Разработка компиляторов на платформе .NET

Кафедра системного программирования, СПбГУ

Курс «Разработка компиляторов на платформе .NET»

Кафедра системного программирования Санкт-Петербургского государственного университета

Введение

- Цели и задачи данного курса
- Рамки курса
- Структура курса
- Необходимые предварительные знания

Введение

Данный курс посвящен принципам разработки компиляторов. Основные задача данного курса — познакомить студентов с базовыми идеями и методами, используемыми при создании современных компиляторов, а также дать практические навыки написания простых компиляторов. В качестве целевой платформы для компиляторов в данном курсе используется Microsoft .NET. Подразумевается, что к моменту окончания данного курса большинство студентов смогут самостоятельно создать работающий компилятор с простого C#-подобного языка программирования.

Теория создания компиляторов активно развивалась в течение последних 50-60 лет и к сегодняшнему дню в данной области накоплено огромное количество знаний. Поэтому практически невозможно подробно осветить все вопросы создания компиляторов в рамках университетского курса. В этом курсе авторы пытаются лишь преподать основные принципы создания компиляторов и познакомить студентов с некоторыми типичными распространенными приемами. Для дальнейшего совершенствования полученных навыков студенту необходима практика и самостоятельное изучение последних достижений в этой области.

Курс состоит из двух частей — теоретической и практической. Теоретическая часть организована в виде презентаций и данного учебника, а практическая часть состоит из демонстраций и самостоятельных упражнений.

Для полноценного понимания курса студенту потребуются базовые знания языка C# и платформы .NET в целом. Но так как эти знания еще нельзя считать повсеместно распространенными, курс содержит краткое введение в .NET, которое поможет студентам получить представление об этих технологиях, а также оценить свои знания платформы .NET.

1. Обзор платформы .NET

- Общая идея архитектуры .NET
- Достоинства и недостатки .NET
- Схема трансляции программ в .NET
- Основные черты промежуточного представления, используемого в .NET (обзор MSIL)
- Безопасность в .NET
- Объектная модель .NET
- Понятие сборки. Манифест сборки
- Модель безопасности в .NET
- Единая система типов

Лекция 1. Обзор платформы .NET

В этой лекции обсуждаются следующие вопросы:

- Общая идея архитектуры .NET
- Достоинства и недостатки .NET
- Схема трансляции программ в .NET
- Основные черты промежуточного представления, используемого в .NET (MSIL)
- Безопасность в .NET
- Объектная модель .NET
- Понятие сборки. Манифест сборки.
- Модель безопасности в .NET
- Единая система типов данных

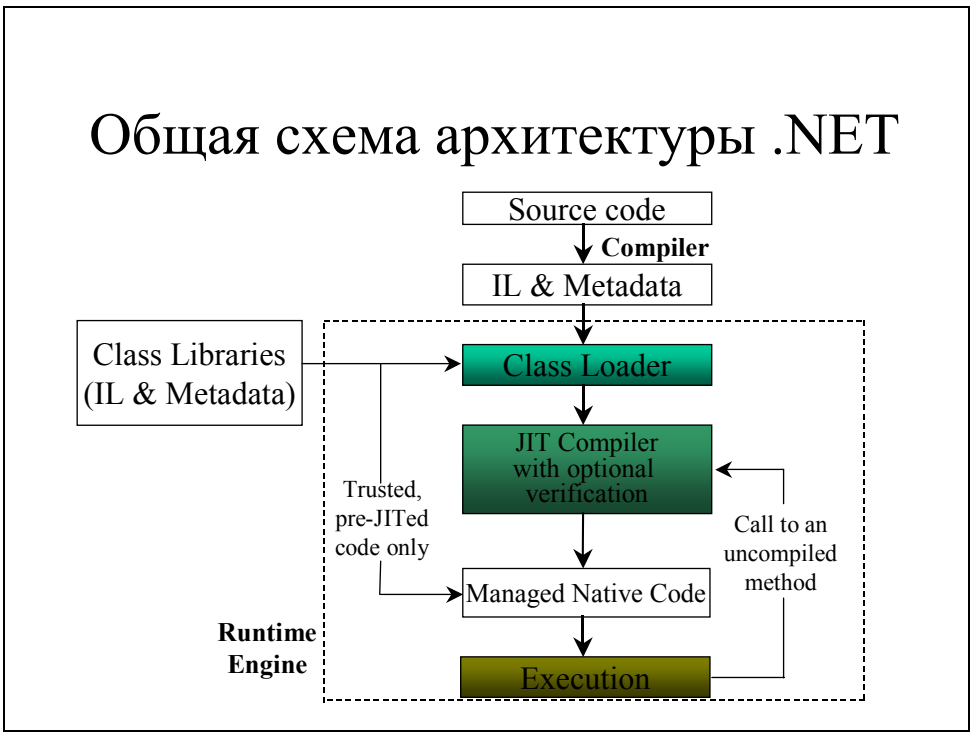
Что такое .NET?

- Перспективы платформы .NET
- Основные понятия платформы .NET
- .NET с точки зрения разработчика компиляторов

Обзор платформы .NET

Платформа Microsoft .NET появилась относительно недавно, в 2000 году. Более того, на момент написания данного курса платформа все еще находилась в стадии бета-тестирования (при создании примеров авторы использовали вторую бета-версию Visual Studio.NET). Тем не менее, технологические преимущества платформы .NET в совокупности с активной маркетинговой поддержкой приводят к тому, что популярность .NET неуклонно растет. Выпуск платформы .NET наверняка коснется всех разработчиков программ для Windows. Поэтому чтение курсов, основанных на .NET, представляется чрезвычайно полезным для студентов: к моменту выпуска студентов из университета у них уже будут знания технологий, с которыми они, скорее всего, будут непосредственно работать.

Первые несколько лекций данного курса посвящены описанию собственно платформы .NET. Практически любой курс, посвященный созданию компиляторов, содержит подробное описание целевой платформы, для которой планируется генерация конечного кода. В большинстве случаев — это описание архитектуры и ассемблера целевого компьютера. Однако в нашем случае мы приводим описание *виртуальной* (т.е. несуществующей физически) машины, которая описывается спецификацией платформы .NET. С точки зрения преподавания разработки компиляторов это и хорошо, и плохо: с одной стороны, студенты на практике познакомятся с популярной сегодня идеей использования виртуальных машин, но с другой стороны, некоторые аспекты работы с конкретной машинной архитектурой остаются скрытыми, так как об этом заботится не разработчик компилятора, а авторы платформы .NET. Тем не менее, мы решили также осветить различные аспекты написания компиляторов, не являющиеся необходимыми при написании компиляторов, ориентированных на .NET, но необходимыми при написании компиляторов для других платформ, в целях создания у студентов более полного представления о различных вариантах создания компиляторов.

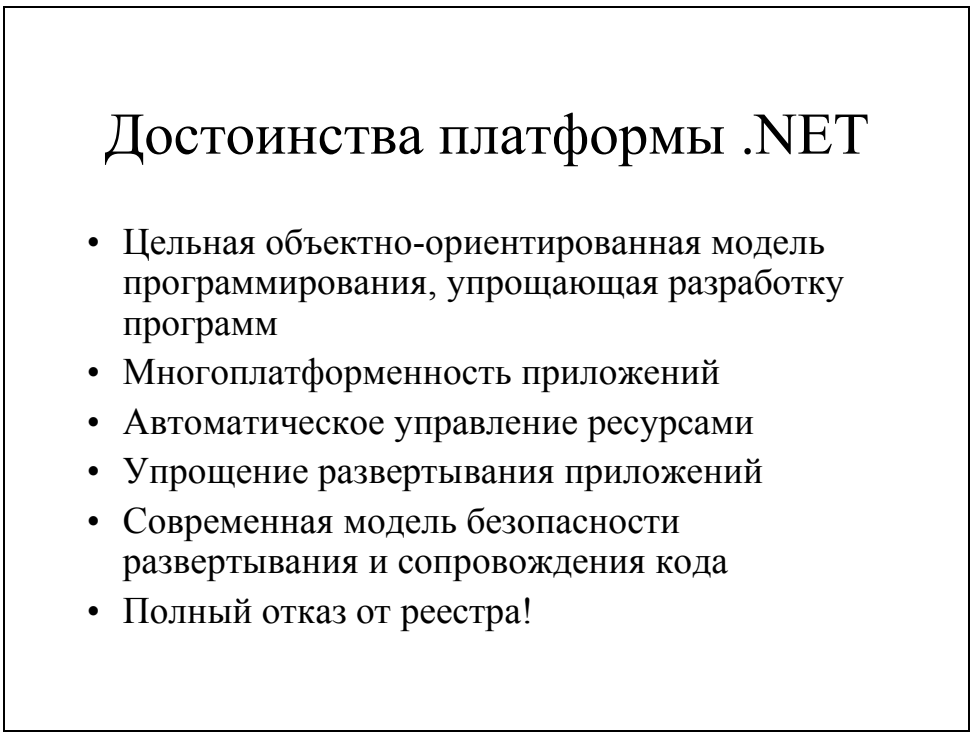


Общая схема архитектуры .NET

На слайде представлена общая схема трансляции в .NET (рисунок заимствован из статьи Дж.Рихтера, опубликованной в сентябрьском выпуске 2000 года журнала MSDN Magazine).

Исходные тексты программ компилируются в специальное *промежуточное представление (Microsoft Intermediate Language*, часто употребляется сокращение IL или MSIL). Промежуточное представление содержит всю необходимую информацию о программе, но не привязано к какому-либо определенному языку программирования или к машинному коду какой-либо целевой платформы. Для запуска программы необходимо специальное окружение, исполняющее программы, и библиотеки динамической поддержки (execution engine & runtime). Важной особенностью трансляции в .NET является то, что промежуточное представление не интерпретируется; вместо этого используется механизм компиляции времени исполнения, который генерирует машинный код. Подразумевается, что большинство программ, исполняемых на платформе .NET, будет использовать многочисленные стандартные классы, предоставляющие базовую функциональность (от работы с данными до встроенных механизмов безопасности).

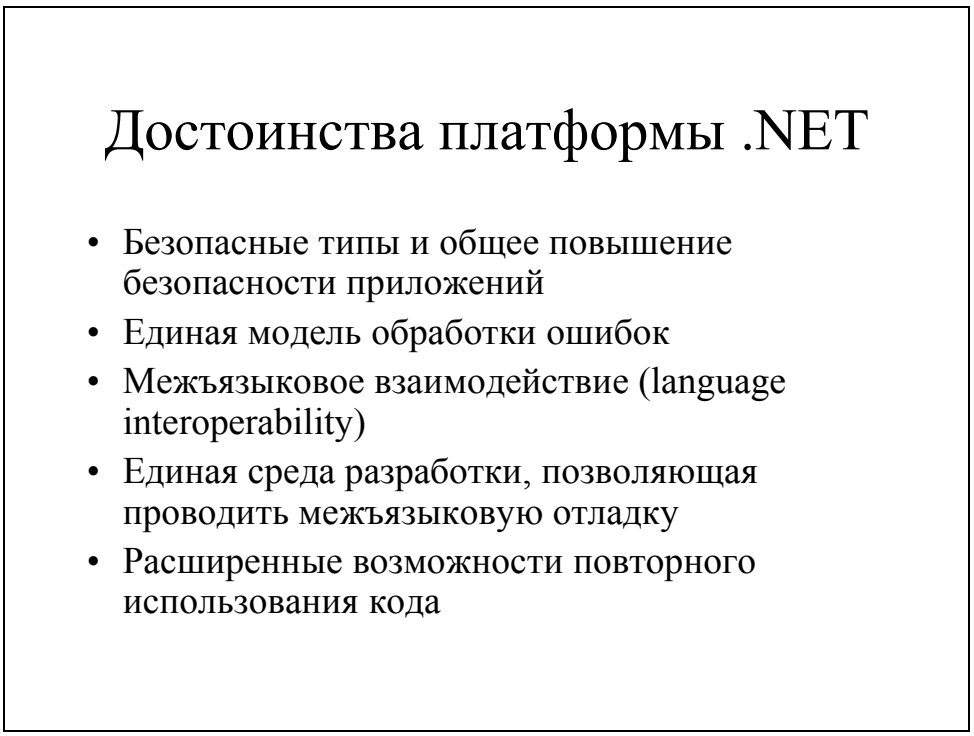
На последующих слайдах мы кратко остановимся на основных преимуществах платформы .NET по сравнению с существующими подходами.



Достоинства платформы .NET

Платформа .NET основана на единой объектно-ориентированной модели; все сервисы, предоставляемые программисту платформой, оформлены в виде единой иерархии классов. Это решает многие проблемы программирования на платформе Win32, когда большинство функций были сосредоточены в COM-объектах, а некоторые функции необходимо было вызывать через DLL.
Благодаря тому, что промежуточное представление .NET не привязано к какой-либо платформе, приложения, созданные в архитектуре.NET, являются многоплатформенными.
Платформа .NET предоставляет автоматическое управление ресурсами. Это решает многие распространенных проблемы, такие как утечки памяти, повторное освобождение ресурса и т.п. На самом деле, в .NET вообще нет никакого способа явно освободить ресурс!

Одной из наиболее распространенных трудностей при развертывании приложения является использование разделяемых библиотек. Из-за этого установка нового приложения может привести к прекращению работы ранее установленного приложения. В архитектуре .NET установка приложения может свестись к простому копированию всех файлов в определенный каталог. При установке используются криптографические стандарты, которые позволяют придавать разную степень доверия различным модулям приложения. Наконец, приложения .NET не используют реестр Windows — возможность отказаться от реестра достигается засчет использования механизма *метаданных*.



Достоинства платформы .NET

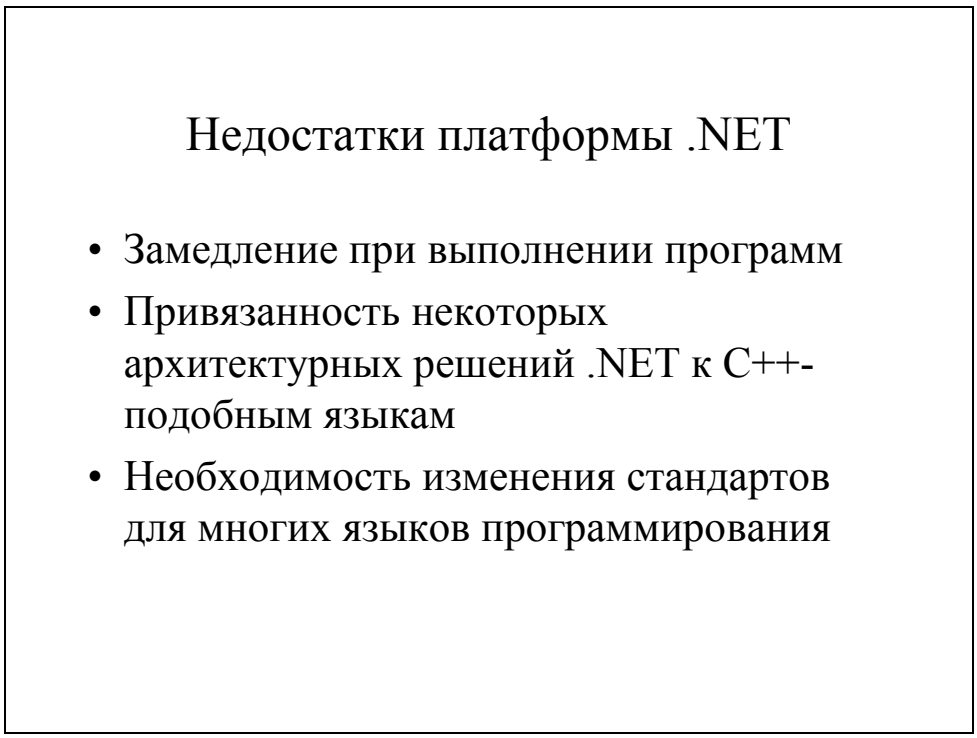
Код, сгенерированный для .NET, может быть проверен на безопасность. Это гарантирует, что приложение не может навредить пользователю или нарушить функционирование операционной системы (так называемая «модель песочницы»). Таким образом, приложения для .NET могут быть сертифицированы на безопасность.

Обработка ошибок в .NET всегда производится через механизм исключительных ситуаций. Это решает неоднозначность ситуации, когда некоторые ошибки обозначаются с помощью кодов ошибки платформы Win32, некоторые возвращают HRESULTS и т.п.

Вероятно, самым большим обещанием .NET остается межъязыковое взаимодействие (language interoperability). Впервые в истории программирования появляется единая модель, позволяющая на равных пользоваться различными языками для создания приложений. Так как MSIL не зависит от исходного языка программирования или от целевой платформы, в рамках .NET становится возможным развивать новые программы на базе старых программ – причем и первый, и второй языки программирования не так уж важны!

Естественно, что для такого подхода к разработке программ необходимо обеспечить, например, межъязыковую отладку (многие сталкивались с трудностями отладки при вызове C++ библиотеки из Visual Basic). Visual Studio.NET поддерживает этот процесс прозрачно для пользователя и не делает различий между языками, на которых было написано исходное приложение.

Перечисленные выше особенности платформы .NET позволяют добиться простоты повторного использования кода. Раньше платформа Win32 позволяла повторное использование только на уровне COM-компонент; теперь можно повторно использовать классы и наследовать от них свои приложения.



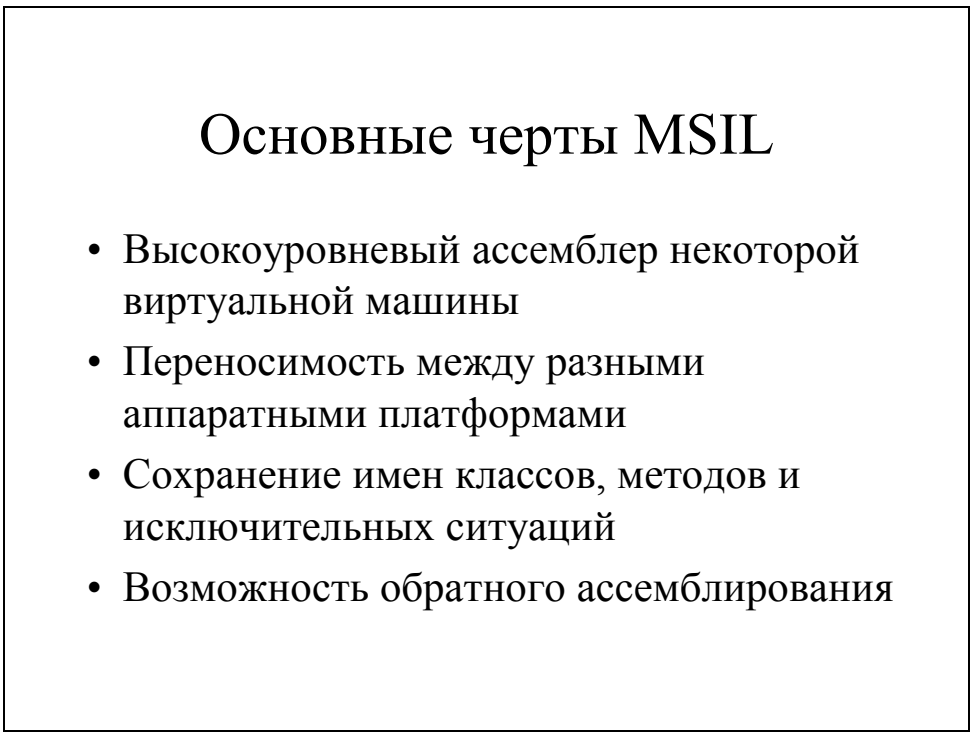
Недостатки платформы .NET

Естественно, что все преимущества .NET, которые мы перечислили выше, не могут быть абсолютно бесплатными. Как и у любой другой архитектуры, у .NET есть свои недостатки.

Самым ощутимым недостатком является существенное замедление выполнения программ. Это неудивительно, так как между исходным языком и машинным кодом вводится дополнительный уровень, MSIL. Однако промежуточное представление .NET с самого начала проектировалось с прицнлом на компиляцию времени исполнения (в отличие, например, от Java bytecode, который разрабатывался с прицелом на интерпретацию).
Это дает некоторые дополнительные возможности по борьбе с замедлением. Например, можно равномерно распределить замедление при запуске, так как обычно компилируется не вся библиотека, а только тот метод, который вызывается, и повторной компиляции одного и того же метода не производится.

Другая проблема .NET заключается в том, что при ее создании основной упор был сделан на C++/Java-подобные языки (например, конструкторы с именем, равным имени метода, запрет множественного наследования и т.п.). Это ограничивает возможности интеграции некоторых языков с более богатыми возможностями, особенно с принципиально отличающимися языками, такими как функциональные языки (ML, Haskell, Scheme) или устаревшие языки (Кобол, PL/I). Во многих случаях разработчикам компиляторов все-таки удается реализовать "проблемные" особенности исходных языков в рамках .NET, пусть даже и не слишком тривиальным образом — достаточно сказать, что уже существуют реализации типичных представителей этих классов языков для платформы .NET. Другое направление связано с развитием самой платформы .NET: например, недавно было заявлено о поддержке платформой .NET механизма параметрического полиморфизма (generics).

Наконец, наблюдается и движение с противоположной стороны: уже сегодня стандарты некоторых языков программирования претерпевают значительные изменения для того, чтобы эти языки могли быть поддержаны в .NET.

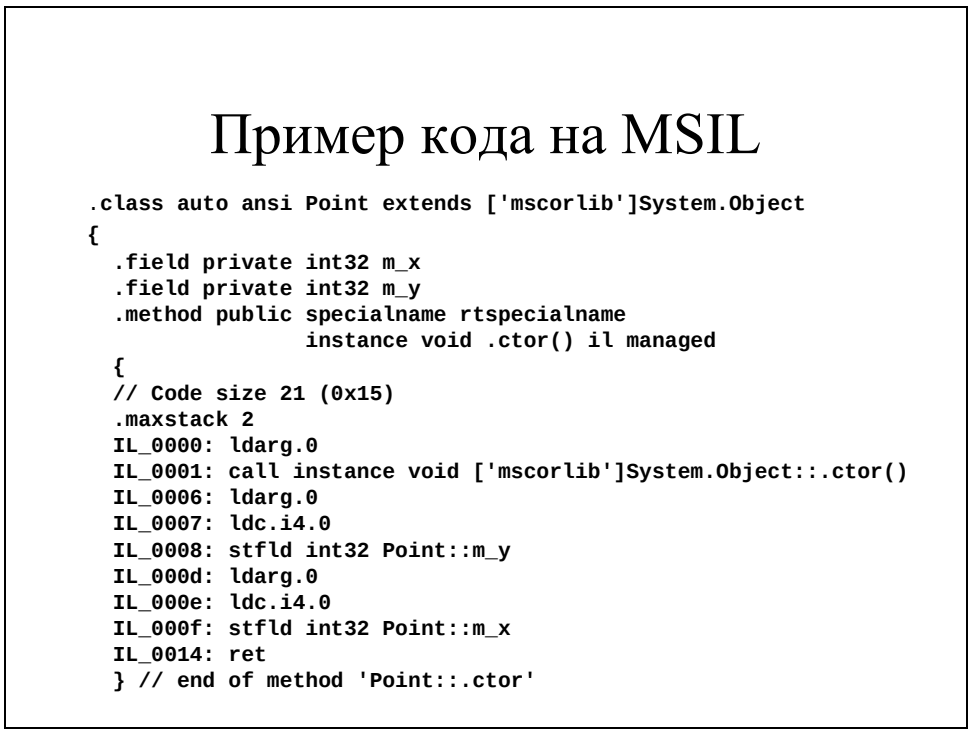


Основные черты MSIL

MSIL можно рассматривать как ассемблер некоторой виртуальной машины. Это нетипичный ассемблер, так как он обладает многими конструкциями, характерными для языков более высокого уровня: например, в нем есть инструкции для описания пространств имен, классов, вызовов методов, свойств, событий и исключительных ситуаций. Кроме того, MSIL является стековой машиной со статической проверкой типов; это позволяет отслеживать некоторые типичные ошибки.

MSIL представляет собой дополнительный уровень абстракции, позволяющий легко справляться с переносом кода с одной платформы на другую, в том числе, и с изменением разрядности платформы: в отличие от Java bytecode MSIL не завязан на 32 бита или какую-либо другую фиксированную разрядность. В данный момент существуют версии MSIL для мобильных 16-разрядных устройств (.NET Compact Framework), стандартная 32-разрядная версия и специальная версия для работы с получающими все более широкое распространение 64-разрядными устройствами.

Отметим, что MSIL сохраняет достаточно много информации об именах, использованных в исходной программе: имена классов, методов и исключительных ситуаций сохраняются и могут быть извлечены при обратном ассемблировании. Однако извлечение из MSIL исходных текстов путем дизассемблированя вряд ли имеет смысл, так как имена локальных переменных, констант и параметров сохраняются только в отладочной версии.



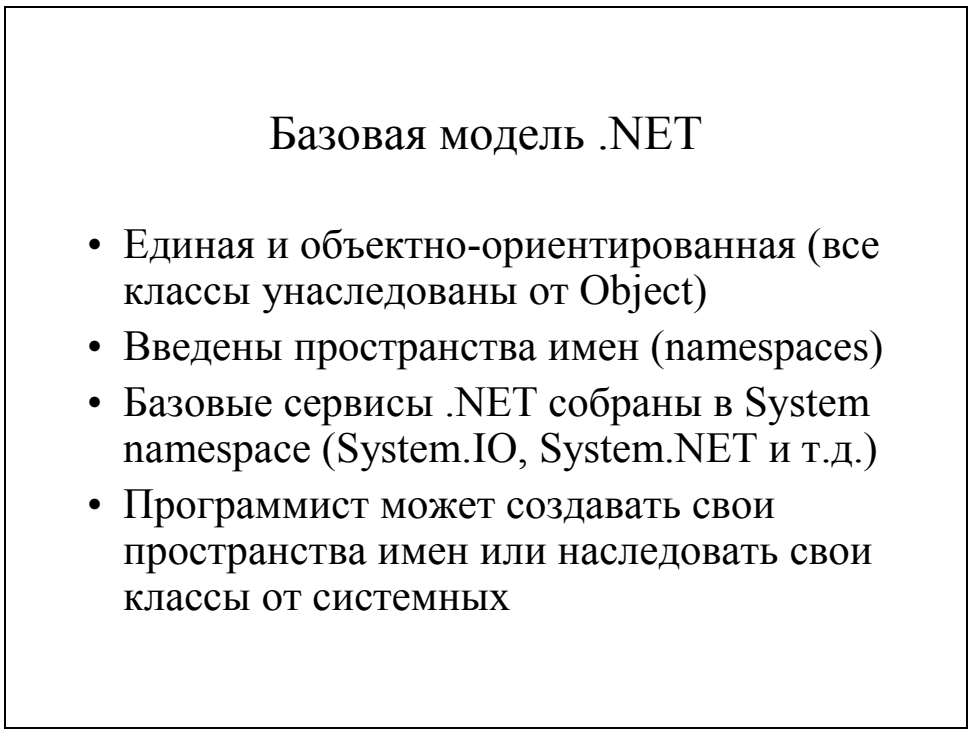
Пример кода на MSIL

На слайде приведен фрагмент MSIL-кода, сгенерированный по следующему классу на C#:

```
class Point
{
  private int m_x, m_y;
  public Point() { m_x = m_y = 0; }
}
```

В сгенерированном коде можно найти описание класса Point и тот факт, что он унаследован от System.Object, описание закрытых переменных типа Int32 m_x и m_y (отметим, что их имена сохраняются при обратном ассемблировании) и, наконец, конструктор класса Point.

На начальном этапе знакомства с .NET изучение сгенерированного MSIL-кода представляется весьма полезным, так что рекомендуем слушателям самостоятельно ознакомиться с утилитой ILDasm.

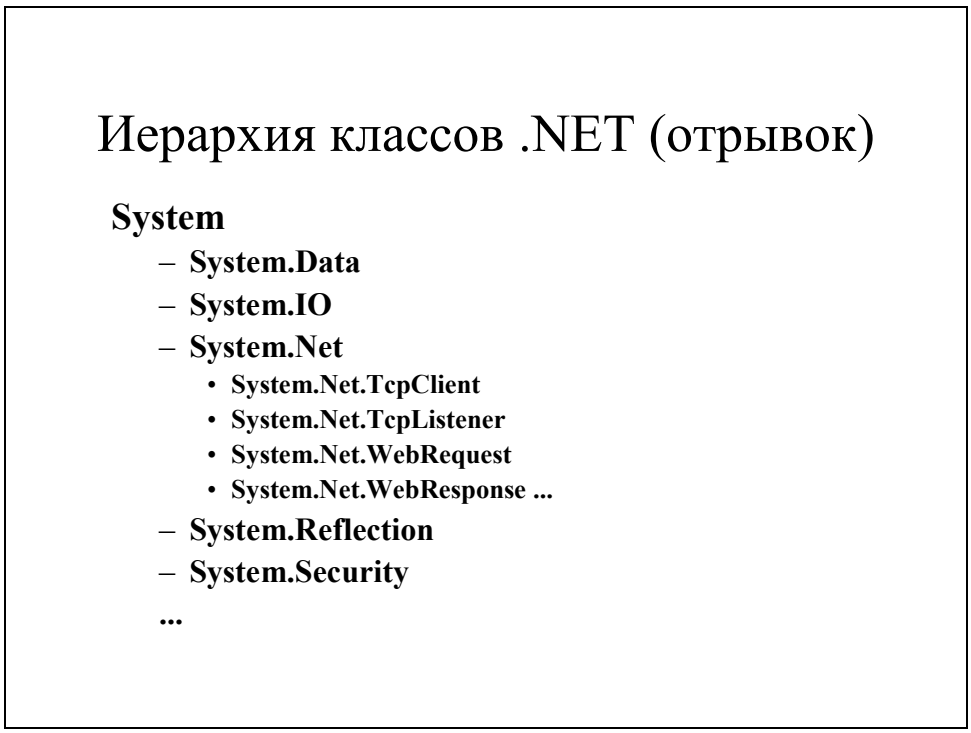


Базовая модель .NET

В основе .NET лежит единая объектно-ориентированная модель классов, в которой все классы унаследованы от базового класса Object. Классы разбиты на пространства имен для избежания накладок при совпадении имен. Основные сервисы .NET сосредоточены в пространстве имен System (например, там находится упоминавшийся выше класс Object).

Пространства имен имеют много уровней вложенности (например, System.WinForms или System.Web.UI.WebControls). На следующем слайде мы приведем часть иерархии классов .NET.

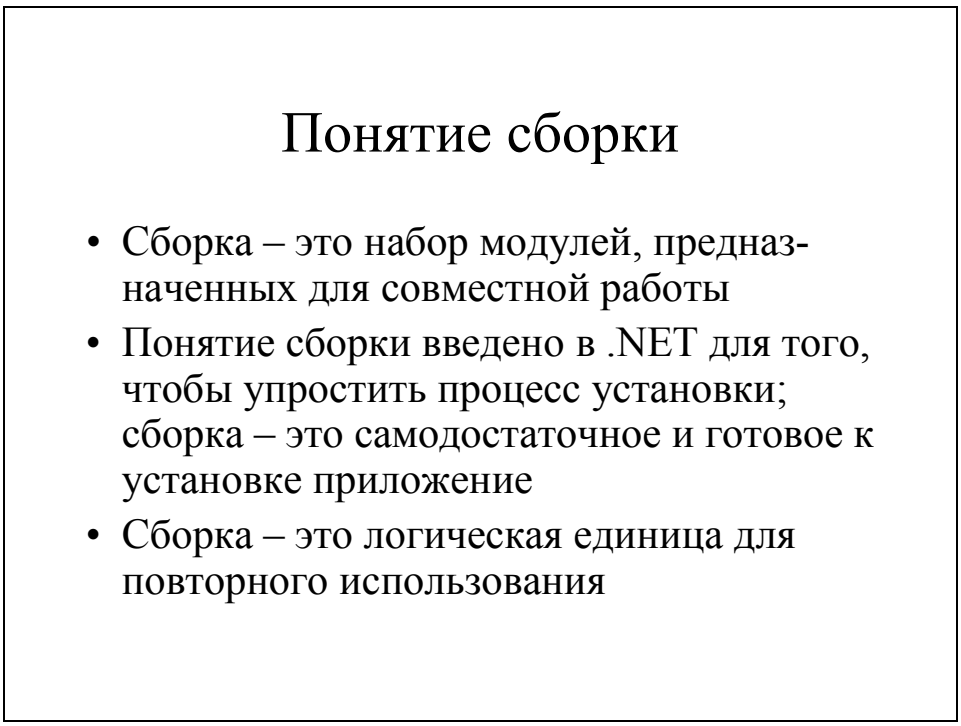
Программисты могут создавать собственные пространства имен для своих классов или пользоваться уже существующими классами, расширяя их функциональность путем наследования и переопределения методов.
Модель платформы .NET может существенно упростить разработку приложений по сравнению с программированием для Windows-платформ, где практически вся функциональность предоставлялась разработчику как неструктурированный набор функций в Windows API.



Иерархия классов (отрывок)

На слайде приведен отрывок из иерархии классов .NET. Несмотря на то, что идея сведения всех объектов в единую иерархию не нова (одним из первых языков с единой иерархией объектов был SmallTalk, затем подобный прием был использован в Java), отличительной особенностью платформы .NET является то, что в .NET единая модель объектов распространяется сразу на все языки программирования.

Системные классы становятся единственным методом взаимодействия программы с внешним миром для управляемого кода. Одним из следствий этого является необходимость переписывания всего ввода/вывода для существующих приложений, которые должны быть интегрированы в .NET. Это можно делать поэтапно, так как практически вся функциональность предыдущих версий языка поддержана в .NET специальными классами в целях обратной совместимости. Единственным заметным исключением является Visual Basic.NET, в котором необходимо целиком переделывать внешний вид форм путем использования классов из System.WinForms.

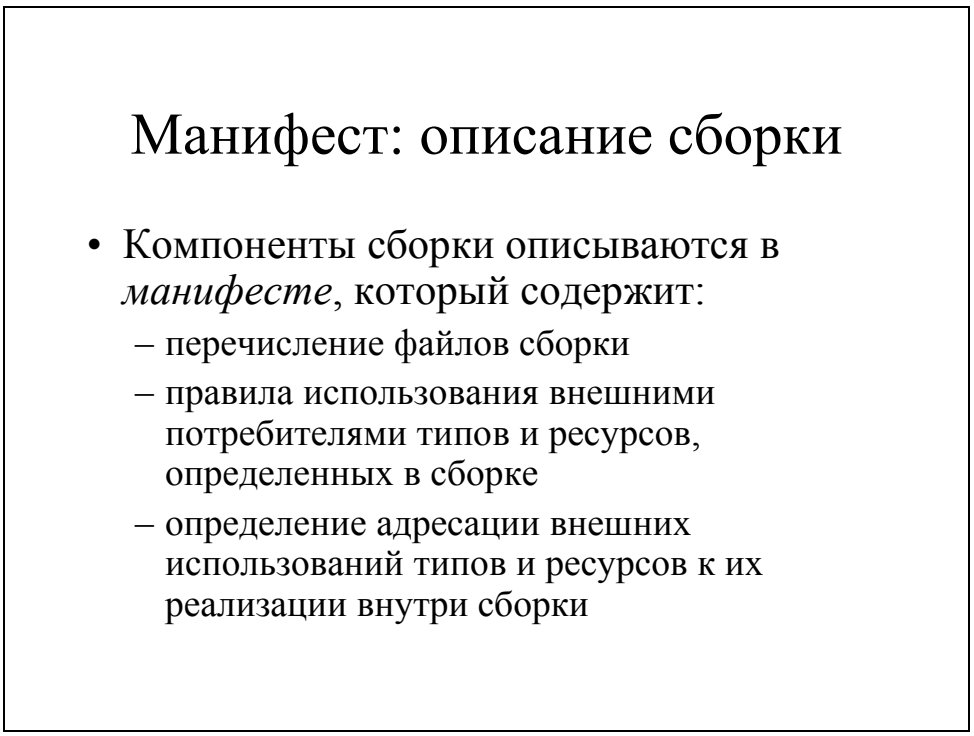


Понятие сборки

В платформе .NET появилось новое понятие — *сборка (assembly)*. В первом приближении сборку можно воспринимать как аналог EXE или DLL; более того, в случае приложения, состоящего из одного файла, сборка даже имеет расширение .exe или .dll. Несмотря на это сходство, сборка содержит существенно больше информации о приложении, чем традиционные исполняемые файлы.

Причиной появления понятия сборки можно считать трудности установки Windows-приложений. Обычное Windows-приложение состоит из множества файлов — запускаемые модули, библиотеки, дополнительные файлы и т.п. Помимо этого, при установке некоторых приложений (особенно COM-компонент) необходимо записывать в реестр Windows сведения о нахождении и способе вызова. Наконец, многие приложения использовали разделяемые DLL, что зачастую приводило к проблемам при установке более новых версий этой DLL.

Понятие сборки было введено для того, чтобы решить эти проблемы. Сборка представляет собой набор файлов, модулей и дополнительной информации, которые должны обеспечить простую установку приложения и последующую работу. Таким образом, можно говорить и о том, что повторное использование приложений может быть реализовано с помощью интеграции различных сборок.



Манифест: описание сборки

Для того чтобы сборки действительно были независимыми от системы и от других сборок, необходимо, чтобы они сопровождались явным описанием предоставляемых ими сервисов и зависимостей от внешнего мира. Роль такого описания выполняет так называемый *манифест сборки*.

В манифесте должны быть перечислены все файлы и модули, из которых состоит данная сборка, а также должны быть четко прописаны все интерфейсы со внешним миром. Кроме того, манифест должен указывать, каким образом реализуются обращения к типам и ресурсам, экспортируемым из данной сборки. Естественно, что впоследствии во время компиляции и загрузки необходимо будет учесть и разрешить все внешние зависимости данного приложения.

Таким образом, манифест является тем инструментом, который позволяет скрыть от потребителя детали реализации. Именно благодаря этому механизму каждая сборка является самодостаточной и не требует привлечения внешних средств, таких как реестр. Это позволяет в большинстве случаев свести установку приложения к простому копированию.

Уникальность сборки

- Каждая сборка имеет заведомо уникальное имя, которое состоит из:
  - префикса, основанного на открытом ключе разработчика (для общих сборок)
  - простого текстового имени
  - номера версии
  - информации о локализации
- Все общие сборки подписываются секретным ключом разработчика

Уникальность сборки

Каждая сборка имеет уникальное имя, которое состоит из следующих частей: префикса, основанного на открытом ключе разработчика, простого текстового имени, номера версии и информации о локализации. Некоторые сборки могут иметь только простое текстовое имя, но в таких случаях их можно использовать только как часть другого приложения (так как иначе нельзя гарантировать их уникальность).

Все остальные сборки, называемые общими или разделяемыми, сопровождаются префиксом, основанном на открытом ключе, номером версии в формате *incompatible.compatible.hotfix* и привязкой к локализации (указание на поддержаный язык общения с пользователем или язык по умолчанию).

Кроме того, все разделяемые сборки подписываются секретным ключом. Это подтверждает аутентичность разработчика и предотвращает несанкционированные изменения кода. И то, и другое важно в том случае, когда установка приложения происходит через Интернет.

Безопасность в .NET

Безопасность – неотъемлемая часть .NET.

Способы обеспечения безопасности:

- Безопасность типов
- Проверка подлинности кода
- Разрешение доступа к ресурсам
- Декларативная безопасность
- Императивная безопасность

Безопасность в .NET

Безопасность является краеугольным камнем .NET. На всех этапах создания и выполнения программ происходят самые различные проверки - от проверки прав на доступ к коду до разрешений на ресурсы. Вот некоторые из типов проверок безопасности:

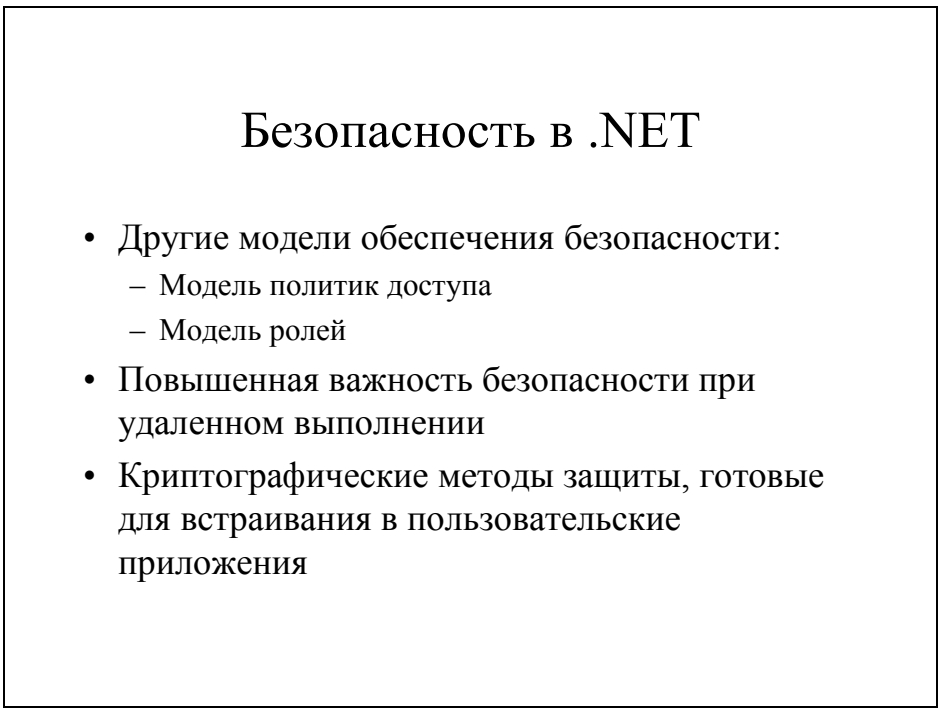
Безопасность типов. Программы, гарантирующие безопасность данных, обращаются только к тем участкам памяти, которые были выделены для них. Доступ к объектам осуществляется только через специальные интерфейсы, в которые встроены проверка безопасности. В целом, безопасность типов может быть проверена не всегда; однако ее наличие гарантирует невозможность одной из самых распространенных атак (чтение указателя большего размера, чем выделенная память).

Подлинность кода. Загрузчик классов сохраняет информацию об исходных текстах всех классов, которые были загружены. Таким образом, можно восстановить некоторые атрибуты кода (откуда загружен код, кто является атором и т.п.). Эту информацию можно использовать для дачи прав на запуск.

Разрешения на доступ к ресурсам. Ресурсы обычно ассоциированы с системой. В качестве ресурсов могут выступать файлы, сетевые соединения, право вызова неуправляемых API (unmanaged APIs). Отметим, что права доступа проверяются не только для вызвавшей сборки, но и для всех прочих, находящихся в данный момент в стеке вызовов. Это позволяет предотвратить классическую атаку, в которой неавторизованный компонент получает доступ к ресурсу путем обращения к нему через вызов компоненты с другими правами доступа.

Декларативная безопасность. Данный механизм предоставляет возможность встраивать проверки безопасности прямо в код путем аннотации классов, полей или методов. Проверка может производиться однократно при загрузке или постоянно (скажем, при каждом запуске метода).

Императивная безопасность. Обычный код внутри разрабатываемого метода, который проверяет права на данную операцию во время запуска. Такие проверки важны для доступа к файлам, пользовательскому интерфейсу и т.п.



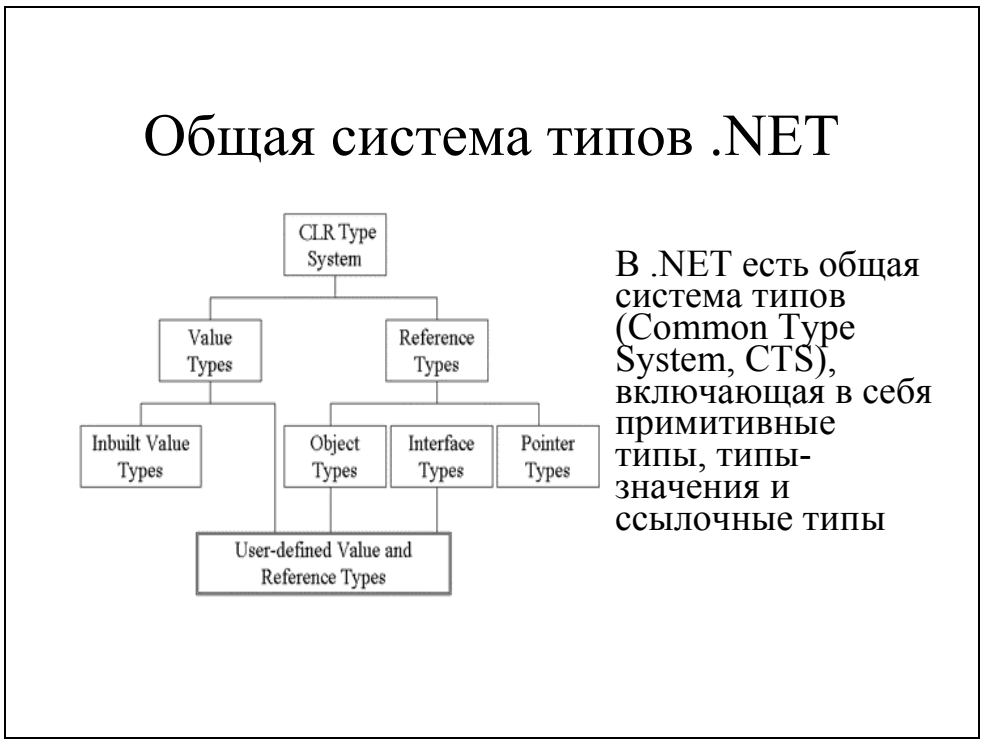
Безопасность в .NET

Упомянем еще два способа обеспечения безопасности и защиты приложений в .NET:

Модель политик доступа. Политики позволяют автоматически присваивать коду определенные привилегии, основываясь на том, откуда были получены исходные тексты данного приложения (понятно, что локальному коду пользователи доверяют больше, чем программам из Интернета). Пользователи могут изменять эти политики.
Модель ролей. Эта модель безопасности проверяет, в какой роли выступает пользователь и разрешает/отказывает в доступе в зависимости от этого. Например, в финансовых приложениях максимальный лимит транзакции может зависить от служебного положения банковского служащего.

Отметим также, что .NET поддерживает удаленный вызов объектов. При удаленном вызове (т.е. при вызове объекта, находящегося в другом процессе или на другой машине) все стандартные вопросы безопасности, такие, как аутентификация, авторизация, конфиденциальность и целостность, приобретают повышенную важность. Платформа .NET поддерживает описанные выше механизмы безопасности и при удаленных вызовах путем тесной интеграции с сетевыми протоколами и прочей инфраструктурой.

Механизмы аутентификации будут пригодны как для идентификации пользователей, так и для идентификации приложений или юридических лиц. Конфиденциальность и целостность будут основаны на криптографических алгоритмах и стандартных сетевых протоколах, таких как SSL или IPSec. Все эти возможности будут доступны как на уровне приложения, так и на уровне передачи данных. Кроме того, .NET предоставляет готовые криптографические библиотеки, которые пригодны не только для использования в системных сервисах, но и в пользовательских приложениях (по информации на данный момент, в .NET будет встроен станларт RSA).



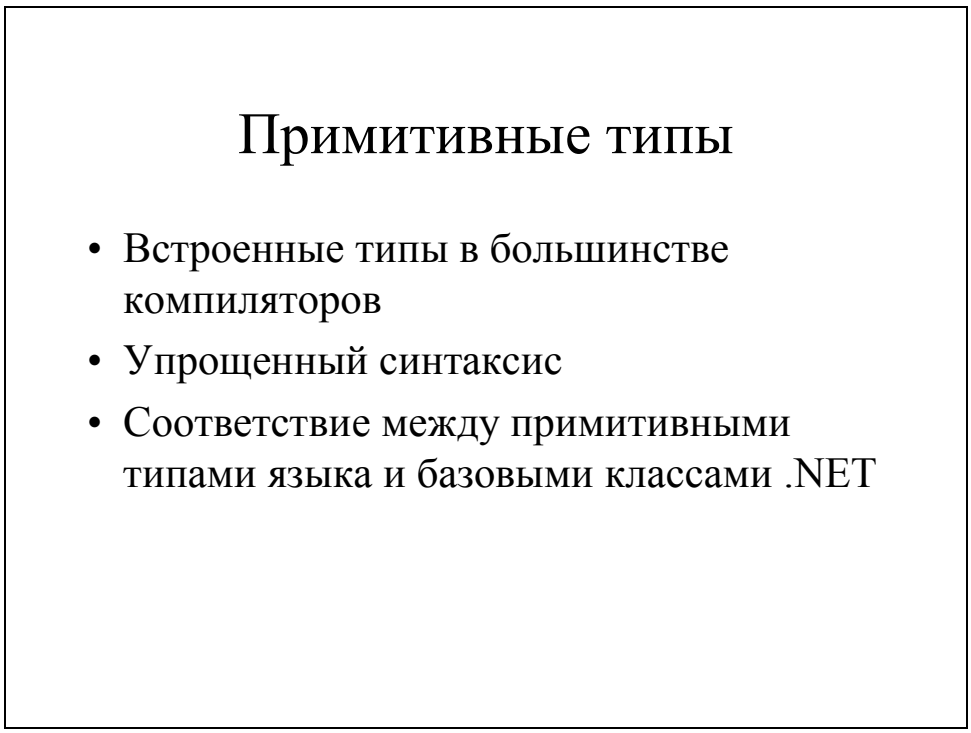
Общая система типов .NET

Для того, чтобы различные языки программирования могли осмысленно общаться между собой, необходимо ввести единую систему типов, которая была бы достаточно полной и при этом оставалась бы ясной. Например, в стандарте CORBA концепции языков и типов определены в Object Management Architecture. В .NET такую роль выполняет Common Type System (CTS). В CTS все типы делятся на следующие категории:

- примитивные типы, типы-значения и ссылочные типы
- объектные и интерфейсные типы.

Два основных вида данных в системе типов .NET — это *типы-значения* (value types) и *ссылочные типы* (reference types). Основное различие между ними заключается в том, что тип-значение представляет собой просто последовательность битов в памяти, а ссылочный тип дополнительно обладает "индивидуальностью". Например, 32-битовое знаковое целое является типом-значением. Если мы будем сравнивать два любых целых, то они будут считаться равными, если содержат одинаковое число. С другой стороны, рассмотрим объекты, являющиеся ссылочными значениями. Два разных объекта, представляющие один и тот же класс, могут содержать абсолютно одинаковые данные, но при этом не будут равными, так как указывают на разные участки памяти.

Идея различения типов-значений и ссылочных типов не нова – например, в Симуле 67 делалось явное различие между присваиванием значения объекта с помощью символа :=, и присваиванием ссылки на объект, которое производилось с помощью символа :-



Примитивные типы

Некоторые типы данных используются настолько часто, что компиляторы позволяют обращаться с ними по упрощенной схеме. Например, в C# возможно следующее определение:

```
string s = "Hello C#";
```

Эта форма записи удобна, коротка и читабельна. Однако на самом деле, под таким оператором подразумевается следующее присваивание:

```
System.String s = new System.String();
s = "Hello C#";
```

так как вообще говоря, string является обычным типом, который ничем не отличается от других типов данных. Типы данных, напрямую поддержанные компилятором и допускающие подобные сокращения записи, называются *примитивными типами*. Примитивные типы преимущественно являются типами-значениями и память под них выделяется на стеке данного потока. Большинство примитивных типов различных языков программирования проецируется на типы данных, существующие в базовой библиотеке классов .NET.

Примитивные типы в .NET

.NET поддерживает достаточно большой набор встроенных типов. Только C# поддерживает полный набор примитивных типов .NET. В следующей таблице приведены некоторые встроенные типы данных и объяснено их назначение:

sbyte	System.SByte	Signed 8-bit value
byte	System.Byte	Unsigned 8-bit value
short	System.Int16	Signed 16-bit value
ushort	System.UInt16	Unsigned 16-bit value
int	System.Int32	Signed 32-bit value
uint	System.UInt32	Unsigned 32-bit value
long	System.Int64	Signed 64-bit value
ulong	System.UInt64	Unsigned 64-bit value
char	System.Char	16-bit Unicode character
float	System.Single	IEEE 32-bit float
double	System.Double	IEEE 64-bit float
boolean	System.Boolean	A True/False value
decimal	System.Decimal	96-bit signed integer times 10^0 through 10^{28} (for financial calculations)

Типы-значения

- Не обязательно примитивный тип!
- Содержит само значение, а не ссылку на него (т.е. улучшает производительность)
- Тип-значение не может наследовать от другого типа и быть наследуемым
- При присваивании значение копируется
- Редко передаются в качестве параметров

Типы-значения

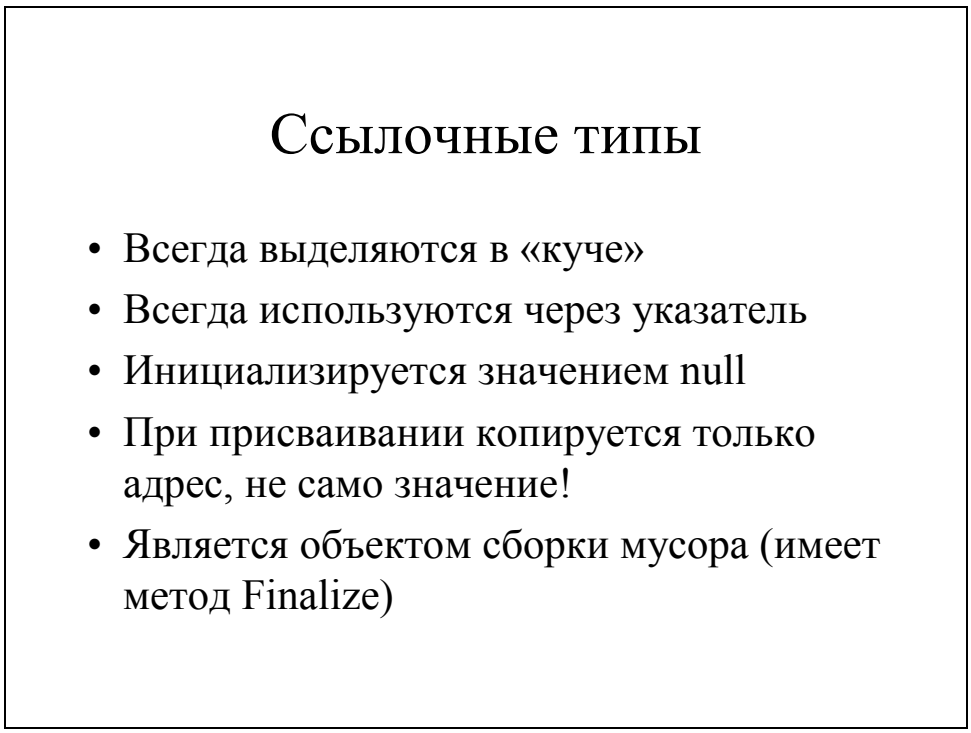
Типы-значения не сводятся к примитивным типам и могут состоять из нескольких переменных. В C# для объявления value type используется ключевое слово struct, как в следующем примере:

```
struct RectVal { public int x, y, cx, cy; }
```

Важно понять, что тип-значение содержит само значение переменной, а не ссылку на него. Это экономит память, так как нет необходимости хранить таблицу виртуальных методов и дополнительный указатель на значение. Кроме того, использование типов-значений улучшает производительность, так как отпадает потребность в лишнем разыменовывании указателя и создании новой копии объекта в "куче". Но с другой стороны, типы-значения имеют множество ограничений. Например, они не могут наследовать от других типов и от них также нельзя ничего унаследовать – они являются *"запечатанными", sealed*. Поэтому можно считать, что типы-значения в целом ведут себя как встроенные типы (кстати, часто рекомендуют не делать типы-значения больше 12-16 байт).

Важно понимать, что при присваивании типов-значений происходит копирование значения, поэтому типы-значения редко используются, если их надо часто передавать в качестве параметров.

Сравнение типов-значений имеет более интересную механику: все типы-значения унаследованы от System.ValueType, который предоставляет такие же методы, как и System.Object, за исключением метода Equals, который выдает True, если значения совпадают, и GetHashCode, который учитывает значение переменной. При создании собственных value types настойчиво рекомендуется переопределять эти методы на свои собственные.



Ссылочные типы

Ссылочные типы представляют собой указатель на ту или иную структуру и потому переменные ссылочных типов всегда выделяются в "куче". При любом обращении к значению ссылочных типов происходит разыменовывание указателя, поэтому они считаютя несколько более тяжеловесными, чем типы-значения. При создании ссылочные типы инициализируются значением null. Попытка обратиться к значению указателя, равного null, приводит к появлению NullPointerException (эта исключительная ситуация не может возникнуть при использовании типов-значений). При присваивании ссылочных типов происходит копирование адреса, а не значения переменной, поэтому изменение значений одной переменной может повлиять на другие, указывающие на тот же объект.

Так как ссылочные типы выделяются в "куче", то после того, как они отрабатывают, они должны быть зачищены сборщиком мусора (напомним, что типы-значения уничтожаются сразу же по выходу из блока или метода, в которых они определены). Для грамотного уничтожения ссылочных типов рекомендуется описывать собственный метод Finalize, который вызывается сборщиком мусора.

В C# ссылочные типы создаются с помощью ключевого слова class:

```
class RectRef {public int x, y, cx, cy; }
```

Упаковка и распаковка

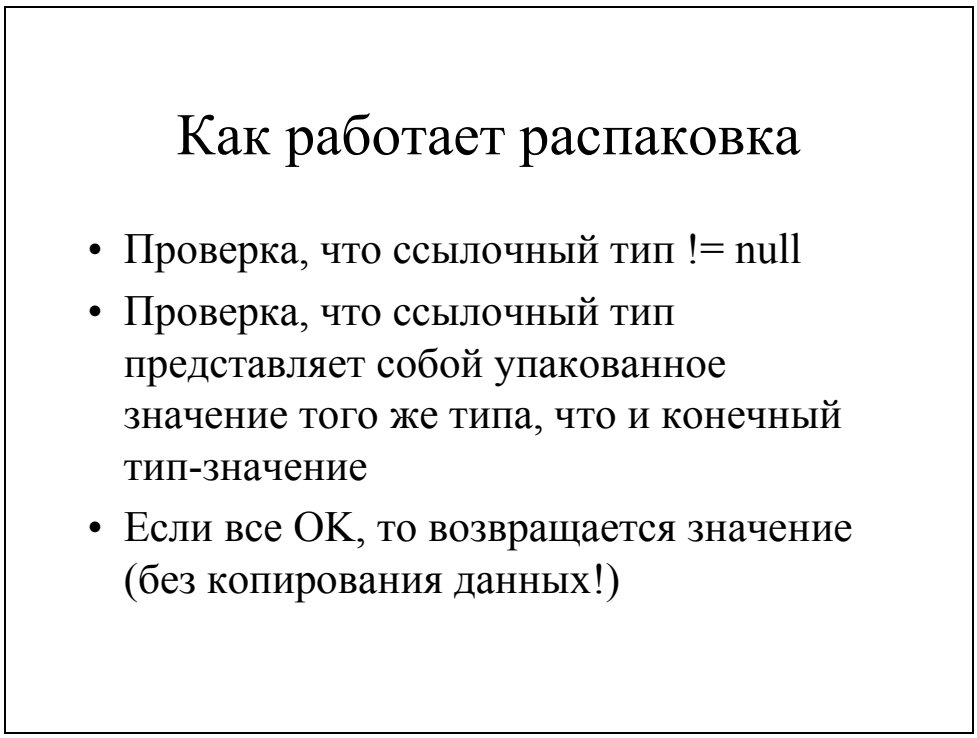
- При присваивании типа-значения в переменную ссылочного типа происходит упаковка (создание нового объекта и копирование значения)
- Обратный процесс называется распаковкой
- Ссылочный тип существует только в упакованной форме
- Тип-значение может быть как упакованным, так и распакованным

Упаковка и распаковка

Зачастую возникает необходимость в интерпретации типа-значения как ссылочного типа. Например, в следующем примере мы добавляем тип-значение в коллекцию:

```
ArrayList a = new ArrayList();
for (int i=0; i < 10; i++) {
  Point p;  // Allocate a Point (not in the heap)
  p.x = p.y = i; // Initialize members in the value type
  a.Add(p); // here we're boxing the value type...
}
```

Для добавления в массив нам необходимо преобразовать значение в ссылочный тип, т.к. метод Add принимает на вход только параметры типа Object. Процесс преобразования типа-значения в ссылочный тип называется *упаковкой (boxing)*. Естественно, существует и обратный процесс, который называется *распаковкой (unboxing)*. Отметим, что ссылочный тип существует только в упакованной форме, а тип-значение может находиться как в упакованной, так и в распакованной форме.



Как работает распаковка

Теперь разберемся с обратной операцией, распаковкой:

1. Проверяется, что исходная ссылочная переменная не равняется null и что она ссылается на значение, полученное упаковкой ожидаемого типа-значения. Если какое-либо из этих условий неверно, то выдается InvalidCallException.
2. Если же типы совпадают, то возвращается указатель на содержимое ссылочного типа (без учета накладных расходов, связанных с организацией объекта).

Важно понимать, что упаковка всегда копирует значение при создании объекта, а распаковка ничего не копирует, а просто возвращает прямую ссылку на само значение (хотя чаще всего результат распаковки все равно куда-нибудь копируется).

На следующем слайде мы рассмотрим пример, иллюстрирующий процесс упаковки и распаковки.

Пример упаковки и распаковки

```
public static void Main() {
  Int32 v = 5; // creating unboxed value type variable
  Object o = v; // o refers to boxed version of v
  v = 123; // changes the unboxed value to 123

  Console.WriteLine (v + “, ” + (Int32) o);
     // displays “123, 5”
}
```

Вопрос: сколько раз в данном примере производится операция упаковки?

Пример упаковки и распаковки

Правильный ответ: операция упаковки производится ровно 3 раза. Дополнительная операция возникает внутри Console.WriteLine, так как оператор ‘+’ означает неявный вызов метода Concat, который ожидает переменные типа Object в качестве параметров. Мы же перед выводом на печать приводим объектную переменную к типу Int32 (т.е. к типу-значению). Для того, чтобы тип-значение мог быть использован в методе Concat, он должен быть приведен обратно в ссылочный вид.

Итак, в данном примере после последнего плюса мы имеем и упаковки, и распаковку. Конечно, это неэффективно, поэтому грамотнее было бы записать последний оператор в следующем виде:

```
Console.WriteLine (v + ", " + o);
```

При этом результаты вывода не изменятся, а эффективность возрастет за счет избавления от лишних операций упаковки и распаковки.

Интересно, что если бы мы не пытались напечатать строку, составленную из нескольких параметров, то лишних операций удалось бы избежать и в примере на слайде, т.к. метод WriteLine может принимать и значения типа Int32.

Упаковка и распаковка в C++

```
#using <mscorlib.dll>
using namespace System;

__value struct V
{
   int i;
};

void Positive(Object*) { };     // expects a managed class

void main()
{
   V v={10};                    // allocate and initialize
   Object* o = __box(v);        // copy to the CLR heap
   Positive( o );               // treat as a managed class
   dynamic_cast<V*>(o)->i = 20; // update the boxed version
}
```

Упаковка и распаковка в C++

Некоторые языки, такие как C# или Visual Basic.NET, поддерживают операции упаковки и распаковки прозрачно для программиста. Это, конечно, хорошо, т.к. упрощает программирование, но как было показано на предыдущем слайде, при недостаточном понимании происходящих "за кадром" процессов это может привести к потере эффективности. Поэтому необходимо вдумчиво подходить к каждому отдельному случаю. Например, иногда выгоднее явно произвести упаковку один раз и затем использовать объектную переменную.

В других языках программирования, например, в Java, данная проблема решена еще проще: все типы данных заведомо представлены только в ссылочной форме, поэтому нет никаких проблем с упаковкой (но при этом имеется потенциальная потеря в скорости выполнения).

Наконец, большинство остальных языков программирования в .NET требуют явной записи для операций упаковки и распаковки. Например, в примере на managed C++, приведенном на слайде, можно увидеть обе операции: __box(v) приводит тип-значение к ссылочному типу, а dynamic_cast<V*>(o) позволяет изменить именно значение переменной i.

Литература к лекции

- Д. Пратт "Знакомство с .NET", Русская редакция, 2001
- J. Richter “Microsoft .NET Framework Delivers the Platform for an Integrated, Service-Oriented Web”, MSDN Magazine, Oct/Nov 2000
- J. Richter “Type Fundamentals”, MSDN Magazine, December 2000
- Т. Арчер "Основы C#", Русская редакция, 2001

Литература к лекции

- Д. Пратт "Знакомство с .NET", Русская редакция, 2001
- J. Richter “Microsoft .NET Framework Delivers the Platform for an Integrated, Service-Oriented Web”, MSDN Magazine, Oct/Nov 2000
- J. Richter “Type Fundamentals”, MSDN Magazine, December 2000
- Т. Арчер "Основы C#", Русская редакция, 2001