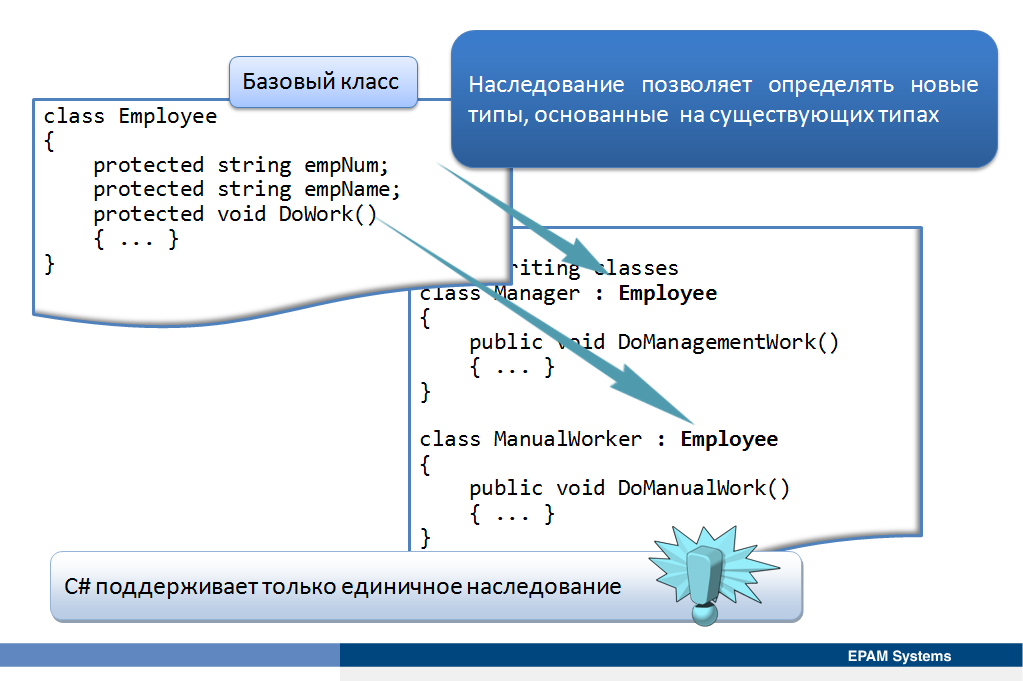
**Наследование, интерфейсы**

**Наследование в C#**



Наследование является ключевым понятием в мире объектно-ориентированного подхода. Наследование можно использовать в качестве инструмента для избегания повторения кода при определении различных классов, имеющих общие черты и связанные друг с другом. Возможно, такие являются различными подклассами одного типа, имеющими свои собственные отличительные черты.

Например, и менеджеры, и рабочие, все являются сотрудниками завода. При разработке приложения, имитирующего сотрудников завода, необходимо уточнить, что руководители и рабочие обладают одинаковыми особенностями, имея при этом отличительные черты. Например, все сотрудники имеют идентификационный номер и имя, однако менеджеры, в отличие от рабочих, имеют другие обязанности и выполняют другие задачи. Для решения этой проблемы можно разработать класс для представления сотрудника, включающий в себя поля для хранения информации, являющейся общей для всех сотрудников: идентификационный номер и имя сотрудника. Тогда можно разработать один класс для представления менеджера и другой класс для представления рабочего. Оба класса (менеджера и рабочего) необходимо будут хранить идентификационный номер сотрудника и его имя. Вместо добавления дублирующих полей для каждого из этих классов можно просто указать, что оба эти классы наследуются от класса сотрудника, уже имеющего поля для хранения этих значений. Кроме того базовый класс может содержать некоторое поведение, являющееся общим для менеджеров и рабочих. Это поведение можно реализовать как метод сотрудника класса. Классы «менеджер» и «рабочий» наследуют это поведение. Использование наследования таким образом снижает необходимость дублирования кода, уменьшая таким образом как время разработки, так и риск внедрения ошибок. В классы менеджера и рабочего можно добавить дополнительные, специфические поля и методы для моделирования различных данных и поведения этих типов.

Указать, что класс наследуется от другого класса (называемого базовым классом или родительским), можно с помощью двоеточия и следующего за ним имени базового класса. Все члены базового класса по умолчанию являются private. Это означает, что они не могут быть доступны коду в других классах, в том числе и в классах, которые наследуются от базового класса. Однако можно сделать члены базового класса открытыми наследуемому классу, используя ключевое слово protected, сохранив при этом их закрытыми от других классов, не являющихся частью иерархии наследования.

// Base class

class Employee

{

protected string empNum;

protected string empName;

protected void DoWork()

{

...

}

}

// Inheriting classes

class Manager : Employee

{

public void DoManagementWork()

{

...

}

}

class ManualWorker : Employee

{

public void DoManualWork()

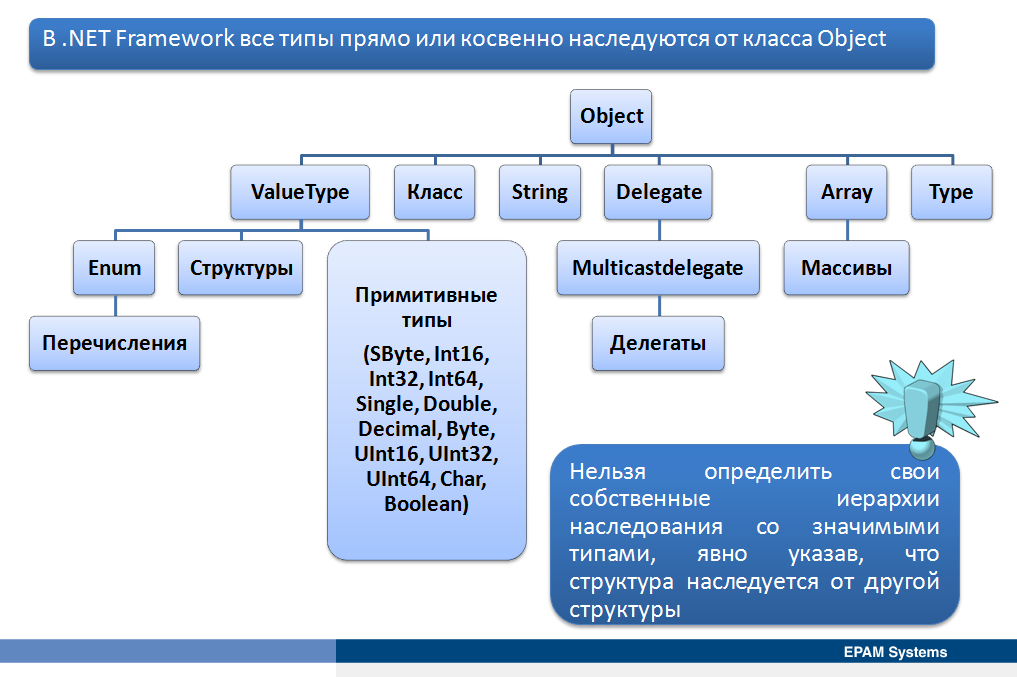
{

...

}

}

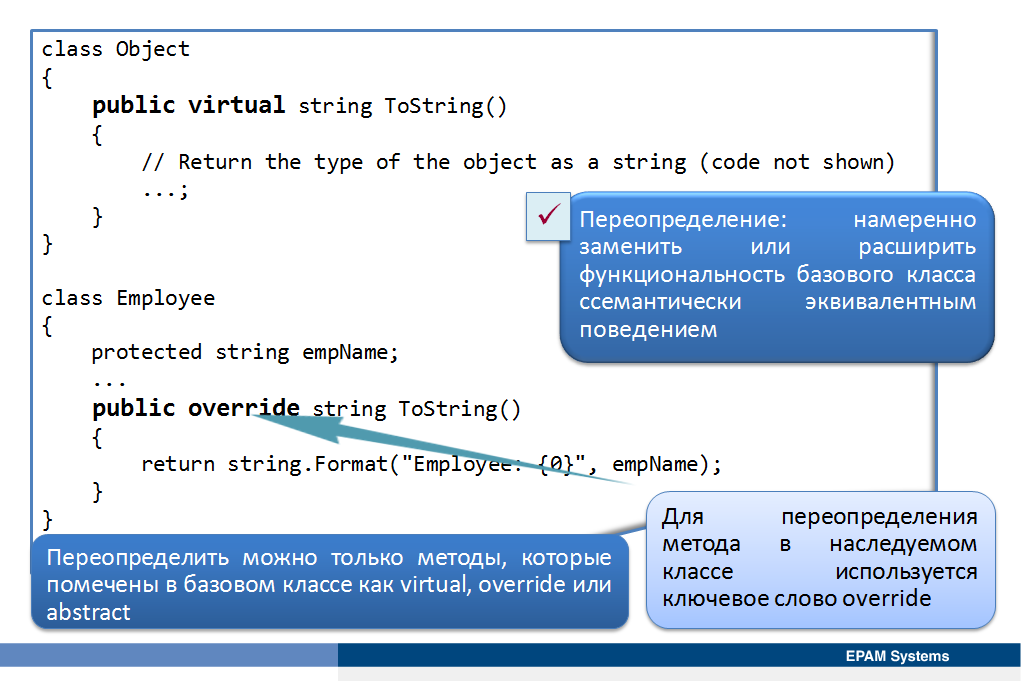
С# поддерживает только единичное наследование. Нельзя определить класс, непосредственно наследуемый от более чем одного базового класса.



В .NET Framework все типы прямо или косвенно наследуются от класса Object пространтсва имен System. Класс Object обеспечивает полезную для всех типов функциональность, например, с помощью методов ToString и Equals. При создании нового ссылочного типа, такого как класс, он наследуется непосредственно от класса Object, при этом указывать эту связь как часть определения класса не обязательно. При наследовании от другого класса, такого как Employee, автоматически наследуются также все функциональные возможности класса Object.

Значимые типы, такие как структуры, наследуются от класса System.ValueType, который, в свою очередь, унаследован от класса Object. Типы enum наследуются от класса System.Enum, который наследуется от ValueType. В отличие от классов описанная иерархия является фиксированной, и нельзя определить свои собственные иерархии наследования со значимыми типами, явно указав, что структура наследуется от другой структуры.

**Переопределение и сокрытие методов**



При наследовании наследующий класс может определять свои собственные методы. Вполне возможно, что некоторые из этих методов имеют те же имена, что и у методов базового класса. В таких ситуациях, необходимо решить, нужно переопределить унаследованные методы или скрыть их.

При переопределении метода предоставляется реализация, имеющая тот же смысл, что и оригинальный метод, но являющая специфической для наследуемого класса. Например, класс Object обеспечивает метод ToString, возвращающий представление объекта в виде строки. При этом, реализация по умолчанию метода ToString в классе Object просто возвращает имя типа в виде строки. Например, для класса Employee логично переопределить поведение метода ToString, вернув из него строку, содержащую имя сотрудника.

Чтобы переопределить метод в наследуемом классе используется ключевое слово override.

class Object

{

public virtual string ToString()

{

// Return the type of the object as a string (code not shown)

...;

}

}

class Employee

{

protected string empName;

...

public override string ToString()

{

return string.Format("Employee: {0}", empName);

}

}

Переопределить можно только методы, которые помечены в базовом классе как virtual, override или abstract. Виртуальные методы, как правило, предлагают реализацию по умолчанию, которую наследуемые классы, как ожидается, заменят своим собственным кодом. При определении класса, от которого могут наследоваться другие классы, следует решить, каким методам будет позволено быть измененными и в этом случае объявить их как виртуальные (virtual). Следует отметить, что при переопределении виртуального метода, нельзя изменить уровень доступа метода, если метод в базовом классе protected, переопределенный метод также должны быть protected.

В наследуемом классе можно определить методы, имеющие то же имя, что и методы базового класса, даже если они не помечены как virtual, abstract или override. Это означает, что не существует никакой связи между переопределенным и оригинальным методами, и новый метод скрывает оригинальный. В этом случае C# компилятор выдает предупреждение (можно не знать, что базовый класс имеет метод с таким же именем, и, изменив его, избежать этого конфликта), хотя код по-прежнему компилируется. Если метод базового класса прячется намеренно, можно выключить предупреждение компилятора, отметив метод ключевым словом new.

class Employee

{

protected void DoWork()

{

...

}

}

class Manager : Employee

{

public new void DoWork()

{

// Hide the DoWork method in the base class

...

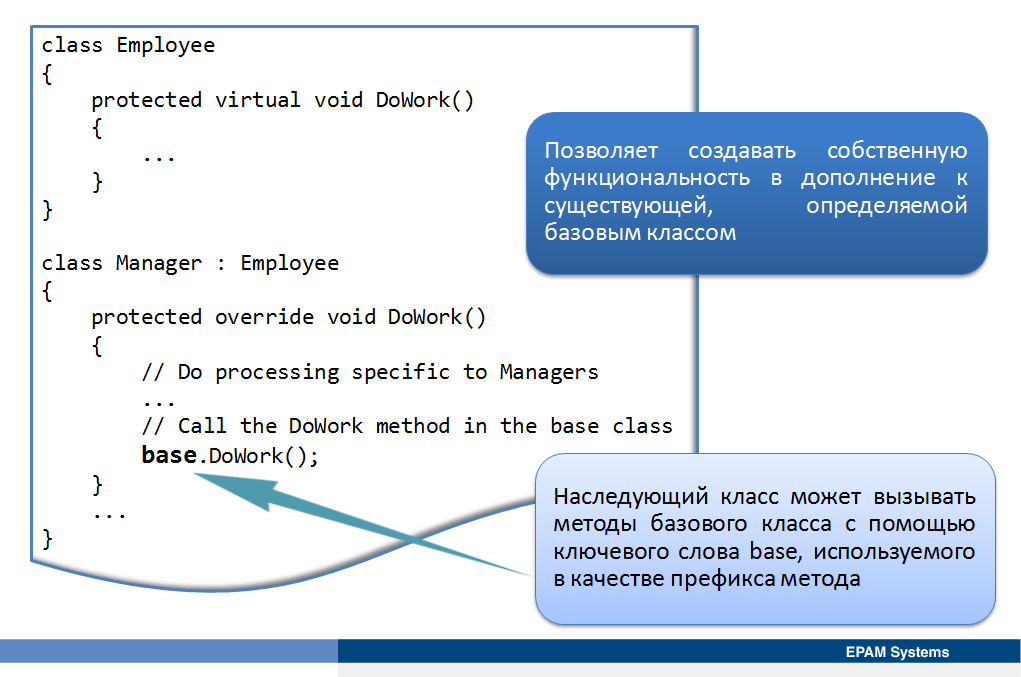
}

...

}

При скрытии метода можно изменить его уровень доступа. Например, можно скрыть protected или private методы базового класса public методом, имеющим то же имя в наследуемом классе, хотя так делать не рекомендуется. Лучше переопределить метод, чем скрыть его, сокрытие метода часто свидетельствует о плохом дизайне.

**Вызов методов и конструкторов базового класса**



Наследующий класс может вызывать методы базового класса с помощью ключевого слова base, используемого в качестве префикса метода. Эта возможность полезна при переопределении методов, поскольку позволяет создавать собственную функциональность в дополнение к существующей, определяемой базовым классом. По сути, это расширяет методы. В следующем примере показан метод DoWork класса Manager, переопределяющий метод DoWork класса Employee, от которого он наследуется, но в соответствующий момент вызывающий метод DoWork класса Employee.

class Employee

{

protected virtual void DoWork()

{

...

}

}

class Manager : Employee

{

protected override void DoWork()

{

// Do processing specific to Managers

...

// Call the DoWork method in the base class

base.DoWork();

}

...

}

Производный класс в дополнение к наследуемым методам автоматически наследует и все поля базового класса. При создании объекта поля требуют инициализации. Как правило, такого рода инициализацию выполняют в конструкторе. Все классы имеют по крайней мере один конструктор[[1]](#footnote-1). Хорошей практикой для конструктора производного класса является вызов конструктора базового класса как части инициализации. Для вызова конструктора базового класса при определении конструктора для наследуемого класса можно указать ключевое слово base.

class Employee

{

protected string empName;

public Employee(string name) // constructor for base class

{

this.empName = name;

}

...

}

class Manager : Employee

{

protected string empGrade;

public Manager(string name, string grade) : base(name)// calls Employee(name)

{

this.empGrade = grade;

}

...

}

Если в конструкторе производного класса нет явного вызова конструктора базового класса, перед выполнением кода компилятор пытается вставить в конструктор производного класса вызов конструктора по умолчанию базового класса.

class Manager : Employee

{

public Manager(string name, string grade)

{

...

}

...

}

Компилятор перепишет этот код в следующий.

class Manager : Employee

{

public Manager(string name, string grade) : base()

{

...

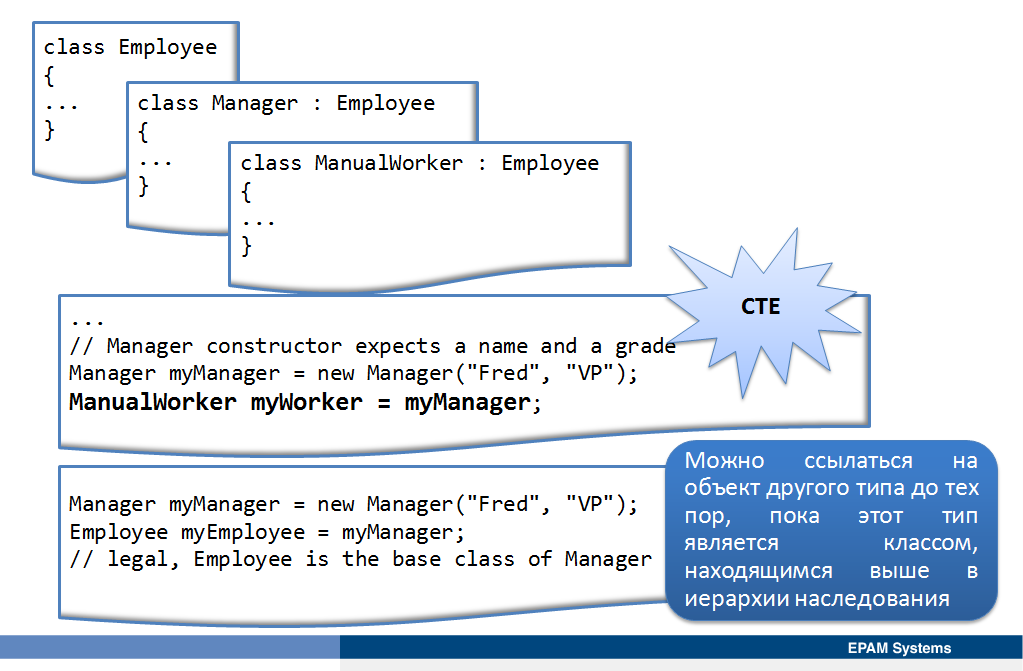
}

...

}

Такой подход работает, когда класс Employee имеет открытый конструктор по умолчанию. Однако, не все классы имеют открытый конструктор по умолчанию (компилятор генерирует конструктор по умолчанию, только если не написаны нестандартные конструкторы), и в этом случае забывая правильно вызвать конструктор базового класса в результате получится ошибка времени компиляции.

## Присваивание и ссылка на классы в иерархии наследования



Правила проверки типов C# помешают присвоить объекту одного типа переменную другого типа. Например, с учетом определения классов Employee, Manager и ManualWorker, следующий пример является неверным.

class Employee

{

...

}

class Manager : Employee

{

...

}

class ManualWorker : Employee

{

...

}

...

// Manager constructor expects a name and a grade

Manager myManager = new Manager("Fred", "VP");

ManualWorker myWorker = myManager;

// error – different types

Однако, можно ссылаться на объект другого типа до тех пор, пока этот тип является классом, находящимся выше в иерархии наследования. Таким образом операторы в следующем примере кода являются верными.

Manager myManager = new Manager("Fred", "VP");

Employee myEmployee = myManager;

// legal, Employee is the base class of Manager

Такой код работает, поскольку иерархия наследования означает, что можно думать об объекте Manager просто как об особом виде Employee, и, следовательно, он имеет все, что имеет Employee с несколькими дополнительными битами, отведенные на методы и поля, добавленные классу Manager. Следовательно, переменную Employee можно также сделать ссылкой на объект ManualWorker. Существует, однако, одно существенное ограничение, при ссылке на объекты Manager или ManualWorker при использовании переменной Employee, можно получить доступ только к методам и полям, определенных в классе Employee. Любые дополнительные определенные классами Manager или ManualWorker методы не доступны через ссылку Employee. Это объясняет, почему почти ничего нельзя назначить переменной Object (все классы наследуют System.Object прямо или косвенно).

Хотя и можно присвоить объект Manager переменной Employee, обратное неверно, нельзя безоговорочно назначить объект Employee переменной Manager потому, что не все Employee объекты являются объектами Manager; некоторые из них могут быть, например, объектами ManualWorker. Однако можно присвоить объекту Employee переменную Manager как только станет известно, что Employee действительно Manager, используя операции as или is, либо с помощью приведения типов.

Операция as проверяет, является ли объект ссылкой на указанный тип. Если это так, она возвращает новую ссылку, используя этот тип, в противном случае возвращает null.

Приведение к типу можно использовать для присваивания ссылки одного типа переменной другому типу, если преобразование возможно. Однако, следует использовать операцию is для предварительной проверки, будет ли операция преобразования успешной. Подобно операции as, операция is проверяет, является ли объект ссылкой на указанный тип и возвращает true, если это так и false в противном случае. При попытке выполнить недопустимое преобразование, которое компилятор не сможет обнаружить во время компиляции, код генерирует исключение InvalidCastException.

В следующем примере используется операция as для проверки, ссылается ли myEmployee на объект Manager. Если это так, происходит результирующее присваивание ссылке myManagerAgain, ссылающейся на тот же объект Manager что и myManager. Если myEmployee ссылается на тип ManualWorker, оператор as возвращает null.

Manager myManager = new Manager("Fred", "VP");

Employee myEmployee = myManager; // myEmployee refers to a Manager

...

Manager myManagerAgain = myEmployee as Manager;

// OK - myEmployee is a Manager

...

ManualWorker myWorker = new ManualWorker("Bert");

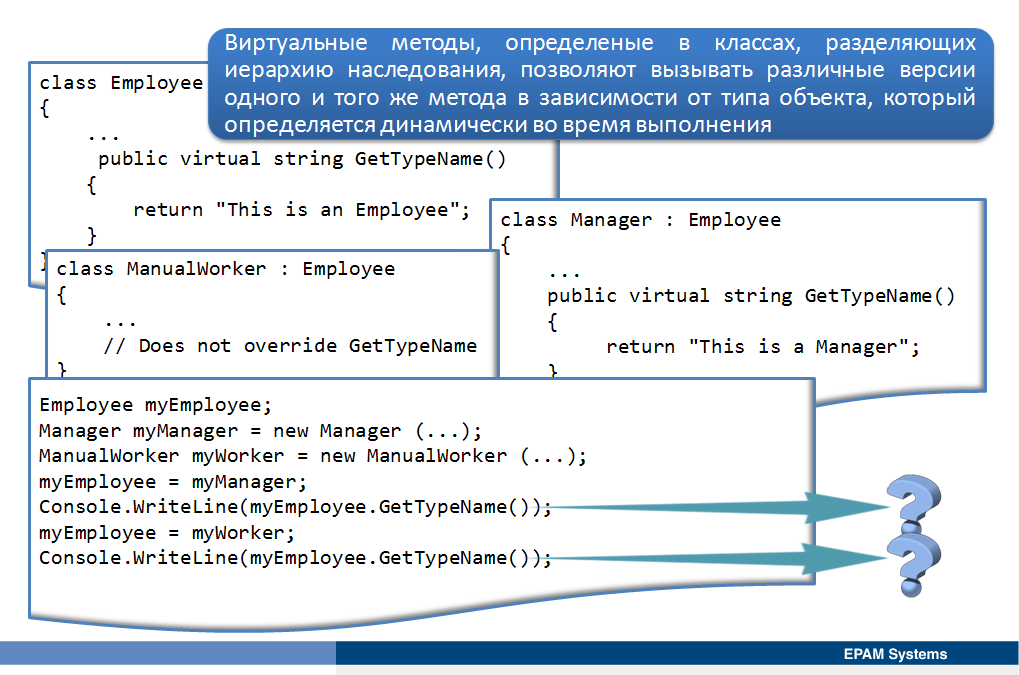
myEmployee = myWorker; // myEmployee now refers to a ManualWorker

...

myManagerAgain = myEmployee as Manager;

// returns null - myEmployee is a ManualWorker

## Полиморфизм



Виртуальные методы, определеные в классах, разделяющих иерархию наследования, позволяют вызывать различные версии одного и того же метода в зависимости от типа объекта, который определяется динамически во время выполнения. Это полиморфизм – очень мощное средство объектно-ориентированных систем.

В следующем примере рассмотрены классы, определяющие вариации иерархии Employee.

class Employee

{

...

public virtual string GetTypeName()

{

return "This is an Employee";

}

}

class Manager : Employee

{

...

public virtual string GetTypeName()

{

return "This is a Manager";

}

}

class ManualWorker : Employee

{

...

// Does not override GetTypeName

}

В этой иерархии метод GetTypeName переопределяется в классе Manager, а в классе ManualWorker нет. Что же будет отображаться двумя операторами Console.WriteLine в следующем примере кода?

Employee myEmployee;

Manager myManager = new Manager (...);

ManualWorker myWorker = new ManualWorker (...);

myEmployee = myManager;

Console.WriteLine(myEmployee.GetTypeName()); // Manager

myEmployee = myWorker;

Console.WriteLine(myEmployee.GetTypeName()); // ManualWorker

Поскольку каждый оператор вызывает метод GetTypeName на переменной myEmployee, которая является ссылками на Employee, можно было бы ожидать от них печати «This is an Employee». Однако, в первом случае, myEmployee фактически ссылается на объект Manager. Метод GetTypeName определен как виртуальный, поэтому во время выполнения он должен вызвать метод GetTypeName класса Manager. Таким образом, первый оператор на самом деле выводит сообщение «This is a Manager». Второй оператор Console.WriteLine вызывает GetTypeName на объекте GetTypeName. Однако, класс ManualWorker не имеет собственной реализации метода GetTypeName, поэтому по умолчанию вызывается метод класса Employee, возвращающий строку «This is an Employee».

*Наследование и исключения*

Создать свой Exception не сложно – нужно определить public-класс, который будет наследоваться от **System.Exception** или **System.ApplicationException**. Хотя это и не является хорошей практикой, кода внутри созданного класса исключения можно не писать вообще:

public class UserNotFoundException : ApplicationException

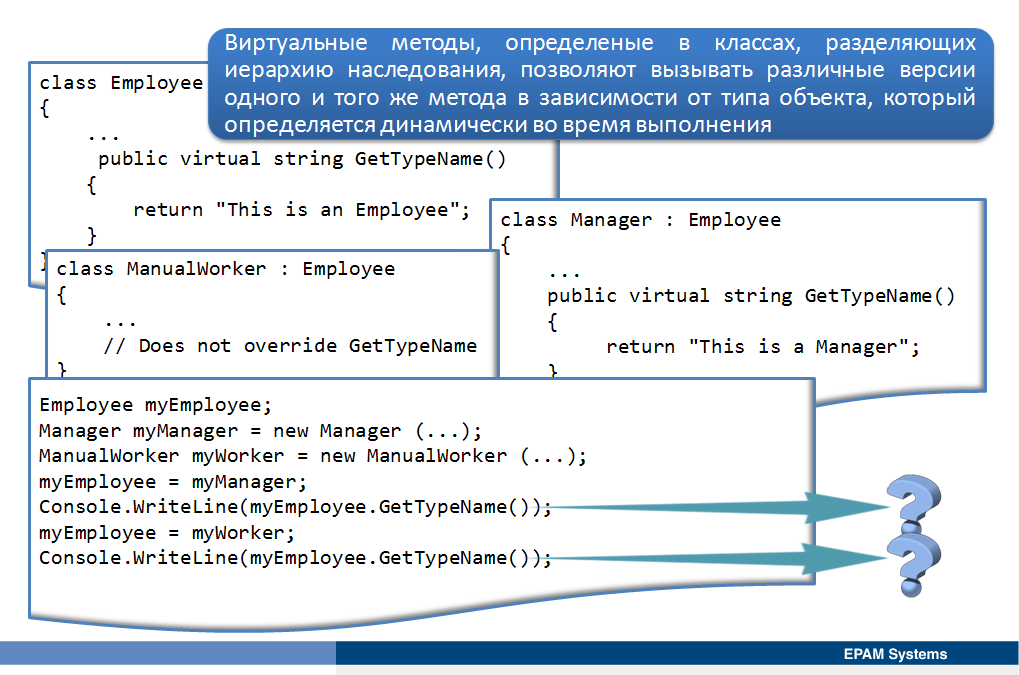
{

}

Каждый из этих типов предназначен для конкретной цели. Тогда как **System.Exception** является общим классом для всех user-defined exceptions, то **System.ApplicationException** определяет исключения, возникающие на уровне конкретного приложения.

К примеру, приложение из данного раздела является отдельной программой, поэтому вполне допустимо наследовать определенный нами exception от System.ApplicationException.

**Виртуальные методы. Полиморфизм.**



Виртуальные методы, определеные в классах, разделяющих иерархию наследования, позволяют вызывать различные версии одного и того же метода в зависимости от типа объекта, который определяется динамически во время выполнения. Это полиморфизм – очень мощное средство объектно-ориентированных систем.

В следующем примере рассмотрены классы, определяющие вариации иерархии Employee.

class Employee

{

...

public virtual string GetTypeName()

{

return "This is an Employee";

}

}

class Manager : Employee

{

...

public virtual string GetTypeName()

{

return "This is a Manager";

}

}

class ManualWorker : Employee

{

...

// Does not override GetTypeName

}

В этой иерархии метод GetTypeName переопределяется в классе Manager, а в классе ManualWorker нет. Что же будет отображаться двумя операторами Console.WriteLine в следующем примере кода?

Employee myEmployee;

Manager myManager = new Manager (...);

ManualWorker myWorker = new ManualWorker (...);

myEmployee = myManager;

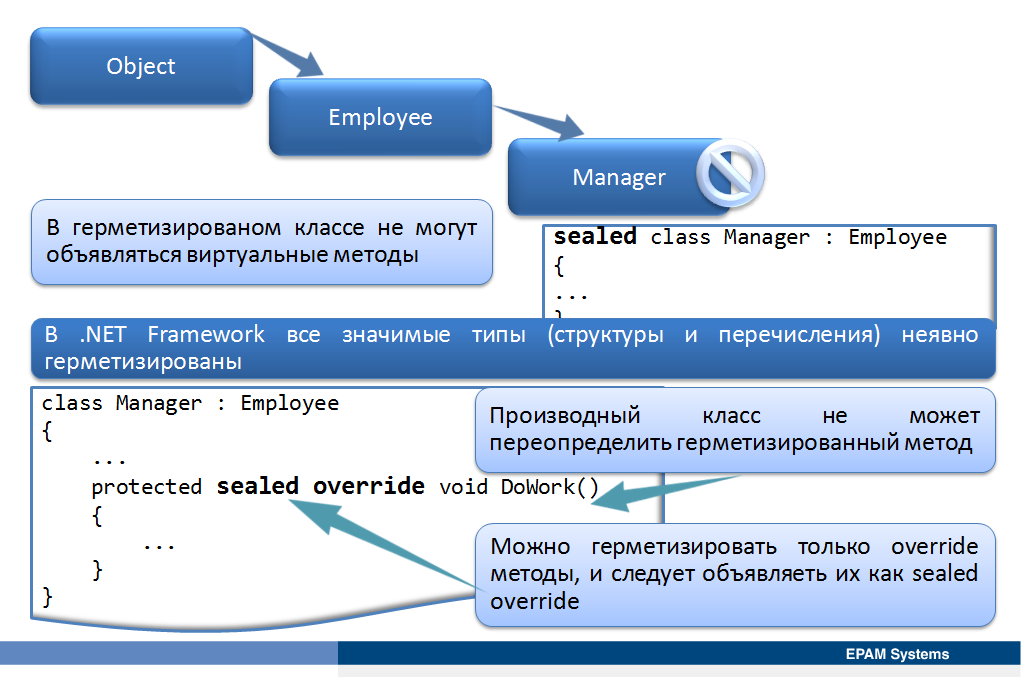
Console.WriteLine(myEmployee.GetTypeName()); // Manager

myEmployee = myWorker;

Console.WriteLine(myEmployee.GetTypeName()); // ManualWorker

Поскольку каждый оператор вызывает метод GetTypeName на переменной myEmployee, которая является ссылками на Employee, можно было бы ожидать от них печати «This is an Employee». Однако, в первом случае, myEmployee фактически ссылается на объект Manager. Метод GetTypeName определен как виртуальный, поэтому во время выполнения он должен вызвать метод GetTypeName класса Manager. Таким образом, первый оператор на самом деле выводит сообщение «This is a Manager». Второй оператор Console.WriteLine вызывает GetTypeName на объекте GetTypeName. Однако, класс ManualWorker не имеет собственной реализации метода GetTypeName, поэтому по умолчанию вызывается метод класса Employee, возвращающий строку «This is an Employee».

**Определение герметизированных классов и методов. Использование ключевого слова sealed.**



По умолчанию при определении класса, пользователи, имеющие доступ к содержащей класс сборке могут наследовать от него, добавляя новые функциональности. Однако, если класс сознательно проектируется с намерением не использовать его в качестве базового класса, крайне маловероятно, что он будет хорошо функционировать в качестве базового класса. C# позволяет использовать ключевое слово sealed для предотвращения использования класса в качестве базового класса. В следующем примере кода класс Manager объявляется как герметизированный (sealed)

sealed class Manager : Employee

{

...

}

Если какой-либо класс попытается использовать определенный таким образом класс Manager в качестве базового, возникнет ошибка времени компиляции. Следует отметить, что в герметизированом классе не могут объявляться виртуальные методы.

В .NET Framework все типы значение (структуры и перечисления) неявно герметизированы.

Ключевое слово sealed также можно использовать для объявления, что отдельный метод в негерметизированном классе герметизирован. Это означает, что производный класс не может переопределить герметизированный метод. Можно герметизировать только override методы, и следует объявляеть их как sealed override.

class Manager : Employee

{

...

protected sealed override void DoWork()

{

...

}

}

**Анализ базового класса Object**

Поддерживает все классы в иерархии классов .NET и предоставляет низкоуровневые службы для производных классов. Является исходным базовым классом для всех классов .NET и корнем иерархии типов.

В следующем примере определяется тип Point, производный от класса [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netframework-4.8), и переопределяются многие виртуальные методы класса [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netframework-4.8). Кроме того, в примере показано, как вызывать многие из статических методов и методов экземпляра класса [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netframework-4.8).

using System;

// The Point class is derived from System.Object.

class Point

{

public int x, y;

public Point(int x, int y)

{

this.x = x;

this.y = y;

}

public override bool Equals(object obj)

{

// If this and obj do not refer to the same type, then they are not equal.

if (obj.GetType() != this.GetType()) return false;

// Return true if x and y fields match.

var other = (Point) obj;

return (this.x == other.x) && (this.y == other.y);

}

// Return the XOR of the x and y fields.

public override int GetHashCode()

{

return x ^ y;

}

// Return the point's value as a string.

public override String ToString()

{

return $"({x}, {y})";

}

// Return a copy of this point object by making a simple field copy.

public Point Copy()

{

return (Point) this.MemberwiseClone();

}

}

public sealed class App

{

static void Main()

{

// Construct a Point object.

var p1 = new Point(1,2);

// Make another Point object that is a copy of the first.

var p2 = p1.Copy();

// Make another variable that references the first Point object.

var p3 = p1;

// The line below displays false because p1 and p2 refer to two different objects.

Console.WriteLine(Object.ReferenceEquals(p1, p2));

// The line below displays true because p1 and p2 refer to two different objects that have the same value.

Console.WriteLine(Object.Equals(p1, p2));

// The line below displays true because p1 and p3 refer to one object.

Console.WriteLine(Object.ReferenceEquals(p1, p3));

// The line below displays: p1's value is: (1, 2)

Console.WriteLine($"p1's value is: {p1.ToString()}");

}

}

// This code example produces the following output:

//

// False

// True

// True

// p1's value is: (1, 2)

//

В большинстве языков не требуется объявлять для классов наследование от [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netframework-4.8), так как они наследуют его неявно.

Поскольку все классы в .NET являются производными от [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netframework-4.8), каждый метод, определенный в классе [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netframework-4.8), доступен во всех объектах системы. Производные классы могут переопределять некоторые из этих методов, включая:

* [Equals](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.equals?view=netframework-4.8) — поддерживает сравнение объектов.
* [Finalize](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.finalize?view=netframework-4.8) — выполняет операции очистки до автоматического освобождения объекта.
* [GetHashCode](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.gethashcode?view=netframework-4.8) — создает число, соответствующее значению объекта для поддержки использования хэш-таблицы.
* [ToString](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.tostring?view=netframework-4.8) создает понятную для человека текстовую строку, описывающую экземпляр класса.

### Вопросы производительности

При разработке класса, например коллекции, который должен обрабатывать объекты любого типа, вы можете создавать члены класса, принимающие экземпляры класса [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netframework-4.8). Однако процесс упаковки-преобразования и распаковки-преобразования для типа требует вычислительных затрат. Если вы знаете, что новый класс будет часто обрабатывать определенные типы значений, для минимизации затрат на упаковку-преобразование можно использовать две тактики.

* Создание общего метода, принимающего тип [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netframework-4.8), и набора специализированных перегрузок метода для каждого из типов значений, которые, как предполагается, будут часто обрабатываться классом. Если существует специализированный метод, который принимает указанный при вызове тип параметра, упаковка-преобразование не выполняется и вызывается специализированный метод. Если у метода отсутствует аргумент, который соответствует указанному при вызове типу параметра, выполняется упаковка-преобразование параметра и вызывается общий метод.
* Использование универсальных шаблонов при разработке метода и его членов. Среда CLR создает закрытый универсальный тип при создании экземпляра класса с указанием аргумента универсального типа. Универсальный метод принимает аргумент определенного типа, и его можно вызывать без упаковки-преобразования параметра.

Несмотря на то, что иногда бывает необходимо создавать классы общего назначения, которые принимают и возвращают типы [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netframework-4.8), можно повысить производительность, создав также специализированный класс для обработки определенного часто используемого типа. Например, наличие специализированного класса для задания и получения логических значений позволяет избежать затрат на их упаковку-преобразование и распаковку-преобразование.

## Методы

|  |  |
| --- | --- |
|  | |
| [Equals(Object)](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.equals?view=netframework-4.8#System_Object_Equals_System_Object_) | Определяет, равен ли указанный объект текущему объекту. |
| [Equals(Object, Object)](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.equals?view=netframework-4.8#System_Object_Equals_System_Object_System_Object_) | Определяет, считаются ли равными указанные экземпляры объектов. |
| [Finalize()](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.finalize?view=netframework-4.8#System_Object_Finalize) | Позволяет объекту попытаться освободить ресурсы и выполнить другие операции очистки, перед тем как он будет уничтожен во время сборки мусора. |
| [GetHashCode()](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.gethashcode?view=netframework-4.8#System_Object_GetHashCode) | Служит в качестве хэш-функции по умолчанию. |
| [GetType()](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.gettype?view=netframework-4.8#System_Object_GetType) | Возвращает объект [Type](https://docs.microsoft.com/ru-ru/dotnet/api/system.type?view=netframework-4.8) для текущего экземпляра. |
| [MemberwiseClone()](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.memberwiseclone?view=netframework-4.8#System_Object_MemberwiseClone) | Создает неполную копию текущего объекта [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netframework-4.8). |
| [ReferenceEquals(Object, Object)](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.referenceequals?view=netframework-4.8#System_Object_ReferenceEquals_System_Object_System_Object_) | Определяет, совпадают ли указанные экземпляры [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netframework-4.8). |
| [ToString()](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.tostring?view=netframework-4.8#System_Object_ToString) | Возвращает строку, представляющую текущий объект. |

**Упаковка, распаковка (boxing,unboxing)**

Зачем нам задумываться об упаковке и распаковке? Разве это не обязанность .NET-среды, которая следит за управлением данных и, соответственно, сама "выбирает" наиболее оптимальный способ их хранения?

На самом деле - нет. Что очень важно знать и понимать -  так это механизм перемещения данных из области стека в кучу - и наоборот.

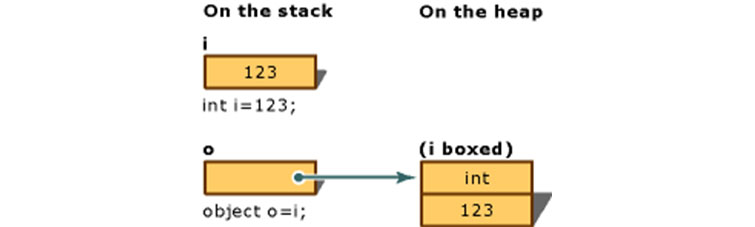
Помните:

* Когда любой значимый тип присваивается к ссылочному типу данных, значение перемещается из области стека в кучу. Эта операция называется упаковкой.
* Когда любой ссылочный тип присваивается к значимому типу данных, значение перемещается из области кучи в стек. Это называется распаковкой.

К примеру, здесь мы имеем следующий пример упаковки:



А вот состояние памяти в момент произведения операции:

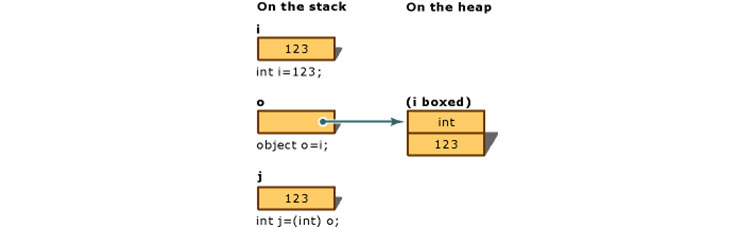


Чтобы сохранить значение "123" в виде объекта, в куче создается "упаковка", куда впоследствии и перемещаются данные.

Когда же производится распаковка:



Вот что происходит с памятью:



Значение "123" было изъято из упаковки и помещено назад в область стека.

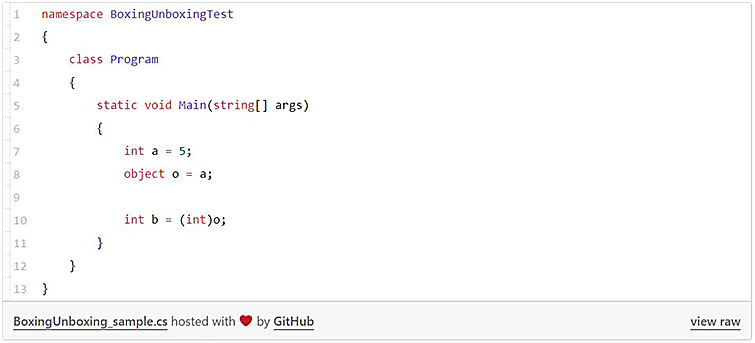
Заметьте, что когда тип данных i упаковывается внутри объекта o, в стеке хранится лишь ссылка, в то время как само значение хранится в куче. Как только производиться распаковка, данные в куче обязаны быть скопированы в стек (переменная j). В обоих случаях наша цель - это работать с тем самым значением (123).

Как вы можете себе представить, сии операции могут быть достаточно ресурсоемкими.

**Давайте рассмотрим IL**

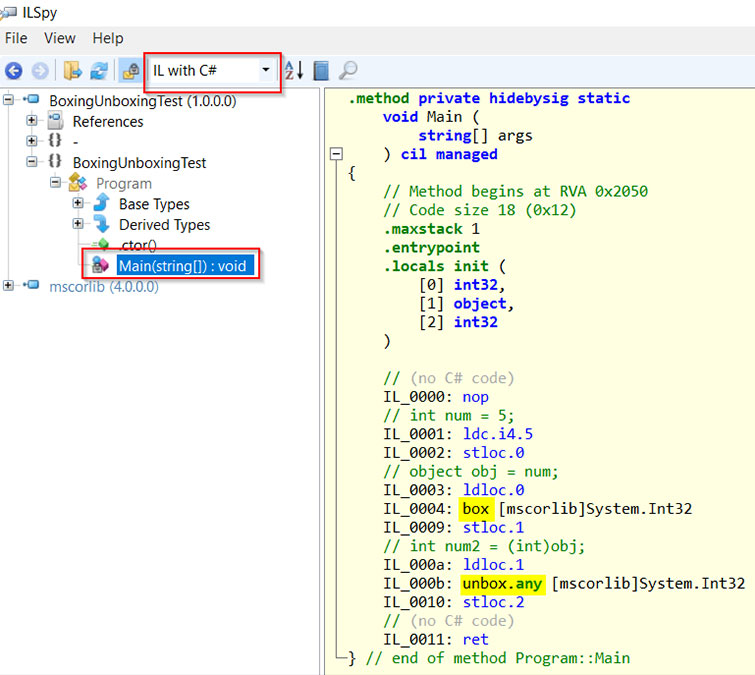
Среда выполнения .NET обязана как-то знать, нужно ли упаковывать или распаковывать определенные переменные. Поэтому для обозначения этих операций также требуются дополнительные затраты памяти.

Давайте создадим несложное .NET консольное приложение:



Теперь скомпилируем приложение и при помощи утилитки ILSpy посмотрим его код внутри EXE.

Как только EXE-файл будет открыт в ILSpy, пронавигируемся к методу Main, выбрав "IL with C#".



Заметьте, что операция box выполняется только после присвоение ссылочному типу значения значимого. И наоборот: unbox.any - только после попытки присвоить ссылочному типу данных значимой переменной.

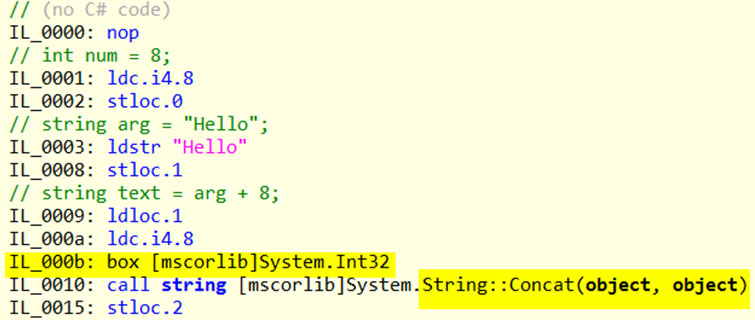
Это де-факто способ, которым операции упаковки и распаковки представлены в IL.

**Конкатенация строк**

Пример в виде конкатенации строк.



Эта операция требует наличия метода String.Concat, который принимает два object-параметра.



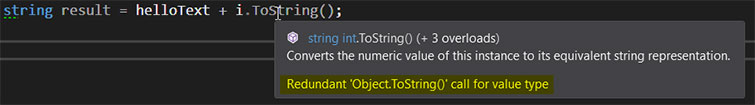
Дабы избежать подобных ситуаций, нам достаточно просто немного изменить код, используя на переменной типа int метод ToString (и здесь стоит проигнорировать сообщение ReSharper о том, что операция бессмысленна:) ).



Никакой упаковки больше нет.

**Производительность**

Как мы уже говорили, упаковка и распаковка требуют определенных затрат производительности. В случае с конкатенацией строк, выигрыш от применения ToString весьма незначителен. Именно потому, даже ReSharper не советовал нам делать подобное:



В этом случае гораздо лучше сохранить читабельность кода без ToString.

Целесообразность оптимизации появляется, как правило, тогда, когда операции упаковки и распаковки предстоит производить в цикле сотни и тысячи раз. В этом случае время выполнения кода с упаковкой может составлять порядка 150 процентов от времени исполнения кода без нее (вы можете сами создать тестовое приложение и сравнить требуемый промежуток времени).

Упакованные значения могут также требовать больше памяти, чем значения в стеке. Копирование значений в/из стека также требует своих затрат. Согласно MSDN, упаковка может занимать порядка 20 раз больше времени, нежели простое присвоение. В то время как распаковка примерно в 4 раза медленней простого присвоения.

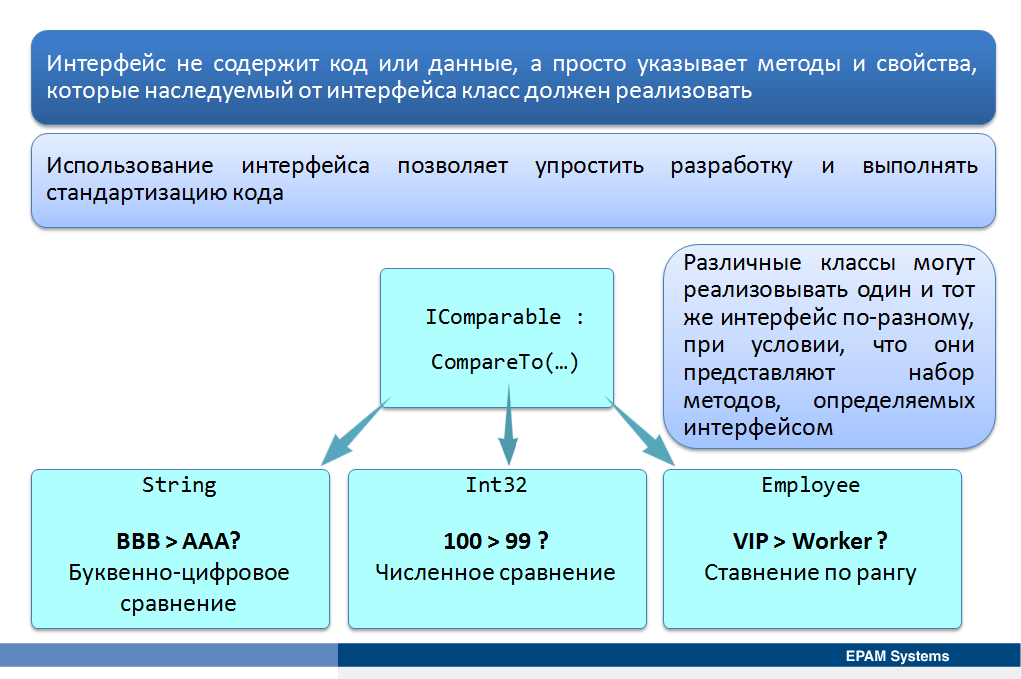
**Итак... зачем же тогда вообще нужно использовать упаковку и распаковку?**

Несмотря на все недостатки в плане падения производительности .NET -приложения, концепции упаковки и распаковки были внедрены в .NET не просто так. И вот причины:

* .NET-стандарт обладает общей системой типов, что позволяет представлять и ссылочные. и значимые типы схожим образом - и все это благодаря упаковке.
* Коллекции можно было использовать для хранения значимых типов до появления обобщений.
* Упрощения кода, вроде конкатенации строк и так далее.

Упаковка и распаковка настолько распространены, что мы не может избежать их полностью. Мы должны знать принцип их работы, чтобы минимизировать их использование, но к этому нужно подходить разумно. Не тратьте свое время на постоянную оптимизацию кода, частую проверку через IL, чтобы убедиться, дабы ни одна лишняя операция упаковки не была использована. Помните, что чистота и простота чтения кода иногда значительно более важна, нежели незаметное, мельчайшее ускорение работы программы.

**Что такое интерфейс?**



Наследование от класса мощный механизм, но реальная мощь наследования происходит при реализации интерфейсов. Интерфейс не содержит код или данные, он просто указывает методы и свойства, которые наследуемый от интерфейса класс должен реализовать. Использование интерфейса позволяет полностью отделить имена и сигнатуры методов класса от их реализации.

Интерфейсы выступают в качестве контракта и гарантируют, что любой класс, реализующий интерфейс, будет представлять членов, которых интерфейс специфицирует. Это позволяет упростить разработку и выполнять стандартизацию кода.

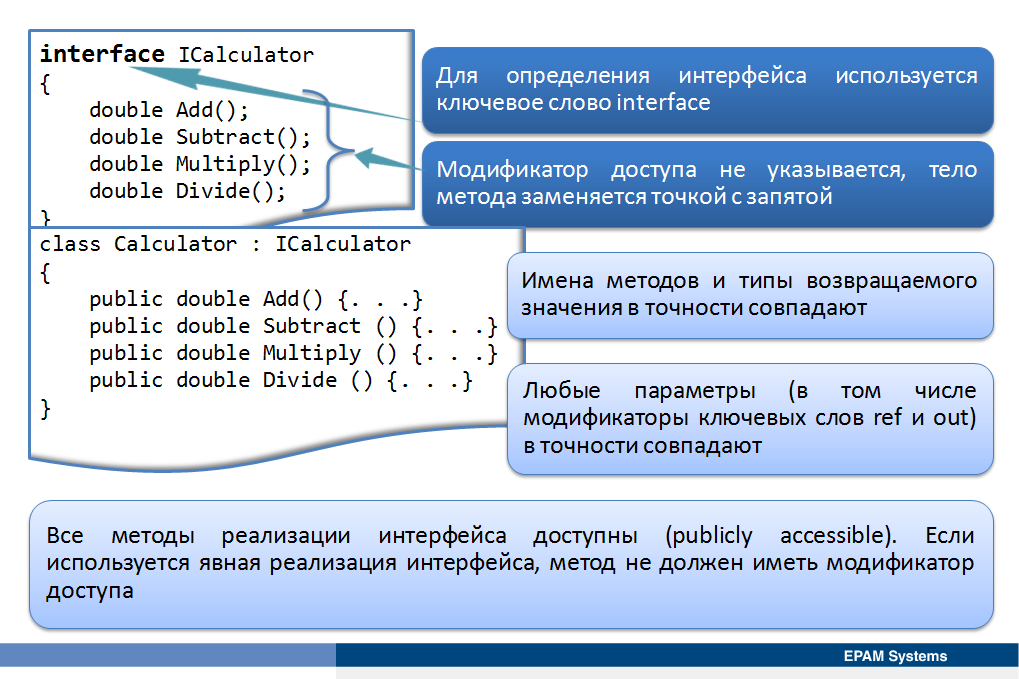
Интерфейсы позволяют указать функциональность, которую должен реализовывать класс. Способ, который класс выбирает для осуществления этой функциональности касается только класса. Различные классы могут реализовывать один и тот же интерфейс по-разному, при условии, что они представляют набор методов, определяемых интерфейсом.

Например, частым требованием для классов в .NET Framework является определение метода, позволяющего двум объектам быть сравнимыми для определения их относительного порядка. Это фактически означает, что сравнение значений двух объектов класса зависит от самого класса. Например, класс Employee может определять, чтобы объекты Employee были ранжированы (два сотрудника одного и того же класса равны, но сотрудник "VIP" класса имеет приоритет перед сотрудником класса "Worker"), тогда как класс String использует буквенно-цифровое сравнение с целью определения относительного порядка строковых значений. Для стандартизации того, каким образом объекты любого типа можно сравнивать, пространство имен System библиотеки классов .NET Framework определяет интерфейс IComparable. Этот интерфейс содержит единственный метод CompareTo, имееющий следующую сигнатуру

int CompareTo(Object obj)

Класс, реализующий интерфейс IСomparable, должен реализовать метод CompareTo. Этот метод возвращает нуль, если объект, указанный как параметр считается равным объекту, для которого метод CompareTo был вызван; значение меньше нуля, если объект считается меньше объекта, который был указан в качестве параметра; и значение больше нуля, если объект считается больше, чем объект, указанный в качестве параметра. Результат зависит от класса, реализующего фактическую логику метода CompareTo интерфейса IСomparable.

**Создание и реализация интерфейсов**



Синтаксически интерфейс похож на класс за исключением того, что методы в нем только объявляются и не предлагается код, их реализующий. Для определения интерфейса используется ключевое слово interface. Внутри интерфейса, объявляются методы также, как в классе или структуре, для которых никогда не указывается модификатор доступа (public, private или protected), тело метода заменяется точкой с запятой.

interface ICalculator

{

double Add();

double Subtract();

double Multiply();

double Divide();

}

В приведенном примере имя интерфейса начинается с заглавной буквы I. Это общее правило именования, а не четкое требование, но документация .NET Framework рекомендует придерживаться этого стандарта: все интерфейсы в пространстве имен System имеют указанный префикс.

Для реализации интерфейса объявляются класс или структура[[2]](#footnote-2), наследующие интерфейс, а также предоставляется код для каждого метода, который интерфейс определяет. При реализации интерфейса, следует убедиться, что каждый имплементируемый метод соответствует в точности соответствующему методу интерфейса, согласно следующими правилами:

* Имена методов и типы возвращаемого значения в точности совпадают.
* Любые параметры (в том числе модификаторы ref и out) в точности совпадают.
* Все методы реализации интерфейса доступны (publicly accessible). Однако, если используется явная реализация интерфейса, метод не должен иметь модификатор доступа.

Если между определением интерфейса и его реализацией есть разница, класс не будет компилироваться.

В следующем примере показан класс, реализующий интерфейс **ICalculator**.

class Calculator : ICalculator

{

// The methods of the ICalculator interface return test data in this code.

#region ICalculator Members

public double Add()

{

return 0;

}

public double Subtract()

{

return 0;

}

public double Multiply()

{

return 0;

}

public double Divide()

{

return 0;

}

#endregion

}

Класс может наследоваться только от одного класса, однако реализовывать несколько интерфейсов. Если необходимо указать, что класс реализует несколько интерфейсов, следует разделить каждый интерфейс запятой в объявлении класса. При реализации более чем одного интерфейса, нужно убедиться, что выполняются вышеуказанные правила для каждого интерфейса, который реализует класс, в противном случае код не будет компилироваться. Например, если необходимо указать, что класс Calculator реализует описанный ранее интерфейс IComparable, можно изменить определение класса следующим образом.

class Calculator : ICalculator, IComparable

{

// Code to implement ICalculator.

#region ICalculator Members

public double Add()

{

return 0;

}

public double Subtract()

{

return 0;

}

public double Multiply()

{

return 0;

}

public double Divide()

{

return 0;

}

# endregion

// Code to implement IComparable.

#region IComparable members

public int CompareTo(Object obj)

{

//...

}

#endregion

}

**Интерфейсные индексаторы, свойства**

Аналогично методам, свойства указываются в интерфейсе вообще без тела. Ниже приведена общая форма объявления интерфейсного свойства:

*// Интерфейсное свойство*

*тип имя{*

*get;*

*set;*

*}*

Очевидно, что в определении интерфейсных свойств, доступных только для чтения или только для записи, должен присутствовать единственный аксессор: get или set соответственно.

Несмотря на то что объявление свойства в интерфейсе очень похоже на объявление автоматически реализуемого свойства в классе, между ними все же имеется отличие. При объявлении в интерфейсе свойство не становится автоматически реализуемым. В этом случае указывается только имя и тип свойства, а его реализация предоставляется каждому реализующему классу. Кроме того, при объявлении свойства в интерфейсе не разрешается указывать модификаторы доступа для аксессоров. Например, аксессор set не может быть указан в интерфейсе как private.

Давайте рассмотрим пример:

using System;

namespace ConsoleApplication1

{

interface IUserInfo

{

string Name

{

get;

set;

}

}

class UI : IUserInfo

{

string myName;

public string Name

{

set

{

myName = value;

}

get

{

return myName;

}

}

}

class Program

{

static void Main()

{

UI user1 = new UI();

user1.Name = "Alexandr";

Console.ReadLine();

}

}

}

В данном примере в классе UI реализуется свойство интерфейса IUserInfo.

В интерфейсе можно также указывать индексаторы. Ниже приведена общая форма объявления интерфейсного индексатора:

*// Интерфейсный индексатор*

*тип\_элемента this[int индекс]{*

*get;*

*set;*

*}*

Как и прежде, в объявлении интерфейсных индексаторов, доступных только для чтения или только для записи, должен присутствовать единственный аксессор: get или set соответственно. Давайте добавим в предыдущий пример реализацию индексатора:

using System;

namespace ConsoleApplication1

{

interface IUserInfo

{

string Name

{

get;

set;

}

string this[int index]

{

get;

set;

}

}

class UI : IUserInfo

{

string myName;

public string Name

{

set

{

myName = value;

}

get

{

return myName;

}

}

public string this[int index]

{

set { myName = value; }

get { return myName; }

}

}

class Program

{

static void Main()

{

UI user1 = new UI();

user1.Name = "Alexandr";

user1[5] = "Dmitryi";

user1[10] = "Alexey";

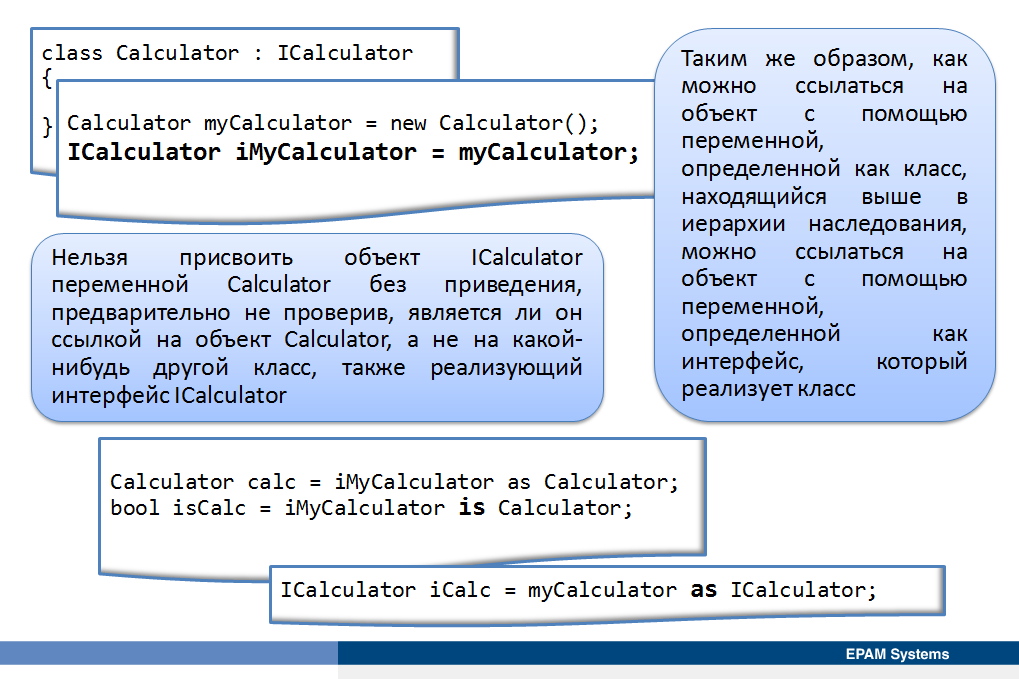
Console.ReadLine();

}

}

}

**Ссылки на объект через интерфейс**



Таким же образом, как можно ссылаться на объект с помощью переменной, определенной как класс, находящийся выше в иерархии наследования, можно ссылаться на объект с помощью переменной, определенной как интерфейс, который реализует класс. Например, можно сослаться на объект Calculator с помощью переменной ICalculator.

Calculator myCalculator = new Calculator();

ICalculator iMyCalculator = myCalculator;

Такое присваивание правомерно, потому что все объекты Calculator имплементируют интерфейс ICalculator. Однако обратное неверно, а, следовательно, нельзя присвоить объект ICalculator переменной Calculator без приведения, предварительно не проверив, является ли он ссылкой на объект Calculator, а не на какой-нибудь другой класс, также реализующий интерфейс ICalculator. Чтобы убедиться, что объект реализует интерфейс, можно использовать ключевые слова is и as, или проверить, что объект, на который ссылается интерфейс, экземпляр определенного класса.

Техника ссылок на объект через интерфейс полезна, поскольку позволяет определить методы, которые могут принимать различные типы в качестве параметров, при условии, что типы реализуют указанный интерфейс. В следующем примере, метод PerformAnalysis может принимать любой аргумент, реализующий интерфейс ICalculator.

int PerformAnalysis(ICalculator calculator)

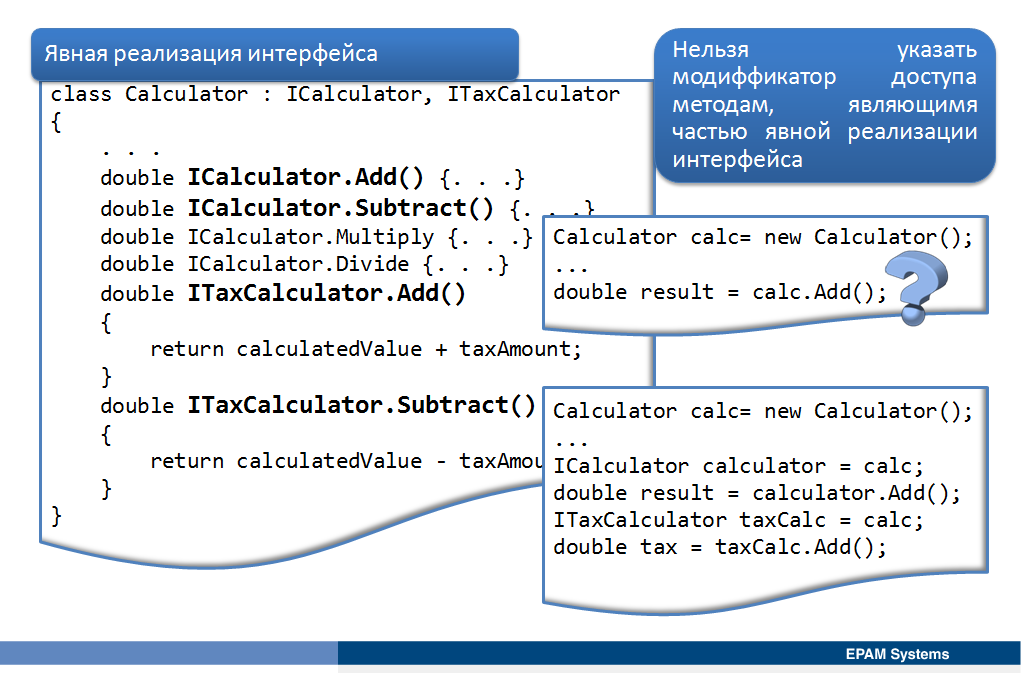
{

//...

}

При ссылке на объект через интерфейс, можно вызвать только методы, которые видны через интерфейс.

**Явная и неявная реализация интерфейса**



Ранее демонстрировались примеры классов, неявно реализующих интерфейсы. Если вернуться к интерфейсу IСalculator и реализующему его классу Calculator, очевидно, что в реализации методов класса Calculator не существует ничего, что говорило бы о том, что они являются частью интерфейса ICalculator.

interface ICalculator

{

double Add();

double Subtract();

double Multiply();

double Divide();

}

class Calculator : ICalculator

{

// The methods of the ICalculator interface return test data in this code.

#region ICalculator Members

public double Add()

{

return 0;

}

public double Subtract()

{

return 0;

}

public double Multiply()

{

return 0;

}

public double Divide()

{

return 0;

}

#endregion

}

Теперь предположим, что класс Calculator реализует несколько интерфейсов. Возможно существование во множестве интерфейсов методов, имеющих одно и то же имя и различную семантику. Например, расширенный класс калькулятор реализует интерфейс ITaxCalculator, включающий методы для сложения и вычитания суммы налога от стоимости, определенный следующим образом.

interface ITaxCalculator

{

double Add();

double Subtract();

}

Если реализовать этот интерфейс в классе Calculator, возникнет проблема, которая показана в следующем примере.

class Calculator : ICalculator, ITaxCalculator

{

public double Add()

{

return 0;

}

public double Subtract()

{

return 0;

}

public double Multiply()

{

return 0;

}

public double Divide()

{

return 0;

}

}

Это компилируемый код, однако методы каких интерфейсов ICalculator или ITaxCalculator реализуют методы Add и Subtract? Ответ несколько обескураживает – обоих! По умолчанию, C# не делает различия, какой метод интерфейса реализуется, таким образом одинаковый метод удовлетворяет обоим интерфейсам. Это же относится и к методу Subtract. Проблема в том, что реализации методов Add и Subtract, вероятно, должны быть различными для обоих интерфейсов. Искушение состоит в попытке переписать класс Calculator следующим образом.

class Calculator : ICalculator, ITaxCalculator

{

... // private fields not shown

// This is the Add method for ICalculator

public double Add()

{

return 0;

}

// This is the Subtract method for ICalculator

public double Subtract()

{

return 0;

}

public double Multiply()

{

return 0;

}

public double Divide()

{

return 0;

}

// This is the Add method for ITaxCalculator?

public double Add()

{

return calculatedValue + taxAmount;

}

// This is the Subtract method for ITaxCalculator?

public double Subtract()

{

return calculatedValue - taxAmount;

}

}

Однако, это уже неверный код, поскольку существуют реализации методов Add и Subtract, имеющие одинаковые сигнатуры. Чтобы решить возникшую проблему и устранить неоднозначность, можно реализовать интерфейсы явно. Для этого необходимо указать, какому интерфейсу принадлежит реализуемый метод.

class Calculator : ICalculator, ITaxCalculator

{

... // private fields not shown

double ICalculator.Add()

{

return 0;

}

double ICalculator.Subtract()

{

return 0;

}

double ICalculator.Multiply()

{

return 0;

}

double ICalculator.Divide()

{

return 0;

}

double ITaxCalculator.Add()

{

return calculatedValue + taxAmount;

}

double ITaxCalculator.Subtract()

{

return calculatedValue - taxAmount;

}

}

Наряду с наличием префикса имени интерфейса перед именем метода, есть еще одно тонкое различие в синтаксисе: методы, не помечены как public. Нельзя указать модиффикатор доступа методам, являющимя частью явной реализации интерфейса. Это приводит к еще одному интересному феномену. Если в коде создается переменная Calculator, нельзя вызывать методы, поскольку они не видны. В рассмотриваемом классе Calculator они являются private. В самом деле, это имеет смысл. Если бы методы были видны через класс Calculator, какой метод будет вызывать код в следующем примере - для интерфейса ICalculator или для интерфейса ITaxCalculator?

Calculator calc= new Calculator();

...

double result = calc.Add();

Итак, как получить доступ к этим методам? Ответ заключается в том, что необходимо добавить ссылку на объект Calculator через соответствующий интерфейс.

Calculator calc= new Calculator();

...

ICalculator calculator = calc;

double result = calculator.Add();

ITaxCalculator taxCalc = calc;

double tax = taxCalc.Add();

Хотя методы являются для класса закрытыми, они доступны через интерфейсы. Следует явно реализовывать интерфейсы, когда это возможно.

**Наследование интерфейсов**

Один интерфейс может наследовать другой. Синтаксис наследования интерфейсов такой же, как и у классов. Когда в классе реализуется один интерфейс, наследующий другой, в нем должны быть реализованы все члены, определенные в цепочке наследования интерфейсов.

Таким образом, интерфейсы могут быть организованы в иерархии. Как и в иерархии классов, в иерархии интерфейсов, когда какой-то интерфейс расширяет существующий, он наследует все абстрактные члены своего родителя (или родителей). Конечно, в отличие от классов, производные интерфейсы никогда не наследуют саму реализацию. Вместо этого они просто расширяют собственное определение за счет добавления дополнительных абстрактных членов.

Использовать иерархию интерфейсов может быть удобно, когда нужно расширить функциональность определенного интерфейса без нарушения уже существующих кодовых баз.

using System;

namespace ConsoleApplication1

{

public interface A

{

int Sum();

}

// Унаследованный интерфейс

public interface B : A

{

int Del();

}

class MyOperation : B

{

int x = 10, y = 5;

public int Sum()

{

return x + y;

}

public int Del()

{

return x / y;

}

}

class Program

{

static void Main()

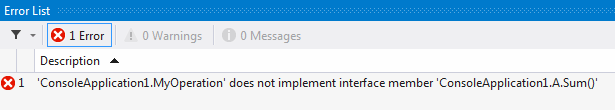
{

}

}

}

Обратите внимание, что класс MyOperation реализует методы обоих интерфейсов, иначе возникла бы ошибка при компиляции:



В отличие от классов, один интерфейс может расширять сразу несколько базовых интерфейсов, что позволяет проектировать очень мощные и гибкие абстракции.

**Проблемы сокрытия имен при наследовании интерфейсов**

**Множественным наследованием**называется ситуация, когда *класс* объявляет N классов ( N > 1 ) своими непосредственными родителями. В языке C# есть ограничения на *множественное наследование*. Ситуация здесь такая. Во-первых, у каждого класса родителем, хотя не всегда непосредственным, является *класс* object. Во-вторых, каждый *класс* может явно объявить один *класс* в качестве непосредственного родителя, а также объявить непосредственными родителями один или несколько интерфейсов. Таким образом, в C# допускается *множественное наследование* интерфейсов и одиночное (не считая наследования от класса object ) *наследование* классов.

Во многом ограничение множественного наследования классов связано с тем, что оно создает ряд проблем. Они остаются и при множественном наследовании интерфейсов, хотя становятся проще. Рассмотрим две основные проблемы - коллизию имен и *наследование* от общего предка.

#### Коллизия имен

Проблема **коллизии имен** возникает, когда два или более интерфейса имеют методы с одинаковыми именами и сигнатурой. Если сигнатуры разные, то это не приводит к конфликтам. Класс реализует методы обоих интерфейсов, и у него просто появляются *перегруженные методы*.

Но что следует делать классу наследнику в тех случаях, когда *сигнатуры методов* совпадают? Возможны две стратегии - **склеивание методов** и **переименование**.

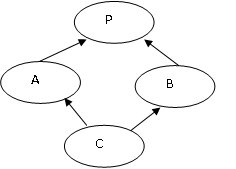
Стратегия склеивания применяется тогда, когда класс - наследник интерфейсов - полагает, что методы разных интерфейсов, имеющие одинаковое имя и сигнатуру, задают один и тот же метод, единая реализация которого и должна быть обеспечена наследником. В этом случае наследник строит единственную общедоступную ( public ) реализацию, соответствующую методам всех интерфейсов c единой сигнатурой.

Стратегия переименования исходит из того, что, несмотря на единую сигнатуру, методы разных интерфейсов должны быть реализованы по-разному. В этом случае необходимо переименовать конфликтующие методы. Конечно, переименование можно сделать в самих интерфейсах, но это неправильный путь: наследники не должны требовать изменений своих родителей - они сами должны меняться. Переименование методов интерфейсов иногда невозможно чисто технически, если интерфейсы являются встроенными или поставляются сторонними фирмами. К счастью, мы уже знаем, как производить переименование метода интерфейса в самом классе наследника. Для этого достаточно реализовать в классе методы разных интерфейсов как закрытые, а затем открыть их в классе с переименованием.

Итак, коллизия имен при множественном наследовании интерфейсов хотя и возможна, но разрешима. Разработчик класса может выбрать одну из двух возможных стратегий, наиболее подходящую для данного конкретного случая.

#### Наследование от общего предка

Проблема **наследования от общего предка** характерна, в первую очередь, для множественного наследования классов. Если класс C является наследником классов A и B, а те, в свой черед, являются наследниками класса P, то класс наследует свойства и методы своего предка P дважды: один раз получая их от класса A, другой - от B. Это явление называется еще дублирующим наследованием. Для классов ситуация осложняется тем, что классы A и B могли по-разному переопределить методы родителя и для потомков предстоит сложный выбор реализации. Ситуация дублирующего наследования показана на [рис](https://www.intuit.ru/studies/courses/1076/429/lecture/9734?page=2#image.5.3)



Для интерфейсов сама ситуация дублирующего наследования маловероятна, но возможна, поскольку интерфейс, как и любой класс, может быть наследником другого интерфейса. Поскольку у интерфейсов наследуются только сигнатуры, а не реализации, как в случае классов, проблема дублирующего наследования сводится к проблеме коллизии имен. По-видимому, естественным решением этой проблемы в данной ситуации является склеивание, когда методам, пришедшим разными путями от одного родителя, будет соответствовать единая реализация.

#### Склеивание и переименование

Приведу пример двух интерфейсов, имеющих методы с одинаковой сигнатурой, и класса - наследника этих интерфейсов, применяющего разные стратегии для конфликтующих методов. У нас уже определен интерфейс IStrings. Предположим, что существует интерфейс ITransform, подобный Istrings:

interface ITransform

{

/// <summary>

/// Преобразование

/// </summary>

/// <returns>результат преобразования</returns>

string Convert();

/// <summary>

/// Шифрование

/// </summary>

/// <param name="code">код </param>

/// <returns>результат шифрования</returns>

string Cipher(string[] code);

}

У этих интерфейсов имена и *сигнатуры методов* совпадают. Вот класс, наследующий оба интерфейса:

/// <summary>

/// Наследник двух интерфейсов,

/// у методов которых Convert и Cipher

/// сигнатуры совпадают.

/// Методы Cipher склеиваются,

/// Convert - переименовываются

/// </summary>

class TwoInterfaces:IStrings,ITransform

{

//Опущена часть класса, общая с классом SimpleText

//Реализация интерфейсов

string IStrings.Convert()

{

string res = "";

foreach (char sym in text)

if (sym != ' ') res += sym.ToString();

res = res.ToLower();

return res;

}

string ITransform.Convert()

{

string res = "";

for (int i = text.Length - 1; i >= 0; i--)

res += text[i];

return res;

}

//Переименование закрытых методов

public string ConvertOne()

{

return ((IStrings)this).Convert();

}

public string ConvertTwo()

{

return ((ITransform)this).Convert();

}

//Склеивание метода Cipher двух интерфейсов

public string Cipher(string[] code)

{

string s = text;

string res = "";

foreach (char sym in s)

{

int k = code[0].IndexOf(sym);

if (k >= 0) res += code[1][k];

else res += sym.ToString();

}

return res;

}

}

Для методов Cipher двух интерфейсов выбрана стратегия склеивания. Для методов Convert выбрана стратегия переименования. Методы интерфейсов реализованы как *закрытые методы*, а затем в классе объявлены два новых метода с разными именами, являющиеся обертками *закрытых методов* класса.

Приведем пример работы с объектами класса и интерфейсными объектами:

public void TestTextTwoInterfaces()

{

Console.WriteLine("Работа с объектом класса TwoInterfaces! ");

TwoInterfaces twoInterfaces = new TwoInterfaces(PAL);

Console.WriteLine("Исходный текст : " + PAL);

string text;

text = twoInterfaces.ConvertOne();

Console.WriteLine("Первое преобразование : " + text);

if (twoInterfaces.IsPalindrom())

Console.WriteLine("Это палиндром!");

text = twoInterfaces.ConvertTwo();

Console.WriteLine("Второе преобразование : " + text);

text = twoInterfaces.Coding();

Console.WriteLine("Шифрованный текст : " + text);

text = "Это простой текст!";

Console.WriteLine("Исходный текст : " + text);

twoInterfaces = new TwoInterfaces(text);

IStrings istrings;

ITransform itransform;

istrings = (IStrings)twoInterfaces;

itransform = (ITransform)twoInterfaces;

Console.WriteLine("Работа с объектом интерфейса IStrings!");

text = istrings.Convert();

Console.WriteLine("Преобразованный текст : " + text);

text = istrings.Cipher(CODE);

Console.WriteLine("Шифрованный текст : " + text);

Console.WriteLine("Работа с объектом интерфейса ITransform!");

text = itransform.Convert();

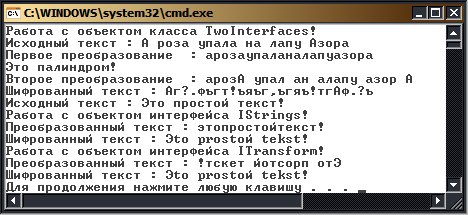
Console.WriteLine("Преобразованный текст : " + text);

text = itransform.Cipher(CODE);

Console.WriteLine("Шифрованный текст : " + text);

}

Результаты работы показаны на рисунке



**Анализ стандартных интерфейсов**

Рассмотрим несколько **встроенных интерфейсов**, являющихся частью *библиотеки FCL*. Они используются многими классами библиотеками, так же, как и классами, создаваемыми пользователем.

#### Упорядоченность объектов и 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, рассмотренном в ["Классы"](https://www.intuit.ru/studies/courses/1076/429/lecture/9726), 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.

Заметьте также, что при проверке на значение null используется отношение Equals, а не обычное равенство, которое будет переопределено.

##### Приведение типов и операции as и is

Давно пора привести отложенное пояснение операций isab и  , полезных при работе с объектами, тип которых может быть не определен. Операция is используется в логических выражениях. Логическое выражение

obj is T

истинно, если объект obj принадлежит типу T, и ложно в противном случае.

Оператор присваивания

obj = P as T;

присваивает объекту obj объект P, *приведенный к типу* T, если такое приведение возможно, иначе объекту присваивается значение null. Семантику as можно выразить следующим условным выражением:

(P is T) ? (T)P : (T)null

##### Порядок на классе Person

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,

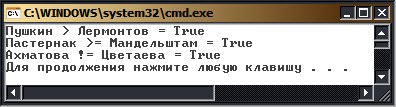
poet4.Fam, (poet3 >= poet4));

Console.WriteLine("{0} != {1} = {2}", poet5.Fam,

poet6.Fam, (poet5 != poet6));

}

Вот результаты работы этого теста.



#### Клонирование и интерфейс ICloneable

При ссылочном присваивании x = y, где x и y - объекты класса T, как уже много раз говорилось, происходит присваивание ссылок. Если до присваивания ссылка y была связана с объектом в динамической памяти, то после присваивания x будет ссылаться на этот же объект. С самим объектом ничего не происходит, никакая копия этого объекта не создается. Иногда требуется создать копию объекта, так, чтобы x и y ссылались на разные объекты.

**Клонированием** называется процесс создания копии объекта, а копия объекта называется **клоном**. Различают два типа клонирования: **поверхностное** ( shallow ) и **глубокое** ( deep ). При *поверхностном клонировании* копируется только один объект, копию которого необходимо создать. Все значимые поля клона получают значения, совпадающие со значениями полей объекта; все ссылочные поля клона являются ссылками на те же объекты, на которые ссылается и сам объект.

При глубоком клонировании копируется вся совокупность объектов, связанных взаимными ссылками. Представьте себе мир объектов, описывающих людей. У этих объектов могут быть ссылки на детей и родителей, учителей и учеников, друзей и родственников. В текущий момент может существовать большое число таких объектов, связанных взаимными ссылками. Достаточно выбрать один из них в качестве корня, и при его клонировании воссоздастся копия существующей структуры объектов.

*Глубокое клонирование* требует рекурсивной процедуры обхода существующей структуры объектов, тщательно отработанной во избежание проблемы зацикливания. В общем случае, когда есть несколько классов, являющихся взаимными клиентами, *глубокое клонирование* требует наличия в каждом классе рекурсивной процедуры. Эти процедуры взаимно вызывают друг друга. Хорошим упражнением является создание проекта, реализующего *глубокое клонирование* для ситуации, когда объекты разных классов связаны взаимными ссылками.

**Поверхностное клонирование** можно выполнить достаточно просто, поскольку оно автоматически поддерживается для всех классов, в том числе, создаваемых программистом. Для создания поверхностного клона достаточно вызвать метод MemberwiseClone, наследуемый от прародителя object. Единственное, что нужно помнить: этот метод защищен, он не может быть вызван непосредственно клиентом класса. Если класс хочет обеспечить возможность создания клона клиентами класса, ему необходимо в классе создать метод, представляющий собой обертку метода MemberwiseClone.

Давайте обеспечим эту возможность для класса Person, создав в нем соответствующий метод:

public Person StandartClone()

{

Person p = (Person)this.MemberwiseClone();

return(p);

}

Теперь клиенты класса могут легко создавать поверхностные клоны. Вот пример:

public void TestStandartClone()

{

Person mother = new Person("Петрова Анна");

Person daughter = new Person("Петрова Ольга");

Person son = new Person("Петров Игорь");

mother[0] = daughter;

mother[1] = son;

Person mother\_clone = mother.StandartClone();

Console.WriteLine("Дети матери: {0}",mother.Fam);

Console.WriteLine (mother[0].Fam);

Console.WriteLine (mother[1].Fam);

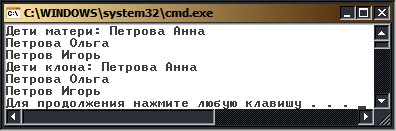
Console.WriteLine("Дети клона: {0}",mother\_clone.Fam);

Console.WriteLine (mother\_clone[0].Fam);

Console.WriteLine (mother\_clone[1].Fam);

}

При создании клона будет создана копия только одного объекта mother. Обратите внимание: при работе с полем children, задающим детей, используется *индексатор* класса Person, выполняющий индексацию по этому полю. Вот как выглядят результаты работы теста.



Если стандартное поверхностное клонирование нас не устраивает, то класс можно объявить наследником интерфейса *ICloneable* и реализовать метод Clone - единственный метод этого интерфейса. В нем можно реализовать полное *глубокое клонирование* или подходящую для данного случая модификацию.

Давайте расширим класс Person, сделав его наследником интерфейса *ICloneable*. Реализация метода Clone будет отличаться от стандартной реализации тем, что к имени объекта - полю Fam - будет приписываться слово " clone ". Вот как выглядит этот метод:

public object Clone()

{

Person clone = (Person)this.MemberwiseClone();

clone.fam = "clone\_" + fam;

return clone;

}

Эта реализация является слегка модифицированной версией стандартного *поверхностного клонирования*. Я добавил несколько строчек в тестирующую процедуру для проверки работы этой версии клона:

Person mother\_clone2 = (Person)mother.Clone();

Console.WriteLine("Дети клона\_2: {0}",mother\_clone2.Fam);

Console.WriteLine (mother\_clone2[0].Fam);

Console.WriteLine (mother\_clone2[1].Fam);

Все работает должным образом.

### Перечислимость объектов и интерфейсы

Если *класс* можно рассматривать как некоторый *контейнер* перечислимых объектов, то для того, чтобы *класс* мог возвращать эти объекты в цикле for each, *класс* должен быть наследником интерфейса *IEnumerable* или иметь в своем составе итераторы - методы, возвращающие результат типа *IEnumerable*.

#### Интерфейс перечислимости IEnumerable

У этого интерфейса всего один метод - GetEnumerator(). У метода нет никаких *входных аргументов*, так что, кажется, организовать перечислимость объектов класса достаточно просто - написать реализацию одного метода. Это действительно просто, но требует понимания процесса перечислимости. Первая сложность состоит в том, что метод GetEnumerator синтаксически определен следующим образом:

IEnumerator GetEnumerator()

Это означает, что в результате вызова метода должен возвращаться интерфейсный объект, принадлежащий интерфейсу *IEnumerator* - еще одному интерфейсу, связанному с перечислимостью. Метод GetEnumerator требует создания объекта перечислителя - объекта, реализующего методы интерфейса *IEnumerator*. У интерфейса *IEnumerator* несколько методов, и в совокупности именно они и позволяют организовать процесс перечисления.

Как же получить интерфейсный объект типа *IEnumerator*? В принципе эту задачу мы уже умеем решать, что неоднократно демонстрировалось в примерах этой лекции. Нужно объявить интерфейсный объект, затем создать объект класса, являющегося наследником интерфейса *IEnumerator*, и далее интерфейсный объект связать с объектом класса, используя приведение типа. Но такой подход все равно требует создания объекта, реализующего методы интерфейса *IEnumerator*, что не так просто.

К счастью, во многих ситуациях есть более простой путь. Пусть нам нужно организовать перечислимость объектов некоторого класса. У этого класса есть поля, являющиеся объектами, для которых перечислимость уже обеспечена, например, одним из полей класса является массив. Все классы массивы независимо от типа элементов реализуют перечислимость элементов массива, и для них определен метод GetEnumerator. Тогда метод GetEnumerator, который необходимо создать для класса, будет сводиться к вызову метода GetEnumerator для поля класса, представляющего собой перечислимый контейнер элементов. В этом случае все будет совсем просто.

Давайте построим пример, моделирующий такой подход. 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();

}

На описание метода в теге summary потребовалось больше времени и строчек текста, чем на *тело метода*, состоящее из одной строчки. Но на описание не стоит жалеть усилий! Задача решена, и можно попробовать протестировать работу с объектами класса. Добавим в класс Testing соответствующий метод:

public void TestEnumeration()

{

int size = 10;

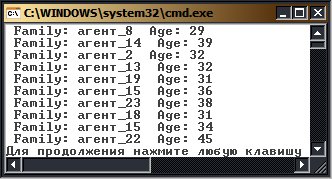
Persons agents = new Persons(size);

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 показаны на [рис.](https://www.intuit.ru/studies/courses/1076/429/lecture/9734?page=4#image.5.8).



#### Итераторы

Нам удалось достаточно просто организовать перечислимость в классе за счет того, что в классе был определен контейнер со встроенным методом GetEnumerator. А что, если такого контейнера нет, или таких контейнеров несколько и хотелось бы организовать перечисление по каждому из контейнеров? Для решения подобных задач в язык C#, начиная с версии 2.0, встроен мощный механизм итераторов, существенно облегчающих задачу перечислимости объектов класса и открывающих новые возможности, которые нельзя реализовать только за счет наследования интерфейса *IEnumerable*.

**Итератором** называется метод класса, возвращающий в качестве результата интерфейсный объект типа *IEnumerable*. В теле итератора должен присутствовать оператор языка *yield*, имеющий следующий синтаксис:

yield return <выражение> ;

При выполнении итератора автоматически создается контейнер, в который добавляется объект при каждом выполнении оператора *yield*. Добавляемый в контейнер элемент определяется выражением оператора *yield*. Порядок выполнения операторов *yield* определяет порядок перечислимости элементов контейнера. Оператор *yield* может быть задан в форме

yield break;

В такой форме он сигнализирует об окончании заполнения контейнера элементами.

Итераторов в классе может быть несколько. Каждый из них создает свой вид перечислимости объектов класса. Итератору не требуется, чтобы в классе существовал контейнер со встроенным методом перечислимости, хотя итератор может использовать контейнер, изменяя порядок перечислимости, заданный для контейнера по умолчанию.

Давайте добавим в класс Persons два итератора. Первый из этих итераторов будет использовать существующий в классе контейнер, изменяя заданный порядок перечислимости элементов на обратный.

/// <summary>

/// Перечисление элементов массива container

/// в обратном порядке

/// </summary>

/// <returns>интерфейсный объект</returns>

public IEnumerable ReverseIterator()

{

for (int i = container.Length - 1; i >= 0; i--)

yield return container[i];

}

Оператор *yield*, работающий в цикле, первым возвратит последний элемент массива, а последним - первый.

Создадим еще один итератор, никак не связанный с элементами массива поля container. Зададим перечисление, содержащее имена известных языков программирования:

public enum ProgrammingLanguages

{

Fortran, Algol, Cobol, Simula, Pascal,

Ada, C, CPlusPlus, Eiffel, Java, CSharp

}

Построим теперь итератор, в котором создадим локальную переменную для данного перечисления и используем ее, чтобы поместить в контейнер элементы перечисления.

/// <summary>

/// Итератор. Создает объект перечисления

/// и строит контейнер,

/// помещая в него элементы перечисления

/// </summary>

/// <returns>интерфейсный объект</returns>

public IEnumerable LanguagesIterator()

{

ProgrammingLanguages lang;

int i = 0;

while (true)

{

lang = (ProgrammingLanguages)i;

yield return lang;

if (lang.ToString() == "CSharp") yield break;

i++;

}

}

В цикле оператор *yield* возвращает элементы перечисления до тех пор, пока не встретится элемент со значением "CSharp". Давайте теперь протестируем то, что у нас получилось. Добавим несколько строк кода в уже рассмотренный метод TestEnumeration:

Console.WriteLine("Обратный порядок перечисления");

foreach(Person agent in agents.ReverseIterator())

Console.WriteLine(agent.ToString());

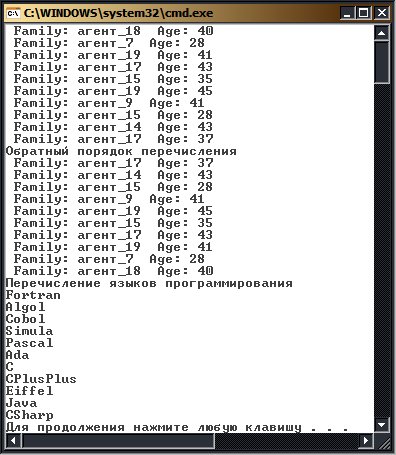
Console.WriteLine("Перечисление языков программирования");

foreach (ProgrammingLanguages lang in

agents.LanguagesIterator())

Console.WriteLine(lang.ToString());

Обратите внимание: когда клиент организует цикл foreach, он указывает не только имя объекта класса, но и имя итератора, организующего перечислимость. Результаты работы теста показаны на [рис](https://www.intuit.ru/studies/courses/1076/429/lecture/9734?page=5#image.5.9).



#### Интерфейс IEnumerator

Итераторы и контейнеры со встроенным методом GetEnumerator позволяют достаточно просто организовать перечислимость объектов класса, позволяя клиентам класса использовать цикл foreach для доступа к объектам на чтение. В основе предоставляемого механизма доступа лежат методы интерфейса *IEnumerator*. Используя итератор или встроенный метод GetEnumerator, можно не вникать в суть того, как, например, метод GetEnumerator создает объект интерфейса *IEnumerator*. Тем не менее в ряде случаев полезно при организации перечислимости объектов класса объявить класс наследником интерфейса *IEnumerator* и реализовать самостоятельно методы этого интерфейса.

Реализация в *классе свойств* и методов этого интерфейса позволяет клиентам класса самостоятельно организовать работу с объектами класса как с коллекцией. Обычно это не делается, а используются возможности, предоставляемые циклом for each, который скрывает детали применения свойств и методов интерфейса.

Общая схема перечисления объектов в коллекции такова. Коллекция рассматривается как список с курсором. Вводится понятие позиции курсора, указывающей на текущий объект из списка. В начальный момент курсор установлен перед первым объектом списка и текущий объект не определен. Курсор можно передвигать к следующему элементу списка. Можно прочесть значение элемента, на который указывает курсор. Методы интерфейса *IEnumerator* позволяют реализовать такую схему работы с коллекцией объектов. Если методы интерфейса предполагается сделать доступными для клиентов, то они реализуются как *открытые методы*.

Метод bool MoveNext () передвигает курсор к следующему элементу списка, так что после его первого вызова курсор будет указывать на первый элемент списка. После его вызова, когда курсор указывает на *последний элемент списка*, текущий элемент снова становится неопределенным, а позиция курсора смещается за последний элемент. Метод MoveNext возвращает значение true, когда курсор указывает на элемент списка, и возвращает false, когда курсор вне списка элементов.

Метод void Reset() возвращает курсор в начальную позицию перед первым элементом списка.

Метод-свойство Object Current { get } возвращает текущий элемент списка. Выбрасывается исключение InvalidOperationException,если при вызове свойства Current курсор установлен перед или после списка элементов.

Приведу теперь пример класса, реализующего интерфейс *IEnumerator*. Давайте построим класс People, который будет наследником класса Persons и интерфейса *IEnumerator*. Будучи наследником класса Persons, новый класс будет обладать всеми возможностями родительского класса, в том числе перечислимостью объектов. Но как наследник интерфейса *IEnumerator* он реализует методы интерфейса и предоставит своим клиентам новые возможности.

/// <summary>

/// Наследник класса Persons

/// Реализует методы интерфейса IEnumerator

/// Предоставляет клиентам класса возможность

/// собственной организации перебора объектов

/// </summary>

class People:Persons,IEnumerator

{

int position = -1;

public People() : base() { }

public People(int size): base(size) { }

public People(Person[] people) : base(people) { }

/// <summary>

/// Передвигает курсор к следующему элементу в контейнере

/// Возвращает true, если курсор установлен на элементе списка,

/// false - в противном случае

/// </summary>

/// <returns></returns>

public bool MoveNext()

{

if (position < size) position++;

return (position < size);

}

/// <summary>

/// Восстанавливает текущее положение курсора

/// в контейнере объектов

/// </summary>

public void Reset()

{

position = -1;

}

/// <summary>

/// Возвращает текущий объект, если курсор установлен

/// на элементе списка

/// Выбрасывает исключение InvalidOperationException,

/// когда курсор вне списка

/// </summary>

public object Current

{

get

{

try

{

return container[position];

}

catch (IndexOutOfRangeException)

{

throw new InvalidOperationException();

}

}

set

{

try

{

container[position] = (Person)value;

}

catch (IndexOutOfRangeException)

{

throw new InvalidOperationException();

}

}

}

У класса три собственных конструктора, которые сводятся к вызову конструкторов родителя, поле position, необходимое для организации доступа, и три *открытых метода*, реализующих методы интерфейса *IEnumerator*. Методы устроены достаточно просто и в особых комментариях не нуждаются. Но стоит обратить внимание на реализацию метода-свойства Current. В самом интерфейсе для этого свойства указан только метод get. В реализации я добавил и реализацию метода set. Это сразу же расширяет возможности клиента. Если при работе с методом foreach объекты доступны только для чтения, то теперь свойство Current позволяет изменять элементы коллекции. Тест, построенный для работы с новым классом People, демонстрирует все эти возможности.

public void TestPeople()

{

People people = new People();

foreach (Person person in people)

Console.WriteLine(person.ToString());

Console.WriteLine("Работа клиента с коллекцией!");

try

{

Person person;

person = new Person("James Bond", 44);

people.MoveNext();

people.Current = person;

people.Reset();

while (true)

{

people.MoveNext();

person = (Person)people.Current;

Console.WriteLine(person.ToString());

}

}

catch (InvalidOperationException)

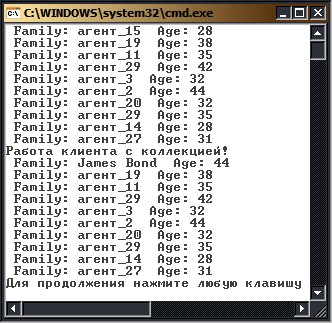
{

people.Reset();

}

}

Результаты работы этого теста показаны на [рис.](https://www.intuit.ru/studies/courses/1076/429/lecture/9734?page=5#image.5.10)



Обратите внимание: клиент сумел изменить объект, полученный в результате перечисления, добавив "Джеймса Бонда" в список агентов.

1. Если не предоставить хотя бы один, компилятор генерирует конструктор по умолчанию. [↑](#footnote-ref-1)
2. Хотя нельзя создать типы структур, которые наследуются от других типов структур или классов, но тип структура может реализовывать интерфейс. [↑](#footnote-ref-2)