Тема 11. Агрегация, области имён и расширенная область видимости

# Парадигмы программирования. C#

# План

- Использование внутренних классов, методов и данных
- Использование агрегации
- Использование областей имён
- Использование модулей и сборок

# Использование внутренних классов, методов и данных

Часть 1

# Для чего нужен внутренний доступ?

- Эффективная система состоит из относительно небольшого числа классов и большого числа объектов
- Эффективность системы можно повысить за счёт организации совместного использования объектов
- Обычных возможностей по ограничению доступа недостаточно
  - Public – открыто для всех
  - Private – открыто только для класса
  - Protected – открыто только для класса и его наследников
- Нужно промежуточное между private и public ограничение доступа
  - Protected не годится

# Внутренний доступ (internal)

- Public – ограничение логическое
  - Где не развернуть public класс – он везде будет public
- Private – ограничение логическое
  - Где не развернуть private класс – он везде будет private
- Internal – ограничение физическое
  - Класс, имеющий модификатор доступа internal доступен только внутри сборки, в которой он развёрнут

# Сравнение internal и friend (C++)

- Доступ friend в C++ используется для получения доступа одного класса к частным (private) членам другого класса
  - Доступ friend закрытый
    - Если классу X нужен доступ к членам класса Y, то только Y может дать доступ к своим внутренним членам для X
  - Доступ friend не рефлексивный
    - Если X является friend для Y, то это не означает, что Y автоматически является friend для X
- Отличия internal от friend
  - Доступ internal открытый
    - Класс можно откомпилировать в модуль и присоединить этот модуль к сборке, что позволит этому классу самостоятельно получить доступ к внутреннему устройству сборки
  - Доступ internal рефлексивный
    - Если X и Y в одной сборке и X имеет internal доступ к Y, то и Y имеет internal доступ к X

# Синтаксис

```
internal class <outername>
{
    internal class <nestedname> { ... }
    internal <type> field;
    internal <type> Method( ) { ... }
    protected internal class <nestedname> { ... }
    protected internal <type> field;
    protected internal <type> Method( ) { ... }
}
```

- protected internal означает protected или internal
  - Можно писать: protected internal
  - А можно писать: internal protected
- Тип без явно указанного модификатора доступа автоматически обозначается protected

# Пример использования доступа internal

```
public interface IBankAccount { ... }

internal abstract class CommonBankAccount
{ ... }

internal class DepositAccount:
    CommonBankAccount, IBankAccount { ... }

public class Bank
{
    public IBankAccount OpenAccount( )
    {
        return new DepositAccount( );
    }
}
```

# Использование агрегации

Часть 2

# Объекты внутри объектов

- Агрегация – это связь между объектами, поясняющая отношение целое-часть
  - Если про отношение между объектами можно сказать «является частью», значит эта связь является агрегацией
- Виды агрегации
  - Агрегация
    - Удаление объекта-части из целого не приводит к разрушению целого
    - Объекты-части могут заменяться другими объектами-частями, один объект-часть может использоваться в нескольких агрегациях
    - Это агрегация по ссылке
  - Композиция
    - Удаление объекта-части из целого приводит к разрушению целого
    - Объекты-части не могут заменяться другими объектами-частями, один объект-часть не может использоваться в нескольких агрегациях
    - Это агрегация по значению
- Все композиции являются агрегациями
  - Если сомнений нет, значит, это композиция
  - Если есть сомнения, значит, это агрегация

# Сравнение агрегации и наследования

- Агрегация
  - Определяет связь между объектами
  - Слабая зависимость между целым и частью
    - Изменение в объекте-части не всегда приводит к изменению в объекте-целом
  - Динамически гибкая
    - Количество элементов в связи может меняться
    - Допустимо перепревязать объект-целое к наследникам объектов-частей
- Наследование
  - Определяет связь между классами
  - Сильная зависимость между наследником и предком
    - Добавили/удалили метод в предке – затрагивает всех потомков
  - Статически негибкая
    - Если класс является наследником другого класса – это навсегда

# Фабрики

- Создание объектов (особенно объектов-целых в агрегациях) часто сопряжено со сложностями
- Многие объекты создаются только в специальных фабриках
- Фабрика инкапсулирует сложное создание объектов
- Фабрика может использоваться и для сложного уничтожения объектов
  - Банковский счёт может удалить только банк, но не клиент
- Фабрики являются часто используемым паттерном при моделировании программного обеспечения
  - Фабричный метод – известный паттерн GoF

# Пример фабрики

```
public class Bank
{
    public BankAccount OpenAccount()
    {
        BankAccount opened = new BankAccount();
        accounts[opened.GetNumber()] = opened;
        return opened;
    }
    private Hashtable accounts = new Hashtable();
}
public class BankAccount
{
    internal BankAccount() { ... }
    public long GetNumber() { ... }
    public void Deposit(decimal amount) { ... }
}
```

# Использование областей имён

Часть 3

# Пересмотр области видимости

- Область видимости имени – это регион в тексте программы, в котором допустимо обращаться к этому имени без дополнений имени

```
public class Bank
{
    public class Account
    {
        public void Deposit(decimal amount)
        {
            balance += amount;
        }
        private decimal balance;
    }
    public Account OpenAccount() { ... }
}
```

# Разрешение столкновений имён

- В огромном проекте, состоящем из нескольких тысяч классов два класса могут случайно получить одинаковые имена

```
// Subsystem A
public class Model { }
// Subsystem B
public class Model { }
```

- Как устранить такое столкновение?
- Можно добавить к именам классов обязательные префиксы
  - Длинные и неуправляемые имена
  - Сложность обслуживания и модификации

```
// Subsystem A
public class SubsystemAModel { }
// Subsystem B
public class SubsystemBModel { }
```

# Объявление областей имён

```
namespace SubsystemA
{
    public class Model { }
}
namespace SubsystemB
{
    public class Model { }
}
```

- Области имён не имеют модификаторов доступа

# Иерархии областей имён

```
namespace Megasoft
{
    namespace DataLayer
    {
        public class Model { }
    }
}

namespace Megasoft.DataLayer
{
    public class Model { }
}
```

# Полные имена

- Полное имя класса включает его область имён
- Сокращённое имя класса (просто имя) может быть использовано только внутри его области видимости

```
namespace SubsystemA
{
    public class Model { }
}
class Test
{
    static void Main()
    {
        // Ошибка компиляции
        Model m1 = new Model();

        // Ошибки нет
        SubsystemA.Model m2 = new SubsystemA.Model();
    }
}
```

- Использование полных имён увеличивает код и делает его трудным для восприятия

# Использование директивы using

- Директива using подключает в текущую область видимости имена типов данных из указанной области имён

```
namespace SubsystemA
{
    public class Model { }
}
…

using SubsystemA;
class Test
{
    static void Main()
    {
        Model m1 = new Model();
    }
}
```

# Подключение вложенных областей имён

```
namespace Megasoft.DataLeyer
{
    public class Model { }
}
...

using Megasoft.DataLeyer;
class Application
{
    static void Main( )
    {
        Model m = new Model( );
    }
}
```

# Использование директивы using в глобальной области видимости

- Директива using должна быть записана до декларирования любых членов в глобальной области видимости

```
class Model {}
// Ошибка компиляции:
// после определения класса
using SystemA;

namespace Megasoft.DataLayer
{
...
}
// Ошибка компиляции:
// после определения области имён
using SystemA;
```

# Использование директивы using внутри области имён

- Допустимо использовать директиву using до определения первого типа

```
namespace Megasoft.DataLayer
{
    using SubsystemA;
    public class Model { ... }
}
```

# Подключения областей имён не являются рекурсивными

```
namespace Megasoft.DataLayer
{
    public class Model { ... }
}
namespace SubsystemB
{
    using Megasoft;

    // Ошибка компиляции
    class NewModel: Model { ... }
}
```

# Столкновения имён при использовании директивы using

```
namespace SubsystemA
{
    public class Model { }
}
namespace SubsystemB
{
    public class Model { }
}
namespace Example
{
    using SubsystemA;
    using SubsystemB;

    class Test
    {
        static void Main()
        {
            // Ошибка компиляции
            Model m = new Model();
        }
    }
}
```

# Использование псевдонимов в директиве using

```
namespace SubsystemA
{
    public class Model { }
}
namespace SubsystemB
{
    public class Model { }
}
namespace Example
{
    using SubsystemA;
    using SubsystemB;
    using MainModel = SubsystemA.Model;
    using B = SubsystemB;

    class Test
    {
        static void Main()
        {
            MainModel m1 = new MainModel();
            B.Model m2 = new B.Model();
        }
    }
}
```

- Допустимо создавать псевдонимы для типов и областей имён

# Правила именования областей имён

- Для обозначения логических компонентов следует использовать PascalCase
  - Megasoft.SubsystemA
- Начальная часть области имён должна соответствовать имени компании или известному бренду
  - Megasoft
  - Ifmo
- Придерживайтесь использования имён существительных во множественном числе
  - Megasoft.Arrays
- Избегайте столкновений имён классов и областей имён

# Использование модулей и сборок

Часть 4

# Создание модулей

- Файлы исходного кода (*.cs) могут быть откомпилированы в управляемые модули (*.netmodule)
- csc /target: module Test.cs
- Visual Studio не поддерживает создание модулей

# Использование сборок

- Сборка – это группа взаимодействующих типов данных, выделенная в отдельный файл откомпилированного кода, снабжённый ресурсами и манифестом
  - Особенности сборок: повторное использование кода, версионность, возможность безопасного развёртывания
  - Физически сборка – это dll или exe
  - Манифест – это метаданные сборки (данные о данных)
  - Ресурсы – это любые данные, необходимые для работы программы (текст, изображения, звук и т.п.)
- Контроль доступа на уровне сборки
  - internal

# Манифест сборки

- Уникальное имя сборки
  - Текстовое имя
  - Номер версии
  - Культура (опционально при наличии локализованных ресурсов)
- Информация о содержимом
  - Данные о типах и ресурсах, содержащихся в сборке
- Зависимости
  - Список сборок, от которых зависит эта сборка

# Однофайловые и многофайловые сборки

- Однофайловая сборка состоит из единственного файла dll или exe
- Многофайловая сборка состоит из нескольких файлов
  - Файлы dll или exe, содержащие манифест
  - Файлы netmodule
  - Файлы ресурсов
- Причины использования многофайловых сборок
  - Возможность скачивания из Интернета по частям
  - Возможность объединять сборки для разных культур
  - Возможность объединять сборки от разных программистов

# Создание сборок

- Создание однофайловой сборки
  - csc /target:library /out:Test.dll Test.cs Data.cs
- Создание многофайловой сборки
  - csc /t:library /addmodule:Test.netmodule /out:Test.dll Data.cs
- Проверка содержимого сборки
  - ildasm Test.dll
- Дополнительная информация
  - http://msdn.microsoft.com/en-us/library/b0b8dk77.aspx
  - http://msdn.microsoft.com/en-us/library/ceats605.aspx
  - http://www.red-gate.com/products/reflector/

# Сравнение областей имён и сборок

- Области имён: механизм логического именования
  - Классы из одной области имён могут располагаться в разных сборках
  - Классы из нескольких областей имён могут располагаться в одной сборке
- Сборка: механизм физической группировки
  - В сборке хранится MSIL и манифест
  - Модули (netmodule) и ресурсы могут быть внешними ссылками

# Версии

- Каждая сборка имеет номер версии, являющийся частью уникального имени сборки
- Номер версии состоит из четырёх частей
  - Major version
  - Minor version
  - Build number
  - Revision
- Пример: 2.5.34.234

# Сборки со строгим именем

- Сборки со строгим именем предназначены для устранения проблемы DLL HELL
- Особенности сборок со строгим именем
  - Эти сборки подписаны с помощью закрытого ключа электронной подписи, поэтому они содержат открытый ключ
  - Наличие открытого ключа и уникального имени делает сборку уникальной во всём мире
  - Никто в мире не может выдать себя за издателя сборки со строгим именем
  - Никто в мире кроме создателя сборки не может выпустить её новую версию
  - Наличие строгого имени гарантирует сохранность целостности сборки, т.е. по дороге к клиенту сборка не может быть модифицирована
- Дополнительная информация
  - http://msdn.microsoft.com/en-us/library/wd40t7ad.aspx

# Глобальный кэш сборок

- Global assembly cache (GAC) глобальный кэш сборок – глобальное хранилище совместно используемых сборок на машине
- В GAC можно класть только сборки со строгим именем
- Дополнительная информация
  - http://msdn.microsoft.com/en-us/library/yf1d93sz.aspx

# Заключение

- Спасибо за внимание!