

Лабораторная работа

9. Наследование

Варианты заданий

Выполнить задания 1-4 по предложенным вариантам:

	№ Задания			
	1	2	3	4
Вариант 1	1.V	2.П	3.П	4.I
Вариант 2	1.IV	2.П	3.I	4.П
Вариант 3	1.I	2.П	3.П	4.П
Вариант 4	1.П	2.П	3.I	4.П
Вариант 5	1.П	2.П	3.П	4.I
Вариант 6	1.I	2.П	3.П	4.П
Вариант 7	1.П	2.П	3.П	4.П
Вариант 8	1.IV	2.П	3.П	4.I
Вариант 9	1.VI	2.I	3.П	4.П
Вариант 10	1.П	2.П	3.I	4.П

Для всех заданий разработать Windows-приложения, демонстрирующие работу классов. Описание классов реализовать в DLL.

Задание 1.

Разработать иерархию классов для игровых юнитов (см. варианты). В программе должно быть реализовано:

- перегрузка метода родительского класса (2 класса-наследника с разными реализациями);
- защищенные (protected) общие поля и методы в родительском классе;
- перегрузка свойства, сокрытие имени метода, доступ к родительскому полю;
- виртуальный метод;
- вызов конструктора базового класса;
- создание родительских и дочерних объектов;
- создание копии объекта.

При запуске приложения пользователь может создать несколько юнитов различных типов (тип выбирается из общего списка). Созданные юниты и здания добавляются в таблицу DataGridView (без возможности редактирования, с параметром

SelectionMode = FullRowSelect – выделяет всю строку при нажатии). Объекты одного типа отличаются по номерам.

Экземпляры классов после создания сериализовать (сохранить в файл, считать из файла при запуске программы).

Конечные параметры юнитов приведены в таблице: (HP – здоровье, А – атака, Б – броня, Д – дальность, СЗ – стоимость в ед. золота)

	HP	А	Б	Д	СЗ	Мана
Knight	90	2-12	4	1	800	
Paladin	90	2-12	4	1	800	250
Destroyer	100	2-35	10	4	700	
TrollAxethrower	40	3-9	2	4	500	
TrollBerserker	50	3-11	2	4	500	
Archer	40	3-9	2	4	500	
Ranger	50	3-11	2	4	500	
Gryphon	100	8-16	0	4	2500	
Dragon	100	8-16	0	4	2500	

Здания:

	HP	А	Б	Д	СЗ
TownHall / GreatHall	1200				1200
Keep / Stronghold	1400				2000
Castle / Fortress	1600				2500
WatchTower	100				550
GuardTower	130	6-16	20	6	500
CannonTower	160	10-50	20	7	1000

Уровни классов (классы 0-го и 1-го уровней – абстрактные):

0

- 1
 - 2
 - 3
 - 4




Для выбранного в таблице юнита реализовать возможность изменить здоровье/скорость/броню/ману (на фиксированную величину) при нажатии на соответствующую кнопку.

Здания изменяются путем последовательного улучшения (согласно иерархии, см. варианты). Т.е., например, создать экземпляр класса CannonTower сразу нельзя, для этого необходимо создать объект WatchTower, «улучшить» его до GuardTower, только затем до CannonTower.

Можно ввести дополнительные свойства для выполнения условий задания и реализации полиморфизма.

I Иерархия классов:

Object

- Unit
 -  Knight (рыцарь)
 -  Paladin (паладин)
 -  Destroyer (разрушитель)




II Иерархия классов:

Object

- Building (здание)
 - TownHall (Ратуша)
 - Keep (Дворец)
 - Castle (Замок)

III Иерархия классов

Object

- Unit
 -  TrollAxethrower (тролль)
 -  TrollBerserker (берсерк)
 -  Dragon (дракон)




IV Иерархия классов

Object

- Building
 - GreatHall (Зал вождей)
 - Stronghold (Цитадель)
 - Fortress (Крепость)

V Иерархия классов

Object

- Unit
 -  Archer (лучник)
 -  Ranger (рейнджер)
 -  Gryphon (грифон)

VI Иерархия классов

Object

- Building (здание)
 - WatchTower (Смотровая башня)
 - GuardTower (Дозорная башня)
 - CannonTower (Орудийная башня)

Задание 2.

I Постройте семейство классов Reader, Author, Book, Library и абстрактного класса Person, связанных отношениями наследования и вложенности.

II Постройте семейство классов Employee, Boss, Department, Firm и абстрактного класса Person, связанных отношениями наследования и вложенности.

III Постройте семейство классов Car, OwnerOfCar, Parking и абстрактного класса Person, связанных отношениями наследования и вложенности.

Задание 3.

I Разработать классы Gun и Camera, наследуемые от интерфейса IShootable с методами: Aim (прицелиться/сфокусироваться), Shoot (снять/выстрелить), Reload (перезарядить); и свойством: Stock (число патронов/снимков).

II Разработать классы iPhone и Toaster, наследуемые от интерфейса IPowerable с методами: включить, выключить, зарядить; и свойствами: уровень заряда, заряжено.

III Разработать классы `Cursor` и `Box`, наследуемые от интерфейса `IMovable` с методами: переместить вверх, вниз, вправо, влево; и свойством «положение в пространстве».

Задание 4.

Создать несколько объектов для разработанного класса, перечислить их и сравнить.

- I Разработать класс `iPad`, наследующий интерфейсы `Comparable` и `Enumerable`.
- II Разработать класс `Shuttle`, наследующим интерфейсы `Comparable` и `Enumerable`.
- III Разработать класс `Pumpkin`, наследующим интерфейсы `Comparable` и `Enumerable`.

Методические указания

Оглавление

9. Наследование	1
Варианты заданий	1
Методические указания	5
9.1. Отношения вложенности и наследования	5
9.1.1. Отношение вложенности	6
9.1.2. Наследование	8
9.1.3. Атрибут <code>Serializable</code>	17
9.2. Интерфейсы	20
9.2.1. Интерфейс <code>IEquatable<T></code>	22
9.2.2. Интерфейс <code>Comparable</code>	23
9.2.3. Интерфейс <code>Enumerable</code>	24

9.1. Отношения вложенности и наследования

Классы программной системы находятся в определенных отношениях друг с другом. Два основных типа отношений между классами определены в ОО-системах. Первое отношение, "клиенты и поставщики", называется часто клиентским отношением или отношением вложенности (встраивания). Второе отношение, "родители и наследники", называется отношением наследования.

Классы `A` и `B` находятся в отношении "клиент - поставщик", если одним из полей класса `B` является объект класса `A`. Класс `A` называется поставщиком класса `B`, класс `B` называется клиентом класса `A`. Следуя этому определению, объект класса `A` "вложен" в класс `B`. По этой причине отношение "клиент - поставщик" называют также отношением вложенности или встраивания. Заметим сразу, что помимо вложенности поля, могут существовать и

другие способы взаимодействия двух классов, связывающие их отношением "клиент - поставщик".

Классы А и В находятся в отношении "родитель - наследник", если при объявлении класса В класс А указан в качестве родительского класса. Класс А называется родителем класса В, класс В называется наследником класса А.

9.1.1. Отношение вложенности

Рассмотрим два класса Provider и Clent, связанных отношением вложенности. Оба класса применяются для демонстрации идей и потому устроены просто, не неся особой смысловой нагрузки. Пусть класс поставщик Provider уже построен. У класса три поля, одно из которых статическое, два конструктора, два метода - один статический и один динамический:

```

/// <summary>
/// Класс поставщик,предоставляет клиентам
/// статический и экземплярный методы,
/// закрывая поля класса
/// </summary>
class Provider
{
    //fields
    string fieldP1;
    int fieldP2;
    static int fieldPS;
    //Конструкторы класса
    /// <summary>
    /// Конструктор с аргументами
    /// </summary>
    /// <param name="p1">аргумент,инициализирующий поле класса</param>
    /// <param name="p2">аргумент,инициализирующий поле класса</param>
    public Provider(string p1, int p2)
    {
        fieldP1 = p1.ToUpper(); fieldP2 = p2*2;
        fieldPS = 0;
    }
    public Provider()
    {
        fieldP1 = ""; fieldP2 = 0; fieldPS = 0;
    }
    //Динамический (Экземплярный) метод
    public string MethodPD()
    {
        fieldPS++;
        string res = "Объект класса Provider" + "\n\r";
        res += string.Format("Мои поля: поле1 = {0}, поле2 = {1}",
            fieldP1, fieldP2);
        return res;
    }
    // Статический (Модульный) метод
    public static string MethodPS()
    {
        string res = "Модуль класса Provider" + "\n\r";
        res += string.Format("Число вызовов метода MethodPD = {0}",
            fieldPS.ToString());
        return res;
    }
}

```

Поля класса закрыты для клиентов. У класса есть конструктор без аргументов, инициализирующий поля класса соответствующими константами, и конструктор с аргументами, который преобразует переданные ему значения, прежде чем записать их в поля класса. Методы класса позволяют получить информацию, хранящуюся в полях. Динамический (экземплярный) метод MethodPD, которому доступны поля класса, хранимые экземплярами класса, возвращает строку с информацией о хранимых значениях в полях. Одновременно этот метод увеличивает значение, хранимое в статическом поле, которое можно рассматривать как счетчик общего числа вызовов динамического метода всеми объектами данного класса. Статический метод MethodPS, которому доступно только статическое поле, возвращает в качестве результата строку с информацией о числе вызовов динамического метода.

Построим теперь класс Client - клиента класса Provider. Класс будет устроен похожим образом. Существенное дополнение состоит в том, что одним из полей является объект provider класса Provider:

```

/// <summary>
/// Клиент класса Provider
/// </summary>
class Client
{
    //fields
    Provider provider;
    string fieldC1;
    int fieldC2;

    const string NEWLINE = "\n\r";
    //Конструкторы класса
    public Client(string p1, int p2, string c1, int c2)
    {
        fieldC1 = c1.ToLower(); fieldC2 = c2-2;
        provider = new Provider(p1,p2);
    }
    public Client()
    {
        fieldC1 = ""; fieldC2 = 0;
        provider = new Provider();
    }
    /// <summary>
    /// Метод, использующий поле класса provider
    /// для работы с методами класса Provider
    /// </summary>
    /// <returns>композиция строк провайдера и клиента </returns>
    public string MethodClient1()
    {
        string res = provider.MethodPD() + NEWLINE;
        res += "Объект класса Client" + NEWLINE;
        res += string.Format("Мои поля: поле1 = {0}, поле2 = {1}",
            fieldC1, fieldC2);
        return res;
    }
}

```

Обратите внимание: конструкторы клиента (класса Client) создают объект поставщика (класса Provider), вызывая конструктор поставщика. Для создания объектов поставщика могут требоваться аргументы, поэтому они передаются конструктору клиента, как это сделано в нашем примере.

Создавая объект класса `Client`, конструкторы этого класса создают и объект класса `Provider`, связывая его ссылкой с полем `provider`. Все динамические методы клиентского класса могут использовать этот объект, вызывая доступные клиенту методы и поля класса поставщика. Метод класса `Client` - `MethodClient1` начинает свою работу с вызова: `provider.MethodPD()`, вызывая сервис, предоставляемый методом класса `Provider`.

Как правило, для клиентов закрываются поля класса. Клиенты класса получают доступ к методам класса поставщика - совокупность этих методов определяет интерфейс класса поставщика и те сервисы, которые поставщик предоставляет своим клиентам. Интерфейс класса составляют методы с модификатором доступа `public` или `internal` для дружественных классов.

Построим тест, проверяющий работу с объектами классов `Provider` и `Client`:

```
class Program
{
    static void Main(string[] args)
    {
        Client client =
            new Client("object of class Provider", 33,
                      "object of class Client", 44);
        string res = client.MethodClient1();
        Console.WriteLine(res);
    }
}
```

9.1.2. Наследование

Мощь ООП основана на наследовании. Когда построен полезный класс, он может многократно использоваться клиентами этого класса. Повторное использование – это одна из главных целей ООП.

Класс потомок наследует все возможности родительского класса – все поля и все методы, открытую и закрытую часть класса, статическую и динамическую части класса. Потомок может не иметь прямого доступа ко всем наследуемым полям и методам. Поля и методы родительского класса, снабженные атрибутом `private`, хотя и наследуются, но являются закрытыми, и методы, создаваемые потомком, не могут к ним обращаться напрямую, а только через методы, наследованные от родителя. Хорошей стратегией при проектировании класса является использование модификатора доступа `protected` вместо модификатора `private`, разрешая потомкам класса прямой доступ ко всем полям и методам родительского класса.

Потомок наследует почти все, но не все. Он не наследует конструкторы родительского класса. Конструкторы потомок должен создавать сам.

Построение родительского класса Found

Рассмотрим класс, названный `Found`, который в наших примерах будет играть роль родительского класса:

```
/// <summary>
/// Родительский класс
/// </summary>
class Found
{
    //fields
```



```

protected string name;
protected int credit;
static protected int count;
const string NL = "\r\n";
//Constructors
public Found()
{
    name = "Nemo";
    credit = 0;
    count++;
}
public Found(string name, int credit)
{
    this.name = name;
    this.credit = credit;
    count++;
}
}

```

У класса `Found` три поля. Поля закрыты для клиентов класса, но открыты для потомков. Это правильная стратегия. Потомкам следует разрешать прямой доступ к полям. Одно из полей является статическим, содержательно оно будет использоваться для подсчета числа созданных объектов класса `Found` клиентами класса. Как и положено, помимо конструктора с аргументами, передаваемыми для инициализации экземплярных полей класса, у класса есть конструктор без аргументов, инициализирующий поля класса некоторым заданным по умолчанию способом. Оба конструктора в процессе работы увеличивают на единицу значение статического поля, которое к этому времени уже создано и инициализировано, поскольку статический конструктор класса вызывается по умолчанию до того, как вызываются конструкторы, создающие экземпляры – объекты класса.

Потомок класса, наследовав от родительского класса какой-либо метод, может его переопределить, задав собственную реализацию, отличную от реализации, которая используется родителем. Переопределяемый метод родителя должен снабжаться модификатором `override`.

Хотя явно для класса `Found` родительский класс не задан, но родитель есть у каждого класса. Если прямой родитель не задан, то таковым является класс `object`. Класс `Found` наследовал от своего родителя – класса `object` ряд методов. Чаще всего методы, наследуемые от `object`, потомок должен переопределить, и в первую очередь это касается метода `ToString`. Как правило, переопределение этого метода сводится к тому, что возвращаемая методом строка содержит информацию о значениях, хранящихся в полях класса. Вот как выглядит переопределяемый метод `ToString` для класса `Found`:

```

public override string ToString()
{
    string s = "Поля: name = {0}, credit = {1}";
    return String.Format(s, name, credit);
}

```

Начнем добавлять в класс `Found` собственные методы:

```

public string NonVirtMethod()
{
    return "Found: " + this.ToString();
}

```

Это обычный метод класса. Он возвращает некоторую строку, которая получена конкатенацией константы и строки, являющейся результатом вызова только что

переопределенного метода `ToString`. Имя метода уточнено именем текущего объекта `this`, чтобы подчеркнуть тот факт, что именно текущий объект вызывает метод класса `ToString`.

Добавим в класс еще один похожий метод:

```
public virtual string VirtMethod()  
{  
    return "Found: " + this.ToString();  
}
```

Тела методов, как видите, ничем не отличаются. Но! В заголовок второго метода добавлено ключевое слово `virtual`. И хотя для объектов класса `Found` вызов обоих методов будет давать одинаковый результат, с позиций наследования эти методы отличаются существенным образом. Добавим еще один метод класса:

```
public string Parse()  
{  
    return "Выполнен разбор кода!";  
}
```

Метод прост и бесхитроsten – возвращает некоторую строку. Реально в этом примере строятся методы, называемые заглушками. Они не выполняют никакой содержательной работы, но выдают информацию о том, что метод проработал.

Рассмотрим теперь чуть более сложный метод:

```
public string Job()  
{  
    string res = "";  
    res += "VirtMethod: " +  
        VirtMethod() + NL;  
    res += "NonVirtMethod: " +  
        NonVirtMethod() + NL;  
    res += "Parse: " +  
        Parse() + NL;  
    return res;  
}
```

Метод `Job` поочередно вызывает методы класса – `VirtMethod`, `NonVirtMethod`, `Parse`. Строки, задающие результаты этих методов, соединяются, и полученная строка возвращается в качестве результата метода `Job`.

Последний метод, который добавим в класс `Found`, – это статический метод, возвращающий информацию из статического поля класса:

```
public static string NumberOfObjects()  
{  
    return "Объектов создано: " + count;  
}
```

Для тестирования наследования создадим Windows-проект. Покажем, как выглядит спроектированная форма для тестирования работы с объектами класса `Found`. Ее вид приведен на рис.

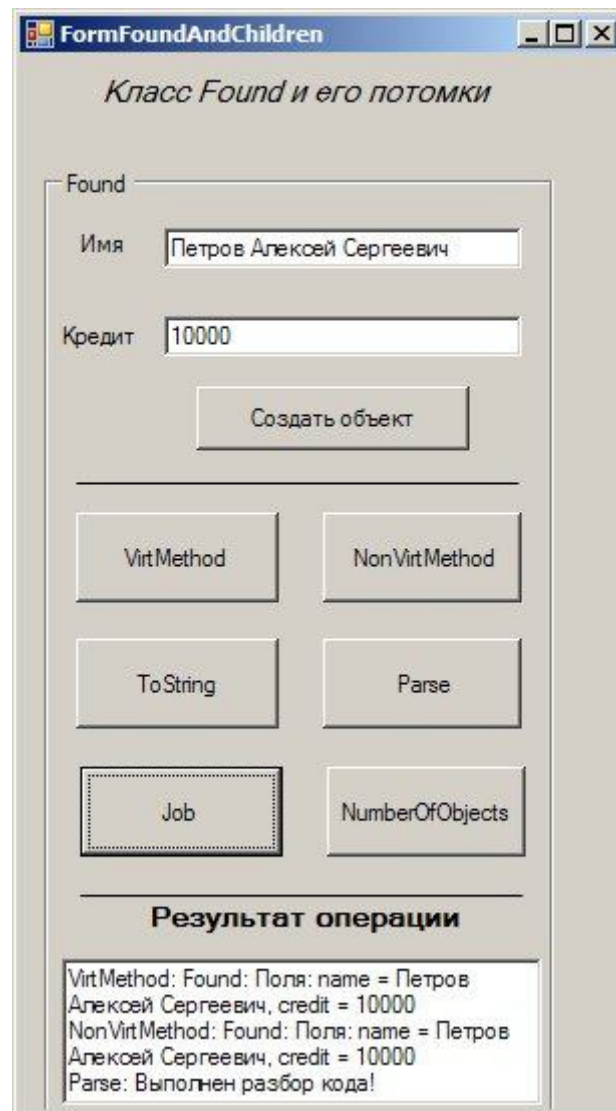


Рис. 1. Интерфейс для работы с объектами класса Found

Как видите, интерфейс устроен достаточно просто. Есть окошки для задания значений, инициализирующих поля класса, есть командные кнопки, позволяющие создать объект класса и вызывать его методы. Результаты работы отображаются в специальном окне. На рисунке показаны результаты, полученные при вызове метода `Job`, который по ходу работы вызывает другие методы класса. Можно видеть, что как виртуальный, так и неvirtуальный метод дают одинаковые результаты.

Построение класса потомка *Derived*

Создадим теперь класс `Derived` - потомка класса `Found`. В простейшем случае объявление класса может выглядеть так:

```
public class Derived : Found
{
}
```

Тело класса `Derived` пусто, но это вовсе не значит, что объекты этого класса не имеют полей и методов: они наследуют все поля и методы (кроме конструктора) класса `Found` и поэтому могут делать все, что могут делать объекты родительского класса. Можно даже не создавать собственных конструкторов класса. В этом случае автоматически добавляется конструктор по умолчанию - конструктор без аргументов, который будет вызывать конструктор без аргументов родительского класса.

FormFoundAndChildren

Класс Found и его потомки

Found

Имя

Кредит

Создать объект

VirtMethod NonVirtMethod

ToString Parse

Job NumberOfObjects

Результат операции

VirtMethod: Found: Поля: name = Nemo, credit = 0
NonVirtMethod: Found: Поля: name = Nemo, credit = 0
Parse: Выполнен разбор кода!

Derived

Имя

Кредит

Создать объект

VirtMethod NonVirtMethod

ToString Parse

Job NumberOfObjects

Результат операции

VirtMethod: Found: Поля: name = Nemo, credit = 0
NonVirtMethod: Found: Поля: name = Nemo, credit = 0
Parse: Выполнен разбор кода!

Рис. 2. Форма для работы с объектами класса Derived

На рис.2 показана знакомая нам форма, расширенная для работы с объектами класса `Derived`. Как видите, несмотря на то, что тело класса пусто, можно создать объект и вызывать многочисленные его методы, наследованные от родителя. На рисунке показаны результаты работы метода `Job` для объектов классов `Found` и `Derived`, созданных по умолчанию. Результаты совпадают. Если потомок ничего не делает, то его объекты ведут себя так же, как и объекты родительского класса. Каждый объект потомка "является" объектом родительского класса.

Поля

Потомок должен идти дальше родителя - вводить новые свойства, новое поведение, изменять старые методы работы. Прежде всего, потомок может добавить новые свойства - поля класса. Он может скрыть поле родителя, добавив собственное поле с тем же именем, что и поле родителя, возможно, изменив тип этого поля и его модификаторы. Скрытие родительского поля не означает, что потомок удаляет поле, наследованное от родителя. Это поле у потомка остается и доступно для вызова, но скрыто, и чтобы добраться до него, необходимо имя поля уточнить именем родителя, задаваемым ключевым словом `base`.

Итак, потомок не может ни удалить, ни изменить поля, наследованные от родителя, он может только добавить собственные поля и может скрыть поля родителя.

Модифицируем наш класс `Derived`. Добавим новое поле класса `debit`. Будем также полагать, что поля `debit` и `credit` должны иметь тип `double`, а не `int`. По этой причине скроем родительское поле `credit`, добавив собственное поле с тем же именем, но изменив его тип:

```
//добавление и скрытие полей
protected double debit;
new protected double credit;
```

Скрываемые поля следует снабжать модификатором `new`. Если этого не сделать, то родительское поле все равно будет скрыто, и на этапе компиляции будет выдано предупреждение о возникшей ситуации.

Для обоих полей задан модификатор доступа `protected`. Напомню, хорошей стратегией является стратегия "ничего не скрывать от потомков". Какой родитель знает, что именно из сделанного им может понадобится потомкам?

Конструкторы

Каждый класс должен позаботиться о создании собственных конструкторов. Он не может в этом вопросе полагаться на родителя, поскольку, как правило, добавляет собственные поля, о которых родитель ничего знать не может. При создании конструкторов классов потомков есть одна важная особенность. Всякий конструктор создает объект класса - структуру, содержащую поля класса. Но потомок не может самостоятельно создать объект, ему нужна помощь родителя. Конструктор потомка начинает свою работу с вызова конструктора родителя, который создает родительский объект, а затем уже конструктор потомка дополняет этот объект полями потомка.

Чтобы подчеркнуть синтаксически такую семантику работы конструктора, вызов конструктора родителя встроен не в тело конструктора, а в его заголовок. Для вызова конструктора используется ключевое слово `base`, именующее родительский класс:

```
//Конструкторы
public Derived(): base()
{
    debit = 0; credit = 0;
}
public Derived(string name, double debit,
double credit)
: base(name, (int)credit)
{
    this.debit = debit;
    this.credit = credit;
}
```

Вызов конструктора родительского класса с именем `base` синтаксически следует сразу за списком аргументов конструктора, будучи отделен от этого списка символом двоеточия. Все аргументы, необходимые для инициализации полей объекта, передаются конструктору потомка. Часть из этих аргументов передается конструктору родителя для инициализации родительских полей.

Итак, вызов конструктора потомка приводит к цепочке вызовов конструкторов предков, заканчивающейся вызовом конструктора прародителя. Затем в обратном порядке создаются объекты. Последним создается объект потомка и выполняется тело конструктора потомка.

Методы

Потомок может создать новый собственный метод с именем, отличным от имен наследуемых методов. В этом случае никаких особенностей нет. Вот пример такого метода, создаваемого в классе `Derived` и возвращающего значение, которое хранится в скрытом поле `credit`:

```
//Методы
public string MyBaseCredit()
{
    return base.credit.ToString();
}
```

Класс потомок может добавлять собственные методы, скрывать методы родителя. Но ситуация с методами более сложная, чем с полями, поскольку потомок может изменять реализацию родителя, задавая собственную реализацию метода. Если потомок создает метод с именем, совпадающим с именем метода предков, то возможны три ситуации:

1. перегрузка метода. Она возникает, когда сигнатура создаваемого метода отличается от сигнатуры наследуемых методов предков. В этом случае в классе потомка будет несколько перегруженных методов с одним именем, и вызов нужного метода определяется обычными правилами перегрузки методов;
2. переопределение метода. Метод родителя в этом случае должен иметь модификатор `virtual`, `abstract` или `override`. Это наиболее интересная ситуация, и она будет подробно рассмотрена. При переопределении сохраняется сигнатура и модификаторы доступа наследуемого метода;
3. скрывание метода. Если родительский метод не является виртуальным или абстрактным, то потомок может создать новый метод с тем же именем и той же сигнатурой, скрыв родительский метод в данном контексте. Здесь ситуация такая же, как и со скрыванием полей. При вызове метода по его имени предпочтение будет отдаваться методу потомка. Это не означает, что метод родителя становится недоступным. Скрытый родительский метод всегда может быть вызван, если при вызове уточнить имя метода родительским именем `base`.

Вернемся к нашему примеру. Класс `Found` имел в своем составе метод `Parse`. Его потомок класс `Derived` расширил возможности метода, добавив проверку в метод разбора. Поскольку родительский метод `Parse` не был ни виртуальным, ни абстрактным, то новый метод `Parse`, добавленный потомком, скрывает родительский метод:

```
new public string Parse()
{
    string res = base.Parse() + NL;
    res += "Выполнена проверка кода!";
    return res;
}
```

В этом примере демонстрируется типичная ситуация, когда потомок в переопределяемом им методе родителя первым делом вызывает метод родителя, а затем уже выполняет свою часть работы, улучшающую результат.

Осталось рассмотреть наиболее интересную ситуацию, когда переопределяется родительский метод изначально объявленный как виртуальный или абстрактный.

Полиморфизм семейства классов `Found` и `Derived`

В классе `Found` определены два виртуальных метода. Один из них - виртуальный метод `VirtMethod` - определен в самом классе, другой - виртуальный метод `ToString` - наследован от родительского класса `object` и переопределен в классе `Found`. Потомок класса `Found` - класс `Derived` переопределяет оба метода, соблюдая контракт,

закрывающийся в этом случае между родителем и потомком. При переопределении виртуального метода сохраняется имя метода и его сигнатура, изменяется лишь реализация:

```
public override string ToString()
{
    string s = "Поля: name = {0}, Basecredit = {1}" +
        "credit = {2}, debit = {3}";
    return String.Format(s, name, base.credit,
        credit, debit);
}
public override string VirtMethod()
{
    return "Derived: " + this.ToString();
}
```

В классе `Found` определены два не виртуальных метода `NonVirtMethod` и `Job`, наследуемые потомком `Derived` без всяких переопределений. Вы ошибаетесь, если думаете, что работа этих методов полностью определяется базовым классом `Found`. Полиморфизм делает их работу куда более интересной. Давайте рассмотрим в деталях работу метода `Job`:

```
public string Job()
{
    string res = "";
    res += "VirtMethod: " +
        VirtMethod() + NL;
    res += "NonVirtMethod: " +
        NonVirtMethod() + NL;
    res += "Parse: " +
        Parse() + NL;
    return res;
}
```

При компиляции метода `Job` будет обнаружено, что вызываемый метод `VirtMethod` является виртуальным, поэтому для него будет применяться динамическое связывание. Это означает, что вопрос о вызове метода откладывается до момента, когда метод `Job` будет вызван объектом, связанным с `x`. Объект может принадлежать как классу `Found`, так и классам `Derived` и `ChildDerived` (потомку от `Derived`), и в зависимости от класса объекта и будет вызван метод этого класса.

Для вызываемых методов `NonVirtMethod` и `Parse`, не являющихся виртуальными, будет применено статическое связывание, так что метод `Job` всегда будет вызывать методы, принадлежащие классу `Found`. Однако и здесь не все просто. Метод `NonVirtMethod`

```
public string NonVirtMethod()
{
    return "Found: " + this.ToString();
}
```

в процессе своей работы вызывает виртуальный метод `ToString`. Опять-таки, для метода `ToString` будет применяться динамическое связывание, и в момент выполнения будет вызываться метод объекта, а не метод ссылки.

Что же касается метода `Parse`, определенного в каждом классе, то всегда в процессе работы `Job` будет вызываться только родительский метод разбора из-за стратегии статического связывания.

Класс `Found` и его потомок `Derived` имеют виртуальные методы и, следовательно, являются полиморфными классами. Они образуют полиморфное семейство классов с полиморфными методами.

Теперь в статике уже нельзя предсказать, как будут выполняться методы этих классов. На рис. 3 показана работа с объектами этих классов.

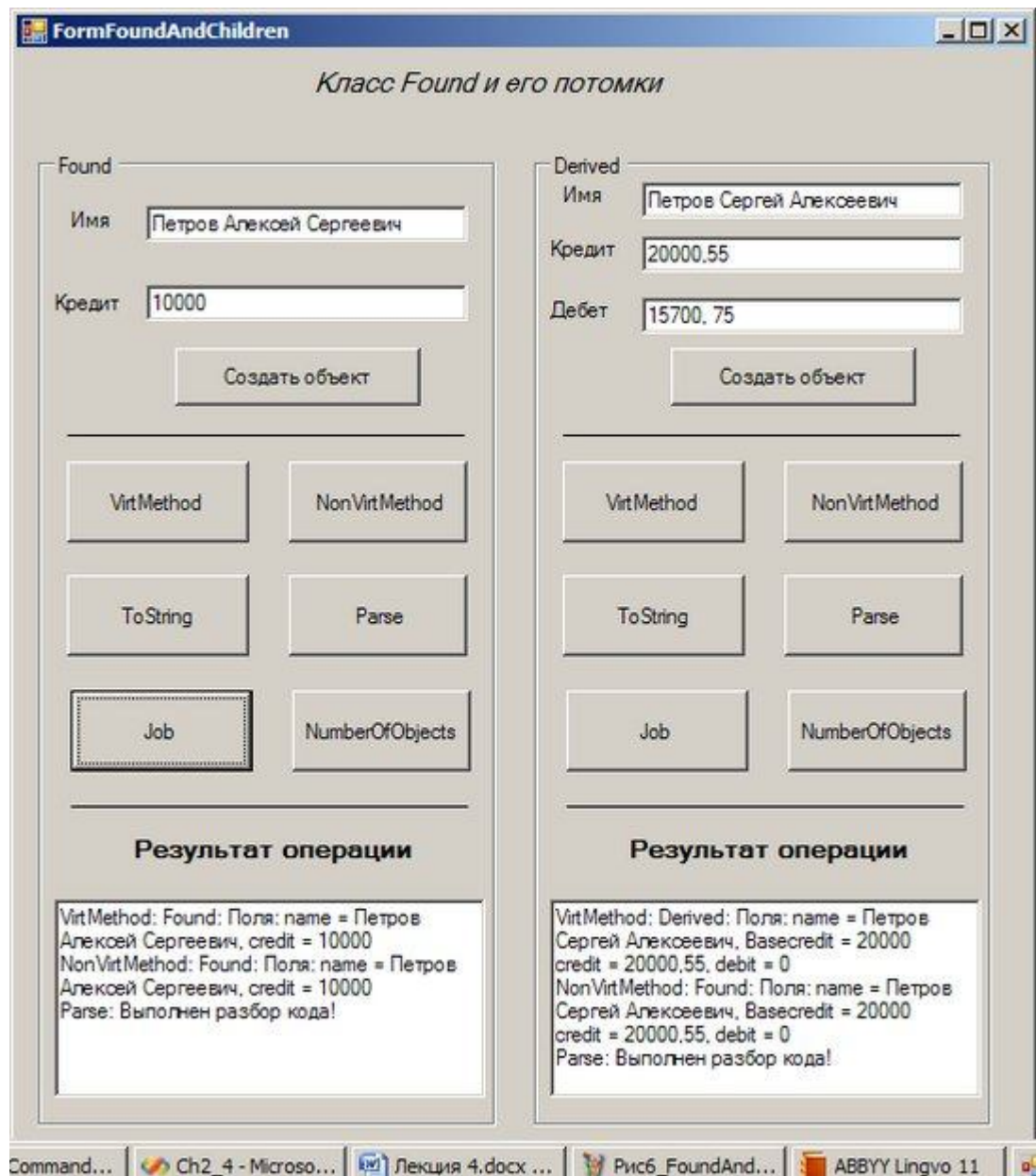


Рис. 3. Полиморфизм семейства классов Found и Derived

В окне результатов отражены данные, полученные в результате вызова метода `Job` объектами этих классов. Несмотря на то, что метод `Job`, определенный в классе `Found`, не является виртуальным методом, потомок `Derived` наследовал этот метод без всякого его изменения, результаты выполнения для обоих объектов различны. Причина проста: не виртуальный метод родителя содержит вызовы виртуальных методов.

Абстрактные классы

С наследованием тесно связан еще один важный механизм проектирования семейства классов - **механизм абстрактных классов**.

Класс называется абстрактным, если он имеет хотя бы один абстрактный метод.

Метод называется абстрактным, если при определении метода задана его сигнатура, но не задана реализация метода.

Объявление абстрактных методов и абстрактных классов должно сопровождаться модификатором `abstract`. Поскольку абстрактные классы не являются полностью определенными классами, нельзя создавать объекты абстрактных классов. Абстрактные классы могут иметь потомков, частично или полностью реализующих абстрактные методы родительского класса. Абстрактный метод представляет виртуальный метод, переопределяемый потомком, поэтому к ним применяется стратегия динамического связывания.

Классы без потомков

Экзотическим, но иногда полезным видом классов являются классы, для которых запрещается строить классы потомки путем наследования. При создании такого класса нет необходимости в выполнении над классом каких-либо болезненных операций. Вполне достаточно приписать классу **модификатор `sealed`**, он и запрещает построение потомков.

9.1.3. Атрибут `Serializable`

Если класс объявить с атрибутом `[Serializable]`, то в него встраивается стандартный механизм сериализации. Необходимость в **сериализации объектов** зачастую возникает при работе с программной системой. Под сериализацией понимают процесс сохранения объектов в долговременной памяти (файлах). Под десериализацией понимают обратный процесс - восстановление состояния объектов, хранимого в долговременной памяти. В бинарном потоке сохраняются все поля объекта, как открытые, так и закрытые. Процессом этим можно управлять, помечая некоторые поля класса атрибутом `[NonSerialized]`, - эти поля сохраняться не будут.

Сериализация позволяет запомнить состояния системы объектов с возможностью последующего возвращения к этим состояниям. Сериализация необходима, когда завершение сеанса работы не означает завершение вычислений. В этом случае очередной сеанс работы начинается с восстановления состояния, сохраненного в конце предыдущего сеанса работы.

Атрибут сериализации может быть задан для таких сущностей, как класс, структура, перечисление и делегат.

В качестве примера рассмотрим два класса `Teacher` и `Student`, связанных взаимными ссылками и допускающих сериализацию объектов. Построим простую модель, в которой создаются объекты этих классов. В некоторый момент состояние этих объектов запоминается за счет использования механизма сериализации. Затем состояние объектов изменяется. А потом, с использованием десериализации, происходит возврат к сохраненному состоянию. Начнем с объявления класса `Teacher`:

```
[Serializable]
class Teacher
{
    string name;
    Student[] students;
    public Teacher(string name, Student[] students)
    {
        this.name = name;
        this.students = students;
    }
}
```

```

    public Student[] Students
    {
        get { return students; }
    }
}

```

Для класса задан атрибут сериализации. У класса два поля: одно задает имя учителя, другое - его учеников. У класса есть конструктор с параметрами и свойство, позволяющее получить доступ к закрытому полю `students`.

Класс `Student` также устроен достаточно просто:

```

[Serializable]
class Student
{
    string name;
    Teacher teacher;
    string topic;
    public Student(string name, Teacher teacher)
    {
        this.name = name;
        this.teacher = teacher;
    }
    public string Topic
    {
        get { return topic; }
        set { topic = value; }
    }
}

```

Для класса также задан атрибут сериализации. У класса три поля, которые задают имя студента, учителя и тему работы, выполняемой под руководством учителя. У класса есть конструктор с параметрами и свойство, позволяющее получить доступ к закрытому полю `topic`.

А теперь построим модель, в которой создаются и существуют объекты этих классов. Как обычно, в классе `Testing` построен соответствующий метод:

```

public void TestTeacherAndStudents()
{
    string teacherName = "Иванов Л.Д.";
    Teacher teacher;
    Student[] students = new Student[3];
    string[] studNames =
    { "Зайцев Г.Ю.", "Гулевич С.А.", "Маргулис В.А." };
    teacher = new Teacher(teacherName, students);
    students[0] = new Student(studNames[0], teacher);
    students[1] = new Student(studNames[1], teacher);
    students[2] = new Student(studNames[2], teacher);
}

```

В нашей модели созданы объекты `teacher` и `students`, задающие учителя и его студентов. Зададим теперь темы работ студентов и сохраним это состояние объектов в бинарном файле.

```

students[0].Topic = "thema0";
students[1].Topic = "thema1";
students[2].Topic = "thema2";
Console.WriteLine("Темы студентов");

```

```

for (int i = 0; i < 3; i++)
    Console.WriteLine("Студент {0}, тема: {1} ",
        studNames[i], students[i].Topic);

Stream tasFile, satFile;
tasFile = File.Create("teach1.bin");
satFile = File.Create("teach2.bin");
BinaryFormatter bif = new BinaryFormatter();
bif.Serialize(tasFile, teacher);
bif.Serialize(satFile, students[0]);
tasFile.Close();
satFile.Close();
Console.WriteLine("Темы запомнили!");

```

Для полноты описания картины сериализации объекты сохраняются в двух разных файлах. В обоих случаях сохраняются все четыре объекта. Граф объектов один и тот же, но корневой объект разный. В одном случае сохранение начинается с учителя, в другом - с одного из студентов. Поскольку при сериализации используется бинарный формater, то сохраняются все поля всех объектов. Заметьте, все классы, участвующие в сериализации, должны иметь атрибут `Serializable`. Метод `Serialize` определен для класса только тогда, когда у класса есть соответствующий атрибут.

Представим себе, что после некоторого обсуждения темы работ решено было изменить:

```

Console.WriteLine("Меняем темы!");
students[0].Topic = "thema7";
students[1].Topic = "thema8";
students[2].Topic = "thema9";
teacher = null;
for (int i = 0; i < 3; i++)
    Console.WriteLine("Студент {0}, тема: {1} ",
        studNames[i], students[i].Topic);

```

Для полноты картины у студентов не стало учителя. Видимо, произошел конфликт. Но потом все уладилось, и нужно вернуться к исходному состоянию. В этом поможет десериализация.

```

Console.WriteLine("Десериализация. Восстанавливаем состояние");
tasFile = File.Open("teach1.bin", FileMode.Open, FileAccess.Read);
bif = new BinaryFormatter();
teacher = (Teacher)bif.Deserialize(tasFile);
students = teacher.Students;
for (int i = 0; i < 3; i++)
    Console.WriteLine("Студент {0}, тема: {1} ",
        studNames[i], students[i].Topic);
}

```

Используя корневой объект, в данном случае `teacher`, восстанавливаем весь граф объектов, связанных с корнем. Но, заметьте, остальные объекты графа - это внутренняя структура, связанная с корнем. Восстановить объект `students` нужно явно, используя свойство `Students` объекта `teacher`. Результаты работы тестового примера:

9.2. Интерфейсы

Интерфейс - это частный случай класса. Интерфейс представляет собой полностью абстрактный класс, все методы которого абстрактны. От абстрактного класса интерфейс отличается некоторыми деталями в синтаксисе и поведении. Синтаксическое отличие состоит в том, что методы интерфейса объявляются без указания модификатора доступа. Отличие в поведении заключается в более жестких требованиях к потомкам. Класс, наследующий интерфейс, обязан полностью реализовать все методы интерфейса. В этом отличие от класса, наследующего абстрактный класс, где потомок может реализовать лишь некоторые методы родительского абстрактного класса, оставаясь абстрактным классом.

Важное назначение интерфейсов, отличающее их от абстрактных классов. Абстрактный класс представляет собой начальный этап проектирования класса, который в будущем получит конкретную реализацию. Интерфейсы задают дополнительные свойства класса. Один и тот же интерфейс позволяет описывать свойства, которыми могут обладать разные классы.

Интерфейс содержит только сигнатуры методов, свойств, событий или индексов. Класс или структура, которые реализуют интерфейс, должны реализовать члены этого интерфейса, указанные в его определении. В следующем примере класс `ImplementationClass` должен реализовать метод с именем `SampleMethod`, который не имеет параметров и возвращает `void`.

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Явная реализация метода интерфейса:
    void ISampleInterface.SampleMethod()
    {
        // Реализация метода.
    }

    static void Main()
    {
        // Создать экземпляр, соответствующий интерфейсу.
        ISampleInterface obj = new ImplementationClass();

        // Вызов метода.
        obj.SampleMethod();
    }
}
```

Интерфейс может быть членом пространства имен или класса и содержать подписи следующих членов:

- Методы
- Свойства
- Индексы
- События

Интерфейс способен наследовать от одного или нескольких базовых интерфейсов.

Если в списке базовых типов содержится базовый класс и интерфейсы, то базовый класс должен стоять в списке на первом месте.

Класс, реализующий интерфейс, может явным образом реализовывать члены этого интерфейса. Явно реализованный член можно вызвать только через экземпляр интерфейса, но не через экземпляр класса.

В следующем примере демонстрируется реализация интерфейса. В этом примере интерфейс содержит объявление свойства, а класс содержит реализацию. Любой экземпляр класса, реализующего IPoint, имеет целочисленные свойства x и y.

```
interface IPoint
{
    // Сигнатуры свойств:
    int x
    {
        get;
        set;
    }

    int y
    {
        get;
        set;
    }
}

class Point : IPoint
{
    // Поля:
    private int _x;
    private int _y;

    // Конструктор:
    public Point(int x, int y)
    {
        _x = x;
        _y = y;
    }

    // Реализация свойств:
    public int x
    {
        get
        {
            return _x;
        }

        set
        {
            _x = value;
        }
    }

    public int y
    {
        get
        {
            return _y;
        }

        set
        {
            _y = value;
        }
    }
}
```

```

    }
}

class MainClass
{
    static void PrintPoint(IPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.x, p.y);
    }

    static void Main()
    {
        Point p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Результат: My Point: x=2, y=3

```

9.2.1. Интерфейс IEquatable<T>

Интерфейс IEquatable<T> объявляет пользователю объекта, что объект может определять, равен ли он другому объекту такого же типа, причем пользователь интерфейса не должен знать, как именно это реализовано.

```

public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Реализация интерфейса IEquatable<T>
    public bool Equals(Car car)
    {
        if (this.Make == car.Make &&
            this.Model == car.Model &&
            this.Year == car.Year)
        {
            return true;
        }
        else
            return false;
    }
}

```

Интерфейсы могут наследовать другие интерфейсы. Класс может наследовать интерфейс несколько раз посредством наследуемых базовых классов или посредством интерфейсов, наследуемых другими интерфейсами. Однако класс может реализовать интерфейс только один раз, и только если интерфейс объявляется в рамках определения класса, как в `class ClassName : InterfaceName`. Если интерфейс унаследован вследствие наследования базового класса, реализующего интерфейс, его реализация предоставляется базовым классом. Базовый класс также может реализовать члены интерфейса с помощью виртуальных членов. В этом случае производный класс может изменить поведение интерфейса путем переопределения виртуальных членов.

9.2.2. Интерфейс IComparable

Часто, когда создается класс, желательно задать отношение порядка на его объектах. Такой класс следует объявить наследником интерфейса `IComparable`. Этот интерфейс имеет всего один метод `CompareTo (object obj)`, возвращающий целочисленное значение, положительное, отрицательное или равное нулю, в зависимости от выполнения отношения "больше", "меньше" или "равно".

Как правило, в классе вначале определяют метод `CompareTo`, а после этого вводят перегруженные операции, чтобы выполнять сравнение объектов привычным образом с использованием знаков операций отношения.

Давайте введем отношение порядка на классе `Person`, сделав этот класс наследником интерфейса `IComparable`. Реализуем в этом классе метод интерфейса `CompareTo`:

```
public class Person : IComparable
{
    public int CompareTo(object pers)
    {
        const string s = "Сравниваемый объект не принадлежит классу Person";
        Person p = pers as Person;
        if (!p.Equals(null))
            return (fam.CompareTo(p.fam));
        throw new ArgumentException(s);
    }
    // другие компоненты класса
}
```

Поскольку аргумент в методе интерфейса принадлежит типу `object`, перед выполнением сравнения его нужно привести к типу `Person`. Для приведения используется операция `as`, позволяющая проверить корректность выполнения приведения. Если приведение невозможно, то невозможно выполнить и сравнение объектов. В этом случае выбрасывается исключение, которое может обработать разумным образом только клиент класса, пытавшийся выполнить сравнение. В самом классе `Person` можно только пояснить ситуацию, передав информацию объекту исключения.

При проверке на значение `null` используется отношение `Equals`, а не обычное равенство, которое будет переопределено.

Определив метод `CompareTo` для класса `Person`, мы тем самым ввели отношение порядка для объектов этого класса. Конечно, сравнение персон может выполняться по разным критериям: возрасту, росту, зарплате. В данном случае отношение порядка на объектах класса `Person` задается как отношение порядка на фамилиях персон. Так как строки наследуют интерфейс `IComparable`, для фамилий персон вызывается метод `CompareTo`, его результат и возвращается в качестве результата метода `CompareTo` для персон.

Введем теперь в нашем классе `Person` перегрузку операций отношения:

```
public static bool operator <(Person p1, Person p2)
{
    return (p1.CompareTo(p2) < 0);
}
public static bool operator >(Person p1, Person p2)
{
    return (p1.CompareTo(p2) > 0);
}
public static bool operator <=(Person p1, Person p2)
{

```

```

        return (p1.CompareTo(p2) <= 0);
    }
    public static bool operator >=(Person p1, Person p2)
    {
        return (p1.CompareTo(p2) >=0);
    }
    public static bool operator ==(Person p1, Person p2)
    {
        return (p1.CompareTo(p2) == 0);
    }
    public static bool operator !=(Person p1, Person p2)
    {
        return (p1.CompareTo(p2) != 0);
    }
}

```

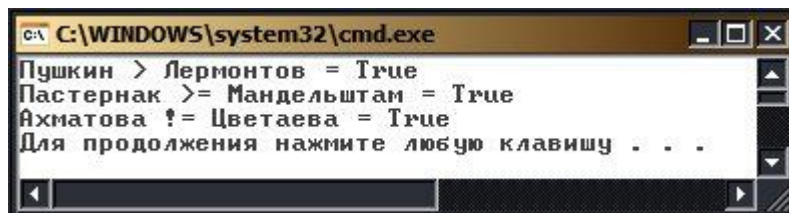
Как обычно, приведу тестовый пример, проверяющий работу с введенными методами:

```

public void TestCompare()
{
    Person poet1 = new Person("Пушкин");
    Person poet2 = new Person("Лермонтов");
    Person poet3 = new Person("Пастернак");
    Person poet4 = new Person("Мандельштам");
    Person poet5 = new Person("Ахматова");
    Person poet6 = new Person("Цветаева");
    Console.WriteLine("{0} > {1} = {2}", poet1.Fam,
        poet2.Fam, (poet1 > poet2));
    Console.WriteLine("{0} >= {1} = {2}", poet3.Fam,
        poet4.Fam, (poet3 >= poet4));
    Console.WriteLine("{0} != {1} = {2}", poet5.Fam,
        poet6.Fam, (poet5 != poet6));
}

```

Вот результаты работы этого теста.



9.2.3. Интерфейс IEnumerable

Интерфейс IEnumerable предоставляет перечислитель, который поддерживает простой перебор элементов неуниверсальной коллекции.

Метод GetEnumerator() Возвращает перечислитель, который выполняет итерацию по элементам коллекции.

Добавим в наш проект новый класс, одним из полей которого будет массив объектов Person. Организуем перечислимость в этом классе, сводящуюся к перечислимости персон массива.

```

/// <summary>
/// Класс - наследник интерфейса IEnumerable,
/// допускающий перечислимость объектов.
/// Перечислимость сводится к перечислимости персон,
/// заданных полем container
/// </summary>

```



```

class Persons : IEnumerable
{
    protected int size;
    protected Person[] container;
    Random rnd = new Random();
    /// <summary>
    /// Конструктор по умолчанию.
    /// Создает массив из 10 персон
    /// </summary>
    public Persons()
    {
        size = 10;
        container = new Person[size];
        FillContainer();
    }
    /// <summary>
    /// Конструктор. Создает массив заданной размерности
    /// Создает его элементы, используя рандомизацию.
    /// </summary>
    /// <param name="size">размерность массива</param>
    public Persons(int size)
    {
        this.size = size;
        container = new Person[size];
        FillContainer();
    }
    /// <summary>
    /// Конструктор, которому передается массив персон
    /// </summary>
    /// <param name="container">массив Person</param>
    public Persons(Person[] container)
    {
        this.container = container;
        size = container.Length;
    }
    /// <summary>
    /// Заполнение массива person
    /// </summary>
    void FillContainer()
    {
        for (int i = 0; i < size; i++)
        {
            int num = rnd.Next(3 * size);
            int age = rnd.Next(27, 46);
            container[i] = new Person("агент_" + num, age);
        }
    }
}

```

Созданный класс `Persons` устроен просто. У него есть поле, названное `container` и представляющее собой массив с элементами класса `Person`. Набор конструкторов класса позволяет передать классу массив персон либо поручить самому классу его формирование, используя метод `FillContainer`. Поскольку класс объявлен наследником интерфейса `IEnumerable`, необходимо реализовать метод `GetEnumerator`, что обеспечит перечислимость объектов класса и даст возможность использовать цикл `for each` при работе с объектами класса. В данном случае реализовать метод интерфейса несложно, и вот как выглядит его реализация:

```

/// <summary>
/// Реализация метода интерфейса IEnumerable
/// Сводится к вызову соответствующего метода
/// для поля container - массива,
/// для которого этот метод реализован в библиотеке FCL

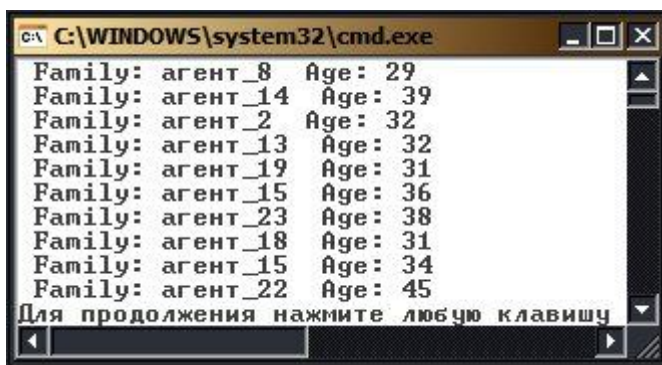
```

```
/// </summary>
/// <returns>перечислитель - интерфейсный объект</returns>
public IEnumerator GetEnumerator()
{
    return container.GetEnumerator();
}
```

Задача решена, и можно попробовать протестировать работу с объектами класса. Добавим соответствующий метод:

```
public void TestEnumeration()
{
    int size = 10;
    Persons agents = new Persons(size);
    foreach (Person agent in agents)
        Console.WriteLine(agent.ToString());
}
```

В тесте создается объект класса `Persons`, и в цикле `foreach` объекты перебираются, возвращая каждый раз очередной объект класса `Person`. Метод `ToString`, определенный в классе `Person`, позволяет выводить информацию об объектах. Результаты работы теста:



Источники:

1. Биллиг В. Основы программирования на С#. Режим доступа: <http://www.intuit.ru/studies/courses/2247/18/info>
2. Биллиг В. Основы программирования на С# 3.0. Ядро языка. Режим доступа: <http://www.intuit.ru/studies/courses/1094/428/info>