|  |
| --- |
| , RD Dep. |
| Конспект и раздаточный материал  NET.C#.06 Наследование. Интерфейсы и абстрактные классы |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial version | Анжелика Кравчук |  |  |  |
| 1.1 | Review and corrections. | Владимир Тихон |  |  |  |

Contents

[1. Урок 1: Использование наследования для определения новых ссылочных типов 3](#_Toc301363542)

[1.1. Что такое наследование? 3](#_Toc301363543)

[1.2. Иерархия наследования .NET Framework 5](#_Toc301363544)

[1.3. Переопределение и сокрытие методов 5](#_Toc301363545)

[1.4. Вызов методов и конструкторов базового класса 7](#_Toc301363546)

[1.5. Присваивание и ссылка на классы в иерархии наследования 8](#_Toc301363547)

[1.6. Полиморфизм 10](#_Toc301363548)

[1.7. Определение герметизированных классов и методов 11](#_Toc301363549)

[1.8. Демонстрация: Использование наследования для построения новых ссылочных типов 12](#_Toc301363550)

[2. Урок 2: Определение и реализация интерфейсов 12](#_Toc301363551)

[2.1. Что такое интерфейс? 13](#_Toc301363552)

[2.2. Создание и реализация интерфейсов 14](#_Toc301363553)

[2.3. Ссылки на объект через интерфейс 16](#_Toc301363554)

[2.4. Явная и неявная реализация интерфейса 17](#_Toc301363555)

[2.5. Демонстрация: Создание интерфейса 20](#_Toc301363556)

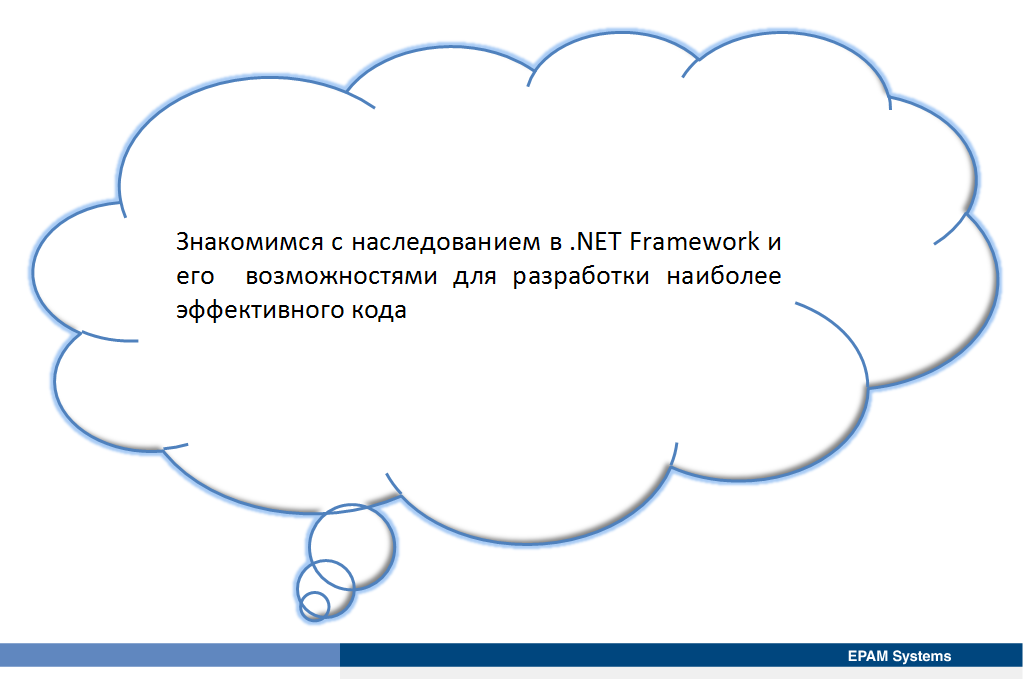
[3. Урок 3: Определение абстрактных классов 20](#_Toc301363557)

[3.1. Что это абстрактный класс? 21](#_Toc301363558)

[3.2. Что такое абстрактный метод? 23](#_Toc301363559)

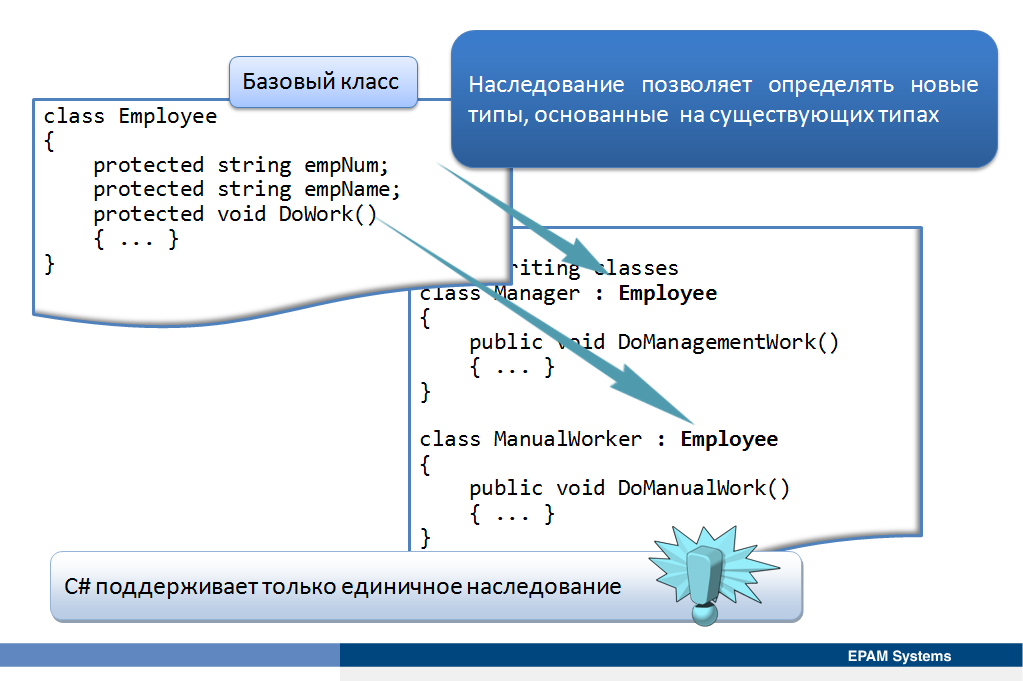
[3.3. Демонстрация: Создание абстрактного класса 24](#_Toc301363560)

# Урок 1: Использование наследования для определения новых ссылочных типов



Урок описывает наследование в .NET Framework и помогает понять, как можно использовать его механизмы для быстрой разработки наиболее эффективного кода с меньшим количеством ошибок. Разработка иерархии объектов является важным процессом. Иерархия объектов должна быть хорошо разработана, при этом следует избегать дублирования кода. Понимание наследования в .NET Framework имеет для этого процесса основополагающее значение.

## Что такое наследование?



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 связаные 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, необходимоуточнить, что руководители и рабочие обладают одинаковыми особенностями, имеяя 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()

{

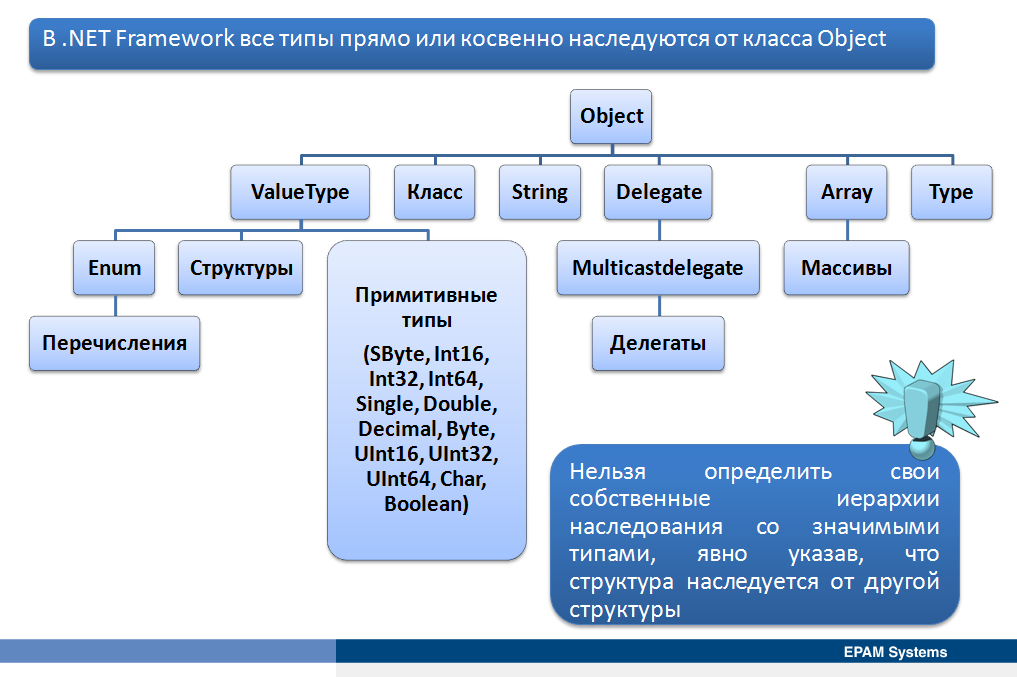
...

}

}

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

## Иерархия наследования .NET Framework

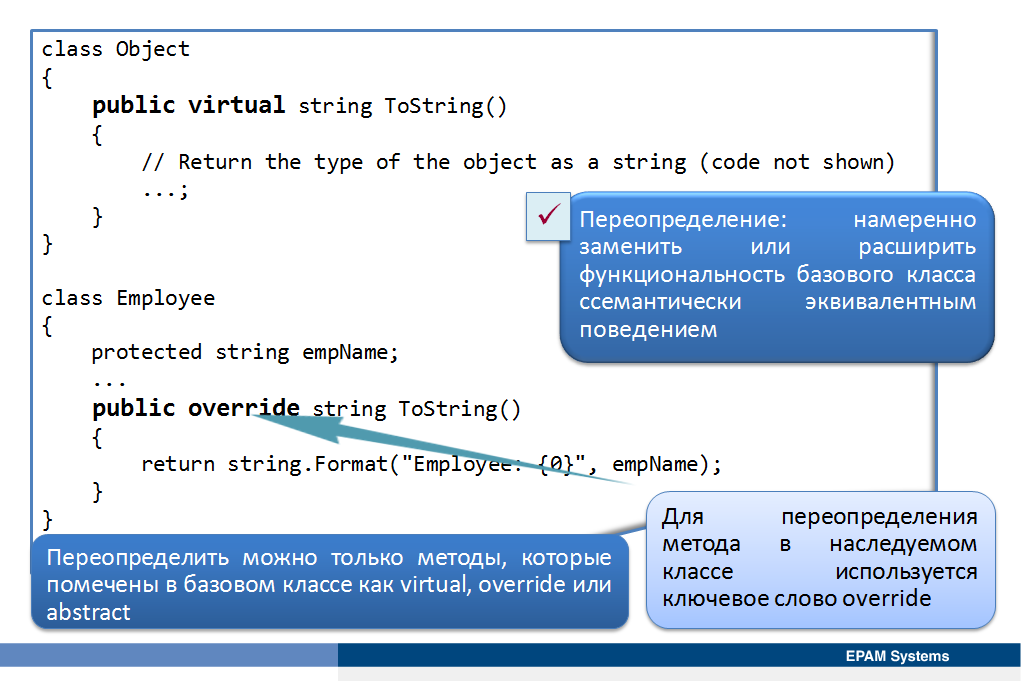


В .NET Framework все типы прямо или косвенно наследуются от класса Object пространтсва имен System. Класс Object обеспечивает полезную для всех типов функциональность, например, с помощью методов ToString и Equals. При создании нового ссылочного типа, такого как класс, он наследуется непосредственно 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.

<http://go.microsoft.com/fwlink/?LinkId=192936>

## 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

...

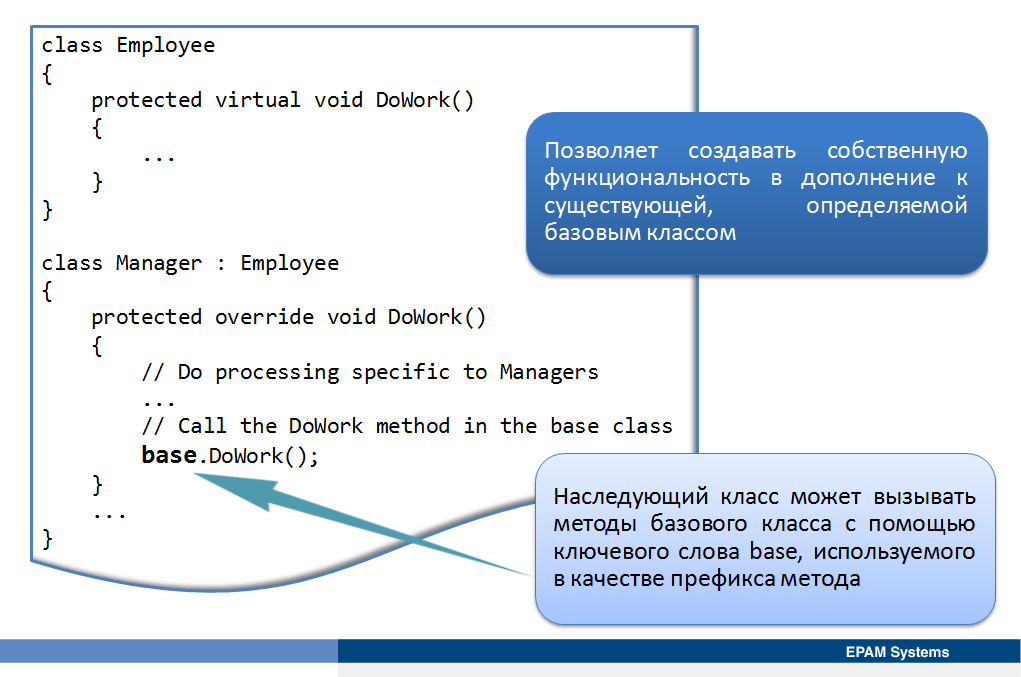
}

...

}

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()

{

...

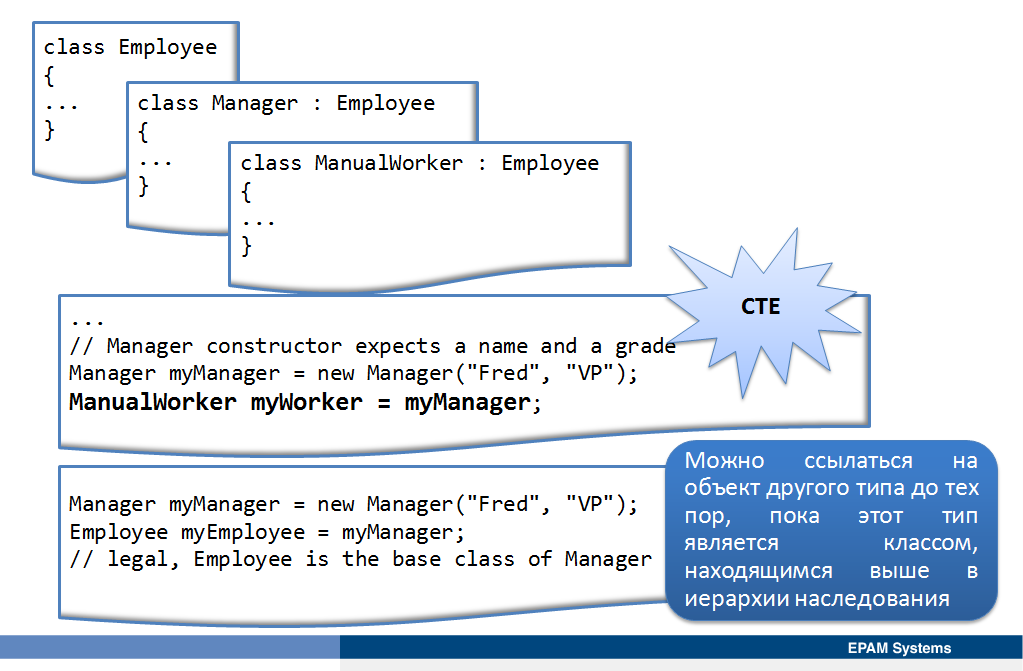
}

...

}

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

...

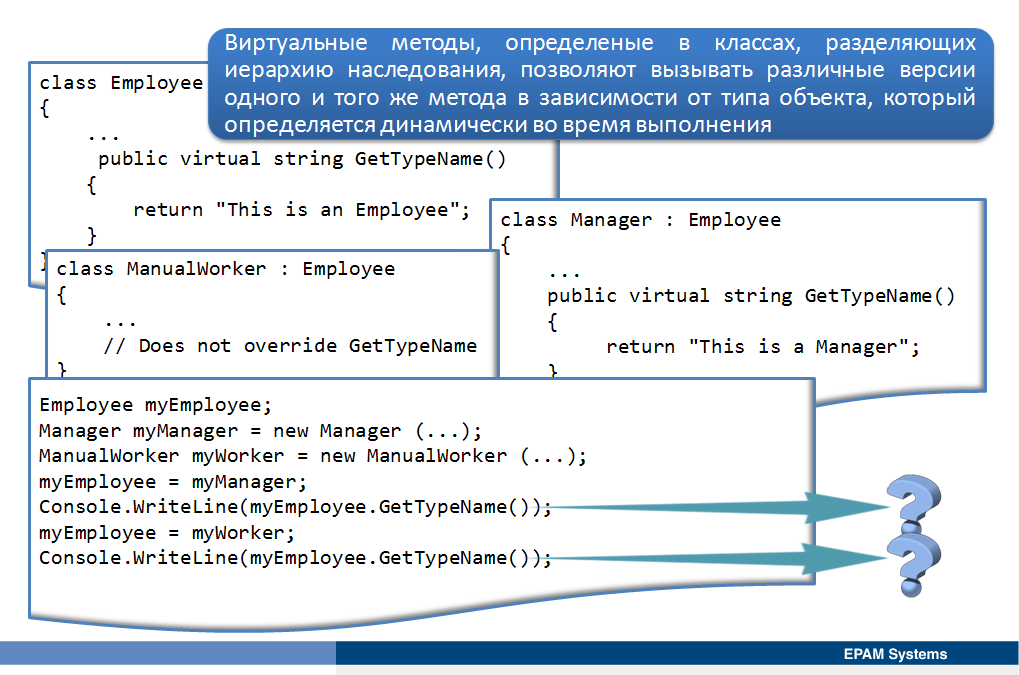
myManagerAgain = myEmployee as Manager;

// returns null - myEmployee is a ManualWorker

<http://go.microsoft.com/fwlink/?LinkId=192941>

<http://go.microsoft.com/fwlink/?LinkId=192942>

## 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;

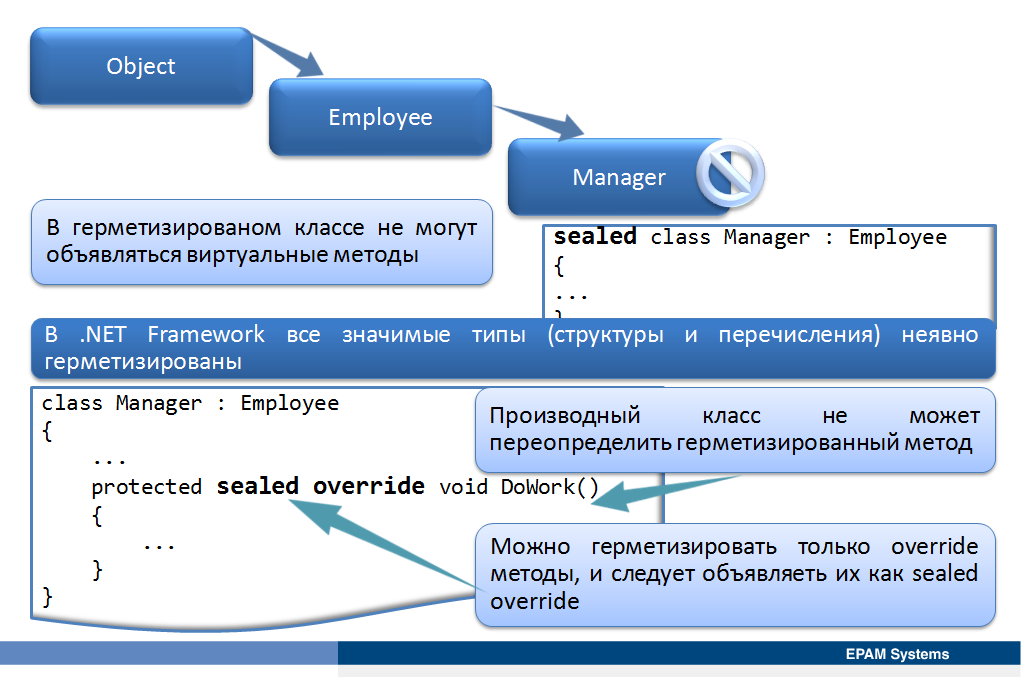
Console.WriteLine(myEmployee.GetTypeName()); // Manager

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



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()

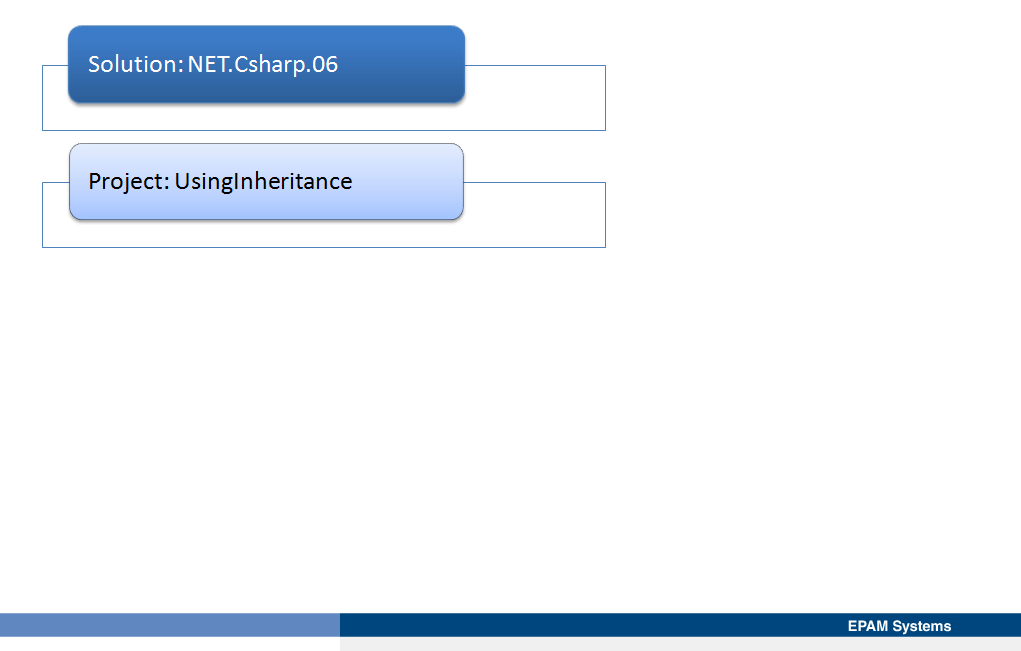
{

...

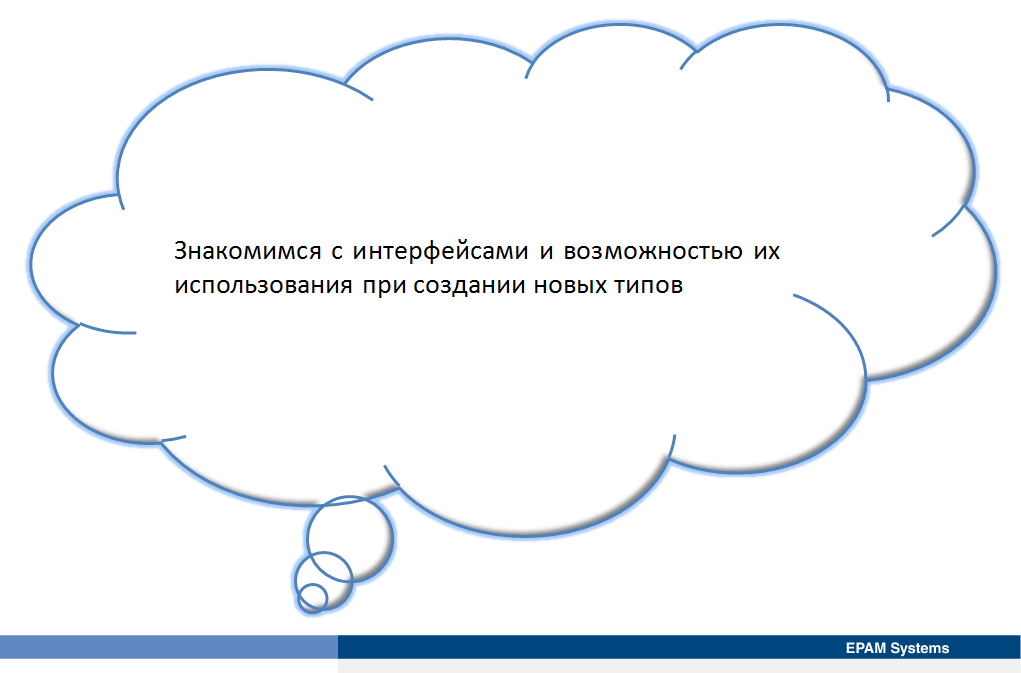
}

}

## Демонстрация: Использование наследования для построения новых ссылочных типов

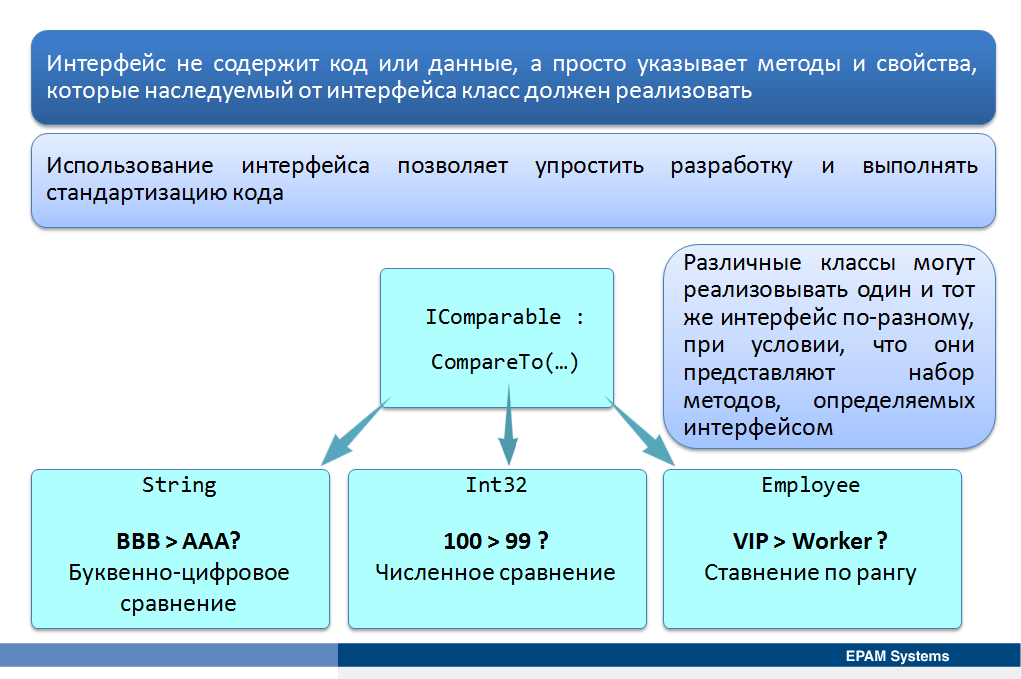


# Урок 2: Определение и реализация интерфейсов



Урок знакомит с интерфейсами и описывает, как их можно использовать для стандартизации кода. Интерфейсы позволяют явно определить соглашения, определяющие методы, которые представляют объекты. Можно реализовать несколько интерфейсов, чтобы указать, что код может выполнять несколько функций. Например, реализовав интерфейс IComparable, можно тем самым указать, что два объекта можно проверить на равенство, а интерфейс IDisposable, чтобы указать, что объект может быть очищен.

## 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.

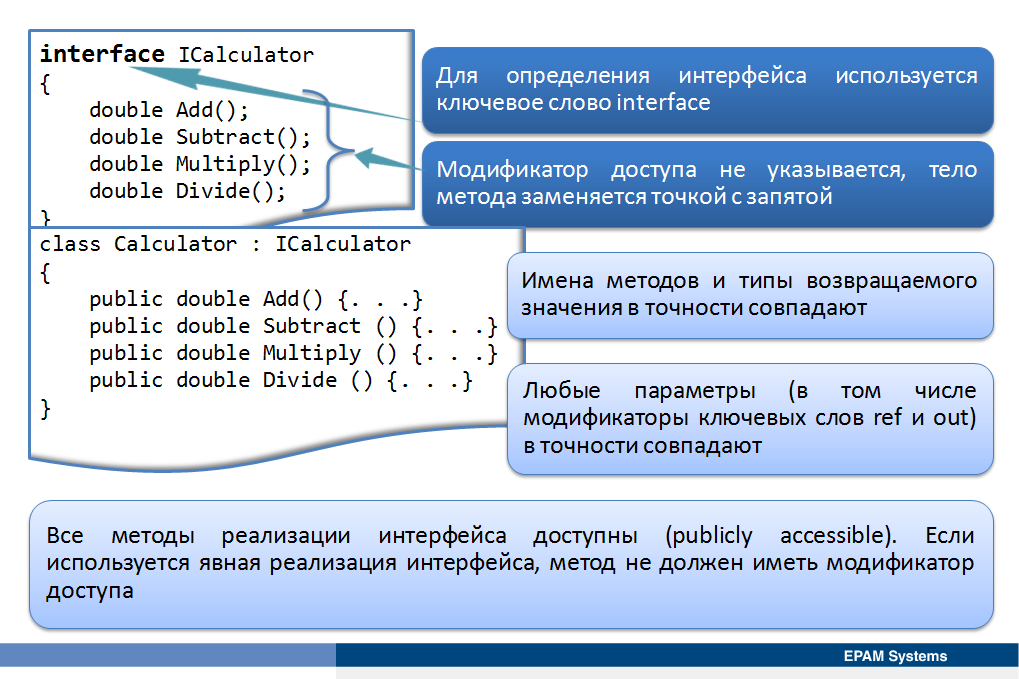
Интерфейсы позволяют указать функциональность, которую должен реализовывать класс. Способ, который класс выбирает для осуществления этой функциональности касается только класса. Различные классы могут реализовывать один и тот же интерфейс по-разному, при условии, что они представляют набор методов, определяемых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

{

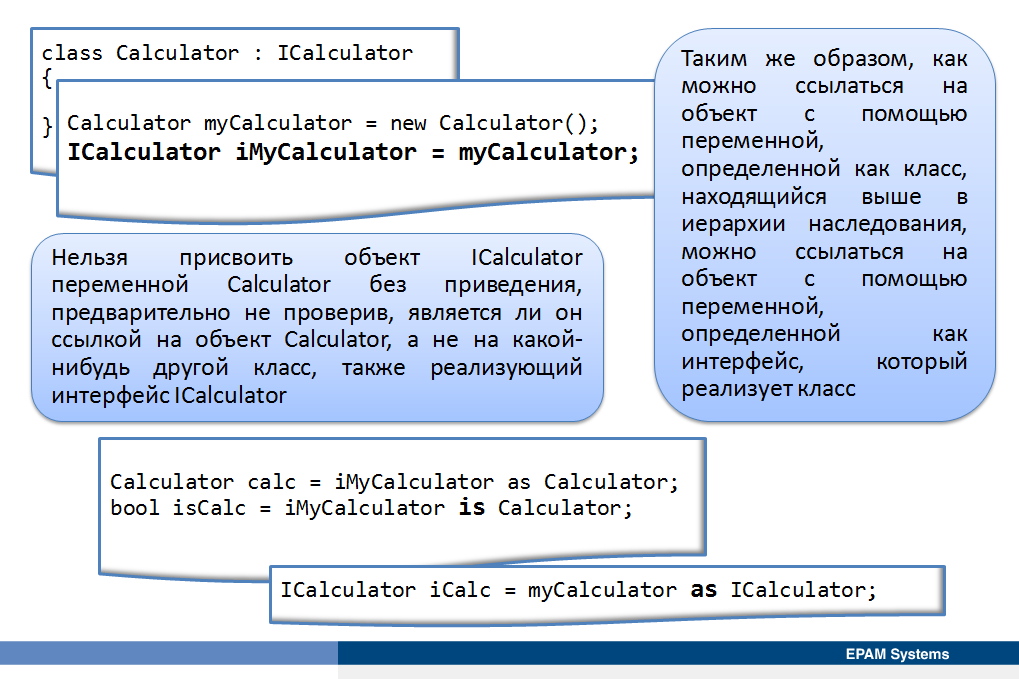
//...

}

#endregion

}

## 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)

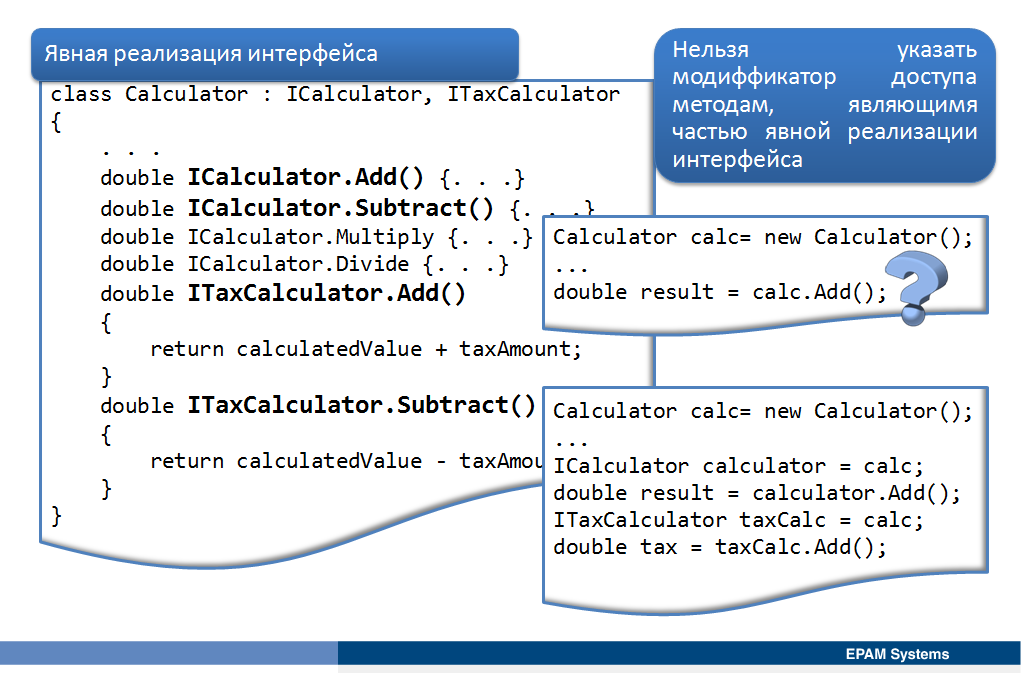
{

//...

}

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();

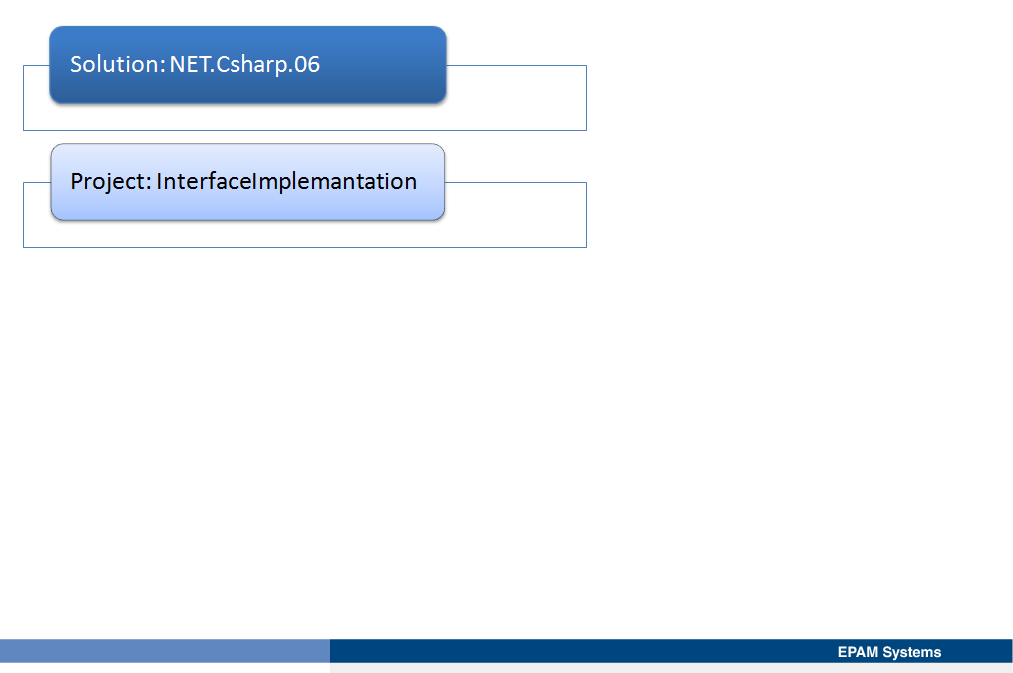
ITaxCalculator taxCalc = calc;

double tax = taxCalc.Add();

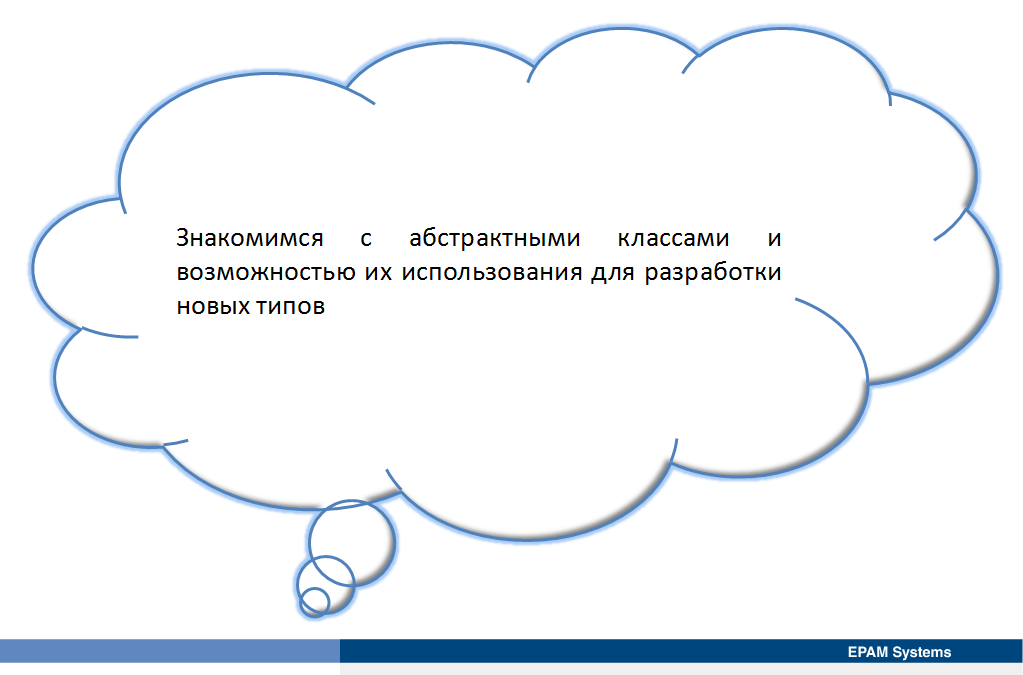
Хотя методы являются для класса закрытыми, они доступны через интерфейсы. Следует явно реализовывать интерфейсы, когда это возможно.

<http://go.microsoft.com/fwlink/?LinkId=192938>

## Демонстрация: Создание интерфейса

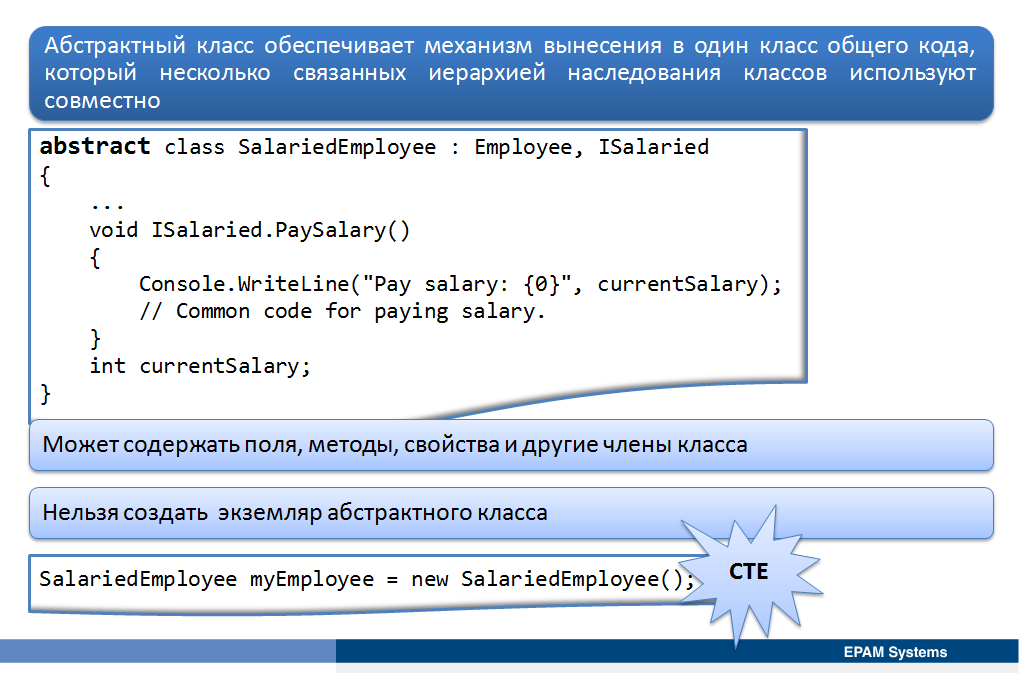


# Урок 3: Определение абстрактных классов



Урок знакомит с абстрактными классами и возможностями их использования для сокращения дублирования кода, ускорения разработки и уменьшения риска внедрения ошибкок, вызываемых дублированием кода. Абстрактные классы объединяют некоторые из свойств наследования объекта с некоторыми из свойств интерфейсов. Можно использовать абстрактные классы для сокращения дублирования кода, используя реализации методов по умолчанию.

## Что это абстрактный класс?



Абстрактный класс обеспечивает механизм вынесения в один класс общего кода, который несколько связанных иерархией наследования классов используют совместно. Например, с помощью иерархии классов Employees, показаной ранее, можно создать несколько классов для представления различных типов сотрудников, таких, как Manager и ManualWorker. В зависимости от числа типов сотрудников, которые нужно представить, число классов может увеличиваться. В таких ситуациях, как эта, весьма характерно, что для некоторых элементов этих классов имеются одни и те же реализации. Например, классы ManualWorker и Manager в следующем примере как реализуют интерфейс ISalaried, так и обеспечивают метод PaySalary, реализация которого совпадает в обоих классах.

interface ISalaried

{

void PaySalary();

}

class ManualWorker : Employee, ISalaried

{

...

void ISalaried.PaySalary()

{

Console.WriteLine("Pay salary: {0}", currentSalary);

// Code for paying salary.

}

}

class Manager : Employee, ISalaried

{

...

void ISalaried.PaySalary()

{

Console.WriteLine("Pay salary: {0}", currentSalary);

// Same code as ManualWorker for paying salary.

}

}

Дублирование кода это предупреждающий знак. Если это возможно, следует реорганизовать код так, чтобы избежать этого дублирования и сократить расходы на его обслуживание. Одним из способов достижения этого рефакторинга – вложить общую имплементацию в новый класс, который создается специально для этой цели. В действительности, можно вставить новый класс в иерархию классов.

class SalariedEmployee : Employee, ISalaried

{

...

void ISalaried.PaySalary()

{

Console.WriteLine("Pay salary: {0}", currentSalary);

// Common code for paying salary.

}

int currentSalary;

}

class ManualWorker : SalariedEmployee, ISalaried

{

...

}

class Manager : SalariedEmployee, ISalaried

{

...

}

Это хорошее, но все еще не совсем верное решение, поскольку можно создавать экземпляры класса SalariedEmployee (и класса Employee в этом отношении), что в действительности не имеет смысла. Класс SalariedEmployee существует для обеспечения общей реализации по умолчанию. Его единственной целью является быть базовым. Класс SalariedEmployee является абстракцией общей функциональности, а не сущности. Не желательно, чтобы разработчики имели возможность создавать экземпляра этого класса. Чтобы заявить, что создание экземпляров класса не допускается, можно объявить, что класс является абстрактным с помощью модификатора abstract.

abstract class SalariedEmployee : Employee, ISalaried

{

...

void ISalaried.PaySalary()

{

Console.WriteLine("Pay salary: {0}", currentSalary);

// Common code for paying salary.

}

int currentSalary;

}

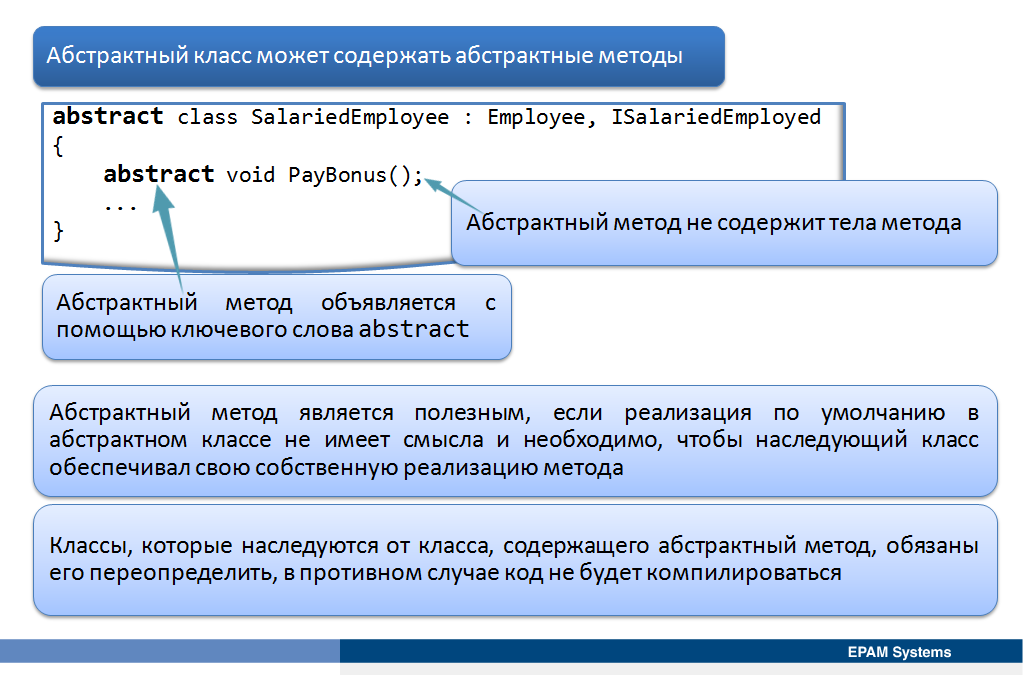
Если попытаться создать сейчас экземпляр класса SalariedEmployee, код не будет компилироваться.

SalariedEmployee myEmployee = new SalariedEmployee(); // Illegal

Интерфейсы сходны с абстрактными классам, что иногда порождает проблему выбора «интерфейс или абстрактный класс». Следующая таблица содержит сравнение возможностей этих пользовательских типов, для того, чтобы помочь сделать правильный выбор между ними.

|  |  |
| --- | --- |
| **Абстрактные классы** | **Интерфейсы** |
| Не могут быть созданы напрямую, но могут содержать конструктор, который вызывается в классе-наследнике | Не могут содержать конструктор |
| Абстрактный класс может быть так дополнен элементами, что это не повлияет на его классы-наследники | Если в интерфейс помещаются дополнительные элементы, все классы, которые его реализуют, должны быть дополнены |
| Может хранить данные в полях | Не может хранить данных |
| Виртуальные элементы могут содержать базовую реализацию. Допустимы невиртуальные элементы | Все элементы являются виртуальными и не включают реализацию |
| Класс может наследоваться от единственного абстрактного класса | Класс может реализовывать несколько интерфейсов |
| Класс-наследник может переопределить только некоторые элементы абстрактного класса | Класс, который реализует интерфейс, должен реализовать все элементы интерфейса |
| Наследование поддерживается только для классов | Интерфейс может быть реализован структурой |

## Что такое абстрактный метод?



Абстрактный класс может содержать абстрактные методы. Абстрактный метод похож на виртуальный за исключением того, что он не содержит тела метода. Производный класс должен переопределить этот метод. В следующем примере в классе SalariedEmployee определяется метод PayBonus как абстрактный. Все сотрудники могут разделять некоторые методы, такие как метод PayBonus, но они должны иметь свои собственные реализации метода PayBonus, потому что логика для выплаты премий может быть различной для различных типов сотрудников. Абстрактный метод является полезным, если реализацию по умолчанию в абстрактном классе не имеет смысла и необходимо, чтобы наследующий класс обеспечивал свою собственную реализацию метода.

abstract class SalariedEmployee : Employee, ISalariedEmployed

{

abstract void PayBonus();

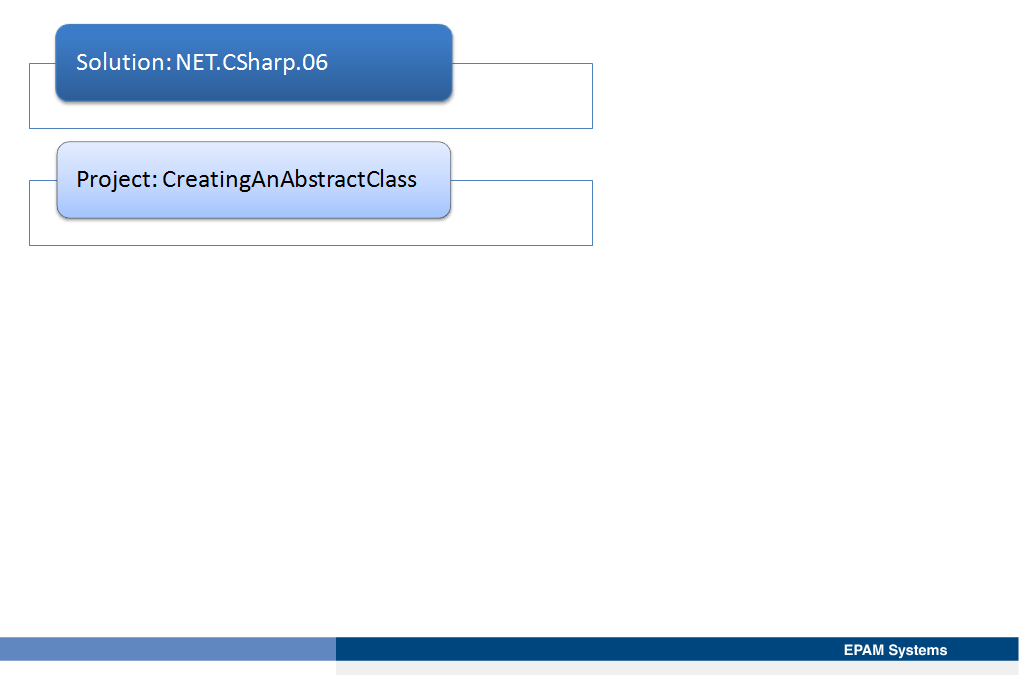
...

}

При определении абстрактного метода используется ключевое слово abstract и тело метода заменяется точкой с запятой. В отличие от интерфейса к абстрактным методам можно добавить модиффикатор доступа. Если попытаться добавить абстрактный метод в неабстрактный класс, возникнет ошибка компиляции.

<http://go.microsoft.com/fwlink/?LinkId=192939>

## Демонстрация: Создание абстрактного класса



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)