1. Основные принципы построения архитектуры .NET. Сравнительная характеристика технологий .NET и Java, а также промежуточного языка IL и байт-кода Java.

Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), которая подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду.

Программа для .NET Framework, написанная на любом поддерживаемом языке программирования, сначала переводится компилятором в единый для .NET промежуточный байт-код Common Intermediate Language (CIL) (ранее назывался Microsoft Intermediate Language, MSIL). В терминах .NET получается сборка. Затем код либо исполняется виртуальной машиной Common Language Runtime (CLR), либо транслируется утилитой NGen.exe в исполняемый код для конкретного целевого процессора. Использование виртуальной машины предпочтительно, так как избавляет разработчиков от необходимости заботиться об особенностях аппаратной части. В случае использования виртуальной машины CLR встроенный в неё JIT-компилятор «на лету» (just in time) преобразует промежуточный байт-код в машинные коды нужного процессора. Современная технология динамической компиляции позволяет достигнуть высокого уровня быстродействия. Виртуальная машина CLR также сама заботится о базовой безопасности, управлении памятью и системе исключений, избавляя разработчика от части работы.

Сравнительная характеристика технологий .Net и Java

1.Технология Java изначально разрабатывалась для встраивания в переносимые устройства, и лишь потом была преобразована для переносимого ПО в настольных компьютерах, позднее для серверного ПО. .Net – сразу для настольных компьютеров и серверов.

2.Обе технологии используют понятие промежуточного кода (для Java – Java-байт код, .Net – IL)

3.Обе среды используют автоматическое управление памятью.

4.Обе среды используют средства безопасности и контроля, которые позволяют использовать в Internet.

Основное преимущество .Net – более высокая продуктивность во всех сферах (более высокая производительность программного кода):

а) более высокоуровневый промежуточный язык. Промежуточный Java-байт код был разработан для создания виртуальных Java-машин. Java-байт код является упрощением всех команд процессора, а модель виртуальной Java-машины – пересечение всех архитектур. В виртуальной Java-машине ограничено количество ресурсов (их 4):
1)на стек
2)на динамическую память
3)на текущую исполняемую инструкцию
4)вспомогательный регистр

Промежуточный язык IL среды .Net принципиально был разработан не для интерпретации, а для динамической компиляции.

Когда запускается программа, то исполняющая система загружает промежуточный код и преобразовывает в двоичную систему.

б) в Java нет деления на размерные типы и ссылочные, а в .Net – есть. Следовательно, для платформы Java стек организован следующим образом: все элементы стека имеют фиксированный размер (8 байт). При этом, в этом элементе хранится признак типа (либо это число, либо это указатель), или само значение, или ссылка на объект. Из-за этого при передаче параметров возьмет в качестве параметра функцию рисования прямоугольника (принимает две точки). Элементом стека такой объект быть не может, этот объект выделяется в динамической памяти (heap).

Среда Java является многоязыковой, .Net - полиязыковой. Это обеспечивается благодаря тому, что в .Net стандартизирован промежуточный язык IL; в .Net разработана общая система типов (CTS), стандартизированы базовые и структурные типы данных, стандартизированы представления данных. Существует стандарт CLS (Common Language Specification) – набор правил, типов данных для межъязыковых вызовов (для доступа к программам, написанных на многих языках). Согласно CLS, программы могут обмениваться только ограниченным набором данных (если целые числа, то только знаковые).

CLS – исполняющая система (вызывает динамический компилятор, базовую библиотеку, сборщик мусора).

В Java нет межязыкового взаимодействия.

в) отличия в организации взаимодействия с подпрограммами, написанными на Assembler целевого процессора (на C, C++, Assembler). В Java это организуется так: виртуальная Java-машина предоставляет API JNI-набор процедур прикладного программирования, который получает, например, по символическому имени адрес процедур, находит по имени какой-то объект или адрес параметра на стеке по его имени. Требуется специальное программирование, чтобы организовать вызов dll-библиотеки из программы, написанной на Java. Этот вызов работает медленно.

В .Net это решено по-другому: средства вызова разных процедур встроены в саму вызывающую систему, преобразование параметров осуществляется автоматически. Вызов из dll-библиотеки быстрее.

Важнейшие свойства IL могут быть сформулированы следующим образом:
Объектная ориентированность и применение интерфейсов.
Строгое различие между типами значений и типами ссылок.
Строгая типизация данных.
Обработка ошибок через использование исключений.
Использование атрибутов.

Независимость .NET от языка имеет некоторые практические ограничения. IL неизбежно должен воплощать некоторую определенную методологию программирования, а это означает, что исходный язык также должен быть совместим с этой методологией. Принципы, которым руководствовалась Microsoft при создании IL: классическое объектно-ориентированное программирование с реализацией одиночного наследования классов. В дополнение к классической объектной ориентации IL также вводит понятие интерфейсов, которые впервые были реализованы под Windows с COM.

Одна из проблем межъязыкового взаимодействия была в том, что отлаживать компоненты, написанные на разных языках, приходилось независимо друг от друга. Невозможно было в отладчике переходить от одного языка к другому. Поэтому в действительности под способностью языкового взаимодействия мы подразумеваем возможность для классов, написанных на одном языке, напрямую обращаться к классам написанным на другом языке. В частности:

Класс, написанный на одном языке, может быть унаследован от класса, реализованного на другом.

Класс может содержать экземпляр другого класса, независимо от того, на каких языках написан каждый из них.

Объект может вызывать методы другого объекта, написанного на другом языке.

Объекты (или ссылки на объекты) могут передаваться между методами.

При вызове методов между языками можно шагать в отладчике по вызовам, даже если это означает необходимость перемещения между фрагментами исходного кода, написанными на различных языках.

Один из наиболее важных аспектов IL состоит в том, что он основан на исключительно строгой типизации данных. Это значит, что все переменные имеют четко определенный конкретный тип данных. В частности, IL обычно не допускает никаких действий, которые дают в результате неопределенные типы данных.

Хотя обеспечение безопасности типов может поначалу принести ущерб производительности, все же во многих случаях преимущества, полученные от служб, предоставляемых .NET, полагающиеся на безопасность типов, намного превышают потери от некоторого снижения производительности. Эти службы включают следующие аспекты:
Способность межъязыкового взаимодействия.
Сборка мусора.
Безопасность.
Домены приложений.

Вопрос с типами данных решается в .NET за счет применения общей системы типов (Common Type System - CTS). CTS описывает предопределенные типы данных, которые доступны в IL, поэтому все языки, ориентированные на среду .NET, генерируют компилированный код, который в конечном итоге базируется на этих типах.

CTS описывает не просто примитивные типы данных, а целую богатую иерархию типов, включающую хорошо определенные точки, в которых код может определять свои собственные типы. Иерархическая структура общей системы типов (CTS) отражает объектно-ориентированную методологию одиночного наследования IL

Байт-код

Во-первых, байт-код программы в отличие от двоичного машинного кода полностью безопасен. Предварительно он проверяется виртуальной машиной Java. Такая проверка гарантирует корректность программы. Этот этап занимает очень важное место в общей модели безопасности Java, позволяя избежать разрушения и потери данных и свести практически к нулю вероятность сбоя или "зависания" системы в результате выполнения недопустимого кода. Технология Java поддерживает модель сетевых вычислений "толстый сервер - тонкий клиент", в соответствии с которой, код формируется в одном месте, хранится и обслуживается - в другом, а для выполнения на локальных станциях распространяется по требованию.

Другим важным преимуществом байт-кода является его высокая плотность или, иными словами, относительно небольшое число байт, необходимое для представления программы. Измерение образов одной и той же программы, написанной на языке Java и C++ и скомпилированной соответственно в виде байт-кода и машинных инструкций, показывает, что объем машинных команд приблизительно в два раза превышает объем байт-кода. (В некоторых случаях разница может оказаться еще более значительной.) Все это не только уменьшает стоимость хранения приложений на Java, но и существенно повышает пропускную способность любой сетевой архитектуры, что особенно важно при работе в беспроводных сетях или в другой медленной сети.

Когда вы пишете код Java, вы обычно планируете использовать его на виртуальной машине Java (JVM). Иными словами, ваш код компилируется в байт-код, а этот байт-код работает под управлением JVM. C#, в свою очередь, обычно работает в общеязыковой исполняющей среде (CLR) от Microsoft. C#, как и Java, компилируется в байт-код.

2. Сборки (assembly) в среде .NET. Проблема версионности сборок и ее решение.

Сборка — двоичный файл, содержащий исполняемый код программы или (реже) другой подготовленный для использования информационный продукт.

В рамках технологии .NET — двоичный файл, содержащий управляемый код. Когда компилятор платформы .NET создает EXE или DLL модуль, содержимое этого модуля называется сборкой. Сборка содержит в себе: номер версии, метаданные и инструкции IL.

Номер версии в .NET

Номер версии сборки в .NET состоит из двух частей: информационной версии в виде строки и версии совместимости в виде идентификатора из четырёх целых чисел. Например, мы создали сборку с информационной версией TheNewAssembly. Предположим, что её версия совместимости выглядит так: 1.35.6.2, где:

- Первый идентификатор — основной номер версии.

- Второй идентификатор — дополнительный номер версии.

- Третий идентификатор — номер сборки.

- Четвёртый идентификатор — номер редакции.

Сборки имеют следующие свойства:

-Сборки реализованы как EXE- или DLL-файлы.

-Сборки можно поместить в глобальный кэш сборок, чтобы обеспечить их использование несколькими приложениями.

-В глобальный кэш сборок могут быть включены только сборки со строгими именами.

-Сборки загружаются в память только по мере необходимости.

-Для программного получения сведений о сборок используется класс reflection.

-Если нужно загрузить сборку только для ее проверки, используйте метод, подобный ReflectionOnlyLoadFrom.

Строгое имя состоит из удостоверения сборки, включающего ее простое текстовое имя, номер версии и сведения о языке и региональных параметрах (если они имеются), а также открытый ключ и цифровую подпись.

Строгое имя создается из файла сборки (файла, содержащего манифест сборки, который в свою очередь содержит имена и хэш-коды всех составляющих сборку файлов) с помощью соответствующего закрытого ключа.

Сборки, строгие имена которых совпадают, считаются идентичными.

Гарантировать глобальную уникальность имени сборки можно, подписав ее строгим именем.

Строгие имена, в частности, удовлетворяют следующим требованиям.

Уникальность строгого имени гарантируется использованием уникальных пар ключей. Никто не сможет создать такое же имя сборки, поскольку имя сборки, созданной с использованием одного закрытого ключа, отличается от имени сборки, созданной с использованием другого закрытого ключа.

Строгие имена защищают развитие версий сборки. Строгое имя гарантирует, что никто другой не сможет создать следующую версию данной сборки. Пользователи могут быть уверены, что загружаемая ими версия сборки и версия, с которой было разработано приложение, произведены одним и тем же издателем.

Строгие имена обеспечивают надежный контроль целостности. Успешный результат при проверке безопасности платформы .NET Framework гарантирует, что содержимое сборки не было

изменено после ее формирования. Однако стоит отметить, что строгие имена сами по себе не подразумевают такой же уровень доверия, который обеспечивается, например, при использовании цифровой подписи и сертификатов.

При использовании ссылки на сборку со строгим именем можно пользоваться определенными преимуществами, например, отслеживанием версий и защитой имен. Если же затем сборка со строгим именем ссылается на сборку с простым именем (которая не имеет указанных преимуществ), то преимущества использования сборки со строгим именем теряются, и опять становятся возможными конфликты DLL-библиотек. Таким образом, сборки со строгими именами могут ссылаться только на другие сборки со строгими именами.

ВЕРСИОННОСТЬ

Управление версиями сборок, использующих среду CLR, производится полностью на уровне сборки. Конкретная версия сборки и версии зависимых от нее сборок указываются в манифесте сборки. Политика управления версиями по умолчанию для среды выполнения заключается в том, что приложения могут выполняться только с версиями, с которыми они были разработаны и протестированы, если иное не переопределено явной политикой использования версий в файлах конфигурации (в файле конфигурации приложения, файле политики издателя и файле конфигурации администратора компьютера).

Управление версиями выполняется только для сборок со строгими именами

Среда выполнения предпринимает несколько шагов для разрешения запроса привязки сборок.

-Проверяет исходную ссылку на сборку для определения версии сборки, с которой будет связано приложение.

-Проверяет все применимые файлы конфигурации для использования политики управления версиями.

-Определяет правильную сборку на основании исходной ссылки на сборку и любого указанного в файлах конфигурации перенаправления, а также версию, которая должна связываться с вызывающей сборкой.

-Проверяет глобальный кэш сборок и указанные в файлах конфигурации базы кода, а затем проверяет папку приложения и вложенные папки с помощью правил проверки, описанных в разделе в обнаружение сборок среде выполнения

Разрешение запроса привязки сборки:

Сведения о версии

В каждой сборке есть два различных способа задания сведений о версии.

Номер версии сборки, который наряду с именем сборки и сведениями о языке и региональных параметрах является частью удостоверения сборки. Это номер используется средой выполнения для применения политики управления версиями и играет ключевую роль в процессе разрешения типов на этапе выполнения.

Информационная версия, которая представляет собой строку с дополнительными сведениями о версии, служащую исключительно в информационных целях.

Номер версии сборки

Каждая сборка имеет номер версии, являющийся частью ее удостоверения. Следовательно, две сборки, имеющие разные номера версий, рассматриваются средой выполнения как совершенно разные сборки. Этот номер версии физически представляется в виде строки из четырех частей следующего формата:

<основная версия >.< дополнительная версия>.<номер построения>.<редакция>

Например, в версии "1.5.1254.0" число "1" представляет основную версию, "5" — младший номер версии, "1254" — номер построения, а "0" — номер редакции.

Номер версии сохраняется в манифесте сборки вместе с другими данными удостоверения, включая имя сборки и открытый ключ, а также сведения о связях и удостоверениях других подключаемых к приложению сборок.

При построении сборки средство разработки записывает сведения о зависимостях каждой сборки, на которую имеется ссылка, в манифест сборки. Среда выполнения использует эти номера версий вместе с конфигурационными данными, установленными администратором, приложением или издателем для загрузки нужной версии сборки, на которую имеется ссылка.

С целью управления версиями среда выполнения разделяет обычные сборки и сборки со строгими именами. Проверка версий производится только для сборок со строгими именами.

Информационная версия сборки

Информационная версия сборки представляет собой строку, которая добавляет к сборке дополнительные данные и служит только для информации. Она не используется на этапе выполнения. Информационная версия (в текстовом формате) соответствует описанию продукта на рынке, данным о комплектации или названии продукта. Эта версия не используется средой выполнения. Например, информационная версия может быть задана как "Common Language Runtime version 1.0" или "NET Control SP 2". Эти сведения выводятся в элементе "Версия продукта" на вкладке "Версия" диалогового окна свойств файла в Microsoft Windows.

3. Объектная модель в среде .NET и языке C#. Общая система типов данных в среде .NET. Ссылочные и скалярные (value-type) типы данных. Упаковка и распаковка скалярных типов данных в среде .NET.

Объектная модель представления данных оперирует такими понятиями, как класс и объект. Классы определяют структуру данных и представляют собой набор атрибутов (текстовая строка, целое число, изображение и т.д.). Представители класса (объекты) имеют определенную структуру и могут содержать другие объекты, образуя произвольную иерархическую структуру. Объекты могут наследовать свойства, содержание и поведение объектов, которые в них содержатся.

Ссылочные типы данных. Объектная модель в среде .NET и языке C#.

Переменные ссылочных типов, называемые объектами, сохраняют ссылки на фактические данные.

Ключевые слова для объявления ссылочных типов:

Class

Interface

Delegate

Object

string

Class

Классы объявляются с помощью ключевого слова class.

В отличие от C++, в C# допускается только одиночное наследование. Другими словами, класс может наследовать реализацию только от одного базового класса. Однако класс может реализовать несколько интерфейсов. В приведенной ниже таблице приведены примеры наследования класса и реализации интерфейса.

Наследование	**Пример**
Отсутствует	class ClassA { }
Одиночный	class DerivedClass: BaseClass { }
Отсутствует, реализует два интерфейса	class ImplClass: IFace1, IFace2 { }
Одиночное, реализует один интерфейс	class ImplDerivedClass: BaseClass, IFace1 { }

Во вложенных классах разрешены только следующие уровни доступа: protected и private. объявить универсальные классы, имеющие параметры типа. Класс может содержать объявления следующих членов.

Конструкторы Деструкторы Константы Поля Методы Свойства Индексаторы Операторы События Делегаты Классы Интерфейсы Структуры. Типы, объявленные в классе без модификатора доступа, по умолчанию являются private.

Interface

Интерфейс содержит только подписи методов, свойств, событий или индексаторов. Интерфейс может быть членом пространства имен или класса и содержать подписи следующих членов:

Методы

Свойства

Индексаторы

События

Интерфейс способен наследовать от одного или нескольких базовых интерфейсов. Если в списке базовых типов содержится базовый класс и интерфейсы, то базовый класс должен стоять в списке на первом месте. Класс, реализующий интерфейс, может явным образом реализовывать члены этого интерфейса. Явно реализованный член можно вызвать только через экземпляр интерфейса, но не через экземпляр класса.

Дополнительные сведения и примеры кода с явной реализацией интерфейса см. в разделе Явная реализация интерфейса (руководство по программированию в C#).

В следующем примере демонстрируется реализация интерфейса. В этом примере интерфейс содержит объявление свойства, а класс содержит реализацию.

interface IPoint

Delegate

Объявление типа делегата аналогично подписи метода. Оно имеет возвращаемое значение и некоторое число параметров какого-либо типа:

public delegate void TestDelegate(string message);

public delegate int TestDelegate(MyType m, long num);

Ключевое слово **delegate** используется для объявления ссылочного типа, который может быть использован для инкапсуляции именованного или анонимного метода.

Делегаты аналогичны используемым в языке C++ указателям на функции, но являются строго типизированными и безопасными.

Делегаты являются основой событий.

Экземпляры делегата могут создаваться путем его связывания с именованным или анонимным методом.

Делегат должен быть создан при помощи метода или лямбда-выражения, имеющего совместимые возвращаемый тип и входные параметры. Дополнительные сведения о допустимой степени вариации сигнатур методов см. в разделе Ковариация и контрвариация делегатов.

Для использования с анонимными методами делегат и код, который должен быть связан с ним, должны быть объявлены вместе. В этом разделе рассматриваются оба способа создания экземпляров делегатов.

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.

String

Тип данных string — это последовательность, содержащая ни одного или любое число знаков Юникода. В платформе.NET Framework string является псевдонимом для String.

Несмотря на то, что тип string является ссылочным типом, операторы равенства (== и !=) определены для сравнения значений объектов типа string, а не ссылок. Это упрощает проверку равенства строк. Пример.

```
string a = "hello";

string b = "h";

// Append to contents of 'b'

b += "ello";

Console.WriteLine(a == b);

Console.WriteLine((object)a == (object)b);
```

В этом примере отображается "True", а затем "False", поскольку содержимое строк одинаково, но a и b ссылаются на разные экземпляры строк.

Оператор + служит для объединения строк.

Строки являются неизменяемыми: содержимое строкового объекта невозможно изменить после создания объекта, хотя из-за синтаксиса изменения кажутся возможными. Например, при написании этого кода компилятор на самом деле создает новый строковый объект для новой последовательности знаков, а переменная b по-прежнему содержит "h".

Общая система типов данных в среде .NET. Скалярные и ссылочные типы данных.

Типы, переменные и значения

Библиотека классов платформы .NET Framework определяет набор встроенных числовых типов, а также более сложных типов, представляющих широкое разнообразие логических конструкций, например, файловую систему, массивы объектов и даты.

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, необходимое для 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.

-Задание типов в объявлениях переменных

При объявлении переменной или константы в программе необходимо либо задать ее тип, либо использовать ключевое слово var, чтобы дать возможность компилятору определить его.

C# предоставляет стандартный набор встроенных числовых типов для представления целых чисел, значений с плавающей запятой, логических выражений, текстовых символов, десятичных значений и других типов данных. Существуют также встроенные типы string и object. Они доступны для использования в любой программе C#. Дополнительные сведения о встроенных типах см. в разделе Справочные таблицы по типам (справочник по C#).

Пользовательские типы

Конструкции структура, класс, интерфейс и перечисление используются для создания собственных пользовательских типов. Сама библиотека классов платформы .NET Framework является коллекцией пользовательских типов, предоставленной корпорацией Microsoft, которую можно использовать в собственных приложениях. По умолчанию наиболее часто используемые типы в библиотеке классов доступны в любой программе C#. Другие становятся доступными только при явном добавлении ссылки проекта на сборку, в которой они определены. Если компилятор имеет ссылку на сборку, то можно объявить переменные (и константы) типов, объявленных в сборке в исходном коде. Дополнительные сведения см. в разделе Библиотека классов .NET Framework.

Система общих типов CTS

Важно понимать две фундаментальные точки о системе типов в .NET Framework:

Она поддерживает принцип наследования. Типы могут быть производными от других типов, которые называются базовыми типами. Производный тип наследует (с некоторыми ограничениями) методы, свойства и другие члены базового типа. Базовый тип, в свою очередь, может быть производным от какого-то другого типа, при этом производный тип наследует члены обоих базовых типов в иерархии наследования. Все типы, включая встроенные числовые типы, например, System.Int32 (ключевое слово int), в конечном счете являются производными от одного базового типа, который является System.Object (ключевое слово C#: объектом). Эта унифицированная иерархия типов называется Система общих типов CTS (CTS).

Каждый тип в CTS определен либо как тип значения, либо как ссылочный тип. Сюда включены все пользовательские типы в библиотеке классов платформы .NET Framework, а также собственные пользовательские типы.

Типы, определяемые с помощью ключевого слова struct, являются типами значений; все встроенные числовые типы являются structs.

Типы, определяемые с помощью ключевого слова class, являются ссылочными типами.

Правила времени компиляции и поведение времени выполнения ссылочных типов отличается от правил времени компиляции и поведения времени выполнения типов значений.

Типы значений

Типы значений являются производными от System.ValueType, являющегося производным от System.Object. Типы, производные от System.ValueType, имеют особое поведение в среде CLR. Переменные типа значения напрямую содержат их значения, что означает, что память встроена в контекст, в котором объявлена переменная. Не существует отдельного размещения кучи или служебных данных сборки мусора для переменных типа значения.

Существует две категории типов значений: структура и перечисление.

Встроенные числовые типы являются структурами, и к их свойствам и методам можно получить доступ.

Но значения объявляются и присваиваются им, как если бы они были простыми не статическими типами:

Типы значений являются запечатанными, что означает, например, что нельзя произвести тип от System.Int32, и нельзя определить структуру для наследования от любого пользовательского класса или структуры, поскольку структура может наследовать только от System.ValueType. Однако структура может реализовать один или несколько интерфейсов. Можно выполнить приведение типа структуры в тип интерфейса; это приведет к операции упаковки-преобразования для создания программы-оболочки структуры внутри объекта ссылочного типа в управляемой куче. Операции упаковки-преобразования возникают при передаче типа значения методу, принимающему System.Object в качестве входного

параметра. Ключевое слово struct используется для создания собственных пользовательских типов значений. Обычно структура используется как контейнер для небольшого набора связанных переменных

Другой категорией типов значений является перечисление. Перечисление определяет набор именованных интегральных констант. Например, перечисление System.IO.FileMode в библиотеке классов платформы .NET Framework содержит набор именованных констант целого типа, которые задают, как должен быть открыт файл.

Все перечисления наследуются от System.Enum, который наследуется от System.ValueType. Все правила, применимые к структурам, также применяются к перечислениям.

Ссылочные типы

Тип, определенный как класс, делегат, массив или интерфейс, является ссылочным типом. Во время выполнения при объявлении переменной ссылочного типа переменная содержит значение null до явного создания экземпляра объекта с помощью оператора new или назначения его объекту, который был создан в другом месте, с помощью new.

Интерфейс должен быть инициализирован вместе с объектом класса, который его реализует.

При создании объекта память размещается в управляемой куче, и переменная хранит только ссылку на расположение объекта. Для типов в управляемой куче требуются служебные данные и при их размещении, и при их удалении сборщиком мусора. Однако сборка мусора также в высокой степени оптимизирована, и в большинстве сценариев она не создает проблем с производительностью. Дополнительные сведения о сборке мусора см. в разделе Автоматическое управление памятью.

Все массивы являются ссылочными типами, даже если их члены являются типами значений. Массивы являются неявно производными от класса System.Array, но объявляются и используются они с упрощенным синтаксисом.

Ссылочные типы полностью поддерживают наследование. При создании класса можно наследовать от любого другого интерфейса или класса, который не определен как запечатанный, а другие классы могут наследовать от этого класса и переопределять виртуальные методы.

Типы литеральных значений

В C# литеральные значения получают тип от компилятора. Можно задать, как числовой литерал должен быть типизирован, путем добавления буквы в конце номера. Например, чтобы задать, что значение 4,56 должно обрабатываться как число с плавающей запятой, добавьте после номера "f" или "F": 4.56f. При отсутствии добавленной буквы компилятор определит тип для литерала.

Литералы являются типизированными и все типы в конечном счете являются производными от System.Object.

Неявные типы, анонимные типы и типы, допускающие значение NULL

Как уже говорилось ранее, можно неявно типизировать локальную переменную (но не члены класса) с помощью ключевого слова var. Переменная все же получает тип во время компиляции, но тип предоставляется компилятором.

Дополнительные сведения см. в разделе Неявно типизированные локальные переменные (Руководство по программированию в C#).

В некоторых случаях неудобно создавать именованный тип для простых наборов связанных значений, которые не будут сохранены или переданы за пределы метода. Для этой цели можно создать анонимные типы. Дополнительные сведения см. в разделе Анонимные типы (Руководство по программированию в C#).

Обычные типы значений не могут иметь значение null. Однако можно создать типы значений, допускающие значение NULL, путем привязки ? после типа. Например, int? является типом int, который

также может иметь значение null. В CTS типы, допускающие значения NULL, являются экземплярами универсального типа структуры System.Nullable<T>. Типы, допускающие значение NULL, особенно полезны при передаче данных в базы данных и из них, в которых числовые значение могут быть равны NULL. Дополнительные сведения см. в разделе Типы, допускающие значения NULL (руководство по программированию на C#).

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

■ **Когда переменной значимого типа присваивается другая переменная значи-мого типа, выполняется копирование всех ее полей. Когда переменной ссы-лочного типа присваивается переменная ссылочного типа, копируется только ее адрес.**

■ **Вследствие сказанного в предыдущем абзаце, несколько переменных ссылоч-ного типа могут ссылаться на один объект в куче, благодаря чему, работая с одной переменной, можно изменить объект, на который ссылается другая пе-ременная. С другой стороны, каждая переменная значимого типа имеет соб-ственную копию данных «объекта», и операции с одной переменной значимого типа не повлияют на другую переменную.**

Упаковка и распаковка размерных типов данных в среде .NET.

Упаковка-преобразование представляет собой процесс преобразования типа значения в тип object или любой другой тип интерфейса, реализуемый этим типом значения. Операция распаковки-преобразования извлекает тип значения из объекта. В следующем примере выполнена операция упаковки-преобразования целочисленной переменой i, которая присвоена объекту o.

```
int i = 123;

object o = (object)i;  // boxing
```

Можно затем выполнить операцию распаковки-преобразования объекта o и назначить его целочисленной переменной i:

```
o = 123;

i = (int)o;  // unboxing
```

Производительность

По сравнению с простыми операциями присваивания операции упаковки-преобразования и распаковки-преобразования являются весьма затратными процессами с точки зрения вычислений. При выполнении упаковки-преобразования типа значения необходимо создать и разместить новый объект. Объем вычислений при выполнении операции распаковки-преобразования, хотя и в меньшей степени, но тоже весьма значителен.

Упаковка–преобразование

Упаковка используется для хранения типов значений в куче "сбора мусора". Упаковка представляет собой неявное преобразование типа значения в тип object или любой другой тип интерфейса, реализуемый этим типом значения. При упаковке типа значения в куче выделяется экземпляр объекта и выполняется копирование значения в этот новый объект.

Рассмотрим следующее объявление переменной типа значения.

```
int i = 123;
```

Следующий оператор неявно применяет операцию упаковки к переменной i.

```
object o = i;  // Implicit boxing
```

Результат этого оператора создает ссылку на объект o в стеке, которая ссылается на значение типа int в куче. Это значение является копией значения типа значения, назначенного переменной i. Разница между двумя этими переменными, i и o, продемонстрирована на рисунке ниже.

Упаковка-преобразование:

Можно также выполнять упаковку явным способом, как в следующем примере, однако явная упаковка не является обязательной.

Распаковка-преобразование

Распаковка является явным преобразованием из типа object в тип значения или из типа интерфейса в тип значения, его реализующее. Операция распаковки состоит из следующих действий.

-Проверка экземпляра объекта на то, что он является упакованным значением заданного типа значения.

-Копирование значения из экземпляра в переменную типа-значения.

В следующих операторах показаны операции по упаковке и распаковке.

```
int i = 123;          // a value type

object o = i;         // boxing

int j = (int)o;  // unboxing
```

На следующем рисунке представлен результат выполнения предыдущих операторов.

Распаковка-преобразование:

Для успешной распаковки типов значений во время выполнения необходимо, чтобы экземпляр, который распаковывается, был ссылкой на объект, предварительно созданный с помощью упаковки экземпляра этого типа значения. Попытка распаковать null или ссылку в несовместимый тип значения вызовет InvalidCastException.

4. Модели ручной и полуавтоматической утилизации динамической памяти (на основе счетчиков ссылок, на основе иерархии владения, на основе тегирования памяти), их сравнительная характеристика.

Модель со счетчиком ссылок. Стремление сделать уничтожение объектов автоматическим, причем в рамках существующих языков программирования, породило модель утилизации памяти на основе счетчика ссылок.

В модели со счетчиком ссылок с каждым объектом ассоциируется целочисленный счетчик ссылок. Обычно он хранится в одном из полей объекта (хотя может быть «навешен» и снаружи). При создании объекта этот счетчик устанавливается в нулевое значение, а потом увеличивается на единицу при создании каждой новой ссылки на объект. При пропадании каждой ссылки значение счетчика уменьшается на единицу, и когда оно становится равным нулю, объект уничтожается (оператором delete). Таким образом, программисту не нужно думать о том, когда следует уничтожить объект, — это происходит автоматически, как только пропадает последняя ссылка на него.

Увеличение и уменьшение счетчика ссылок выполняется с помощью двух специальных методов объекта, в технологии COM называемых AddRef и Release. Метод AddRef вызывается при любом копировании ссылки, а также при ее передаче в качестве параметра подпрограммы. Метод Release вызывается при пропадании или обнулении ссылки, например, в результате выхода программы за область видимости ссылки или при завершении подпрограммы, в которую ссылка была передана в качестве параметра.

Очевидный недостаток этой модели — наличие дополнительных накладных расходов на элементарное копирование ссылок. Еще более серьезный недостаток состоит в том, что счетчики ссылок не учитывают возможных циклических связей между объектами. В этом случае счетчики ссылок никогда не уменьшаются до нуля, что ведет к «утечкам памяти».

Для решения проблемы циклических связей используется следующий прием. Ссылки делят на два вида: «сильные» ссылки и «слабые» ссылки. Сильные ссылки влияют на счетчик ссылок, а слабые ссылки — нет. При уничтожении объекта слабые ссылки автоматически обнуляются. Для доступа к объекту слабую ссылку нужно предварительно превратить в сильную (это предотвращает уничтожение объекта во время операций с ним).

Деление ссылок на виды больше запутывает программиста, чем помогает ему. При написании программы ответить на вопрос, какого вида должна быть ссылка, порой затруднительно. Кроме того, в программе все равно возникает опасность циклических связей, образованных сильными ссылками. Попытка минимизации количества сильных ссылок (вплоть до одной на каждый объект) и повсеместное использование слабых ссылок приводят нас по сути к модели с ручным освобождением памяти, с той лишь разницей, что уничтожение объекта выполняется не вызовом оператора delete, а обнулением главной ссылки на объект. Единственная проблема, которая при этом решается, это проблема «зависших» ссылок.

Модель с иерархией владения. Анализ структуры многих программ показывает, что динамические объекты часто объединяются в иерархию. Например, в программах с графическим пользовательским интерфейсом главный объект управления программой содержит в себе объекты окон, а те в свою очередь содержат объекты панелей и кнопок. Отношением подчиненности могут быть связаны не только объекты пользовательского интерфейса, но и любые данные в программе. Используя эту особенность, можно реализовать модель утилизации памяти, которая будет существенно более надежна, чем предыдущие.

Модель с иерархией владения основана на том, что при создании любого объекта ему назначается объект-владелец, отвечающий за уничтожение подчиненных объектов. Создав объект и назначив ему владельца, можно больше не заботиться о том, что ссылки на него пропадут и произойдет «утечка памяти». Этот объект будет обязательно уничтожен при удалении владельца.

Объект можно уничтожить принудительно, даже если у него есть владелец. При этом объект либо изымается из списка подчиненных объектов своего владельца, либо помечается как уничтоженный для предотвращения повторного уничтожения.
Объект может быть создан без владельца, и тогда он требует явного уничтожения. Модель управления временем жизни объектов без владельца ничем не отличается от уже рассмотренной модели с ручным освобождением памяти.
Модель с иерархией владения не избавляет программиста полностью от необходимости явно освобождать память, однако значительно сокращает риск «утечек памяти». Эта модель, так же как и предыдущие, не решает проблему фрагментации памяти, но позволяет более успешно бороться с «зависшими» указателями, например, путем рассылки сообщений об уничтожении объектов по иерархии. Обрабатывая эти сообщения, объекты-получатели могут обнулять сохраненные ссылки на уничтожаемые объекты.
Модель с иерархией владения применяется во многих графических библиотеках и средах визуального программирования (для управления компонентами), она успешно использовалась авторами этой статьи для управления объектами в САПР СБИС.
Модель с иерархией владения иногда совмещается с моделью на основе счетчиков ссылок. Такая гибридная модель используется, например, в новейшей технологии драйверов для ОС Windows .
Модель с автоматической «сборкой мусора» и принудительным освобождением памяти(ТЕГИРОВАНИЕ). Наличие «сборки мусора» означает, что программист может быть уверен: система следит за потерей ссылок на объекты и устраняет «утечку памяти». Наличие безопасного принудительного освобождения памяти означает, что программист вправе уничтожить объект; при этом память объекта возвращается системе, а все имеющиеся на него ссылки становятся недействительными (например, обнуляются). Эта модель, называем/ая нами моделью с автоматической «сборкой мусора» и принудительным освобождением памяти, на самом деле не нова и уже давно применяется в компьютерах «Эльбрус» (на основе одноименного процессора) и AS/400 (на основе процессора PowerPC), которые обеспечивают очень эффективную реализацию этой модели за счет аппаратной поддержки.
На каждое машинное слово в этих компьютерах отводится два дополнительных бита, называемых битами тегов. Значения этих битов показывают, свободно ли машинное слово или занято, и если занято, то хранится ли в нем указатель или скалярное значение. Этими битами управляют аппаратура и операционная система, прикладным программам они недоступны. Программа не может создать ссылку сама, например, превратив в нее число или другие скалярные данные. Созданием объектов занимается система, которая размещает в памяти объекты и создает ссылки на них. При уничтожении объектов соответствующие теги памяти устанавливаются в состояние, запрещающее доступ. Попытка обратиться к свободной памяти по «зависшему» указателю приводит к аппаратному прерыванию (подобно обращению по нулевому указателю). Поскольку вся память помечена тегами, «сборщику мусора» нет необходимости анализировать информацию о типах, чтобы разобраться, где внутри объектов располагаются ссылки на другие объекты. Что более важно, ему почти не нужно тратить время на поиск недостижимых объектов, поскольку освобожденная память помечена с помощью тех же тегов .
Ниже сформулированы базовые принципы модели на уровне спецификации для языков программирования:

- Выделение динамической памяти выполняется оператором/процедурой new (это действие считается элементарным в системе). Выделенная память автоматически инициализируется нулями и всегда привязывается к типу созданного в памяти объекта.
- Уничтожение объекта — освобождение занимаемой им динамической памяти — выполняется автоматически при пропадании всех ссылок на объект. Для дефрагментации освободившихся участков памяти периодически выполняется «сборка мусора», в результате которой объекты сдвигаются, а ссылки на них корректируются.

- Объекты можно уничтожать принудительно с помощью оператора/процедуры delete. В результате этого действия все ссылки на объект становятся недействительными, а попытка последующего доступа к объекту приводит к исключительной ситуации. Дефрагментация освобожденной этим способом памяти выполняется во время «сборки мусора».

Главный вопрос, который пока остается открытым, — можно ли реализовать эту модель программно, чтобы она эффективно работала для популярных аппаратных архитектур, в которых нет тегирования памяти. Первое простейшее решение состоит в том, чтобы по каждому вызову оператора delete выполнять просмотр памяти с корректировкой недействительных ссылок. Просмотр занимает значительно меньше времени, чем полная «сборка мусора» с дефрагментацией памяти. Решение подходит для мобильных и встроенных устройств с небольшим объемом ОЗУ и без поддержки виртуальной памяти.

Второе решение основано на использовании средств аппаратной поддержки виртуальной памяти, которая существует в большинстве современных компьютерных архитектур. В этом механизме нас интересует возможность аппаратно перехватывать обращения к страницам виртуальной памяти. На самом деле страницы могут оставаться в памяти и на диск не выгружаться. Идея состоит в том, чтобы при вызове оператора delete помечать страницы, в которых располагается удаляемый объект, как отсутствующие. Обращение к данным на этих страницах будет вызывать аппаратное прерывание. Обрабатывая это прерывание, система проверяет, куда именно выполняется обращение: к освобожденному участку памяти или к занятому. Если обращение выполняется к занятому участку страницы, то запрос удовлетворяется и работа продолжается в штатном режиме. Если к освобожденному — то создается программная исключительная ситуация.

Это решение имеет очевидный недостаток — при большом количестве обращений к «живым» объектам, расположенным на одной странице рядом с удаленными объектами, будут возникать холостые аппаратные прерывания, которые снизят производительность системы. Для борьбы с этой проблемой система должна подсчитывать частоту холостых прерываний и, если ее значение превысит некоторый порог, досрочно запускать процесс обнуления недействительных ссылок.

5. Механизм сборки мусора в среде .NET. Построение графа достижимых объектов сборщиком мусора в среде .NET. Механизм поколений объектов в сборщике мусора среды .NET.

Модель с автоматической «сборкой мусора» предусматривает лишь возможность создавать объекты, но не уничтожать их. Система сама следит за тем, на какие объекты еще имеются ссылки, а на какие уже нет. Когда объекты становятся недостижимы через имеющиеся в программе ссылки (превращаются в «мусор»), их память автоматически возвращается системе.

Эта работа периодически выполняется «сборщиком мусора» и происходит в две фазы. Сначала «сборщик мусора» находит все достижимые по ссылкам объекты и помечает их. Затем он перемещает их в адресном пространстве программы (с соответствующей корректировкой значений ссылок) для устранения фрагментации памяти.

Обход графа достижимых объектов начинается от «корней», к которым относятся все глобальные ссылки и ссылки в стеках имеющихся программных потоков. Анализируя метаданные (информацию о типах данных, которая размещается внутри выполняемых модулей), «сборщик мусора» выясняет, где внутри объектов имеются ссылки на другие объекты. Следуя по этим ссылкам, он обходит все цепочки объектов и выясняет, какие блоки памяти стали свободными. После этого достижимые по ссылкам объекты перемещаются для устранения фрагментации, а ссылки на перемещенные объекты корректируются.

Эта модель вроде бы решает все проблемы: нет «утечек памяти», нет фрагментации памяти, нет «зависших» указателей.

По скорости выделения памяти эта модель сравнима со стеком, ведь выделение объекта — это по сути увеличение указателя свободной области памяти на размер размещаемого объекта. Однако по достижении этим указателем определенного предела запускается «сборка мусора», которая может потребовать много времени и привести к ощутимой задержке в работе программы. Моменты наступления таких задержек и их длительность обычно непредсказуемы. Поэтому одна из проблем «сборщика мусора» — это недетерминизм связанных с его работой задержек.

Для амортизации задержек в «сборщиках мусора» применяются различные подходы. Например, в среде .NET используется принцип поколений, основанный на том наблюдении, что объекты, создаваемые раньше, как правило, живут дольше. Вся память делится на поколения (их количество обычно соответствует числу уровней кэширования с учетом ОЗУ; в современных архитектурах обычно три поколения). Нулевое (младшее) поколение — самое маленькое по объему, первое поколение в несколько раз больше, чем нулевое, а второе в несколько раз больше, чем первое.

Алгоритм: в 0-м поколении имеется указатель, новые объекты всегда находятся в 0-м поколении. Когда в поколении не хватает места для выделения очередного объекта, запускается механизм частичной сборки мусора, который делает следующее: проходит по указателям в программе, которые указывают на 0-ое поколение, и определяет, какие блоки в 0-м поколении еще используются, а какие уже нет. Все еще используемые блоки переносятся в 1-ое поколение. Перенос – просто перенос указателя с укладыванием блоков. В программе указатели корректируются автоматически. В результате, 0-ое поколение становится полностью свободным, и обычная работа программы продолжается. Если при переносе блока из 0-го поколения в 1-ое оказывается, что 1-ое поколение переполнилось, выполняется та же операция, что и для 0-го: делается проход по всем указателям, из них отбираются только те, которые указывают не 1-ое поколение. Используемые указатели переносятся из 1-го во 2-ое поколение. Указатели в 1-м поколении корректируются.

Замечания:

1) 2-ое поколение переполняется реже 1-го
2) размер 0-го поколения такой, чтобы он помещался в КЭШ 2-го уровня

3) размер 1-го поколения такой, чтобы он умещался в КЭШ 1-го уровня (КЭШ оперативной памяти)

В связи с этим, работа в 0-м поколении – работа в процессоре.

При выделении: проверка границы (256Кб, 2М, 10М), если больше, сразу выделяем в 3-м. Если 3-ее переполнено, проводим дефрагментацию всего. Если больше 10М, надо всегда запускать сборщик мусора и упаковывать память.

Объекты создаются в младшем поколении и перемещаются в старшие поколения, пережив «сборку мусора». Последняя выполняется не во всей памяти, а лишь в тех поколениях, в которых исчерпалось свободное место, — чаще в нулевом, реже в первом и еще реже во втором поколении. Таким образом, задержек при «сборке мусора» много, но их средняя длительность небольшая .

В модели с автоматической «сборкой мусора» программный код завершения жизни объекта (метод Finalize, или, говоря иначе, деструктор) выполняется асинхронно в контексте «сборщика мусора». Момент и порядок вызова этого метода у того или иного объекта никак не детерминированы, что порождает проблему, если объект управляет некоторым ресурсом, например сетевым соединением. Открытие соединения происходит при создании и инициализации объекта, т. е. предсказуемо, а закрытие соединения — во время «сборки мусора», т. е. непредсказуемо и далеко не сразу после потери последней ссылки на объект. В результате лимит сетевых соединений или других ресурсов может временно исчерпаться.

Для решения указанной проблемы в среде .NET используется детерминированное завершение жизни объектов через интерфейс IDisposable. Этот интерфейс имеет единственный метод Dispose, который реализуется в объектах, управляющих ресурсами. Метод Dispose, как правило, освобождает ресурсы и отменяет работу процедуры-завершителя (метода Finalize), чтобы ускорить освобождение памяти. После вызова метода Dispose объект не уничтожается, а остается в памяти до тех пор, пока не пропадут все ссылки на него.

6. Завершение объектов в среде .NET. Метод Finalize. Список завершаемых объектов (finalization queue) и очередь завершения (freachable queue).

Финализацией (finalization) называется поддерживаемый CLR механизм, позволяющий объекту выполнить корректную очистку, прежде чем сборщик мусора освободит занятую им память. Любой тип, выполняющий функцию оболочки машинного ресурса, например файла, сетевого соединения, сокета, мьютекса и др., должен поддерживать финализацию. Для этого в типе реализуют метод финализации. Определив, что объект стал мусором, сборщик вызывает метод финализации объекта (если он есть).

Для определения метода финализации в C# перед именем класса нужно добавить знак тильды (~) :

```
internal sealed class SomeType {
        // Метод финализации
        ~SomeType ( ) {
        // Код метода финализации
        }
}
```

Скомпилировав этот код и проверив полученную сборку с помощью утилиты ILDasm .exe, вы увидите, что компилятор C# внес метод с именем Finalize в метаданные этого модуля. При изучении IL-кода метода Finalize также становится ясно, что код в теле метода генерируется в блок try, а вызов метода base . Finalize - в блок finally.

В предыдущих версиях спецификации C# этот метод назывался деструктором (destructor). Однако метод финализации работает совсем не так, как неуправляемый деструктор (не означает в C# детерминированное уничтожение объектов типа, как это происходит в C++). CLR не поддерживает детерминированное уничтожение, поэтому C# не может предоставить этот механизм.

Метод финализации будет вызван, даже если конструктор экземпляра этого типа выбросит исключение.

При конструировании типа метода финализации лучше избегать по ряду причин:

- Выделение памяти для объектов, поддерживающих финализацию, занимает

больше времени, так как указатели на них должны размещаться в списке финализации.

- Объекты, поддерживающие финализацию, переходят в старшие поколения,

что повышает нагрузку на память и не позволяет освободить память объекта в тот момент, когда сборщик мусора посчитает его мусором. Кроме того, все объекты, на которые прямо или косвенно ссылается этот объект, тоже переходят в старшие.

- Объекты, поддерживающие финализацию, замедляют работу приложения,

потому что каждое удаление объекта при сборке мусора требует дополнительного расхода ресурсов.

Невозможно контролировать момент исполнения метода финализации.

Методы *Finalize* вызываются при завершении сбора мусора, который происходит в результате одного из пяти следующих событий.

■ **Заполнение поколения 0** приводит к запуску сборщика мусора. Это собы-тие намного чаще остальных приводит к вызову метода *Finalize,* так как явля-ется естественным следствием создания новых объектов во время работы кода приложения.

■ **Явный вызов статического метода *Collect* объекта *System.GC*** Код может явно запросить сбор мусора у CLR. Хотя Microsoft настоятельно не рекоменду-ет так поступать, порой принудительный сбор мусора имеет смысл.

■ **Windows сообщает о нехватке памяти** Для общего мониторинга систем-ной памяти CLR использует Int32-функции *CreateMemoryResourceNotification* и *QueryMemoryResourceNotification.* Если Windows сообщает о нехватке памя-ти, CLR запускает сбор мусора, чтобы освободить нерабочие объекты и умень-шить рабочий набор процесса.

■ **Выгрузка домена приложения CLR** Выгружая домен приложения, CLR счи-тает, что ни один объект в нем не является корнем, и выполняет сбор мусора всех поколений.

■ **Закрытие CLR** CLR завершает работу после нормального завершения рабо-ты процесса (в отличие от внешнего завершения, например диспетчера задач). При этом CLR считает, что в процессе нет корней, и вызывает метод *Finalize* для всех объектов в управляемой куче. Учтите, что CLR не пытается сжать или освободить память, потому что процесс завершается, а Windows освобождает всю занятую им память.

CLR использует особый выделенный поток для вызова методов *Finalize.* В слу-чае первых четырех событий, если метод *Finalize* зацикливается, выделенный по-ток блокируется и вызов методов *Finalize* прекращается. Это очень плохо, потому что приложение не сможет освободить память, занятую объектами, у которых есть метод *Finalize,* и в итоге будет испытывать нехватку памяти.

В случае пятого события каждому методу *Finalize* дается примерно 2 секунды на то, чтобы вернуть управление программе. Если он не успевает, CLR просто за-вершает процесс и методы *Finalize* больше не вызываются. А если для вызова ме-тодов *Finalize* всех объектов требуется более 40 секунд, CLR также просто завер-шит процесс.

Когда приложение создает новый объект, оператор *new* выделяет для него па-мять из кучи. Если в типе объекта определен метод *Finalize,* прямо перед вызовом конструктора экземпляра типа указатель на объект помещается в *список завершения* (finalization list) — внутреннюю структуру данных, управляемую сборщиком му-сора. Каждая запись этого списка указывает на объект, для которого нужно выз-вать метод *Finalize,* прежде чем освободить занятую им память.

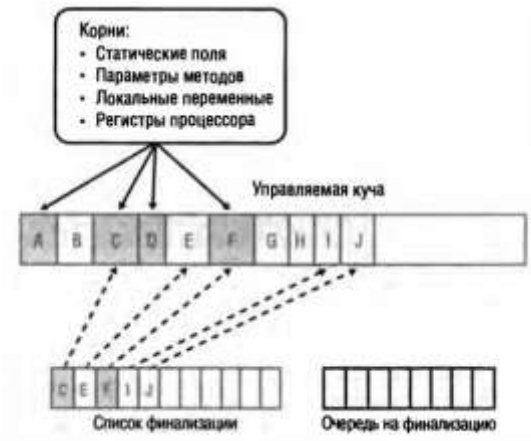


Рис. 21.5. Управляемая куча с указателями в списке финализации

Сначала сборщик мусора определяет, что объекты B, E, G, H, I и J – это мусор. Сборщик сканирует список завершения в поисках указателей на эти объекты. Обнаружив указатель, он удаляет его из списка завершения и добавляет в конец очереди freachable — еще одной внутренней структуры данных сборщика мусо-ра. Каждый указатель в этой очереди идентифицирует объект, готовый к вызову своего метода *Finalize.*

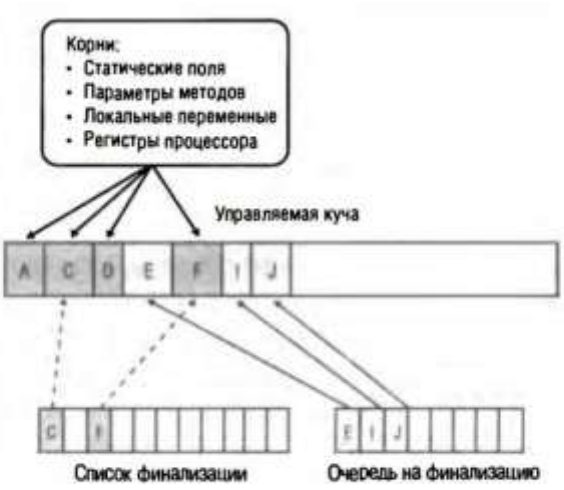


Рис. 21.6. Управляемая куча с указателями, перемещенными из списка финализации в очередь на финализацию

В CLR есть особый высокоприоритетный поток, выделенный для вызова методов финализации. Он нужен для предотвращения возможных проблем синхронизации, которые могли бы возникнуть при использовании вместо него одного из потоков приложения с обычным приоритетом. При пустой очереди на финализацию (это ее обычное состояние) данный поток бездействует. Но как только в ней появляются элементы, он активизируется и последовательно удаляет элементы из очереди, вызывая соответствующие методы финализации. Особенности работы данного потока запрещают исполнять в методе финализации любой код, имеющий какие-либо допущения о потоке, исполняющем код. Например, в методе финализации следует избегать обращения к локальной памяти потока.

Очередь на финализацию можно было бы назвать очередью ссылок на объекты, доступные для финализации. Эту очередь можно рассматривать и просто как корень, подобно статическим полям, которые являются корнями. Таким образом, **находящийся в очереди на финализацию объект доступен и не является мусором.** Короче говоря, если объект недоступен, сборщик считает его мусором. Далее, когда сборщик перемещает ссылку на объект из списка финализации в очередь на финализацию, объект перестает считаться мусором, а это означает, что занятую им память освобождать нельзя. По мере маркировки объектов из очереди другие объекты, на которые ссылаются их поля ссылочного типа, также рекурсивно помечаются - все эти объекты должны пережить сборку мусора. На этом этапе сборщик завершает поиск мусора, и некоторые объекты, идентифицированные как мусор, перестают считаться таковым - они как бы воскрешают. Сборщик мусора сжимает освобожденную память, а особый поток CLR очищает очередь на финализацию, выполняя метод финализации для каждого объекта из очереди. Вызванный снова, сборщик обнаруживает, что финализированные объекты стали мусором, так как ни корни приложения, ни очередь на финализацию больше на них не указывают. Память, занятая этими объектами, попросту освобождается. Важно понять, что для освобождения памяти, занятой объектами, требующими финализации, сборку мусора нужно выполнить дважды. На самом деле может понадобиться и больше операций сборки мусора, поскольку объекты переходят в следующее поколение.

7. Модель детерминированного освобождения ресурсов в среде .NET. Интерфейс IDisposable и его совместное использование с завершителем (методом Finalize).

Нельзя гарантировать вызов метода финализации в определенное время, а поскольку он не является открытым методом, пользователь класса не может вызвать его явно. Возможность детерминированного уничтожения или закрытия объекта часто полезна при работе с неуправляемыми типами, которые играют роль оболочки машинных ресурсов, таких как файлы, соединения с базой данных или битовые карты.

Для решения указанной проблемы в среде .NET используется детерминированное завершение жизни объектов через интерфейс IDisposable. Этот интерфейс имеет единственный метод Dispose, который реализуется в объектах, управляющих ресурсами. Метод Dispose, как правило, освобождает ресурсы и отменяет работу процедуры-завершителя (метода Finalize), чтобы ускорить освобождение памяти. После вызова метода Dispose объект не уничтожается, а остается в памяти до тех пор, пока не пропадут все ссылки на него.

Применение интерфейса IDisposable на практике выявило новые проблемы. Оказалось, что после вызова метода Dispose в программе могут оставаться ссылки на объект, находящийся уже в некорректном состоянии. Программе никто не запрещает обращаться по этим физически доступным, но логически «зависшим» ссылкам и вызывать у некорректного объекта различные методы. Метод Dispose может вызываться повторно, в том числе рекурсивно. Кроме того, в программе с несколькими вычислительными потоками может происходить асинхронный вызов метода Dispose для одного и того же объекта.

Для решения проблем этого метода программистам было предписано делать следующее: 1) определять в объекте булевский флаг, позволяющий выяснить, работал ли в объекте код завершения; 2) блокировать объект внутри метода Dispose на время работы кода завершения; 3) игнорировать повторные вызовы метода Dispose, проверяя упомянутый булевский флаг; 4) в начале public-методов проверять, находится ли объект уже в завершенном состоянии, и если это так, то создавать исключение класса ObjectDisposedException.

Заканчивая критику «сборщиков мусора», укажем еще на одну серьезную проблему с ними — легальную «утечку памяти». Если в модели с ручным освобождением объектов «утечка памяти» возникает из-за невыполненных операторов delete, то в модели с автоматической «сборкой мусора» — из-за невыполненного обнуления ссылок. Такой вид «утечек памяти» характерен для программного кода, в котором одними объектами регистрируются обработчики событий в других объектах. Программисты порой забывают отключать обработчики событий, в результате ассоциированные объекты остаются в памяти, несмотря на кажущееся отсутствие ссылок на них в программе.

Учтите, что методы Dispose и Close - это просто средство заставить объект выполнить самоочистку в определенное время. Эти методы не управляют памятью, занятой объектом в управляемой куче. Это значит, что методы объекта можно вызывать даже после его очистки.

```
// Реализация интерфейса IDisposable  сигнализирует пользователям
// этого класса о том, что он поддерживает эталон освобождения ресурсов
public abstract class SafeHandle : CriticalFinalizerObject, IDisposable
    // Этот открытый метод можно вызвать, чтобы наверняка
    // уничтожить ресурс. Он реализует метод Dispose интерфейса IDisposable
    public void Dispose ( ) {
        //Вызов метода, реально выполняющего очистку
        Dispose ( true );
    }
    // Этот открытый метод можно вызвать в место Dispose
    public void Close( ){
```

```
            Dispose (true);
        }
        // При сборке мусора этот метод финализации вызывается, чтобы закрыть ресурс
        ~SafeHandle() {
            //Вызов метода, реально выполняющего очистку
            Dispose(false);
        }
        //Этот общий метод реально выполняет очистку. Его вызывают метод финализации,
        //а так же методы Dispose и Close. Поскольку этот класс не изолированный, метод
        //является защищенным и виртуальным. Если бы класс был изолирован, метод был //бы
        закрытым
        protected virtual void Dispose(Boolean disposing) {
            if(disposing) {
                //Объект явно уничтожается или закрывается, но не финализируется
                //Поэтому в этой условной инструкции обращение к полям,
                //ссылающимся на другие объекты, безопасно для кода, тк
                //метод финализации этих объектов еще не вызывался

                //Классу SafeHandle здесь делать ничего не нужно
            }
            //Выполняется уничтожение\закрытие или финализация объекта. А
            //происходит вот что: Если ресурс уже освобожден, просто возвращается
            //управление.
            //Если значение owsHandle равно false, возвращается управление.
            //Устанавливается флаг, указывающий, что данный ресурс был освобожден.
            //Вызов виртуального метода ReleaseHandle.
            //Вызов GC.SuppressFinalize(this), запрещающий вызов метода финализации.
        }
}
```

8. «Слабые» ссылки (тип данных WeakReference) и кэширование данных в среде .NET.

В программировании слабая ссылка (англ. weak reference) — специфический вид ссылок на динамически создаваемые объекты в системах со сборкой мусора. Отличается от обычных ссылок тем, что не учитывается сборщиком мусора при выявлении объектов, подлежащих удалению. Ссылки, не являющиеся слабыми, также иногда именуют «сильными». Такие ссылки используются так же, как и обычные, но не влияют на сборку мусора, поскольку не учитываются механизмом подсчёта ссылок, и объект, на который существуют такие ссылки, может быть удалён, если только на него не существует обычных ссылок (которые в этом контексте могут именоваться «сильными ссылками»).

Слабые ссылки поддерживаются на уровне системных библиотек в C# — класс System.WeakReference. Когда требуется сохранить слабую ссылку, создаётся объект-референер (экземпляр класса WeakReference), которому передаётся сильная ссылка. Переданная сильная ссылка тут же освобождается, и в референере хранится ссылка, существование которой не может помешать сборщику мусора удалить соответствующий объект. Когда требуется воспользоваться слабой ссылкой, у референера вызывается метод get() (в C# — свойство Target), который возвращает сильную ссылку на объект, если он ещё существует, или null, если объект уже удалён сборщиком мусора. Поскольку get возвращает именно сильную ссылку, во время её использования объект удалён не будет, но после того, как использование этой ссылки прекратилось, он снова становится доступным для сборки мусора. Таким образом, при каждом использовании слабой ссылки программа должна обязательно проверить её на равенство NULL.

Чтобы слабые ссылки на уже не существующие объекты не засоряли память, системные библиотеки предоставляют механизмы для учёта таких ссылок. Вариантом такого механизма являются очереди ссылок — специальные объекты, которые передаются референеру при создании. Когда сборщик мусора уничтожает объект, на который ссылается слабая ссылка, он помещает в ранее переданную очередь ссылок ссылку на соответствующий референер. Таким образом, программе доступен список референеров, содержащих «мёртвые» ссылки, и она может выполнить их удаление в любой подходящий момент.

Кэш данных является наиболее часто используемым хранилищем информации в приложениях ASP.NET. Это кэш, хранящийся в памяти и доступный через объект Cache. Чтобы добавить новый элемент в кэш, вам понадобится указать ключ для него (где ключом является строка). Аналогично для того, чтобы получить элемент из кэша, вам понадобится сослаться на него, используя ключ.

Элементы управления ObjectDataSource, SqlDataSource, AccessDataSource и XmlDataSource содержат свойства, которые указывают элементам управления источником данных, что необходимо кэшировать информацию в кэше данных. Данное кэширование выполняется автоматически и не требует написания кода. Ключевыми свойствами являются:

EnableCaching - значение типа Boolean, от которого зависит, будет произведено кэширование или нет. По умолчанию оно установлено в False. Вам понадобится установить данное значение в True, чтобы элемент управления источником данных выполнял кэширование.

CacheDuration - значение типа Integer, которое указывает длительность кэширования информации (в секундах).

CacheExpirationPolicy - указывает тип окончания кэширования - может быть установлен в Absolute (абсолютное) либо Sliding (плавное). Абсолютное означает то, что элемент кэша будет удален через определенное в CacheDuration время с момента его записи в кэше. Плавный тип подразумевает удаление из кэша через время, указанное в CacheDuration с момента последнего доступа к данным. По умолчанию установлено в Absolute.

CacheKeyDependency - значение типа String, которое указывает на дополнительный элемент в кэше, служащий в качестве отношения. Если указано данное значение, то когда изменяется кэшированный элемент в CacheKeyDependency, источник данных также изменит информацию в кэше.

Одним из недостатков кэширования информации является старение данных. Кроме удаления кэшированной информации по истечению какого-либо срока, существуют другие способы удаления. Элементы управления источником данных автоматически высвобождают данные из кэша, если вызывается метод Insert, Update или Delete элемента управления источником данных. Вкратце, методы элемента управления источником данных изменяют хранилище данных, как об этом свидетельствуют их имена, и также могут быть декларативно настроены. Когда выполняется какой-либо из этих методов, элемент управления источником данных изменяет хранилище данных, таким образом благоразумно удаляя кэшированную информацию, поскольку известно, что она уже не является настолько новой.

9.**Динамические массивы в среде .NET и языке C#. Виды многомерных массивов и их представление в оперативной памяти.(НЕТ РИСУНКА ПРЕДСТАВЛЕНИЯ В ОП)**

Массив представляет собой механизм, позволяющий рассматривать набор элементов как единую коллекцию. Исполняющая среда Microsoft .NET (CLR) поддерживает *одномерные (single-dimension), многомерные (multidimension) и нерегулярные (jagged)* массивы. Базовым для всех массивов является абстрактный класс **`System.Array`**, производный от **`System.Object`**. Значит, массивы всегда относятся к ссылочному типу и размещаются в управляемой куче, а переменная в приложении содержит не элементы массива, а ссылку на массив.
Рассмотрим пример:

```
Int32[] myIntegers;     // Объявление ссылки на массив
myIntegers = new Int32[100]; // Создание массива типа Int32 из 100 элементов
```

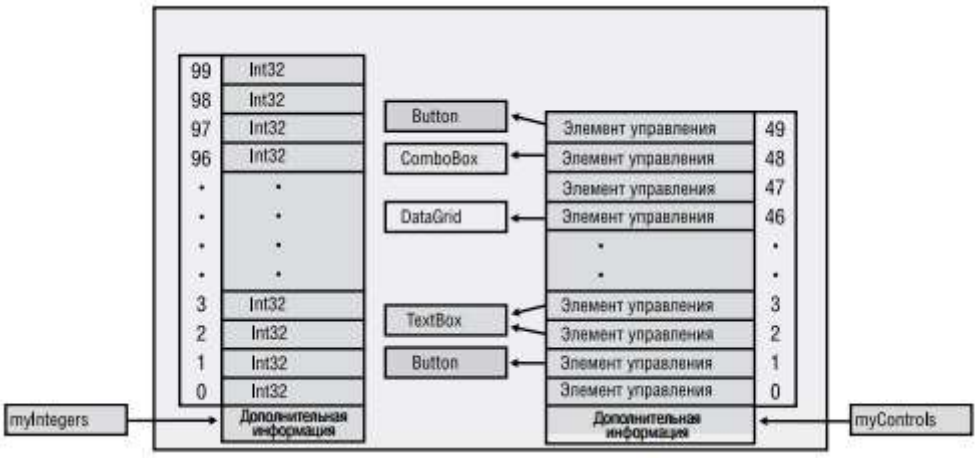
В первой строке объявляется переменная **`myIntegers`**, которая будет ссылаться на одномерный массив элементов типа **`Int32`**. Вначале ей присваивается значение **`null`**, так как память под массив пока не выделена. Во второй строке выделяется память под 100 значений типа **`Int32`**; и всем им присваивается начальное значение 0. Поскольку массивы относятся к ссылочным типам, блок памяти для хранения 100 неупакованных экземпляров типа **`Int32`** выделяется в управляемой куче.

Можно также создать массивы с элементами ссылочного типа:

```
Control[] myControls; // Объявление ссылки на массив
myControls = new Control[50]; // Создание массива из 50 ссылок на переменную Control
```

Переменная **`myControls`** из первой строки может указывать на одномерный массив ссылок на элементы **`Control`**. Вначале ей присваивается значение **`null`**, ведь память под массив пока не выделена. Во второй строке выделяется память под 50 ссылок на **`Control`**, и все они инициализируются значением **`null`**.

На рисунке показано, как выглядят массивы значимого и ссылочного типов в управляемой куче



На этом рисунке показан массив **`Controls`** после выполнения следующих инструкций:

```
myControls[1] = new Button();
myControls[2] = new TextBox();
myControls[3] = myControls[2]; // Два элемента ссылаются на один объект
myControls[46] = new DataGrid();
myControls[48] = new ComboBox();
myControls[49] = new Button();
```

По возможности нужно ограничиваться одномерными массивами с нулевым начальным индексом, которые называют иногда *SZ-массивами*, или *векторами*. Векторы обеспечивают наилучшую производительность, поскольку для операций с ними используются команды промежуточного языка (Intermediate Language, IL)

Создание многомерных массивов:

```
// Создание двухмерного массива типа Double
Double[,] myDoubles = new Double[10, 20];
// Создание трехмерного массива ссылок на строки
String[,,] myStrings = new String[5, 3, 10];
```

CLR поддерживает также нерегулярные (jagged) массивы — то есть «массивы массивов». Производительность одномерных нерегулярных массивов с нулевым начальным индексом такая же, как у обычных векторов. Однако обращение к элементу нерегулярного массива означает обращение к двум или больше массивам одновременно.

```
// Создание одномерного массива из массивов типа Point
Point[][] myPolygons = new Point[3][];
// myPolygons[0] ссылается на массив из 10 экземпляров типа Point
myPolygons[0] = new Point[10];
// myPolygons[1] ссылается на массив из 20 экземпляров типа Point
myPolygons[1] = new Point[20]; //
myPolygons[2] ссылается на массив из 30 экземпляров типа Point
myPolygons[2] = new Point[30];
// вывод точек первого многоугольника
for (Int32 x = 0; x < myPolygons[0].Length; x++)
      Console.WriteLine(myPolygons[0][x]);
```

Базовый класс System.Array

Рассмотрим объявление переменной массива:

```
FileStream[] fsArray;
```

Объявление переменной массива подобным образом приводит к автоматическому созданию типа **FileStream[]** для домена приложений. Тип **FileStream[]** является производным от **System.Array** и соответственно наследует оттуда все методы и свойства. Для их вызова служит переменная **fsArray**. Это упрощает работу с массивами, ведь в классе **System.Array** есть множество полезных методов и свойств, в том числе **Clone**, **CopyTo**, **GetLength**, **GetLongLength**, **GetLowerBound**, **GetUpperBound**, **Length** и **Rank**. Класс **System.Array** содержит также статические методы для работы с массивами, в том числе **AsReadOnly**, **BinarySearch**, **Clear**, **ConstrainedCopy**, **ConvertAll**, **Copy**, **Exists**, **Find**, **FindAll**, **FindIndex**, **FindLast**, **FindLastIndex**, **ForEach**, **IndexOf**, **LastIndexOf**, **Resize**, **Reverse**, **Sort** и **TrueForAll**. В качестве параметра они принимают ссылку на массив. У каждого из этих методов существует множество перегруженных версий. Более того, для многих из них имеются обобщенные перегруженные версии, обеспечивающие контроль типов во время компиляции и высокую производительность.

10. Делегаты в среде .NET и механизм их работы. Анонимные делегаты. События в среде .NET; реализация событий посредством делегатов.

Событие представляет собой сообщение, посылаемое объектом, чтобы сигнализировать о совершении какого-либо действия. Это действие может быть вызвано в результате взаимодействия с пользователем, например при нажатии кнопки мыши, или может быть обусловлено логикой работы программы. Объект, вызывающий событие, называется отправителем события. Объект, который захватывает событие и реагирует на него, называется получателем события.

При обмене событиями классу отправителя событий не известен объект или метод, который будет получать (обрабатывать) сформированные отправителем события. Необходимо, чтобы между источником и получателем события имелся посредник (или механизм подобный указателю). .NET Framework определяет специальный тип (Delegate), обеспечивающий функциональные возможности указателя функции.

Делегат является классом, который может хранить ссылку на метод. В отличие от других классов класс делегата имеет подпись и может хранить ссылки только на методы, соответствующие этой подписи. Таким образом, делегат эквивалентен строго типизированному указателю функции или обратному вызову. Хотя делегаты имеют и другие направления использования, здесь будут рассматриваться только функциональные возможности делегатов по обработке событий. Объявление делегата является достаточным для определения класса делегата. Объявление предоставляет подпись делегата, а среда CLR обеспечивает реализацию. В следующем примере показан порядок объявления делегата событий.

C#

```
public delegate void AlarmEventHandler(object sender, AlarmEventArgs e);
```

Синтаксис сходен с синтаксисом объявления метода. Однако зарезервированное слово **delegate** сообщает компилятору, что AlarmEventHandler является типом делегата. По соглашению делегаты событий в .NET Framework имеют два параметра: источник, вызвавший событие, и данные для события.

Экземпляр делегата AlarmEventHandler можно привязать к любому методу, который соответствует его подписи, такому как метод AlarmRang класса WakeMeUp, как показано в следующем примере.

C#

```
public class WakeMeUp
{
   // AlarmRang has the same signature as AlarmEventHandler.
   public void AlarmRang(object sender, AlarmEventArgs e)
   {...};
   ...
}
```

Пользовательские делегаты событий необходимы только в случаях, когда событие создает данные для события. Многие события, включая некоторые события пользовательского интерфейса, например щелчки мышью, не создают данных для события. В таких ситуациях делегат события, предоставляемый библиотекой классов для события без данных, System.EventHandler является целесообразным. Его объявление приводится ниже.

C#

```
delegate void EventHandler(object sender, EventArgs e);
```

Делегаты событий являются многоадресными. Это означает, что они могут хранить ссылки на несколько методов обработки событий. Дополнительные сведения см. в разделе Delegate. Делегаты позволяют осуществлять гибкий и детальный контроль при обработке событий. Делегат действует как диспетчер событий для класса, вызвавшего событие, обслуживая для события список зарегистрированных обработчиков событий.

11. Исключительные ситуации и реакция на них в среде .NET. Особенности исключительной ситуации класса ThreadAbortException. Создание программного кода, устойчивого к прерыванию с помощью исключения ThreadAbortException.

```
private void SomeMethod() {
try {
// Внутрь блока try помещают код, требующий корректного
// восстановления работоспособности или очистки ресурсов.
}
catch (InvalidOperationException) {
// В такие блоки catch помещают код, который должен восстанавливаться
// после исключений типа InvalidOperationException (или любого исключения,
// производного от него).
}
catch (IOException) {
// В такие блоки catch помещают код, который должен восстанавливаться // после исключений типа IOException (или любого исключения, // производного от него).
catch {
// В такие блоки catch помещают код, который должен
// восстанавливаться после исключений любого типа.
// После перехвата исключений их, как правило,
// генерируют повторно.
throw;
}
finally {
// Внутрь блоков finally помещают код, выполняющий очистку ресурсов после любых действий, начатых в блоке try. Код из этого блока исполняется ВСЕГДА независимо от того, было исключение или нет.
}
// Код после блока finally исполняется, если в блоке try не было исключения или если исключение было перехвачено блоком catch и не было сгенерировано то же самое или другое исключение.)
```

«Допустим, у меня есть приложение, которое должно считать из файла структуру размером в 20 байт, но файл оказался лишь 10-байтовым. В этом случае я не ожидаю встретить конец файла при чтении, но это неожидан-но происходит. Наверное, здесь должно возникнуть исключение, не так ли?» Фак-тически большинство файлов содержит структурированные данные. Довольно ред-ко бывает так, что приложение читает из файла байт за байтом, тут же обрабаты-вая прочитанные байты, пока не достигнет конца файла. Поэтому я думаю, логичнее будет, если при попытке чтения за пределами файла метод Read всегда будет ге-нерировать исключение.

Есть масса причин, по которым вызванный метод может сгенерировать исключение:

¦ если недостаточно памяти в стеке, генерируется исключение Stack-Overflowlixception\

¦если не удается обнаружить сборку, в которой определен тип, генери-руется исключение PileNotFoundException-,

¦если IL-код метода не поддается верификации, генерируется исключе-ние VetificationException;

¦если недостаточно памяти для JIT-компиляции IL-кода, генерируется исключение OutOJMemoryException.

Фильтрация исключений среды выполнения

Перехватываемые и обрабатываемые исключения можно отфильтровать либо по типу, либо по некоторым критериям, определяемым пользователем.

Обработчики с фильтрацией по типу управляют определенным типом исключения (или производные из этого типа классы).В следующем примере показан обработчик с фильтрацией по типу, позволяющий перехватить определенное исключение, в данном случае исключение FileNotFoundException.

```
catch(FileNotFoundException e)
{
Console.WriteLine("[Data File Missing] {0}", e);
}
```

Обработчики с пользовательской фильтрацией перехватывают и обрабатывают исключения на основе требований к исключению, определяемых пользователем.

ThreadAbortException - Исключение, выдаваемое при вызове метода Abort. Этот класс не может наследоваться.

При вызове метода Abort для уничтожения потока, общеязыковая среда выполнения выдает исключение ThreadAbortException. Исключение ThreadAbortException является особым исключением, которое может быть перехвачено, но снова возникает в конце блока catch. При возникновении этого исключения среда выполнения перед тем, как завершить поток, выполняет все блоки finally. Так как поток может выполнять код в блоках finally неограниченное время или вызывать метод Thread.ResetAbort для отмены аварийного завершения потока, нет гарантии, что поток когда-либо закончит работу. Чтобы дождаться окончания работы прерванного потока можно вызвать метод Thread.Join. Вызов метода Join является блокирующим, возврата не происходит до действительного окончания выполнения потока.

Когда среда CLR останавливает фоновые потоки после того, как все потоки переднего плана в управляемом исполняемом файле завершились, метод Thread.Abort не используется. Следовательно, нельзя использовать перехват исключения ThreadAbortException для обнаружения прерывания фоновых потоков средой CLR.

```
 public static void DoWork() {
      try {
         for(int i=0; i<100; i++) {
            Console.WriteLine("Thread - working.");
            Thread.Sleep(100);
         }
```

```
    }
    catch(ThreadAbortException e) {
        Console.WriteLine("Thread - caught ThreadAbortException - resetting.");
        Console.WriteLine("Exception message: {0}", e.Message);
        Thread.ResetAbort();
    }
    Console.WriteLine("Thread - still alive and working.");
    Thread.Sleep(1000);
    Console.WriteLine("Thread - finished working.");
}
```

12. Средства многопоточного программирования в среде .NET. Автономные потоки. Пул потоков.

Понятие многозадачности существует и поддерживается в самой операционной системе. Однако принципы многозадачного программирования, которые есть в среде .NET отличаются от того, что есть в ОС.

Если в ОС существует понятие процесс и поток, то в платформе .NET существует понятие домен приложения и поток. Домен приложения – это изолированное виртуальное пространство процесса. У одного процесса таких виртуальных пространств может быть несколько.

Потоки бывают:

1) выделенные и

2) потоки в пуле.

Они принципиально отличаются. За каждым выделенным потоком стоит поток операционной системы. Поток в пуле потоков – это набор программных потоков, за которыми так же стоят потоки ОС, но эти потоки пересоздаются, не уничтожаются, их количество может динамически изменяться и они предоставляют очень гибкую модель асинхронного программирования.

Независимо от того, потоки находятся в пуле или они выделенные, потоки так же делятся на:

1) фоновый (автоматически уничтожается, когда прекращает работу процесс)

2) основной (программный процесс не завершается, пока не завершены основные потоки.)

Фоновые потоки явно уничтожать не нужно. В пуле потоков – все потоки фоновые. Потокам, которые вы создаёте сами, вы можете указать, является ли он фоновым или основным.

У потока есть идентификатор – это идентификатор, выдаваемый самой платформой .NET. Помимо этого, существует идентификатор, выданный самой операционной системой. Они между собой отличаются. Рассмотрим пример.

Создание выделенных потоков:

```
static void DedicatedThreads()
{
Thread t1 = new Thread(DoWork, 16 * 1024);
t1.Start("o");
Thread t2 = new Thread(DoWork);
t2.Start("X");
}
public static void DoWork(object obj)
{
 Console.WriteLine(" {0}, {1}", obj, Thread.CurrentThread.ManagedThreadId);
Thread.Sleep(2000);
}
```

В методе Dedicatedthreads создаётся поток t1, путём создания объекта класса Thread, куда передаются делегат, который будет выполняться. Идентификатор потока запрашивается следующим образом: в классе Thread есть свойство – CurrentThread, возвращающее объект класса Thread для текущего потока, а уже у него берется свойства ID. Значение параметра, передаваемое для потока, передаётся в вызов Start. Потоку в качестве дополнительного

параметра можно передать размер стека потока. Если он не указан, то .NET передаёт стандартное значение – 1 мб. Этот размер стека является серьёзным ограничением, если мы создаём много потоков. В ОС Windows можно создать столько потоков, на сколько хватит памяти. А количество памяти определяется как раз таки суммарным размером стеков всех потоков. Для того, что бы можно было создавать больше потоков, нужно указывать значительно меньшее значение стека. Размер стека для потоков из пула потоков менять нельзя.

Для потока при запуске можно установить свойство:

t1.IsBackground = false;

Кроме того, у потока существует свойство, позволяющее отличать его от потока в пуле потока:

t2.IsThreadPoolThread

Если поток запускается для выполнения какого-либо действия, то потоки могут дожидаться выполнения друг друга. К примеру, запустив эти 2 потока, можно дождаться их выполнения через вызов:

t1.Join();

Join означает, что поток, вызывающий этот метод, останавливается в этой точке, пока указанный поток не закончится.

Кроме выделенных потоков, есть возможность запуска программных процедур в пуле потоков.

```
static void ThreadPoolThreads()
{
ThreadPool.QueueUserWorkItem(DoWork, "o");
ThreadPool.QueueUserWorkItem(DoWork, "X");
    }
```

Метод ThreadPoolThreads выполняет всё то же самое, но для этого используется вызов статической процедуры QueueUserWorkItem класс ThreadPool. Если ваша программа ни разу не вызывает метод QueueUserWorkItem, значит в вашей программе никакого пула потоков не будет. Пул потоков создаётся при первой необходимости, как только программа в первый раз его запрашивает. Кстати, здесь, на уровне компилятора всё выглядит немного иначе:

ThreadPool.QueueUserWorkItem(new WaitCallback(DoWork), "o");

Так более короткую версию интерпретирует компилятор, но синтаксис C# позволяет вам создать делегат, просто указав подпрограмму. Важно помнить, что главный программный поток не ждёт завершения потоков помещенный в пул потоков. Начнут выполнение ровно столько потоков, сколько есть в пуле свободных потоков. Остальные будут ждать.

Когда надо использовать пул потоков, а когда выделенные потоки?

1) Если количество действий, выполняемых параллельно велико, то создание большого количества выделенных потоков приводит к тому, что начинает одновременно выполняться слишком много действий и ОС очень много времени тратит на переключение задач.

2) Если вы для выполнения короткого действия, которое необходимо выполнять асинхронно к работе все программы, создаёте каждый раз поток, то это большие накладные расходы на его создание и уничтожение. Поэтому предпочитать всегда нужно пул потоков.

Но есть и случаи, когда невозможно обойтись пулом потоков:

1) если вам нужно выполнять какое-то действие постоянно. В этом случае, если вы будете использовать пул потоков, то в случае загрузки системы, поставленная в очередь процедура будет выполняться не сразу, а лишь когда до неё дойдет дело.

Для того, что бы основной поток мог дождаться завершения работы всех заданий из пула потоков, можно завести статическую переменную, которая инкрементируется при добавлении нового задания в пул потоков и декрементируется перед завершение работы каждого потока, выполняющего задания. В основном же потоке мы будет ждать, пока значение статической переменной не будет равно нулю:

```
DedicatedThreads();

while (Count != 0)

Thread.Yield();
```

Вызов Yield – отдаёт процессорное время текущего потока другому потоку операционной системы. Каждому потоку выделяется определённый квант времени. Если поток работает и вызывает Yield в какой-то точке, то в этой точке его квант времени отдаётся какому-то другому потоку. Поэтому, если какой-то поток выполняет свои действия очень быстро (в данном случае проверка переменной), то он не потребляет весь квант времени, а оставшийся квант времени отдаётся ОС, которая более эффективно его потратит. Потом опять, когда поток получит квант времени, он снова проверит значение переменной и отдаст квант времени обратно системе.

Можно также не просто отдавать процессорное время, а делать вызов:

```
Thread.Sleep(0);
```

Засыпание приведёт к тому, что поток не просто отдаст процессорное время, а отдаст его на заданное количество миллисекунд. Это может быть более эффективно, если мы не хотим очень быстро обнаруживать изменение этой переменной.

Однако прямое обращение к переменной Count в многопоточной среде недопустимо, поскольку операции ++ и – не являются атомарными.

В пуле потоков существует понятие минимального количество потоков и максимального. В пуле потоков есть метод:

```
GetMinThreads();
```

Этот метод возвращает минимальное количество потоков, существующий в пуле. При чём, возвращаемых значений 2. Количество рабочих потоков(4) и количество портов завершения(4). Если вы ставите в очередь на выполнение какую-то подпрограмму, то по завершении этой подпрограммы, в очередь на выполнение ставится другая подпрограмма, которая будет выполнена для сигнализации извещения о том, что выполнено какое-то действие. На этот случай специально выделен резерв дополнительных потоков, которые всегда держаться активными, что бы можно было сообщать о завершении каких-то действий.

Ограничение максимального количества потоков в пуле нужно потому, что, если вы запускаете задачи и у вас нет максимального ограничения, то вы получаете ситуацию как с выделенными потоками. Если вы вызовете GetMaxThread(), вы увидите, что максимальное значение потоков равно 1023(а потоков завершения - 1000). Dead lock - все потоки в пуле начали выполнять действие, поставили в очередь задачи и стали ждать, при этом нет ни одного свободного потока, который мог бы извлечь делегат, выполнить его и дать шанс разбудить другой поток. Поэтому для работы пула потока был выбран следующий алгоритм работы.

- Максимальное количество потоков, которое может быть в пуле установлено в теоретический максимум. Потоки в пуле не добавляются сразу же, когда они нужны. Они создаются только если количество задач в пуле большое и они не уменьшаются в течение некоторого времени. Тогда пул потоков начинает добавлять новые потоки, но не чаще чем 1 поток в секунду. Если

за это время, пока они нарастают, нагрузка падает, то на этом рост количества потоков останавливается. Кроме того, если потоки достигаются какого-то предела, к примеру, 24, то они выделяются не раз в секунду, а раз, к примеру, в минуту. Интенсивность выделения потоков падает с течением времени.

13. Асинхронные операции в среде .NET и асинхронный вызов делегатов.

Предположим, что строится однопоточное приложение, вызывающее метод на удаленном объекте, который выполняет длительный запрос к базе данных, загружает большой документ либо выводит 500 строк текста во внешний файл. На протяжении выполнения этих операций приложение "замирает" на некоторый период времени. И пока эта задача не будет завершена, все поведение программы (вроде активизации пунктов меню, щелчков в панели инструментов или вывода на консоль) будет заморожено.

Отсюда вопрос: как заставить делегат выполнить назначенный ему метод в отдельном потоке выполнения, чтобы имитировать "одновременное" выполнение многочисленных задач? К счастью, каждый тип делегата .NET автоматически оснащен такой возможностью.

public delegate int BinaryOp (int x, int y);

Когда компилятор C# обрабатывает ключевое слово delegate, динамически сгенерированный класс определяет два метода с именами BeginInvoke() и EndInvoke(). Учитывая определение делегата BinaryOp, эти методы прототипированы следующим образом:

public sealed class BinaryOp : System.MulticastDelegate

{

// Используется для асинхронного вызова метода,

public IAsyncResult BeginInvoke(int x, int y, AsyncCallback cb, object state);

// Используется для получения возвращаемого значения вызванного метода,

public int EndInvoke(IAsyncResult result);

}

Первый стек параметров, переданный BeginInvoke(), будет основан на формате делегата C# (в случае BinaryOp — два целых). Последними двумя аргументами всегда будут System.AsyncCallback и System.Object.

Метод BeginInvoke() всегда возвращает объект, реализующий интерфейс IAsyncResult, в то время как EndInvoke() ожидает единственный параметр совместимого с IAsyncResult типа. Совместимый с IAsyncResult объект, возвращаемый из BeginInvoke() — это в основном связывающий механизм, который позволяет вызывающему потоку получить позже результат вызова асинхронного метода через EndInvoke(). Интерфейс IAsyncResult (находящийся в пространстве имен System) определен следующим образом:

public interface IAsyncResult

{

object AsyncState { get; }

WaitHandle AsyncWaitHandle { get; }

bool CompletedSynchronously { get; }

bool IsCompleted { get; }

}

Чтобы заставить делегат BinaryOp вызывать Add()(метод для сложения подаваемых на вход параметров) асинхронно:

// Вызвать Add() во вторичном потоке.

BinaryOp b = new BinaryOp(Add) ;

IAsyncResult IftAR = b.BeginInvoke(10, 10, null, null);

// Выполнить другую работу в первичном потоке...

Console.WriteLine("Doing more work in Main()!");

// Получить результат метода Add() по готовности,

int answer = b.EndInvoke(iftAR);

Console.WriteLine ("10 + 10 is {0}.", answer);

Как только сообщение "Doing more work in Main()!" будет выведено на консоль, вызывающий поток блокируется и ожидает, пока вторичный поток завершится. Чтобы позволить вызывающему потоку определять, когда асинхронно вызванный метод завершит свою работу, в интерфейсе IAsyncResult предусмотрено свойство IsCompleted. Если метод еще не завершился, то IsCompleted вернет false, и вызывающий поток может продолжать заниматься своей работой. Если Incompleted вернет true, то вызывающий поток может получить результат в наименее блокирующей манере.

while('IftAR.IsCompleted)

{

Console.WriteLine("Doing more work in Main()' ");

Thread.Sleep (1000);

}

// Теперь известно, что метод Add() завершен,

int answer = b.EndInvoke(IftAR);

В дополнение к свойству IsCompleted интерфейс IAsyncResult предлагает свойство AsyncWaitHandle для реализации более гибкой логики ожидания. Это свойство возвращает экземпляр типа WaitHandle, который предоставляет метод по имени WaitOne(). Преимущество WaitHandle.WaitOne() в том, что можно задавать максимальное время ожидания. Если это время истекает, то WaitOne() возвращает false.

while (!IftAR.AsyncWaitHandle.WaitOne (1000, true))

{

Console.WriteLine("Doing more work in Main()!");

}

Вместо опроса делегата о том, завершился ли асинхронно вызванный метод, было бы более эффективно заставить вторичный поток информировать вызывающий поток о завершении выполнения задания. Чтобы включить такое поведение, необходимо передать экземпляр делегата System.AsyncCallback в качестве параметра методу BeginInvoke(). Если передается объект AsyncCallback, делегат автоматически вызовет указанный метод по завершении асинхронного вызова.

На заметку! Метод обратного вызова будет вызван во вторичном потоке, а не в первичном.

Как и любой делегат, AsyncCallback может вызывать только методы, соответствующие определенному шаблону, который в данном случае требует единственного параметра IAsyncResult и ничего не возвращает:

void MyAsyncCallbackMethod(IAsyncResult ltfAR)

class Program

```
{
static void Main(string [ ] args)
{
        …
        BinaryOp b = new BinaryOp(Add);
        IAsyncResult iftAR = b.Beginlnvoke(10, 10, new AsyncCallback(AddComplete) , null);
        // Предположим, здесь выполняется какая-то другая работа...
        …
}
static void AddComplete(IAsyncResult itfAR)
{
        Console.WriteLine("AddComplete() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
        Console.WriteLine("Your addition is complete");
}
}
```

Статический метод AddComplete() будет вызван делегатом AsyncCallback по завершении метода Add ().

Входной параметр IAsyncResult, передаваемый цели делегата AsyncCallback — это на самом деле экземпляр класса AsyncResult (обратите внимание на отсутствие префикса I), определенного в пространстве имен System.Runtime.Remoting.Messaging. Статическое свойство AsyncDelegate возвращает ссылку на первоначальный асинхронный делегат, который был создан где-то в другом месте.

```
static void AddComplete(IAsyncResult itfAR)
{
…
// Теперь получить результат.
AsyncResult ar = (AsyncResult)itfAR;
BinaryOp b = (BinaryOp)ar.AsyncDelegate;
Console.WriteLine("10 + 10 is {0}.", b.EndInvoke(itfAR));
}
```

Последний аргумент метода Beginlnvoke() (который до сих пор был равен null) позволяет передавать дополнительную информацию о состоянии методу обратного вызова от первичного потока. Поскольку этот аргумент прототипирован как System.Object, его можно передавать в любом типе данных, если только метод обратного вызова знает, чего в нем ожидать.

```
static void Main(string[] args)
{
IAsyncResult lftAR = b.Beginlnvoke(10, 10,
new AsyncCallback(AddComplete),
```

"Main() thanks you for adding these numbers.");

}

Для получения этих данных в контексте AddComplete () используется свойство AsyncState входящего параметра IAsyncResult.

static void AddComplete(IAsyncResult itfAR)

{

string msg = (string)itfAR.AsyncState;

Console.WriteLine(msg);

}

14. Синхронизация программных потоков с помощью блокировок. Реализация блокировок в среде .NET. Атомарные (Interlocked-) операции. Volatile-переменные. Барьеры памяти (Memory Barrier). Глобальные переменные, локальные относительно программного потока.

Чтобы предотвратить повреждение ресурса по причине одновременного до-ступа многих потоков, нужно использовать конструкции синхронизации потоков.

Многие из конструкций синхронизации потоков в CLR в действительности всего лишь объектно-ориентированные оболочки классов, построенные на базе кон-струкций синхронизации потоков Win32. В конце концов, потоки CLR — это по-токи Windows, а это значит, что именно Windows планирует и контролирует их синхронизацию.

Ключевое слово lock используется для того, чтобы выполнение блока кода не прерывалось кодом, выполняемым в других потоках. Для этого нужно получить взаимоисключающую блокировку для данного объекта на время длительности блока кода.

Оператор lock начинается с ключевого слова lock, которому в качестве аргумента указывается объект, и за которым следует блок кода, который должен выполняться одновременно только в одном потоке.

Аргумент, предоставляемый ключевому слову lock, должен быть объектом на основе ссылочного типа; он используется для определения области блокировки.

Строго говоря, объект, предоставляемый для lock, используется только для того, чтобы уникальным образом определить ресурс, к которому предоставляется доступ для различных потоков, поэтому это может быть произвольный экземпляр класса. В действительности этот объект обычно представляет ресурс, для которого требуется синхронизация потоков. Например, если объект контейнера должен использоваться в нескольких потоках, то контейнер можно передать блокировке, а блок синхронизированного кода после блокировки должен получить доступ к контейнеру. Если другие потоки блокируются для того же контейнера перед доступом к нему, обеспечивается безопасная синхронизация доступа к объекту.

Как правило, рекомендуется избегать блокировки типа public или экземпляров объектов, которыми не управляет код вашего приложения. Например, использование lock(this) может привести к неполадкам, если к экземпляру разрешен открытый доступ, поскольку внешний код также может блокировать объект. Это может привести к созданию ситуаций взаимной блокировки, когда два или несколько потоков будут ожидать высвобождения одного и того же объекта. По этой же причине блокировка открытого типа данных (в отличие от объектов) может привести к неполадкам. Блокировка строковых литералов наиболее опасна, поскольку строковые литералы интернируются средой CLR. Это означает, что если во всей программе есть один экземпляр любого строкового литерала, точно такой же объект будет представлять литерал во всех запущенных доменах приложения и во всех потоках. В результате блокировка, включенная для строки с одинаковым содержимым во всем приложении, блокирует все экземпляры этой строки в приложении. По этой причине лучше использовать блокировку закрытых или защищенных членов, для которых интернирование не применяется. В некоторых классах есть члены, специально предназначенные для блокировки. Например, в типе Array есть SyncRoot. Во многих типах коллекций есть член SyncRoot.

Класс Interlocked содержит методы для синхронизации доступа к переменной, которая совместно используется несколькими потоками. Если переменная находится в общей памяти, этот механизм может использоваться потоками различных процессов. Блокируемые операции являются атомарными. Это значит, что вся операция является неделимой и не может прерываться другой блокируемой операцией с той же переменной. Это важно в операционных системах, использующих многопоточность с вытеснением, где поток может быть приостановлен

после того, как он загрузит значение из адреса памяти, но до того, как он получит возможность изменить это значение и сохранить в памяти.

Класс Interlocked выполняет следующие операции.

В платформе .NET Framework версии 2.0 метод Add добавляет к переменной целочисленное значение и возвращает новое значение переменной.

В платформе .NET Framework версии 2.0, метод Read считывает 64-разрядное целочисленное значение в ходе атомарной операции. Это полезно в 32-разрядных операционных системах, где считывание 64-разрядного целого числа обычно не является атомарной операцией.

Методы Increment и Decrement увеличивают или уменьшают переменную и возвращают результирующее значение.

Метод Exchange выполняет атомарный обмен значений для указанной переменной, возвращая текущее значение и заменяя его новым значением. Универсальная перегружаемая версия этого метода в платформе .NET Framework версии 2.0 может использоваться для обмена значений в переменной любого ссылочного типа.

Метод CompareExchange также обменивает два значения, но делает это в зависимости от результата сравнения. Универсальная перегружаемая версия этого метода в платформе .NET Framework версии 2.0 может использоваться для обмена значений в переменной любого ссылочного типа.

Современные процессоры часто позволяют реализовать методы класса Interlocked одной инструкцией. Поэтому они обеспечивают очень высокую производительность синхронизации и позволяют построить механизмы синхронизации более высокого уровня, такие как спин-блокировка.

Ключевое слово volatile указывает, что поле может быть изменено несколькими потоками, выполняющимися одновременно. Поля, объявленные как volatile, не проходят оптимизацию компилятором, которая предусматривает доступ посредством отдельного потока. Это гарантирует наличие наиболее актуального значения в поле в любое время.

Как правило, модификатор volatile используется для поля, обращение к которому выполняется через несколько потоков без использования оператора lock для сериализации доступа.

Ключевое слово volatile можно применять к полям следующих типов.

· Ссылочные типы.

· Типы указателей (в небезопасном контексте). Обратите внимание, что несмотря на то, что сам указатель Ключевое слово volatile можно применить только к полям класса или структуры. Локальные переменные не могут быть объявлены как volatile.

Если в разных потоках идёт запись в одну и ту же переменную, то их порядок в любом случае случаен. Для решения данной проблемы существует универсальный метод — добавление барьера памяти (memory barrier, memory fence). Существует несколько видов барьеров памяти: полный, release fence и accure fence.

Полный барьер гарантирует, что все чтения и записи расположенные до/после барьера будут выполнены так же до/после барьера, то есть никакая инструкция обращения к памяти не может перепрыгнуть барьер.

Accure fence гарантирует что инструкции, стоящие после барьера, не будут перемещены в позицию до барьера.

Release fence гарантирует, что инструкции, стоящие до барьера, не будут перемещены в позицию после барьера.

Еще пару слов о терминологии. Термин volatile write означает выполнение записи в память в сочетании с созданием release fence. Термин volatile read означает чтение памяти в сочетании с созданием accure fence. .NET предоставляет следующие методы работы с барьерами памяти:

· метод Thread.MemoryBarrier() создает полный барьер памяти

· ключевое слово volatile превращает каждую операцию над переменной, помеченной этим словом в volatile write или volatile read соответственно.

· метод Thread.VolatileRead() выполняет volatile read

· метод Thread.VolatileWrite() выполняет volatile write

Ознакомимся с методами Thread.VolatileWrite и Thread.VolatileRead более подробно. В MSDN о VolatileWrite написано: “Записывает значение непосредственно в поле, так что оно становится видимым для всех процессоров компьютера.” На самом деле это описание не совсем корректно. Эти методы гарантируют две вещи: отсутствие оптимизаций компилятора и отсутствие перестановок инструкций в соответствии с свойствами volatile read или write. Строго говоря метод VolatileWrite не гарантирует, что значение немедленно станет видимым для других процессоров, а метод VolatileRead не гарантирует, что значение не будет прочитано из кэша.

15. Способы прерывания программных потоков в среде .NET.

Поток автоматически завершается при выходе из метода, указанного при создании делегата Thread.Start, но иногда требуется завершить метод (следовательно, и поток) при возникновении определенных факторов. В таких случаях в потоках обычно проверяется условная переменная, в зависимости от состояния которой принимается решение об аварийном выходе из потока. Как правило, для этого в процедуру включается цикл Do-While. На опрос условной переменной уходит некоторое время. Постоянный опрос в условии цикла следует использовать лишь в том случае, если вы ожидаете преждевременного завершения потока. Доступ к условной переменной необходимо синхронизировать, чтобы воздействие со стороны других потоков не помешало ее нормальному использованию. Заблокированный поток может быть преждевременно разблокирован двумя путями: • Thread.Interrupt.

•Thread.Abort.

Это должно быть сделано из другого потока; ожидающий поток бессилен что-либо сделать в блокированном состоянии.

Метод Interrupt может вызываться только для потоков, находящихся в состоянии Wait, Sleep или Join. Если вызвать Interrupt для потока, находящегося в одном из перечисленных состояний, то через некоторое время поток снова начнет работать, а исполнительная среда инициирует в потоке исключение ThreadlnterruptedException. Это происходит даже в том случае, если поток был переведен в пассивное состояние на неопределенный срок вызовом Thread.Sleepdimeout. Infinite). Если Interrupt вызывается для неблокированного потока, поток продолжает исполнение до точки следующей блокировки, в которой и генерируется исключение ThreadInterruptedException. Это поведение освобождает от необходимости вставлять проверки которые не являются потокобезопасными, так как могут быть прерваны другим потоком между оператором if и worker.Interrupt.

Вызов Interrupt без должных на то оснований таит в себе опасность, так как любой метод framework-а, или другой сторонний метод в стеке вызовов может получить его раньше, чем ваш код, которому он предназначался. Все, что для этого требуется – чтобы поток хотя бы кратковременно встал на простой блокировке или синхронизации доступа к ресурсу, и любой ждущий своего часа Interrupt тут же сработает. Если метод изначально не разрабатывался с учетом возможности такого прерывания (с соответствующим кодом очистки в блоках finally), объекты могут остаться в неработоспособном состоянии, или ресурсы будут освобождены не полностью.

Прерывать исполнение потока безопасно, если вы точно знаете, чем сейчас занят поток.Блокированный поток также может быть принудительно освобожден при помощи метода Abort. Эффект аналогичен Interrupt, только вместо ThreadInterruptedException генерируется ThreadAbortException. Кроме того, это исключение будет повторно сгенерировано в конце блока catch (в попытке успокоить поток навеки), если только в блоке catch не будет вызван Thread.ResetAbort. До вызова Thread.ResetAbort ThreadState будет иметь значение AbortRequested. Большое отличие между Interrupt и Abort состоит в том, что происходит, если их вызвать для неблокированного потока. Если Interrupt ничего не делает, пока поток не дойдет до следующей блокировки, то Abort генерирует исключение непосредственно в том месте, где сейчас находится поток – может быть, даже не в вашем коде. Аварийное завершение неблокированного потока может иметь существенные последствия.

16. Мониторы в среде .NET. Ожидание выполнения условий с помощью методов Wait и Pulse.

Вероятно, самой популярной из гибридных конструкций синхронизации потоков является класс Monitor, обеспечивающий взаимоисключающее блокирование с зацикливанием, владением потоком и рекурсией. Данная конструкция используется чаще других потому, что является одной из самых старых, для ее поддержки в C# существует встроенное ключевое слово, с ней по умолчанию умеет работать JIT компилятор, а CLR пользуется ею от имени приложения. Класс Monitor является статическим, и его методы принимают ссылки на любой объект кучи. Управление полями эти методы осуществляют в блоке синхронизации заданного объекта.

Очевидно, что привязка блока синхронизации к каждому объекту в куче является достаточно расточительной, особенно если учесть тот факт, что большинство объектов никогда не пользуются этим блоком. Чтобы снизить потребление памяти, разработчики CLR применили более эффективный вариант реализации описанной функциональности. Во время инициализации CLR выделяется массив блоков синхронизации. Как уже не раз упоминалось в этой книге, при создании объекта в куче с ним связываются два дополнительных служебных поля. Первое поле — указатель на объект-тип — содержит адрес этого объекта в памяти. Второе поле содержит индекс блока синхронизации (sync block index), то есть индекс в массиве таких блоков. В момент конструирования объекта этому индексу присваивается значение –1, что означает отсутствие ссылок на блок синхронизации. Затем при вызове метода Monitor.Enter CLR обнаруживает в массиве свободный блок синхронизации и присваивает ссылку на него объекту. То есть привязка объекта к блоку синхронизации происходит «на лету». Метод Exit проверяет наличие потоков, ожидающих блока синхронизации. Если таких потоков не обнаруживается, метод возвращает индексу значение –1, означающее, что блоки синхронизации свободны и могут быть связаны с какими-нибудь другими объектами. Указатель на объект-тип объектов A, B и C ссылается на тип T. Это говорит о принадлежности всех трех объектов к одному и тому же типу. Объект тип также находится в куче и подобно всем остальным объектам обладает двумя служебными членами: индексом блока синхронизации и указателем на объект-тип. То есть блок синхронизации можно связать с объектом-типом, а ссылку на этот объект можно передать методам класса Monitor. Кстати, массив блоков синхронизации при необходимости может увеличить количество блоков, поэтому не стоит беспокоиться, что при одновременной синхронизации нескольких объектов блоков не хватит.

Индекс блоков синхронизации объектов в куче (включая объекты-типы) может ссылаться на запись в массиве блоков синхронизации CLR

Реализация класса Monitor как статического создаст ряд дополнительных проблем:

- Если тип объекта-представителя является производным от System.MarshalByRefObject, на такой объект может ссылаться переменная. При передаче методам класса Monitor ссылки на такой представитель блокируется представитель, а не представляемый им объект.
- Если поток вызывает метод Monitor.Enter и передает в него ссылку на объект-тип, загруженный нейтрально по отношению к домену, поток блокирует этот тип во всех доменах процесса. Это известная недоработка CLR, нарушающая декларируемую изолированность доменов. Исправить ее без потери производительности сложно, поэтому никто этим не занимается. Пользователям просто рекомендуют никогда не передавать ссылку на объект-тип в методы класса Monitor.
- Так как строки допускают интернирование, два разных фрагмента кода могут ошибочно сослаться на один и тот же объект String в памяти. При передаче ссылки на этот объект в методы типа Monitor выполнение этих двух фрагментов кода будет непреднамеренно синхронизироваться.

- При передаче строки через границу домена CLR не создает ее копию; ссылка на строку просто передается в другой домен. Это повышает производительность, и в теории все должно быть в порядке, так как объекты типа String неизменны. Но с ними, как и с любыми другими объектами, связан индекс блока синхронизации, который может изменяться. Таким образом, потоки в различных доменах начинают синхронизироваться друг с другом. Это еще одна недоработка CLR, связанная с недостаточной изолированностью доменов. Поэтому пользователям рекомендуется никогда не передавать ссылок на объекты типа String методам класса Monitor.
- Так как методы класса Monitor принимают параметры типа Object, передача им значимого типа приводит к его упаковке. В результате поток блокирует упакованный объект. При каждом вызове метода Monitor.Enter блокируется другой объект, и синхронизация потоков вообще отсутствует.
- Применение к методу атрибута [MethodImpl(MethodImplOptions.Synchronized)] заставляет JIT-компилятор окружить машинный код метода вызовами Monitor.Enter и Monitor.Exit. Если метод является экземплярным, этим методам передается this, что приводит к установлению неявно открытой блокировки. В случае статического метода этим двум методам передается ссылка на объект-тип, что приводит к потенциальной блокировке нейтрального по отношению к домену типа. Поэтому использовать данный атрибут не рекомендуется.
- При вызове конструктора типов CLR блокирует для типа объект-тип, гарантируя, что всего один поток примет участие в инициализации данного объекта и его статических полей. И снова загрузка этого типа нейтрально по отношению к домену создает проблемы. К примеру, если код конструктора типа войдет в бесконечный цикл, тип станет непригодным для использования всеми доменами в процессе. В данном случае рекомендуется по возможности избегать конструкторов типа или хотя бы делать их как можно более короткими и простыми.

К сожалению, дальше все становится только хуже. Так как разработчики привыкли в одном и том же методе устанавливать блокировку, что-то делать, а затем снимать блокировку, в C# появился упрощенный синтаксис в виде ключевого слова lock.

Рассмотрим следующий метод:

```
private void SomeMethod() {
        lock (this) {
        // Этот код имеет эксклюзивный доступ к данным...
        }
}
```

Приведенный фрагмент эквивалентен следующему:

```
private void SomeMethod() {
        Boolean lockTaken = false;
        try {
                Monitor.Enter(this, ref lockTaken);
                // Этот код имеет монопольный доступ к данным...
        } finally {
                if (lockTaken) Monitor.Exit(this);
        }
}
```

Первая проблема в данном случае состоит в принятом разработчиками C# решении, что метод Monitor.Exit лучше вызывать в блоке finally. Они считали, что это гарантирует снятие блокировки вне зависимости от происходящего в блоке try. Однако ничего хорошего в этом нет. Если в блоке try в процессе изменения состояния возникнет исключение, состояние окажется поврежденным. И снятие блокировки в блоке finally приведет к тому, что с поврежденным

состоянием начнет работать другой поток. Лучше позволить приложению зависнуть, чем оставить его работать с поврежденными данными и потенциальными брешами в защите. Кроме того, вход в блок try и выход из блока снижает производительность метода. Некоторые JIT-компиляторы не поддерживают подстановку для методов, в которых имеются блоки try, что еще больше снижает производительность. В итоге мы получаем более медленный код, к тому же допускающий доступ потоков к поврежденному состоянию. *Поэтому я крайне не рекомендую вам пользоваться инструкцией lock (Мнение автора книги – Д.Рихтера).* Теперь перейдем к переменной lockTaken типа Boolean и к проблеме, которую призвана решить эта переменная. Предположим, поток вошел в блок try и был прерван до вызова метода Monitor. После этого вызывается блок finally, но его код не должен снимать блокировку. В этом нам поможет переменная lockTaken. Ей присваивается начальное значение false, означающее, что блокировка еще не установлена. Если вызванный метод Monitor.Enter успешно получает блокировку, переменной lockTaken присваивается значение true. Блок finally по значению этой переменной определяет, нужно ли вызывать метод Monitor.Exit.

Методы Wait() и Pulse()

Рассмотрим следующую ситуацию. Поток T выполняется в кодовом блоке lock, и ему требуется доступ к ресурсу R, который временно недоступен. Что же тогда делать потоку T? Если поток T войдет в организованный в той или иной форме цикл опроса, ожидая освобождения ресурса R, то тем самым он свяжет соответствующий объект, блокируя доступ к нему других потоков. Это далеко не самое оптимальное решение, поскольку оно лишает отчасти преимуществ программирования для многопоточной среды.

Более совершенное решение заключается в том, чтобы временно освободить объект и тем самым дать озможность выполняться другим потокам. Такой подход основывается на некоторой форме сообщения между потоками, благодаря которому один поток может уведомлять другой о том, что он заблокирован и что другой поток может возобновить свое выполнение. Сообщение между потоками организуется в C# с помощью методов Wait(), Pulse() и PulseAll().

Методы Wait(), Pulse() и PulseAll() определены в классе Monitor и могут вызываться только из заблокированного фрагмента блока. Они применяются следующим образом. Когда выполнение потока временно заблокировано, он вызывает метод Wait(). В итоге поток переходит в состояние ожидания, а блокировка с соответствующего объекта снимается, что дает возможность использовать этот объект в другом потоке. В дальнейшем ожидающий поток активизируется, когда другой поток войдет в аналогичное состояние блокировки, и вызывает метод Pulse() или PulseAll().

При вызове метода Pulse() возобновляется выполнение первого потока, ожидающего своей очереди на получение блокировки. А вызов метода PulseAll() сигнализирует о снятии блокировки всем ожидающим потокам.

Ниже приведены две наиболее часто используемые формы метода Wait():

```
public static bool Wait(object obj)
public static bool Wait(object obj, int миллисекунд_простоя)
```

В первой форме ожидание длится вплоть до уведомления об освобождении объекта, а во второй форме — как до уведомления об освобождении объекта, так и до истечения периода времени, на который указывает количество миллисекунд_простоя.

В обеих формах obj обозначает объект, освобождение которого ожидается. Ниже приведены общие формы методов Pulse() и PulseAll():

```
public static void Pulse(object obj)
public static void PulseAll(object obj)
```

где obj обозначает освобождаемый объект.

Если методы Wait(), Pulse() и PulseAll() вызываются из кода, находящегося за пределами синхронизированного кода, например из блока lock, то генерируется исключение SynchronizationLockException.

Предположим, что некий поток выполняет код при соблюдении сложного условия. Можно просто организовать зацикливание этого потока с периодической проверкой условия. Однако, во-первых, это пустая трата процессорного времени, во-вторых, невозможно атомарно проверить несколько переменных, входящих в условие. К счастью, существует шаблон программирования, позволяющий потокам эффективно синхронизировать свои операции на основе сложного условия. Он называется паттерном условной переменной (condition variable pattern), а для его применения можно воспользоваться следующими методами класса Monitor:

```
public static class Monitor {
	public static Boolean Wait(Object obj);
	public static Boolean Wait(Object obj, Int32 millisecondsTimeout);
	public static void Pulse(Object obj);
	public static void PulseAll(Object obj);
}
```

Вот как выглядит данный паттерн:

```
internal sealed class ConditionVariablePattern {
	private readonly Object m_lock = new Object();
	private Boolean m_condition = false;
	public void Thread1() {
		Monitor.Enter(m_lock); // Взаимоисключающая блокировка
		// "Атомарная" проверка сложного условия блокирования
		while (!m_condition) {
			// Если условие не соблюдается, ждем, что его поменяет другой
		поток
			Monitor.Wait(m_lock); // На время снимаем блокировку, чтобы
		//другой поток мог ее получить
		}
		// Условие соблюдено, обрабатываем данные...
		Monitor.Exit(m_lock); // Снятие блокировки
	}
	public void Thread2() {
		Monitor.Enter(m_lock); // Взаимоисключающая блокировка
		// Обрабатываем данные и изменяем условие...
		m_condition = true;
		// Monitor.Pulse(m_lock); // Будим одного ожидающего ПОСЛЕ отмены
	блокировки
		Monitor.PulseAll(m_lock); // Будим всех ожидающих ПОСЛЕ отмены
	блокировки
		Monitor.Exit(m_lock);	// Снятие блокировки
	}
}
```

В этом коде поток, выполняющий метод Thread1, входит в код взаимоисключающей блокировки и осуществляет проверку условия. В данном случае я всего лишь проверяю значение поля Boolean, но условие может быть сколь угодно сложным. К примеру, можно взять текущую дату и удостовериться, что сейчас вторник и март, а заодно проверить, что коллекция состоит из 10 элементов. Если условие не соблюдается, поток не зацикливается на проверке, так как это было бы напрасной тратой процессорного времени, а вызывает метод Wait. Данный метод снимает блокировку, чтобы ее мог получить другой поток, и приостанавливает вызывающий поток. Метод Thread2 содержит код, выполняемый вторым потоком. Он вызывает метод Enter для блокировки, обрабатывает какие-то данные, меняя при этом состояние условия, после чего вызывает метод Pulse(All), разблокирующий поток после вызова метода Wait. Метод Pulse разблокирует поток, ожидающий дольше всех (если такие имеются), в то время как метод

PulseAll разблокирует все ожидающие потоки (если такие есть). Однако ни один из этих потоков пока не просыпается. Поток, выполняющий метод Thread2, должен вызвать метод Monitor.Exit, давая шанс другому потоку выполнить блокировку. Кроме того, в результате выполнения метода PulseAll потоки разблокируются не одновременно. После освобождения потока, вызвавшего метод Wait, он становится владельцем блокировки, а так как это взаимоисключающее блокирование, в каждый момент времени им может владеть только один поток. Другие потоки имеют шанс получить право на блокировку только после того, как текущий владелец вызовет метод Wait или Exit.

Проснувшись, поток, выполняющий метод Thread1, снова проверяет условие в цикле. Если оно все еще не соблюдено, он опять вызывает метод Wait. В противном случае он обрабатывает данные и, в конце концов, вызывает метод Exit, снимая блокировку и давая доступ другим потокам возможность получить ее. Таким образом данный паттерн позволяет проверить несколько формирующих сложное условие переменных при помощи простой логики синхронизации (всего одной блокировки), а несколько ожидающих потоков могут разблокироваться без нарушения какой-либо логики, хотя при этом возможна напрасная трата процессорного времени.

17. Задачи (Task) в платформе .NET, асинхронные методы (async) и средства ожидания их выполнения (await) в языке C#. Параллельно выполняемые запросы к коллекциям объектов (Parallel). Параллельные (Concurrent-) коллекции.

В эпоху многоядерных машин, которые позволяют параллельно выполнять сразу несколько процессов, стандартных средств работы с потоками в .NET уже оказалось недостаточно. Поэтому одним из новшеств платформы .NET 4.0 стала библиотека параллельных задач TPL (Task Parallel Library), основной функционал которой располагается в пространстве имен System.Threading.Tasks. Данная библиотека позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах, если на целевом компьютере имеется несколько ядер. Кроме того, упрощается сама работа по созданию новых потоков. Поэтому начиная с .NET 4.0. рекомендуется использовать именно TPL и ее классы для создания многопоточных приложений, хотя стандартные средства и класс Thread по-прежнему находят широкое применение.

В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов .NET задача представлена специальным классом - классом Task, который находится в пространстве имен System.Threading.Tasks. Данный класс описывает отдельную задачу, которая запускается в отдельном потоке. Хотя при желании ее также можно запускать синхронно в текущем потоке.

Пример:

```
class Program
    {
        static void Main(string[] args)
        {
            Task task = new Task(Display);
            task.Start();
            Console.WriteLine("Выполняется работа метода Main");
            Console.ReadLine();
        }

        static void Display()
        {
            Console.WriteLine("Начало работы метода Display");
            Thread.Sleep(3000);
            Console.WriteLine("Завершение работы метода Display");
        }
    }
```

Класс Task в качестве параметра принимает делегат Action. Этот делегат имеет определение public delegate void Action(), то есть указывает на метод, который не имеет параметров и который не возвращает никакого значения (то есть имеет тип void). А именно таким и является метод Display, который мы передаем в конструктор класса Task. Далее чтобы запустить задачу,

вызываем метод Start: task.Start(), и после этого метод Display начнет выполняться во вторичном потоке.

Однако в данном случае у нас есть небольшая проблема: если основной код в методе Main уже отработал, а задача еще не завершилась, то мы можем нажать любую клавишу, и приложение завершит свою работу, вне зависимости, завершилась ли задача или нет. И чтобы указать, что метод Main должен подождать до конца выполнения задачи, нам надо использовать метод Wait

Также как и с потоками, мы можем создать и запустить массив задач:

```
class Program
{
    static void Main(string[] args)
    {
        Task[] tasks = new Task[5];
        for (int i = 0; i < tasks.Length;i++ )
        {
            tasks[i] = Task.Factory.StartNew(Display);
        }
        Task[] tasks2 = new Task[5];
        for (int i = 0; i < tasks2.Length; i++)
        {
            tasks2[i] = Task.Factory.StartNew(()=>{
                Console.WriteLine("Задача из лямбда-выражения");
            });
        }
        Console.WriteLine("Выполняется работа метода Main");
        Task.WaitAll(tasks);
        Task.WaitAll(tasks2);
        Console.ReadLine();
    }

    static void Display()
    {
        Console.WriteLine("Начало работы метода Display");
        Thread.Sleep(3000);
        Console.WriteLine("Завершение работы метода Display");
    }
}
```

здесь мы создаем два массива задач tasks и tasks2. Только теперь в качестве альтернативы мы используем для создания и запуска задач статический метод Task.Factory.StartNew(). В качестве параметра этот метод принимает делегат Action, а значит мы ему можем передать опять же метод Display (как для массива tasks), так и лямбда-выражение (как для массива tasks2). И возвращает этот метод создаваемую и запускаемую задачу.

Для ожидания окончания выполнения сразу всего массива задач используется метод Task.WaitAll()

Ключевые слова async и await, введённые в C# 5.0, значительно упрощают асинхронное программирование. Они также скрывают за собой некоторые сложности, которые, если вы потеряете бдительность, могут добавить проблем в ваш код. Описанные ниже практики пригодятся вам, если вы создаёте асинхронный код для .NET приложений.

Асинхронность необходима для действий, которые потенциально являются блокирующими, например, когда приложение получает доступ к вебу. Доступ к веб-ресурсам может быть медленным или осуществляться с задержками. Если такая активность заблокирована внутри синхронного процесса, все приложение должно ждать ответа. В асинхронном процессе, приложение может продолжить выполнять другую работу, которая не зависит от веб-ресурса, пока потенциально блокирующая задача завершится.

Поддержка асинхронности оказывается особенно важной для приложений с доступом к потоку пользовательского интерфейса, поскольку все действия, связанные с пользовательским интерфейсом, обычно исполняются в одном потоке. Если какой-либо процесс блокируется в синхронном приложении, все приложение блокируется. Приложение перестает отвечать, и можно предположить, что произошел сбой, в то время как оно просто ожидает.

При использовании асинхронных методов, приложение продолжает отвечать на действия с пользовательским интерфейсом. Например, можно изменить размер или свернуть окно, либо закрыть приложение, если вы не хотите дожидаться его завершения.

Подход, основанный на асинхронности, добавляет эквивалент автоматической передачи в список параметров из которых можно выбирать при разработке асинхронных операций. Таким образом, пользователь получает все преимущества стандартного асинхронного программирования, но с гораздо меньшими усилиями со стороны разработчика.

Ключевые **async** и **await** в C#, являются основой асинхронного программирования. С помощью этих двух ключевых слов возможно использование ресурсов платформы .NET Framework или среды выполнения Windows для создания асинхронного метода почти так же легко, как и для создания синхронного. Далее методы с асинхронным кодом, которые определяются с помощью ключевых слов async и await будут называться асинхронными методами.

```
async Task<int> AccessTheWebAsync()

{

      HttpClient client = new HttpClient();

      Task<string> getStringTask =
      client.GetStringAsync("http://msdn.microsoft.com");

      DoIndependentWork();

      string urlContents = await getStringTask;

      return urlContents.Length;

}
```

Следующие свойства резюмируют то, что делает в предыдущем примере метод async.

- Сигнатура метода включает модификатор **async**.
- Имя асинхронного метода, по соглашению, обычно заканчивается словом "Async".
- Возвращаемый тип один из следующих:
    - Task<TResult> , если метод имеет возвращаемое выражение, в котором операнд имеет тип TResult.
    - Task , если метод не имеет возвращаемого выражения или имеет инструкцию return без операнда.
    - Void, если необходимо написать асинхронный обработчик событий.

- Метод обычно содержит по крайней мере одно выражение await, которое отмечает место, в котором он не может продолжаться до тех пор, пока ожидаемая асинхронная операция не будет завершена. В то же время, метод приостанавливается, и управление передаётся объекту, вызвавшему его. В следующем подразделе этого раздела показано, что происходит в точке приостановки.

В асинхронных методах, следует использовать предоставленные ключевые слова и типы, чтобы указать, что требуется, и компилятор делает все остальное, включая отслеживание того, что должно произойти, когда управления возвращается в точку ожидания в приостановленном методе. Некоторые регулярные процессы, такие как: циклы и обработка исключений, трудно обрабатывать в традиционном асинхронном коде. В асинхронных методах вы пишите эти элементы как в синхронном подходе и проблема решена.

Объявив метод как асинхронный с помощью модификатора **async**, вы можете использовать следующие две возможности:

- Асинхронный метод сможет использовать ключевое слово await для обозначения точек приостановки. Оператор await сообщает компилятору, что асинхронный метод не может продолжить своё выполнение после этого места, пока ожидаемый асинхронный процесс не завершится. В то же время, управление возвращается в объект, вызвавший асинхронный метод.

  Приостановка асинхронного метода в выражении await не вызывает выход из метода и блоки **finally** не выполняются.
- Метод, отмеченный ключевым словом async, сам может являться ожидаемым методами, которые вызывают его.

Асинхронный метод обычно содержит один или несколько вхождений оператора await, но отсутствие выражения await не вызывает ошибку компилятора. Если асинхронный метод не использует оператор await, чтобы отметить точку приостановки, метод выполняется как обычный синхронный метод, несмотря на модификатор async. При компиляции таких методов выдается предупреждение.

async, **await** — являются контекстными ключевыми словами.

Параллелизм задач (task parallelism) – это подход наиболее низкого уровня для распараллеливания задач с помощью PFX. Классы этого уровня определены в пространстве имен System.Threading.Tasks.

По сути, задание представляет собой легковесный объект для управления распараллеливанием единицы работы (unit of work). Задание избегает накладных расходов по запуску выделенного потока путем использования пула потоков CLR: тот же самый пул потоков используется при вызове функции ThreadPool.QueueUserWorkItem, настроенный в CLR 4.0

специальным образом для более эффективной работы вместе с заданиями (и более эффективной работы в целом).

Задания могут применяться всегда, когда вам нужно выполнить что-либо параллельно. Однако они оптимизированы для повышения эффективности многоядерных процессоров: фактически класс Parallel и PLINQ построены на основе конструкций параллелизма задач.

Задания делают значительно больше, нежели предоставляют эффективный способ использования пула потоков.

PFX представляет базовую форму структурного параллелизма с помощью трех методов класса Parallel:

Parallel.Invoke: выполняет параллельно массив делегатов

Parallel.For: параллельный эквивалент цикла for

Parallel.ForEach: параллельный эквивалент цикла foreach

Все три метода блокируют управление до окончания выполнения всех действий. Как и при использовании PLINQ при возникновении необработанного исключения, оставшиеся рабочие потоки прекращают выполнение после завершения обработки текущего элемента и исключение (или исключения), завернутые в AggregationException, пробрасываются вызывающему коду.

Параллельные коллекции иногда бывают полезны при решении обычных задач многопоточности, когда вам нужна потокобезопасная коллекция. Однако есть несколько пояснений:

Параллельные коллекции оптимизированы для параллельного программирования. Стандартные коллекции их превосходят во всех случаях, кроме сценариев с высокой конкурентностью.

Потокбезопасные коллекции не гарантируют, что код, который их использует будет потокобезопасным.

Если в процессе перебора элементов параллельной коллекции другой поток ее модифицирует, исключение сгенерировано не будет. Вместо этого, вы получите смесь старого и нового содержимого.

Не существует параллельной версии List<T>.

Параллельные классы стека, очереди и набора (bag) внутри реализованы на основе связных списков. Это делает их менее эффективными в плане потребления памяти по сравнению с непараллельными версиями классов Stack и Queue, но более предпочтительными для параллельного доступа, поскольку связные списки являются отличными кандидатами для lock-free или low-lock реализаций. (Поскольку вставка узла в связный список требует модификации лишь пары ссылок, в то время как вставка элемента в структуру данных наподобие List<T> может потребовать перемещения тысяч существующих элементов.)

Другими словами, использование этих коллекций не эквивалентно использованию стандартных коллекций с операторами lock. Например, если мы выполним этот код в одном потоке:

var d = new ConcurrentDictionary<int,int>();

for (int i = 0; i < 1000000; i++) d[i] = 123;

он будет выполняться втрое медленнее, нежели этот код:

var d = new Dictionary<int,int>();

for (int i = 0; i < 1000000; i++) lock (d) d[i] = 123;

(Однако чтение ConcurrentDictionary выполняется быстрее, поскольку чтения являются lock-free.)

Параллельные коллекции еще отличаются от стандартных коллекций тем, что они содержат специальные методы для выполнения атомарных операций типа «проверить-и-выполнить» (test-and-act), такие как TryPop. Большинство этих методов унифицированы посредством интерфейса IProducerConsumerCollection<T>.

18. Области (домены) приложений (AppDomain). Создание и использование объектов в разных доменах приложений.

• Домены приложения являются ключевым аспектом независимой от ОС природы платформы .NET, поскольку такое логическое деление абстрагируется от того, как именно ОС представляет загруженный выполняемый объект.

• Домены приложения являются существенно менее ресурсоемкими в отношении времени процессора и памяти, чем весь процесс в целом. Поэтому среда CLR способна загружать и выгружать домены приложений намного быстрее, чем формальный процесс.

• Домены приложения обеспечивают лучший уровень изоляции для загруженного приложения. Если один домен приложения в рамках процесса "терпит неудачу", остальные домены приложений могут продолжать функционировать.

Из приведенного списка следует, что один процесс может содержать любое число доменов приложения, каждый из которых полностью изолирован от других доменов приложения в рамках данного процесса (а также любого другого процесса). С учетом этого следует понимать, что приложение, выполняющееся в одном домене приложения, не может получить данные (в частности, значения глобальных переменных или статических полей) другого домена приложений иначе, как с помощью протокола удаленного взаимодействия .NET).

Хотя один процесс и может принять множество доменов приложения, так бывает не всегда. Как минимум, процесс ОС будет содержать то, что обычно называют доменом приложения, созданным по умолчанию. Этот специальный домен приложения автоматически воздается средой CLR во время запуска процесса.

После этого CLR создает дополнительные домены приложения по мере необходимости, Если потребуется (хотя это и маловероятно), вы можете программно создавать домены приложения в среде выполнения в рамках выполняемого процесса, используя статические методы класса System.AppDomain. Этот класс оказывается также полезным для осуществления низкоуровневого контроля доменов приложения.

Члены класса:

- Свойство BaseDirectory: базовый каталог, который используется для получения сборок (как правило, каталог самого приложения)
- Свойство CurrentDomain: домен текущего приложения
- Свойство FriendlyName: имя домена приложения
- Свойство SetupInformation: представляет объект AppDomainSetup и хранит конфигурацию домена приложения
- Метод CreateDomain(): статический метод, позволяющий создать новый домен приложения в рамках текущего процесса
- Метод CreateInstance(): создает экземпляр типа из внешней сборки после ее загрузки в домен приложения
- Метод ExecuteAssembly(): запускает сборку exe в рамках текущего домена приложения
- Метод GetAssemblies(): получает набор сборок .NET, загруженных в домен приложения
- Метод Load(): загружает сборку домена приложения
- Метод Unload(): выгружает домен приложения из определенного процесса

События:

- AssemblyLoad Возникает при загрузке компоновочного блока

- AssemblyResolve Возникает, когда не удается идентифицировать компоновочный блок
- DomainUnload Возникает перед началом выгрузки домена приложения
- ProcessExit Возникает для домена приложения, созданного по умолчанию, когда завершается родительский процесс этого домена
- ResourceResolve Возникает, когда не удается идентифицировать ресурс
- TypeResolve Возникает, когда не удается идентифицировать тип
- UnhandledException Возникает, когда остается без обработки сгенерированное исключение

Список доменов приложения процесса

Для примера программного взаимодействия с доменами приложений .NET предположим, что у нас есть новое консольное приложение C# с именем AppDomainManipulator, в рамках которого определяется статический метод PrintAllAssembliesInAppDomain(). Этот вспомогательный метод использует AppDomain.GetAssemblies(), чтобы получить список всех двоичных файлов .NET, выполняющихся в рамках данного домена приложения.

Соответствующий список представляется массивом типов System.Reflection. Assembly, поэтому необходимо использовать пространство имен System. Получив массив компоновочных блоков, вы выполняете цикл по элементам массива и печатаете понятное имя и версию каждого модуля.

```
public static void PrintAllAssembliesInAppDomain(ApsDomain ad) {
 Assembly[] loadedAssemblies = ad.GetAssemblies();
 Console.WriteLine("*** Компоновочные блоки в рамках {0} ***\n",
ad.FriendlyName);
 foreach (Assembly a in loadedAssemblies) {
  Console.WriteLine("-› Имя: {0}", a.GetName().Name);
  Console.WriteLine("-› Версия: {0}\n", a.GetName().Version);
 }
}
```

Теперь обновим метод Main(), чтобы перед вызовом PrintAllAssembliesInAppDomain() получить ссылку на текущий домен приложения, используя свойство AppDomain.CurrentDomain.

Чтобы сделать пример более интересным, метод Main() открывает окно сообщения Windows Forms.

```
static void Main(string[] args) {
 Console.WriteLine("***** Чудесное приложение AppDomain *****\n");
 // Чтение информации для текущего AppDomain.
 AppDomain defaultAD= AppDomain.CurrentDomain;
 MessageBox.Show(''Привет");
 PrintAllAssembliesInAppDomain(defaultAD);
 Console.ReadLine();
}
```

Программное создание новых доменов приложения

Напомним, что один процесс может содержать множество доменов приложения, И хотя вам в программном коде вряд ли понадобится вручную создавать домены приложения, вы имеете возможность сделать это с помощью статического метода CreateDomain(). Нетрудно догадаться, что метод AppDomain.CreateDomain() перегружен. Вы, по крайней мере, должны указать понятное имя нового домена приложения, как показано ниже.

```
static void Main(string[] args) {
 // Создание нового AppDomain в рамках текущего процесса.
 AppDomain anotherAD = AppDomain.CreateDomain("SecondAppDomain");
 PrintAllAssembliesInAppDomain(anotherAD);
 Console.ReadLine();
}
```

Если выполнить приложение теперь вы увидите, что компоновочные блоки System.Windows.Forms.dll, System.Drawing.dll и System.dll будут загружены только в рамках домена приложения, созданного по умолчанию. Это может показаться нелогичным для тех. кто имеете опыт программирования с использованием традиционных подходов Win32 (скорее, оба домена приложения должны иметь доступ к одному и тому же множеству компоновочных блоков). Напомним, однако, о том. **что компоновочный блок загружается в рамки домена приложения а не в рамки непосредственно самого процесса.**

Далее, обратите внимание на то, что домен приложения SecondAppDomain автоматически получает свою собственную копию mscorlib.dll, поскольку этот ключевой компоновочный блок автоматически загружается средой CLR для каждого домена приложения. Это порождает следующий вопрос "Как можно программно загрузить компоновочный блок в домен приложения?" Ответ: с помощью метода AppDomain.Load() (или, альтернативно, с помощью AppDomain.executeAssembly()). В предположении, что вы скопировали CarLibrary.dll в каталог приложения AppDomainManipulator.exe, вы можете загрузить CarLibrary.dll в домен приложения SecondAppDomain так.

```
static void Main(string[] args) {
 Console.WriteLine("***** Чудесное приложение AppDomain *****\n");
 …
 // Загрузка CarLibrary.dll в новый AppDomain.
 AppDomain anotherAD = AppDomain.CreateDomain("SecondAppDomain");
 anotherAD.Load("CarLibrary");
 PrintAllAssembliesInAppDomain(anotherAD);
 Console.ReadLine();
}
```

Программная выгрузка доменов приложения

Важно понимать, что среда CLR не позволяет выгружать отдельные компоновочные блоки .NET. Однако, используя метод AppDomain.Unload(), вы можете избирательно выгрузить домен приложения из объемлющего процесса. При этом домен приложения выгрузит по очереди каждый компоновочный блок.

Напомним, что тип AppDomain определяет набор событий, одним из которых является DomainUnload. Это событие генерируется тогда, когда домен приложения (не являющийся доменом, созданным по умолчанию) выгружается из содержащего этот домен процесса.

Другим заслуживающим внимания событием является событие ProcessExit, которое генерируется при выгрузке из процесса домена, создаваемого по умолчанию (что, очевидно, влечет за собой завершение всего процесса). Так, если вы хотите программно выгрузить anotherAD из процесса AppDomainManipulator.exe и получить извещение о том, что соответствующий домен приложения закрыт, можете использовать следующую программную логику событий.

```
static void Main(string[] args) {
 …
 // Привязка к событию DomainUnload.
 anotherAD.DomainUnload += new EventHandler(anotherAD_DomainUnload);
 // Теперь выгрузка anotherAD.
AppDomain.Unload(anotherAD);
}
```

Обратите внимание на то, что событие DomainUnload работает в паре с делегатом System.EventHandler, поэтому формат anotherAD_DomainUnload() требует следующих аргументов.

```
public static void anotherAD_DomainUnload(object sender, EventArgs e) {
 Console.WriteLine("***** Выгрузка anotherAD! *****\n");
}
```

Если вы хотите получить извещение при выгрузке домена приложения, созданного по умолчанию, измените метод Main() так, чтобы обработать событие ProcessEvent, соответствующее домену приложения по умолчанию:

```
static void Main(string [] args) {
 …
   AppDomain defaultAD = AppDomain.CurrentDomain;
   defaultAD.ProcessExit +=new EventHandler(defaultAD_ProcessExit);
}
```

и определите подходящий обработчик событий.

```
   private static void defaultAD_ProcessExit (object sender, EventArgs
e) {
   Console.WriteLine("***** Выгрузка defaultAD! *****\n");
}
```

19. Средства обобщенного (generic) программирования в языках C++ и C#.

После появления первого выпуска платформы .NET программисты часто использовали пространство имен System.Collections для получения более гибкого способа управления данными в приложениях. Однако, начиная с версии .NET 2.0, язык программирования C# был расширен поддержкой средства, которое называется обобщением (generic). Вместе с ним библиотеки базовых классов пополнились совершенно новым пространством имен, связанным с коллекциями — System.Collections.Generic.

Термин обобщение, по существу, означает параметризированный тип. Особая роль параметризированных типов состоит в том, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра. С помощью обобщений можно, например, создать единый класс, который автоматически становится пригодным для обработки разнотипных данных. Класс, структура, интерфейс, метод или делегат, оперирующий параметризированным типом данных, называется обобщенным, как, например, обобщенный класс или обобщенный метод.

Следует особо подчеркнуть, что в C# всегда имелась возможность создавать обобщенный код, оперируя ссылками типа object. А поскольку класс object является базовым для всех остальных классов, то по ссылке типа object можно обращаться к объекту любого типа. Таким образом, до появления обобщений для оперирования разнотипными объектами в программах служил обобщенный код в котором для этой цели использовались ссылки типа object.

Но дело в том, что в таком коде трудно было соблюсти типовую безопасность, поскольку для преобразования типа object в конкретный тип данных требовалось приведение типов. А это служило потенциальным источником ошибок из-за того, что приведение типов могло быть неумышленно выполнено неверно. Это затруднение позволяют преодолеть обобщения, обеспечивая типовую безопасность, которой раньше так недоставало. Кроме того, обобщения упрощают весь процесс, поскольку исключают необходимость выполнять приведение типов для преобразования объекта или другого типа обрабатываемых данных. Таким образом, обобщения расширяют возможности повторного использования кода и позволяют делать это надежно и просто.

Обобщения — это не совсем новая конструкция; подобные концепции присутствуют и в других языках. Например, схожие с обобщениями черты имеют шаблоны C++. Особенность компиляции шаблонов является то, что компилируется не все классы сразу, а только классы к которым обращается шаблон. Значит можно допустить синтаксическую ошибку и компилятор ее обнаружит далеко не сразу. При использовании шаблонов C++ существует три больших недостатка:

1. шаблоны невозможно отлаживать.
2. существенно замедляется время компиляции. В больших проектах оно может доходить до 30-60 минут.
3. очень быстро растут размеры объектных модулей и библиотек на диске.

Однако между шаблонами C++ и обобщениями .NET есть большая разница. В C++ при создании экземпляра шаблона с конкретным типом необходим исходный код шаблонов. В отличие от шаблонов C++, обобщения являются не только конструкцией языка C#, но также определены для CLR. Это позволяет создавать экземпляры шаблонов с определенным типом-параметром на языке Visual Basic, даже если обобщенный класс определен на C#.

Если в программе используются обобщения, то очень полезно, когда переменные обобщенных типов легко можно отличить от необобщенных. Ниже представлены рекомендации по именованию обобщенных типов:

1.Имена обобщенных типов должны начинаться с буквы T.

2.Если обобщенный тип может быть заменен любым классом, поскольку нет никаких специальных требований, и используется только один обобщенный тип, T — вполне подходящее имя для обобщенного типа:

```
1  public class List<T> { }
2  public class LinkedList<T> { }
```

3. Если к обобщенному типу предъявляются специальные требования (например, что тип должен реализовывать интерфейс или наследоваться от определенного класса), либо же используется два или более обобщенных типа в качестве параметров, то следует применять осмысленные имена типов:

```
1  public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e) ;
2  public delegate TOutput Converter<TInput, TOutput>(TInput from);
3  public class SortedList<TKey, TValue> { }
```

Достоинства Generics

1.Защита исходного кода Разработчику, использующему обобщенный алго-ритм, не нужен доступ к исходному тексту алгоритма. А при работе с шабло-нами C++ или обобщениями Java разработчику нужен был исходный текст алгоритма.

2.Безопасность типов. Когда обобщенный алгоритм применяется с конкрет-ным типом, компилятор и CLR понимают это и обеспечивают, чтобы в алго-ритме использовались лишь объекты, совместимые с этим типом данных. По-пытка использования объекта, не совместимого с указанным типом, приведет к ошибке компиляции или ошибке во время выполнения.

3.Более простой и понятный код Поскольку компилятор обеспечивает бе-зопасность типов, в исходном тексте нужно меньше приведений типов, и та-кой код проще писать и поддерживать.

4.Повышение производительности До появления обобщений одним из спо-собов определения обобщенного алгоритма было определение всех его чле-нов так, чтобы они по определению «умели» работать с типом данных Object. Чтобы этот алгоритм работал с экземплярами значимого типа, перед вызовом членов алгоритма CLR должна была упаковать этот экземпляр. Приве-дения типов также не нужны поэтому CLR не нужно проверять безопасность типов при их преобразовании, что также ускоряет работу кода.

20. Итераторы в среде .NET. Создание и использование итераторов. Реализация итераторов с использованием и без использования оператора yield return языка C#.

Итератор — метод доступа get или оператор, выполняющий настраиваемую итерацию класса массива или коллекции с помощью ключевого слова yield. При использовании оператора return ключевого слова yield элемент в исходной последовательности немедленно возвращается вызывающему объекту до того, как будет получен доступ к следующему элементу. Хотя итератор создается как метод, компилятор переводит его во вложенный класс, который фактически является конечным автоматом. Данный класс отслеживает положения итератора, пока в клиентском коде выполняется цикл foreach.

Итератор вызывается из клиентского кода с помощью оператора foreach. Итераторы в .NET Framework называются 'перечислителями' (enumerators) и представлены интерфейсом IEnumerator. IEnumerator реализует метод MoveNext(), который переходит к следующему элементу и указывает достигнут ли конец коллекции; свойство Current служит для получения значения указываемого элемента; дополнительный метод Reset() возвращает перечислитель на его исходную позицию. Перечислитель первоначально указывает на специальное значение перед первым элементом, поэтому вызов MoveNext() необходим для начала итерации.

Перечислители обычно передаются вызовом метода GetEnumerator() объекта, реализующего интерфейс IEnumerable. Классы контейнеров обычно реализуют этот интерфейс. Тем не менее, выражение foreach в языке C# может оперировать любым объектом, поддерживающим подобный метод, даже если он не реализует IEnumerable.

```
// 'явная' версия
        IEnumerator<MyType> iter = list.GetEnumerator();
        while (iter.MoveNext())
                Console.WriteLine(iter.Current);
// 'неявная' версия
        foreach (MyType value in list)
                Console.WriteLine(value);
```

Например, можно создать итератор для класса, возвращающего элементы в обратном порядке или выполняющего операцию над каждым элементом перед тем, как итератор возвратит его. При создании итератора для класса или структуры реализация всего интерфейса IEnumerator не требуется. Когда компилятор обнаруживает итератор, он автоматически создает методы Current, MoveNext и Dispose интерфейса IEnumerator или IEnumerator<T>.

Общие сведения о итераторах

1. Итератор — это раздел кода, возвращающий упорядоченную последовательность значений одинакового типа.
2. Итератор может использоваться в качестве основной части метода, оператора или метода доступа get.
3. В коде итератора используется оператор yield return для возвращения каждого элемента по очереди. Ключевое слово yield break завершает итерацию.
4. В классе можно реализовать несколько итераторов. Каждый итератор должен иметь уникальное имя, как и любой член класса, и может вызываться клиентским кодом в операторе foreach: foreach(int x in SampleClass.Iterator2){}.

5. Тип возвращаемого итератором значения должен быть IEnumerable,IEnumerator,IEnumerable<T> или IEnumerator<T>.
6. Итераторы используются для обеспечения отложенного выполнения в запросах LINQ.

Ключевое слово yield используется для указания возвращаемого значения или значений. При подходе к оператору yield return текущее положение сохраняется. При следующем вызове итератора выполнение возобновляется с этого места.

Итераторы также удобно использовать с классами коллекций, так как это обеспечивает простой способ итерации сложных структур данных, например двоичных деревьев.

Реализации:

1.Yeld return

```
class MyClass {
        int limit = 0;
        public MyClass(int limit) { this.limit = limit; }
        public IEnumerable<int> CountFrom(int start)
         {
        for (int i =
        start; i <= limit; i++)
        yield return i;
}}
```

2. Без

```
class TestDaysOfTheWeek
        {
                static void Main()
                {
                        DaysOfTheWeek week = new DaysOfTheWeek();
                        foreach(string day in week)
                        {
                                System.Console.Write(day+ " ");
                        }
                }
        }
```

21. Атрибуты в среде .NET и языке C#. Создание своих атрибутов. Применение атрибутов для динамической загрузки и использования сборок при создании программ с расширяемой (plugin-) архитектурой.

В C# разрешается вводить в программу информацию декларативного характера в форме атрибута, с помощью которого определяются дополнительные сведения (метаданные), связанные с классом, структурой, методом и т.д. Например, в программе можно указать атрибут, определяющий тип кнопки, которую должен отображать конкретный класс. Атрибуты указываются в квадратных скобках перед тем элементом, к которому они применяются. Следовательно, атрибут не является членом класса, но обозначает дополнительную информацию, присоединяемую к элементу.

Одной из задач компилятора .NET является генерирование описаний метаданных для всех типов, которые были определены, и на которые имеется ссылка в текущем проекте. Помимо стандартных метаданных, которые помещаются в любую сборку, программисты в .NET могут включать в состав сборки дополнительные метаданные с использованием атрибутов. В сущности, атрибуты представляют собой не более чем просто аннотации, которые могут добавляться в код и применяться к какому-то конкретному типу (классу, интерфейсу, структуре и т.д.), члену (свойству, методу и т.д.), сборке или модулю.

В .NET атрибуты представляют собой типы классов, которые расширяют абстрактный базовый класс System.Attribute. В поставляемых в .NET пространствах имен доступно множество предопределенных атрибутов, которые полезно применять в своих приложениях. Более того, можно также создавать собственные атрибуты и тем самым дополнительно уточнять поведение своих типов, создавая для атрибута новый тип, унаследованный от Attribute.

Ниже перечислены некоторые из предопределенных атрибутов, предлагаемые в различных пространствах имен .NET:

[CLSCompliant]

Заставляет элемент, к которому применяется, отвечать требованиям CLS (Common Language Specification — общеязыковая спецификация). Вспомните, что типы, которые отвечают требованиям CLS, могут без проблем использоваться во всех языках программирования .NET

[DllImport]

Позволяет коду .NET отправлять вызовы в любую неуправляемую библиотеку кода на C или C++, в том числе и API-интерфейс лежащей в основе операционной системы. Обратите внимание, что при взаимодействии с программным обеспечением, работающим на базе COM, этот атрибут не применяется

[Obsolete]

Позволяет указать, что данный тип или член является устаревшим. Когда другие программисты попытаются использовать элемент с таким атрибутом, они получат соответствующее предупреждение от компилятора

[Serializable]

Позволяет указать, что класс или структура является "сериализируемой", т.е. способна сохранять свое текущее состояние в потоке

[NonSerialized]

Позволяет указать, что данное поле в классе или структуре не должно сохраняться в процессе сериализации

[WebMethod]

Позволяет указать, что метод может вызываться посредством HTTP-запросов, и CLR-среда должна преобразовывать его возвращаемое значение в формат XML

Важно уяснить, что при применении атрибутов в коде размещающиеся внутри них метаданные, по сути, остаются бесполезными до тех пор, пока не запрашиваются явным образом в каком-то другом компоненте программного обеспечения посредством рефлексии. Если этого не происходит, они спокойно игнорируются и не причиняют никакого вреда.

Несмотря на то, что .NET предоставляет нам большой набор встроенных атрибутов валидации, их может не хватать, нам могут потребоваться более изощренные в плане логики атрибуты. И в этом случае мы можем определить свои классы атрибутов.

При создании атрибута надо понимать, к чему именно он будет применяться - к свойству модели или ко всей модели в целом. Поэтому создадим два атрибута.

Для создания атриубта нам надо унаследовать свой класс от класса ValidationAttribute и реализовать его метод IsValid():

```
public class UserNameAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        if (value != null)
        {
            string userName = value.ToString();
            if (!userName.StartsWith("T"))
                return true;
            else
                this.ErrorMessage = "Имя не должно начинаться с буквы T";
        }
        return false;
    }
}
```

Данный атибут будет применяться к строковому свойству, поэтому в метод IsValid(object value) в качестве value будет передаваться строка. Поэтому в ходе программы мы можем преобразовать значение value к строке: value.ToString() или так string userName = value as string.

Далее в методе мы проверяем, начинается ли переданное значение с буквы "T". Если не начинается, то возвращаем true, что значит, что свойство, к которому будет применяться

данный атрибут, прошло валидацию. Если же строка начинается с буквы "T", то ваозвращаем false и устанавливаем сообщение об ошибке.

Подобным образом определим еще один атрибут, который будет применяться ко всей модели:

```
public class UserValidationAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        User user = value as User;
        if (user.Name=="Alice" && user.Age==33)
        {
            this.ErrorMessage = "Имя не должно быть Alice и возраст одновременно не должен быть равен 33";
            return false;
        }
        return true;
    }
}
```

Поскольку атрибут будет применяться ко всей модели, то в метод IsValid в качестве параметра value будет передаваться объект User. Как правило, атрибуты, которые применяются ко всей модели, валидируют сразу комбинацию свойств класса. В данном случае смотрим, чтобы имя и возраст одновременно не были равны "Alice" и 33.

Теперь применим эти атрибуты:

```
[UserValidation]
public class User
{
    [Required]
    public string Id { get; set; }
    [Required]
    [UserName]
    public string Name { get; set; }
    [Required]
    public int Age { get; set; }
}
```

Применение атрибутов. Используйте следующую процедуру для применения атрибутов к элементам собственного кода.

1. Определите новый атрибут или используйте существующий, импортировав его пространство имен из .NET Framework.

2. Примените этот атрибут к элементу кода, поместив его непосредственно перед элементом.
3. Каждый язык имеет свой собственный синтаксис атрибутов.В C++ и C# атрибут заключается в квадратные скобки и отделяется от элемента пробелами, которые могут содержать разрыв строки.В Visual Basic атрибут заключается в угловые скобки и должен находиться на той же логической строке. При необходимости вставки разрыва строки может использоваться символ продолжения строки.В J# атрибут присоединяется с помощью специального синтаксиса комментариев.
4. Укажите позиционные и именованные параметры атрибута.
5. Позиционные параметры являются обязательными и должны задаваться до всех именованных параметров. Они соответствуют параметрам одного из конструкторов атрибута.Именованные параметры являются необязательными и соответствуют свойствам атрибута.В C++, C# и J# укажите name=value для каждого необязательного параметра, где name — это имя свойства.В Visual Basic укажите name:=value.

При компиляции кода выполняется добавление атрибута в метаданные. Добавленный атрибут становится доступным среде CLR и любому пользовательскому инструменту или приложению через службы отражения среды выполнения.

По соглашению все имена атрибутов заканчиваются словом Attribute.Однако в некоторых языках, исполняемых в среде выполнения, например Visual Basic и C#, не требуется указание полного имени атрибута.Например, если требуется инициализироватьSystem.ObsoleteAttribute, то можно ссылаться на него как на Obsolete.

Применение атрибута к методу

В следующем примере кода показано объявление System.ObsoleteAttribute, помечающего код как устаревший.Атрибуту передается строка "Will be removed in next version".Этот атрибут вызывает предупреждение компилятора, в котором отображается переданная строка при вызове кода, описываемого данным атрибутом.

```
public class Example
{
    // Specify attributes between square brackets in C#.
    // This attribute is applied only to the Add method.
    [Obsolete("Will be removed in next version.")]
    public static int Add(int a, int b)
    {
        return (a + b);
    }
}

class Test
{
    public static void Main()
    {
        // This generates a compile-time warning.
        int i = Example.Add(2, 2);
    }
}
```

Применение атрибутов на уровне сборки

Если атрибут нужно применять на уровне сборки, используйте ключевое слово assembly (Assembly в Visual Basic).В следующем коде показан атрибут AssemblyTitleAttribute, применяемый на уровне сборки.

```
using System.Reflection;
[assembly:AssemblyTitle("My Assembly")]
```

При применении этого атрибута строка "My Assembly" помещается в манифест сборки в разделе метаданных файла.Атрибут можно просмотреть с помощью дизассемблера MSIL (Ildasm.exe) или путем создания пользовательской программы, извлекающей атрибут.

22. Взаимодействие управляемого и неуправляемого кода в среде .NET на примере вызова функций Windows API. Управление расположением полей в объектах на примере создания изменяемой строки (создать класс MutableString).

Управляемый код является кодом, динамически распределяемая память которого управляется автоматически (т.е. сборщиком мусора) общеязыковой средой выполнения CLR. Таким образом, программист может размещать объекты в управляемой динамически распределяемой области памяти, используя оператор new (создать), причем освобождать их с помощью соответствующих операторов delete (удалить) не нужно. Это освобождает программиста от заботы об утечках памяти, и позволяет сосредоточить основное внимание на важных и полезных задачах, таких как более точная реализация проекта программы, что повышает производительность программирования и качество программного обеспечения.
Неуправляемый код C++ .NET должен самостоятельно управлять динамически распределяемой областью в памяти традиционными способами C++, используя операторы new (создать) и delete (удалить). Так как одним из наиболее общих недостатков в программах на C++ является ужасающая утечка памяти, использование управляемых расширений VC++ .NET может оказать очень положительное воздействие на многие разработки программного обеспечения. Важно отметить, что оператор delete (удалить) может явно применяться к указателю на управляемый объект, если вы хотите самостоятельно управлять освобождением памяти, занятой объектом. Указанное обстоятельство окажется полезным в ситуациях, когда желательно выполнить деструктор объекта до того, как это сделает сам сборщик мусора, — это позволит избежать разделения . данных событий по времени в многопотоковых программах.
Причины смешивания управляемого и неуправляемого кодов
Если управляемые расширения C++ являются такими хорошими, тогда зачем может потребоваться создавать неуправляемый код? На этот вопрос существует несколько ответов:

1. Как и в других средах, где проводится автоматическая сборка мусора (таких как Smalltalk и Java), во время выполнения часто снижается производительность из-за накладных расходов на отслеживание использования объектов (отслеживание ссылок) и удаление их в нужное время.
2. Еще одним нежелательным эффектом, который часто ассоциируется с автоматической сборкой мусора, является повышение объема физической памяти, требуемой для хранения объектов, которые могут быть удалены, но еще не удалены сборщиком мусора. Более агрессивные схемы сборки мусора проигрывают в производительности, а менее агрессивные — в избыточном использовании памяти. В традиционной программе C++ программист сам решает, когда именно каждый объект удаляется из динамически распределяемой области памяти. Такой подход потенциально позволяет программисту написать программу, которая одновременно выиграет и в производительности, и в использовании памяти. К сожалению, для этого требуется большой опыт программирования и большие усилия.
3. У вас могут быть действующие приложения Win32, написанные на языке C++, которые вы хотите в течение некоторого периода времени преобразовать в приложения .NET. Тогда, по крайней мере в течение переходного периода, будет существовать программа, содержащая смесь управляемого и неуправляемого кода.

4. Вы можете обладать реальным опытом программирования в C++ и быть знакомым с традиционным программированием неуправляемого кода. Но если вам потребовалось разработать новые приложения для платформы .NET, то в этом случае вы можете захотеть написать программу, содержащую управляемые и неуправляемые части, в качестве простейшего подхода к миру программирования .NET, вместо того, чтобы нырять с головой в чистый C# или VB.NET.

Управляемые и неуправляемые ссылки и типы значений

Существуют фундаментальные отличия между тем, как управляемые и неуправляемые коды обрабатывают ссылки и типы значений. Неуправляемый код C++ позволяет объявлять локальные переменные, параметры методов и члены классов как относящиеся к типам, определенным неуправляемыми классами или структурами. Такие типы называются типами значений, так как подобные переменные содержат значения, которые в действительности являются данными. C++ также позволяет определять переменную как указатель или как ссылку на тип, определенный классом или структурой. Такие типы называются ссылочными типами, или типами ссылки, так как переменные реально не содержат значения, которые являются объектом, а вместо этого являются ссылками на объект соответствующего типа в неуправляемой динамически распределяемой области памяти. Это может немного удивить, потому что C++ пытается провести концептуальное разграничение между типами указателей и ссылочными типами. И все же в действительности ссылка в C++ является просто разновидностью постоянного (константного) указателя.

Объявление типа значения представляет собой в действительности выделение пространства памяти под реальные значения. Однако в управляемом коде нельзя объявить управляемый класс или структуру в качестве типа значения (если только не используется ключевое слово _value (значение)).

К сожалению, существует множество правил, которые ограничивают использование управляемых типов (классов, структур, интерфейсов), что ведет к затруднениям в работе с ними по сравнению с традиционными неуправляемыми типами в C++. Эти правила также оказывают некоторое влияние на то, как управляемый и неуправляемый коды могут взаимодействовать друг с другом.

1. Управляемый тип не может быть наследован из неуправляемого типа. С другой стороны, неуправляемый тип не может быть наследован из управляемого типа. Это значит, что структуры иерархий наследственности управляемых и неуправляемых классов всегда отделены друг от друга.

2. Управляемые типы не могут иметь друзей (т.е. дружественных функций, классов, структур и интерфейсов). Естественно, данное правило не распространяется на неуправляемые классы C++. Это, конечно, может быть и не так важно для многих программистов, ведь многие рассматривают дружественность как нарушение важной концепции объектно-ориентированного программирования, называемой инкапсуляцией. Неуправляемые типы могут иметь друзей, как и классы в традиционном C++; однако неуправляемые типы могут иметь друзей только из числа неуправляемых типов.

3. В отличие от неуправляемых типов, управляемые не поддерживают множественную наследуемость реализации. Однако как управляемые, так и неуправляемые типы поддерживают множественную наследуемость интерфейсов. Возможность наследовать только одну реализацию является еще одним ограничением, с которым могут смириться многие программисты. Несмотря на то, что традиционный C++ поддерживает множественную наследуемость реализаций,

большинство объектно-ориентированных языков (в том числе Java и Smalltalk) ее не поддерживают. Даже модель компонентных объектов Microsoft (COM), которая на двоичном уровне основывается на таблице виртуальных функций в стиле C++, не поддерживает эту возможность.

4. Управляемый тип может, очевидно, содержать член, который является указателем на управляемый объект. Управляемый тип также может содержать элемент данных, который является неуправляемым объектом или указателем на таковой. С другой стороны, неуправляемый тип не может включать в себя экземпляр управляемого типа или указатель на таковой. Все, что здесь сказано, касается не только указателей, но также и ссылок.

5. Неуправляемый класс, в котором не указан явно базовый класс, является независимым корневым классом. В то же время, управляемый класс, в котором не указан явно ни один класс в качестве базового, является производным от корневого класса System: :Object (Система::Объект).

6. К объекту, участвующему в сборке мусора (т.е. к экземпляру управляемого класса, который использует ключевое слово _дс (сборщик мусора), а не _value (значение) или _поде), можно получить доступ только посредством указателя (или ссылки) на объект в управляемой динамически распределяемой области памяти. Это является отличием от неуправляемых типов, которые могут содержаться либо непосредственно в переменной типа значения, либо к ним можно получить доступ посредством указателя на неуправляемую динамически распределяемую область памяти.

.

Когда строка конструируется классом String, выделяется ровно столько памяти, сколько необходимо для хранения данной строки. Однако, в пространстве имен System.Text имеется класс StringBuilder, который поступает лучше и обычно выделяет больше памяти, чем нужно в данный момент. У вас, как разработчика, есть возможность указать, сколько именно памяти должен выделить StringBuilder, но если вы этого не сделаете, то будет выбран объем по умолчанию, который зависит от размера начального текста, инициализирующего экземпляр StringBuilder. Класс StringBuilder имеет два главных свойства:

Length, показывающее длину строки, содержащуюся в объекте в данный момент

Capacity, указывающее максимальную длину строки, которая может поместиться в выделенную для объекта память

Любые модификации строки происходят внутри блока памяти, выделенного экземпляру StringBuilder. Это делает добавление подстрок и замену индивидуальных символов строки очень эффективными. Удаление или вставка подстрок неизбежно остаются менее эффективными, потому что при этих операциях приходится перемещать в памяти части строки. Выделять новую память и, возможно, полностью перемещать ее содержимое приходится только при выполнении ряда действий, которые приводят к превышению выделенной емкости строки. В дополнение к избыточной памяти, выделяемой изначально на основе экспериментов, StringBuilder имеет свойствоудваивать свою емкость, когда происходит переполнение, а новое значение емкости не установлено явно.

Методы класса StringBuilder

Метод	Назначение
Append()	**Добавляет строку к текущей строке**
AppendFormat()	**Добавляет строку, сформированную в соответствии со спецификатором формата**
Insert()	**Вставляет подстроку в строку**
Remove()	**Удаляет символ из текущей строки**
Replace()	**Заменяет все вхождения символа другим символом или вхождения подстроки другой подстрокой**
ToString()	**Возвращает текущую строку в виде объекта System.String (переопределение метода класса System.Object)**

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

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         StringBuilder hello = new StringBuilder("Привет, меня зовут Александр Ерохин",120);
         hello.AppendFormat(" Я рад вас приветствовать на моем сайте professorweb.ru");
         // Зашифруем строку, хранящуюся в переменной hello
         Console.WriteLine("Исходная строка: \n {0}\n",hello);

         for (int i = 'я'; i >= 'а'; i--)
            hello = hello.Replace((char)i,(char)(i+3));


         Console.WriteLine("Зашифрованная строка:\n {0}",hello);

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

23. Создание небезопасного кода в среде .NET. Использование адресной арифметики и Си-подобных указателей. Включение и выключение ошибок переполнения.

Небезопасный код - код, выделение и освобождение памяти для которого не контролируется исполняющей средой .NET. Небезопасный код дает особенно заметные преимущества при использовании указателей для взаимодействия с унаследованным кодом (таким как API для C) или когда вашему приложению требуется прямое манипулирование памятью (как правило, из соображений повышения производительности).

Вы можете писать небезопасный код, применяя ключевые слова unsafe и fixed. Первое указывает, что помеченный им блок будет работать в неуправляемом контексте. Его можно применять ко всем методам, включая конструкторы и свойства, и даже к блокам кода внутри методов. Второе отвечает за фиксацию (pinning) управляемых объектов. Фиксация налагает запрет на перемещение данного объекта сборщиком мусора (GC). По мере исполнения приложения выделенная под объекты память освобождается, в итоге остаются "фрагменты" свободной памяти. Чтобы не допустить фрагментации памяти, исполняющая среда .NET перемещает объекты, обеспечивая максимально эффективное использование памяти. Что хорошего в том, что при наличии указателя на конкретный адрес в памяти, .NET сдвинет находившийся по этому адресу объект без вашего ведома, оставив вас с неверным указателем! Поскольку GC перемещает объекты в памяти, чтобы увеличить эффективность работы приложения, будьте благоразумны, используя fixed.

Операторы C# могут выполняться в проверяемом или непроверяемом контексте. В проверяемом контексте арифметическое переполнение вызовет исключение. В непроверяемом контексте арифметическое переполнение будет проигнорировано, а результат усечен.

· checked - используется для явного включения проверки переполнения при выполнении арифметических операций и преобразований с данными целого типа. По умолчанию выражение, содержащее только константные значения, вызывает ошибку компилятора в том случае, если результат его вычисления выходит за допустимые пределы значений конечного типа. Если выражение содержит одно или несколько неконстантных значений, компилятор не выполняет проверку переполнения.

· unchecked - 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 переполнение не помечается. Вычисление в приведенном ниже примере выполняется в блоке или выражении unchecked, поэтому факт превышения результатом максимального значения целого числа игнорируется.

Если не указано ни checked, ни unchecked, контекст по умолчанию зависит от внешних факторов, например параметров компилятора.

24. Средства работы с данными в формате XML. Загрузка и сохранение данных в формате XML. Сериализация и десериализация программных объектов в формате XML.

XML — это язык разметки, позволяющий описывать данные.Он упрощает точное объявление контента и помогает получать более значимые результаты поиска на различных платформах.Кроме того, XML позволяет разделять представление от данных.Например, в HTML теги используются, чтобы сообщить браузеру, что данные должны быть выделены полужирным шрифтом или курсивом. В XML теги применяются только для описания данных, например названия города, температуры и атмосферного давления.В XML для представления данных в браузере используются таблицы стилей, например таблицы XSL и каскадные таблицы стилей (CSS).XML отделяет данные от представления и обработки.Это позволяет отображать и обрабатывать данные необходимым способом, применяя различные таблицы стилей и приложения.

XML — это подмножество языка SGML, оптимизированное для доставки через Интернет.Его спецификация определяется консорциумом World Wide Web (W3C).Такая стандартизация гарантирует, что структурированные данные будут согласованными и независимыми от приложений или поставщиков.

XML и связанные технологии играют важную роль в обработке данных в Windows Server 2008 Visual Studio.Visual Studio включает инструменты и функции, облегчающие работу с XML, XSLT и XML-схемами.

Редактор XML

Редактор XML используется для редактирования XML-документов.Он включает полную проверку синтаксиса XML 1.0, проверку правильности по схеме по мере ввода, цветовое кодирование и технологию IntelliSense.Если предоставлена схема или определение типа документа, технология IntelliSense выводит список разрешенных элементов и атрибутов.

Дополнительные возможности:

- поддержка XML-фрагментов, включая фрагменты, сформированные схемой;
- структурирование документов, позволяющее разворачивать и сворачивать элементы;
- возможность выполнять преобразования XSLT и просматривать результаты в виде текста, XML или HTML;
- возможность создавать схемы на языке XSD из экземпляров XML-документов;
- поддержка редактирования таблиц стилей XSLT, включая поддержку технологии IntelliSense;
- обозреватель XML-схем

Конструктор XML-схем

Конструктор XML-схем интегрирован с Microsoft Visual Studio 2010, а редактор XML позволяет работать со схемами языка определения схемы XML (XSD).

Отладка XSL

Visual Studio поддерживает отладку таблиц стилей XSLT.С помощью отладчика можно задавать точки останова в таблице стилей XSLT, переходить в таблицу стилей XSLT из кода и т. д.

XDocument

Класс XDocument содержит информацию, необходимую для корректного документа XML. Это включает в себя декларацию XML, инструкции по обработке и комментарии. Обратите внимание, что вам необходимо создание объектов XDocument, если вам требуется конкретный функциональные возможности, предоставляемые класса XDocument. Во многих случаях, вы можете работать непосредственно с XElement (простая модель программирования). XDocument происходит от XContainer. Таким образом, он может содержать дочерние узлы. Тем не менее, объекты XDocument может иметь только один дочерний узел XElement. Это отражает XML стандарт: может быть только один корневой элемент в XML документе.

XDocument может содержать следующие элементы:

- Один XDeclaration объект. XDeclaration позволяет указывать вам отдельные части XML декларации: XML версия, кодировку документа, и является ли XML документ автономным.
- Один XElement объект (корень XML documen).
- Любое количество XProcessingInstruction объектов. Инструкция обработки передает информацию в приложение, которое обрабатывает XML.
- Любое количество XComment объектов. Комментарии будут детьми в корневом элементе. Объект XComment не может быть первым аргументом в списке, потому что это не корректно для XML-документа, начинать с комментария.
- Один XDocumentType для DTD.

Следующий код создает объект XDocument и связывает с ней содержатся объекты.

```
XDocument d = new XDocument(
    new XComment("This is a comment."),
    new XProcessingInstruction("xml-stylesheet",
        "href='mystyle.css' title='Compact' type='text/css'"),
    new XElement("Pubs",
        new XElement("Book",
            new XElement("Title", "Artifacts of Roman Civilization"),
            new XElement("Author", "Moreno, Jordao")
        ),
        new XElement("Book",
            new XElement("Title", "Midieval Tools and Implements"),
            new XElement("Author", "Gazit, Inbar")
        )
    ),
    new XComment("This is another comment.")
);
d.Declaration = new XDeclaration("1.0", "utf-8", "true");
Console.WriteLine(d);
```

```
d.Save("test.xml");
```

Файл test.xml:

```
?xml version="1.0" encoding="utf-8"?>
<!--This is a comment.-->
<?xml-stylesheet href='mystyle.css' title='Compact' type='text/css'?>
<Pubs>
  <Book>
    <Title>Artifacts of Roman Civilization</Title>
    <Author>Moreno, Jordao</Author>
  </Book>
  <Book>
    <Title>Midieval Tools and Implements</Title>
    <Author>Gazit, Inbar</Author>
  </Book>
</Pubs>
<!--This is another comment.-->
```

XmlSerializer- класс

Сериализует и десериализует объекты в документы XML и из них. XmlSerializer позволяет контролировать способ кодирования объектов в XML.

Для мониторинга формирования XML можно применить специальные атрибуты к классам и членам. Например, чтобы указать другое имя xml-элемента, примените XmlElementAttribute к открытому полю или свойству, и установите свойство ElementName. Полный список подобных атрибутов см. в разделе Attributes That Control XML Serialization. Кроме того, можно реализовать интерфейс IXmlSerializable для наблюдения за выходными xml-данными.

Если созданный XML должен соответствовать части 5 всемирного документа консорциума (www.w3.org) "Протокол простого доступа к объекту (SOAP) 1.1 " необходимо построить XmlSerializer с XmlTypeMapping. Для контроля в кодировке XML SOAP, используйте атрибуты, перечисленные в разделах Атрибуты управления SOAP-сериализацией с кодировкой.

XmlSerializer обеспечивает преимущество работы с классами строгих типов и, в то же время, обеспечивает гибкость XML. С помощью поля или свойства типа XmlElement, XmlAttribute или XmlNode в строго типизированных классах можно считывать части xml-документа непосредственно в объекты XML.

При работе с расширяемыми схемами XML, можно также воспользоваться XmlAnyElementAttribute и XmlAnyAttributeAttribute атрибутами для сериализации и десериализации элементов или атрибутов, не найденых в исходной схеме. Для использования объектов, применитеXmlAnyElementAttribute на поле, возвращающее массив XmlElement объектов, или примените XmlAnyAttributeAttribute на поле, возвращающее массив XmlAttribute объектов.

Если свойство или поле возвращает сложный объект (например, массив или экземпляр класса) XmlSerializer преобразовывает его к элементу, вложенному в

основной документ xml. Например, первый класс в следующем коде возвращает экземпляр второго класса.

```
public class MyClass
{
    public MyObject MyObjectProperty;
}
public class MyObject
{
    public string ObjectName;
}
```

Сериализованный вывод XML выглядит следующим образом:

```
<MyClass>
  <MyObjectProperty>
  <ObjectName>My String</ObjectName>
  </MyObjectProperty>
</MyClass>
```

25. Средства работы с деревьями выражений (класс Expression и его производные). Преобразование лямбда-выражения на языке C# в дерево выражений. Программное создание деревьев выражений. Компиляция выражения в делегат, и выполнение делегата.

Деревья выражений не являются чем-то мистическим, хотя некоторые случаи их использования выглядят подобными магии. Как следует из названия, они представляют собой деревья объектов, в которых каждый узел сам является выражением. Разные типы выражений представляют различные операции, которые можно выполнять в коде: бинарные операции, такие как сложение; унарные операции наподобие определения длины массива; вызовы методов; обращения к конструкторам и т.д. Пространство имен System.Linq.Expressions содержит разнообразные классы, которые представляют выражения. Все они унаследованы от класса Expression, который является абстрактным и по большей части состоит из статических фабричных методов, предназначенных для создания экземпляров других классов выражений. Тем не менее, класс Expression предлагает два свойства. • Свойство Type представляет тип .NET вычисляемого выражения — его можно рассматривать как возвращаемый тип. Например, типом выражения, которое извлекает свойство Length строки, будет int. • Свойство NodeType возвращает вид выражения в форме члена перечисления ExpressionType со значениями вроде LessThan, Multiply и Invoke. Придерживаясь того же самого примера, в myString.Length часть, отвечающая за доступ к свойству, имела бы тип узла MemberAccess. Существует множество классов, производных от Expression, и некоторые из них могут иметь множество узлов разных типов. Например, BinaryExpression представляет любую операцию с двумя операндами: арифметическую, логическую, сравнения, индексации массива и т.д. Именно здесь становится важным свойство NodeType, т.к. оно позволяет отделять различные виды выражений, которые представлены одним и тем же классом.

Давайте начнем с создания одного из простейших деревьев выражений, суммирующего две целочисленных константы. Код в листинге создает дерево выражения для представления 2+3.

Простое дерево выражения, суммирующего числа 2 и 3

```
Expression firstArg = Expression.Constant(2);
Expression secondArg = Expression.Constant(3);
Expression add = Expression.Add(firstArg, secondArg);
Console.WriteLine(add);
```

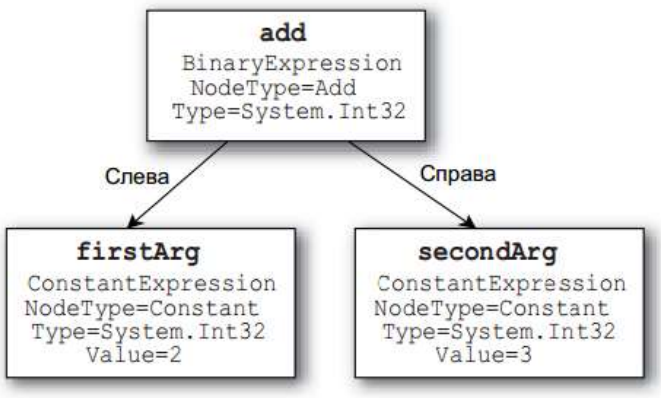
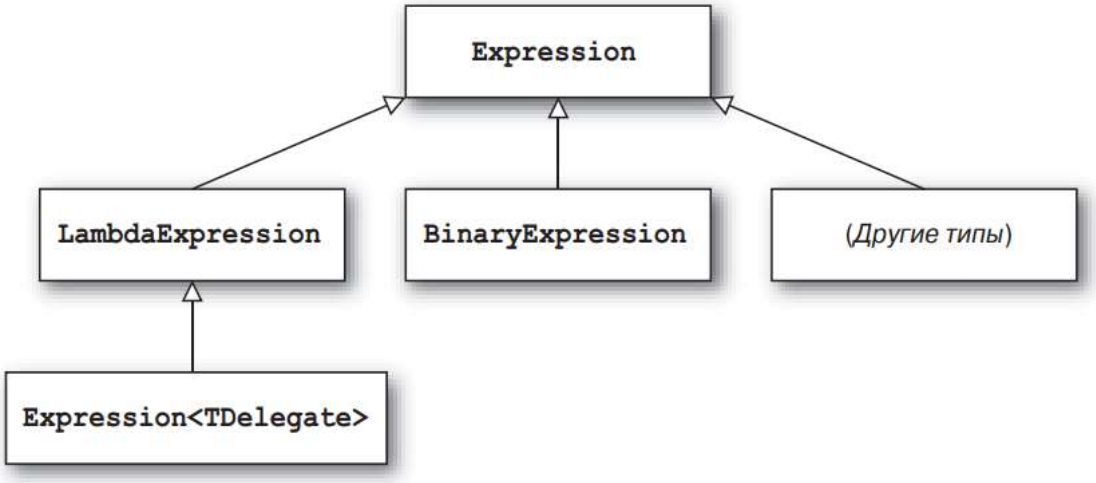


Рис. 9.2. Графическое представление дерева выражения

В число типов, производных от Expression, входит LambdaExpression. В свою очередь, от LambdaExpression унаследован обобщенный класс Expression. Все это немного запутывает, поэтому с целью прояснения на рис. показана иерархия типов.



Разница между Expression и Expression<TDelegate> в том, что обобщенный класс является статически типизированным для отражения вида представляемого выражения в терминах возвращаемого типа и параметров. Очевидно, это выражается параметром типа TDelegate, который должен быть типом делегата. Например, простое выражение сложения не принимает параметров и возвращает целочисленное значение, что соответствует сигнатуре Func, поэтому для представления такого выражения в статически типизированной манере можно было бы применить Expression<>. Это делается с использованием метода Expression.Lambda(), который имеет несколько перегруженных версий. В рассматриваемых примерах применяется обобщенный метод, который использует параметр типа для указания типа представляемого делегата. С альтернативами можно ознакомиться в MSDN. Итак, какой смысл делать все это? В классе LambdaExpression определен метод Compile(), который создает делегат подходящего типа; в Expression<> имеется другой метод с таким же именем, но он статически типизирован для возвращения делегата типа TDelegate. Затем этот делегат может быть выполнен обычным образом, как если бы он был создан с применением нормального метода или любыми другими средствами. Сказанное иллюстрируется в листинге 9.7 на примере того же самого выражения, что и ранее.

Компиляция и выполнение дерева выражения

```
Expression firstArg = Expression.Constant(2);
Expression secondArg = Expression.Constant(3);
Expression add = Expression.Add(firstArg, secondArg);
Func compiled = Expression.Lambda<Func<int>>(add).Compile();
Console.WriteLine(compiled());
```

Код в листинге является, вероятно, одним из самых запутанных путей вывода на консоль числа 5, какие только можно представить. В то же время он весьма выразителен. Здесь программно создаются логические блоки и представляются в виде обычных объектов, после чего у инфраструктуры запрашивается их компиляция в действительный код, который может быть выполнен. Возможно, вам никогда не придется использовать деревья выражений подобным образом или даже вообще строить их программно, но это дает полезную справочную информацию, которая поможет лучше понять функ- ционирование LINQ.

Преобразование лямбда-выражений C# в деревья выражений

Как уже было указано, лямбда-выражения могут быть преобразованы в соответствующие экземпляры делегатов, либо неявно, либо явно. Это не единственное

доступное преобразование. Компилятору можно также предложить построить дерево выражения из заданного лямбда-выражения, создавая экземпляр Expression во время выполнения. Например, в листинге показан намного более короткий путь создания выражения для “возврата числа 5”, его компиляции и обращения к результирующему делегату.

Листинг 9.8. Использование лямбда-выражений для создания деревьев выражений

```
Expression<Func<int>> return5 = () => 5;

Func compiled = return5.Compile();

Console.WriteLine(compiled());
```

Часть () => 5 в первой строке листинга 9.8 — это лямбда-выражение. Никакие приведения не требуются, поскольку компилятор может проверить все самостоятельно. Вместо 5 можно было бы написать 2+3, но компилятор применил бы к этому сложению оптимизацию, заменив его результатом суммы. Важный момент здесь в том, что лямбда-выражение было преобразовано в дерево выражения.

Давайте рассмотрим более сложный пример, чтобы увидеть, как все работает, особенно то, что касается параметров. На этот раз будет написан предикат, который принимает две строки и проверяет, находится ли первая строка в начале второй. Код оказывается простым, когда он представлен в виде лямбда-выражения (листинг 9.9).

Демонстрация более сложного дерева выражения

```
Expression<Func<string, string, bool>> expression = (x, y) => x.StartsWith(y);

var compiled = expression.Compile();

Console.WriteLine(compiled("First", "Second"));

Console.WriteLine(compiled("First", "Fir"));
```

Это дерево выражения сложнее само по себе, особенно к тому времени, как оно преобразуется в экземпляр LambdaExpression. В листинге показано, как его можно было бы построить в коде.

Построение выражения с вызовом метода в коде

```
MethodInfo method = typeof(string).GetMethod ("StartsWith", new[] { typeof(string) });

var target = Expression.Parameter(typeof(string), "x");

var methodArg = Expression.Parameter(typeof(string), "y");

Expression[] methodArgs = new[] { methodArg };

Expression call = Expression.Call(target, method, methodArgs);

var lambdaParameters = new[] { target, methodArg };

var lambda = Expression.Lambda (call, lambdaParameters);

var compiled = lambda.Compile();

Console.WriteLine(compiled("First", "Second"));

Console.WriteLine(compiled("First", "Fir"));
```

Как видите, объем кода в листинге значительно превышает версию с лямбда-выражением C#. Но при этом код делает более очевидным то, что в точности происходит в дереве, и показывает, как привязываются параметры.

Сначала определяется все, что необходимо знать о вызове метода, который формирует тело финального выражения n: цель метода (строка, на которой вызывается StartsWith()); сам метод (как MethodInfo); и список аргументов (в этом случае содержащий всего один элемент). Так получилось, что цель и аргумент нашего метода являются параметрами, передаваемыми выражению, однако они могли быть другими типами выражений — константами, результатами других вызовов методов, значениями свойств и т.д.

После построения вызова метода как выражения о нужно преобразовать его в лямбда-выражение p, по пути привязав параметры. В качестве информации для вызова метода повторно используются те же самые ранее созданные значения ParameterExpression: порядок, в котором они были указаны при создании лямбда-выражения — это порядок, в котором они будут выбираться, когда в конечном итоге вызывается делегат.

На рис. 9.4 окончательное дерево выражения представлено графически. По правде говоря, хотя это по-прежнему называется деревом выражения, факт повторного использования выражений параметров (и так должно делаться — создание нового выражения параметра с тем же самым именем и попытка привязки параметров подобным образом привела бы к генерации исключения во время выполнения) означает, что в строгом смысле оно действительно деревом не является

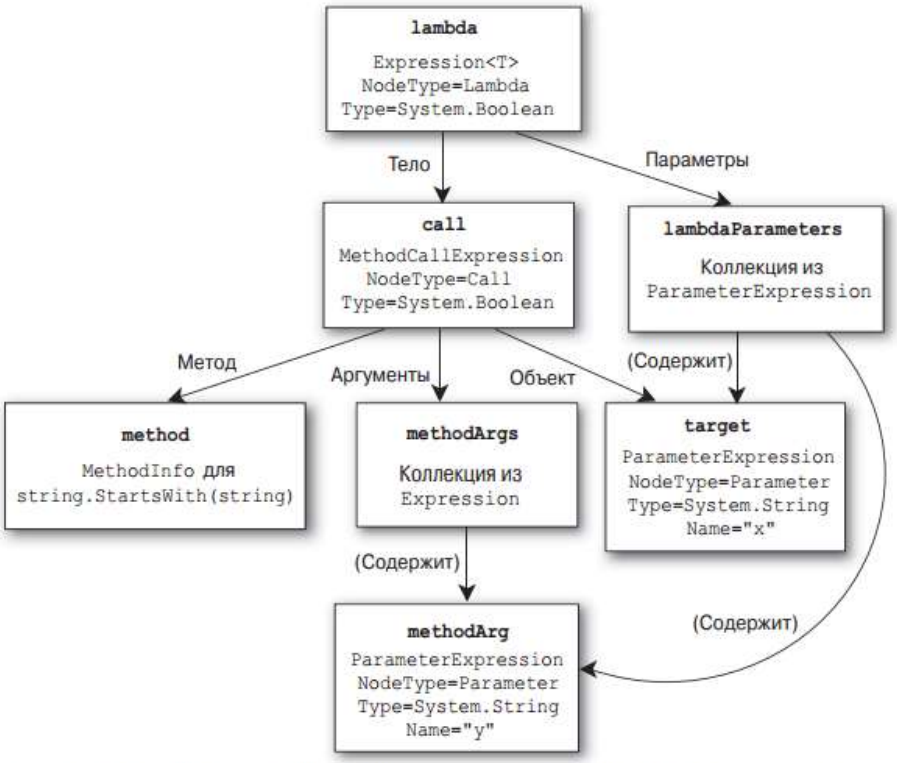


Рис. 9.4. Графическое представление дерева выражения, которое вызывает метод и использует параметры из лямбда-выражения

26. Средства работы с данными в бинарном виде. Сериализация и десериализация программных объектов, включая произвольные графы объектов, в бинарном виде.

Средства работы с данными в бинарном виде

Для работы с бинарными файлами предназначена пара классов BinaryWriter и BinaryReader. Эти классы позволяют читать и записывать данные в двоичном формате.

Основные метода класса BinaryWriter:

Close(): закрывает поток и освобождает ресурсы

Flush(): очищает буфер, дописывая из него оставшиеся данные в файл

Seek(): устанавливает позицию в потоке

Write(): записывает данные в поток

Основные метода класса BinaryReader:

Close(): закрывает поток и освобождает ресурсы

ReadBoolean(): считывает значение bool и перемещает указать на один байт

ReadByte(): считывает один байт и перемещает указать на один байт

ReadChar(): считывает значение char, то есть один символ, и перемещает указать на столько байтов, сколько занимает символ в текущей кодировке

ReadDecimal(): считывает значение decimal и перемещает указать на 16 байт

ReadDouble(): считывает значение double и перемещает указать на 8 байт

ReadInt16(): считывает значение short и перемещает указать на 2 байта

ReadInt32(): считывает значение int и перемещает указать на 4 байта

ReadInt64(): считывает значение long и перемещает указать на 8 байт

ReadSingle(): считывает значение float и перемещает указать на 4 байта

ReadString(): считывает значение string. Каждая строка предваряется значением длины строки, которое представляет 7-битное целое число

С чтением бинарных данных все просто: соответствующий метод считывает данные определенного типа и перемещает указатель на размер этого типа в байтах, например, значение типа int занимает 4 байта, поэтому BinaryReader считает 4 байта и переместит указать на эти 4 байта.

Посмотрим на реальной задаче применение этих классов. Попробуем с их помощью записывать и считывать из файла массив структур:

```
struct State
{
    public string name;
    public string capital;
    public int area;
    public double people;

    public State(string n, string c, int a, double p)
```

```
    {
        name = n;
        capital = c;
        people = p;
        area = a;
    }
}
```

Итак, у нас есть структура State с некоторым набором полей. В основной программе создаем массив структур и записываем с помощью BinaryWriter. Этот класс в качестве параметра в конструкторе принимает объект Stream, который, например, создается вызовом File.Open(path, FileMode.OpenOrCreate).

Затем в цикле пробегаемся по массиву структур и записываем каждое поле структуры в поток. В том порядке, в каком значения полей записываются, в том порядке они и будут размещаться в файле.

Затем считываем из записанного файла. Конструктор класса BinaryReader также в качестве параметра принимает объект потока, только в данном случае устанавливаем в качестве режима FileMode.Open: new BinaryReader(File.Open(path, FileMode.Open))

В цикле while считываем данные. Чтобы узнать окончание потока, вызываем метод PeekChar(). Этот метод считывает следующий символ и возвращает его числовое представление. Если символ отсутствует, то метод возвращает -1, что будет означать, что мы достигли конца файла.

В цикле последовательно считываем значения поле структур в том же порядке, в каком они записывались.

Таким образом, классы BinaryWriter и BinaryReader очень удобны для работы с бинарными файлами, особенно когда нам известна структура этих файлов. В то же время для хранения и считывания более комплексных объектов, например, объектов классов, лучше подходит другое решение - сериализация.

Сериализация и десериализация программных объектов в бинарном виде.

Когда объект сериализуется, среда CLR учитывает все связанные объекты, чтобы гарантировать корректное сохранение данных. Этот набор связанных объектов называется графом объектов. Графы объектов представляют простой способ документирования набора отношений между объектами.

Чтобы сделать объект доступным для служб сериализации .NET, понадобится только декорировать каждый связанный класс (или структуру) атрибутом [Serializable]. Если выясняется, что некоторый тип имеет члены-данные, которые не должны (или не могут) участвовать в схеме сериализации, можно пометить такие поля атрибутом [NonSerialized]. Это помогает сократить размер хранимых данных, при условии, что в сериализуемом классе есть переменные-члены, которые не следует "запоминать" (например, фиксированные значения, случайные значения, кратковременные данные и т.п.).

Для начала создадим новое консольное приложение. Добавим в него новый класс по имени Radio, помеченный атрибутом [Serializable], у которого исключается одна переменная-член (radioID), помеченная атрибутом [NonSerialized] и потому не сохраняемая в специфицированном потоке данных:

```
[Serializable]
        public class Radio
        {
        public bool hasTweeters;
        public bool hasSubWoofers;
        public double[] stationPresets;
        [NonSerialized]
        public string radioID = "XF-552RF6";
        }
```

Затем добавим два дополнительных типа, представляющих базовые классы JamesBondCar и Car (оба они также помечены атрибутом [Serializable]), и определим в них следующие поля данных:

```
[Serializable]
public class Car
{
  public Radio theRadio = new Radio();
  public bool isHatchBack;
}


[Serializable]
public class JamesBondClass : Car
{
  public bool canFly;
  public bool canSubmerge;
}
```

Тип BinaryFormatter сериализует состояние объекта в поток, используя компактный двоичный формат. Этот тип определен в пространстве имен System.Runtime.Serialization.Formatters.Binary, которое входит в сборку mscorlib.dll.

Когда используется тип BinaryFormatter, он сохраняет не только данные полей объектов из графа, но также полное квалифицированное имя каждого типа и полное имя определяющей его сборки (имя, версия, маркер общедоступного ключа и культура). Эти дополнительные элементы данных делают BinaryFormatter идеальным выбором, когда необходимо передавать объекты по значению (т.е. полные копии) между границами машин для использования в .NET-приложениях.

Serialize() сохраняет граф объектов в указанный поток в виде последовательности байтов;

Deserialize() преобразует сохраненную последовательность байт в граф объектов.

Предположим, что после создания экземпляра JamesBondCar и модификации некоторых данных состояния требуется сохранить этот экземпляр в файле *.dat. Первая задача — создание самого файла *.dat. Для этого можно создать экземпляр типа System.IO.FileStream. Затем следует создать экземпляр BinaryFormatter и передать ему FileStream и граф объектов для сохранения.

```
class Program
        {
        static void Main()
        {
        JamesBondClass jbc = new JamesBondClass();
        jbc.canFly = true;
        jbc.canSubmerge = false;
        jbc.theRadio.stationPresets = new double[] { 89.3, 105.1, 97.1 };
        jbc.theRadio.hasTweeters = true;
        // Сохранить объект в указанном файле в двоичном формате
        SaveBinaryFormat(jbc, "carData.dat");
        Console.ReadLine();

        }

        static void SaveBinaryFormat(object objGraph, string fileName)
        {
        BinaryFormatter binFormat = new BinaryFormatter();
        using (Stream fStream = new FileStream(fileName, FileMode.Create, FileAccess.Write,
FileShare.None))
        {
        binFormat.Serialize(fStream, objGraph);
        }
        Console.WriteLine("--> Сохранение объекта в Binary format");
        }
        }
```

Теперь предположим, что необходимо прочитать сохраненный объект JamesBondCar из двоичного файла обратно в объектную переменную. После открытия файла carData.dat (методом File.OpenRead()) просто вызовите метод Deserialize() класса BinaryFormatter.

```
static void LoadFromBinaryFile(string fileName)
{
        BinaryFormatter binFormat = new BinaryFormatter();


        using (Stream fStream = File.OpenRead(fileName))
        {
          JamesBondClass carFromDisk =
          (JamesBondClass)binFormat.Deserialize(fStream);
        Console.WriteLine(carFromDisk.canFly);
        }
}
```

Обратите внимание, что при вызове Deserialize() ему передается тип-наследник Stream, представляющий местоположение сохраненного графа объектов. Приведя возвращенный объект к правильному типу, вы получаете объект в том состоянии, в каком он был на момент сохранения.

27. Динамические объекты, тип данных dynamic и его использование в программе. Реализация своих классов динамических объектов.

Языки программирования по критерию проверки типов обычно делятся на статически типизированные (переменная связывается с типом в момент объявления и тип не может быть изменен позже) и динамически типизированные (переменная связывается с типом в момент присваивания значения и тип не может быть изменен позже). Однако обратите внимание, что динамическая проверка типов вовсе не означает полной свободы в смешивании типов. Например, даже при такой проверке типов вы все равно не сможете добавить булево значение к целому. Вся разница в том, что проверка осуществляется при выполнении программы, а не при ее компиляции.

C# является примером языка со статической типизацией, в то время как Python и PHP. — динамические. Оба языка динамические, позволяют использовать переменные и разрешают исполняющей среде вычислять их реальные типы. Но в PHP есть возможность хранения, скажем, целых значений и строк в одной переменной в той же области видимости

В C# 4.0 есть средства, которые являются одновременно динамическими и статическими, а также слабо и строго типизированными. Хотя C# зарождался как статически типизированный язык, он стал динамически типизированным в любом контексте, где вы можете использовать ключевое слово dynamic, например:

```
dynamic number = 10;
Console.WriteLine(number);
```

Зачем может понадобиться динамический объект? Во-первых, вы можете не знать тип объекта, с которым вам приходится иметь дело. Во-вторых, вашему объекту может быть присуща изменчивая природа. Вы можете работать с объектами, созданными в средах динамического программирования, таких как IronPython и IronRuby.

Важно понимать, что dynamic в системе типов C# является типом. У него очень специфический смысл, но это именно тип, и поэтому его нужно обрабатывать как таковой. Вы можете указать dynamic в качестве типа объявляемой вами переменной, типа элементов в наборе или возвращаемого значения метода. Кроме того, dynamic можно применять как тип параметра метода. Но использовать dynamic с оператором typeof или в качестве базового типа класса нельзя.

```
public void Execute()  {
dynamic calc = GetCalculator();
int result = calc.Sum(1, 1);
}
```

Если вы достаточно знаете о типе объекта, возвращаемого методом GetCalculator, то можете объявить переменную calc соответствующего типа или как var, Но если такого метода нет, вы получите ошибку компиляции.С помощью dynamic вы откладываете любое решение о корректности выражения до периода выполнения, компилятор просто полагает, что объект в динамической переменной поддерживает любые операции. То есть вы можете написать код, который вызывает метод объекта, который, как вы ожидаете, появится в ней в период выполнения.

Гибкость ключевого слова dynamic показана в программе

```
class Program {
  static void Main(string[] args) {
```

```
        dynamic x = DoubleIt(2);
        Console.WriteLine(x);
        Console.WriteLine("Press any key");
        Console.ReadLine();
    }
        private static dynamic DoubleIt(dynamic p) {
        return p + p;
    }
}
```

В основной программе у меня есть динамическая переменная, экземпляр которой создается на основе значения, возвращаемого из вызванной функции. Если вы передадите в функцию число 2, получите значение 4. А если вы передадите 2 как строку, получите 22.

Var указывает тип переменной, который должен быть присвоен типу инициализатора периода компиляции.

Но dynamic подразумевает, что тип переменной является динамическим, поддерживаемым в C# 4.0. В конечном счете dynamic и var имеют совершенно разный смысл. Var улучшает статическую типизацию. Это ключевое слово помогает компилятору корректно распознать тип переменной по точному типу, возвращаемому инициализатором.

Ключевое слово dynamic обеспечивает полный уход от статической типизации. При использовании в объявлении переменной оно указывает компилятору вообще не обрабатывать тип переменной. Тип должен быть таким, каким он окажется в период выполнения. В случае var ваш код статически типизируется — так же, как если бы вы выбрали классический подход с применением явных типов в объявлении переменной.

Другое различие между этими двумя ключевыми словами состоит в том, что var может появляться только в объявлении локальной переменной. Использовать var для определения свойства класса, возвращаемого значения или параметра функции нельзя.

Как разработчик вы можете применять dynamic с переменными, которые могут содержать объекты неопределенного типа, например объекты, возвращаемые COM или DOM API, получаемые от динамических языков (скажем, IronRuby), через механизм отражения или от объектов, динамически встраиваемых в C# 4.0 с использованием новых средств расширения.

Но динамический тип не обходит проверки типов. Он лишь отодвигает их на период выполнения. Если в период выполнения будут обнаружены несовместимости по типам, вы получите исключения.

28. **Библиотека Windows Forms и основные принципы ее использования для создания программ с пользовательским интерфейсом. Синхронизация с главным потоком при доступе к визуальным компонентам из второстепенных потоков.**

Windows Forms позволяет разрабатывать интеллектуальные клиенты. *Интеллектуальный клиент* — это приложение с полнофункциональным графическим интерфейсом, простое в развертывании и обновлении, способное работать при наличии или отсутствии подключения к Интернету и использующее более безопасный доступ к ресурсам на локальном компьютере по сравнению с традиционными приложениями Windows.

Windows Forms — это технология интеллектуальных клиентов для .NET Framework. Она представляет собой набор управляемых библиотек, упрощающих выполнение стандартных задач, таких как чтение из файловой системы и запись в нее. С помощью среды разработки типа Visual Studio можно создавать интеллектуальные клиентские приложения Windows Forms, которые отображают информацию, запрашивают ввод от пользователей и обмениваются данными с удаленными компьютерами по сети.

В Windows Forms *форма* — это видимая поверхность, на которой выводится информация для пользователя. Обычно приложение Windows Forms строится путем помещения элементов управления на форму и написания кода для реагирования на действия пользователя, такие как щелчки мыши или нажатия клавиш. *Элемент управления* — это отдельный элемент пользовательского интерфейса, предназначенный для отображения или ввода данных.

При выполнении пользователем какого-либо действия с формой или одним из ее элементов управления создается событие. Приложение реагирует на эти события с помощью кода и обрабатывает события при их возникновении.

Windows Forms включает широкий набор элементов управления, которые можно добавлять на формы: текстовые поля, кнопки, раскрывающиеся списки, переключатели и даже веб-страницы. Если существующий элемент управления не удовлетворяет потребностям, в Windows Forms можно создать пользовательские элементы управления с помощью класса UserControl.

Если нужно создать свои собственные элементы пользовательского интерфейса, пространство имен System.Drawing содержит широкий набор классов, необходимых для отрисовки линий, кругов и других фигур непосредственно на форме.

Для синхронизации с главным потоком при доступе к визуальным компонентам из второстепенных потоков используется компонент System.ComponentModel.BackgroundWorker. Класс BackgroundWorker появился в .NET 2.0 и был предназначен для упрощения работы с потоками в приложениях Windows Forms. Компонент BackgroundWorker предоставляет почти идеальный способ запуска длительно выполняющихся задач в отдельном потоке. Он использует диспетчер "за кулисами" и абстрагирует сложности маршализации с помощью модели событий.

BackgroundWorker также поддерживает два дополнительных удобства: события продвижения и сообщения отмены. В обоих случаях детали многопоточности скрыты, что облегчает кодирование.

BackgroundWorker незаменим, если есть единственная асинхронная задача, которая выполняется в фоновом режиме от начала до конца (с необязательной поддержкой уведомлений о продвижении и возможностью отмены). Если же имеется в виду что-то еще, например, асинхронная задача, которая работает на протяжении всей жизни приложения, или асинхронная задача, взаимодействующая с приложением, пока оно выполняет свою работу, то придется спроектировать специальное решение, воспользовавшись поддержкой многопоточности .NET.

Чтобы использовать BackgroundWorker, следует начать с создания его экземпляра. Для этого необходимо создать BackgroundWorker в коде и присоединить программно все обработчики событий.

Когда BackgroundWorker начинает работу, он захватывает свободный поток из пула потоков CLR и затем инициирует событие DoWork из этого потока. В обработчике события DoWork запускается длительно выполняющаяся задача. Как только работа будет завершена, BackgroundWorker инициирует событие RunWorkerCompleted, чтобы уведомить приложение. Это событие инициируется в потоке диспетчера, что позволит обратиться к разделяемым данным и пользовательскому интерфейсу, не порождая никаких проблем.

Отслеживание продвижения

BackgroundWorker также предоставляет встроенную поддержку первоначальной установки свойства BackgroundWorker.WorkerReportsProgress в true. На самом деле предоставление и отображение информации о продвижении — двухшаговый процесс.

Первым делом коду обработки события DoWork необходимо вызвать метод *BackgroundWorker.ReportProgress()* и показать предполагаемый процент готовности (от 0% до 100%). Это можно делать редко или часто — как нравится. При каждом вызове ReportProgress() объект BackgroundWorker инициирует событие ProgressChanged. На это событие можно отреагировать, чтобы прочитать процент готовности и обновить пользовательский интерфейс. Поскольку событие ProgressChanged инициировано в потоке пользовательского интерфейса, в применении Dispatcher.BeginInvoke() нет необходимости.

Поддержка отмены

С помощью BackgroundWorker столь же просто добавить поддержку отмены длительно выполняющейся задачи. Первый шаг состоит в установке в true свойства *BackgroundWorker.WorkerSupportsCancellation*.

Чтобы запросить отмену, код должен вызвать метод BackgroundWorker.CancelAsync(). При вызове CancelAsync() ничего автоматически не происходит. Вместо этого код, выполняющий задачу, должен явно проверить запрос на отмену, выполнить необходимую очистку и вернуть управление.

29. **Библиотека Windows Presentation Foundation (WPF) и основные принципы ее использования для создания программ с пользовательским интерфейсом. Синхронизация с главным потоком при доступе к визуальным компонентам из второстепенных потоков.**

Технология WPF является часть экосистемы платформы .NET и представляет собой подсистему для построения графических интерфейсов.

Если при создании традиционных приложений на основе WinForms за отрисовку элементов управления и графики отвечали такие части ОС Windows, как User32 и GDI+, то приложения WPF основаны на **DirectX**. В этом состоит ключевая особенность рендеринга графики в WPF: используя WPF, значительная часть работы по отрисовке графики, как простейших кнопочек, так и сложных 3D-моделей, ложиться на графический процессор на видеокарте, что также позволяет воспользоваться аппаратным ускорением графики.

В основе WPF лежит векторная система визуализации, не зависящая от разрешения устройства вывода и созданная с учётом возможностей современного графического оборудования. WPF предоставляет средства для создания визуального интерфейса, включая язык XAML (Extensible Application Markup Language), элементы управления, привязку данных, макеты, двухмерную и трёхмерную графику, анимацию, стили, шаблоны, документы, текст, мультимедиа и оформление.

XAML представляет собой язык декларативного описания интерфейса, основанный на XML. Также реализована модель разделения кода и дизайна, позволяющая кооперироваться программисту и дизайнеру. Кроме того, есть встроенная поддержка стилей элементов, а сами элементы легко разделить на элементы управления второго уровня, которые, в свою очередь, разделяются до уровня векторных фигур и свойств/действий.

WPF предоставляет следующие возможности:

- Использование традиционных языков .NET-платформы - C# и VB.NET для создания логики приложения
- Возможность декларативного определения графического интерфейса - некоторая схожесть с языками разметки html/xml, представляя альтернативу программному созданию графики и элементов управления, а также возможность комбинировать XAML и C#/VB.NET
- **Независимость от разрешения экрана**: поскольку в WPF все элементы измеряются в независимых от устройства единицах, приложения на WPF легко масштабируются под разные экраны с разным разрешением.
- Новые возможности, которых сложно было достичь в WinForms, например, создание трехмерных моделей, привязка данных, создание браузерных приложений, использование таких элементов, как стили, шаблоны, темы и др.
- Хорошее **взаимодействие с WinForms**, благодаря чему, например, в приложениях WPF можно использовать традиционные элементы управления из WinForms.
- Богатые возможности по созданию различных приложений: это и мультимедиа, и двухмерная и трехмерная графика, и богатый набор встроенных элементов управления, а также возможность самим создавать новые элементы, создание анимаций, привязка данных, стили, шаблоны, темы и многое другое
- Создание приложений под множество ОС семейства Windows - от Windows XP до Windows 8

В тоже время WPF имеет определенные ограничения. Несмотря на поддержку трехмерной визуализации, для создания приложений с большим количеством трехмерных изображений, прежде всего игр, лучше использовать другие средства - DirectX или XNA. WPF может не поддерживаться старыми видеокартами. Кроме того, на целевом компьютере должна быть установлена среда .NET не ниже 3.0, а поскольку платформа с каждой версией сильно обновляется (значительнее чем WinForms), то, чтобы максимально использовать возможности

платформы, рекомендуется новейшая версия .NET. Также необходима, как минимум, Windows XP SP 2. Соответственно для разработки приложений нужна среда Visual Studio 2008 и выше или Expression Blend.

Для синхронизации с главным потоком при доступе к визуальным компонентам из второстепенных потоков используется компонент System.ComponentModel.BackgroundWorker. Класс BackgroundWorker появился в .NET 2.0 и был предназначен для упрощения работы с потоками в приложениях Windows Forms. Однако BackgroundWorker в той же мере применим и в WPF. Компонент BackgroundWorker предоставляет почти идеальный способ запуска длительно выполняющихся задач в отдельном потоке. Он использует диспетчер "за кулисами" и абстрагирует сложности маршализации с помощью модели событий.

BackgroundWorker также поддерживает два дополнительных удобства: события продвижения и сообщения отмены. В обоих случаях детали многопоточности скрыты, что облегчает кодирование.

BackgroundWorker незаменим, если есть единственная асинхронная задача, которая выполняется в фоновом режиме от начала до конца (с необязательной поддержкой уведомлений о продвижении и возможностью отмены). Если же имеется в виду что-то еще, например, асинхронная задача, которая работает на протяжении всей жизни приложения, или асинхронная задача, взаимодействующая с приложением, пока оно выполняет свою работу, то придется спроектировать специальное решение, воспользовавшись поддержкой многопоточности .NET.

Чтобы использовать BackgroundWorker, следует начать с создания его экземпляра. При этом на выбор доступны два подхода:

- Можно создать BackgroundWorker в коде и присоединить программно все обработчики событий.
- Можно объявить BackgroundWorker в XAML-разметке. Преимущество такого подхода в возможности присоединения обработчиков событий через атрибуты. Поскольку BackgroundWorker не является видимым элементом WPF, его нельзя поместить куда угодно. Вместо этого его понадобится объявить как ресурс для окна.

Когда BackgroundWorker начинает работу, он захватывает свободный поток из пула потоков CLR и затем инициирует событие DoWork из этого потока. В обработчике события DoWork запускается длительно выполняющаяся задача. Как только работа будет завершена, BackgroundWorker инициирует событие RunWorkerCompleted, чтобы уведомить приложение. Это событие инициируется в потоке диспетчера, что позволит обратиться к разделяемым данным и пользовательскому интерфейсу, не порождая никаких проблем.

Отслеживание продвижения

BackgroundWorker также предоставляет встроенную поддержку первоначальной установки свойства BackgroundWorker.WorkerReportsProgress в true. На самом деле предоставление и отображение информации о продвижении — двухшаговый процесс.

Первым делом коду обработки события DoWork необходимо вызвать метод *BackgroundWorker.ReportProgress()* и показать предполагаемый процент готовности (от 0% до 100%). Это можно делать редко или часто — как нравится. При каждом вызове ReportProgress() объект BackgroundWorker инициирует событие ProgressChanged. На это событие можно отреагировать, чтобы прочитать процент готовности и обновить пользовательский интерфейс. Поскольку событие ProgressChanged инициировано в потоке пользовательского интерфейса, в применении Dispatcher.BeginInvoke() нет необходимости.

Поддержка отмены

С помощью BackgroundWorker столь же просто добавить поддержку отмены длительно выполняющейся задачи. Первый шаг состоит в установке в true свойства *BackgroundWorker.WorkerSupportsCancellation*.

Чтобы запросить отмену, код должен вызвать метод BackgroundWorker.CancelAsync(). При вызове CancelAsync() ничего автоматически не происходит. Вместо этого код, выполняющий задачу, должен явно проверить запрос на отмену, выполнить необходимую очистку и вернуть управление.

30. Библиотека Windows Communication Foundation (WCF) и основные принципы ее использования для коммуникации клиентов и серверов. Пример создания клиента и сервера.

Windows Communication Foundation (WCF) — это среда выполнения и набор API-интерфейсов для создания систем, которые обеспечивают обмен сообщениями между службами и клиентами. Те же инфраструктура и интерфейсы API используются для создания приложений, обменивающихся данными с другими приложениями на данном компьютере или на компьютере, который находится в другой компании, и доступ к которому можно получить через Интернет.

В основе WCF лежит принцип связи с помощью обмена сообщениями, и любые объекты, моделируемые в виде сообщений (например, HTTP-запрос или сообщение очереди сообщений, MSMQ), можно представить единым образом в модели программирования. Это обеспечивает универсальный интерфейс API для разных транспортных механизмов.

В модели различаются клиенты, являющиеся приложениями, которые инициируют связь, и службы, являющиеся приложениями, которые ожидают связи клиентов с ними и отвечают им. Одно приложение может быть, как клиентом, так и службой.

Между конечными точками выполняется обмен сообщениями. Конечные точки — это места, где отправляются или принимаются сообщения и где определяются все сведения, необходимые для обмена сообщениями. Служба предоставляет одну или несколько конечных точек приложения (а также ноль или более конечных точек инфраструктуры), а клиент создает конечную точку, совместимую с одной из конечных точек службы.

Конечная точка стандартным способом описывает место отправки сообщений, способ отправки сообщений и вид сообщений. Служба может предоставлять эту информацию в виде метаданных, которые клиенты могут обрабатывать для создания соответствующих клиентов WCF и стеков связи.

Одним из обязательных элементов стека связи является транспортный протокол. Сообщения можно отправлять через интрасети или через Интернет с помощью общих транспортов, таких как HTTP и TCP. Предусмотрены другие транспорты, поддерживающие связь с приложениями очереди сообщений и узлами в сетке одноранговой сети. С помощью встроенных точек расширения WCF можно добавить дополнительные транспортные механизмы.

Модель программирования WCF

Модель программирования WCF основана на коммуникации между двумя сущностями: службой WCF и клиентом WCF. Эта модель программирования инкапсулирована в пространство имен System.ServiceModel в .NET Framework.

Служба WCF

Служба WCF основана на интерфейсе, задающем контракт между службой и клиентом. Она помечается атрибутом ServiceContractAttribute, как показано в следующем коде:

```
[ServiceContract]
public interface IService
{
   [OperationContract]
   string GetAnswer(string mesage);
```

```
}
```

Методы и функции, которые предоставляются службой WCF, задаются путем пометки их атрибутом OperationContractAttribute. Кроме того, можно предоставить сериализованные данные, пометив составной тип атрибутом DataContractAttribute. Это делает возможным привязку данных на клиенте.

После того как интерфейс и его методы были определены, они инкапсулируются в класс, реализующий этот интерфейс. Множественные контракты службы WCF может реализовать единственный ее класс.

Служба WCF предоставляется для использования посредством того, что называется *конечной точкой*. Конечная точка — это единственный способ сообщения с этой службой; невозможно получить к ней доступ посредством прямой ссылки, в отличие от других классов..

Конечная точка состоит из адреса, привязки и контракта. Адрес задает расположение сервера; это может быть URL-адрес, FTP-адрес, а также сетевой или локальный путь. Привязка задает способ сообщения с этой службой. Привязки WCF предоставляют гибкую модель для задания протокола, например HTTP или FTP, механизма обеспечения безопасности, например проверка подлинности Windows или имена и пароли пользователей, и многого другого. Контракт включает операции, предоставляемые классом службы WCF.

Для одной службы WCF может быть предоставлено несколько конечных точек. Это дает возможность разным клиентам взаимодействовать с одной и той же службой разными способами. Например, банковская служба может предоставить одну конечную точку для сотрудников, а другую — для внешних клиентов, причем каждая конечная точка будет использовать другой адрес, привязку и/или контракт.

Клиент WCF

Клиент WCF состоит из прокси, позволяющего приложению сообщаться со службой (WCF), и конечной точки, соответствующей конечной точке, заданной для этой службы. Прокси создается на стороне клиента в файле app.config и включает информацию о типах и методах, предоставляемых службой. В случае служб, предоставляющих несколько конечных точек, клиент может выбрать одну, лучше всего соответствующую его требованиям; например, клиент может выбрать способ сообщения посредством протокола HTTP и проверку подлинности Windows.

Visual Studio упрощает задачу создания клиентов WCF автоматически создавая прокси и конечную точку для службы, добавляемые с помощью Добавить ссылку на службу диалоговое окно. Вся необходимые сведения о конфигурации добавляются в файл app.config. Большей частью все, что необходимо сделать — это создать службу, чтобы потом ее использовать. Для использования службы необходимо создать прокси, в качестве параметра конструктора указав имя конечной точки.

Пример

Пример WCF службы, которая возвращает сообщение, отправленное клиентом.

Интерфейс службы:

```
[ServiceContract]
   public interface IService
{
```

```
[OperationContract]
string GetAnswer(string mesage);
}
```

Реализация интерфейса:

```
public class Service : IService
{
   public string GetAnswer(string mesage)
   {
          return "You said : " + mesage;
   }
   }
```

Определение конечных точек службы, файл App.config:

```
<services>
<service behaviorConfiguration="serviceBehavior" name="WCFServer.Service">
<endpoint address="http://localhost:8888/Service" binding="basicHttpBinding"
bindingConfiguration="" name="ServiceEndpoint"
contract="WCFServer.IService"></endpoint>
<endpoint address="http://localhost:8888/mex" binding="mexHttpBinding"
bindingConfiguration="" bindingName="mex" contract="IMetadataExchange"></endpoint>
</service>
</services>
```

Запуск службы:

```
static void Main(string[] args)          {
          using( var host = new ServiceHost(typeof(Service)))
          {
          host.Open();
          Console.WriteLine("Service is open...");
          Console.ReadKey();
          }
}
```

В приложении клиента добавляем ссылку на службу. Код клиента:

```
static void Main(string[] args)
{
          ServiceClient client = new ServiceClient("ServiceEndpoint");
          if(client != null)
          {
          Console.Write(" Write message : ");
```

```
            string str = Console.ReadLine();
            Console.WriteLine(client.GetAnswer(str));
            }
            Console.ReadKey();
        }
```

App.config клиента, а именно конечная точка:

```
        <client>
            <endpoint address="http://localhost:8888/Service"
binding="basicHttpBinding"
            bindingConfiguration="ServiceEndpoint" contract="ServiceRef.IService"
            name="ServiceEndpoint" />
        </client>
```

Где ServiceRef ссылка на службу, IService – контракт службы.