CLR

Common Language Runtime, CLR — общеязыковая среда выполнения — это среда выполнения, которая подходит для разных языков программирования.

Основные возможности CLR: управление памятью, загрузка сборок, безопасность, обработка исключений, синхронизация, которые доступны в любых языках программирования, использующих CLR.

Процесс компиляции файлов в среде CLR

Исходный код программы может быть написан на любом языке, поддерживающем среду выполнения CLR. Затем соответствующий компилятор проверяет синтаксис и анализирует исходный

код программы. Результатом компиляции будет являться managed module. Помимо управляемого модуля компилятор генерирует IL-код и метаданные.

managed module

Управляемый модуль (portable executable, PE) — стандартный переносимый исполняемый файл 32-разрядной или 64-разрядной Windows, и требует для своего выполнения CLR.

Упровляемый модуль состоит:

- Заголовок PE32/PE32+ - содержит версию разрядности Windows в которой может испольнятся этот модуль, а также, тип файла, временную метку, когда файл был собран.

- Заголовок CLR - заголовок включает нужную версию CLR, флаги, метку метаданных MethodDef - точки входа в управляемый модуль (метод Main), а также месторасположение,

размер метаданных модуля, ресурсов, строгого имени, некоторых флагов и пр;

- Метаданные - Есть два основных вида таблиц — это таблицы, описывающие типы данных и их члены, определенные в исходном коде, и таблицы, описывающие типы данных и их члены,

на которые имеются ссылки в исходном коде; — это блок двоичных данных, состоящий из нескольких таблиц. Существуют три категории таблиц: определений, ссылок и манифестов.

- Код Intermediate Language (IL) - код, создаваемый компилятором при компиляции исходного кода. Впоследствии CLR компилирует IL в машинные команды.

IL-код

IL-код иногда называют управляемым (managed code), потому что CLR управляет его выполнением.

код, создаваемый компилятором при компиляции исходного кода. Впоследствии CLR компилирует IL в машинные команды.

Это не зависящий от процессора машинный язык, разработанный компанией

Microsoft после консультаций с несколькими коммерческими и академическими организациями, специализирующимися на разработке языков и компиляторов.

IL — язык более высокого уровня по сравнению с большинством других машинных языков. Он позволяет работать с объектами и имеет команды для создания

и инициализации объектов, вызова виртуальных методов и непосредственного манипулирования элементами массивов. В нем даже есть команды инициирования и перехвата исключений

для обработки ошибок. IL можно рассматривать как объектно-ориентированный машинный язык.

Метаданные

Метаданные — это набор таблиц данных, описывающих то, что определено в управляемом модуле, например типы и их члены. В метаданных также есть таблицы, указывающие, на что

ссылается управляемый модуль, например на импортируемые типы и их члены, которые всегда связаны с файлом, содержащим IL-код. Благодаря тому что компилятор генерирует метаданные и код

одновременно и привязывает их к конечному управляемому модулю, возможность рассинхронизации метаданных и описываемого ими IL-кода исключена.

Среда Microsoft Visual Studio использует метаданные для облегчения написания кода. Ее функция IntelliSense анализирует метаданные и сообщает, какие методы, свойства,

события и поля предпочтительны в данном случае и какие именно параметры требуются конкретным методам.

Метаданные позволяют сериализовать поля объекта, а затем передать эти данные по сети на удаленный компьютер и там провести процесс десериализации, восстановив объект и его

состояние на удаленном компьютере.

Метаданные позволяют сборщику мусора отслеживать жизненный цикл объектов. При помощи метаданных сборщик мусора может определить тип объектов и узнать, какие именно поля в них

ссылаются на другие объекты

— это блок двоичных данных, состоящий из нескольких таблиц. Существуют три категории таблиц: определений, ссылок и манифестов.

Для каждой сущности, определяемой в компилируемом исходном тексте, компилятор генерирует строку в одной из таблиц, перечисленных в табл. 2.1. В ходе

компиляции исходного текста компилятор также обнаруживает типы, поля, методы, свойства и события, на которые имеются ссылки в исходном тексте. Все сведения

о найденных сущностях регистрируются в нескольких таблицах ссылок, составляющих метаданные.

assembly

Сборка обеспечивает логическую группировку одного или нескольких управляемых модулей или файлов ресурсов.

Сборка позволяет разделить логическое и физическое представления компонента, поддерживающего многократное использование, безопасность и управление версиями. Например,

редко используемые типы и ресурсы можно вынести в отдельные файлы сборки. Отдельные файлы могут загружаться по запросу из Интернета по мере необходимости в процессе выполнения

программы.

Модули сборки также содержат сведения о других сборках, на которые они ссылаются (в том числе номера их версий). Эти данные делают сборку самоописываемой (self-describing).

Другими словами, среда CLR может определить все прямые зависимости данной сборки, необходимые для ее выполнения.

Каждая создаваемая сборка представляет собой либо исполняемое приложение, либо библиотеку DLL, содержащую набор типов для использования в исполняемом

приложении.

сборка (assembly), то есть совокупность одного или нескольких файлов с определениями типов и файлов ресурсов.

Один из файлов сборки выбирается для хранения ее манифеста. Манифест (manifest) — это еще один набор таблиц метаданных, которые в основном

содержат имена файлов, составляющих сборку. Кроме того, эти таблицы описывают версию и региональные стандарты сборки, ее издателя, общедоступные экспортируемые типы,

а также все составляющие сборку файлы.

Подводя итог, можно сказать, что сборка — это единица многократного использования, управления версиями и безопасности типов. Она позволяет распределять

типы и ресурсы по отдельным файлам, чтобы ее пользователи могли решить, какие файлы упаковывать и развертывать вместе. Загрузив файл с манифестом, среда

CLR может определить, какие файлы сборки содержат типы и ресурсы, на которые ссылается приложение. Любому потребителю сборки достаточно знать лишь имя

файла, содержащего манифест, после чего он сможет, не нарушая работы приложения, абстрагироваться от особенностей распределения содержимого сборки по

файлам, которое со временем может меняться.

manifest

Манифест представляет собой обычный набор таблиц метаданных. Эти таблицы описывают файлы, которые входят в сборку, общедоступные экспортируемые типы, реализованные в файлах сборки,

а также относящиеся к сборке файлы ресурсов или данных.

JIT-компилятор

unsafe

По умолчанию компилятор C# компании Microsoft генерирует безопасный код.

Под этим термином понимается код, безопасность которого подтверждается в процессе верификации. Тем не менее компилятор Microsoft C# также позволяет разработчикам писать

небезопасный код, способный напрямую работать с адресами памяти и манипулировать с байтами по этим адресам.

Однако использование небезопасного кода создает значительный риск: небезопасный код может повредить структуры данных и использовать (или даже создавать)

уязвимости в системе безопасности. По этой причине компилятор C# требует, чтобы все методы, содержащие небезопасный код, помечались ключевым словом unsafe,

а при компиляции исходного кода использовался параметр компилятора /unsafe.

FCL

FCL (Framework Class Library) - одним из компонентов .NET Framework, который содержит набор сборок в формате DLL, содержащих несколько тысяч определений типов.

Так как FCL содержит буквально тысячи типов, взаимосвязанные типы объединяются в одно пространство имен.

Чтобы использовать возможности FCL, необходимо знать, какое пространство имен содержит типы, предоставляющие нужную функциональность. Многие типы

поддерживают настройку своего поведения; для этого тип просто объявляется производным от нужного типа FCL.

CTS

CTS (Common Type System) спецификация, которая описывает способ определения и поведение типов

Компания Microsoft предоставляет CTS вместе с другими частями .NET Framework в органкомитет ECMA с целью стандартизации.

Согласно спецификации CTS, тип может содержать нуль и более членов. Иметь Поле Метод Свойство Событие

CTS также задает правила видимости типов и доступа к членам типа. Например, помечая тип как открытый public и тд

Кроме того, CTS определяет правила, управляющие наследованием, работой виртуальных методов, сроком жизни объектов и т. д.

CLI

CLI (Common Language Infrastructure) - общеязыковая инфраструктура, а в документ ECMA-335 содержится спецификация общеязыковой инфраструктуры (CLI)

CLS

CLS (Common Language Speciication); в ней перечислен минимальный набор возможностей, которые должны поддерживаться компилятором для генерирования типов, совместимых с

другими компонентами, написанными на других CLS-совместимых языках на базе CLR.

CLS определяет правила, которым должны соответствовать типы и методы с внешней видимостью, для того чтобы они могли использоваться в любом CLS-совместимом языке программирования

Глобальный кэш сборок

Место, где располагаются совместно используемые сборки, называют глобальным кэшем сборок (global assembly cache, GAC).

Зачем «регистрировать» сборку в каталоге GAC? Представьте себе, что две компании сделали каждая свою сборку OurLibrary, состоящую из единственного файла:

OurLibrary.dll. Очевидно, эти файлы нельзя записывать в один каталог, поскольку файл, копируемый последним, перезапишет первый и тем самым нарушит работу

какого-нибудь приложения. Если для установки в GAC использовать специальный инструмент, он создаст в каталоге %SystemRoot%\Microsoft.NET\Assembly отдельные

папки для каждой из этих сборок и скопирует каждую сборку в свою папку.

System.Object

В CLR каждый объект прямо или косвенно является производным от System.Object.

Благодаря тому, что все типы, в конечном счете, являются производными от System.Object, любой объект любого типа гарантированно имеет минимальный набор методов.

- Equals - возвращает true, если два объекта имеют одинаковые значения.

- GetHashCode - возвращает хеш-код для значения данного объекта.

- ToString - по умолчанию возвращает полное имя типа (this.GetType().FullName). На практике этот метод переопределяют, чтобы он возвращал объект String,

содержащий состояние объекта в виде строки.

- GetType - возвращает экземпляр объекта, производного от Type, который идентифицирует тип объекта, вызвавшего GetType. Возвращаемый объект Type может использоваться

с классами, реализующими отражение для получения информации о типе в виде метаданных. Метод GetType невиртуальный, его нельзя переопределить, поэтому классу не удастся

исказить сведения о своем типе; таким образом обеспечивается безопасность типов

Кроме того, типам, производным от System.Object, доступны некоторые защищенные методы

- MemberwiseClone этот невиртуальный метод создает новый экземпляр типа и присваивает полям нового объекта соответствующие значения объекта this. Возвращается

ссылка на созданный экземпляр

- Finalize - этот виртуальный метод вызывается, когда уборщик мусора определяет, что объект является мусором, но до возвращения занятой объектом памяти в кучу.

В типах, требующих очистки при сборке мусора, следует переопределить этот метод.

Что делает опператор new?

CLR требует, чтобы все объекты создавались оператором new.

Вычисляет количества байт, необходимых для хранения всех экземплярных полей типа и всех его базовых типов.

Для каждого объекта в кучи добавляет дополнительные члены, указатель на объект-тип (type object pointer) и индекс блока синхронизации (sync block index).

Байты этих дополнительных членов добавляются к байтам, необходимым для размещения самого объекта.

Выдялет и резервирует необходимое количество памяти в управляемой куче. Выделенные байты инициализируются нулями (0).

Инициализация указателя на объект-тип и индекса блока синхронизации.

Вызов конструктора экземпляра типа с параметрами, указанными при вызове new.

А так же вызывается конструктор System.Object, но он ничего не делает и просто возвращает управление.

В итоге new возвращает ссылку на вновь созданный объект.

Приведение типов

Приведение типов - это механизм при помощи которого мы можем преобразовать один тип в другой.

C# поддерживает несколько вариантов приведения типов такие как:

Неявные преобразования - в таком случае специального синтаксиса не требуется например преобразования из меньших в большие целочисленные типы, или

Upcusting - преобразования из производных классов в базовые классы.

Явные привидине - если в ходе преобразования данные могут быть утрачены или преобразование может завершиться сбоем по другим причинам.

Downcasting - для приведения типа к производному от него типу.

Для явных преобразований требуется выражение приведения такие как is, as, typeof или (T).

Object o = new Employee();

Employee e = (Employee) o;

Пользовательские преобразования - для этого используются ключевые слова operator и implicit или explicit.

Преобразования с использованием вспомогательных классов, таких как BitConverter, Convert, Parse, TryParse.

Операторы is и as, typeof

Оператор is проверяет совместимость объекта с данным типом и возвращает true или false. Оператор is никогда не генерирует исключение. Для null-ссылок оператор is всегда

возвращает false, так как объекта, тип которого нужно проверить, не существует.

if (o is Employee) { Employee e = (Employee) o; }

Но C# предложен механизм, повышающий эффективность кода с помощью оператора as.

Employee e = o as Employee;

if (e != null) { // Используем e внутри инструкции if}

as возвращает ненулевой указатель на этот объект, а если нет — оператор as возвращает null. При этом верифицирует тип объекта только один раз, а if лишь

сравнивает e с null — такая проверка намного эффективнее, чем определение типа объекта. Если не сравнить полученный оператором результат с null и попытаться

работать с пустой ссылкой, возникнет исключение System.NullReferenceException.

Namespace

Пространства имен используются для логической группировки родственных типов, чтобы разработчику было проще найти нужный тип.

Для компилятора пространство имен — простое средство, позволяющее удлинить имя типа и сделать его уникальным за счет добавления к началу имени групп символов, разделенных точками.

alias - В C# есть еще одна форма директивы using, позволяющая создать псевдоним для отдельного типа или пространства имен. Она удобна, если вы намерены использовать

несколько типов из пространства имен, но не хочется «захламлять» глобальное пространство имен всеми используемыми типами

Примитивные типы

Типы данных, которые поддерживаются компилятором напрямую. У них существуют прямые аналоги в библиотеке классов FCL. Например, типу int System.Int32.

sbyte, byte, short, ushort, int, uint, long, ulong, float, double, decimal, bool, char, string, object, dynemic.

Можно неявное делать приведение типа, если не будет потери данных; пример — преобразование из Int32 в Int64. Но если есть вероятность потери данных, то требуется

явного приведения типа. Например из Int32 в Byte требует явного приведения к типу, так как при больших величинах Int32 теряется точность.

Можно применять литеральную форма записи т.к литералы сами по себе считаются экземплярами типа поэтому можно вызывать экземплярные методы, например, следующим образом:

Console.WriteLine(123.ToString() + 456.ToString()); // "123456"

checked и unchecked

checked и unchecked - механизм управления проверки переполнения, при выполнени арифметических операции над примитивными типами.

checked - если требуется указать, что выражение будет проверяться на переполнение. При переполнении выдастся исключение OverflowException

unchecked - если требуется проигнорировать переполнение. В этом случае результат усекается, чтобы не выйти за пределы диапазона представления чисел для целевого типа выражения.

Примитивный тип данных dynamic

Тип данных dynamic позволяет опустить проверку типов во время компиляции. Кроме того, объекты, объявленные как dynamic, могут в течение работы программы менять свой тип.

Компилятор конвертирует тип dynamic в Object и применяет DynamicAttribute к различным частям метаданных, где это необходимо.

Компилятор не позволит неявно присводить тип Object к другому типу, нужно использовать явное приведение типов, но в случае с типом dynamic, его можно неявно привести к

другому типу, но в таком случае CLR на этапе выполнения проверит правильность приведения с целью обеспечения безопасности типов. Если тип объекта несовместим с приведением,

CLR выдает исключение InvalidCastException.

А так же любое выражение может быть явно приведено к типу dynamic.

Тип dynamic можно использовать для определения аргументов типов обобщенных классов, структур, интерфейсов, делегатов или методов.

В C# следующие типы считаются примитивными и могут использоваться для определения констант:

Boolean, Char, Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal и String.

Ссылочные типы (малова-то)

Память для ссылочных типов всегда выделяется из управляемой кучи.

Оператор new возвращает адрес в памяти, где размещается сам объект.

Память для ссылочных типов всегда выделяется из управляемой кучи.

Дополнительные члены, подлежащие инициализации.

Незанятые полезной информацией байты объекта обнуляются (это касается полей);

Со временем ссылочный тип инициирует сборку мусора.

Значимые типы

Экземпляры значимый типов обычно размещаются в стеке потока, но могут быть встроены и в объект ссылочного тип).

В представляющей экземпляр переменной нет указателя на экземпляр;

Поля экземпляра размещаются в самой переменной.

Экземпляры значимых типов не обрабатываются уборщиком мусора, что соответственно ускоряет работу приложения.

Значимые типы в называются структурами (structure) и перечислениями (enumeration).

Все структуры являются прямыми потомками абстрактного типа System.ValueType, который является производным от типа System.Object.

Все перечисления являются производными от типа System.Enum, производного от System.ValueType.

Значимый тип может реализовать один или несколько выбранных вами интерфейсов.

Значимый тип является изолированным, то есть он не может служить базовым типом для какого-либо другого ссылочного или значимого типа.

Объекты значимого типа существуют в двух формах : неупакованной (unboxed) и упакованной (boxed)

Нельзя создавать в значимом типе новые виртуальные методы.

Методы не могут быть абстрактными и неявно являются запечатанными, то есть их нельзя переопределить.

При инициализации всем членам этого типа присваивается 0.

При обращении к значимому типу не может возникнуть исключение NullReferenceException.

При присвоении переменной значимого типа присваивается другая переменная значимого типа, выполняется копирование всех ее полей

Каждая переменная значимого типа имеет собственную копию данных «объекта», поэтому операции с одной переменной значимого типа не влияют на другую переменную.

Так как неупакованные значимые типы не размещаются в куче, отведенная для них память освобождается сразу при возвращении управления методом, в котором описан экземпляр

этого типа (в отличие от ожидания уборки мусора).

Можно вызывать виртуальные методы (такие, как Equals, GetHashCode или ToString), унаследованные или прееопределенные этим типом.

Приведение неупакованного экземпляра значимого типа к одному из интерфейсов этого типа требует, чтобы экземпляр был упакован, так как интерфейсные переменные

всегда должны содержать ссылку на объект в куче

экземпляр значимого типа упаковывается при вызове реализации базового типа, чтобы в указателе this базового метода передавалась ссылка на объект в куче.

Вместе с тем вызов невиртуального унаследованного метода (такого, как GetType или MemberwiseClone) всегда требует упаковки значимого типа, так как эти методы

определены в System.Object, поэтому методы ожидают, что в аргументе this передается указатель на объект в куче.

Упаковка и распаковка значимых типов

Упаковка (boxing) - служит для преобразования значимого типа в ссылочный.

В управляемой куче выделяется память.

Ее объем определяется длиной значимого типа и двумя дополнительными членами — указателем на типовой объект и индексом блока синхронизации.

Поля значимого типа копируются в выделенную кучу.

Возвращается адрес объекта. Этот адрес является ссылкой на объект, то есть значимый тип превращается в ссылочный.

Распаковка (unboxing) - копирование полей из упакованного объекта, котрый находится в куче в неупакованный объект, который находится в стеке потока.

Сначала извлекается адрес полей упакованного объекта.

Копирвоание полей из кучи в экземпляр значимого типа, находящийся в стеке.

Она гораздо менее ресурсозатратна, чем упаковка.

При распаковке упакованного значимого типа происходит следующее.

1. Если переменная, содержащая ссылку на упакованный значимый тип, равна null, генерируется исключение NullReferenceException.

2. Если ссылка указывает на объект, не являющийся упакованным значением требуемого значимого типа, генерируется исключение InvalidCastException1

Константы

Константа является неявно статическими членами типа значение которой никогда не меняется и определяться во время компиляции, а хранятся в метаданных модуля.

Используется для упрощения чтения и сопровождения кода.

Можно определять только примитивных типов или если присвоить ей значение null.

Нельзя получать адрес константы и передавать ее по ссылке - это означает, что изменять значения константы в разных версиях модуля нельзя соответственно константу надо

использовать, только когда точно известно, что ее значение никогда не изменится, а иначе лучше использовать поля.

Поля

Поле (field) — это член типа, который хранит экземпляр значимого типа или ссылку на ссылочный тип.

Поля могут быть для чтения и записи (изменяемые), или только для чтения (неизменяемые).

Модификаторы:

экземплярое Static readonly volatile

запись в поле разрешается только из кода конструктора. Код, обращающийся к полю, не должен оптимизироваться компилятором,

CLR или оборудованием с целью обеспечения безопасности потоков.

Неизменность поля ссылочного типа означает неизменность ссылки, которую этот тип содержит, а вовсе не объекта, на которую указывает ссылка

Конструкторы

Конструктор — это метод, у которого не задан тип возвращаемого значени, а сигнатура которого совпадает с именем типа, и служит для инициализации экземпляра типа и его поля.

Конструкторыбываюь двух типов: конструктор по умолчанию, без аргументов, и конструктор с параметрами.

Конструкторы могут быть экземпляров, статические и структур.

Конструкторы экземпляров:

Любые поля, не инициализированные конструктором явно, будут содержат 0 или null.

Можно перегружать конструкторы, но количество параметров или их тип должены отличаться.

this - которое позволяет явно вызвать один конструктор через другой конструктор.

base - позволяет вызывать конструктор из базового типа.

Конструкторы экземпляров нельзя наследовать соответственно нельзя применять модификаторы virtual, new, override, sealed и abstract.

Если определить класс без явно заданных конструкторов то будет создан конструктор по умолчанию, реализация которого просто вызывет конструктор без параметров базового класса.

Если в базовом классе нет конструктора без параметров, производный класс должен явно вызвать конструктор базового класса base(), иначе компилятор вернет ошибку.

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных.

В конечном счете всегда вызывается открытый конструктор объекта System.Object без параметров, который просто возвращает управление т.е в System.Object не определено никаких

экземплярных полей.

Для абстрактных классов компилятор создает конструктор по умолчанию с модификатором protected, в противном случае область действия будет открытой (public).

Для статических классов (запечатанных и абстрактных) компилятор не создает конструктор по умолчанию.

Cтатические конструкторы, конструкторы классов, типов или инициализаторы типов.

Статический конструктор служат для инициализации состояния типа или действия, которое выполняется только один раз. Он вызывается автоматически перед созданием первого

экземпляра или ссылкой на какие-либо статические члены. По умолчанию у типа не определено конструктора.

Конструкторы типов можно применять к ссылочным и значимым типам.

У типа не может быть более одного конструктора, а также никогда не бывает параметров.

Конструкторы типов поумолчанию я закрытыми и нельзя использовать модификаторы доступа.

Код конструктора типа может обращаться только к статическим полям типа.

Конструкторы структур

Конструкторы у значимого типа можно не определять, но тогда поля будут инициализированны 0 или null.

Конструктор экземпляра значимого типа выполняется только при явном вызове.

Нельзя определять конструкторы для значимого типа без параметров, должен инициализировать все поля этого типа, но зависит от версии С# т.к в версияъ после 10+ уже можно

создавать конструктор без параметров и инициализировать поля по умолчанию, а так же с 11 не все поля.

Так же можно применять this,

Методы перегруженных операторов

Перегруженные операторы представляют собой метод, который должен сделать что-то с объектом при применении к нему конкретного опператора.

Перегруженные операторные методы должны быть открытыми и статическими.

Тип, по крайней мере, одного из параметров или возвращаемого значения совпадал с типом, в котором определен операторный метод.

public sealed class Complex {

public static Complex operator+(Complex c1, Complex c2) { ... }

}

Методы операторов преобразования

Метод операторов преобразования — методы, преобразующий объект одного типа в объект другого типа.

Методы преобразования должный быть открытыми и статическими.

Тип, хотябы одного из параметров или возвращаемого значения совпадал с типом, в котором определен операторный метод.

Указать implicit или explicit.

operator - сообщаете компилятору, что данный метод представляет собой оператор преобразования.

После ключевого слова operator указывается целевой тип, в который преобразуется объект, а в скобках — исходный тип объекта.

implicit указывает компилятору C#, что наличие в исходном тексте явного приведения типов не обязательно для генерации кода, вызывающего метод оператора преобразования.

explicit позволяет компилятору вызывать метод только тогда, когда в исходном тексте происходит явное приведение типов.

public sealed class Rational {

public static implicit operator Rational(Int32 num) { // Неявно создает Rational из Int32 и возвращает полученный объект

return new Rational(num); }

public static implicit operator Rational(Single num) { // Неявно создает Rational из Single и возвращает полученный объект

return new Rational(num); }

public static explicit operator Int32(Rational r) { // Явно возвращает объект типа Int32, полученный из Rational

return r.ToInt32(); }

public static explicit operator Single(Rational r) { // Явно возвращает объект типа Single, полученный из Rational

return r.ToSingle();

}

}

Методы расширения

Позволяют определить статический метод, который вызывается при помощи синтаксиса экземплярного метода и добавлять новые методы в уже существующие типы

без создания нового производного класса.

Метод расширения должен быть объявлен в статическом необобщенном классе.

Для того чтобы превратить метод в метод расширения нужно добавить ключевое слово this перед первым параметром метода и только первый параметр может быть отмечен

ключевым словом this.

С# поддерживает только методы расширения, а не например свойства расширения.

Методами расширениями можно расширять типы, интерфейсные типы, типов-делегатов(Action) и перечислимые типы.

Для статических классов и структур создавать методы расширения нельзя.

Во время поиска метода расширения компилятор просматривает все статические классы, определенные в области файла, и сканирует их статические методы.

Если существуют два и более методов расширения, должен применяться синтаксис статического метода с указанием имени статического класса, чтобы точно сообщить компилятору,

какой именно метод нужно вызвать.

Существует потенциальная проблема с версиями. Если в будущем разработчики Microsoft добавят экземплярный метод IndexOf к классу StringBuilder с тем же прототипом,

что и в моем примере, то когда я перекомпилирую свой программный код, компилятор свяжет с программой экземплярный метод IndexOf компании Microsoft вместо моего статического

метода IndexOf.

public static class StringBuilderExtensions {

public static Int32 IndexOf(this StringBuilder sb, Char value) {

for (Int32 index = 0; index < sb.Length; index++)

if (sb[index] == value) return index;

return -1;

}

}

Частичные методы

Ключевое слово partial говорит компилятору C#, что исходный код класса, структуры или интерфейса может располагаться в нескольких файлах. Компилятор

объединяет все частичные файлы класса во время компиляции

Методы объявляются только внутри частичного класса или структуры.

Методы должны всегда иметь возвращаемый тип void (в новых версиях шарпа необязательно и тоже самое про модификаторы доступа).

Не могут иметь параметров out, но может иметь параметры, помеченные ключевым словом ref, а также универсальные параметры, экземплярные, статические или unsafe.

Объявление частичного метода и его реализующее объявление должны иметь идентичные сигнатуры.

Если не существует реализующего объявления частичного метода, то нельзя создать делегат, ссылающегося на частичный метод.

Частичные методы всегда считаются закрытыми, компилятор C# запрещает писать ключевое слово private перед объявлением частичного метода

Необязательные и именованные параметры

Необязательными параметрамия являются параметры, которым присвоено значение поумолчанию, после необязательных параметров все последующие параметры также должны быть

необязательными.

Именованные параметры это параметры у которых указано имя параметра и через двоеточие его значение.

Значения по умолчанию указываются для параметров методов, конструкторов методов и индексаторов, а так же для параметров, являющихся частью определения делегатов.

Параметры со значениями по умолчанию должны следовать за всеми остальными параметрами. Существует только одно исключение из правил — параметр массива, помеченный ключевым

словом params. Он должен располагаться после всех прочих параметров, в том числе имеющих значение по умолчанию.

Задавать значения по умолчанию можно для параметров примитивных типов. Сюда относятся также перечислимые типы и ссылочные типы, допускающие присвоение значения null.

Для параметров произвольного значимого типа значение по умолчанию задается как экземпляр этого типа с полями, содержащими нули. Можно использовать как ключевое слово default,

так и ключевое слово new, в обоих случаях генерируется одинаковый IL-код.

Для параметров, помеченных ключевыми словами ref или out, значения по умолчанию не задаются.

Неявно типизированные локальные переменные

Неявно типизированные локальная переменная - переменная помеченная ключевым словом var.

Чтобы определить ее тип, компилятор смотрит на тип выражения с правой стороны от оператора присваивания (=).

var name = "Jeff";

Передача параметров в метод по ссылке

По умолчанию CLR предполагает, что все параметры методов передаются по значению. При передаче объекта ссылочного типа методу передается ссылка на этот объект. То есть метод

может изменить переданный объект, влияя на состояние вызывающего кода. Если параметром является экземпляр значимого типа, методу передается его копия. В этом случае метод

получает собственную копию объекта, а исходный экземпляр сохраняется неизменным

Ключевые слова out и ref позволяеют передавать параметры по ссылке, а не по значению.

Если параметр метода помечен ключевым словом out, вызывающий код может не инициализировать его, пока не вызван сам метод, но вызванный метод должен записать в него знчение,

прежде чем вернуть управление.

Если параметр помечен ключевым словом ref, вызывающий код должен инициализировать его перед вызовом метода, а вызванный метод может как читать, так и записывать значение

параметра.

Использование ключевого слова out со значимыми типами повышает эффективность кода, так как предотвращает копирование экземплярных полей значимого типа при вызовах методов.

Со значимыми типами ключевые слова out и ref позволяют методу управлять единственным экземпляром значимого типа.

Вызывающий код должен выделить память для этого экземпляра, а вызванный метод управляет выделенной памятью.

В случае ссылочных типов вызывающий код выделяет память для указателя на передаваемый объект, а вызванный код управляет этим указателем. В силу этих особенностей использование

ключевых слов out и ref со ссылочными типами полезно, лишь когда метод собирается «вернуть» ссылку на известный ему объект.

Передача переменного количества аргументов

Примененное ключевое слова params позволяет методу принимает массив значений

params заставляет компилятор рассматривать параметр как экземпляр настраиваемого атрибута System.ParamArrayAttribute

Ключевым словом params может быть помечен только последний параметр метода. Он должен указывать на одномерный массив произвольного типа. В последнем параметре метода

допустимо передавать значение null или ссылку на массив, состоящий из нуля элементов.

Метод, принимающий произвольное количество параметров любого типа? Для этого достаточно params Object[].

Вызов метода, принимающего переменное число аргументов, снижает производительность. Чтобы уменьшить негативное влияние этих операций на производительность, можно определить

несколько перегруженных методов, в которыхне используется ключевое слово params.

static Int32 Add(params Int32[] values) {

Типы параметров и возвращаемых значений

Объявляя тип параметров метода, нужно указывать «минимальные» типы, предпочитая интерфейсы базовым классам используя интерфейс IEnumerable<T> вместо сильного типа, List<T>.

Причина,в том, что вызывать метод, передав в него массив, объект List<T>, объект String и т. п., то есть любой объект, тип которого реализует интерфейс IEnumerable<T>.

Второй метод принимает только объекты List<T>, с массивами или объектами String он работать уже не может.

В то же время, объявляя тип возвращаемого методом объекта, желательно выбирать самый сильный из доступных вариантов.

Свойства

Свойства позволяет получить или установить состояние типа или объекта.

Свойства могут быть статические, экземплярные, абстрактные и виртуальные.

Свойства могут помечаться модификатором доступа и определяться в интерфейсах.

Свойства должны иметь имя и тип, но не void.

Не могут быть обобщенными.

Нельзя перегружать, то есть определять несколько свойств с одинаковыми именами, но разным типом.

Свойства нельзя передавать в метод в качестве параметров с ключевым словом out или ref.

set, определяет, что свойство, доступное только для записи.

get, определяет, что свойство, доступное только для чтения.

Свойства могут быть доступны только для чтения или только для записи, в то время как поля всегда доступны и для чтения, и для записи.

Свойство, являясь по сути методом, может выдавать исключения, а при обращениям к полям исключений не бывает.

Свойства с параметрами

У таких свойств методы доступа get получают один или несколько параметров.

Индексатор позволяет разработчику перегружать оператор this[...].

У каждого индексатора должен быть хотя бы один параметр, но параметров может быть и больше.

Тип параметров и и тип возвращаемого значения может быть любым.

В отличие от свойств без параметров, тип может поддерживать множество перегруженных индексаторов при условии, что их сигнатуры различны.

C# индексаторы могут определяться только для экземпляров объектов.

В одном типе можно определять несколько индексаторов при условии, что они получают разные наборы параметров

С# не позволяет различать свойства с одинаковыми наборами параметров и разными именами методов доступа.

CLR не различает свойства без параметров и с параметрами. Для среды любое свойство — это всего лишь пара методов, определенных внутри типа.

Инициализаторы объектов и коллекций

Инициализации объекта

Employee e = new Employee() { Name = "Jeff", Age = 45 };

Реальная выгода от синтаксиса инициализатора объекта состоит в том, что он позволяет программировать в контексте выражения, строя функции, которые

улучшают читабельность кода.

String s = new Employee() { Name = "Jeff", Age = 45 }.ToString().ToUpper();

При инициализации коллекции методу Add можно передать несколько аргументов, для чего используется синтаксис с фигурными скобками:

var table = new Dictionary<String, Int32> { { "Jeffrey", 1 }, { "Kristin", 2 }, { "Aidan", 3 }, { "Grant", 4 }};

Анонимные типы

Механизм который позволяет автоматически объявить кортежный тип при помощи простого синтаксиса. Кортежный тип (tuple type) — это тип,

который содержит коллекцию свойств, каким-то образом связанных друг с другом.

Анонимный тип определяется с помощью ключевого слова var и инициализатора объектов.

Свойства анонимного объекта доступны для установки только в инициализаторе

var o1 = new { Name = "Jeff", Year = 1964 };

Тип System.Tuple

Это обобщенный кортежный тип который наследуется от от класса Object и отличаются количеством обобщенных параметров.

Глава 11. События

Механизм статических событий позволяет типу отправлять уведомления статическим или экземплярным методам.

Механизм экземплярных событий позволяет объекту посылать уведомление статическому или экземплярному методу.

События обычно инициируются в ответ на изменение состояния типа или объекта, порождающего событие.

Событие состоит из двух методов, позволяющих регистрировать и отменять регистрацию на событие, а также используется поле-делегат для управления набором зарегистрированных методов.

Тип, в котором определено событие может уведомлять другие объекты о некоторых особых ситуациях, которые могут случиться.

События — это члены типа, обеспечивающие такого рода взаимодействие.

Определения события в типе означает:

Регистрация своей заинтересованности в событии. Отмена регистрации своей заинтересованности в событии. Оповещение зарегистрированных методов о произошедшем событии.

Модель событий CLR основана на делегатах (delegate). Делегаты обеспечивают реализацию механизма обратного вызова, безопасную по отношению к типам.

Методы обратного вызова (callback methods) позволяют объекту получать уведомления, на которые он подписался.

1. При возникновении события объект, в котором оно возникло, должен передать дополнительную информацию объектам-получателям уведомления о событии. Для

предоставления получателям эту информацию нужно инкапсулировать в собственный класс, содержащий набор закрытых полей и набор открытых только для чтения свойств.

В соответствии с соглашением, классы, содержащие информацию о событиях, передаваемую обработчику события, должны наследовать от типа System.EventArgs, а имя типа

должно заканчиваться словом EventArgs.

2. В C# событие объявляется с ключевым словом event. Каждому члену-событию назначаются область действия, тип делегата, указывающий на прототип вызываемого метода, и имя.

public event EventHandler<NewMailEventArgs> NewMail; // Этап 2. Определение члена-события

3. Определение метода, ответственного за уведомление зарегистрированных объектов о событии.

Для этого должен быть виртуальный защищенный метод, вызываемый из кода класса и его потомков при возникновении события.

Этот метод принимает один параметр, объект MailMsgEventArgs, содержащий дополнительные сведения о событии. Реализация по умолчанию этого метода просто проверяет, есть ли объекты,

зарегистрировавшиеся для получения уведомления о событии, и при положительном результате проверки сообщает зарегистрированным методам о возникновении события.

4. Определение метода, преобразующего входную информацию в желаемое событие

У класса должен быть метод, принимающий некоторую входную информацию и в ответ генерирующий событие.

Обобщения

Обобщения (generics) — еще один механизм, для многократного использования кода — а именно многократного использования алгоритмов.

Разработчик описывает алгоритм, но не указывает типы данных, с которыми тот работает, что позволяет применять алгоритм к объектам разных типов.

Поддерживается обобщенные:

ссылочные и значимые типы, обобщенные интерфейсы, делегаты и методы определенные в ссылочном, в значимом типе или в интерфейсе, а так же при наследовании.

Не поддерживается обобщенные:

перечислимые типы, свойства, индексаторы, события, конструкторы, деструкторы и опператорные методы, все кроме первого могут использоваться в обобщенных типах, чтобы в их коде

использовать параметры-типы этого типа.

При определении обобщенного типа или метода (virtual) переменные, указанные вместо типа T, называются параметрами типа type parameters. T — это имя переменной, которое применяется

в исходном тексте во всех местах, где используется соответствующий тип данных. В случае обобщенного типа или метода указанные типы данных называют аргументами-типами (type arguments).

Преимущества обобщений:

Защита исходного кода. Разработчику, использующему обобщенный алгоритм, не нужен доступ к исходному тексту алгоритма.

Безопасность типов. Попытка использования несовместимого объекта приведет к ошибке на этапе компиляции или исключению во время выполнения.

Более простой и понятный код. Поскольку компилятор обеспечивает безопасность типов, в исходном тексте требуется меньше операция приведения типов, а такой код проще

писать и сопровождать.

Повышение производительности. До появления обобщений один из способов определения обобщенного алгоритма заключался в таком определении всех его членов, чтобы они «умели»

работать с типом данных Object. Но тогда значимые типы, приходилось упаковать этот экземпляр, перед вызовом членов алгоритма, что приводит к более частым процедурам уборки мусора

Поскольку обобщенный алгоритм можно создать для работы с конкретным значимым типом, экземпляры значимого типа могут передаваться по значению и CLR не нужно выполнять упаковку.

Операции приведения типа также не нужны, поэтому CLR не нужно контролировать безопасность типов при их преобразовании, что также ускоряет работу кода.

Результаты тестирования для ссылочного типа не столь впечатляющие: временные показатели и число операций уборки мусора здесь примерно одинаковы.

Поэтому в данном случае у обобщенного алгоритма List реальных преимуществ нет. Тем не менее помните, что применение обобщенного алгоритма значительно

упрощает код и контроль типов при компиляции. Таким образом, хотя выигрыша в производительности практически нет, обобщенный алгоритм обычно имеет и другие преимущества

Обобщенные типы и наследование

Обобщенные тип как и всякий другой может быть произведен от других типов.

Обобщенные интерфейсы

Без них любая попытка работы со значимым типом через необобщенный интерфейс, например, IComparable, будет приводить к необходимости упаковки и потере безопасности типов.

Ссылочный и значимый типы реализуют обобщенный интерфейс путем задания аргументов-типов, или же любой тип реализует обобщенный интерфейс, не задавая аргументы-типы.

Обобщенные делегаты

Поддержка обобщенных делегатов в CLR позволяет передавать методам обратного вызова любые типы объектов, обеспечивая при этом безопасность типов. Более того, благодаря

обобщенным делегатам экземпляры значимого типа могут передаваться методам обратного вызова без упаковки.

Делегат — это просто определение класса с помощью четырех методов: конструктора и методов Invoke, BeginInvoke и EndInvoke. При определении типа делегата с параметрами типа

компилятор задает методы класса делегата, а параметры типа применяются ко всем методам, параметры и возвращаемые значения которых относятся к указанному параметру типа.

Контравариантные и ковариантные аргументы-типы в делегатах и интерфейсах (ПЕРЕЧИТАТЬ)

Каждый из параметров-типов обобщенного делегата должен быть помечен как ковариантный или контравариантный. Это позволяет вам осуществлять приведение типа переменной

обобщенного делегата к тому же типу делегата с другим параметром-типом. Параметры-типы могут быть:

Инвариантными. Параметр-тип не может изменяться.

Контравариантными. Параметр-тип может быть преобразован от класса к классу, производному от него. В языке C# контравариантный тип обозначается ключевым словом in.

Контравариантный параметр-тип может появляться только во входной позиции, например, в качестве аргументов метода.

Ковариантными. Аргумент-тип может быть преобразован от класса к одному из его базовых классов. В языке С# ковариантный тип обозначается ключевым словом out. Ковариантный

параметр обобщенного типа может появляться только в выходной позиции, например, в качестве возвращаемого значения метода.

Обобщенные методы

Все методы, определенные в обобщенных типах, могут использовать их параметр-тип.

Параметр-тип может использоваться для параметров метода, возвращаемого значения и типа заданной внутри него локальной переменной, а так же метод может иметь собственные

параметры-типы, которые могут применяться для параметров, возвращаемых значений или локальных переменных.

Использование обобщенных типов с методами, получающими параметры out и ref, должны быть того же типа, что и параметр метода, чтобы избежать возможного нарушения безопасности типов.

Верификация и ограничения

Ограничение сужает перечень типов, которые можно передать в обобщенном аргументе, и расширяет возможности по работе с этими типами.

public static T Min<T>(T o1, T o2) where T : IComparable<T>

Маркер where сообщает компилятору, что указанный в T тип должен реализовывать обобщенный интерфейс IComparable того же типа (T). Благодаря этому

ограничению компилятор разрешает методу вызвать метод CompareTo, потому что последний определен в интерфейсе IComparable<T>.

Ограничения можно применять к параметрам типа обобщенных типов и обобщенных методов.

Не поддерживается перегрузка по именам параметров типа или по именам ограничений, только по арности:

При переопределении виртуального обобщенного метода в переопределяющем методе должно быть задано то же число параметров-типов, а они, в свою очередь,

наследуют ограничения, заданные для них методом базового класса. Переопределяемый метод вообще не вправе задавать ограничения для своих параметров-типов, а только может

переименовывать параметры-типы. Аналогично, при реализации интерфейсного метода в нем должно задаваться то же число параметров-типов, что и в интерфейсном методе,

причем эти параметры-типы наследуют ограничения, заданные для них методом интерфейса.

internal sealed class AType {}

internal sealed class AType<T> {}

internal sealed class AType<T1, T2> {}

К параметру-типу могут применяться следующие ограничения основное (primary), дополнительное (secondary) и/или ограничение конструктора (constructor constraint).

Основные ограничения

Основным ограничением может быть ссылочный тип, указывающий на незапечатанный класс, а так же в параметре-типе можно задать не более одного основного ограничения.

Задании ограничения ссылочного типа означает, что заданный аргумент-тип будет относиться либо к типу, указанному в ограничении, либо к производному от него.

Есть два особых основных ограничения: class и struct.

Ограничение class гарантирует компилятору, что указанный аргумент-тип будет иметь ссылочный тип. К нему относятся все классы, интерфейсы, делегаты и массивы.

Ограничение struct гарантирует компилятору, что указанный аргумент типа будет иметь значимый тип. К нему относятся все значимые типы, и перечисления, кроме Nullable<T>

Нельзя использовать следующие ссылочные типы: System.Object, System.Array, System.ValueType и System.Void, но в новых версиях уже можно использовать System.Enum,

System.Delegate, System.MulticastDelegate

Если параметр-тип не задает основное ограничение, автоматически задается тип System.Object, но нельзя указывать явно.

Дополнительные ограничения

Для параметра-типа могут быть заданы нуль или более дополнительных ограничений.

Существуют интерфйесные ограничения и ограничение параметра типа.

При задании ограничения интерфейсного типа вы гарантируете компилятору, что указанный аргумент-тип будет определять тип, реализующий этот интерфейс. А так как можно задать

несколько интерфейсных ограничений, в аргументе типа должен указываться тип, реализующий все интерфейсные ограничения и все основные ограничения, если они заданы.

Ограничением параметра типа (type parameter constraint). Позволяет задавать отношение между агрументами типа. К параметрутипу может быть применено нуль и более ограничений

Ограничения конструктора

Можно задать не более одного ограничения конструктора.

Ограничение конструктора гарантирует компилятору, что указанный аргумент-тип будет иметь неабстрактный тип, имеющий открытый конструктор без параметров.

Нельзя задавать одновременно struct(значимых типов) и new(), т.к является избыточным.

internal sealed class ConstructorConstraint<T> where T : new() {}

Наследование в классах и интерфейсах

Любой производный от Object класс наследует:

Сигнатуры методов и реализацию этих методов.

В CLR у класса может быть один и только один прямой «родитель». Базовый класс предоставляет набор сигнатур и реализации этих методов. При этом новый класс может стать

базовым для другого класса, и при этом новый производный класс унаследует все сигнатуры методов и их реализации

CLR позволяет реализовать ограниченное множественное наследование через интерфейсы (interfaces).

Интерфейс представляет собой средство назначения имени набору сигнатур методов и содержит реализаций методов.

Класс может наследовать интерфейс через указание его имени, но должен явно содержать реализации интерфейсных методов.

Класс может наследовать несколько интерфейсов, но класс при этом должен реализовать все унаследованные методы интерфейсов.

Одна из замечательных особенностей наследования классов — возможность подстановки экземпляров производного типа в любые контексты, в которых выступают

экземпляры базового типа. Аналогичным образом наследование от интерфейсов позволяет подставлять экземпляры типа, реализующего интерфейс, во все контексты,

где требуются экземпляры указанного интерфейсного типа.

Интерфейс представляет собой именованный набор сигнатур методов.

В интерфейсах можно определять методы, события, свойства и индексаторы.

В интерфейсе нельзя определять конструкторы, экземплярные поля и статические челены.

Для определения интерфейса используется ключевое слово interface.

При определении интерфейсного типа можно указать требуемую область видимости и доступа (public, protected, internal и т. п.).

В соответствии с соглашением имена интерфейсных типов начинаются с прописной буквы I.

CLR поддерживает обобщенные интерфейсы и интерфейсные методы.

Интерфейса может «наследовать» другие интерфейсы, а любой класс, наследующий интерфейс, должен реализовать все методы, определенные в интерфейсах.

Наследование интерфейсов

Компилятор C# требует, чтобы метод, реализующий интерфейс, отмечался модификатором public.

CLR требует, чтобы интерфейсные методы были виртуальными. Если метод явно не определен в коде как виртуальный, компилятор сделает его таковым и, вдобавок, запечатанным.

Это не позволяет производному классу переопределять интерфейсные методы.

Если явно задать метод как виртуальный, компилятор сделает его таковым и оставит незапечатанным, что предоставит производному классу возможность переопределять интерфейсные

методы.

Производный класс не в состоянии переопределять интерфейсные методы, объявленные запечатанными, но может повторно унаследовать тот же интерфейс и предоставить собственную

реализацию его методов. При вызове интерфейсного метода объекта вызывается реализация, связанная с типом самого объекта.

Подробнее о вызовах интерфейсных методов

Тип System.String из библиотеки FCL наследует сигнатуры и реализации методов System.Object. Кроме того, тип String реализует несколько интерфейсов:

IComparable, ICloneable, IConvertible, IEnumerable, IComparable<String>, IEnumerable<Char> и IEquatable<String>.

Это значит, что типу String не требуется реализовывать методы, имеющиеся в его базовом типе Object. Однако тип String должен реализовывать методы, объявленные во всех интерфейсах.

CLR допускает определение полей, параметров или локальных переменных, имеющих интерфейсный тип. Используя переменную интерфейсного типа, можно вызывать методы,

определенные этим интерфейсом.

К тому же CLR позволяет вызывать методы, определенные в типе Object, поскольку все классы наследуют его методы

Явные и неявные реализации интерфейсных методов

Если в C# перед именем метода указано имя интерфейса, в котором определен этот метод (в нашем примере — IDisposable.Dispose), то вы создаете явную реализацию интерфейсного

метода (Explicit Interface Method Implementation, EIMI).

При явной реализации интерфейсного метода в C# нельзя указывать уровень доступа. Однако когда компилятор создает метаданные для метода,

он назначает ему закрытый уровень доступа (private), что запрещает любому коду использовать экземпляр класса простым вызовом интерфейсного метода. Единственный способ вызвать

интерфейсный метод — обратиться через переменную этого интерфейсного типа.

EIMI-метод не может быть виртуальным, а значит, его нельзя переопределить.

Обобщенные интерфейсы

Обобщенный интерфейс - это механиз который позволяет не указывать аргумент-тип, а только описать набор сигнатур(посмотреть определение Интерфейсов), а уже в

ссылочном или значимом типе, который реализуют обобщенный интерфейс указать этот аргументов-типов. Любая попытка работы со значимым типом через необобщенный интерфейс,

будет приводить к необходимости упаковки и потере безопасности типов, cоответственно :

Обеспечивают безопасность типов на стадии компиляции.

При работе со значимыми типами требуется меньше операций упаковки.

Класс может реализовать один интерфейс многократно, просто используя параметры различного типа.

Параметры интерфейса обобщенного типа могут быть также помечены как контравариантые или ковариантные, что позволяет более гибко использовать интерфейсы

Обобщения и ограничения интерфейса

При обобщении можно использовать интерфейс в качестве ограничения это позволяет:

Ограничить параметр-тип одним и более интерфейсами. В этом случае тип передаваемого параметра должен реализовывать все ограничения.

Избавляет от упаковки при передаче экземпляров значимых типов. Соответственно, если тип аргумента объявить интерфейсом, то будет выполнена упаковка.

Для ограничений интерфейсов компилятор генерирует определенные IL-инструкции, которые вызывают интерфейсный метод для значимого типа напрямую, без упаковки. Кроме

использования ограничений интерфейса нет другого способа заставить компилятор генерировать такие IL-инструкции; следовательно, во всех других случаях вызов интерфейсного

метода для значимого типа всегда приводит к упаковке.

Реализация нескольких интерфейсов с одинаковыми сигнатурами и именами методов

Иногда нужно определить тип, реализующий несколько интерфейсов с методами, у которых совпадают имена и сигнатуры. Допустим, два интерфейса определены следующим образом:

Требуется определить тип, реализующий оба этих интерфейса. В этом случае нужно реализовать члены типа путем явной реализации методов так как этот тип должен реализовывать

несколько различных методов, нужно сообщить компилятору C#, какой из методов реализацию для конкретного интерфейса.

Код, в котором используется объект MarioPizzeria, должен выполнять приведение типа к определенному интерфейсу для вызова нужного метода:

Опасности явной реализации интерфейсных методов

Отсутствие документации, объясняющей, как именно тип реализует EIMI-метод, а также отсутствие IntelliSense-поддержки в Microsoft Visual Studio;

При приведении к интерфейсному типу экземпляры значимого типа упаковываются;

EIMI нельзя вызвать из производного типа

Требование приведения типа далеко не очевидно, многие разработчики не могут самостоятельно до этого додуматься. Но на этом проблемы не заканчиваются — при

приведении значимого типа Int32 к интерфейсному типу IConvertible значимый тип упаковывается, что приводит к лишним затратам памяти и снижению производительности. Это вторая

серьезная проблема.

Третья и, наверное, самая серьезная проблема с EIMI состоит в том, что явная реализация интерфейсного метода не может вызываться из производного класса.

Дилемма разработчика: базовый класс или интерфейс?

Связь потомка с предком. Любой тип может наследовать только одну реализацию. Если производный тип не может ограничиваться отношением типа «является частным случаем» с

базовым типом, нужно применять интерфейс, а не базовый тип. Интерфейс подразумевает отношение «поддерживает функциональность». Например, тип может преобразовывать экземпляры самого

себя в другой тип (IConvertible), может создать набор экземпляров самого себя (ISerializable) и т. д. Заметьте, что значимые типы должны наследовать от типа System.ValueType и

поэтому не могут наследовать от произвольного базового класса. В этом случае нужно определять интерфейс.

Простота использования. Разработчику проще определить новый тип, производный от базового, чем создать интерфейс. Базовый тип может предоставлять массу функций, и в производном

типе потребуется внести лишь незначительные изменения, чтобы изменить его поведение. При создании интерфейса в новом типе придется реализовывать все члены.

Четкая реализация. Как бы хорошо ни был документирован контракт, вряд ли будет реализован абсолютно корректно. По сути, проблемы COM связаны именно с этим — вот почему

некоторые COM-объекты нормально работают только с Microsoft Word или Microsoft Internet Explorer. Базовый тип с хорошей реализацией основных функций — прекрасная отправная точка,

вам останется изменить лишь отдельные части.

Управление версиями. Когда вы добавляете метод к базовому типу, производный тип наследует стандартную реализацию этого метода без всяких затрат.

Пользовательский исходный код даже не нужно перекомпилировать. Добавление нового члена к интерфейсу трребует изменения пользовательского исходного кода и его перекомпиляции

Символы

Символы в .NET Framework всегда представлены 16-разрядными кодами стандарта Юникод

Символ представляется экземпляром структуры System.Char

Тип System.Char у него лишь два открытых неизменяемых поля: константа MinValue, и константа MaxValue.

Для облегчения работы с типом Char имеется несколько статических методов также несколько собственных экземплярных методов

Три способа преобразования различных числовых типов в экземпляры типа Char, и наоборот.

Приведение типа. Самый эффективный способ, так как компилятор генерирует IL-команды преобразования без вызовов каких-либо методов. Для преобразования типа Char в числовое

значение, такое как Int32, приведение подходит лучше всего.

Использование типа Convert. У типа System.Convert есть несколько статических методов, корректно преобразующих Char в числовой тип и обратно. Все эти методы выполняют

преобразование как проверяемую операцию, чтобы в случае потери данных при преобразовании возникало исключение OverflowException.

Использование интерфейса IConvertible. В типе Char и во всех числовых типах библиотеки .NET Framework Class Library (FCL) реализован интерфейс

IConvertible, в котором определены такие методы, как ToUInt16 и ToChar. Этот способ наименее эффективен, так как вызов интерфейсных методов для числовых типов приводит к упаковке

экземпляра: Char и все числовые типы являются значимыми типами. Методы IConvertible генерируют исключение System.InvalidCastException, если преобразование невозможно

(например, преобразование типа Char в Boolean) или грозит потерей данных. Во многих типах (в том числе Char и числовых типах FCL) используются EIMI-реализации методов

IConvertible (см. главу 13), а значит, перед вызовом какого-либо метода этого интерфейса нужно выполнить явное приведение экземпляра к IConvertible.

Все методы IConvertible за исключением GetTypeCode принимают ссылку на объект, реализующий интерфейс IFormatProvider. Этот параметр полезен,

когда по какой-либо причине при преобразовании требуется учитывать региональные стандарты. В большинстве операций преобразования в этом параметре

передается null, потому что он все равно игнорируется

Строки

System.String, — представляет неизменяемый упорядоченный набор символов. Будучи прямым потомком Object, он является ссылочным типом, по этой причине строки всегда

размещаются в куче

Способы формирования строк:

Во многих языках (включая C#) String относится к примитивным типам, то есть компилятор разрешает вставлять литеральные строки непосредственно в исходный код.

Компилятор помещает эти литеральные строки в метаданные модуля, откуда они загружаются и используются во время выполнения.

String s = "Hi" + " " + "there."; // Конкатенация трех литеральных строк образует одну литеральную строку

Поскольку все строки в этом коде литеральные, компилятор выполняет их конкатенацию на этапе компиляции, в результате в метаданных модуля оказывается лишь строка "Hi there.".

Конкатенация нелитеральных строк с помощью оператора + происходит на этапе выполнения. Для конкатенации нескольких строк на этапе выполнения оператор + применять нежелательно,

так как он создает в куче несколько строковых объектов. Вместо него рекомендуется использовать тип System.Text.StringBuilder.

буквальные строки (verbatim strings) — обычно используют при задании пути к файлу или каталогу и при работе с регулярными выражениями.

String file = "C:\\Windows\\System32\\Notepad.exe"; String file = @"C:\Windows\System32\Notepad.exe";

Неизменяемые строки

Самое важное, что нужно помнить об объекте String — то, что он неизменяем; то есть созданную однажды строку нельзя сделать длиннее или короче, в ней нельзя изменить ни одного

символа. Неизменность строк дает определенные преимущества.

Для начала можно выполнять операции над строками, не изменяя их.

Благодаря неизменности строк отпадает проблема синхронизации потоков при работе со строками.

В CLR несколько ссылок String могут указывать на один, а не на несколько разных строковых объектов, если строки идентичны. А значит, можно сократить количество строк в

системе и уменьшить расход памяти — это именно то, что непосредственно относится к интернированию строк (string interning). По соображениям производительности тип String

тесно интегрирован с CLR. В частности, CLR «знает» точное расположение полей в этом типе и обращается к ним напрямую. За повышение производительности и прямой доступ приходится

платить небольшую цену: класс String является запечатанным.

Сравнение строк

Две причины, по которым приходится сравнивать строки. Мы сравниваем две строки для выяснения, равны ли они, и для сортировки

При сортировке всегда нужно учитывать регистр символов. Две строки, отличающиеся лишь регистром символов, будут считаться одинаковыми и поэтому при каждой

сортировке они могут упорядочиваться в произвольном порядке, что может приводить пользователя в замешательство.

Интернирование строк

Если в приложении строки сравниваются часто методом порядкового сравнения с учетом регистра или если в приложении ожидается появление множества одинаковых строковых объектов, то

для повышения производительности надо применить механизм интернирования строк (string interning).

При инициализации CLR создает внутреннюю хеш-таблицу, в которой ключами являются строки, а значениями — ссылки на строковые объекты в управляемой куче. Вначале таблица,

разумеется, пуста. В классе String есть два метода, предоставляющие доступ к внутренней хеш-таблице:

String Intern(String str); String IsInterned(String str);

Первый из них, Intern, ищет String во внутренней хеш-таблице. Если строка обнаруживается, возвращается ссылка на соответствующий объект String. Иначе создается копия строки,

она добавляется во внутреннюю хеш-таблицу, и возвращается ссылка на копию. Если приложение больше не удерживает ссылку на исходный объект String, уборщик мусора вправе

освободить память, занимаемую этой строкой. Обратите внимание, что уборщик мусора не вправе освободить строки, на которые ссылается внутренняя хеш-таблица, поскольку в ней самой

есть ссылки на эти String. Объекты String, на которые ссылается внутренняя хеш-таблица, нельзя освободить, пока не выгружен соответствующий домен приложения или не закрыт поток.

Как и Intern, метод IsInterned получает параметр String и ищет его во внутренней хеш-таблице. Если поиск удачен, IsInterned возвращает ссылку на интернированную строку.

В противном случае он возвращает null, а саму строку не вставляет в хеш-таблицу.

Создание пулов строк

При обработке исходного кода компилятор должен каждую литеральную строку поместить в метаданные управляемого модуля. Если одна строка встречается в исходном коде много раз,

размещение всех таких строк в метаданных приведет к увеличению размера результирующего файла.Чтобы не допустить роста объема кода, многие компиляторы (в том числе C#)

хранят литеральную строку в метаданных модуля только в одном экземпляре. Все упоминания этой строки в исходном коде компилятор заменяет ссылками на ее экземпляр в метаданных.

StringBuilder

У объекта StringBuilder предусмотрено поле со ссылкой на массив структур Char. Используя члены StringBuilder, можно эффективно манипулировать этим массивом, сокращая строку и

изменяя символы строки.

При увеличении строки, представляющей ранее выделенный массив символов, StringBuilder автоматически выделит память для нового, большего по размеру массива,

скопирует символы и приступит к работе с новым массивом.

А прежний массив попадет в сферу действия уборщика мусора.

Сформировав свою строку с помощью объекта StringBuilder, «преобразуйте» массив символов StringBuilder в объект String, вызвав метод ToString типа StringBuilder.

Этот метод просто возвращает ссылку на поле-строку, управляемую объектом StringBuilder. Поскольку массив символов здесь не копируется, метод выполняется очень быстро

У типа StringBuilder несколько конструкторов. Задача каждого из них — выделять память и инициализировать три внутренних поля, управляемых любым

объектом StringBuilder.

Максимальная емкость (maximum capacity) — поле типа Int32, которое задает максимальное число символов, размещаемых в строке. По умолчанию оно равно

Int32.MaxValue (около двух миллиардов).

Емкость (capacity) — поле типа Int32, показывающее размер массива символов StringBuilder. По умолчанию оно равно 16. Если известно, сколько символов

предполагается разместить в StringBuilder, укажите это число при создании объекта StringBuilder. При добавлении символов StringBuilder определяет,

не выходит ли новый размер массива за установленный предел. Если да, то StringBuilder автоматически удваивает емкость, и исходя из этого значения,

выделяет память под новый массив, а затем копирует символы из исходного массива в новый. Исходный массив в дальнейшем утилизируется сборщиком мусора.

Динамическое увеличение массива снижает производительность, поэтому его следует избегать, задавая подходящую емкость в начале работы с объектом.

Массив символов (character array) — массив структур Char, содержащий набор символов «строки». Количество символов в строке можно получить через

свойство Length типа StringBuilder.

Члены типа StringBuilder

Тип StringBuilder в отличие от String представляет изменяемую строку. Это значит, что многие члены StringBuilder изменяют содержимое в массиве символов,

не создавая новых объектов, размещаемых в управляемой куче. StringBuilder выделяет память для новых объектов только в двух случаях:

при динамическом построении строки, размер которой превышает установленную емкость;

при вызове метода ToString типа StringBuilder.

Форматирование нескольких объектов в одну строку

String s = String.Format("On {0}, {1} is {2} years old.", new DateTime(2012, 4, 22, 14, 35, 5), "Aidan", 9);

Получение объекта посредством разбора строки

Любой тип, способный разобрать строку, имеет открытый, статический метод Parse. Он получает String, а на выходе возвращает экземпляр данного типа; в какомто смысле Parse

ведет себя как фабрика. В FCL метод Parse поддерживается всеми числовыми типами, а также типами DateTime, TimeSpan и некоторыми другими

public static Int32 Parse(String s, NumberStyles style, IFormatProvider provider);

Взглянув на прототип, вы сразу поймете суть работы этого метода. Параметр s типа String идентифицирует строковое представление числа, которое необходимо разобрать для получения

объекта Int32. Параметр style типа System.Globalization.NumberStyles — это набор двоичных флагов для идентификации символов, которые метод Parse должен найти в строке.

А параметр provider типа IFormatProvider идентифицирует объект, используя который метод Parse может получить информацию о региональных стандартах, о чем речь шла ранее.

Так, в следующем фрагменте при обращении к Parse генерируется исключение System.FormatException, так как в начале разбираемой строки находится пробел:

Int32 x = Int32.Parse(" 123", NumberStyles.None, null);

Чтобы «пропустить» пробел, надо вызвать Parse с другим параметром style:

Int32 x = Int32.Parse(" 123", NumberStyles.AllowLeadingWhite, null);

Перечислимые типы

Перечислимым (enumerated type) называют тип, в котором описан набор пар, состоящих из символьных имен и значений.

Программу, где используются перечислимые типы, проще написать и понять, а у разработчиков возникает меньше проблем с ее сопровождением т.к программист видит осмысленные

символьные имена, а не цифры.

Перечислимые типы подвергаются строгой проверке типов. Например, компилятор сообщит об ошибке, если в качестве значения я попытаюсь передать методу

тип Color.Orange (оранжевый цвет), когда метод ожидает перечислимый тип Fruit (фрукт).

Каждый перечислимый тип напрямую наследуется от типа System.Enum <= System.ValueType <= System.Object. Из этого следует, что перечислимые типы относятся к значимым типам(упаковываются)

В отличии от других значимых типов, у перечислимого типа не может быть методов, свойств и событий. Методы можно имитировать при помощи методов расширения.

При компиляции перечислимого типа компилятор C# превращает каждый идентификатор в константное поле типа.

Перечислимый тип — это обычная структура, внутри которой описан набор константных полей и одно экземплярное поле. Константные поля попадают в метаданные сборки,

откуда их можно извлечь с помощью механизма отражения. Это означает, что в период выполнения можно получить все идентификаторы и их значения, связанные перечислимым типом,

а также преобразовать строковый идентификатор в эквивалентное ему числовое значение. Эти операции предоставлены базовым типом System.Enum, который предлагает статические и

экземплярные методы, выполняющие специальные операции над экземплярами перечислимых типов, избавляя вас от необходимости использовать отражение

В основе любого перечисления лежит один из основных типов, например byte, sbyte, short, ushort, int используется в C# по умолчанию, uint, long и ulong.

Компилятор C# считает перечислимые типы примитивными, поэтому для операций с их экземплярами применяются уже знакомые нам операторы (==, !=, <, >, <=, >=, +, –, ^, &,

|, ~, ++ и ––).

Компилятор C# допускает приведение экземпляров одного перечислимого типа к другому.

Также поддерживается явное и неявное приведение к числовому типу.

Имеющийся экземпляр перечислимого типа можно связать со строковым представлением — для этого следует вызвать ToString, унаследованный от System.Enum:

public static Array GetValues(Type enumType) и public Array GetEnumValues();

Этот метод вместе с методом ToString позволяет вывести все идентификаторы и числовые значения перечисления:

Битовые флаги

Перечиления можно использовать для создании битовых флагов, благадаря этому экземпляр enum может иметь комбинацию значений констант, определенных в списке.

Для создания перечисления с битовым флагом нужно добавить атрибут [Flags] или [FlagsAttribute]

Определяя перечислимый тип, битовыми флагами, каждому идентификатору следует явно присвоить числовое значение.

Стоит добавлять идентификатор None, значение которого определено как 0.

Константы должны быть степенями двойки - это гарантирует, что комбинации флагов не будут перекрываться.

Если есть часто используемые комбинации флагов, то их можно добавить в перечисление.

Стоит использовать только положительные константы, чтобы не вводить никоо в заблуждение.

Т.к константы должны содержать степеньдвойки, то можно задавать их несколькими способами: двоичными литералами, числовым литераом с операцией сдвига, десятичными числами и

шестнадцатиречными.

Для комбинирования, сброса и проверки флагов используются битовые операции. Битовые маски широко используются в среде .Net, в частности для указания привязки объектов к контейнерам

и определения уровня доступа к файлам, это лишь малая часть из перечислений которые входят в состав .Net Framework.

https://programm.top/c-sharp/tutorial/enum-bit-flags/

Массивы

Массив представляет собой механизм, позволяющий рассматривать набор элементов как единую коллекцию.

Поддерживает одномерные (single-dimension), многомерные (multidimension) и нерегулярные (jagged) массивы.

Базовым для всех массивов является абстрактный класс System.Array, производный от System.Object и являются ссылочным типом и размещаются в управляемой куче,

а переменная в приложении содержит не элементы массива, а ссылку на массив.

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 (IL).

Для многомерных массивов характерным есть ранг(rank), или количество измерений. Двумерный массив – это таблица, трехмерный – куб и тд

Доступ к элементам осуществляется посредством двух и более индексов

Double[,] myDoubles = new Double[10, 20]; // Создание двухмерного массива типа Doubles

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.

Point[][] myPolygons = new Point[3][] // Создание одномерного массива из массивов типа Point

Чтобы привести массив к другому оба типа массивов должны иметь одинаковую размерность; кроме того, должно иметь место неявное или явное преобразование из типа элементов

исходного массива в целевой тип.

CLR не поддерживает преобразование массивов с элементами значимых типов в другие типы, но можно обойти это ограничение при помощи метода Array.Copy, который создает новый

массив и заполняет его элементами.

В CLR поддерживаются массивы двух типов:

Одномерные массивы с нулевым начальным индексом. Иногда их называют SZ-массивами (от английского single-dimensional, zero-based), или векторами.

Одномерные и многомерные массивы с неизвестным начальным индексом.

Доступ к элементам одномерного массива с нулевой нижней границей осуществляется немного быстрее, чем доступ к элементам многомерных массивов

или массивов с ненулевой нижней границей. Есть несколько причин такому поведению.

Во-первых, специальные команды для работы с одномерными массивами с нулевой нижней границей (newarr, ldelem, ldelema, ldlen и stelem) позволяют

JIT-компилятору генерировать оптимизированный код. При этом предполагается, что первый индекс равен нулю, то есть при доступе к элементам отсутствует

необходимость вычислять смещение. Кроме того, в общем случае компилятор умеет выносить код проверки границ за пределы цикла.

К сожалению, обращение к элементам многомерного массива или массива с ненулевой нижней границей происходит намного медленней. Ведь в этих случаях код

проверки индекса не выносится за пределы цикла и проверка осуществляется на каждой итерации. Кроме того, компилятор добавляет код, вычитающий из текущего

индекса нижнюю границу массива. Это также замедляет работу программы даже в случае многомерных массивов с нулевой нижней границей. Если вы серьезно оза

бочены проблемой производительности, имеет смысл использовать нерегулярные массивы (массивы массивов).

Делегаты

Делегат (delegates) - функциями обратного вызова, которая обеспечивает безопасность типов при выполнении обратного вызова методов, а так же позволяют последовательно вызовать

нескольо методов.

Делегаты могут вызывать статическин и экземплярные методоы.

internal delegate void Feedback(Int32 value);// Объявление делегата;

Обратный вызов статических методов

Созданный делегат Feedback. Этот делегат служит оболочкой для другого метода, позволяя выполнить обратный вызов метода, через оболочку. В примере имя статического

метода Program.FeedbackToConsole передается конструктору Feedback, указывая, что именно для него требуется создать оболочку. Возвращенная оператором new ссылка

передается третьему параметру метода Counter, который в процессе выполнения будет вызывать статический метод FeedbackToConsole.

Обратный вызов экземплярных методов

Когда при вызове метода Counter создается делегат Feedback, его конструктору передается объект p.FeedbackToFile. В результате делегат превращается в оболочку для

ссылки на метод FeedbackToFile, который является не статическим, а экземплярным методом. Когда метод Counter обращается к методу обратного вызова, который задан

аргументом fb, вызывается экземплярный метод FeedbackToFile, а адрес только что созданного объекта p передается этому методу в качестве неявного аргумента this.

Делегаты определяются при помощи ключевого слова C# delegate, а оператор new создает экземпляр делегата.

Сигнатура метода должна быть совместима с сигнатурой делегата - это означает что оба типа должны принимать один и тот же аргумент и возвращать значение одного и тогоже типа.

Класс делегата, определенный компилятором, содержит четыре метода: конструктор, а также методы Invoke, BeginInvoke и EndInvoke.

Делегаты можно определять как внутри класса (вложенные в другой класс), так и в глобальной области видимости. По сути, так как делегаты являются классами,

их можно определить в любом месте, где может быть определен класс.

Класс System.MulticastDelegate является производным от класса System.Delegate, который, в свою очередь, наследует от класса System.Object.

Любые типы делегатов — это потомки класса MulticastDelegate, от которого они наследуют все поля, свойства и методы

Делегаты поддерживают ковариантность и контравариантность ссылочных типов при привязке метода к делегату.

Ковариантность (covariance) означает, что метод может возвратить тип, производный от типа, возвращаемого делегатом.

Контравариантность (contra-variance) означает, что метод может принимать параметр, который является базовым для типа параметра делегата.

Например: delegate Object MyCallback(FileStream s);

Определив делегат таким образом, можно получить экземпляр этого делегата, связанный с методом, прототип которого выглядит примерно так:

String SomeMethod(Stream s);

Здесь тип значения, возвращаемого методом тип String, является производным от типа, возвращаемого делегатом (Object); такая ковариантность разрешена.

Тип параметра метода SomeMethod (тип Stream) является базовым классом для типа параметра делегата (FileStream); такая контравариантность тоже разрешена.

Обратите внимание, что ковариантность и контравариантность поддерживаются только для ссылочных типов, но не для значимых типов или значения void.

Обратный вызов нескольких методов (цепочки делегатов) (смотреть картинки)

Цепочкой (chaining) называется коллекция делегатов, дающая возможность вызывать все методы, представленные этими делегатами.

Открытый статический метод Combine класса Delegate добавляет в цепочку делегатов: fbChain = (Feedback) Delegate.Combine(fbChain, fb1);

При выполнении этой строки метод Combine видит, что мы пытаемся объединить значение null с переменной fb1. В итоге он возвращает значение в переменную fb1, а затем заставляет

переменную fbChain сослаться на делегата, на которого уже ссылается переменная fb1.

Чтобы упростить задачу разработчиков, компилятор C# автоматически предоставляет перегруженные версии операторов += и -= для экземпляров делегатов. Эти операторы вызывают методы

Delegate.Combine и Delegate.Remove соответственно.

Дополнительные средства управления цепочками делегатов

К примеру, сохраняется только последнее из значений, возвращаемых методами обратного вызова. Получить все остальные значения нельзя. И это не единственное ограничение.

Скажем, в ситуации, когда один из делегатов в цепочке становится причиной исключения или блокируется на очень долгое время, выполнение цепочки останавливается.

В качестве альтернативы можно воспользоваться экземплярным методом GetInvocationList класса MulticastDelegate. Этот метод позволяет в явном виде

вызвать любой из делегатов в цепочке:

Метод GetInvocationList работает с объектами классов, производных от MulticastDelegate. Он возвращает массив ссылок, каждая из которых указывает

на какой-то делегат в цепочке. По сути, этот метод создает массив и инициализирует его элементы ссылками на соответствующие делегаты; в конце возвращается

ссылка на этот массив. Если поле \_invocationList содержит null, возвращаемый массив будет содержать всего один элемент, ссылающийся на единственного делегата в цепочке — экземпляр

самого делегата.

Упрощенный синтаксис работы с делегатами

Упрощение 1: не создаем объект делегата

internal sealed class AClass {

public static void CallbackWithoutNewingADelegateObject() {

ThreadPool.QueueUserWorkItem(SomeAsyncTask, 5);

}

private static void SomeAsyncTask(Object o) {

Console.WriteLine(o);

}

}

Статический метод QueueUserWorkItem класса ThreadPool ожидает ссылку на делегата WaitCallback, который, в свою очередь, ссылается на метод SomeAsyncTask.

Так как компилятор в состоянии догадаться, что именно имеется в виду, можно опустить строки, относящиеся к созданию делегата WaitCallback, чтобы упростить

чтение и понимание кода. В процессе компиляции IL-код, генерирующий нового делегата WaitCallback, создается автоматически, а запись является всего лишь

упрощенной формой синтаксиса

Упрощение 2: не определяем метод обратного вызова

В приведенном фрагменте кода метод обратного вызова SomeAsyncTask передается методу QueueUserWorkItem класса ThreadPool. C# позволяет подставить реализацию

метода обратного вызова непосредственно в код, а не в отдельный метод. Скажем, наш код можно записать так:

internal sealed class AClass {

public static void CallbackWithoutNewingADelegateObject() {

ThreadPool.QueueUserWorkItem( obj => Console.WriteLine(obj ), 5);

}

}

Формально в C# он называется лямбда-выражением (lambda expression) и распознается по наличию оператора =>. Лямбда-выражения используются в тех местах, где компилятор

ожидает присутствия делегата. Обнаружив лямбда-выражение, компилятор автоматически определяет в классе новый закрытый метод (в нашем примере — AClass). Этот метод называется

анонимной функцией (anonymous function), так как вы обычно не знаете его имени, которое автоматически создается компилятором.

Упрощение 3: не создаем обертку для локальных переменных для передачи их методу обратного вызова

Вы уже видели, что код обратного вызова может ссылаться на другие члены класса. Но иногда бывает нужно обратиться из этого кода к локальному параметру или переменной внутри

определяемого метода

Обобщенные делегаты

В .NET Framework имеются 17 делегатов Action, от не имеющих аргументов вообще до имеющих 16 аргументов.

Кроме делегатов Action в .NET Framework имеется 17 делегатов Func, которые позволяют методу обратного вызова вернуть значение.

Однако, если нужно передать аргумент по ссылке, используя ключевые слова ref или out, может потребоваться определение собственного делегата:

delegate void Bar(ref Int32 z);

Аналогично нужно действовать в ситуациях, когда требуется передать делегату переменное число параметров при помощи ключевого слова params, если вы хотите задать значения

по умолчанию для аргументов делегата или если потребуется установить ограничение для аргумента-типа. При работе с делегатами, использующими обобщенные аргументы и возвращающими

значения, не следует забывать про ковариантность и контравариантность, так как это расширяет область применения делегатов.

Настраиваемые атрибуты

Настраиваемые атрибуты представляют собой лишь средство передачи некой дополнительной информации. Компилятор помещает эту информацию в метаданные управляемого модуля.

Компилятор обнаруживает их в исходном коде и создает для них соответствующие метаданные

В C# имена настраиваемых атрибутов помещаются в квадратные скобки непосредственно перед именем класса, объекта и т. п.

Атрибуты можно применять только к исходному коду, определяющему такие элементы, как сборки, модули, типы (класс, структура, перечисление, интерфейс, делегат), поля, методы,

конструкторы, параметры методов, возвращаемые значения методов, свойства, события, параметры обобщенного типа.

Настраиваемый атрибут — это всего лишь экземпляр типа, который должен прямо или косвенно наследовать от абстрактного класса System.Attribute.

Атрибуты являются экземплярами класса. И этот класс должен иметь открытый конструктор для создания экземпляров. А значит, синтаксис применения атрибутов аналогичен

вызову конструктора.

Следует заметить, что к одному элементу можно применить несколько атрибутов.

Все неабстрактные атрибуты должны содержать хотя бы один открытый конструктор. Простейший конструктор FlagsAttribute не имеет параметров и не выполняет никаких действий

Null-совместимые значимые типы

Т.к переменная значимого типа не может принимать значение null; ее содержимым всегда является значение соответствующего типа.

Microsoft разработали для CLR null-совместимые значимые типы (nullable value type).

Класс System.Nullable<T> - этот класс реализует значимый тип, который может принимать значение null. Так как Nullable<T> также относится к значимым типам, его экземпляры

достаточно производительны, поскольку экземпляры могут размещаться в стеке, а их размер совпадает с размером исходного типа, к которому приплюсован размер

поля типа Boolean.

В качестве параметра T типа Nullable могут использоваться только структуры — ведь переменные ссылочного типа и так могут принимать значение null.

C# предлагает достаточно удобный синтаксис для работы с такими типами. Переменные x и y можно объявить и инициализировать прямо в коде, воспользовавшись

знаком вопроса: Int32? y = null;

Запись Int32? аналогична записи Nullable<Int32>. При этом вы можете выполнять преобразования, а также приведение null-совместимых экземпляров к другим типам.

Можно применять операторы к экземплярам null-совместимых значимых типов.

Позволяет применять операторы к экземплярам null-совместимых типов.

Следует учесть, что для операций с экземплярами null-совместимых типов генерируется большой объем кода.

В результате компиляции будет создан большой объем IL-кода, вследствие чего операции с null-совместимыми типами выполняются медленнее аналогичных операций с другими типами

Напоследок напомню о возможности определения ваших собственных значимых типов, перегружающих упомянутые операторы.

В C# существует оператор объединения null-совместимых значений (null-coalescing operator). Он обозначается знаками ?? и работает с двумя операндами. Если левый

операнд не равен null, оператор возвращает его значение. В противном случае возвращается значение правого операнда. Оператор объединения null-совместимых значений удобен при

задании предлагаемого по умолчанию значения переменной.

Упаковка null-совместимых значимых типов

Представим переменную типа Nullable<Int32>, которой логически присваивается значение null. Для передачи этой переменной методу, ожидающему ссылки на тип Object, ее следует

упаковать и передать методу ссылку на упакованный тип Nullable<Int32>. Однако при этом в метод будет передано отличное от null значение, несмотря на то что тип Nullable<Int32>

содержит null. Эта проблема решается в CLR при помощи специального кода, который при упаковке null-совместимых типов создает иллюзию их принадлежности к обычным типам.

При упаковке экземпляра Nullable<T> проверяется его равенство null и в случае положительного результата вместо упаковки возвращается null. В противном случае CLR упаковывает

значение экземпляра. Другими словами, тип Nullable<Int32> со значением 5 упаковывается в тип Int32 с аналогичным значением. Следующий код демонстрирует такое поведение:

Распаковка null-совместимых значимых типов

В CLR упакованный значимый тип T распаковывается в T или в Nullable<T>. Если ссылка на упакованный значимый тип равна null и выполняется распаковка в тип Nullable<T>,

CLR присваивает Nullable<T> значение null.

При вызове метода GetType для объекта типа Nullable<T> CLR возвращает тип T вместо Nullable<T>.

Вызов интерфейсных методов через null-совместимый значимый тип

В приведенном далее фрагменте кода переменная n типа Nullable<Int32> приводится к интерфейсному типу IComparable<Int32>. Но тип Nullable<T> в отличие от

типа Int32 не реализует интерфейс IComparable<Int32>. Тем не менее код успешно компилируется, а механизм верификации CLR считает, что код прошел проверку,

чтобы вы могли использовать более удобный синтаксис.

Int32? n = 5;

Int32 result = ((IComparable) n).CompareTo(5); // Компилируется и выполняется

Механика обработки исключений

Класс System.Exception

CLR позволяет генерировать в качестве исключений экземпляры любого типа — от Int32 до String. Но в Microsoft решили, что не стоит заставлять все языки генерировать и

перехватывать исключения произвольного типа. Соответственно, был создан тип System.Exception.

В основе обработки исключений в .NET Framework лежит структурная обработка исключений (Structured Exception Handling, SEH) Windows

private void SomeMethod() {

try { // Код, требующий корректного восстановления или очистки ресурсов}

catch (InvalidOperationException) { // Код восстановления работоспособности после исключения InvalidOperationException}

catch (IOException) { // Код восстановления работоспособностипосле исключения IOException}

catch { // Код восстановления работоспособности после остальных исключений. После перехвата исключений их обычно генерируют повторно

throw;

}

finally { // Здесь находится код, выполняющий очистку ресурсов после операций, начатых в блоке try. Этот код выполняется ВСЕГДА вне зависимости от наличия исключения }

// Код, следующий за блоком finally, выполняется, если в блоке try не генерировалось исключение или если исключение было перехвачено блоком catch, а новое не генерировалось

}

Блок try

В блок try помещается код, требующий очистки ресурсов и/или восстановления после исключения, а сам код очистки содержится в блоке finally.

В блоке try может располагаться также код, приводящий к генерации исключения. Код же восстановления вставляют в один или несколько блоков catch.

Один блок catch соответствует одному событию, после которого по вашим предположениям может потребоваться восстановление приложения.

Блок try должен быть связан хотя бы с одним блоком catch или finally; сам по себе он не имеет смысла, и C# запрещает такие определения.

Блок catch

В блок catch помещают код, который должен выполняться в ответ на исключение.

Если код в блоке try не порождает исключение, CLR никогда не переходит к выполнению кода в соответствующем блоке catch, а сразу переходя к коду блока finally, если таковой

существует. Выполнив код блока finally, поток переходит к инструкции, следующей за этим блоком.

Выражение в скобках после ключевого слова catch называется типом исключения (catch type).

Поиск подходящего блока catch в CLR осуществляется сверху вниз, поэтому наиболее конкретные обработчики должны находиться в начале списка. Сначала следуют потомки с наибольшей

глубиной наследования, потом — их базовые классы (если таковые имеются) и, наконец, — класс System.Exception (или блок с неуказанным типом исключений).

Исключение, сгенерированное при выполнении кода блока try (или любого вызванного этим блоком метода), инициирует поиск блоков catch соответствующего типа. При отсутствии

совпадений CLR продолжает просматривать стек вызовов в поисках типа исключения, соответствующего данному исключению. Если при достижении вершины стека блок catch нужного типа

обнаружен не будет, исключение считается необработанным.

При обнаружении блока catch нужного типа CLR исполняет все внутренние блоки finally, начиная со связанного с блоком try, в котором было вброшено исключение,

и заканчивая блоком catch нужного типа. При этом ни один блок finally не выполняется до завершения действий с блоком catch, обрабатывающим исключение.

После того как код внутренних блоков finally будет выполнен, исполняется код из обрабатывающего блока catch. Здесь выбирается способ восстановления после исключения. Затем можно

выбрать один из трех вариантов действий:

еще раз сгенерировать то же исключение для передачи информации о нем коду, расположенному выше в стеке;

сгенерировать исключение другого типа для передачи дополнительной информации коду, расположенному выше в стеке;

позволить программному потоку выйти из блока catch естественным образом.

В последнем же случае происходит переход к блоку finally (если он, конечно, существует). После выполнения всего содержащегося в нем кода управление переходит к расположенной

после блока finally инструкции. Если блок finally отсутствует, поток переходит к инструкции, расположенной за последним блоком catch. В C# после типа перехватываемого исключения

можно указать имя переменной, которая будет ссылаться на сгенерированный объект, потомок класса System.Exception. В коде блока catch эту переменную можно использовать для получения

информации об исключении (например, данных трассировки стека, приведшей к исключению).

Блок finally

Код блока finally выполняется всегда.

Обычно этот код производит очистку после выполнения блока try. Если в блоке try был открыт некий файл, блок finally должен содержать закрывающий этот файл код

Если код блока try выполняется без исключений, файл закрывается. Впрочем, поскольку даже исключение не помешает выполнению кода в блоке finally, файл

гарантированно будет закрыт. А если поместить инструкцию закрытия файла после блока finally, в случае неперехваченного исключения файл останется открытым

(до следующего прохода уборщика мусора).

Блок try может существовать и без блока finally, ведь иногда его код просто не требует последующей очистки. Однако если вы решили создать блок finally,

его следует поместить после всех блоков catch.

И помните, что одному try может соответствовать только один блок finally.

Генерирование исключений

Во-первых, следует понять, к какому производному от типа Exception типу будет относиться ваше исключение.

Подумайте о том, каким образом код, расположенный выше по стеку вызовов, сможет получать информацию о неудачной работе метода, чтобы выполнить восстановительные операции.

Можно воспользоваться для этой цели одним из типов, определенных в FCL, но может оказаться и так, что там пока отсутствует подходящий тип. В таком случае вам потребуется

определить собственный тип, производный от класса System.Exception.

Если вы собираетесь создать иерархию исключений, постарайтесь, чтобы она содержала как можно меньше базовых классов. Дело в том, что базовые классы зачастую обрабатывают

несколько ошибок по одним правилам, а это может быть опасно. Соответственно, никогда не следует создавать объекты System.Exception и всегда нужно соблюдать максимальную

осторожность при генерировании исключений базовых классов.

Во-вторых, следует решить, какое строковое сообщение должно быть передано конструктору исключения. Генерирование исключения должно сопровождаться подробной информацией о том,

почему метод не смог решить свою задачу. При обработке перехваченного исключения это сообщение остается невидимым. С другой стороны, если исключение останется необработанным,

оно с большой вероятностью будет зарегистрировано в журнале. Необработанное исключение свидетельствует о наличии в приложении дефекта, об искоренении которого должен позаботиться

разработчик. Конечные пользователи не имеют доступа к исходному коду и не могут перекомпилировать программу. Соответственно, не видят они и данного сообщения, поэтому туда можно

включать всю техническую информацию, необходимую для устранения дефекта

Создание классов исключений

Активно используйте блоки finally

По-моему, блоки finally — прекрасное средство! Они позволяют определять код, который будет гарантированно исполнен независимо от вида сгенерированного потоком исключения.

Блоки finally нужны, чтобы выполнить очистку после любой успешно начатой операции, прежде чем вернуть управление или продолжить исполнение кода, расположенного после них.

Блоки finally также часто используются для явного уничтожения любых объектов с целью предотвращения утечки ресурсов.

Гарантированное исполнение кода очистки при любых обстоятельствах настолько важно, что большинство языков поддерживает соответствующие программные конструкции.

При использовании инструкций lock, using и foreach блоки try/finally создаются автоматически. Компилятор строит эти блоки и при переопределении деструктора класса метод Finalize.

При работе с упомянутыми конструкциями написанный вами код помещается в блок try, а код очистки — в блок finally. А именно:

если вы используете инструкцию lock, то внутри блока finally снимается блокировка;

если вы используете инструкцию using, то внутри блока finally для объекта вызывается метод Dispose;

если вы используете инструкцию foreach, то внутри блока finally для объекта IEnumerator вызывается метод Dispose;

если вы определяете деструктор, то внутри блока finally вызывается метод Finalize базового класса

Не надо перехватывать все исключения

Распространенная ошибка — слишком частое и неуместное использование блоков catch. Перехватывая исключение, вы тем самым заявляете, что ожидали его, понимаете его причины и знаете,

как с ним разобраться. Другими словами, вы определяете политику для приложения. Эта тема подробно раскрыта в разделе «Продуктивность вместо надежности

catch (Exception) {

...

}

Этот код демонстрирует, что в нем предусмотрены все исключения любых типов и он способен восстанавливаться после любых исключений в любых ситуациях.

Тип из библиотеки классов ни в коем случае не должен перехватывать все исключения подряд: ведь он не может знать наверняка, как приложение должно реагировать на исключения.

Такой тип будет часто вызывать код приложения через делегата, виртуальный метод или интерфейсный метод. Если в одной части приложения возникает исключение, то в другой части,

вероятно, есть код, способный перехватить его. Исключение должно пройти через фильтр перехвата и быть передано вверх по стеку вызовов, чтобы код приложения смог обработать его как

надо. Если исключение осталось необработанным, CLR завершает процесс. Большинство из них обнаруживаются на стадии тестирования. Для борьбы с ними следует либо заставить код

реагировать на определенное исключение, либо переписать его, устранив условия, ставшие причиной сбоя. Число необработанных исключений в окончательной версии программы,

предназначенной для выполнения в производственной среде, должно быть минимальным, а сама программа должна быть исключительно устойчивой

Кстати, вполне допустимо перехватить исключение System.Exception и выполнить определенный код внутри блока catch при условии, что в конце этого кода исключение будет

сгенерировано снова. Перехват и поглощение (без повторного генерирования) исключения System.Exception недопустимо, так как оно приводит к сокрытию факта сбоя и продолжению работы

приложения с непредсказуемыми результатами, что означает нарушение безопасности.

Наконец, допускается перехватить исключение, возникшее в одном потоке, и повторно сгенерировать его в другом потоке. Такое поведение поддерживает модель асинхронного

программирования (см. главу 28). Например, если поток из пула потоков выполняет код, который вызывал исключение, CLR перехватывает и игнорирует исключение, позволяя потоку

вернуться в пул. Позже один из потоков должен вызвать метод EndXxx, чтобы выяснить результат асинхронной операции. Метод EndXxx сгенерирует такое же исключение, что и поток из пула,

выполнявшего заданную работу. В данной ситуации исключение поглощается первым потоком, но повторно генерируется потоком, вызывавшим метод EndXxx, в результате ошибка

не оказывается скрытой от приложения

Корректное восстановление после исключения

Отмена незавершенных операций при невосстановимых исключениях

Обычно для выполнения единственной абстрактной операции методу приходится вызывать несколько других методов, одни из которых могут завершаться успешно,

а другие — нет. Допустим, происходит сериализация набора объектов в файл. После сериализации 10 объектов генерируется исключение (например, из-за переполнения

диска или из-за отсутствия атрибута Serializable у следующего сериализуемого объекта). После этого исключение фильтруется и передается вызывающему методу,

но в каком состоянии остается файл? Он оказывается поврежденным, так как в нем находится частично сериализованный граф объектов. Было бы здорово, если бы

приложение могло отменить незавершенные операции и вернуть файл в состояние, в котором он был до записи сериализованных объектов. Правильная реализация

должна выглядеть примерно так:

Для корректной отмены незавершенных операций код должен перехватывать все исключения. Да, здесь нужно перехватывать все исключения, так как важен не тип

ошибки, а возвращение структур данных в согласованное состояние. Перехватив и обработав исключение, не поглощайте его — вызывающему коду необходимо сообщить о ситуации.

Это делается путем повторного генерирования того же исключения. В С# и многих других языках это осуществляется просто: просто укажите ключевое слово throw без продолжения,

как в предыдущем фрагменте кода. Обратите внимание, что в предыдущем примере не указан тип исключения в блоке catch, поскольку здесь требуется перехватывать абсолютно

все исключения. К счастью, в C# достаточно опустить тип исключения, и инструкция throw повторно сгенерирует это исключение.

Сокрытие деталей реализации для сохранения контракта

Необработанные исключения

Итак, при появлении исключения CLR начинает в стеке вызовов поиск блока catch, тип которого соответствует типу исключения. Если ни один из блоков catch

не отвечает типу исключения, возникает необработанное исключение (unhandled exception). Обнаружив в процессе поток с необработанным исключением, CLR немедленно уничтожает

этот поток. Необработанное исключение указывает на ситуацию, которую не предвидел программист, и должно считаться признаком серьезной ошибки в приложении. На этом этапе

о проблеме следует уведомить компанию, где разработано приложение, чтобы авторы могли устранить неполадку и выпустить исправленную версию

Автоматическое управление памятью (уборка мусора)

Хостинг CLR и домены приложений

Глава 23. Загрузка сборок и отражение

Глава 24. Сериализация

Глава 25. Взаимодействие с компонентами WinRT

Потоки

На заре компьютерной эры операционные системы не поддерживали концепцию потоков. Соответственно любые задачи которые выполнялись длительное время приостанавливали

выполнение других задач, либо приводили к ошибки и проходилось перезагружать систему. Поэтому для решения этих проблем были придуманы потоки и процессы. Каждому процессу выделяется

собственный поток исполнения, который работает как виртуальный процессор. Если код приложения войдет в бесконечный цикл, то блокируется только связанный с этим кодом процесс,

а остальные процессы продолжают функционировать.

Процесс - набор ресурсов, используемый отдельным экземпляром приложения и имеет собственно виртуальное адресное пространство, что гарантирует, что код и данные одного процесса

будут изолированы для другого

Различие между процессом и потоком:

Поток и процесс служать для разделения работы.

Поток стоит выбирать для легковесных операций, в то время как процесс для тяжелых операций.

Каждый процесс имеет собственное виртуальное пространство с набором ресурсов, а инициализированный поток использует эти ресурсы.

Обменн даных между потоками в рамках одного процесса быстрее, чем у отдельных процессов.

Переключения контекста (context switching) - механизм который позволяет распределять физический процессор между всеми своими потоками путем переключения контекста на другой поток.

После переключения контекста процессор исполняет выбранный поток, пока не истечет выделенное потоку время, после этого снова происходит переключение контекста.

Переключение контекста повышает надежность системы и скорость реагирования на действия конечных пользователей.

Если поток какого-то приложения зацикливается, то Windows его периодически выгружает и передает процессору другой поток для исполнения.

Процесс в результате прекращает свою работу, теряя несохраненные данные, но остальные процессы в системе продолжают функционировать, как ни в чем не бывало.

Переключение контекстов приводит к повышению отказоустойчивости, хотя за это приходится платить снижением производительности.

Приоритет потоков

Лучше снизить приоритет одного потока, чем повысить приоритет другого. Повышать приоритет имеет смысл, если поток должен быстро отреагировать на какое-то событие,

запуститься на короткий промежуток времени и вернуться в состояние ожидания.

Фоновые и активные потоки

В CLR все потоки делятся на активные (foreground) и фоновые (background).

При завершении активных потоков в процессе CLR принудительно завершает также все запущенные на этот момент фоновые потоки.

Следовательно, активные потоки имеет смысл использовать для исполнения заданий, которые обязательно требуется завершить. Фоновые же потоки можно оставить для некритичных задач,

как пересчет ячеек электронных таблиц или индексирование записей.

Поток можно превращать из активного в фоновый и обратно. Основной поток приложения и все потоки, в явном виде созданные путем конструирования объекта

Thread, по умолчанию являются активными. А вот потоки из пула по умолчанию являются фоновыми.

Способы создания потоков.

Первый способ используя явное создание потока при помощи Thread, то в результате будет создан объект потока в коде который можно будет запустить. Для создания которого следует

передать конструктору имя метода, а для запуска потока нужно вызвать метода Start.

Сигнатура метода должна совпадать с сигнатурой делегата delegate void ParameterizedThreadStart(Object obj);

Потоки, созданные путем конструирования объекта Thread, по умолчанию являются активными. А вот потоки из пула по умолчанию являются фоновыми.

Недостатки:

Для каждого отдельного задания необходимо создавать новый поток и нельзя переопределить в объекте потока задание, что приводит к проблеме производительности.

Второй способ использовать пул потоков.

Среда CLR способна управлять собственным пулом потоков. Если в один процесс загружаются несколько экземпляров CLR, для каждого из них формируется собственный пул.

При инициализации CLR пул потоков пуст. Т.к пул потоков поддерживает очередь выполнения асинхронных операций. То при появлении запроса он помещается в очередь пула

потоков. Далее код пула извлекает эти записи из очереди и распределяет их среди потоков из пула. Если пул пуст, то создается новый поток, а при завершении исполнения своего

задания поток не уничтожается, а возвращается в пул и ожидает следующего запроса. Поскольку поток не уничтожается, производительность не страдает.

Если приложение отправляет пулу много запросов, то он пытается обслужить их все с помощью одного потока, но если приложение создает очередь запросов быстрее, чем поток

из пула их обслуживает, то создаются дополнительные потоки. Если приложение прекращает отправлять запросы в пул, появляются незанятые потоки, впустую занимающие память, то через

некоторое время бездействия поток пробуждается и самоуничтожается, освобождая ресурсы.

Для добавления в очередь пула потоков асинхронных вычислительных операций обычно вызывают один из следующих методов класса ThreadPool:

ThreadPool.QueueUserWorkItem(new WaitCallback(SomeMethod));

Создаваемый метод обратного вызова должен соответствовать делегату System.Threading.WaitCallback, который определяется так:

delegate void WaitCallback(Object state);

Преимуществ: нет необходимости создавать новый поток на каждое задание, что повышает производительность приложения.

Самой большой проблемой является отсутствие встроенного механизма, позволяющего узнать о завершении операции и получить возвращаемое значение.

Третий способ - задания Task.

заданий (tasks), выполнение которых осуществляется посредством типов из пространства имен System.Threading.Tasks.

В качестве параметра объект Task принимает делегат Action, например, лямбда-выражение или ссылку на какой-либо метод.

1. Task task = new Task(() => Console.WriteLine("Hello Task!"));

task.Start(); // Метод Start для запускает задания.

2. Task task = Task.Run(() => Console.WriteLine("Hello Task!"));

3. Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000000000);

4. существует способ узнать о завершении задания. Оно может просто инициировать выполнение следующего задания, чтобы избежать блокировки потоков:

Task<Int32> t = Task.Run(() => Sum(CancellationToken.None, 10000));

Task cwt = t.ContinueWith(task => Console.WriteLine("The sum is: " + task.Result));

5. Дочерние задания или массив задач

Task<Int32[]> parent = new Task<Int32[]>(() => {много тасков}

6. Фабрика заданий TaskFactory используется для создания группы заданий, а TaskFactory<TResult> если эти задания должны возвращать некое значение.

7. Планировщики заданий TaskScheduler. Экземпляр TaskScheduler класса представляет планировщик задач и гарантирует, что работа над задачей в итоге будет выполнена.

8. Класс Parallel и методы For, ForEach и Invoke, класс позволяет распределить работу между несколькими потоками из пула.

Все методы класса Parallel заставляют вызывающий поток принимать участие в их обработке, вызывающий поток не блокируется, ожидая выполнения работы потоками пула. Выполнение

вызывающего потока не возобновляется, покане будет завершена вся работа. Если какая-либо операция станет источником необработанного исключения, вызванный вами метод Parallel

выдаст исключениеAggregateException

Результат работы цикла можно определить при помощи свойств. Если свойство IsCompleted возвращает значение true, значит, цикл пройден полностью, и все элементы обработаны.

9. Асинхронные функции C#

Когда метод помечается ключевым словом async, компилятор преобразует код метода в тип, реализующий конечный автомат. Это позволяет потоку выполнить часть кода в конечном

автомате, а затем вернуть управление без выполнения всего метода до завершения. Также стоит отметить, что слово async, которое указывается в определении метода, НЕ делает

автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений await.

Ключевое слово await позволяет обойтись без блокировки для запуска задачи, а затем продолжить выполнение, когда задача завершается.

Оператор await заставляет компилятор сгенерировать код, который запрашивает свойство Result объекта Task<TResult> и присваивает результат локальной переменной bytesRead, или

выдает исключение в случае ошибки, а затем выполняется оставшаяся часть кода.

Асинхронная функция не может иметь параметры out и ref и in.

Оператор await не может использоваться в блоке catch, finally или unsafe.

t.Wait() - метод позволяет дождаться завершения задания и после этого получить результат его выполнения.

t.Result - получаем результат выполнения, возвращаемого вычислительной операцией.

Метод WaitAny позволяет ожидать завершения не только одного задания, но и массива объектов Task. Метод возвращает значение true после завершения всех объектов и

значение false, если истекает время ожидания, а так же блокирует вызывающий поток до завершения всех объектов Task в массиве.

t.ContinueWith - метод,который, создает продолжение, которое выполняется асинхронно после завершения выполнения целевой задачи Task. Во внутренней реализации объект Task

содержит коллекцию ContinueWith. Это дает возможность несколько раз вызвать метод ContinueWith при помощи единственного объекта Task. Когда это задание завершится, все

задания из коллекции ContinueWith окажутся в очереди в пуле потоков.

При вызове потоком метода Wait система проверяет, началось ли выполнение задания Task, которого ожидает поток. В случае положительного результата проверки поток,

вызывающий метод Wait, блокируется до завершения задания. Но если задание еще не начало выполняться, система может выполнить его при помощи потока, вызывающего метод Wait.

В этом случае данный поток не блокируется. Он выполняет задание Task и немедленно возвращает управление. Это снижает затраты ресурсов, повышает производительность. Однако и это

может быть не очень хорошо. Например, если перед вызовом метода Wait в рамках синхронизации потока происходит его блокирование, а затем задание пытается получить доступ к тем же

запертым ресурсам, возникает взаимная блокировка (deadlock)!

Если вычислительное задание генерирует необработанное исключение, оно поглощается и сохраняется в коллекции, а потоку пула разрешается вернуться в пул.

Затем при вызове метода Wait или свойства Result эти члены вбросят исключение System.AggregateException.

Для написания масштабируемого программного обеспечения следует избегать блокировки потоков. Вызов метода Wait или запрос свойства Result при незавершенном задании приведет,

скорее всего, к появлению в пуле нового потока, что увеличит расход ресурсов и отрицательно скажется на расширяемости.

Виды синхронизации потоков при многопоточном программировании

lock

Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. Остальный потоки

помещаются в очередь ожидания и ждут, пока текущий поток не освободит данный блок кода.

Для блокировки с ключевым словом lock используется объект-заглушка.

Когда выполнение доходит до оператора lock, объект блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, а после окончания работы

блока кода, объект освобождается и становится доступным для других потоков.

Не используйте один и тот же экземпляр объекта блокировки для разных общих ресурсов: это может привести к взаимоблокировке или состязанию при блокировке.

Удерживайте блокировку в течение максимально короткого времени, чтобы сократить число конфликтов при блокировке.

lock это упращенный синтаксис класса Monitor.

Monitor

Метод Monitor.Enter принимает два параметра - объект блокировки и значение типа bool, которое указывает на результат блокировки (если он равен true, то блокировка успешно

выполнена). Фактически этот метод блокирует объект locker так же, как это делает оператор lock. А в блоке try...finally с помощью метода Monitor.Exit происходит освобождение

объекта locker, если блокировка осуществлена успешно, и он становится доступным для других потоков. Но есть опасность, что в случаше ошибки следующий поток может работать

с поврежденными данными.

lock и monitor - одно и то же( lock - синтаксический сахар - вызывает методы Monitor.Enter и Monitor.Exit). lock работает на порядки быстрее чем Mutex, но к нему невозможно

получить доступ из другого процесса, а вот к Mutex - можно.

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

Конструкции режима ядра:

Они работают намного медленнее конструкций пользовательского режима, так как требуют координации со стороны операционной системы.

К примитивным конструкциям синхронизации потоков в режиме ядра относятся события (events) и семафоры (semaphores). На их основе строятся более сложные конструкции аналогичного

назначения, например мьютексы

В пространстве имен System.Threading существует абстрактный базовый класс WaitHandle.

Вот как выглядит иерархия этих классов:

WaitHandle

EventWaitHandle

AutoResetEvent

ManualResetEvent

Semaphore

Mutex

Mutex

Как уже упоминалось, классы AutoResetEvent, ManualResetEvent, Semaphore и Mutex являются производными от класса WaitHandle, то есть наследуют методы этого класса и их поведение.

Конструкции режима ядра часто используются для создания приложений, которые в любой момент времени могут существовать только в одном экземпляре.

К критической секции может получить только один поток.

lock и monitor - одно и то же( lock - синтаксический сахар - вызывает методы Monitor.Enter и Monitor.Exit). lock работает на порядки быстрее чем Mutex, но к нему невозможно

получить доступ из другого процесса, а вот к Mutex - можно.

Сначала создаем объект мьютекса:

Mutex mutexObj = new Mutex()

Основную работу по синхронизации выполняют методы WaitOne() и ReleaseMutex().

Метод mutexObj.WaitOne() приостанавливает выполнение потока до тех пор, пока не будет получен мьютекс mutexObj. Изначально мьютекс свободен, поэтому его получает один из потоков.

После выполнения всех действий, когда мьютекс больше не нужен, поток освобождает его с помощью метода mutexObj.ReleaseMutex(). А мьютекс получает один из ожидающих потоков.

Таким образом, когда выполнение дойдет до вызова mutexObj.WaitOne(), поток будет ожидать, пока не освободится мьютекс. И после его получения продолжит выполнять свою работу.

Если два или более потоков должны получить доступ к общему ресурсу одновременно, системе требуется механизм синхронизации, чтобы обеспечить использование ресурса только одним

потоком. Mutex — это примитив синхронизации, предоставляющий монопольный доступ к общему ресурсу только одному потоку. Если поток получает мьютекс, второй поток, который хочет

получить этот мьютекс, приостанавливается до тех пор, пока первый поток не выпустит мьютекс.

Класс Mutex применяет удостоверение потока, поэтому мьютекс может быть освобожден только потоком, который его приобрел. Напротив, Semaphore класс не применяет удостоверение потока.

Мьютекс также можно передавать через границы домена приложения.

Semaphore

Данный класс работает подобно Mutex, но позволяет определить несолько потоков, чтобы получить доступ к критической секции, а в конструкторе указывается колличестов потоков.

Количество семафоров уменьшается каждый раз, когда поток входит в семафор, и увеличивается, когда поток освобождает семафор. Если счетчик равен нулю, последующие запросы

блокируются до тех пор, пока другие потоки не отпустит семафор.

Семафоры бывают двух типов: локальные семафоры и именованные системные семафоры.

Если объект создается Semaphore с помощью конструктора, принимающего имя, он связывается с семафором операционной системы с таким именем. Именованные системные семафоры видны

всей операционной системе и могут использоваться для синхронизации действий процессов. Можно создать несколько Semaphore объектов, представляющих один и тот же именованный

системный семафор, и использовать OpenExisting метод для открытия существующего именованного системного семафора.

Локальный семафор существует только внутри процесса. Его может использовать любой поток в вашем процессе, имеющий ссылку на локальный объект Semaphore. Каждый Semaphore

объект является отдельным локальным семафором.

Для работы с потоками класс Semaphore имеет два основных метода:

WaitOne(): ожидает получения свободного места в семафоре

Release(): освобождает место в семафоре

Семафоры (semaphores) также представляют собой обычные переменные типа Int32, управляемые ядром. Ожидающий семафора поток блокируется при значении 0 и освобождается при значениях

больше 0. При снятии блокировки с ожидающего семафора потока ядро автоматически вычитает единицу из счетчика. С семафорами связано максимальное значение типа Int32, которое ни при

каких обстоятельствах не могут превысить текущие показания счетчика иначе будет вызвано исключение SemaphoreFullException.

События

События (events) представляют собой переменные типа Boolean, находящиеся под управлением ядра. Ожидающий события поток блокируется, если оно имеет значение false, и освобождается в

случае значения true.

AutoResetEvent

Этот класс представляет событие синхронизации потоков, который позволяет при получении сигнала переключить данный объект-событие из сигнального в несигнальное состояние.

Для этого создаем переменную типа AutoResetEvent. Передавая в конструктор значение true, указывая, что создаваемый объект изначально будет в сигнальном состоянии.

Когда начинает работать поток, то первым делом срабатывает вызов waitHandler.WaitOne(). Метод WaitOne указывает, что текущий поток переводится в состояние ожидания,

пока объект waitHandler не будет переведен в сигнальное состояние. И так все потоки у нас переводятся в состояние ожидания.

После завершения работы вызывается метод waitHandler.Set, который уведомляет все ожидающие потоки, что объект waitHandler снова находится в сигнальном состоянии, и

один из потоков "захватывает" данный объект, переводит в несигнальное состояние и выполняет свой код. А остальные потоки снова ожидают.

Если у нас в программе используются несколько объектов AutoResetEvent, то мы можем использовать для отслеживания состояния этих объектов статические методы WaitAll и WaitAny, \

которые в качестве параметра принимают массив объектов класса WaitHandle - базового класса для AutoResetEvent.

waitHandler может быть в сигнальном состоянии и не сигнальном.

Потоки стоят в очереди за ним.

Получить его(увидеть) они могут как только он стал подавать сигнал(сигнальное состояние).

Итак, что происходит при выполнении программы:

1.AutoResetEvent waitHandler = new AutoResetEvent(true) - подал сигнал

2. Первый поток в очереди к нему:

3. waitHandler.WaitOne(); - - поймал сигнал, погасил его - погасил текущий поток - главный(ввёл поток в режим ожидания), выполняет наш второстепенный поток;

4. Выполнился блок кода

5. waitHandler.Set(); - отдал-включил сигнал. - включил главный поток(вывел главный поток из режима ожидания)

6. Второй поток в очереди: пункты 3,4,5 - поймал сигнал, погасил, выполнился, отдал-включил

7. Третий поток в очереди: пункты 3,4,5 - поймал сигнал, погасил, выполнился, отдал-включил

8. И так далее

Deadlock, livelock, starvation.

Deadlock

Взаимоблокировкой (deadlock) называется ситуация, когда как минимум два потока останавливаются и ожидают друг от друга снятия блокировки. Поскольку оба потока

ожидают друг от друга выполнения соответствующего действия, получается, что они блокируют друг друга, из-за чего их ожидание может длиться бесконечно.

livelock

Если поток, использующий в данный момент конструкцию, не освободит ее, ожидающий конструкции поток может оказаться заблокированным навсегда. В этом случае в

пользовательском режиме поток бесконечно исполняется процессором; этот вариант блокировки называется активной (живой) блокировкой (livelock) или зависанием.

starvation

Во время включения потока Б эксклюзывный ресурс всегда занят другим потоком, из-за чего поток Б прекращает свою работу не может выполнить свои опперации.

Классы коллекций для параллельного доступа

В FCL существует четыре безопасных в отношении потоков класса коллекций, принадлежащих пространству имен System.Collections.Concurrent:

ConcurrentQueue, ConcurrentStack, ConcurrentDictionary и ConcurrentBag

Атрибут [ThreadStatick]