться и в другой инструкции присвоения. Например:

```
моеВтороеЗначение = моеЗначение = 100;
```

Инструкцией такого вида вы можете инициализировать любое число переменных с одним и тем же значением, например 20:

```
a = b = c = d = e = 20;
```

Инструкция объявления представляет новую переменную или константу. Объявление переменной может и присвоить значение переменной. В объявлении константы необходимо назначение значения.

```
double Площадь = 0;              // Переменная с инициализацией
double Радиус = 2;
const double pi = 3.14159;       // Константа pi с конечным числом символов.
```

Инструкция с выражением вычисляет значение выражения, которое сохраняется в указанной переменной.

```
Площадь = PI * (Радиус * Радиус);
```

7.2. Разделители

В языке C# как разделители рассматриваются:

- пробелы,
- знаки табуляции,
- переход на новую строку.

В инструкциях языка C# лишние разделители компилятором игнорируются. Таким образом, вы можете написать:

```
myValue=100;
myValue  =  100;
```

Компилятор обработает эти две инструкции как абсолютно идентичные. Исключение состоит в том, что пробелы в пределах строки не игнорируются. Если вы напишете:

```
Console.WriteLine("Я изучаю C# !");
```

то каждый пробел между словами «Я», «изучаю», «C#» и знаком «!» будет обрабатываться, как отдельный символ строки.

Разделители применяются для того, чтобы сделать программу более удобной для программиста, для компилятора разделители абсолютно безразличны.

Надо заметить, что есть случаи, в которых использование пробелов является весьма существенным. Например, при объявлении имени переменной. Компилятор знает, что пробел с обеих сторон оператора присвоения игнорируется (сколько бы много их не было), но пробел между объявлением типа int и именем переменной myVaiue должен быть обязательно, иначе для компилятора получится ложный идентификатор

Разделители в тексте программы позволяют компилятору находить и анализировать ключевые слова языка.

8. Решения и ветвления

8.1. Безусловный переход вызовом функций

Когда компилятор находит в основном тексте программы имя функции, то происходит приостановка выполнения текущего кода программы и осуществляется переход к найденной функции. Когда функция выполнится и завершит свою работу, то произойдет возврат в основной код программы, на ту инструкцию, которая следует за именем функции.

Имя функции должно содержать пару круглых скобок (), даже если у функции нет аргументов. Это признак функции или метода.

8.2. Ветвление if; else

Применяется для ветвления по двум ветвям. Синтаксис инструкции:

```
if (условие)
    {
        Блок инструкций 1;
    }
else
    {
        Блок инструкций 2;
    }
```

Фраза else может отсутствовать.

Если условие выполняется, то исполняется Блок инструкций 1, в противном случае исполняется Блок инструкций 2. Если в блоке только одна инструкция, то фигурные скобки можно пропустить.

Пример. В нем сравниваются значения One, Two. В зависимости от результата выводится одно из двух сообщений.

```
using System;
    class Conditional
    {
        static void Main()
        {
            int One = 50;
            int Two = 5;
            if ( One > Two )
                Console.WriteLine ("One: {0} больше Two: {1}", One, Two);
            else
                Console.WriteLine("One: {0} не больше Two:  {1}" , One, Two);
```

```
        }
    }
```

8.3. Вложенные ветвления if; else

Применяются для множественного ветвления. Синтаксис инструкции:

```
if (условие_1)
    {
        Блок инструкций 1;
    }
else
    if (условие_2)
        {
            Блок инструкций 2;
        }
    else
        {
            Блок инструкций 3;
        }
```

Фраза else может отсутствовать.

Если условие_1 выполняется, то исполняется Блок инструкций 1, в противном случае проверяется условие_2. Если оно выполняется, то исполняется Блок инструкций 2, в противном случае Блок инструкций 3.

Пример. В нем сравниваются значения One, Two. В зависимости от результата выводится одно из трех сообщений.

```
using System;
    class Conditional
    {
        static void Main()
        {
            int One = 50;
            int Two = 5;
            if ( One > Two )
                Console.WriteLine ("One: {0} больше Two: {1}", One, Two);
            else
                if ( One == Two )
                    Console.WriteLine ("One: {0} равно Two: {1}", One, Two);
                else
                    Console.WriteLine("One: {0} меньше Two:  {1}" , One, Two);
        }
```

```
}
```

8.4. Выбор switch; case

Когда вы имеете сложный набор условий, то использование вложенных инструкций if…else приводит к громоздкому коду. Лучше воспользоваться инструкцией выбора **switch**. Инструкция switch выбирает нужное действие из списка возможных, размещенных во фразах выбора **case**.

Синтаксис инструкции

```
switch (Условие)
{
    case константа_1 : инструкция действия; инструкция прерывания;
    case константа_2 : инструкция действия; инструкция прерывания;
    ……..
    default:: инструкция;
}
```

Условие (помещено в круглые скобки) возвращает константу Далее следует блок из секций.

- Секция выбора — **case**. Она нужна для определения действия, которое будет выполняться при совпадении значения Условия с константой в секции case. В этой секции после двоеточия (:) следуют инструкции действий (хотя бы одна), а также инструкция прерывания действия (она обязательна, иначе будет сквозное выполнение секций выбора).
- Секция действия по умолчанию — **default**. Она может отсутствовать. Она выполняется в том случае, если со значением константы Условие не совпала ни одна константа из секции выбора.

Если результат Условия совпадет с константным значением секции case, то будет выполняться соответствующий ему блок инструкций. В качестве инструкции прерывания действия используют **break**, которая прерывает выполнение инструкции switch. Альтернативой может быть и инструкции **goto**, которую обычно применяют для перехода в другое место программы.

Пример. Программа запрашивает номер пользователя. В зависимости от введенного номера выводится строка из списка.

```
using System;
namespace SwitchStatement
{
    class MyClass
    {
```

```
        static void Main(string[] args)
        {
            int user = 1;
            Console.Write("Ваш номер = ");
            user = Convert.ToInt32(Console.ReadLine());
            switch (user)
            {
                case 1: Console.WriteLine("Здравствуйте Userl"); break;
                case 2: Console.WriteLine("Здравствуйте User2"); break;
                case 3: Console.WriteLine("Здравствуйте User3"); break;
                default: Console.WriteLine("Здравствуйте новичок"); break;
            }
        }
    }
}
```

9. Циклы

Циклом называется группа инструкций, повторяющихся многократно с разными данными. Выбор типа цикла зависит от задачи программирования и личных предпочтений кодирования. Для циклов применяются инструкции: goto, for, while, do while

Одним из основных отличий C# от других языков, таких как C++, является цикл foreach, разработанный для упрощения итерации по массиву или коллекции.

9.1. Команда goto и метки

Команда **goto** в первых языках программирования была основой для реализации циклов и многократных переходов, вследствие чего возникла запутанность кода программы. Известный программист Дейкстра своим ученикам за использование **goto** ставил неуды. Поэтому опытные программисты стараются ее не использовать, но для того чтобы узнать все возможности языка, рассмотрим и ее.

Инструкция goto используется следующим образом:

- В коде программы создается метка с именем (например, M).
- Организуется переход на эту метку инструкцией goto M.

Имя метки M в коде обязательно должно заканчиваться двоеточием (:). Оно указывает на точку в программе, с которой будет выполняться программа после использования инструкции goto. Обычно инструкция goto привязывается к условию, как показано в примере.

Пример. Программа использует цикл (метка M - начало), в котором в консоль выводится последовательность чисел.

```
using System;
public class Labels
    {
    public static int Main()
    {
            int i = 0;
        M : Console.WriteLine("i: {0 } ", i);
            i = i + 1;
            if (i < 10) goto M;
        }
    }
```

9.2. Цикл for

Это цикл с заданным числом повторений. В нем изменение индекса цикла заложено в инструкцию. Задаются - начальное значение индекса, условие выполнения, правило изменения индекса после итерации. Разделители для параметров инструкции for – точка с запятой (;)/

Синтаксис инструкции:

```
for (индекс цикла = начало; условие выполнения; изменение индекса)
{
    Инструкции тела цикла;
}
```

Пример. Программа использует цикл, в котором в консоль выводится последовательность чисел от 0 до 9.

```
using System;
public class Labels
    {
    public static int Main()
    {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("i = {0 } ", i);
            }
        }
    }
```

9.3. Цикл while

Это цикл с неизвестным числом повторений с предусловием. Тело цикла повторяется, **пока выполняется условие**. Тело цикла первый раз выполняется с проверкой условия. Ее синтаксис выглядит следующим образом:

```
while (Условие)
{
    Инструкции тела цикла;
}
```

Пример. Программа использует цикл, в котором в консоль выводится последовательность чисел от 0 до 9.

```
using System;
public class CycleWhile
    {
```

```
    public static int Main()
        {
            int i = 0;
            while (i < 10)
            {
                Console.WriteLine("i = {0 } ", i);
                i = i + 1;
            }
        }
    }
```

9.4. Цикл do - while

Это цикл с неизвестным числом повторений с постусловием. Тело цикла повторяется, пока выполняется условие. Тело цикла первый раз выполняется без проверки условия. Выход из цикла при **не выполнении условия**.

Эта циклическая инструкция работает по принципу: «Пока выполняется условие — повторить». Ее синтаксис выглядит следующим образом:

```
do
{
    Инструкции тела цикла;
}
while (Условие)
```

Пример. Программа использует цикл, в котором в консоль выводится последовательность чисел.

```
using System;
public class DoWhile
    {
    public static int Main()
        {
            int i = 0;
            do
            {
                Console.WriteLine("i = {0 } ", i);
                i = i + 1;
            }
            while (i < 10)
        }
    }
```

9.5. Безусловные переходы

Бывают ситуации, когда необходимо прекратить выполнение цикла досрочно (до того как перестанет выполняться условие), не прерывая при этом цикла. Для таких случаев удобно использовать команды break и continue.

Команда **break.** Если вы хотите на каком-то шаге цикла его прекратить, не выполняя до конца описанные в нем действия, то лучше всего использовать break. Следующий пример иллюстрирует его работу.

Программа выполняет поиск не простых чисел (которые имеют делитель без остатка) в интервале от 2 до 11. В программе используется два цикла for.

Первый цикл перебирает все числа делимого от 11 до 2. Переменная i инициализируется значением 11 и затем уменьшается на 1 с каждой итерацией. Второй цикл перебирает все числа делителя от (j = i -1) до 2.

В теле внутреннего цикла проверяется условие: делится ли число i на число j без остатка. Если условие выполняется, то число i нельзя отнести к разряду простых чисел, и флаг, определяющий число как простое, устанавливается в false. Дальнейший поиск не имеет смысла. Цикл завершается командой break.

```
using System;
namespace Break
{
   class Program
   {
      static void Main()
      {
         bool IsPrimeNumber = true;  // устанавливаем флаг простого числа
         for (int i = 11; i > 1; i--)
         {
            for (int j = i - 1; j > 1; j--)
            {
               IsPrimeNumber = true;
               // если есть делитель с нулевым остатком, сбрасываем флаг
               if (i % j == 0)
               {
                  IsPrimeNumber = false; // дальнейшая проверка бессмысленна
                  Console.WriteLine("{0} — не простое число", i);
                  Console.WriteLine("Нажмите любую клавишу");
                  Console.ReadKey();
               }
               if (IsPrimeNumber == false) break;
```

```
            }
        }
    }
  }
}
```

Команда **continue** в отличие от break не прерывает хода выполнения цикла. Она лишь приостанавливает текущую итерацию и переходит к следующей итерации.

Пример. В нем в консоль выводятся нечетные числа в заданном диапазоне. Поверка осуществляется проверкой остатка от деления на 2, для нечетных чисел он не равен 0. В цикле перебираются все числа от 1 до 100. Если очередное число четное, то итерация завершается командой continue с пропуском последующих инструкций тела цикла и переходом к следующей итерации.

```
using System;
class PoiskNechet
{
    static void Main()
    {
        for ( int i = 100; i > 0; i--)
        {
            if ( i%2 ==0 ) continue;
            Console.WriteLine("{0} - нечетное число", i);
            Console.WriteLine("Нажмите любую клавишу");
            Console.ReadKey();
        }
    }
}
```

9.6. Вечные циклы

При написании приложений с использованием циклов следует остерегаться зацикливания программы. Зацикливание — это ситуация, при которой условие выполнения цикла всегда истинно и выход из цикла невозможен.

9.7. Команда foreach

Команда **foreach** (для каждого) предназначена для обработки массивов. Она повторяет группу вложенных в нее инструкций для каждого элемента массива.

Синтаксис:

foreach (<ИндексЦикла> in <ИмяМассива>)

```
{
    Инструкции тела цикла;
}
```

В любой точке блока foreach цикл можно разорвать с помощью ключевого слова **break** или перейти к следующей итерации в цикле с помощью ключевого слова **continue**.

Цикл foreach также может быть разорван при помощи **goto**, **return** или **throw**.

Пример. Вывод в консоль содержимого массива целых чисел.

```
using System;
class ForEachTest
  {
    static void Main()
    {
      int[] Massiv = { 0, 1, 2, 3, 5, 8, 13 }; .// Определен массив чисел
      foreach (int i in Massiv)
      {
        Console.WriteLine(i);
      }
    }
  }
```

10. Обработка ошибок и исключений

Если во время выполнения программы что-то работает неправильно, создается исключение. Исключение останавливает текущий поток программы и если никакие меры не предпринимаются, программа просто прекращает выполнение. Причиной исключений могут быть ошибки в программе (например, деление числа на ноль) или неожиданный ввод (например, выбор несуществующего файла). Задачей программиста является предоставление программе возможности устранить проблемы, не приводя к сбою.

Программист может перехватить исключительные ситуации и сделать их обработку. Для этого в C# представлено несколько ключевых слов, — try, catch и finally — с помощью которых программа обнаруживает исключения, устраняет их и продолжает выполнение. Они способствуют повышению надежности приложений.

Исключения имеют типы, являющиеся производными от System.Exception. Исключения, генерируемые при компиляции:

Исключение	Описание
ArithmeticException	Основной класс исключений, происходящих при выполнении арифметических операций, таких как DivideByZeroException и OverflowException.
ArrayTypeMismatchException	Создается, когда массив не может хранить данный элемент, поскольку фактический тип элемента несовместим с фактическим типом массива.
DivideByZeroException	Создается при попытке разделить целое число на ноль.
IndexOutOfRangeException	Создается при попытке индексирования массива, если индекс меньше нуля или выходит за границы массива.
InvalidCastException	Создается, когда происходит сбой явного преобразования из основного типа в интерфейс либо в производный тип во время выполнения.
NullReferenceException	Создается при попытке ссылки на объект, значение которого равно null.
OutOfMemoryException	Создается при неудаче попытки выделения памяти с помощью оператора new. Это означает, что память, доступная для среды выполнения, уже исчерпана.
OverflowException	Создается при переполнении арифметической

	операции в контексте checked.
StackOverflowException	Создается, когда стек выполнения переполнен за счет слишком большого количества вызовов отложенных методов; обычно является признаком очень глубокой или бесконечной рекурсии.
TypeInitializationException	Создается, когда статический конструктор создает исключение, и не существует ни одного совместимого предложения catch для его захвата.

10.1. Try, Catch

Ключевые слова try и catch используются вместе. Если предполагается, что блок кода может вызвать исключение, воспользуйтесь ключевым словом **try**, и используйте **catch**, чтобы сохранить код, который будет выполнен при возникновении исключения. В этом примере в результате деления на ноль создается исключение, которое затем перехватывается. При отсутствии блоков try и catch произойдет сбой программы.

Синтаксис инструкции:

```
try
{
   // Проверяемый код здесь
}
catch (SomeSpecificException ex)
{
   // Код обработчика исключения здесь.
}
```

Пример. Обработка исключения **DivideByZeroException** –деление на 0

```
using System;
class ProgramTryCatch
{
   static void Main()
   {
      int x=0, y=0;

      try
      {
         x = 10 / y;   // Проверяемая инструкция, возможно деление на 0
      }
```

```
        catch (DivideByZeroException)      // Обработчик исключения
        {
            Console.WriteLine("Попытка деления на 0.");
        }
    }
}
```

10.2. Try, Catch, Finally

Код, содержащийся в блоке finally, выполняется всегда, вне зависимости от возникновения исключения. Чтобы гарантировать возвращение ресурсов, например, убедиться, что файл закрыт, используйте блок finally.

Синтаксис инструкции:

```
class ProgramTryCatchFinally
{
    static void Main()
    {
try
{
    // Проверяемый код здесь
}
catch (<ИмяИсключения>)
{
    // Код обработчика исключения здесь.
}
finally
{
    // Код, выполняемый после try (и возможно catch)
}
```

11. Работа со строками

11.1. Представление строк

Строка C# представляет собой группу одного или нескольких знаков, объявленных с помощью ключевого слова **string**.

Каждый знак запоминается в кодировке unicod (16 бит, 2 байта).

В отличие от массивов знаков в C или C++, строки в C# гораздо проще в использовании и менее подвержены ошибкам программирования.

Строковый литерал объявляется с помощью двойных кавычек, как показано в следующем примере.

string Приветствие = "Hello, World!";

В классе String определены методы для обработки строк. Функциональное назначение методов:

- Создание и удаление строк.
- Копирование и объединение.
- Длина и позиционирование.
- Представление строки.
- Преобразования строки в иной тип.
- Преобразования иного типа в строку.
- Сравнение строк.

11.2. Метод ToString()

Все встроенные типы данных C# предоставляют метод ToString(), преобразующий значение в строку. Этот метод может быть использован для преобразования числовых значений в строки.

Пример. Создается строка Сообщение путем конкатенации (объединения) задаваемой строки и преобразованного в строку значения целочисленной переменной Год.

```
int Год = 1999;
string Сообщение = "Я родился в " + Год.ToString();
```

11.3. Доступ к отдельным знакам

К отдельным знакам, содержащимся в строке str, можно получить доступ с помощью таких методов:

Вызов	Действие

Clone()	Возвращает ссылку на экземпляр класса
Copy(str)	Копирование строки str
Concat(strA, strB)	Сцепление строки strA со строкой strB
Compare(strA, indA, StrB, indB)	Сравнивает подстроки строк strA strB в позициях indA и indB
str. Substring(E,L)	Выделяет, начиная с позиции E (нумерация от 0), подстроку длиной L,
strA.CompareTo(strB)	Сравнивает строку strA со строкой strB
strA.Replace(strA, strB)	Заменяет строку strA на строку strB
str.Remove(Ind, Count)	Удаляет Count знаков после позиции Ind
strA.Insert(Ind, strB)	Вставляет строку strB в строку strA с позиции ind
Equals(strA, strB)	Проверка совпадения строк strA и strB
ToCharArray(str)	Возвращает массив символов строки str
str.Split('R')	Возвращает массив строк из подстрок, разделенных символом R.
str.Trim()	Удаляет символы пробела в начале и конце строки.
str.GetHashCode()	Возвращает хэш-код для этой строки
str.Length	Возвращает число знаков в str
str.ToLower()	Копия str в нижнем регистре
str.ToUpper()	Копия str в верхнем регистре

Подстрокой является последовательность символов, содержащихся в строке.

Метод Substring используется для создания новой строки на основании части исходной строки. Синтаксис метода:

СтрокаИсходник.Substring(Начало, Длина));

Метод Replace используется для замены всех вхождений заданной подстроки новой строкой. Синтаксис метода:

СтрокаИсходник.Replace(СтрокаЗаменяемая, ПодстрокаЗамещающая);

Одно или несколько вхождений подстроки можно найти с использованием метода IndexOf. Он возвращает номера позиций, с которых начинаются обнаруженные вхождения. Синтаксис метода:

СтрокаИсходник. IndexOf(ПодстрокаПоиска);

Пример.

```
using System;
string s3 = "Visual C# Express";                    // Исходная строка
```

```
Console.WriteLine(s3.Substring(7, 2));              // Подстрока "C#"
Console.WriteLine(s3.Replace("C#", "Basic")); // Замена "Visual Basic Express"
int index = s3.IndexOf("C");                        // index = 7
```

12. Массивы и коллекции

12.1. Коллекции

Коллекция – способ хранения наборов данных. Различают виды коллекций:

- **Массив** является одним из вариантов хранения набора данных, используемых C#.
- **Список** работает, как правило, быстрее массива при добавлении элемента в начало или в середину коллекции.
- **Хэш-таблица**. Это набор данных с категориями с определенным признаком.
- **Связаный список.** Это набор данных, связанных друг с другом.
- **Стек**. Это данные в стековой памяти.

12.2. Массивы

Массив — это структура данных, содержащая несколько переменных одного типа. Массивы объявляются со следующим синтаксисом.

type[] ИмяМассива;

- type - имя типа значений элементов.
- [ ] – признак массива. Запятые внутри скобок задают размерность массива. Запятых нет – массив одномерный, запятая одна – массив двумерный.

Элементы массива могут быть любых типов, включая тип массива.

Доступ к элементу массива: ИмяМассива [НомерЭлемента].

Индексация массивов начинается с нуля: массив с элементами n индексируется от 0 до n-1.

При инициализации массива значения его элементов помещаются в фигурные скобки и разделяются запятыми.

В примере показано создание одномерного и двумерного массивов.

```
class TestArraysClass
{
    static void Main()
    {
        int[ ]  1Массив1 = new int[5];                  // 1-массив из 5 чисел
        nt[  ] 1Массив2 = new int[] { 1, 3, 5, 7, 9 };  // 1-массив инициализирован
        int[ ] 1Массив3 = { 1, 2, 3, 4, 5, 6 };        // Альтернативный синтаксис
```

```
        int[,] 2Массив1 = new int[2, 3];                // 2-массив 2x3 чисел
        int[,] 2Массив2= { { 1, 2, 3 }, { 4, 5, 6 } };  // 2-массив инициализирован
    }
}
```

12.3. Использование инструкции foreach, in

В C# также предусмотрена инструкция foreach. Она обеспечивает простой и понятный способ выполнения итерации элементов в массиве.

Синтаксис инструкции foreach:

```
foreach (int ИндексМассива in ИмяМассива)
{
    Инструкции теле цикла
}
```

- int ИндексМассива – тип и имя переменной для номера элемента массива (целое число).
- Слово in – в.
- ИмяМассива – имя массива.

Пример. Следующий код создает массив Числа и осуществляет его итерацию с помощью инструкции foreach.

```
int[] Числа = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in Числа)
{
    System.Console.Write("{0} ", i);
}
```

Вывод в консоль строки: 4 5 6 1 2 3 -2 -1 0

Для лучшего контроля элементов в многомерных массивах можно использовать вложенный цикл for.

13. Графика

13.1. Объект Graphics

Объект **Graphics** - это указатель на место где будут рисоваться примитивы. Пусть мы хотим рисовать в форме Windows. Синтаксис задания ссылки на нее:

```
Graphics g = Graphics.FromHwnd(this.Handle);
```

Здесь:

- Graphics - тип объекта,
- g - имя переменной,
- Graphics.FromHwnd(this.Handle) - используемый метод FromHwnd из класса Graphics, который задает ссылку Handle на форму Windows.

В C# инструменты рисования определены в пространстве имен System.Drawing. Там находятся классы:

- Pen (перо). Объекты пера используются в методах рисования линий и графических фигур.
- Brush (кисть). Объекты кисти используются в методах заливки графических фигур.

13.2. Перо (Pen)

Объекты пера используются в методах рисования линий и графических фигур.

Объекты Pen выбираются из класса Pens (перья). Класс Pens содержит набор объектов для выбора. У них толщина линии (1 пиксель), стиль линии – сплошная. У каждого объекта свой цвет линии, имя которого идентифицирует объект. Такой объект нельзя редактировать, его можно только применять. Например, создаем объект myPen, совпадающий с шаблоном:

```
Pen myPen = Pens.Black;
```

Объекты Pen с изменяемыми свойствами создаются из класса Pen (перо). В этом случае для объекта пера можно устанавливать много свойств. Основные свойства:

- Color - цвет линии;
- Brush – ссылка на кисть, используемую в качестве пера ;
- Width – толщина линии;
- DashStyle – стиль пунктирной линии (Dash – штрих, DashDot – штрих пунктир, DashDotDot - штрих двойной пунктир, Dot - пунктир).

Сначала объект myPen можно создать с указанием цвета.

Pen myPen = new Pen(Color.Black);

Затем ему можно изменить свойства:

```
myPen.DashStyle = System.Drawing.Drawing2D.DashStyle.Solid;
myPen.PenType = System.Drawing.Drawing2D.PenType.SolidColor;
myPen.Width = 2;
```

Можно задать и красиво пишущее перо, в нем перо это кисть.

Pen myFancyPen = new Pen(myBrush);

13.3. Кисть (Brush)

Объекты кисти используются в методах заливки графических фигур.

Определены кисти разного типа:

- Brush –простая кисть, одноцветная заливка.
- HatchBrush – кисть со штриховой заливкой.
- LinearGradientBrush - кисть с линейной градиентной заливкой, цвет фрагментов фигуры меняется плавно.
- PathGradientBrush - кисть с градиентной заливкой, цвет фрагментов фигуры меняется скачкообразно.

Объекты Brush выбираются из класса Brushes, который содержит кисти со сплошной заливкой. Класс Brushes содержит набор объектов для выбора, у которых по умолчанию определен цвет. У каждого объекта выбора имя – это цвет заливки. Например, создаем объект myBrush, совпадающий с шаблоном:

```
Brush myBrush = Brushes.Blue;          // Заливка синим
```

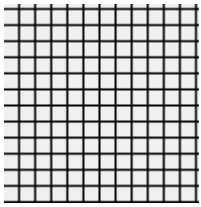
Примеры нарисованных графиков:

Объекты HatchBrush выбираются из класса HatchBrushes. Класс HatchBrushes содержит набор объектов для выбора, у которых по умолчанию определены стиль заливки HatchStyle, цвет переднего плана ForeColor и цвет фона BackColor. Определено много стилей заливки: сетка (Cross), диагональная сетка (DiagonalCross), прямая диагональ (ForwardDiagonal), обратная диагональ

(BackwardDiagonal) и др. Например, создаем кисть с заливкой сеткой HatchStyle.Cross:

HatchBrush brush2 = new HatchBrush(HatchStyle.Cross, ForeColor, BackColor);

Это результат заливки прямоугольника такой кистью.



13.4. Шрифты и текст

Для вывода в форму текста используется метод DrawString. Рисует заданную текстовую строку в заданном прямоугольнике с помощью определяемых объектов кисти и шрифта, используя атрибуты форматирования заданного формата. Синтаксис метода:

DrawString(S, Font, Brush, RectangleF, StringFormat);

- S –текстовая строка для рисования.
- Font – шрифт текстовой строки.
- Brush – кисть. Определяет цвет и текстуру создаваемого текста.
- RectangleF – прямоугольник вывода.
- StringFormat – формат. Определяет атрибуты форматирования, такие как междустрочный интервал и выравнивание, которые применяются к создаваемому тексту.

Метод перегружаемый. Возможны несколько способов вызова, отличающиеся друг от друга числом аргументов (формат можно не указывать, будет использован формат по умолчанию) и способом задания прямоугольника вывода.

Font. Это шрифт текстовой строки. Выбирается с помощью методов класса Font. Они предоставляет возможность выбора размера и стиля шрифта. Методы перегружаемые. Возможны несколько способов вызова, отличающиеся друг от друга числом аргументов и способом задания нового шрифта. Например:

font МойШрифт = new Font("Arial" , 24 , FontStyle.Bold) ;

RectangleF

Это прямоугольник, в котором рисуется строка текста. Задается двумя способами:

- 4 координаты - левого верхнего (X1, Y1) и правого нижнего (X2, Y2) углов.
- Объект точка с координатами левого верхнего угла (P) и размеры (H - ширина и W -высота).

13.5. Методы рисования

В C# определены методы рисования линий и фигур. Все методы перегружаемые, то есть выполняются по-разному с разными аргументами.

При рисовании можно использовать перо с разными стилями линий LineStyle. Например, solid (сплошная), Dash (штрих), Dot (пунктир), DashDot (штрих-пунктир), DashDotDot (штрих-пунктир-пунктир).

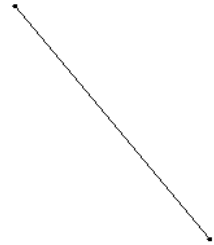
DrawLine.

Прямая линия между двумя точками. Синтаксис метода.

g.DrawLine(pen, p[0], p[2]);

Здесь

- g – где рисуем,
- DrawLine – метод - рисуем линию,
- Pen – перо,
- p[0], p[2] – точки границы линии, отмечены точками.



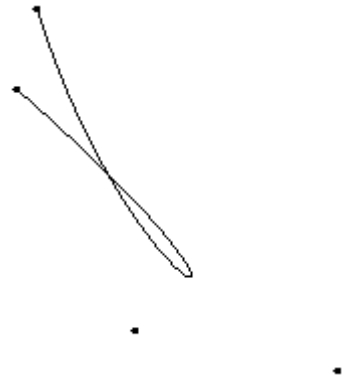
DrawBezier

Кривая Безье, плавная кривая, проходящая вблизи 4-ех точек. Синтаксис метода.

g.DrawBezier(pen, p[0], p[1], p[2], p[3]);

Здесь

- g – где рисуем,
- DrawBezier –рисуем кривую Безье,
- Pen – перо,
- p[0], p[1], p[2], p[3] – точки.



Алгоритм рисования использует интерполяционный полином ограниченного порядка. Поэтому линия может заметно отклоняться от точек.

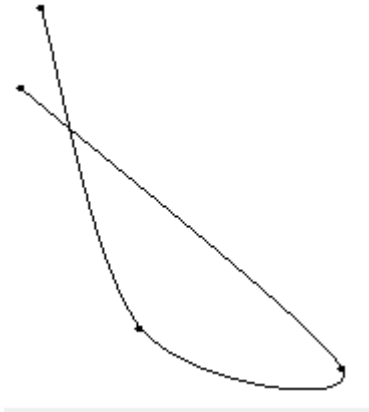
DrawCurve

Плавная кривая, проходящая через набор из массива точек. Синтаксис метода.

g.DrawCurve(pen, p);

Здесь

- g – где рисуем,
- DrawCurve –рисуем кривую,
- Pen – перо,
- p – массив точек.



Алгоритм рисования использует аппроксимационный полином. Поэтому линия проходит через все точки.

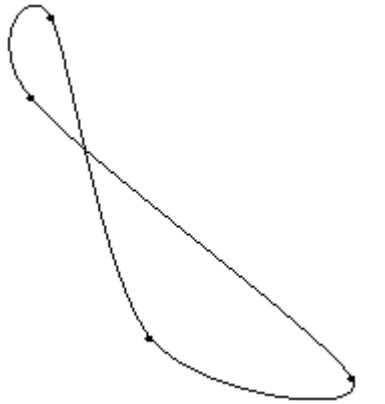
DrawClosedCurve

Замкнутая плавная кривая, проходящая через набор из массива точек. Синтаксис метода.

g.DrawClosedCurve(pen, p);

Здесь

- g – где рисуем,
- DrawClosedCurve –рисуем замкнутую кривую,
- pen – перо,
- p – массив точек.



Алгоритм рисования использует аппроксимационный полином. Поэтому линия проходит через все точки.

DrawRectangle

Прямоугольник. Синтаксис метода.

g.DrawRectangle(pen, rect);

Здесь

- g – где рисуем,
- DrawRectangle –рисуем прямоугольник,
- pen – перо,
- rect – прямоугольник, свойства которого задаются целыми числами

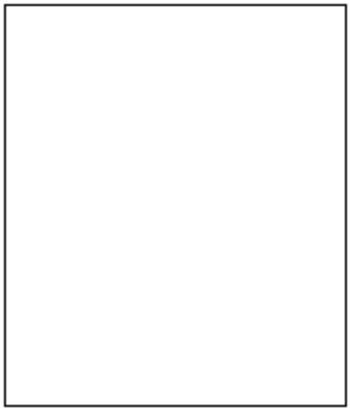
rectangle Прямоугольник = new Rectangle(x, y, h, w);.

Здесь

- x, y – ккординаты левого верхнего угла,
- h – ширина,
- w – высота.

Возможен вариант задания свойств числами в формате с плавающей точкой

rectangle Прямоугольник = new RectangleL(x, y, h, w);



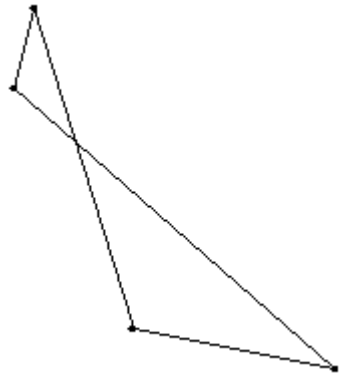
DrawPoligon

Полигон. Это многоугольник, формируемый соединением линиями точек массива. Крайние точки массива замыкаются. Образуется замкнутая фигура с возможными пересечениями линий. Синтаксис метода.

g.DrawPoligon (pen, p[]);

Здесь

- g – где рисуем,
- DrawPoligon – рисуем полигон,
- pen – перо,
- p[] – массив точек.



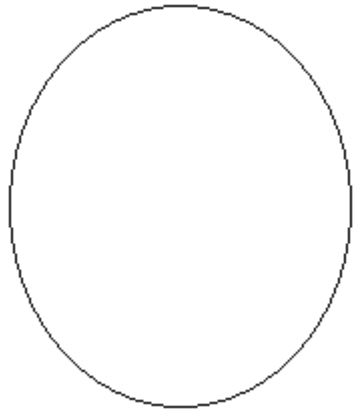
DrawEllipse

Эллипс. Синтаксис метода.

g.DrawEllipse(pen, rect);

Здесь

- g – где рисуем,
- DrawEllipse –рисуем эллипс,
- pen – перо,
- rect – прямоугольная область, в которую вписывается эллипс.



DrawArc

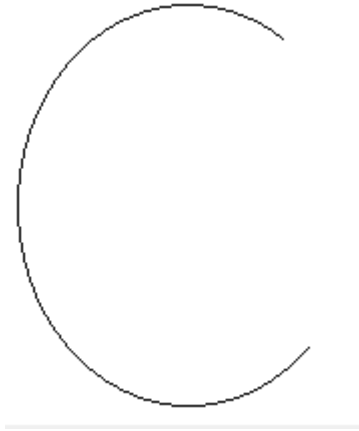
Дуга, часть эллипса. Синтаксис метода.

g.DrawArc(pen, rect, StartAngle, EhdAngle);

Здесь

- g – где рисуем,
- DrawEllipse – рисуем эллипс,
- pen – перо,
- rect – прямоугольная область, в которую вписывается эллипс,
- StartAngle – начальный угол,
- EhdAngle – конечный угол

Углы в градусах, по часовой стрелке, начало отсчета – горизонтальная ось.

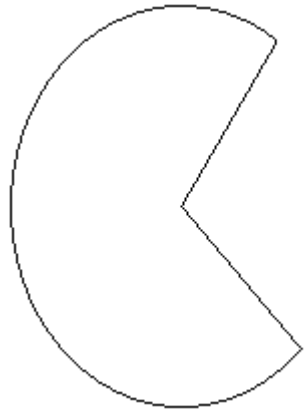


DrawPie

Сектор эллипса. От дуги отличается тем, что концы дуги соединяются с центром эллипса радиусами. Синтаксис метода.

g.DrawPie(pen, rect, StartAngle, EhdAngle);

- DrawPie – рисуем сектор.



13.6. Методы заливки

В C# определены методы заливки фигур. Все методы перегружаемые, то есть выполняются по-разному с разными аргументами.

В примерах заливка разных фигур осуществляется простой кистью с Brush.Cyan и кистью HatchBrush с разными стилями заливки DashStyle.

FillClosedCurve.

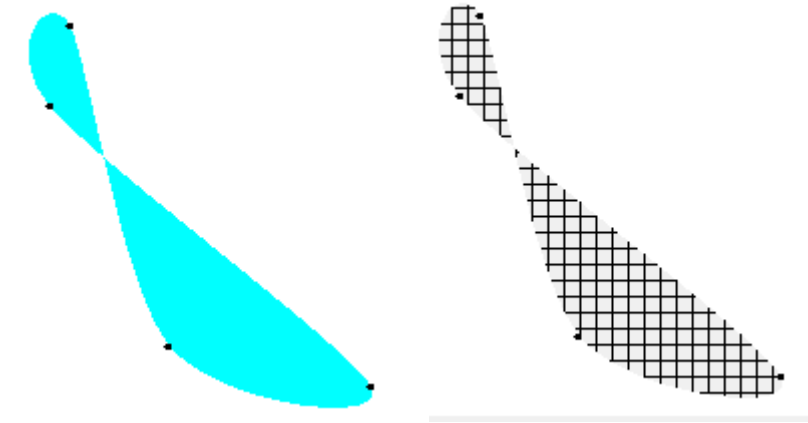
Закрашивает внутренние области замкнутой кривой, проходящей через набор из массива точек (они обозначены точками). Синтаксис метода.

g.FillClosedCurve(brush, p);

Здесь

- g – где рисуем,
- FillClosedCurve – рисуем залитую замкнутую кривую,
- brush – кисть,
- p – массив точек.

Получаемый результат зависит от стиля заливки. Рисунок слева использована простая кисть с цветом Cyan. Справа использована кисть, у которой стиль заливки cross (сетка), цвет переднего плана черный, фона белый.



FillRectangle.

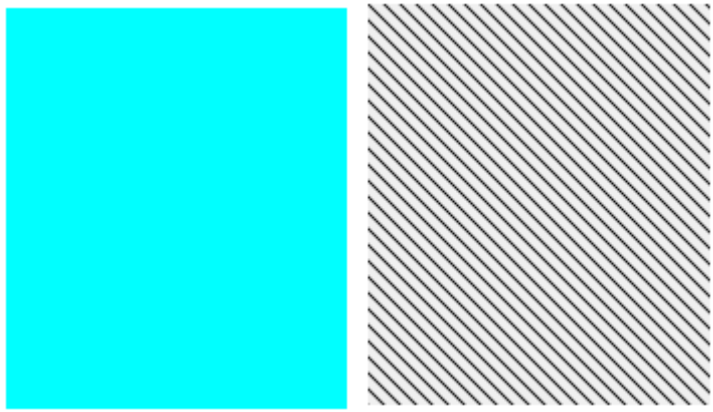
Закрашивает прямоугольник. Синтаксис метода.

g.FillRectangle (brush, p);

Здесь

- g – где рисуем,
- FillRectangle – рисуем залитый прямоугольник
- brush – кисть,
- p – массив точек.

Получаемый результат зависит от стиля заливки. Рисунок слева использована простая кисть с цветом Cyan. Справа использована кисть, у которой стиль заливки BacwardDiagonal (обратная диагональ), цвет переднего плана черный, фона белый.



FillPoligon.

Закрашивает внутренние области полигона, узлы которого совпадают с точками из массива (они обозначены точками).

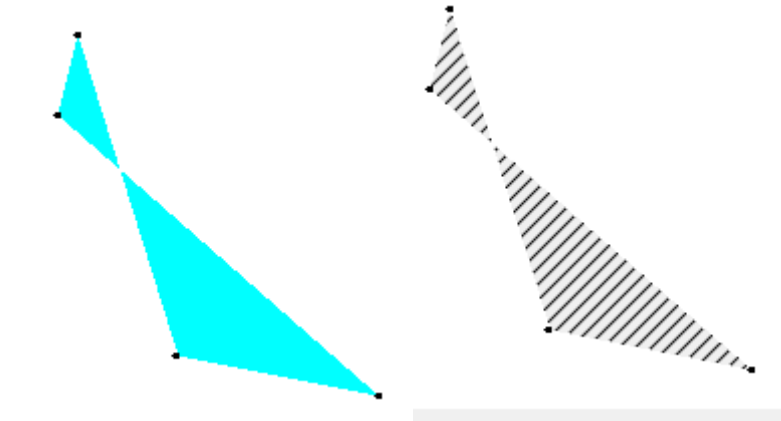
Синтаксис метода.

g. FillPoligon(brush, p);

Здесь

- g – где рисуем,
- FillPoligon – рисуем залитый полигон
- brush – кисть,
- p – массив точек.

Получаемый результат зависит от стиля заливки. Рисунок слева использована простая кисть с цветом Cyan. Справа использована кисть, у которой стиль заливки ForwardDiagonal (прямая диагональ), цвет переднего плана черный, фона белый.



FillEllipse.

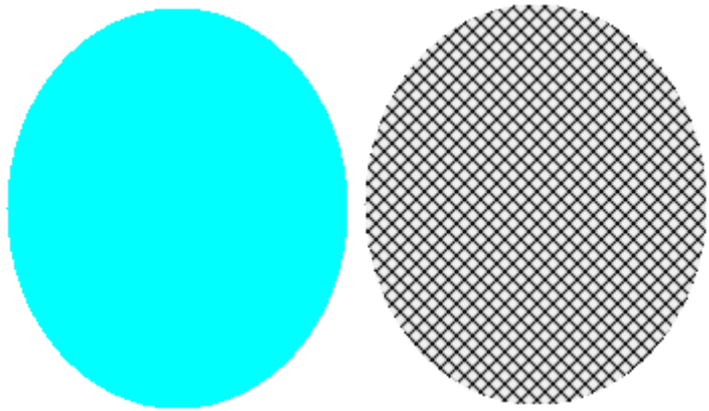
Закрашивает эллипс. Синтаксис метода

g.FillEllipse (brush, rect);

Здесь

- g – где рисуем,
- FillEllipse – рисуем залитый эллипс,
- brush – кисть,
- rect – прямоугольная область, в которую вписывается эллипс/

Получаемый результат зависит от стиля заливки. Рисунок слева использована простая кисть с цветом Cyan. Справа использована кисть, у которой стиль заливки DiagonalCross (диагональная сетка), цвет переднего плана черный, фона белый.



FillPie.

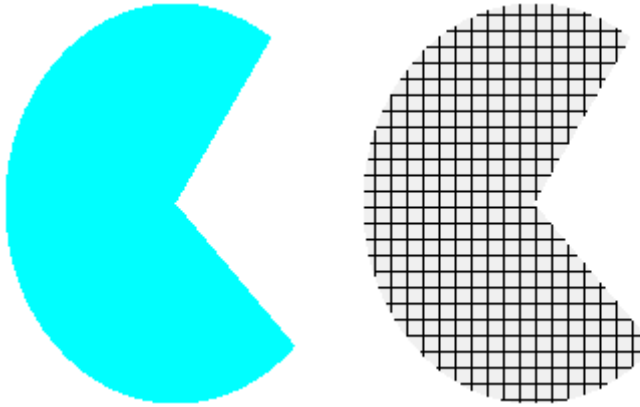
Закрашивает сектор эллипса. Синтаксис метода

g.FillPie (brush, rect);

Здесь

- g – где рисуем,
- FillPie – рисуем залитый сектор эллипса,
- brush – кисть,
- rect – прямоугольная область, в которую вписывается эллипс/

Получаемый результат зависит от стиля заливки. Рисунок слева использована простая кисть с цветом Cyan. Справа использована кисть, у которой стиль заливки Cross (сетка), цвет переднего плана черный, фона белый.



13.7. Рисование графика функции

Рассмотрим пример программы, которая в форме Windows рисует график синусоиды с текстовым заголовком. Форма включает кнопку с надписью Старт, которая запускает обработчик, выполняющий все операции.

В обработчике выполняются действия:

- Вычисляется функция.
- Задается ссылка на объект графики.
- Выводится текст заголовка.
- Рисуются координатные оси.
- Рисуется график функции.

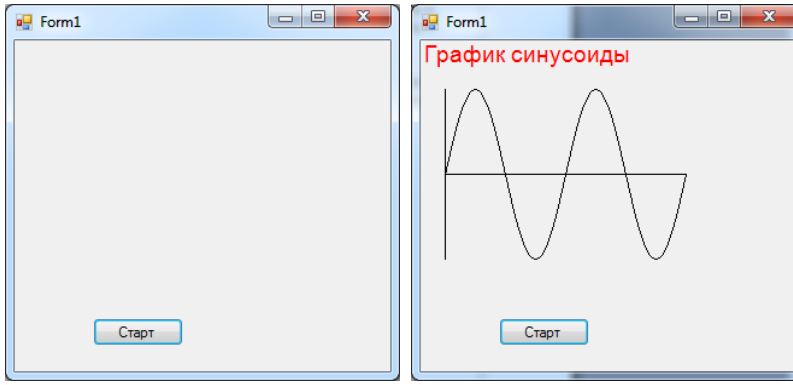
Листинг программы

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace GraphicFunctionApplication
{
   public partial class Form1 : Form
   {
```

```
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            int imax =100;                  //число точек в периоде
            int t=2;                        //число периодов
            int amp=70;                     //амплитуда
            int h = 40;                     //отступ для текста
            int x0=20;                      //начала координат
            int y0 = h+amp;
            double[] f = new double [imax*t+10];
            // Функция
            for (int i = 0; i < imax * t; i++)
            {
                f[i] = Math.Round(amp * Math.Sin(2 * Math.PI / imax * i));
            }
            // Инструменты рисования
            Graphics g = Graphics.FromHwnd(this.Handle);         // Где рисуем
            Pen pen = Pens.Black;                                // Чем рисуем
            // Текст заголовка
            g.DrawString("График синусоиды", new Font("Arial", 14),
                 Brushes.Red, 0, 0);       //Вывод текста
            //textBox1.Text = "График синусоиды";
            //Рисуем график
            g.DrawLine(pen , x0, y0, x0+imax*t, y0);             //Рисуем ось X
            g.DrawLine(pen, x0, y0-amp, x0, y0+amp);             //Рисуем ось Y
            for (int i = 0; i < imax * t; i++)           //Рисуем график
            {
                int f1 = y0 - (int)f[i];                         //Координата Y[i]
                int f2 = y0 - (int)f[i + 1];                     //Координата Y[i+1]
                g.DrawLine(pen, x0+i, f1, x0+i+1, f2);
            }
        }
    }
}
```

При прогоне отображается форма, в которой затем рисуется график.



13.8. Растровая графика

Для растровой графики применяются различные форматы:

- BitMap (bmp), попиксельная графика.
- TIFF
- GIF
- PNG
- Exif

BMP (Bitmap Picture) — формат хранения растровых изображений в виде пикселей.

С форматом BMP работает огромное количество программ, так как его поддержка интегрирована в операционные системы. Файлы формата BMP могут иметь расширения .bmp, .dib и .rle.

Представляет собой набор пикселей, каждый из которых отображается 3 байтами. В байтах фиксируется интенсивность компонент цветов пикселя R (красный), G (зеленый), B (синий). Это самый точный, но и самый емкий по объему формат

В формате BMP есть поддержка сжатия по алгоритму RLE, однако теперь существуют форматы с более сильным сжатием, и из-за большого объёма BMP редко используется в Интернете, где для сжатия без потерь используются PNG и более старый GIF.

TIFF (Tagged Image File Format) — формат хранения с использованием тегов. Изначально был разработан компанией Aldus в сотрудничестве с Microsoft для использования с PostScript. TIFF стал популярным форматом для хранения изображений с большой глубиной цвета. Файлы формата TIFF, как правило, имеют расширение .tiff или .tif.

Имеется возможность сохранять изображение в файле формата TIFF со сжатием и без сжатия. Степени сжатия зависят от особенностей самого сохраняемого изображения, а также от используемого алгоритма. Формат TIFF позволяет использовать следующие алгоритмы сжатия:

- RLE, с обнаружением длинных одноцветных фрагментов.
- Lempel-Ziv-Welch (LZW), с использованием словарей повторяющихся фраз, как в архиваторах.
- ZIP, архивирование.
- JPEG.

GIF (Graphics Interchange Format) — формат для обмена изображениями. Формат GIF способен хранить сжатые данные без потери качества в формате до 256 цветов. Независящий от аппаратного обеспечения формат GIF был разработан в 1987 году фирмой CompuServe для передачи растровых изображений по сетям. GIF использует LZW-компрессию, что позволяет неплохо сжимать файлы, в которых много однородных заливок (логотипы, надписи, схемы).

Изображение в формате GIF хранится построчно, поддерживается только формат с индексированной палитрой цветов. Стандарт разрабатывался для поддержки 256-цветовой палитры.

Один из цветов в палитре может быть объявлен «прозрачным». В этом случае в программах, которые поддерживают прозрачность GIF (например, большинство современных браузеров) сквозь пиксели, окрашенные «прозрачным» цветом будет виден фон.

Формат GIF допускает чересстрочное хранение данных. При этом строки разбиваются на группы, и меняется порядок хранения строк в файле. При загрузке изображение проявляется постепенно, в несколько проходов. Благодаря этому, имея только часть файла, можно увидеть изображение целиком, но с меньшим разрешением.

В чересстрочном GIF сначала записываются строки 1, 5, 9 и т. д. Таким образом, загрузив 1/4 данных, пользователь будет иметь представление о целом изображении. Вторым проходом следуют строки 3, 7, 11, разрешение изображения в браузере ещё вдвое увеличивается. Наконец, третий проход передаёт все недостающие строки (2, 4, 6…). Таким образом, задолго до окончания загрузки файла пользователь может понять, что внутри и решить, стоит ли ждать полной загрузки изображения. Чересстрочная запись незначительно увеличивает размер файла, но это, как правило, оправдывается приобретаемым свойством.

Формат GIF поддерживает анимационные изображения. Фрагменты представляют собой последовательности нескольких статичных кадров, а также ин-

формацию о том, сколько времени каждый кадр будет показан на экране. Анимация может быть закольцована, тогда после последнего кадра будет вновь показан первый и так далее.

Первая спецификация была создана в 1987 году компанией CompuServe для замены устаревшего формата RLE. GIF стал популярен в ходе развития Интернета, так как позволял использовать более компактные (по размеру файла) по сравнению с другими форматами картинки на веб-страницах. Хотя к настоящему времени формат во многом устарел, и для его замены создан формат PNG, он по-прежнему широко используется.

GIF первоначально был проприетарным форматом, однако срок его патентной защиты истёк.

PNG (portable network graphics) — растровый формат хранения графической информации, использующий сжатие без потерь. PNG был создан как для улучшения, так и для замены формата GIF графическим форматом, не требующим лицензии для использования. Обычно файлы формата PNG имеют расширение .PNG (.png).

Неофициально PNG расшифровывают как «PNG is Not GIF» («PNG — это не GIF») по аналогии с известным рекурсивным акронимом «GNU is Not Unix» («GNU — это не UNIX»).

PNG произносится по-английски, как слово ping.

JPEG (Joint Photographic Experts Group, по названию организации-разработчика) — один из популярных графических форматов, применяемый для хранения фотоизображений и подобных им изображений. Файлы, содержащие данные JPEG, обычно имеют расширения .jpeg, .jfif, .jpg, .JPG, или .JPE. Однако из них .jpg самое популярное расширение на всех платформах.

Алгоритм JPEG является алгоритмом сжатия данных с потерями.

При сохранении JPEG-файла можно указать степень качества, а значит и степень сжатия, которую обычно задают в некоторых условных единицах, например, от 1 до 100 или от 1 до 10. Большее число соответствует лучшему качеству, но меньшему сжатию, при этом увеличивается размер файла. Обыкновенно, разница в качестве между 90 и 100 на глаз уже практически не воспринимается.

Широкая поддержка формата JPEG в разнообразном ПО нередко приводит к кодированию в JPEG изображений, для того не предназначенных. Даже безо всякого выигрыша по степени сжатия в сравнении с правильно сделанными PNG или GIF, но с прискорбными последствиями для качества. Например, попытка записать в JPEG изображение, содержащее мелкие контрастные детали

(особенно, цветные) приведёт к появлению характерных хорошо заметных артефактов даже при высокой «степени качества».

При сжатии изображение преобразуется из цветового пространства RGB в YCbCr (яркостное Y и два цветоразностных Cb = Y - B, Cr = Y - R). Для цветоразностных пространств можно за счет прореживания уменьшить размеры, это первый шаг сжатия.

Далее, яркостный компонент Y и отвечающие за цвет компоненты Cb и Cr разбиваются на блоки 8x8 пикселов. Каждый такой блок подвергается дискретному косинусному преобразованию (ДКП). Коэффициенты ДКП затем квантуются по уровням. Высокочастотные коэффициенты подвергаются более сильному квантованию, чем низкочастотные. Это приводит к огрублению мелких деталей на изображении. Чем выше степень сжатия, тем более сильному квантованию подвергаются все коэффициенты.

JPEG 2000 (или jp2) — графический формат, который вместо дискретного косинусного преобразования, характерного для JPEG, использует технологию вейвлет-преобразования, основывающуюся на представлении сигнала в виде суперпозиции некоторых конечных базовых функций — волновых пакетов.

В результате такой компрессии изображение получается более гладким и чётким, а размер файла по сравнению с JPEG при одинаковом качестве уменьшается ещё на 30 %.

EXIF (Exchangeable Image File Format) — стандарт, позволяющий добавлять к изображению информацию, комментирующую его. Например, условия и способы его получения, авторство. и др. Получил широкое распространение в связи с появлением цифровых фотокамер. Информация, записанная в этом формате, может использоваться как пользователем, так и различными устройствами, например, принтером.

Разработчик формата — Japan Electronics and Information Technology Association (JEITA).

13.9. Примитивные компоненты

Формат метафайл Windows (с расширением *.wmf) хранит картинку в виде набора описаний или определений всех компонент графика и их характеристик (например, отрезков линий, шаблонов заполнения, текста и его атрибутов и т.п.). Как правило, размер метафайла значительно меньше файла растровой графики.

При открытии метафайла график можно "разобрать" на компоненты, выделить и изменить отдельные линии, шаблоны заполнения и цвета, отредактировать текст и изменить его параметры и т.д.

Не во всех приложениях, поддерживающих графический формат метафайла, доступны все характеристики, поддерживаемые стандартным метафайлом Windows (например, некоторые приложения не поддерживают вращение текста, и все метки вертикальных осей, сохраненные в системе STATISTICA в формате метафайла, будут расположены горизонтально при открытии в таких приложениях).

14. Подробнее о CIL

Материал ниже для особо любопытных.

Разработчик CIL Лидин Сергей – канадец Российского происхождения.

14.1. Ассемблер CIL

В составе .Net Framework SDK поставляется ассемблер ILASM, который позволяет компилировать текстовые файлы, содержащие CIL-код и метаданные.

Программы в CIL-формате состоят из следующих лексических элементов:

- идентификаторы;
- метки;
- константы;
- зарезервированные слова;
- специальные знаки;
- комментарии.

Идентификаторы чаще всего представляют последовательности символов, начинающиеся с **латинской** буквы (или с символов «_», «$», «@» и «?»), за которой следуют латинские буквы, цифры или символы «_», «$», «@» и «?».

Кроме того, для идентификаторов и меток существует особая форма записи в апострофах: она позволяет включать в идентификаторы любые символы Unicode. Например:

Label_1 $Name 'Идентификатор'

Несколько идентификаторов могут быть объединены в один с помощью точек. Например:

System.Console.WriteLine

Метка применяется для обозначения точки перехода. Признак метки - двоеточие после имени.

Целочисленные константы записываются либо в десятичной системе счисления, либо в шестнадцатеричной (тогда перед ними ставится префикс «0x»). Например:

128 -10 0xFF10B000

В **вещественных константах точка** используется для разделения целой и дробной части, а символы «e» и «E» служат для указания экспоненциальной части. Кроме того, поддерживается особая форма записи float32 (целая_константа) и float64 (целая_константа), позволяющая представить целое число в виде числа с плавающей точкой. Например:

5.5 -1.05e10 float32(128) float64(50)

Строковые константы записываются в двойных кавычках и могут содержать Escape-последовательности «\t», «\n» и «\xxx», где восьмеричное число xxx задает код символа от 0 до 255. Для переноса строковой константы на другую строку программы используется символ «\». Кроме того, для строковых констант поддерживается операция конкатенации «+». Например:

“Alpha Beta Gamma” “Hello, World\n” “Concat”+”enation”

Комментарии в CIL-программах записываются так же, как в языке C#. Начинаются с символов //. Многострочные комментарии помещаются в пару особых скобок /*…*/.

Синтаксис строки кода:

Метка команда // комментарии

Следуйте этим рекомендациям:

- Все инструкции должны начинаться с метки или пробела.
- Метки не обязательны, если они используется, они должны начаться в столбце 1.

- Один (или больше) пробелов должно отделять каждое поле. Символы табуляции интерпретируются, как пробелы.
- Комментарии необязательны.
- Мнемоника не может начинаться с 0 или 1, иначе это будет интерпретироваться, как метка.

14.2. Архитектура виртуальной машины CIL

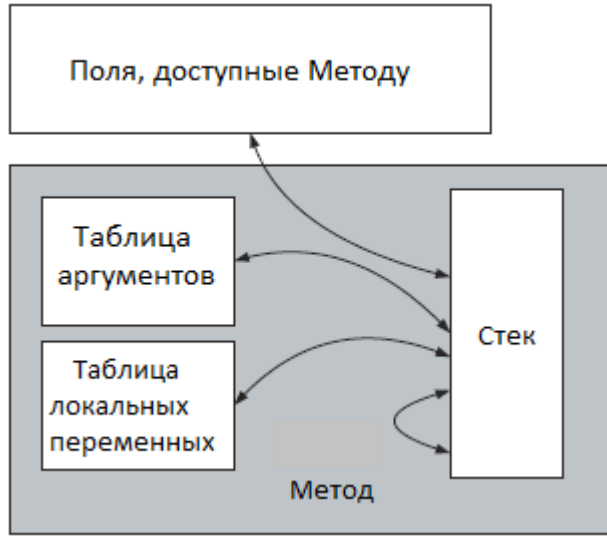
Основные черты архитектуры виртуальной машины CIL таковы:

- Машина является объектно-ориентированной: структура CIL отражает разбиение кода на классы, методы и т.п.
- Машина является стековой. Стек используется только для хранения промежуточных результатов вычисления.
- Ячейки стека представлены как 4-байтовые или 8-байтовые знаковые целые (обозначаемые как I4 и I8);
- Большинство команд CIL получают свои аргументы на стеке, удаляют их со стека и помещают вместо них результаты вычисления.

14.2.1. Память для метода

При возбуждении метода память выделяется для:

- Таблиц аргументов метода.
- Локальных переменных метода.
- Стека, в котором происходят вычисления. **Для вычислений не используются регистры процессора**, так как на уровне кода CIL аппаратная платформа не определена.
- Полей, к которым метод может иметь доступ. Они размещены в динамической памяти - «куче».



Все поля описываются начальным адресом и типами данных, которые в них размещаются.

Стек вычислений состоит из **слотов**, которые в разные моменты времени может содержать данные разных типов. Максимальное число слотов фиксировано (по умолчанию 8).

Область локальных данных является составной частью состояния метода и используется для размещения объектов, тип и/или размер которых неизвестен на этапе компиляции, но которые по тем или иным причинам нежелательно размещать в куче. Область существует ровно столько, сколько исполняется метод, состоянию которого она принадлежит. После прекращения работы метода она автоматически освобождается.

Для хранения локальных переменных и аргументов метода используются два массива, которые, как и стек вычислений, состоят из **слотов**. При этом каждой переменной и каждому аргументу соответствует один слот. Для доступа к локальным переменным и аргументам используются их индексы в массивах. При этом нумерация осуществляется с нуля.

Компилятор, генерирующий CIL-код, не должен делать никаких предположений о том, как переменные и параметры размещены в памяти. Дело в том, что реализации CLI могут любым образом переупорядочивать переменные и параметры, могут произвольно выравнивать их

14.2.2. Система типов CTS

В библиотеке классов использована общая система типов CTS (Common Types System), которая определяет доступные типы для использования во время выполнения. На следующей схеме показана связь различных типов.

Все типы делятся на две категориис

- Типы-значения (value types). Представляют собой примитивные типы данных (целые числа и числа с плавающей запятой). Использование типов-значений всегда связано с копированием их значений.
- Ссылочные типы (reference types). Описывают объектные ссылки (object references), которые представляют собой адреса объектов. Работа со ссылочными типами всегда осуществляется через адреса их значений.

Значения любого типа хранятся в ячейках (location). В качестве ячеек могут выступать локальные и глобальные переменные, параметры методов, поля объектов и элементы массивов. Для каждой ячейки известен тип значений, которые она может содержать.

Особо важным является то обстоятельство, что ячейки не могут содержать объекты. Все объекты размещаются в специальной области памяти, называемой **кучей (heap)**. Таким образом, в ячейках могут храниться только значения типов-значений или объектные ссылки.

14.2.3. Типы в базовых классах .NET, C# и CIL

В таблице показаны соответствия между базовыми классами .NET, ключевыми словами C# и представлениями CIL. Кроме того, приведены сокращенные константные обозначения для каждого типа в CIL. Как будет показано чуть позже, именно на такие константы очень часто ссылаются многие коды операций в CIL.

Базовый класс .NET	Ключевое слово в C#	Представление CIL	Константная нотация CIL
System.SByte	sbyte	int8	I1
System.Byte	byte	unsigned int8	UI
System.Intl6	short	intl6	I2
System. UIntl6	ushort	unsigned intl6	U2
System.Int32	int	int32	I4
System.UInt32	uint	unsigned int32	U4
System.Int64	long	int64	I8
System. UInt64	ulong	unsigned int64	U8
System.Char	char	char	CHAR
System.Single	float	float32	R4

System.Double	double	float64	R8
System.Boolean	bool	bool	BOOLEAN
System.String	string	string	-
System.Object	object	object	-
System.Void	void	void	VOID

14.2.4. Пользовательские типы данных

Как показал опыт платформы Java, которая была разработана задолго до платформы .Net, одной из основных причин ухудшения производительности Java-программ является медленная работа сборщика мусора, вызванная большим количеством мелких объектов в куче. Это явление можно наблюдать в двух случаях:

- Интенсивное создание временных объектов с очень малым временем жизни. Зачастую такие объекты создаются и используются в теле одного метода.
- Использование гигантских массивов объектов, при котором в массиве хранятся ссылки на огромное количество небольших объектов.

Разработчиками .Net был подмечен тот факт, что использование типов-значений вместо объектов позволяет избежать описанных выше проблем, потому что:

- временные значения хранятся не в куче, а непосредственно в локальных переменных метода;
- в массивах типов-значений содержатся не ссылки на значения, а непосредственно сами значения.

Поэтому в общую систему типов были добавлены так называемые пользовательские типы-значения. Эти типы могут быть объявлены программистом, но, как и встроенные типы-значения, размещаются не в куче, а в ячейках. Пользовательские типы-значения делятся на структуры и перечисления.

Структуры являются аналогом классов. Они, как и классы, могут содержать поля, методы, свойства и события. Все структуры неявно наследуют от библиотечного класса System.ValueType, и, более того, встроенные типы-значения также наследуют от этого класса. Система типов не предусматривает никакого наследования структур, кроме данного неявного. Другими словами, структуры не могут наследоваться друг от друга и, тем более, не могут наследоваться от классов (кроме System.ValueType).

Перечисления представляют собой структуры с одним целочисленным полем Value. Кроме того, перечисления содержат набор констант, определяющих возможные значения поля Value. При этом для каждой константы в перечисле-

нии хранится ее имя. Перечисления неявно наследуют от библиотечного класса System.Enum, который, в свою очередь, является наследником все того же класса System.ValueType.

14.2.5. Упакованные типы-значения

Наличие в общей системе типов структур, которые во многом напоминают классы, но в действительности классами не являются, в некоторых случаях вызывает некоторые неудобства. Например, в библиотеке классов .Net существуют достаточно удобные контейнерные классы (наиболее часто используется класс ArrayList, представляющий массив с динамически меняющимся размером). Эти классы могут хранить ссылки на любые объекты, но не могут работать с типами-значениями.

Для решения этой проблемы в общей системе типов предусмотрены так называемые упакованные типы-значения. Эти типы являются ссылочными. Объекты этих типов предназначены для хранения значений типов-значений.

Упакованные типы-значения не могут быть объявлены программистом. Система автоматически определяет такой тип для любого типа-значения.

Получение объекта упакованного типа-значения осуществляется путем упаковки (boxing). **Упаковка** заключается в том, что в куче создается пустой объект нужного размера, а затем значение копируется внутрь этого объекта.

С помощью упаковки мы можем превратить значение любого типа-значения (встроенного примитивного типа, структуры, перечисления) в объект и в дальнейшем работать с этим значением как с настоящим объектом (в том числе, мы можем положить его в ArrayList).

Если же нам требуется произвести обратное действие, мы можем осуществить распаковку (unboxing). **Распаковка** заключается в том, что мы получаем управляемый указатель на содержимое объекта упакованного типа-значения.

14.3. Виртуальная система выполнения

Виртуальная система выполнения (Virtual Execution System – VES) представляет собой абстрактную виртуальную машину, способную выполнять управляемый код. Можно сказать, что виртуальная система выполнения существует только «на бумаге», потому что ни одна из реализаций CLI не содержит интерпретатора CIL-кода (вместо этого используется JIT-компилятор, транслирующий инструкции CIL в команды процессора).

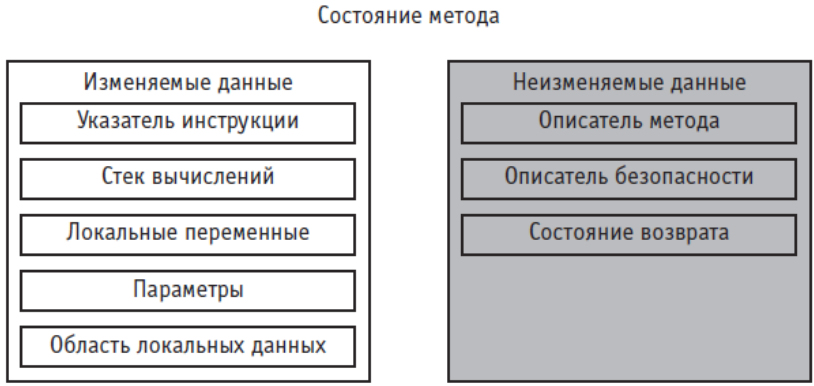
Если сравнить CLI с ее ближайшим конкурентом – платформой Java, можно прийти к выводу, что VES является значительно более абстрактной моделью, чем виртуальная машина Java (Java Virtual Machine – JVM). Причина такого

отличия кроется в том, что изначально Java была ориентирована на реализацию в бытовых приборах. При этом, естественно, подразумевалось, что байт-код Java будет непосредственно выполняться специальными процессорами, и поэтому JVM является фактически спецификацией такого процессора.

Аппаратная реализация VES никогда даже не предполагалась, и это позволило избежать при составлении ее спецификации ненужных деталей, дав тем самым каждой реализации CLI большую свободу выбора наиболее оптимальной стратегии выполнения CIL-кода.

Изучение работы виртуальной машины CLI заключается в том, чтобы понять, что представляет собой состояние виртуальной машины и как это состояние меняется во времени.

На рисунке изображена схема состояния метода. Элементы состояния метода можно условно разделить на две группы: изменяемые данные и неизменяемые данные. Изменяемые данные доступны из тела метода для чтения и записи, в то время как неизменяемые данные либо доступны только для чтения либо вообще предназначены для внутреннего использования в системе выполнения.



Элементы состояния метода, входящие в группу изменяемых данных:

- Указатель инструкции (Instruction Pointer). Содержит адрес следующей инструкции в теле метода, которая будет выполнена системой выполнения. (Когда мы говорим, что указатель инструкции относится к изменяемым данным, мы имеем в виду, что его значение изменяется при переходе от инструкции к инструкции.)
- Стек вычислений (Evaluation Stack). Виртуальная система выполнения работает по принципу стекового процессора. Это означает, что операнды

инструкций, а также возвращаемые инструкциями значения хранятся в специальной области памяти, а именно на стеке вычислений. Каждое состояние метода имеет собственный стек вычислений, содержимое которого сохраняется при вызове методов (то есть, если наш метод вызывает другой метод, то по завершении работы вызванного метода содержимое стека никуда не денется).

- Локальные переменные (Local Variable Array). Для хранения локальных переменных в состоянии метода предусмотрена отдельная область памяти, состоящая из так называемых слотов (slots). Каждой локальной переменной соответствует свой слот. Значения локальных переменных сохраняются при вызове методов аналогично содержимому стека вычислений.
- Параметры (Argument Array). Фактические параметры, переданные методу, записываются в специальную область памяти, которая организована так же, как и область локальных переменных.
- Область локальных данных (Local Memory Pool). В языке CIL предусмотрена инструкция localloc, которая позволяет динамически размещать объекты в области памяти, локальной для метода. Объекты в этой области живут до тех пор пока метод не завершится.

Обратите внимание, что стек вычислений, локальные переменные и параметры, а также локальные данные метода представляют собой логически отдельные области памяти. Каждая конкретная реализация CLI самостоятельно решает вопрос, где размещать эти области.

В группу неизменяемых данных входят элементы состояния метода:

- Описатель метода (methodInfo handle). Содержит сигнатуру метода, в которую входят количество и типы формальных параметров, а также тип возвращаемого значения. Кроме этого, описатель метода включает в себя информацию о количестве и типах локальных переменных и об обработчиках исключений. Описатель метода доступен из кода метода, но в основном он используется системой выполнения при сборке мусора и обработке исключений.
- Описатель безопасности (Security Descriptor). Используется системой безопасности CLI и недоступен из кода метода.
- Состояние возврата (Return State Handle). Служит для организации списка состояний методов внутри системы выполнения и недоступно из кода метода. Фактически представляет собой указатель на состояние метода, из тела которого был вызван текущий метод.

14.4. Стек вычислений

Несмотря на то, что большинство современных процессоров для организации вычислений используют регистры, в виртуальной системе выполнения вместо регистров применяется **стек вычислений**. Это связано, скорее всего, с тем, что стековые вычисления достаточно легко можно отобразить на регистры процессора, так как модель, использующая стек, более абстрактна, чем регистровая модель.

Стек вычислений в VES состоит из слотов. При этом глубина стека (максимальное количество слотов) всегда ограничена и задается статически в заголовке метода. Решение ограничить глубину стека было принято разработчиками спецификации CLI для того, чтобы облегчить создание JIT-компиляторов.

На входе метода стек вычислений всегда пуст. Затем он используется для передачи операндов инструкциям CIL, для передачи фактических параметров вызываемым методам, а также для получения результатов выполнения инструкций и вызываемых методов. Если метод возвращает какое-то значение, то оно кладется на стек вычислений перед завершением метода.

Важной особенностью организации стека вычислений является то обстоятельство, что его слоты не адресуются, то есть мы не можем получить указатель на какой-либо слот и записать в него значение.

Каждый слот стека вычислений может содержать ровно одно значение одного из следующих типов:

- int64 – 8-байтовое целое со знаком;
- int32 – 4-байтовое целое со знаком;
- native int – знаковое целое, разрядность которого зависит от аппаратной платформы (может быть 4 или 8 байт);
- F – число с плавающей точкой, разрядность которого зависит от аппаратной платформы (не может быть меньше 8 байт);
- & – управляемый указатель;
- 0 – объектная ссылка (не 0, а буква);
- Пользовательский тип-значение.

Таким образом, слоты стека вычислений могут иметь различный размер в зависимости от типов записанных в них значений. Также мы можем видеть, что допустимые типы значений для стека вычислений не совпадают с общей системой типов CTS. Например, в CTS существуют целые типы разрядности 1 и 2 байта, которые не могут содержаться на стеке вычислений. И наоборот, тип F стека вычислений не имеет аналога в CTS. Кроме того, для стека вычислений

все управляемые указатели и объектные ссылки отображаются в два типа: & и O соответственно.

Давайте обсудим, как в VES осуществляется работа с типами данных, не поддерживаемыми напрямую стеком вычислений.

- Во-первых, короткие целые типы (bool, char, int8, int16, unsigned int8, unsigned int16) при загрузке на стек вычислений расширяются до int32. При этом знаковые короткие целые типы (int8, int16) расширяются с сохранением знака, а беззнаковые расширяются путем добавления нулевых битов. При сохранении значения со стека вычислений в переменной, параметре, поле объекта или элементе массива происходит обратное сужающее преобразование.
- Во-вторых, беззнаковый тип unsigned int32 при загрузке на стек вычислений становится знаковым int32, и аналогично, беззнаковый unsigned int64 становится знаковым int64. При этом, естественно, никаких преобразований не происходит – просто последовательность бит, которая раньше считалась беззнаковым целым, копируется на стек вычислений.
- В-третьих, типы float32 и float64 при копировании на стек вычислений преобразуются к типу F. Разрядность этого типа определяется конкретной реализацией CLI, которая, однако, должна гарантировать, что точность типа F не ниже, чем точность типа float64.
- В-четвертых, типы-перечисления при копировании на стек вычислений автоматически превращаются в целые типы. Вообще, VES устроена таким образом, что типы-перечисления и целые типы являются совместимыми по присваиванию. Этим они отличаются от обычных типов-значений, которые при копировании на стек сохраняют свой тип и не совместимы с целыми типами.

14.5. Автоматическое управление памятью

Одной из основных особенностей платформы .Net, делающих ее привлекательной для разработки приложений, является механизм автоматического управления памятью, известный как сборка мусора (garbage collection).

Спецификация CLI утверждает, что память для объектов, используемых в программе, выделяется в управляемой куче (managed heap), которая периодически очищается от ненужных объектов сборщиком мусора.

Принцип работы сборщика мусора в спецификации не определен, поэтому разработчики реализаций CLI могут использовать любые алгоритмы, корректно выполняющие очистку управляемой кучи.

В .Net реализован так называемый сборщик мусора с поколениями (generational garbage collector), работающий на основе построения графа достижимости объектов.

Выделение памяти в управляемой куче. Под управляемую кучу резервируется непрерывная область адресного пространства процесса. Система выполнения поддерживает специальный указатель (назовем его HeapPtr), содержащий адрес, по которому будет выделена память для следующего объекта. Когда куча не содержит ни одного объекта, HeapPtr указывает на начало кучи. Выделение памяти для объекта заключается в увеличении HeapPtr на количество байт, занимаемое этим объектом в куче.

14.6. Лексемы в CIL

Набор лексем, которые может распознавать компилятор CIL, по семантическому признаку делится на три отдельных категории:

- директивы CIL;
- атрибуты CIL;
- коды операций CIL.

Лексемы CIL каждой из этих категорий представляются с помощью определенного синтаксиса и затем объединяются для получения полноценной .NET-сборки.

14.6.1. Директивы CIL

В CIL имеется ряд лексем, которые применяются для описания общей структуры .NET-сборки. Эти лексемы называются директивами.

Директивы в CIL позволяют информировать компилятор CIL о том, как ему следует определять пространства имен, типы и члены, которые будут входить в состав сборки. Синтаксически директивы представляются с использованием префикса в виде точки, например

```
.namespace
.class
.assembly.
```

Следовательно, при наличии в файле с расширением *.il (принятое расширение для файлов, содержащих CIL-код) одной директивы .namespace и трех директив .class, компилятор CIL будет генерировать сборку с единственным пространством имен и тремя соответствующими типами классов .NET.

Помимо директив .assembly и .module, существуют и другие CIL-директивы, которые позволяют еще больше уточнять общую структуру создаваемого двоичного файла .NET:

Директива	Описание
.maxstack	Определяет размер виртуального стека, По умолчанию 8.
.namespace	Определяет пространства имен.
.class	Определяет класс.
.module	Определяет модуль.
assembly.	Определяет сборку, включаемую в модуль.
.mresources	Если в сборке будут использоваться внутренние ресурсы (такие как растровые изображения или таблицы строк), с помощью этой директивы можно указывать имя файла, в котором содержатся включаемые в сборку ресурсы.
.subsystem	С помощью этой CIL-директивы можно указывать предпочитаемый пользовательский интерфейс, внутри которого должна выполняться сборка. Например, значение 2 означает, что сборка должна работать в рамках графического интерфейса Windows Forms, а путем значение 3 — что она должна работать как консольное приложение.

14.6.2. Атрибуты CIL

Во многих случаях сами по себе директивы в CIL оказываются недостаточно описательными для того, чтобы полностью выражать определение типа или члена типа .NET. Поэтому нередко директивы еще сопровождаются различными атрибутами CIL, уточняющими, как они должны обрабатываться. Например, директива .class может сопровождаться атрибутом public (уточняющим видимость типа), атрибутом extends (явно указывающим на базовый класс типа) и атрибутом implements (перечисляющим набор интерфейсов, которые должен поддерживать данный тип).

В таблице описана часть атрибутов, которые могут использоваться вместе с директивой .class.

Атрибут	Описание
public private nested assembly nested famandassem nested family nested famorassem nested public	Эти атрибуты применяются для указания степень видимости типа. В CIL предлагается много других возможностей помимо тех, что предлагаются в C#.

nested private	
extends	Явно указывает на базовый класс типа
abstract sealed	Эти два атрибута могут присоединяться к директиве .class для определения, соответственно, абстрактного или герметизированного класса.
auto sequential explicit	Эти атрибуты применяются для указания CLR-среде, каким образом, она должна размещать данные полей в памяти. Для типов классов используемый по умолчанию флаг auto является вполне подходящим.
extends implements	Эти атрибуты позволяют, соответственно, определять базовый класс для типа и реализовать для него интерфейс.

Коды операций CIL. После определения сборки, пространства имен и набора типов на CIL с помощью различных директив и соответствующих типов, напоследок остается предоставить для каждого из типов логику реализации. Для этого применяются коды операций (operation codes — opcodes). Как и в других низкоуровневых языках программирования, коды операций в CIL обычно имеют непонятный и нечитабельный вид. Например, для определения строковой переменной в CIL нужно использовать не удобный для восприятия код операции LoadString, а код операции ldstr.

Для каждого двоичного кода операции в CIL существует соответствующий мнемонический эквивалент. Например, вместо кода 0X58 может использоваться его мнемонический эквивалент add, вместо кода 0X59 — мнемонический эквивалент sub.

14.7. Коды операций в CIL

Кодом операции называется лексема, которая в CIL служит для построения логики реализации отдельно взятого члена. Все поддерживаемые в CIL коды операций можно в общем поделить на три следующих основных категории:

- коды операций, позволяющие управлять выполнением программы;
- коды операций, позволяющие вычислять выражения;
- коды операций, позволяющие получать доступ к значениям в памяти (через параметры, локальные переменные и т.д.).

Чтобы можно было получить общее представление о реализации членов в CIL, в таблице приведены некоторые наиболее полезные кодщ операций, имеющие непосредственное отношение к логике реализации членов (с группированием по функциональности).

Операция	Описание

add sub mul div rem	Сложение Вычитание Умножение Деление Остаток от деления
and or not xor	И ИЛИ НЕ Исключающее ИЛИ
ceq cgt clt	Сравнение на равенство Сравнение на больше Сравнение на меньше
box unbox	Упаковка, тип-значение ==> ссылочный тип Упаковка, ссылочный тип ==> тип-значение
ret	Выхода из метода и возврат значения вызывающему коду
beq bgt Ыe bit switch	Переход по равно Переход по больше Переход по меньше или равно Переход по меньше Переход по таблице переходов по значению на верхушке стека
call	Вызов члена определенного типа
nearer newobj	Размещать в памяти новый массив Размещать в памяти новый объект

Коды операций из следующей обширной категории (часть которых приведена в таблице) применяются для загрузки (заталкивания) аргументов в виртуальный стек выполнения. Важно обратить внимание, что все эти ориентированные на выполнение загрузки коды операций сопровождаются префиксом ld (означает load - загрузка).

Операция	Описание
ldarg	Загружать в стек аргумент метода
ldc	Загружать в стек значение константы
ldf	Загружать в стек значение поля
ldloc	Загружать в стек значение локальной переменной
ldobj	Получать все значения размещаемого в куче объекта и помещать их в стек
dstr	Загружать в стек строковое значение

Помимо ряда связанных с выполнением загрузки кодов операций в CIL еще поддерживаются и такие коды операций, которые позволяют явным образом извлекать из стека самое верхнее значение. Как уже показывалось в несколь-

ких приводившихся ранее в этой главе примерах, извлечение значения из стека обычно подразумевает его сохранение во временном локальном хранилище для дальнейшего использования (например, параметра для последующего вызова метода). Из-за этого многие из кодов операций, которые позволяют извлекать текущее значение из виртуального стека выполнения, сопровождаются префиксом st (store— сохранить). В таблице перечислены некоторые наиболее часто используемые из них.

Операция	Описание
pop	Удалить значение, которое в текущий момент находится в самом верху стека вычислений, но не сохранять его
starg	Сохранить самое верхнее значение из стека в аргументе метода с определенным индексом
stloc	Извлекать текущее значение из самой верхней части стека вы и сохранять его в списке локальных переменных с определенным индексом.
stobj	Копировать значение определенного типа из стека вычислений в память по определенному адресу
stsfld	Заменять значение статического поля значением из стека вычислений

Следует также обязательно знать о том, что различные коды операций в CIL могут подразумевать неявное извлечение значений из стека во время выполнения задач. Например, при попытке выполнить операцию вычитания одного числа из другого с помощью кода операции sub должно быть очевидным, что sub придется извлечь два следующих доступных значения, прежде чем выполнить вычисление. По окончании процесса вычисления результат будет помещен обратно в стек.

14.7.1. Команды загрузки

Это некоторые команды загрузки подробнее.

Формат	Действие
ldimm <число>	Загрузка константы
ldstr <строка>	Загрузка строковой константы
ldsflda <поле>	Загрузка адреса статического поля
ldloca <#переменной>	Загрузка адреса локальной переменной
ldflda <поле>	Загрузка адреса поля объекта
ldind	Косвенная загрузка, берет адрес со стека и помещает на его место значение, размещенное по этому адресу

Поскольку, как правило, необходим не адрес переменной, а ее значение, то существуют команды загрузки значения на стек: ldsfld, ldloc, ldfld. Каждая из этих команд эквивалентна паре команд ldxxxa; ldind.

14.7.2. Команды выгрузки

Команды выгрузки в основном построены так же, как и команды загрузки (только с противоположным результатом работы), и потому не особо нуждаются в комментариях. Это некоторые команды выгрузки подробнее.

Формат	Действие
stind	Берет со стека адрес значения и само значение и записывает значение по выбранному адресу.
stloc Stfld stsfld	Команды эквивалентны парам ldxxxa; stind

14.7.3. Вычислительные команды

Команды CIL позволяют выполнять вычисления. Все они берут аргументы со стека и кладут на их место результат.

- Арифметические команды (add, mul, sub…) существуют в знаковом и беззнаковом (.u) вариантах, а также могут выполнять контроль за переполнением (.ovf).
- Логические команды and, or,xor (только знаковые, без контроля переполнения).
- Операции сравнения.

14.7.4. Арифметические инструкции

Это некоторые команды подробнее.(в скобках двоичное представление)

Манипуляции со стеком.

Формат	Действие
nop (0x00)	Нет операции.
dup (0x25)	Дублирует содержимое верхушки стека
pop (0x25)	Удаляет содержимое верхушки стека

Загрузка констант. Константы могут быть десятичными или 16-ричными.

Формат	Действие
ldc.i4 –1	Загрузка константы -1
ldc.i4 0xFFFFFFFF	Загрузка 16-ричной константы 0xFFFFFFFF
ldc.i4 <int32> (0x20).	Загрузка 4-байтовой константы в стек размером 32

	бита
ldc.i4 <int8> (0x20).	Загрузка 4-байтовой константы в стек размером 32 бита

Загрузка. По адресу, взятому из указателя стека, осуществляется загрузка в верхушку стека.

Формат	Действие
ldind.i1 (0x46).	Загрузка знакового 1-байтового целого
ldind.u1 (0x46).	Загрузка беззнакового 1-байтового целого
ldind.i2 (0x46).	Загрузка знакового 2-байтового целого
ldind.u2 (0x46).	Загрузка беззнакового 2-байтового целого
ldind.i4 (0x46).	Загрузка знакового 4-байтового целого
ldind.u4 (0x46).	Загрузка беззнакового 4-байтового целого
ldind.i8 (0x46).	Загрузка знакового 8-байтового целого
ldind.u8 (0x46).	Загрузка беззнакового 8-байтового целого
ldind.i (0x46).	Загрузка целого в формате платформы
ldind.r4 (0x46).	Загрузка числа в формате single
ldind.r8(0x46).	Загрузка числа в формате double
ldind.ref(0x46).	Загрузка ссылки на объект

Сохранение. По адресу, взятому из указателя стека, осуществляется сохранение из верхушки стека.

Формат	Действие
stind.i1 (0x46).	Сохранение 1-байтового целого
stind.i2 (0x46).	Сохранение 2-байтового целого
stind.i4 (0x46).	Сохранение 4-байтового целого
stind.i8 (0x46).	Сохранение 8-байтового целого
stind.i (0x46).	Сохранение целого в формате указателя
stind.r4 (0x46).	Сохранение числа в формате single
stind.r8(0x46).	Сохранение числа в формате double
stind.ref(0x46).	Сохранение ссылки на объект

Команды целочисленной арифметики существуют в знаковом и беззнаковом (с суффиксом .u) вариантах и могут быть записаны с суффиксом обработки переполнения (.ovf), который порождает исключение при возникновении переполнения. К этим командам относятся: ADD, SUB, MUL, DIV, MOD.

Арифметические операции. Все, кроме инверсии, используют два операнда из стека и помещают результат в верхушку стека.

Формат	Действие
add (0x46).	Сложение

sub (0x46).	Вычитание
mul (0x46).	Перемножение
div (0x46).	Деление
div.un (0x46).	Деление целочисленное
rem (0x46).	Остаток от деления
rem.un (0x46).	Остаток от деления целочисленного
neg (0x46).	Смена знака числа

Арифметические операции с переполнением. Подобны операциям без переполнения, но дополнительно запускают исключение переполнения Overflow Exception, если результат операции не согласуется с допустимым. Код команды включает фрагмент <.ovf> после имени команды.

Операции побитовые. Осуществляют побитовые логические операции над операндами.

Формат	Действие
and (0x46).	Операция И - AND
or (0x46).	Операция Или - OR
xor (0x46).	Операция Исключающее ИЛИ - XOR
not (0x46).	Операция Не -Not

Операции сдвига. Осуществляют побитовые операции сдвига над целочисленными операндами.

Формат	Действие
shl (0x46).	Сдвиг влево
shr (0x46).	Сдвиг вправо. Левый бит (знака) фиксирован
shr.un (0x46).	Сдвиг вправо для чисел без знака. Левый бит принимает значение 0

Операции преобразования. Осуществляют преобразование типа операнда.

Формат	Действие
conv.i1 (0x46).	Преобразование в int8
conv.u1 (0x46).	Преобразование в int8 без знака
conv.i2 (0x46).	Преобразование в int16
conv.u2 (0x46).	Преобразование в int16 без знака
conv.i4 (0x46).	Преобразование в int32
conv.u4 (0x46).	Преобразование в int32 без знака
conv.i8 (0x46).	Преобразование в int64
conv.u8 (0x46).	Преобразование в int64 без знака
conv.i (0x46).	Преобразование в int платформы

conv.u (0x46).	Преобразование в int64 платформы без знака
conv.r4 (0x46).	Преобразование в float32
conv.r8 (0x46).	Преобразование в float64
conv.r (0x46).	Преобразование целого без знака в float

Операции преобразования с переполнением. Подобны операциям без переполнения, но дополнительно запускают исключение переполнения Overflow Exception, если результат операции не согласуется с допустимым. . Код команды включает фрагмент <.ovf> после имени команды.

Формат	Действие
conv.ovf.i1 (0x46).	Преобразование в int8
conv.ovf.u1 (0x46).	Преобразование в int8 без знака
conv.ovf.i1.un (0x46).	Преобразование целого без знака в int8
conv.ovf.u1.un (0x46).	Преобразование целого без знака в int8 без знака

В IL есть некоторый набор операций сравнения. Эти операции снимают со стека операнды и помещают на их место результат 0 или 1. Они могут быть беззнаковыми или знаковыми (с суффиксом .s). Кроме того, существуют специальные варианты сравнения, учитывающие возможность сравнения чисел с плавающей запятой различного порядка (такие операции имеют суффикс .un).

Интересно отметить, что при наличии полного комплекта операций перехода, создатели IL не включили в систему команд операций сравнения "<=" и ">=". Это приводит к тому, что для целочисленных значений операцию "<=" приходится эмулировать с помощью следующего набора команд: cgt; ldc.i4.0; ceq

Соответственно, для вещественных значений операцию "<=" необходимо представлять аналогично, только первая команда должна быть заменена на cgt.un. Тем не менее, с точки зрения конечной программы в машинных кодах это, видимо, несущественно, так как такой набор операций легко оптимизировать в одну ассемблерную команду целевой архитектуры.

14.7.5. Переходы и вызовы в IL

Переходы в .NET мало чем отличаются от используемых в обычных ассемблерах. Все команды переходов существуют в стандартном и коротком виде (для записи коротких переходов используется суффикс .s). Помимо обычного безусловного перехода (br), в IL существует целый ряд условных переходов (beq, bne, bgt,brfalse – переход по false, null или нулю на верхушке стека – и все прочие переходы, включая беззнаковые и неупорядоченные варианты).

Существует две основных команды вызова:

- вызов статического метода (call)

- вызов виртуального метода (callvirt)

Если вызывается метод экземпляра объекта, то объект, которому он принадлежит, должен быть первым параметром; для callvirt этот параметр обязателен, поскольку виртуальных статических методов в .NET не бывает.

Команда вызова может быть снабжена префиксом tail. Это означает, что значение, возвращаемое вызываемой процедурой, является также возвращаемым значением и для вызывающей процедуры. В таком случае можно превратить вызов процедуры с последующим возвратом значения в одну команду безусловного перехода на вызываемую процедуру; для этого также необходимо удалить текущую рамку стека. Эта оптимизация позволяет избежать разрастания стека во многих рекурсивных алгоритмах. Недостатком такой оптимизации являются трудности отслеживания стека вызовов при отладке.

Возврат осуществляется командой ret, которая для методов, не возвращающих результат, не имеет параметров. Для всех прочих методов эта команда ожидает параметр – возвращаемое значение на верхушке стека.

14.8. Трансляция в CIL

Продемонстрируем трансляцию в CIL на примере программы на C#, вычисляющей числа Фибоначчи:

Исходный текст на C#

```
using System;
class Fib // числа Фибоначчи
    {
        public static void Main (String[] args)
            {
                int a = 1, b = 1;
                for (int i = 1; i != 10; ++i)
                {
                    Console.WriteLine (a);
                    int c = a + b;
                    a = b;
                b = c;
                }
            }
    }
```

Результаты трансляции этой программы в IL.

```
// объявление имени assembly
    .assembly fib as "fib" { /* здесь могут быть параметры */ }
```

```
.class public Fib
{
    .method public static void Main ()
    {
    .entrypoint                 // означает начало assembly
        .locals (int32 a, int32 b)
        ldc.i4.1                // загрузка константы 1
        stloc   a               // сохранение 1 в a (a = 1)
        ldc.i4.1
        stloc   b               // аналогично: b = 1
        ldc.i4.1                // загрузка 1 на стек (счетчик цикла)
        Loop:
        ldloc   a
        call    void System.Console::WriteLine(int32)
        ldloc   a               // stack: 1 a
        ldloc   b               // stack: 1 a b
        add                     // stack: 1 (a+b)
        ldloc   b
        stloc   a               // a = b
        stloc   b               // b = (a+b)
        ldc.i4.1
        add                     // инкремент счетчика
        dup
        ldc.i4.s    10
        bne.un.s    Loop    // сравнение и переход на новую итерацию
        pop                     // удаление счетчика цикла со стека
        ret
    }
}
```

Программа на CIL начинается с объявления имени сборки, в которую входит данная программа. Используется директива .assembly.

Затем объявляется класс Fib, в котором производятся вычисления. Используется директива .class.

Затем объявляется метод Main(), с которого начинается исполнение кода. Используется директива .method с атрибутами.

Внутри Main находится основная точка входа в сборку (директива .entrypoint).

Затем объявляются локальные переменные; отметим, что в процессе реальной трансляции имена этих переменных будут утеряны.

Наконец, происходит инициализация переменных, подготовка к началу цикла (загрузка счетчика цикла на стек) и выполнение основных вычислений программы:

- печать очередного числа Фибоначчи,
- загрузка рабочих переменных на стек,
- их сложение, присваивание результатов и увеличение счетчика.

Затем происходит сравнение счетчика цикла с максимальным значением цикла и в случае выполнения неравенства "счетчик не равен 10" происходит переход на начало цикла. По окончании цикла происходит удаление счетчика цикла со стека и выход из метода.