

ВВЕДЕНИЕ

Платформа .NET, ее назначение и структура. Обзор технологий .NET

В 2000 году компания Microsoft объявила о создании нового языка программирования – языка C#. Эта акция стала частью более значительного события: объявления о платформе .NET (.NET Framework). Платформа .NET, по сути, представляла собой новую модель создания приложений, включающую в себя следующие возможности:

- 1) использование библиотеки базовых классов, предлагающих целостную объектно-ориентированную модель программирования для всех языков программирования, поддерживающих .NET;
- 2) полное и абсолютное межязыковое взаимодействие, позволяющее разрабатывать фрагменты одного и того же проекта на различных языках программирования;
- 3) общую среду выполнения приложений .NET, независимо от того, на каких языках программирования для данной платформы они были созданы; при этом среда берет на себя контроль за безопасностью выполнения приложений и управление ресурсами;
- 4) упрощенный процесс развертывания приложений, в результате чего процесс установки может свестись к простому копированию файлов приложения в определенный каталог.

Одним из основных элементов .NET Framework является библиотека классов под общим именем FCL (Framework Class Library), к которой можно обращаться из различных языков программирования, в частности, из C#. Эта библиотека разбита на модули таким образом, что имеется возможность использовать ту или иную ее часть в зависимости от требуемых результатов. Так, например, в одном из модулей содержатся "кирпичики", из которых можно построить Windows-приложения, в другом – "кирпичики", необходимые для организации работы в сети и т.д.

Кроме FCL, в состав платформы .NET входит Common Language Runtime (CLR – единая среда выполнения программ), название которой говорит само за себя – эта среда ответственна за поддержку выполнения всех типов приложений, разработанных на различных языках программирования с использованием библиотек .NET.

Замечания

Следует отметить, что основными языками, предназначенными для платформы .NET Framework, являются C#, VB.NET, Managed C++ и JScript .NET. Для данных языков Microsoft предлагает собственные компиляторы, переводящие программу в специальный код, называемый IL-кодом, который выполняется средой CLR.

Кроме Microsoft, еще несколько компаний и академических организаций создали свои собственные компиляторы, генерирующие код, работающий в CLR. На сегодняшний момент известны компиляторы для Pascal, Cobol, Lisp, Perl, Prolog и т.д. Это означает, что можно написать программу, например, на языке Pascal, а затем, воспользовавшись соответствующим компилятором, создать специальный код, который будет работать в среде CLR.

Процесс компиляции и выполнения программы в среде CLR более подробно будет рассмотрен позже.

Среда CLR берет на себя всю низкоуровневую работу, например, автоматическое управление памятью. В языках программирования предыдущих поколений управление ресурсами, в частности, управление памятью, являлось одной из важных проблем. Объекты, созданные в

памяти, должны были быть удалены из нее, иначе память будет исчерпана и программа не сможет продолжить выполнение. При этом программисты часто просто забывали удалять неиспользуемые объекты.

При разработке платформы .NET эту проблему постарались решить. Теперь управление памятью берет на себя среда CLR. В процессе работы программы среда следит за объектами и автоматически уничтожает неиспользуемые.

Замечание

Система управления памятью называется Garbage Collector (GC).

Среда CLR обеспечивает интеграцию языков и позволяет объектам, созданным на одном языке, использовать объекты, написанные на другом. Такая интеграция возможна благодаря стандартному набору типов и информации, описывающей тип (метаданным). Интеграция языков очень сложная задача, так как некоторые языки не учитывают регистры символов, другие не поддерживают методы с переменным числом параметров и т.д. Чтобы создать тип, доступный для других языков, придется задействовать лишь те возможности языка, которые гарантированно доступны в других языках. С этой целью Microsoft разработал:

- 1) общую систему типов (Common Type System, CTS), которая описывает все базовые типы данных, поддерживаемые средой CLR, и определяет, как эти типы будут представлены в формате метаданных .NET.
- 2) общезыковую спецификацию (Common Language Specification, CLS), описывающую минимальный набор возможностей, который должен быть реализован производителями компиляторов, чтобы их продукты работали в CLR, а также определяющую правила, которым должны соответствовать видимые извне типы, чтобы к ним можно было получить доступ из любых других CLS-совместимых языков программирования.

Важно понимать, что система CLR/CTS поддерживает гораздо больше возможностей для программиста, чем спецификации CLS (см. Рис. 1).

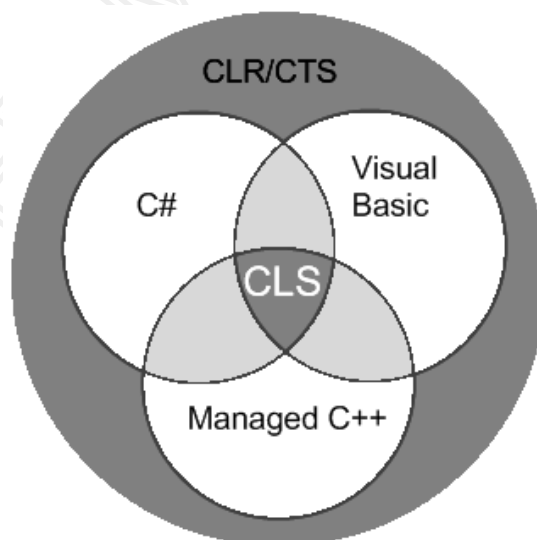


Рис. 1. Схема пересечения возможностей языков

Если при разработке какого-либо типа требуется, чтобы он был доступен другим языкам, нельзя использовать возможности своего языка, выходящие за рамки возможностей, определяемых CLS. Иначе созданный тип может оказаться недоступным программистам,

пишущим код на других языках. Если межъязыковое взаимодействие не требуется, то можно разрабатывать очень мощные типы, ограничиваясь лишь возможностями языка.

Принцип компиляции и выполнения программы в среде CLR. Управляемый и неуправляемый код

Создание приложений с помощью .NET Framework означает написание программы на любом языке программирования, который поддерживается этой платформой. Для того чтобы написанная, например, на C# программа была выполнена, ее необходимо преобразовать в программу на языке, «понятном» компьютеру (исполняемый код). Такой процесс преобразования называется компиляцией, а программа, которая его выполняет, компилятором. В прошлом почти все компиляторы генерировали код для конкретных процессорных архитектур. При разработке платформы .NET от этой зависимости постарались избавиться. Для этого ввели двухшаговую компиляцию.

На первом этапе все .NET компиляторы генерируют промежуточный код на языке Intermediate Language (IL — промежуточный язык) или IL-код. IL-код не является специфичным ни для какой операционной системы и ни для какого языка программирования. Он может быть выполнен в любой среде, для которой реализована CLR-система.

На втором этапе IL-код переводится в код, специфичный для конкретной операционной системы и архитектуры процессора. Эта работа возлагается на JIT-компилятор (Just In Time compiler – компилирование точно к нужному моменту). Только после этого операционная система может выполнить приложение.

Замечание

JIT-компилятор входит в состав среды CLR.

IL-код, выполняемый под управлением CLR, называется управляемым (managed). Это означает, что среда CLR полностью управляет жизненным циклом программы: отслеживает безопасность выполнения команд программы, управляет памятью и т.д. Это, несомненно, является достоинством управляемого кода. Конечно, использовать приложения, разработанные на основе управляемого кода, можно только тогда, когда на компьютере установлена .NET Framework.

Использование IL-кода имеет и обратную сторону – поддержка любой платформы означает отказ от функциональности, специфичной для конкретной платформы. Обойти это ограничение позволяет использование неуправляемого кода (unmanaged), т.е. кода, который не контролируется CLR и выполняется самой операционной системой. Такой код приходится использовать при необходимости обращения к низкоуровневым функциям операционной системы (например, понятие реестра существует только в Windows и функции, работающие с реестром, приходится вызывать из операционной системы). Иногда использование неуправляемого кода позволяет ускорить выполнение некоторых алгоритмов.

Назначение и возможности Visual Studio .NET

В рамках данного курса мы будем изучать язык C# – один из языков программирования, который может использоваться для создания приложений, выполняемых в среде CLR. Этот язык был создан компанией Microsoft специально для использования на платформе .NET.

В общем случае создавать файлы с исходным кодом на языке C# можно с помощью обычного текстового редактора, например, Блокнота. Затем необходимо будет скомпилировать их в

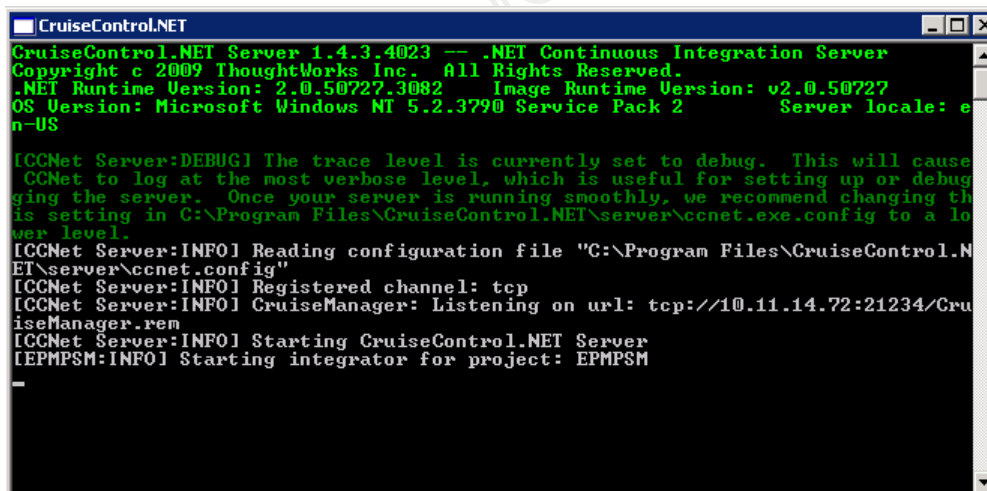
управляемый код через командную строку. Однако наиболее удобно для этих целей использовать среду Visual Studio .NET (VS), потому что:

- 1) VS автоматически выполняет все шаги, необходимые для создания IL-кода;
- 2) текстовый редактор VS изначально настроен для работы с теми .NET языками, которые были разработаны Microsoft, в том числе C#, поэтому он может интеллектуально обнаруживать ошибки и «подсказывать» в процессе ввода, какой именно код можно использовать на данном этапе разработки (технология IntelliSense);
- 3) в состав VS входят средства, позволяющие создавать Windows и Web-приложения путем простого перетаскивания мышью элементов пользовательского интерфейса.
- 4) многие типы проектов, создание которых возможно на C#, могут разрабатываться на основе готовых шаблонов проектов. Вместо того чтобы каждый раз начинать с нуля, VS позволяет использовать уже имеющиеся файлы с исходным кодом, что уменьшает временные затраты на создание проекта.

Перечислим некоторые типы приложений, которые позволяет создавать VS.

Console Application

Приложения данного типа позволяют выполнять вывод на «консоль», то есть, в окно командного процессора. Данный тип приложений существует со времен операционных систем с текстовым пользовательским интерфейсом, например MS-DOS. Тем не менее, консольные приложения продолжают активно использоваться и в наши дни. Их применение может быть связано с отсутствием необходимости в графическом интерфейсе. Например, утилиты автоматической компиляции приложений, как правило, выполняются в заранее назначенные интервалы времени без участия пользователя (см. Рис. 2).



```
CruiseControl.NET
CruiseControl.NET Server 1.4.3.4023 -- .NET Continuous Integration Server
Copyright c 2009 ThoughtWorks Inc. All Rights Reserved.
.NET Runtime Version: 2.0.50727.3082 Image Runtime Version: v2.0.50727
OS Version: Microsoft Windows NT 5.2.3790 Service Pack 2 Server locale: e
n-US

[CCNet Server:DEBUG] The trace level is currently set to debug. This will cause
CCNet to log at the most verbose level, which is useful for setting up or debug
ging the server. Once your server is running smoothly, we recommend changing th
is setting in C:\Program Files\CruiseControl.NET\server\ccnet.exe.config to a lo
wer level.
[CCNet Server:INFO] Reading configuration file "C:\Program Files\CruiseControl.N
ET\server\ccnet.config"
[CCNet Server:INFO] Registered channel: tcp
[CCNet Server:INFO] CruiseManager: Listening on url: tcp://10.11.14.72:21234/Cru
iseManager.rem
[CCNet Server:INFO] Starting CruiseControl.NET Server
[EPMPMSM:INFO] Starting integrator for project: EPMPMSM
```

Рис. 2. Процедура запуска системы контроля кода проекта

Еще одним вариантом применения консольного ввода/вывода является встраивание его в программы с графическим интерфейсом. Дело в том, что современные программы содержат очень большое число команд, значительная часть которых никогда не используется обычными пользователями. В то же время, эти команды должны быть доступны в случае необходимости. Ярким примером использования данного подхода являются компьютерные игры.



Рис. 3. Пример использования консоли в игре

Windows Forms Application

Приложения данного типа используют элементы графического оконного интерфейса, включая формы, кнопки, флажки и т.д. Приложения такого типа более удобны для пользователя, так как позволяют ему отдавать команды щелчком мыши, а не ручным вводом команд, что позволяет значительно повысить скорость работы по сравнению с консольными приложениями. Типичным примером приложения, построенного с применением графического интерфейса, является MS Word (см. Рис. 4).

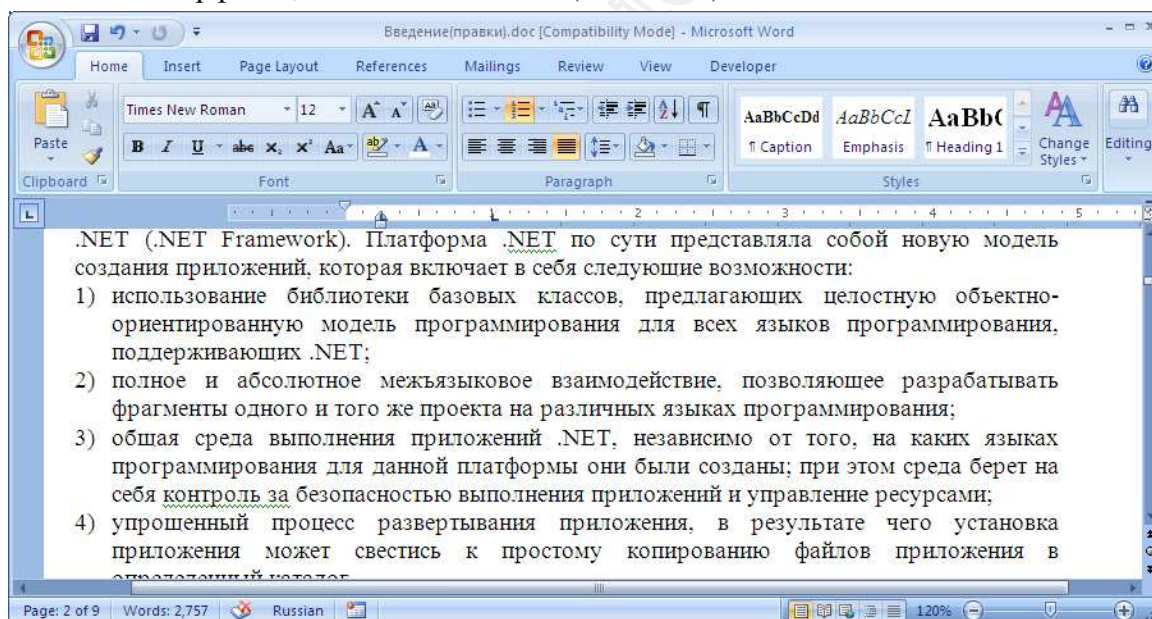


Рис. 4. Графический интерфейс пользователя на примере MS Word 2007

Web Application

Приложения данного типа представляют собой набор web-страниц, которые могут просматриваться любым web-браузером. Web-приложения позволяют создавать многопользовательские системы с единообразным интерфейсом, практически не зависящим от установленных у пользователя операционной системы и остальных программ. Пример типичного web-приложения представлен на Рис. 5.

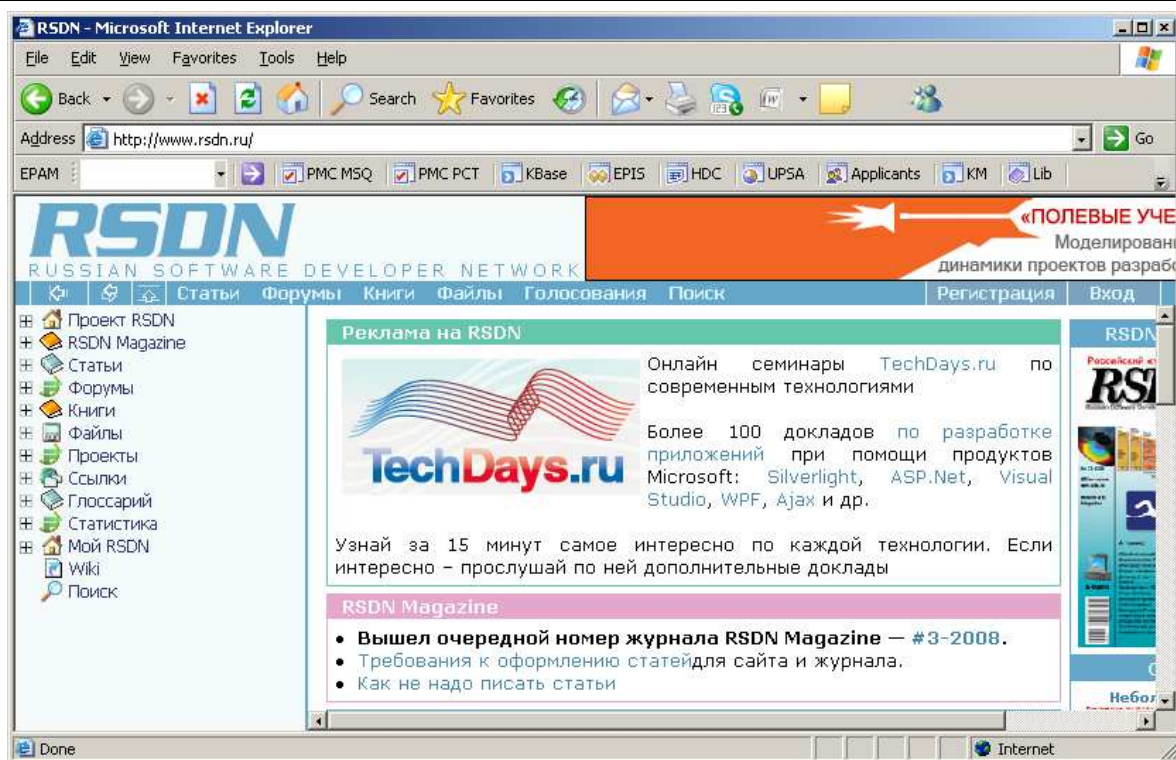


Рис. 5. Пример web-приложения

Web Service

Приложения данного типа представляют собой специализированные web-приложения, предназначенные для предоставления другим приложениям доступа к своим данным. В сети Интернет представлено довольно много различных сервисов: прогнозы погоды, котировки валют и др. Использование сервисов позволяет с минимальными затратами получать доступ к актуальной информации, которой владеют люди, не имеющие ни малейшего отношения к разрабатываемой нами программе.

Class Library

Приложения данного типа представляют собой библиотеки, содержащие классы и методы. Библиотеки не являются полноценными самостоятельными приложениями, но могут использоваться в других программах. Как правило, в библиотеки помещают алгоритмы и структуры данных, которые могут быть полезны более чем одному приложению.

Специфика платформы .NET такова, что она «подходит» для разработки «офисных» приложений, Web-приложений, сетевых приложений и приложений для мобильных устройств. В то же время она не предназначена для создания операционных систем и драйверов.

В рамках данного курса мы рассмотрим основы программирования на языке C#, разрабатывая консольные приложения в среде Visual Studio .NET (VS).

Приложение, находящееся в процессе разработки, называется проектом (project). Несколько проектов могут быть объединены в решение (solution). Совсем не обязательно, чтобы проекты в решении были одного типа. Например, в одно решение могут быть объединены Web-приложение и библиотеки, которые оно использует.

Сначала мы будем создавать решение, состоящее из одного проекта.

Создание первого проекта в среде Visual Studio

Для создания проекта следует запустить VS, а затем в главном меню VS выбрать команду **File – New – Project**. После этого откроется диалоговое меню **New Project** (см. Рис. 6.)

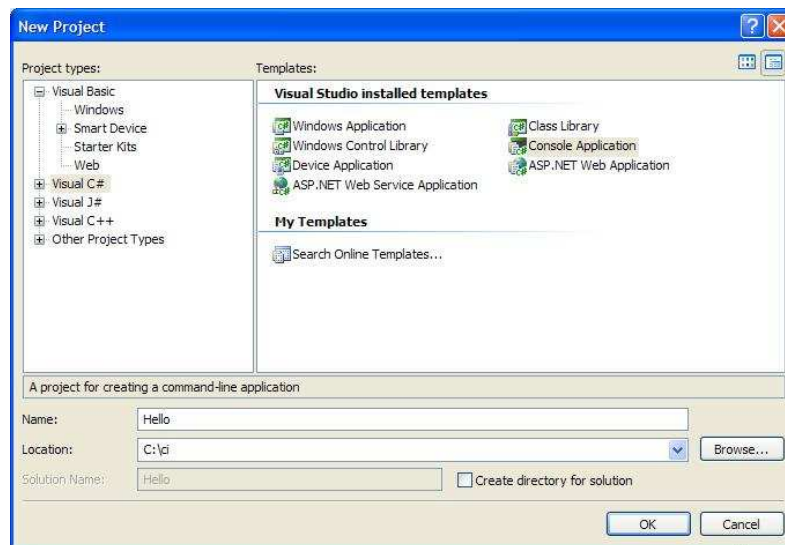


Рис. 6. Диалоговое окно New Project

В поле **Project types** следует выбрать **Visual C#**, а в поле **Templates** – **Console Application**.

В строчке **Name** введите имя приложения: **Hello**. Обратите внимание на то, что это же имя появится в строчке **Solution Name**. Уберите галочку в поле **Create directory for solution** (пока мы создаем простое приложение, и нам нет необходимости усложнять его структуру).

В строчке **Location** определите положение на диске, куда нужно сохранять ваш проект, и нажмите кнопку **OK**. Примерный вид экрана изображен на Рис. 7.

Замечание

В зависимости от версии VS и от того, как в данный момент настроена среда, вид экрана может немного отличаться. Как это, так и все дальнейшие описания среды разработки подразумевают, что среда настроена по умолчанию.

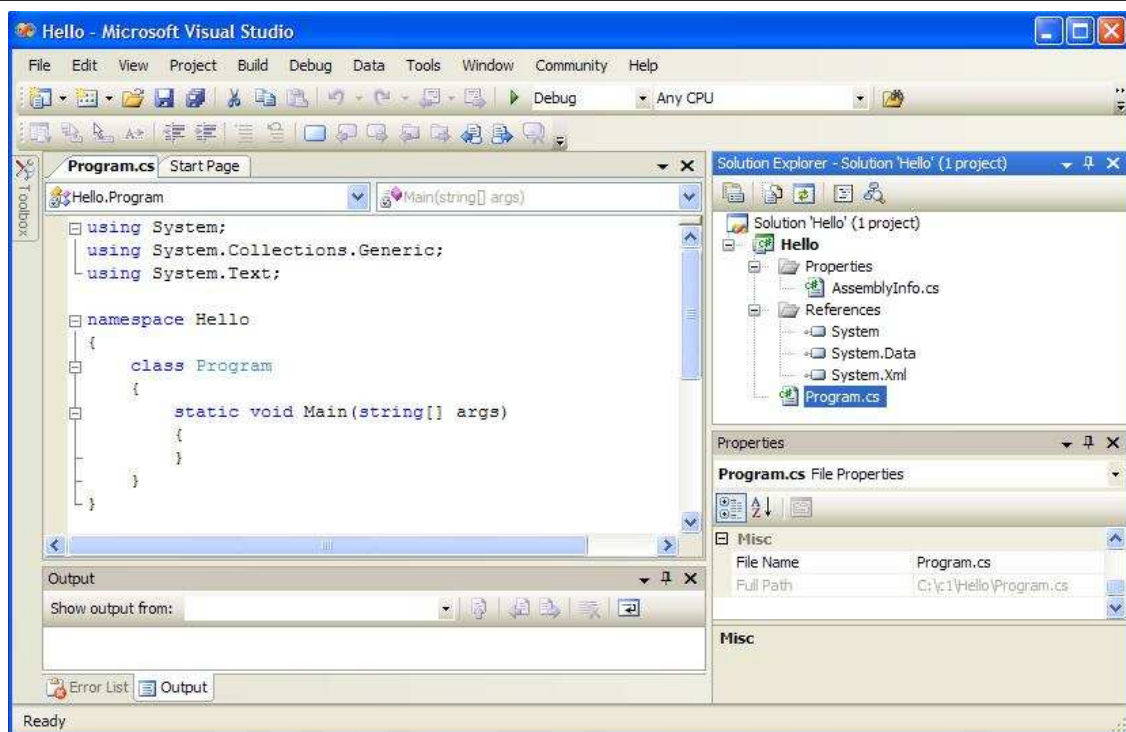



Рис. 7. Рабочая область VS

В правой верхней части экрана располагается окно управления проектом ***Solution Explorer***. Если оно закрыто, то его можно включить командой ***View – Solution Explorer***. В этом окне перечислены все ресурсы, входящие в проект:

- 1) ***AssemblyInfo.cs*** – информация о *сборке*. Компилятор в качестве результата своего выполнения создает *сборку* – файл с расширением *exe*, или *dll*, который содержит IL-код.
- 2) ***System, System.Data, System.Xml*** – ссылки на стандартные библиотеки.
- 3) ***Program.cs*** – текст программы на языке C#.

В правой нижней части экрана располагается окно свойств ***Properties***. Если оно закрыто, то его можно включить командой ***View – Properties***. В этом окне отображаются основные характеристики выделенного элемента.

Основное пространство экрана занимает окно редактора, в котором располагается текст программы, созданный средой автоматически. Текст представляет собой каркас, в который программист будет добавлять нужный ему код и изменять его. При этом зарезервированные (ключевые) слова отображаются синим цветом, комментарии – зеленым, основной текст – черным. Текст структурирован. Щелкнув на знаке минус в первой колонке текста, мы скроем блок кода, щелкнув на знаке плюс – откроем.

Для того чтобы запустить приложение, нужно нажать на кнопку ***Start*** () стандартной панели инструментов VS. В результате программа скомпилируется в IL-код и этот код будет передан CLR на выполнение. Программа запустится, и, так как в данный момент наш код не содержит никаких команд, тут же завершит свою работу. Визуально это будет выглядеть как быстро мелькнувший на экране черный прямоугольник.

Откройте папку, содержащую проект, и изучите ее структуру. На данном этапе особый интерес для нас будут представлять следующие файлы:

- 1) *Hello.sln* – основной файл, отвечающий за всё решение. Если необходимо открыть решение для работы, то нужно выбрать именно этот файл. Остальные файлы откроются автоматически. В нашем случае имеется только один проект (*Hello.csproj*), поэтому при открытии решения откроется только он.
- 2) *Program.cs* – файл, в котором содержится исходный код, написанный на языке C#. Именно с этим файлом мы и будем непосредственно работать.
- 3) *Hello\bin\Debug\Hello.exe* – файл, в котором содержится сгенерированный IL-код проекта. Другими словами, этот файл и есть готовое приложение, которое может выполняться на любом компьютере, на котором установлена платформа .NET.

Теперь рассмотрим сам текст программы.

using System – это директива, которая разрешает использовать имена стандартных классов из пространства имен **System** непосредственно, без указания имени пространства, в котором они были определены. Так, например, если бы этой директивы не было, то нам пришлось писать бы `System.Console.WriteLine` (назначение данной команды мы рассмотрим позже). Конечно, писать полное пространство имен каждый раз очень неудобно. При указании директивы `using` можно писать просто имя, например, `Console.WriteLine`.

Замечание

У разработчиков, хорошо знакомых с Delphi, может возникнуть аналогия между использованием using и знакомым по этой среде разработке оператором with. Это не так. Оператор using применим только для пространств имен, и не может использоваться, например, с классами.

Для консольных программ ключевое слово **namespace** создает свое собственное пространство имен, которое по умолчанию называется именем проекта. В нашем случае пространство имен называется *Hello*, однако программист вправе указать другое имя.

Каждое имя, которое встречается в программе, должно быть уникальным. В больших и сложных приложениях используются библиотеки разных производителей. В этом случае трудно избежать конфликта между используемыми в них именами. Пространства имен предоставляют простой механизм предотвращения конфликтов имен. Они создают разделы в глобальном пространстве имен.

C# – полностью объектно-ориентированный язык, поэтому написанная на нем программа будет представлять собой совокупность взаимодействующих между собой классов. Даже сейчас, в самом простейшем случае, автоматически был создан класс с именем **Program**. Данный класс содержит только один метод – метод **Main()**, который является точкой входа в программу. Это означает, что именно с данного метода начнется выполнение приложения. Каждая консольная программа на языке C# должна иметь метод `Main()`.

Замечание

*Перед объявлением типа возвращаемого значения **void** (который означает, что метод не возвращает значение) стоит ключевое слово **static**, которое означает, что метод `Main()` можно вызывать, не создавая экземпляр класса **Program**.*

Тело метода `Main()` ограничивают парные фигурные скобки. Добавим в тело метода следующий код:

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

Здесь *Console* имя стандартного класса из пространства имен *System*. Его метод *WriteLine* выводит на экран текст, заданный в кавычках.

Для запуска программы вместо кнопки Start можно нажать на клавиатуре клавишу F5, или выполнить команду **Debug – Start Debugging**. Если код программы не содержит ошибок, то сообщение выведется в консольное окно, которое мелькнет и быстро закроется. Чтобы просмотреть сообщение в нормальном режиме, нужно нажать клавиши Ctrl+F5, или выполнить команду **Debug – Start Without Debugging**. В нашем случае откроется консольное окно, которое показано на Рис. 8.

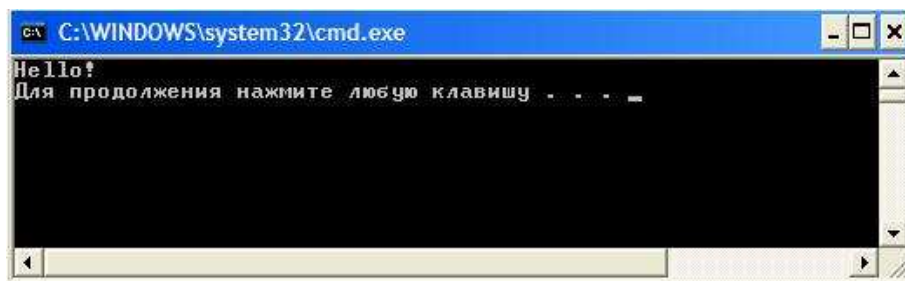


Рис. 8. Окно консольного приложения

Замечание

В более сложных приложениях имеет смысл явно вставить в конец программы команду `Console.ReadLine()`, позволяющую подождать до тех пор, пока пользователь не нажмет на клавишу <Enter>. Данный вариант является более предпочтительным, потому что нажатие Ctrl+F5, или выбор соответствующей команды в меню, приводит к запуску приложения с отключенным режимом отладки, который довольно часто бывает необходим для поиска ошибок.

Если код программы будет содержать ошибки, например, пропущена точка с запятой после команды вывода, то после нажатия клавиши F5 откроется диалоговое окно, в котором выведется сообщение о том, что обнаружена ошибка, и вопрос, продолжать ли работу дальше. Если вы ответите **Yes**, то будет выполнена предыдущая удачно скомпилированная версия программы (если такая компиляция уже была). Иначе процесс будет остановлен и появится окно ошибок **Error List**.

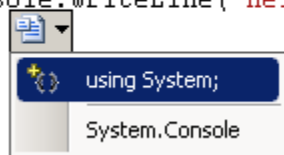
Задание 1

Если вы удалили все директивы `using`, то код не откомпилируется. Установите курсор ввода на слово `Console` и обратите внимание на бордовую линию, появившуюся около последней буквы этого слова:

```
static void Main(string[] args)
{
    Console.WriteLine("Hello!");
}
```

Это работает подсказка Visual Studio. Подведите курсор мыши к этой подсказке. Умная среда предлагает два варианта решения проблемы:

```
static void Main(string[] args)
{
    Console.WriteLine("Hello!");
}
```



Попробуйте каждый из них. В чем их отличия?

Задание 2

Измените текст кода так, чтобы на экран выводилось сообщение:

«Привет! Меня зовут...».

Попробуйте сделать несколько ошибок, например, не закрыть скобку или поставить лишнюю скобку, и посмотрите, какое сообщение появится в окне списка ошибок.

Эволюция технологий программирования. Основные понятия объектно-ориентированного программирования

В окончательном виде любая программа представляет собой набор инструкций процессора. Все, что написано на любом языке программирования – это более удобная для человека запись этого набора инструкций, облегчающая написание, отладку и последующую модификацию программ.

Первые программы создавались посредством переключателей на передней панели компьютера. Такой способ программирования подходил для создания только небольших программ. Потом программы стали писать на языке машинных команд, т.е. непосредственно в машинных кодах. Затем был изобретен язык ассемблер, в котором появились символические имена для обозначения переменных. Язык ассемблер относится к языкам низкого уровня, но не потому, что программы, разработанные на данном языке «низкого» качества, а потому, что для написания самой простой программы программисту требовалось знать команды конкретного типа процессора и напрямую обращаться к данным, размещенным в его регистрах.

Развитие вычислительной техники вело к быстрой смене типов и моделей процессоров, поэтому стало необходимо обеспечивать аппаратную переносимость программ. Это привело к появлению языков программирования высокого уровня, первым из которых был Фортран. Его существенным отличием стало то, что программа разрабатывалась на языке, «схожем» с естественным языком, а для перевода программы в машинный код (в набор инструкций для конкретного типа процессора) стал использоваться транслятор. Теперь, чтобы перенести программу с одной аппаратной платформы на другую, следовало «перетранслировать» исходный код программы с помощью соответствующей версии транслятора.

С ростом объема программ становилось невозможным удерживать в памяти все детали, и стало необходимым структурировать информацию, выделять главное и отбрасывать несущественное. Этот процесс получил название повышения степени абстракции программы и привел к появлению структурного программирования.

Первым шагом в данном направлении стало использование подпрограмм. Использование подпрограмм позволяет после их разработки и отладки отвлечься от деталей реализации. При этом для вызова подпрограммы требуется знать только ее интерфейс, который, если не используются глобальные переменные, полностью определяется заголовком подпрограммы.

Вторым шагом в данном направлении стала возможность создания собственных типов данных, позволяющих структурировать и группировать информацию. Для работы с собственными типами данных стали разрабатываться специальные подпрограммы.

Объединение в модули описаний собственных типов данных и подпрограмм для работы с ними, в совокупности с сокрытием от пользователя данного модуля деталей его реализации, стало следующим шагом в развитии структурного программирования. Создаваемые модули стали помещать в библиотеки подпрограмм. Таким образом, во всех языках программирования появились обширные библиотеки стандартных подпрограмм.

Структурный подход к программированию позволил успешно создавать достаточно крупные проекты, а также, благодаря использованию готовых библиотек, уменьшить время разработки проектов и облегчить возможность их модификации. Однако сложность программного обеспечения продолжала возрастать, и требовались все более сложные средства ее преодоления. Идеи структурного программирования получили свое дальнейшее развитие в объектно-ориентированном программировании (ООП) – технологии, позволяющей достичь простоты структуры и управляемости очень крупных программных систем.

ООП основывается на введении понятия «класс», который является естественным продолжением модульности программирования. В классе структуры данных и подпрограммы (методы) для их обработки объединяются. При этом класс используется только через свой интерфейс, а детали реализации для пользователя класса недоступны.

Например, во введении мы уже познакомились с одним из стандартных классов языка C# – *Console*, определенном в пространстве имен *System*. Мы обращались к его методу *WriteLine* для вывода информации на консольное окно. При этом реализация класса, а также использованного метода для нас несущественны.

Следует отметить, что с точки зрения компилятора класс является типом данных, определяемым пользователем. В классе задаются свойства и поведение какого-либо предмета или процесса в виде членов класса – полей данных (членов-данных) и методов (членов-методов) для работы с ними. Создаваемый тип данных обладает практически теми же свойствами, что и стандартные типы. Тип данных определяет внутреннее представление данных в памяти компьютера, множество значений, которые могут принимать величины этого типа, а также операции, применяемые к этим величинам. Все это можно задать и в классе.

Существенным отличием классов является то, что они отражают строение объектов реального мира. В реальном мире можно, например, управлять автомобилем, не имея представления о принципе внутреннего сгорания и устройстве двигателя. Аналогично, зная только интерфейс класса (заголовки его методов), можно управлять его данными.

Конкретные величины типа данных «класс» называются экземплярами класса, или объектами. Создание экземпляров класса и присваивание им начальных значений выполняется с помощью специальных членов класса – конструкторов.

Основными принципами ООП являются инкапсуляция, наследование и полиморфизм. Рассмотрим данные принципы более подробно.

Инкапсуляция – это объединение данных и методов их обработки в сочетании с сокрытием ненужной для использования этих данных информации. Инкапсуляция повышает степень абстракции программы – для использования в программе какого-то класса не требуется знаний о его реализации, достаточно знать только его интерфейс. Это позволяет изменить реализацию класса, не затрагивая саму программу, при условии, что интерфейс класса останется прежним.

Наследование – это возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые. При этом свойства повторно не описываются, что сокращает объем программы. Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные – на ветвях или листьях.

Полиморфизм – это возможность использования в рамках одного класса или различных классов одного имени для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы. Понятие полиморфизма широко используется в C#. Самым простым его примером может служить перегрузка методов. Например, известный нам метод *WriteLine* класса *Console* имеет 19 вариантов. При выполнении данного метода компилятор автоматически выбирает наиболее подходящий вариант в соответствии с количеством и типами передаваемых в метод параметров. Однако чаще всего понятие полиморфизма связано с механизмом использования виртуальных методов.

Если вспомнить нашу первую программу:

```
class Program //класс
{
    static void Main () //метод класса
    {
        Console.WriteLine("Hello!!!");
    }
}
```

то можно увидеть, что она фактически представляет собой класс *Program* с единственным статическим (static) методом *Main*. Таким образом, хотим мы того или нет, мы фактически разрабатываем собственный класс.

Самостоятельная работа №1

1. Прочитать [1] – стр. 18, 23-29, [3] – стр.1-20, [5] – стр. XIII-XX, 2-27.
2. Выяснить, чем отличаются режимы Start Debugging и Start Without Debugging
3. Объяснить, что такое метаданные, и какую роль они играют в создании IL-кода и выполнении программы в среде CLR.
4. Объясните, чем exe-файл, созданный для платформы .NET, отличается от «классических» exe-файлов.
5. Объясните, чем exe-файл для платформы .NET отличается от dll-файлов.