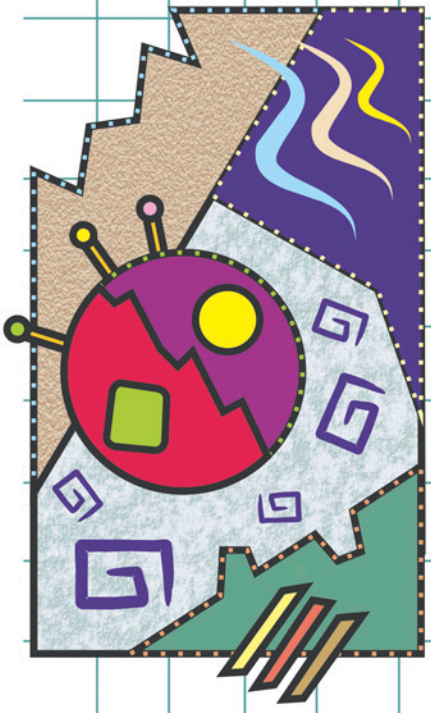
bhv
www.bhv.ru
www.bhv.kiev.ua

Николай Секунов

# C#

САМОУЧИТЕЛЬ



Среда разработки приложений Visual Studio.NET

Справочник по языку C#

Создание распределенных приложений

Реализация пользовательского интерфейса

*Первый компонентно-ориентированный язык программирования*

Николай Секунов

# САМОУЧИТЕЛЬ C#


Дюссельдорф ◆ Киев ◆ Москва ◆ Санкт-Петербург

УДК 681.3.06

Книга посвящена первому компонентно-ориентированному языку программирования распределенных приложений C#. Приведены сведения обо всех основных элементах данного языка, начиная с примитивов. Подробно рассматривается предназначенная для разработки приложений среда Microsoft Visual Studio.NET; структура программ на C# и этапы компиляции; объединение компонентов, написанных на различных языках высокого уровня; реализация пользовательского интерфейса; обеспечение безопасности приложений. Изложенная методика создания приложений и многочисленные примеры позволят приобрести устойчивые навыки программирования на языке C#.

*Для программистов*

**Группа подготовки издания:**

| | |
|---|---|
| Главный редактор | *Екатерина Кондукова* |
| Зав. редакцией | *Наталья Таркова* |
| Редактор | *Алексей Птухин* |
| Компьютерная верстка | *Виктории Капецкой* |
| Корректор | *Светлана Симуни* |
| Дизайн обложки | *Игоря Цырульникова* |
| Зав. производством | *Николай Тверских* |

**Секунов Н. Ю.**
Самоучитель C#. — СПб.: БХВ-Петербург, 2001. — 576 с.: ил.

ISBN 5-94157-028-7



Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.06.01.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 46,44.
Тираж 5000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99
от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с диапозитивов
в Академической типографии "Наука" РАН.
199034, Санкт-Петербург, 9-я линия, 12.

# Содержание

# Введение

В процессе конкурентной борьбы перед корпорацией Microsoft встала задача приостановить победное шествие языка программирования Java, разработанного корпорацией Sun. Попытки расколоть стандарт этого языка изнутри путем введения в Visual Java элементов, ускоряющих его работу, но делающих его несовместимыми со всеми платформами, за исключением Windows, привели к судебному разбирательству, которое закончилось не в пользу Microsoft. Ответом на это стала разработка языка программирования C#, решающего те же задачи, но являющегося лицензионно чистым, хотя в нем в большой степени учтен опыт разработки языка Visual Java.

Язык C# представляет собой язык программирования более высокого уровня, чем язык C++ и может рассматриваться как его дальнейшее усовершенствование. Изначально язык C разрабатывался как инструментальный язык операционной системы UNIX. Поэтому перед его транслятором ставилась задача создания машинных кодов, не уступающих по эффективности машинным кодам, созданным языком ассемблера. Такие жесткие требования наложили свой отпечаток на синтаксис языка. Для расширения возможностей языка C на его основе был создан объектно-ориентированный язык C++, который иногда называли "C с классами". C++ идеально подходит для создания приложений на одном компьютере. Однако в настоящее время все большее распространение получают распределенные приложения. При их создании программист имеет дело с обширными библиотеками готовых программ, которые нужно только объединить в одно приложение. В этом случае требования эффективности создаваемого машинного кода отступают на второй план, поскольку эффективность данных приложений в основном определяется эффективностью используемых библиотек, а на первый план выступает обеспечение надежности взаимодействия объединяемых компонентов приложения, которые могут быть созданы разработчиками, не имеющими друг о друге ни малейшего представления. Язык C# как раз и является языком программирования распределенных приложений.

## Для кого предназначена эта книга?

Язык программирования C# является абсолютно новым и его стандарт еще не устоялся, однако, уже сейчас к нему проявляется определенный интерес со стороны тех программистов, которые хотели бы оценить возможности его

использования для реализации своих проектов. Пока что единственной средой программирования, в которой реализован язык C#, является среда Microsoft Visual Studio 7.0, известная, так же как Microsoft Visual Studio.NET, поэтому в данной книге рассмотрен не только сам язык, но и его поддержка со стороны среды программирования.

В приложении приведено описание интегрированной среды разработки Microsoft Visual Studio.NET, в которой тестировались все приведенные в книге примеры. Кроме того, там же дается краткое описание среды исполнения приложений, написанных на языке C#, что позволяет лучше понять его особенности и логику построения. В главе 2 изложена методика построения консольных приложений C#, используя которые можно получить навык работы практически со всеми конструкциями описываемого языка (по крайней мере, все приведенные в данной книге примеры представляют собой консольные приложения или же их фрагменты).

## Структура книги

Как и в большинстве книг по C и C++, в книге, предлагаемой вашему вниманию, сначала дано краткое описание языка программирования C#, а затем подробно рассмотрены некоторые его аспекты. В приведённом ниже списке глав приводится краткое их описание, позволяющее пользователю лучше ориентироваться в структуре книги.

- Глава 1. C# и Visual Studio.NET.

  В главе рассмотрены концепции объектно-ориентированного и компонентно-ориентированного программирования. Рассмотрена так же реализация концепции компонентно-ориентированного программирования в среде .NET и роль языка C# в этой реализации. Здесь же изложены основные особенности данного языка программирования.

- Глава 2. Создание простейшего приложения.

  В главе описана процедура создания простейшего консольного приложения в среде Visual Studio.NET. Здесь также содержится описание структуры консольного приложения и основных его компонентов, что позволит читателю подобные приложения самостоятельно создавать и модифицировать.

- Глава 3. Первое знакомство с C#.

  В главе приведено краткое описание языка программирования C#. Эту главу можно рассматривать как краткий конспект глав последующих. Здесь же подробно обсуждаются некоторые вопросы, вынесение которых в отдельную главу, по мнению автора, было бы нецелесообразно.

- Глава 4. Лексическая структура языка.

  В главе рассмотрена структура языка и этапы его компиляции, содержится описание директив препроцессора и определены понятия идентификатора, ключевого слова, литерала, операции и знака препинания.

- Глава 5. Пространства имен.

  В главе рассмотрена структура программ C#, определяются режимы доступа и область действия членов пространств имен и типов. Здесь же рассмотрено скрытие имен и другие аналогичные вопросы.

- Глава 6. Переменные.

  Глава посвящена переменным, используемым в языке C#, описанию стандартных операций данного языка, а также способам обращения к его элементам в выражениях.

- Глава 7. Преобразование типов.

  В главе идет речь об операциях преобразования типов в C#, дан полный перечень допустимых явных и неявных преобразований типов, а также подробно описан алгоритм каждого из этих типов преобразований.

- Глава 8. Выражения.

  В главе рассмотрены выражения, используемые в языке C#, стандартные операции и их перегрузка, описана работа с функциональными членами типов, поиск требуемой реализации и другие связанные с этим вопросы.

- Глава 9. Операторы.

  Глава посвящена операторам языка C#.

- Глава 10. Классы.

  В главе рассматривается работа с классами языка C#, наследование классов, создание статических и виртуальных членов класса, скрытие членов класса, а также дано описание всех членов класса.

- Глава 11. Интерфейсы.

  В главе описана работа с интерфейсами языка C#. Рассмотрено наследование интерфейсов, их реализация в классах и структурах, а также дано описание всех членов интерфейса.

- Глава 12. Атрибуты.

  Глава посвящена атрибутам языка C#. Атрибуты используются для связывания с программными единицами C# некоторой информации, которая может быть получена в процессе выполнения приложения.

- Глава 13. Реализация пользовательского интерфейса.

  В главе рассматриваются методы работы со строками, форматирование выводимой информации, модификация формата, а также работа с файлами и потоками.

- Глава 14. Включение отладочной информации.

  В главе идет речь о классах и атрибутах, позволяющих включать отладочные функции не только в отладочную, но и в распространяемую версию приложения.

- Глава 15. Обработка исключений.

  В главе описываются принципы обработки исключений в языке C#. В этой же главе рассмотрен способ управления проверкой переполнения при выполнении операций над целыми числами.

- Глава 16. Обеспечение безопасности.

  Глава посвящена методам, используемым в среде .NET для обеспечения безопасности использования компонентов, полученных из ненадежных источников, например, из Internet.

## Соглашения, принятые в книге

В книге автором использовалось специальное форматирование для выделения некоторых текстов или фрагментов текста. Ниже приведены основные принципы используемого форматирования:

- исходные тексты фрагментов программ, представляющие собой одну или более строк текста, выделяются специальным шрифтом, как это показано ниже:

```
class Some_Class
{
       private static bool   isEmpty;       // Статическое поле
       public string         Name;          // Нестатическое поле
}
```

- имя класса, функции, имя типа переменной или фрагмент текста длиной менее строки выделяются специальным шрифтом. Например: `int`, `System`, `name` и т. д.;
- имя клавиши заключается в угловые скобки (<>) и выводится жирным шрифтом. Например: <Ctrl>, <F5> и т. д.;
- при первом появлении нового термина он выделяется курсивом. Например: *новый термин*;
- ссылка на другую главу содержит только ее номер. Например: *См. главу 5.*

# Глава 1

# C# и Visual Studio.NET

На протяжении всей истории программирования особое внимание уделялось выработке единого, стандартного подхода к проектированию и реализации компьютерных программ. Потребность в таком подходе обуславливалась прежде всего усложнением задач, решаемых программистами, и необходимостью максимально уменьшить число ошибок в программах еще на этапе их разработки. Первые попытки выработки единого подхода к программированию связаны с так называемым структурным программированием.

Приверженцы структурного подхода к программированию пытались разработать некоторые универсальные методы проектирования систем, организации данных и проектирования программ. Однако такой подход не получил широкого распространения, поскольку не содержал никаких революционных идей и представлял собой, по сути, набор благих пожеланий и полезных советов. Революция в разработке программного обеспечения наступила с появлением концепции объектно-ориентированного программирования.

## Объектно-ориентированное программирование

Концепция объектно-ориентированного программирования позволяет рассматривать программу как набор модулей, каждый из которых выполняет отведенную ему функцию. Если вам необходимо отправить письмо, то вы не идете на вокзал, не отыскиваете человека, едущего в тот же город и не просите его передать письмо по определенному адресу. Вы просто опускаете письмо в почтовый ящик и надеетесь, что оно вовремя дойдет до адресата. Точно также, если вам необходимо вызвать на экран окно, то вам совсем не обязательно разбираться в том, каким образом отображается его рамка, а тем более, каким образом производится обработка поступающих в него и отправляемых им сообщений. Тем более что, если вы попытаетесь это сделать, то вы что-нибудь упустите (а это естественно при отсутствии у вас полного описания системы) и, скорее всего, вызовете крах всей системы.

Однако концепция объектно-ориентированного программирования (ООП) не ограничивается только предоставлением пользователю удобных библио-

тек для решения сложных задач. Эти вопросы могут быть решены и без использования данной концепции, например, в рамках структурного программирования. Программные объекты были разработаны для облегчения процесса программирования и приближения структуры программ к привычным для человека формам.

Можно смело сказать, что именно способность обобщать, классифицировать и генерировать абстракции явилась причиной превращения обезьяны в человека. Без этой способности невозможно было бы создание языка общения, поскольку каждое имя существительное является по сути своей классом объектов, обладающих некоторым набором атрибутов или отличительных черт. С другой стороны, каждый подобный класс является базовым для других классов, формируемых, например, путем добавления к этому существительному конкретизирующего его прилагательного. Например, "белый дом".

В основе концепции объектно-ориентированного программирования лежат три основные принципа:

1. Инкапсуляция:

   Совмещение структур данных с функциями (методами), предназначенными для манипулирования этими данными. Инкапсуляция достигается путем введения нового механизма структурирования и типизации данных — класса.

2. Наследование:

   Создание новых, производных классов, которые наследуют данные и функции от одного или нескольких ранее определенных базовых классов. При этом возможно переопределение или добавление новых данных и методов. В результате создается иерархия классов.

3. Полиморфизм:

   Присвоение методу единого имени или идентификатора в рамках иерархии классов, чтобы любой класс в иерархии имел возможность по-своему выполнять связанные с этим методом действия.

Одновременно с появлением и детализацией концепции ООП появились и основанные на ней языки программирования. Одним из первых языков стал алгоритмический язык Modula 2. Язык программирования Turbo Pascal, разработанный фирмой Borland, начиная с версии 5.5, превратился в объектно-ориентированный. Но наиболее последовательное воплощение концепция объектно-ориентированного программирования нашла в алгоритмическом языке C++, разработанном сотрудником Bell Laboratories AT&T Брайаном Страуструпом на базе языка программирования C как средство моделирования больших телефонных коммутационных систем.

Рассмотрим основополагающие принципы ООП подробнее.

## Инкапсуляция

Одним из главных отличий стандартного структурного программирования от объектно-ориентированного программирования является инкапсуляция. Совмещение структур данных с методами, предназначенными для манипулирования этими данными, позволяет вам сосредоточить в одном месте всю информацию, касающуюся определенных данных и методов их обработки, и тем самым облегчить доступ к этим методам и данным из любой точки программы.

Действительно, представьте себе, что все данные в программе имеют глобальную область видимости (что неизбежно при тесном взаимодействии всех частей программы). При этом вам придется для всех данных и для каждого метода использовать уникальный идентификатор, причем этот идентификатор должен быть информативным, то есть длинным. Программы с такими идентификаторами станут практически нечитаемыми, особенно на экранах дисплеев. Вам придется постоянно следить, чтобы новые имена не пересекались с уже используемыми, чтобы в каждой функции использовались именно те данные, для которых она была написана. Кроме того, вам придется отказаться от таких элегантных возможностей, как перегрузка операций и создание собственных типов данных, позволяющих автоматически проверять корректность выполняемых над ними операций.

Кроме того, инкапсуляция позволяет вам скрыть данные и функции, используемые исключительно членами данного класса или членами производного от него класса. Это упрощает описание класса для его использования внешними функциями и повышает надежность его работы, поскольку критически важные данные могут быть, таким образом, защищены от несанкционированного доступа.

Хорошим тоном считается скрытие всех данных, используемых данным классом, и обеспечение доступа к ним только посредством соответствующих методов класса. Это прежде всего устраняет все вопросы, связанные с областью видимости переменных, и позволяет использовать в каждом классе простые и информативные идентификаторы, не заботясь о том, что подобные идентификаторы уже используются в других классах.

Обеспечение доступа к данным исключительно через методы класса дает возможность избегать потенциально очень опасных ситуаций. Предположим, что в процессе разработки какого-либо проекта одному из разработчиков была поставлена задача написать обработчик для некоторого массива, содержащего указатели на символьные строки. На каком-то этапе этот разработчик решил заменить этот массив связанным списком и соответствующим образом переписал методы обращения к нему.

Если он оформил этот массив в виде класса и обеспечил доступ к массиву исключительно через методы этого класса, то другие разработчики ни при каких обстоятельствах не должны почувствовать последствия этой замены

(разве что она может отразиться на скорости выполнения приложения). Но если к этому массиву был обеспечен прямой доступ, то никто не может поручиться, что другой программист, работающий с теми же данными, не захочет воспользоваться собственными, более эффективными с его точки зрения методами работы с массивом, которого уже нет.

## Наследование

Принцип наследования в объектно-ориентированном программировании позволяет создать класс, включающий в себя все данные и методы некоторого другого класса, и, кроме того, содержащий свои, присущие только ему данные и методы обработки. Такой подход позволяет строить целые иерархии классов, каждый следующий уровень которых отличается все большей детализацией по сравнению с предшествующими ему уровнями. Достоинства такого подхода наиболее полно проявляются при программировании достаточно сложных объектов, к которым по праву относится сама операционная система Windows.

При всем своем различии окно представления и диалоговое окно являются окнами Windows и, как таковые, имеют много общего. Представьте себе, во что превратилось бы описание класса диалогового окна, если бы в нем непосредственно были бы описаны все обрабатываемые им данные и все методы обработки, учитывая тот факт, что каждое текстовое поле и каждый переключатель по сути являются самостоятельным и полноправным окном Windows. Не говоря уже о том, сколько места займет описание каждого класса и сколько в них будет повторяющихся и практически аналогичных по тексту функций.

Наследование позволяет описать сначала общий класс окна Windows, на его основе описать общий класс диалогового окна Windows, а уже на основе этого класса сформировать класс конкретного диалогового окна, используемого для реализации конкретного пользовательского интерфейса.

## Полиморфизм

Последней отличительной чертой объектно-ориентированного программирования является полиморфизм. Использование полиморфизма позволяет существенно расширить набор функций в базовом классе, что, в свою очередь, приводит к существенному сокращению текста программы и улучшению ее читабельности.

Греческое слово "полиморфизм" было заимствовано из биологии, где оно означает наличие в пределах одного вида резко отличающихся по облику особей. В C++ полиморфизм реализуется с помощью так называемых виртуальных функций, которые позволяют в пределах иерархии классов использовать одно имя метода для выполнения одной и той же операции над различными типами данных. Решение, какая именно версия данного метода

будет использована в каждом конкретном случае, принимается на этапе исполнения программы и носит название позднего связывания.

Суть полиморфизма может быть наглядно проиллюстрирована на следующем примере. Предположим, что вы решили создать иерархию классов для манипулирования отображаемыми на экране графическими объектами. Вы создали базовый класс, в который включили среди прочих функцию перемещения графических объектов. Казалось бы, данную функцию очень просто реализовать: необходимо только определить координаты нового положения объекта, уничтожить его изображение на старом месте и нарисовать на новом.

Однако при ближайшем рассмотрении перед вами встает, казалось бы, неразрешимая задача: функции отображения конкретных графических объектов задаются в производных классах, а обращение к ним предполагается производить в базовом классе. Конечно, у данной проблемы существует простое решение: перенести функцию перемещения объекта из базового в производный класс и для каждого класса написать свой вариант функции. Однако если следовать такой логике, в базовом классе практически не останется методов обработки, а все преимущества наследования будут практически сведены к нулю.

Поэтому, как уже упоминалось выше, в рамках ООП был разработан аппарат *виртуальных функций*. Если в производном и базовом классах одна и та же функция объявлена виртуальной (для этого функции должны иметь не только одно и то же имя, но и один и тот же набор формальных параметров и одинаковый тип возвращаемой величины), то при обращении к этой функции из любого метода, унаследованного данным классом от своих базовых классов, будет вызываться функция, определенная в данном классе, а не функция соответствующего базового класса, как это имело бы место в противном случае.

Если виртуальные функции имеют различный набор формальных параметров, то они рассматриваются как разные функции. Различный тип возвращаемой величины у виртуальных функций с одинаковым именем и набором формальных параметров рассматривается как ошибка.

## Компонентно-ориентированное программирование

Объектно-ориентированное программирование позволило перейти на новый этап сложности разрабатываемых приложений, поскольку используемые в нем объекты (классы, структуры, интерфейс и т. д.) могли разрабатываться и тестироваться раздельно, что существенно облегчало их разработку и повышало их надежность. Отлаженные объекты, объединенные в библиотеки динамической компоновки, могли быть впоследствии использованы другими программистами для решения своих задач.

Однако развитие сетевых приложений (особенно это касается Сети, как часто называют Internet) поставило вопрос о необходимости перехода на следующий уровень разработки программного обеспечения. Теперь программной единицей становится не объект, а компонент, под которым можно понимать отдельное приложение, распространяемое по сети. Компьютер, на который поступило данное приложение, должен запустить его на исполнение и предоставить ему все необходимые ресурсы. Поскольку эти приложения являются довольно самостоятельными, работающее с ними серверное приложение может не отличаться особой сложностью, но перед ним во всей своей полноте встает проблема обеспечения безопасности.

Когда мы говорим о безопасности, имеются в виду не вирусы и атаки троянцев, хотя и от них нужно защищаться, а тот простой факт, что серверное приложение должно координировать работу приложений, разработчики которых, скорее всего, даже не подозревали о существовании друг друга и, конечно же, не проверяли разрабатываемые ими приложения на совместимость. Кроме того, никто не может гарантировать, что для написания всех координируемых приложений использовался один и тот же язык программирования. Поэтому следует изначально полагать, что в данной среде должны работать приложения, написанные на разных языках программирования.

Поставленные задачи предъявляют довольно-таки специфические требования к языку программирования, на котором будет написано приложение сервера. В свое время корпорация Microsoft попыталась решить эту проблему, создав технологию COM (Component Object Module, Модель Многокомпонентных Объектов). Данная технология оказалась вещью в себе, поскольку ее использование в широко распространенных языках программирования (например, Visual C++) было сопряжено с определенными трудностями. Не говоря уже о том, что в других не менее распространенных языках программирования, таких как Visual Basic, некоторые ее возможности были вообще недоступны.

Убедившись, что технология COM не решает поставленных перед нею задач, Microsoft создала новую технологию .NET Runtime, основным языком которой и является C#. В этой технологии один способ описания программного кода (метаданные), одна среда выполнения (Common Language Runtime) и одна базовая библиотека (Frameworks).

## Среда Common Language Runtime

Технология .NET Runtime реализована в среде Common Language Runtime (CLR). Среду CLR не стоит путать со средой разработки Visual Studio, которую можно рассматривать как надстройку над CLR. CLR лучше интерпретировать как некоторый препроцессор, преобразующий команды некоторого промежуточного языка *Intermediate Language* (IL) в команды процессора. CLR не только предоставляет средства, облегчающие программирование, но и обеспечивает выполнение программного кода

Использование CLR для разработки приложения обеспечивает:

- межъязыковую интеграцию (используется Common Language Specification);
- автоматическое управление памятью (сборка мусора);
- независимую от языка обработку исключений;
- контроль типов данных;
- контроль версий программ.

Большинство из этих возможностей CLR реализовано за счет использования метаданных. Метаданные представляют собой объединение программы и данных в файле формата Portable Executable (PE). Приложение, создаваемое с использованием CLR, называется контролируемым приложением. Это приложение содержит метаданные, используемые средой CLR для проверки соответствия имеющихся версий компонентов требованиям данного приложения. Объединение программного кода и данных позволяет снять многие проблемы с системным реестром.

Одной из приятных особенностей CLR является автоматическое управление памятью и сборка мусора. Среда сама выявляет объекты, на которые нет ни одной ссылки, и уничтожает их. Единственный недостаток такого подхода — невозможность ручного уничтожения объектов, что приводит к некоторому перерасходу оперативной памяти. Однако если задуматься, немногие злоупотребляли возможностью ручного удаления объектов при программировании на C++. Как правило, память выделялась в конструкторе класса и освобождалась в его деструкторе.

Как уже говорилось выше, создаваемый компилятором C# контролируемый программный код представляет собой не набор инструкций процессора, а код на некотором промежуточном языке Intermediate Language. Преимущество промежуточного кода состоит в его независимости от типа процессора. Для преобразования этого кода в команды процессора на компьютере должен быть установлен компилятор промежуточного языка.

Программа на промежуточном языке содержит не только сам программный код, но и метаданные, предоставляющие среде выполнения информацию обо всех используемых типах, сигнатурах всех элементов этих типов и другую информацию, которая в приложениях COM помещалась в библиотеки типов и системный реестр. Таким образом, программа и вся необходимая ей информация размещена в одном файле, что существенно облегчает установку распространяемого в нем приложения в системе, поскольку исключает операции регистрации в системном реестре и поиск соответствующей библиотеки типов. Формат файлов, создаваемых средой Common Language Runtime, является расширением формата Portable Executable, используемого хранения исполняемых файлов (.exe) и библиотек динамической компоновки (.dll).

Чтобы лучше понять, что собой представляет IL-код, необходимо рассмотреть его примитивы, то есть элементы, на которые он разлагает программу на языке C#. Ниже приведен далеко не полный список этих примитивов:

- арифметические и логические операции;
- условные операции;
- операции выделения и освобождения памяти;
- работа со стеком;
- передача аргументов;
- объектная модель;
- создание объектов;
- работа с исключениями;
- массивы;
- типизированные адреса.

## Компиляторы времени выполнения приложения

Как уже говорилось выше, компиляторы C# и других языков программирования в Visual Studio.NET создают контролируемый код, представляющий собой программный код на языке Intermediate Language. Поскольку этот язык существенно отличается от языка команд процессора, перед тем как этот код будет исполнен, его следует откомпилировать. Этот процесс происходит в процессе выполнения приложения, поэтому используемые компиляторы носят название компиляторов времени выполнения приложения (Just-in-Time Compiler или JIT-компиляторы).

Компиляция в процессе выполнения приложения позволяет существенно сэкономить память, поскольку компоненты, как правило, представляют собой достаточно объемные и многофункциональные приложения, большинство функций которых не используется при решении конкретной задачи, а программа на языке высокого уровня обычно занимает намного меньше места, чем исполняемый код. JIT-компиляторы достаточно эффективны, поскольку при создании промежуточного языка основное внимание уделялось именно эффективности работы компилятора, а не удобству работы с ним разработчика, как это имеет место в традиционных языках программирования высокого уровня. Наличие промежуточного этапа компиляции не должно вызывать у разработчиков особых опасений. Если опасения все-таки возникнут, стоит вспомнить о том, что этот этап преобразования программного кода, скорее всего, не последний, поскольку все современные процессоры преобразуют команды формата x86 в свои внутренние команды, что только способствует повышению их быстродействия.

Работа с компонентом, использующим промежуточный язык, происходит следующим образом: при загрузке компонента для каждого метода каждого из его типов создается заглушка, передающая управление JIT-компилятору, преобразующему IL-код данного метода в набор команд процессора. При вызове метода управление передается этой заглушке. При первом обращении к заглушке после получения машинного кода в ней заменяется указатель, по которому производится передача управления. Теперь он указывает на скомпилированный машинный код данного метода.

В среде CLR для Windows используется не один, а целых три JIT-компилятора. Ниже приведено их краткое описание:

- JIT — компилятор, используемый в CLR по умолчанию. Этот компилятор предварительно анализирует поток выполнения, в который будет включен исходный код, и создает оптимизированный контролируемый машинный код. Этот компилятор использует большой объем памяти, требователен к другим системным ресурсам и его работа занимает достаточно много времени. Этот компилятор может работать с полным набором команд Intermediate Language.
- EconoJIT — представляет собой упрощенную версию JIT-компилятора. Этот компилятор поддерживает кэширование создаваемого машинного кода, но оптимальность этого кода оставляет желать лучшего. EconoJIT рекомендуется использовать при недостаточном объеме оперативной памяти, поскольку он позволяет отбрасывать неиспользуемый скомпилированный код и таким образом уменьшать объем программы. Следует отметить также и высокое быстродействие данного компилятора.
- PreJIT — используется для преобразования IL-кода в машинный код до запуска приложения. Такая стратегия может использоваться в том случае, если компонент имеет малый размер и существует большая вероятность того, что большинство его методов будут использованы. Как можно догадаться, предварительная компиляция существенно сокращает время загрузки и старта приложения, хотя и создает некоторые проблемы с версиями компонент, поскольку замена самой компоненты не означает автоматической замены ее откомпилированного кода, как это имеет место при традиционном подходе. Этот компилятор запускается, например, при установке компонент NGWS.

Выбор JIT-компилятора и настройка параметров его работы производится утилитой JIT Compiler Manager. Эта утилита содержится в файле jitman.exe, расположенном в поддиректории /bin папки, в которую установлен пакет Microsoft NGWS SDK. Эта программа работает в фоновом режиме, поэтому при ее запуске не выводится никакого окна, а ее значок помещается в область системных программ панели задач (рядом с часами). Для ее настройки дважды щелкните левой кнопкой мыши на этом значке. Появится диалоговое окно **JIT Compiler Manager**, изображенное на рис. 1.1.

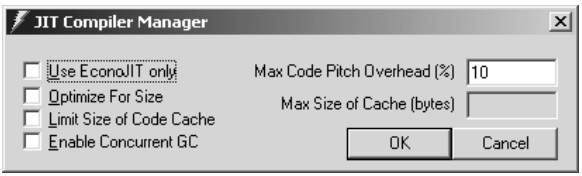



**Рис.1.1.** Диалоговое окно **JIT Compiler Manager**

Рассмотрим подробнее его элементы управления:

- **Use EconoJIT only** — как уже говорилось, в CLR по умолчанию используется компилятор JIT. При установке этого флажка будет использоваться компилятор EconoJIT.
- **Optimize For Size** — создаваемый машинный код оптимизируется по размеру, а не по быстродействию, как это имеет место по умолчанию.
- **Limit Size of Code Cache** — ограничивает размер кэша объемом, установленным в текстовом поле **Max Size of Cache (bytes)**. По умолчанию этот флажок не установлен, что позволяет использовать под кэш весь доступный объем оперативной памяти. В кэше размещается скомпилированный код, поэтому необходимо, чтобы заданный объем кэша превышал максимальный размер кода метода.
- **Enable Concurrent GC** — помещает процедуру сборки мусора (garbage collection) в отдельный поток. По умолчанию эта процедура выполняется в потоке пользовательского кода, что в некоторых случаях может замедлить реакцию приложения на внешние воздействия. Процедура параллельной сборки мусора работает медленнее, чем та же процедура, помещенная в пользовательский поток. Кроме того, выделение этой процедуры в отдельный поток возможно, по имеющимся сведениям, только в Windows 2000.
- **Max Code Pitch Overhead (%)** — текстовое поле, в котором можно указать максимальную долю времени, которое должно расходоваться на компиляцию. При превышении этого порога автоматически увеличивается размер кэша кода. Это текстовое поле используется только компилятором EconoJIT, хотя по непонятным причинам остается доступным и при сбросе соответствующего флажка.

## Система виртуальных объектов

Правила объявления и использования типов в среде CLR определяются системой виртуальных объектов (NGWS Virtual Object System или VOS). Использование системы VOS упрощает межъязыковую интеграцию и обеспечивает безопасность типов без существенного снижения производительности создаваемого приложения.