**Практика 13. АГРЕГАЦИИ, ПРОСТРАНСТВА ИМЕН, СБОРКИ И МОДУЛИ**

**1.** Создать новый класс, который будет являться фабрикой объектов класса банковский счет. Изменить модификатор доступа у конструкторов класса банковский счет на internal. Добавить в фабричный класс перегруженные методы создания счета CreateAccount, которые бы вызывали конструктор класса банковский счет и возвращали номер созданного счета. Использовать хеш-таблицу для хранения всех объектов банковских счетов в фабричном классе. В фабричном классе предусмотреть метод закрытия счета, который удаляет счет из хеш-таблицы (методу в качестве параметра передается номер банковского счета).

Использовать утилиту ILDASM для просмотра структуры классов.

**2.** Разбить созданные классы, связанные с банковским счетом, и тестовый пример в разные исходные файлы. Разместить классы в одно пространство имен и создать сборку. Подключить сборку к проекту и откомпилировать тестовый пример со сборкой. Получить исполняемый файл, проверить с помощью утилиты ILDASM, что тестовый пример ссылается на сборку и не содержит в себе классов, связанный с банковским счетом.

**3.** Для реализованного класса из практики 9, № 4 создать новый класс Creator, который будет являться фабрикой объектов класса здания.

Для этого изменить модификатор доступа к конструкторам класса, в новый созданный класс добавить перегруженные фабричные методы CreateBuild для создания объектов класса здания. В классе Creator все методы сделать статическими, конструктор класса сделать закрытым. Для хранения объектов класса здания в классе Creator использовать хеш-таблицу. Предусмотреть возможность удаления объекта здания по его уникальному номеру из хеш-таблицы.

Создать тестовый пример, работающий с созданными классами.

**4.** Разбить созданные классы (здания, фабричный класс) и тестовый пример в разные исходные файлы. Разместить классы в одном пространстве имен. Создать сборку (DLL), включающие эти классы. Подключить сборку к проекту и откомпилировать тестовый пример со сборкой. Получить исполняемый файл, проверить с помощью утилиты ILDASM, что тестовый пример ссылается на сборку и не содержит в себе классов здания и Creator.

## Использование внутренних (internal) классов, методов и данных

Модификаторы доступа определяют возможность доступа к элементам класса, таким как методы и свойства. При разработке класса необходимо явно указывать модификатор доступа у каждого члена класса. Модификаторы доступа:

* **Public** – элементы доступны всюду внутри области видимости;
* **Protected** – элементы доступны внутри класса и у всех потомков класса;
* **Private** – элементы доступны только внутри класса;
* **Internal** – элементы доступны внутри одной сборки Microsoft.NET (одного исполняемого файла или одного .dll файла), из других сборок доступ запрещён;
* **Protected internal** – элементы доступны внутри классов-потомков и во всех классах внутри одной сборки.

Public – слишком открытый доступ, private – слишком закрытый, protected

* открыт только для потомков класса. Для создания промежуточного типа доступа был создан внутренний (internal) доступ. Типы доступа public и private – логические, то есть не зависят от физического размещения класса. Для internal классов или элементов размещение определяет область доступа: если класс находится в исполняемом файле, то он доступен для всех классов файла; если класс относится к конкретной сборке, то доступ разрешен только внутри данной сборки. В исполняемом файле может быть несколько сборок, тогда доступ к internal классу из другой сборки будет запрещен.

Когда класс определен как internal, доступ к нему имеют все классы из сборки. Классы и элементы protected internal доступны всей сборке, а также всем потомкам текущего класса.

Ограничения на использование модификаторов доступа:

* 1. Вне классов и пространств имен нельзя объявлять классы protected и private, эти типы доступа можно использовать только внутри какого-либо класса. Для элементов внутри класса модификатором доступа по

умолчанию будет private. В С# есть возможность внутри одного класса объявлять другой класс.

* 1. При объявлении типа в глобальной области видимости можно использовать модификаторы доступа public и internal. В глобальной области видимости или в пространстве имен модификатор доступа по умолчанию будет internal.

## Использование агрегаций

Агрегации используются для группировки объектов вместе в иерархию объектов, которую можно неоднократно использовать. Агрегации определяют связь целое/часть между объектами, не классами. В одном случае в этой связи время жизни «целого» и «части» могут быть не связаны. В этом случае агрегация называется агрегацией по ссылке. В случае, когда время жизни

«целого» и «части» связаны друг с другом, агрегация называется агрегацией по значению. В агрегации «целое» – это всего лишь класс, который используется для группировки «частей»-классов в единое целое, т.е. «целый» класс реально не существует. Например, что такое компьютер? Это всего лишь имя, которое используется для описания конкретных частей: CPU, монитор, клавиатура и т.д. Через это имя можно получить доступ к методам, которые работают с составными частями.

Сравнение агрегаций и наследования:

* + - Агрегации определяют связь на уровне объектов, наследование – на уровне классов. В случае агрегации у целого объекта может быть несколько объектов частей одного типа. Например, у агрегации компьютер может быть несколько мониторов, несколько процессоров. В случае наследования речь идет не о конкретных объектах, а о классах. В этом случае компьютер и монитор – это просто два разных класса, никак не связанные друг с другом.
    - В агрегации при изменении методов «частей» методы «целого» автоматически не меняются. В наследовании же при изменении методов у предков, меняются и потомки.
    - В агрегации в объект «целого» можно добавлять и удалять объекты

«части» без каких-либо ограничений. В главном объекте сохраняются

ссылки на части, при необходимости они могу изменяться, можно создавать новые, удалять старые. В механизме наследования все статично: класс потомок всегда будет оставаться потомком.

## Фабрики классов

Иногда бывает необходимо запретить создание объекта класса напрямую, разрешить создавать объекты только определенным объектам некоторого другого класса. Например, самостоятельно нельзя открыть счет в банке. Чтобы открыть счет, нужно пойти в банк и банковский работник открывает счет. В программировании, пусть есть два класса Bank и BankAccount. Открывать счет в банке – создавать объект класс BankAccount можно только внутри банка, вызвав внутри класса Bank нужный конструктор:

public class Bank

{

public BankAccount OpenAccount( )

{

BankAccount opened = new BankAccount( ); accounts[opened.Number( )] = opened; return opened;

}

private Hashtable accounts = new Hashtable( );

}

public class BankAccount

{

internal BankAccount( ) { ... } public long Number( ) { ... }

public void Deposit(decimal amount) { ... }

}

В примере для хранения всех открытых в банке счетов внутри класса Bank предусмотрено поле accounts в виде хэш-таблицы. Каждый открытый банковский счет сохраняется в этой хэш-таблице. В данном случае класс Bank будет являться фабрикой. Для того чтобы конструктор класса BankAccount был доступен внутри класса Bank, его объявили с модификатором доступа internal.

## Пространства имен

Рассмотрим пример

public class Bank

{

public class Account

{

public void Deposit(decimal amount)

{

balance += amount;

}

private decimal balance;

}

public Account OpenAccount( ) { ... }

}

В этом примере четыре области видимости:

* + - глобальная область – в ней располагается класс Bank;
    - область класса Bank – в ней класс Account и метод OpenAccount;
    - область класса Account – в ней метод Deposit и поле balance;
    - область метода Deposit – в ней объявляется параметр amount.

Если объект или метод находится вне области видимости или его имя скрыто, то вызвать его можно только при помощи полного пути.

class Top

{

public void M(){…}

}

class Bottom:Top

{

new public void M()

{

M(); // рекурсия

base.M(); //обращение к скрытому методу

}

}

При обращении к элементам базового класса используется ключевое слово base. Точно так же и для полей класса:

public struct Point

{

public Point(int x, int y)

{

this.x = x; this.y = y;

}

private int x,y;

}

В сложных проектах, в которых используется большое количество классов, когда над проектом работает несколько программистов, возможна ситуация, что в одной области видимости окажутся два класса с одинаковым именем. В таком случае следует использовать пространства имен (namespaces). Пространства имен разделяют большой проект на несколько подсистем, в каждой подсистеме – своя глобальная область видимости. Рекомендуется внутри одного пространства имен размещать логически связанные друг с другом элементы.

namespace Outer

{

namespace Inner

{

class Widget{…}

}

}

В С# можно записать предыдущий класс короче.

namespace Outer.Inner

{

class Widget{…}

}

Для того чтобы использовать класс из другого пространства имен, необходимо использовать полное его имя или подключить соответствующее пространство имен, используя директиву using. Например, для обращения к классу Widget из предыдущего примера из другого пространства имен можно написать

Outer.Inner.Widget w; //полный путь к классу Widget или

using Outer.Inner; //подключаем пространство имен

Widget w; //объявляем объект класса Widget без указания полного пути к классу.

Директива using должна располагаться перед объявлением каких-либо элементов либо в глобальной области видимости, либо внутри пространства имен. Директива разрешает доступ по имени к классам только данного пространства имен, то есть доступ к классам вложенных пространств имен по короткому имени запрещен.

namespace Microsoft.PowerPoint

{

public class Widget { ... }

}

namespace VendorB

{

using Microsoft; // но не Microsoft.PowerPoint

class SpecialWidget: Widget { ... } // ошибка на этапе компиляции

}

Кроме того, если подключить при помощи директивы два пространства имен с несколькими классами с одним именем, то при вызове этого класса получим ошибку компиляции.

namespace Test

{

using VendoreA; // содержит класс Widget

using VendoreB; // также содержит класс Widget

// Здесь ошибки нет

class Application

{

static void Main()

{

Widget w = new Widget(); // ошибка при компиляции

}

}

}

Ошибка возникает, так как компилятор не знает, какой класс нужен. Надо явно прописать необходимое пространство имен, например

VendoreA.Widget w = new VendoreA.Widget();

Директиву using можно использовать для объявления коротких имён классов напрямую.

using Widget = VendoreA.SuiteB.Widget

В этом случае можно сразу обращаться по короткому имени к данному классу. Все ограничения на использование директивы остаются. Директиву using можно использовать в обеих её возможностях одновременно, единственное, что надо помнить, что действие директивы начинается с первого элемента пространства имен в которой она объявлена, то есть друг на друга директивы не действуют, и следующая конструкция даст ошибку:

using System;

using TheConsole = Console; //ошибка, хотя Console класс из System

Правильно будет:

using System;

using TheConsole = System.Console;

## Модули и сборки

В С# присутствует возможность компилировать исходные .cs файлы в управляемый модуль – промежуточный язык MSIL, который содержит метаданные для описания модуля. Для этого в опциях компилятору надо задать параметр /target:module имя\_файла.cs. При выполнении программы управляемый модуль преобразовывается в машинный код. При компиляции исходный файл компилируется в управляемый модуль с расширением

.netmodule. Команда командной строки:

csc /target:module Bank.cs

На выходе получим файл Bank.netmodule.

Выполняемый файл может запускать модули только в составе какой-либо сборки. В сборке физически размещается группа взаимодействующих классов. Классы, находящиеся в одной сборке, имеют доступ к internal элементам друг друга. Для классов из других сборок доступ к этим элементам закрыт.

Сборка – это многократно-используемый, безопасный, и самоописываемый модуль, содержащий типы и ресурсы с поддержкой версий;

это первичный стандартный блок приложения .NET. Сборка состоит из двух логических частей:

* наборов типов и ресурсов, которые формируют некоторый логический модуль из функциональных возможностей,
* метаданные, которые описывают, как эти элементы связаны и от чего зависят их правильная работа.

Метаданные, которые описывают сборку, называют манифестом. В манифесте содержится следующая информация:

* Уникальность. Уникальность сборки включает в себя простое текстовое название, номер версии и дополнительный открытый ключ, который гарантирует уникальность названия и защищает от нежелательного использования.
* Содержание. Сборка содержит типы и ресурсы. Манифест перечисляет названия всех типов и ресурсов, которые видимы извне сборки. Содержит информацию о том, где они могут быть найдены во время трансляции.
* Ссылки. Каждая сборка явно описывает другие сборки, от которых она зависит.

В простейшем случае, сборка состоит из одного файла, который содержит код, ресурсы, метаданные и манифест. В более общем случае, сборка состоит из нескольких файлов, тогда сборка существует как автономный файл или содержится в одном из файлов PE, которые содержат типы и ресурсы.

Соответствующие команды компилятора:

сsc /target:library /out:Bank.dll Bank.cs Account.cs

сsc /t:library /addmodule:Account.netmodule /out:Bank.dll Bank.cs

Каждая сборка имеет свой уникальный номер версии. Две сборки, которые отличаются только номером, для исполняющей среды различны. Номер версии состоит из четырех частей:

старшая версия . младшая версия . номер компиляции . редакция Например, номер 1.5.12540.0 означает, что старшая версия 1, младшая 5,

номер компиляции 12540 и редакция 0.

Пространства имен – это логический механизм компиляции, он обеспечивает структуру имен сущностей исходного кода. При выполнении программы пространства имен не рассматриваются. Сборки – это физический

механизм выполнения, обеспечивающий структуру компонент при исполнении программы. Можно разместить классы одного пространства имен в разных сборках, также в одной сборке могут быть классы разных пространств имен. В сборку можно подключать внешние модули, тогда в сборке записывается именованная ссылка на управляемый модуль.