|  |  |  |  |
| --- | --- | --- | --- |
| **.NET Framework**   * NET Framework — это управляемая среда для разработки и исполнения приложений * Состав:   + Общеязыковая среда исполнения CLR (Common Language Runtime)     - Управление ходом выполнения программы     - Контроль типов     - Выделение и освобождение памяти под данные и команды   + Библиотека классов FCL (Framework Class Library)   + Средства разработки (компиляторы и т.п.)   **Общеязыковая среда исполнения CLR. Некоторые особенности.**   * CIL (Common Intermediate Language) – общий промежуточный язык (псевдомашинный язык, ассемблер для виртуальной машины) * JIT(just-in-time)-компиляция для повышения быстродействия * Домены приложений * Автоматическая сборка мусора (реализовано на уровне виртуальной машины) * Возможность (крайне нежелательная) вызова неуправляемого кода   **Языки программирования .NET** Framework   * Microsoft-компиляторы   + C#   + VB.NET   + J# (в новых версиях отсутствует)   + JScript .NET   + Managed C++ * Специфичность платформы накладывает ограничения на язык программирования (VB.NET не совместим со старыми версиями VB):   + объектность   + контроль типов   + классы только из FCL (например, строки) * Сторонние компиляторы:   + Delphi   + PascalABC.NET   + IronPython   + IronRuby   + Nemerle   + и др.   **Управляемые модули (managed modules)**   * Расширение **\*.netmodule** (иногда сборки \*.exe или \*.dll, содержащие один управляесый модуль, также называют управляемым модулем) * Состав:   + Заголовок файла Windows Portable Executable (PE)   + Заголовок CLR, содержит информацию о местоположении CIL и метаданных   + Метаданные, описывают все, что находится внутри модуля и внешние зависимости   + Команды CIL (скомпилированные классы) * Метаданные обязательны, с помощью них модуль описывает сам себя * Доступ к метаданным может быть получен программно с помощью *отражения* (классов FCL из пространства имен System.Reflection)   **Сборки**  Декларация сборки:   * Имя сборки * Список всех файлов с хэш-значениями, вычисляемыми по содержимому файлов * Список экспортируемых типов данных и информация о файлах, где они определены * Номер версии в формате major.minor.build.version * Название поставщика сборки * Описание * Строка региональных стандартов и т.д.   **Строго- и нестрого-именованные сборки**   * *Нестрого именованные (weakly named)* – для идентификации используется только имя файла сборки без расширения * *Строго именнованные (strongly named)*, содержат открытый ключ своего создателя и цифровую подпись, сгенерированную для декларации сборки (ключ хранится в декларации). Т.о. сборка становится устойчивой к подделкам.   + борьба с «адом DLL»   + для создания используется утилита AL (al.exe) * Глобальный кэш сборок   **Типичные шаблоны применения try-catch-finally**   * Перехват с помощью **catch** специфических исключений, которые известно, как обработать (FileNotFoundException) * Гарантированное приведение состояния приложения к согласованному виду в блоке **finally** * Проглатывание ошибок в прикладном коде, который является “дополнительным” и существенно не влияет на работу приложения, в блоке **catch** (например, если во время чтения конфигурации в приложении возникает исключение, то можно использовать конфигурацией по-умолчанию; такое исключение можно подавить, предварительно проинформировав о нем пользователя или записав в лог ошибок)   **Собственные исключения**   * Программист может создавать свои собственные исключения, унаследовав их от класса **Exception** (в случае прикладного кода следует наследовать исключения от **ApplicationException**);   см. проект .\OOP.Lecture1-2.Samples   * Как правило, новые классы исключений не добавляют новой функциональности (существующей, см. свойства Exception, вполне достаточно), а лишь специфицируют некоторый класс ошибок, чтобы их можно было обрабатывать отдельно от других классов и от базового * В случае, когда по смыслу возникающей ошибки она соответствует какому-либо системному исключению, возможно, имеет смысл воспользоваться системным исключением вместо создания нового класса исключений | **Типы данных C#**   * *Ссылочные типы (reference types)*   + Классы (class)     - object     - string     - Массивы   + Интерфейсы (interface)   + Делегаты * *Типы-значения (value types)*   + Встроенные типы (за исключением object и string)   + Структуры (struct)   + Перечисления (enum) * Указатели (в unsafe-контексте) * Обобщенные типы * Делегаты (функциональный тип)   **Выделение памяти для ссылочных и структурных типов**   * Для ссылочных типов память под экземпляр выделяется в куче * Для структурных типов память под экземпляр выделяется в стеке (или в экземпляре класса)   **Упаковка и распаковка**   * И ссылочные и структурные типы являются производными от корневого базового класса **object** (**System.Object**). В тех ситуациях, когда структурный тип должен действовать как **object**, в куче создается временный объект, имитирующий поведение ссылочного объекта. В созданный объект копируются данные из структурного объекта. Этот процесс называется ***упаковкой (boxing)***, а обратный процесс называется ***распаковкой (unboxing)***. Механизм упаковки обеспечивает возможность интерпретировать любой тип как **object**:   **Структуры. Особенности**   * Структуры обычно используются для описания некоторых логически связанных переменных (Point, Rectangle и т.п.); можно провести аналогию между структурами в C# и записями в Pascal * Как правило, поля структур описываются c видимостью public (соответственно для полей не описываются аналогичные им свойства) * **Структуры являются типами-значениями**   + Все структуры наследуются **System.ValueType** (который наследуется от System.Object)   + Под экземпляр структуры автоматически выделяется память в стеке (структуры не создаются и не уничтожаются сборщиком мусора), оператор new можно не использовать   + При копировании структуры (оператор =) копируются данные структуры, а не ссылка, поэтому изменение полей одной структуры не приведет к изменению поле в другой   + В методы структура передается копированием, т.е. по значению * Так как структуры не требуется создавать в куче и обращаться к ним по ссылке, то быстродействие при работе с ними максимально (при этом надо учитывать, что потери быстродействия могут возникать при частой упаковке/распаковке структурных типов)   **Синтаксис структур**   * Совпадает с синтаксисом классов за исключением   + Вместо ключевого слова class используется **struct**   + Не статические поля структур не могут быть инициализированы при объявлении   + У структур не может быть объявлен конструктор по умолчанию (без параметров) и деструктор   **Перечисления**   * Особый тип для описания набора констант; можно провести аналогию между перечислениями в C# и перечислимыми типами в Pascal * Наследуются от **System.Enum** * Могут базироваться на основе типов byte, sbyte, short, ushort, int, uint, long, ulong; по-умолчанию на **int** * Удобен для описания битовых флагов (помечается атрибутом [Flags]) * Примеры из .NET FCL   + System.ConsoleColor   + System.Drawing.Drawing2D.PenType   + и др.   **Общие рекомендации по обработке исключений**   * В системном (библиотечном) коде некорректно “проглатывать” общие исключения (Exception, SystemException) * В прикладном коде ситуации, когда такой подход допустим, существуют, но крайне редки * Следует перехватывать только те специфические исключения, которые понятно, как можно обработать в текущем контексте (например, в случае FileNotFoundException можно сообщить об отсутствии файла пользователю и предложить указать корректное имя файла) * Не следует слишком увлекаться перехватом исключений, многие из них корректно (в плане дизайна приложения) могут быть переданы на обработку на верхний уровень (в стеке вызовов методов) * Не следует использовать catch-блоки для приведения состояния приложения к согласованному виду (например, закрывать открытые файлы и т.п.) в случае возникновения исключения, для этого предусмотрен finally-блок; в грамотно написанном коде finally, как правило, встречается чаще catch * В случае, если надо передать перехваченное исключение наверх, следует пользоваться пустым throw вместо throw e; во втором случае меняется позиция (номер строки) на отличную от позиции, в которой действительно произошло исключение | **Массивы**   * Многомерные (прямоугольные) * Ступенчатые (массивы массивов)   Во время выполнения массивы — экземпляры **System.Array**  **Коллекции System.Collections.Generics**   * Коллекции — специальные классы, реализующие структуры данных для хранения множества элементов   + Список (IList<T>, List<T>, LinkedList<T>)   + Множество (ISet<T>, HashSet<T>, SortetSet<T>)   + Словарь (IDictionary<TKey, TValue>, Dictionary<TKey, TValue>, SortedDictionary<TKey, TValue>)   + Стек (Stack<T>)   + Очередь (Queue<T>)   **Что такое делегаты?**   * Делегаты – это функциональный тип в .NET Framework;   делегаты иногда называют **“безопасными указателями на функции”**   * Делегаты обладают некоторым свойством с интерфейсами, т.к. также определяют условия контракта между вызывающей и реализующей стороной (в отличие от интерфейса делегат лишь задает форму одной функции) * Из-за своей динамической природы делегаты часто используют в ситуациях, когда необходимо дать возможность пользователю изменить поведение класса или метода (например, в классе коллекции порядок сортировки можно задать в виде делегата, определяющего функцию сравнения) * Делегаты также используются в качестве обработчиков событий, а также при асинхронных взаимодействиях   **Тип делегата**   * Так как C# строго типизированный язык, то и делегаты также имеют тип (можно понимать как тип метода, на который может ссылаться делегат заданного типа)   + Переменной делегата можно присвоить какое-либо значение только при совпадении типов:   DelegateType d = <выражение с типом DelegateType>;   * Тип делегата определяется набором параметров делегата (имеют значения только количество и типы параметров, а также их порядок) и типом возвращаемого значения * Типы делегатов можно также называть функциональными типами   **Создание делегатов**   * Делегаты создаются в виде ссылки (безопасного указателя)   + на именованный метод (named method)     - статический метод     - метод объекта (динамический метод)   + на анонимную функцию (anonymous method)     - анонимный метод     - лямбда-выражение   **Анонимные функции**   * Анонимная функция – это оператор или выражение "inline", которое можно использовать каждый раз, когда ожидается тип делегата * Ее можно использовать для инициализации именованного делегата или подставить вместо типа именованного делегата в качестве параметра метода * Анонимная функция – это оператор или выражение "inline", которое можно использовать каждый раз, когда ожидается тип делегата * Ее можно использовать для инициализации именованного делегата или подставить вместо типа именованного делегата в качестве параметра метода   **Общий вид блока try-catch-finally**   * (Для I, J>0)   любой Exception<I>  не может быть более общим, чем Exception<I+J>  (проверяется компилятором)   * В случае исключения всегда срабатывает catch c наиболее частным (верхним в описании) исключением   **Иерархия исключений**   * Все исключения наследуются от **System.Exception** * Системные исключения (которые генерируются .NET Framework) наследуются от **System.SystemException** * Пользовательские классы исключений рекомендуется наследовать от **System.ApplicationException**   **Свойства Exception**   * **ICollection Data**   – коллекция для хранения дополнительной пользовательской информации об исключении   * **String HelpLink**   – ссылка на файл с описанием исключения   * **Exception InnerException**   – вложенное исключение   * **string Message**   – текстовое сообщение об ошибке   * **string Source**   – имя объекта, вызвавшего исключение   * **string StackTrace**   – трассировка стека вызова методов до того места, в котором сгенерированно исключение   * **MethodBase TargetSite**   – метод, вызвавший исключение | **Замыкания**   * + Замыкание (англ. closure) в программированиия — процедура, которая ссылается на свободные переменные в своём лексическом контексте.   + Замыкание, так же как и экземпляр объекта, есть способ представления функциональности и данных, связанных и упакованных вместе.   + Замыкание — это особый вид функции. Она определена в теле другой функции и создаётся каждый раз во время её выполнения. В записи это выглядит как функция, находящаяся целиком в теле другой функции. При этом вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит создание нового экземпляра внутренней функции, с новыми ссылками на переменные внешней функции.   + В случае замыкания ссылки на переменные внешней функции действительны внутри вложенной функции до тех пор, пока работает вложенная функция, даже если внешняя функция закончила работу, и переменные вышли из области видимости   + Анонимные функции в C# являются замыканиями   + Внешними переменными анонимного метода называют локальные переменные и параметры, область действия которых содержит объявление анонимного метода   **Системные (библиотечные) generic-делегаты**   * public delegate void Action<T>(T obj) * public delegate bool Predicate<T>(T obj) * public delegate int Comparison<T>(T x, T y) * public delegate TOutput   Converter<TInput, TOutput>(TInput input)   * public delegate void EventHandler<TEventArgs> (Object sender, TEventArgs e)   where TEventArgs: EventArgs  **Суть событий в языках программирования**   * При помощи событий класс оповещает другие классы о произошедшем изменении ситуации * В механизме событий используется идиома “публикация/подписка”; класс публикует те события, которые он может инициировать, другие классы могут подписаться на событие   **События в C#**   * События – особый вид поля класса, к которому можно привязывать обработчики событий * Функция, вызываемая при поступлении события, задается при помощи делегата * К одному событию может быть привязано несколько обработчиков (цепочка обработчиков или цепочка экземпляров делегатов), при инициировании события все они будут вызваны   + Порядок вызова привязанных обработчиков не определен   + В случае, если выполнение какого-либо из обработчиков породит необработанное исключение, остальные обработчики в цепочке вызваны не будут * Т.о. событие можно представить как **список экземпляров делегатов** (список безопасных указателей на функции) одного типа (совпадает с типом события), которые вызываются при инициировании события (наступлении события)   **Соглашение о событиях в .NET FCL**   * Несмотря на то, что можно объявить события с любыми сигнатурами, в .NET FCL действует негласное соглашение, что все события имеют следующую сигнатуру:   void (object sender, EventArgs e)  вместо EventArgs может быть его наследник  Пример **Samples.DelegateAndEvent.ChangeNumEventDemonstration (см. проект .\OOP.Lecture1-2.Samples)**   * ChangeNumEventArgs – класс для описания передаваемого сообщения, наследник от EventArgs, добавляет свойство Number * ChangeNumEventHandler – тип делегата и, соответственно, тип события * ChangeNumEventObject – объект с событием OnChangeNumHandler типа ChangeNumEventHandler, рассылаемом при изменении свойства Number * ListenerForm – форма, которая может привязать и убрать собственный обработчик события к экземпляру ChangeNumEventObject * ChangeNumForm – форма, в которой можно “заставлять” экземпляр ChangeNumEventObject порождать события, изменяя свойство Number данного экземпляра * При старте примера создается экземпляр ChangeNumEventObject, который передается созданным формам ListenerForm и ChangeNumForm; также к данному экземпляру привязывается анонимный обработчик печати нового чиста, получаемого из параметра ChangeNumEventArgs, на консоль * Изменяя свойство Number из ChangeNumForm можно наблюдать за обработкой событий   **Чем плохи коды возврата?**   * + На каждый вызов функции, которая может привести к ошибке, надо писать код обработки исключения   + Способы возврата результата успешности выполнения может быть различным (HRESULT, bool, out/ref-параметр)   **Другой способ обработки ошибок – исключения**   * В .NET Framework основной механизм обработки ошибочных состояний – исключения * Перехват исключений требует специфической организации программного кода (блок try-catch-finally) * Исключения в отличие от кодов возврата не получится игнорировать, они всегда изменяют ход выполнения программы   + или исключение перехватывается пользовательским обработчиком исключения   + или срабатывает стандартный обработчик исключений, который (при настройках по-умолчанию) в консоль или в диалоговое окне сообщений печатает информацию о произошедшем исключении и останавливает выполнение программы |