|  |
| --- |
| , RD Dep. |
| Конспект и раздаточный материал  NET.C#.05 Инкапсуляция данных и методов. Перегрузка операций |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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: Управление видимостью членов типа 3](#_Toc301349547)

[1.1. Что такое инкапсуляция? 3](#_Toc301349548)

[1.2. Сравнение private и public членов типа 4](#_Toc301349549)

[1.3. Сравнение internal и public типов 6](#_Toc301349550)

[2. Урок 2: Создание и использование свойств 8](#_Toc301349551)

[2.1. Что такое свойство? 8](#_Toc301349552)

[2.2. Определение свойства 9](#_Toc301349553)

[2.3. Автоматические свойства 10](#_Toc301349554)

[2.4. Создание экземпляра объекта с помощью свойства 12](#_Toc301349555)

[2.5. Определение свойств в интерфейсе 14](#_Toc301349556)

[2.6. Рекомендации по определению и использованию свойств 15](#_Toc301349557)

[2.7. Демонстрация: Использование свойств 16](#_Toc301349558)

[3. Урок 3: Создание и использование индексаторов 17](#_Toc301349559)

[3.1. Что такое индексатор? 17](#_Toc301349560)

[3.2. Создание индексатора 18](#_Toc301349561)

[3.3. Сравнение индексаторов и массивов 19](#_Toc301349562)

[3.4. Определение индексатора в интерфейсе 20](#_Toc301349563)

[3.5. Демонстрация: Создание и использование индексатора 21](#_Toc301349564)

[4. Урок 4: Статические методы и данные 21](#_Toc301349565)

[4.1. Создание и использование статических полей 22](#_Toc301349566)

[4.2. Создание и использование статических методов 24](#_Toc301349567)

[4.3. Создание статических типов и использование статических конструкторов 25](#_Toc301349568)

[4.4. Создание и использование методов расширения 26](#_Toc301349569)

[4.5. Демонстрация 28](#_Toc301349570)

[5. Урок 5: Перегрузка операций 28](#_Toc301349571)

[5.1. Что такое перегрузка операций? 29](#_Toc301349572)

[5.2. Перегрузка операций 30](#_Toc301349573)

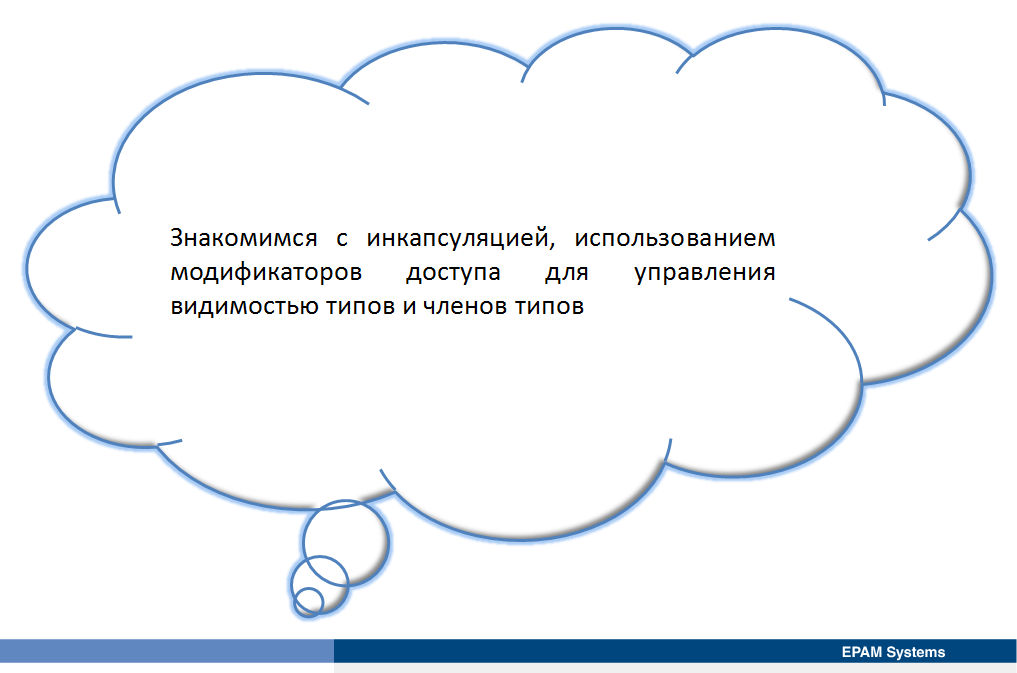
[5.3. Ограничения при перегрузке операций 32](#_Toc301349574)

[5.4. Рекомендации при перегрузке операций 33](#_Toc301349575)

[5.5. Реализация и использование операций преобразования 35](#_Toc301349576)

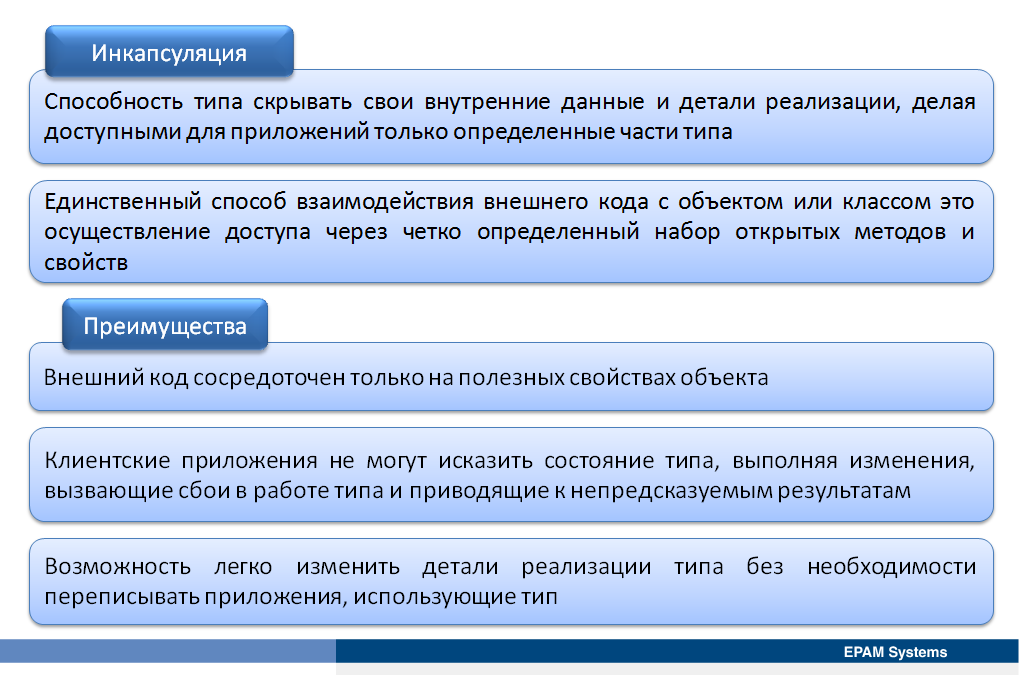
[5.6. Демонстрация: Перегрузка операций 37](#_Toc301349577)

# Урок 1: Управление видимостью членов типа



Инкапсуляция является одним из основополагающих объектно-ориентированных принципов. Она позволяет скрывать закрытые или приваные (private) данные и внутреннюю работу типа таким образом, что они становятся недоступными коду, определенному в других типах. Тип может предоставить внешнему коду открытые (public) члены, определяющие его поведение, однако он должен постоянно следить за осуществлением принципа инкапсуляции для своих private членов. В уроке объясняется, как использовать модификаторы доступа для управления видимостью типов и членов типов.

## Что такое инкапсуляция?



Все приложения управляют данными. Многие приложения старых версий, построенные с использованием языков программирования, не являюющихся объектно-ориентированными, таких как C и COBOL, как правило, отделяют код, обрабатывающий данные, от кода, который хранит эти данные и управляет ими. Код управления данными часто предлагается в виде библиотек кода, а обработка данных элементов приложения при этом встроена в бизнес-логику приложения. Некоторые приложения, реализующие различные части бизнес-системы, могут использовать все те же данные, к которым они получают доступ через библиотеку кода, но выполняют свои собственные операции над этими данными. В этом случае, если формат данных меняется, может также нуждаться в изменении и библиотека кода, управляющая данными. Тогда все приложения, которые используют оригинальную библиотеку кода, возможно, должны быть обновлены для использования новой версии, а логику этих приложений, также необходимо изменить для обработки новой структуры данных. Такие изменения могут быть трудно выполнимыми, а любые приложения, которые не обновятся правильно, могут генерировать ошибки.

Инкапсуляцией является способность типа скрывать свои внутренние данные и детали реализации, делая доступными для приложений только определенные части типа. Например, при определении класса всегда следует определять поля как private члены для того, чтобы внешний код не мог напрямую получить доступ к полям. Тогда единственным способом взаимодействия внешнего кода с объектом или классом будет осуществление доступа через четко определенный набор открытых методов и свойств. Свойства обеспечивают структурированный способ выявления способности private полей получать и устанавливать значения.

Инкапсуляция позволяет скрыть информацию. Когда скрывается такая информация, как внутреннее состояние и детали реализации типа, внешний код сосредоточен только на полезных свойствах объекта. Например, внутренний механизм телефона скрыт от пользователя; внутренние части телефона инкапсулированы под его оболочкой и недоступны для пользователей. Можно не иметь никакого представления о том, как телефон работает или как он устроен внутри, и при этом успешно его использовать. Кроме того, когда внутреннее состояние типа скрыто, клиентские приложения не могут изменять или искажать это состояние изменениями, вызвающими сбои в работе типа и приводящими к непредсказуемым результатам. Используя инкапсуляцию, можно легко изменить детали реализации типа без необходимости переписывания приложений, которые используют этот тип. Пока общедоступные методы и свойства, предоставляемые классом, не меняются, любые существующие клиентские приложения должны работать правильно.

<http://go.microsoft.com/fwlink/?LinkId=192935>

## Сравнение private и public членов типа



C# предоставляет ключевые слова, известные как модификаторы доступа, позволяющие задать уровень доступа для типов и их членов. Эти модификаторы доступа используются для реализации принципа инкапсуляции, чтобы скрыть данные и методы типа, используемого в приложениях.

C# предоставляет несколько различных модификаторов доступа, обеспечивающие различные степени защиты. Некоторые из этих модификаторов применяются к типам, другие применяются к членам типа. В этом разделе основное внимание уделяется тому, как использовать модификаторы доступа private и public, чтобы скрыть и открыть члены типа.

Если не указать модификатор доступа для члена типа, уровень доступа для него по умолчанию будет private. Такие члены могут быть доступны только в коде типа и не видимы для других типов.

Определение класса Sales в следующем примере содержит private поле monthlyProfit, private метод SetMonthlyProfit и private метод GetAnnualProfitForecast. Чтобы использовать модификатор доступа, следует добавить при объявлении префикс к члену, который будет его использовать.

class Sales

{

private double monthlyProfit;

private void SetMonthlyProfit(double monthlyProfit)

{

this.monthlyProfit = monthlyProfit;

}

private double GetAnnualProfitForecast()

{

return (this.monthlyProfit \* 12);

}

}

Все члены класса объявлены как private, поэтому они доступны только для других членов в классе Sales. Методы SetMonthlyProfit и GetAnnualProfitForecast имеют доступ к полю monthlyProfit, поскольку поле объявлено на уровне класса и, следовательно, находится в его области видимости. Если попытаться использовать член, к которому нет доступа, возникнет ошибка компиляции.

С помощью модификатора доступа private можно защитить реализацию и состояние типа от всеобщего использования типами и, следовательно, инкапсулировать данные. Однако, типы не очень полезны, если они не доступны для использования другим членам и типам. Например, определение класса Sales в предыдущем примере непригодно, потому что не существует точки входа в класс, а , следлвательно, никакой другой класс не сможет получить доступ к его данным или сослаться на какие-либо его функциональные возможности. Чтобы предоставить члены класса другим типам, можно использовать модификатор доступа public.

В отличие от модификатора доступа private модификатор доступа public является наибольшим разрешительным уровенем доступа, не накладывающим никаких ограничений. Если член типа объявляется как public, любой другой тип может получить доступ к этому члену.

Используя пример класса Sales, можно объявить методы SetMonthlyProfit и GetAnnualProfitForecast как public, тогда другие типы смогут использовать эти методы, например, как класс Program теперь может вызвать методы SetMonthlyProfit и GetAnnualProfitForecast.

class Sales

{

private double monthlyProfit;

public void SetMonthlyProfit(double monthlyProfit)

{

this.monthlyProfit = monthlyProfit;

}

public double GetAnnualProfitForecast()

{

return (this.monthlyProfit \* 12);

}

}

class Program

{

static void Main()

{

Sales companySales = new Sales();

companySales.SetMonthlyProfit(3400);

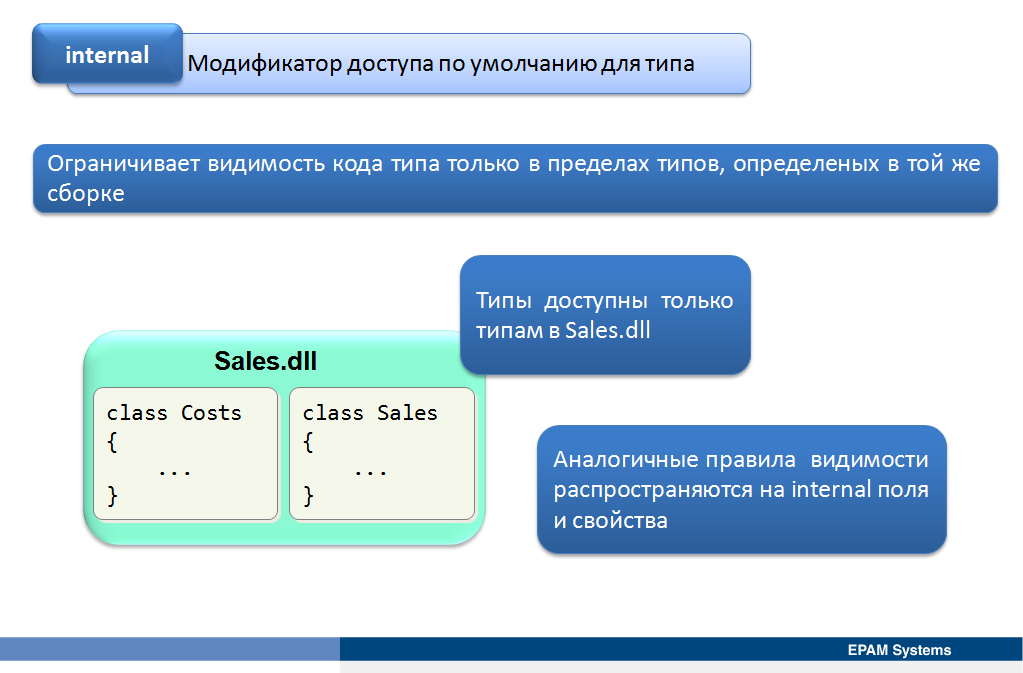
Console.WriteLine(companySales.GetAnnualProfitForecast());

}

}

При именовании private и public членов класса важно принять соглашение об именовании. Часто для именования private полей используется нотация «camel case», а и для public методов - «рascal case». При этом, все методы используют именование в ситиле «рascal case» независмо от того, являются ли они public или private. Конкретная организация может иметь свои собственные соглашения об именовании, отличные от указанных.

## Сравнение internal и public типов



Модификаторы доступа используются не только для того, чтобы открыть и скрыть члены типа, но также и для того, чтобы открыть и скрыть сами типы. В этом разделе основное внимание уделяется модификаторам доступа для определения классов, хотя те же принципы могут быть применены к любому типу, будь то класс, структура или перечисление.

Если для типа не указать модификатор доступа, по умолчанию применяется уровень доступа internal. Модификатор доступа internal ограничивает видимость кода типа только в пределах типов, определеных в той же сборке. Модификатор доступа public делает тип доступным для кода во всех использующих его типах.

Определение класса Sales в следующем примере явно не определяет модификатор доступа, поэтому ему неявно назначен модификатор доступа internal. У internal-класса public-элементы за пределами сборки не видны.

class Sales

{

private double monthlyProfit;

public void SetMonthlyProfit(double monthlyProfit)

{

this.monthlyProfit = monthlyProfit;

}

public double GetAnnualProfitForecast()

{

return (this.monthlyProfit \* 12);

}

}

Можно также использовать члены типа internal. Методы internal, например, доступны для других типов, являющихся частью той же сборки, но недоступные для типов, определенных в других сборках. Те же правила распространяются на internal поля и свойства.

.NET Framework организует типы в сборки. Сборка может представлять либо код приложения, либо содержит библиотеку типов и данных, которые могут использовать приложения. Библиотека классов .NET Framework содержит несколько сборок с большим количеством многократно используемых типов. Например, сборка System.IO .NET Framework предлагает необходимую для взаимодействия с файловой системой функциональность.

От сборок не будет большой пользы, если они будут предлагать только internal типы, поскольку в этом случае никакая другая сборка не сможет получить доступ к их функциональности. Поэтому, когда разработчики разрабытывают многократно используемый приложениями программный интерфейс (API), он обычно определяется publiс типом.

В следующем примере при определении классу Sales был явно назначен модификатор доступа publiс. Другие типы в других сборках теперь могут получить доступ к классу Sales.

public class Sales

{

private double monthlyProfit;

public void SetMonthlyProfit(double monthlyProfit)

{

this.monthlyProfit = monthlyProfit;

}

public double GetAnnualProfitForecast()

{

return (this.monthlyProfit \* 12);

}

}

Типы можно также объявить как private. Тип можно определить как private, только если он вложен в другой тип. Этот модификатор доступа наиболее часто используется для определения private перечислителей, хотя можно также определить классы и структуры, которые необходимо использовать в коде, не открывая другим типам доступ к ним.

В следующем примере кода показан publiс класс Sales, в котором содержится private структура Revenue. Структура Revenue инкапсулирована в классе Sales и непосредственно извне не доступна.

public class Sales

{

private Revenue salesRevenue;

public void SetRevenue(string currency, double amount)

{

this.salesRevenue = new Revenue(currency, amount);

}

private struct Revenue

{

string currency;

double amount;

public Revenue(string currency, double amount)

{

this.currency = currency;

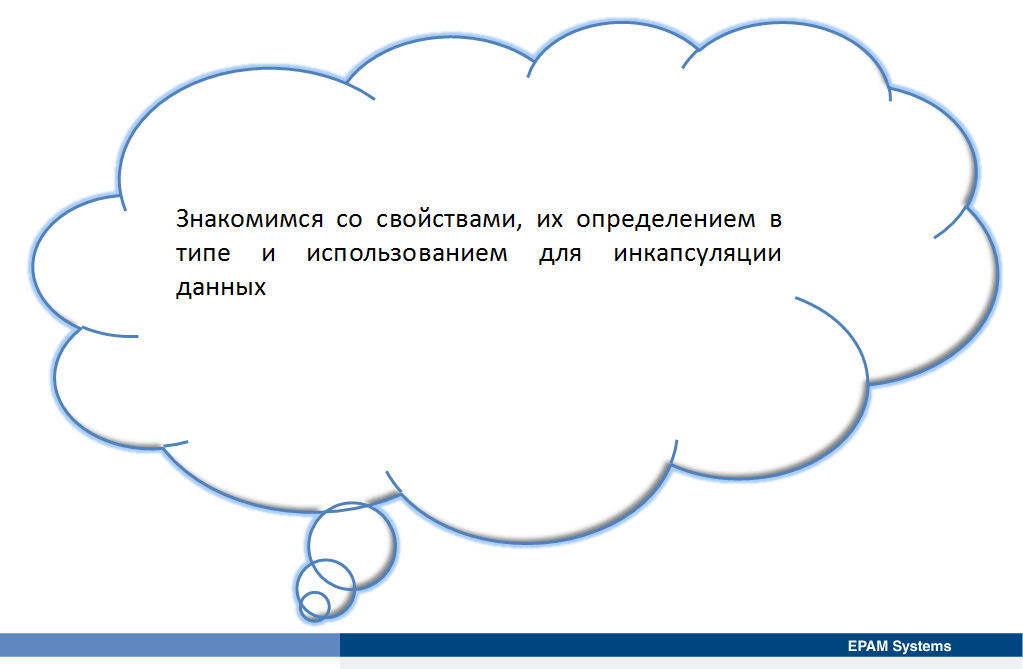
this.amount = amount;

}

}

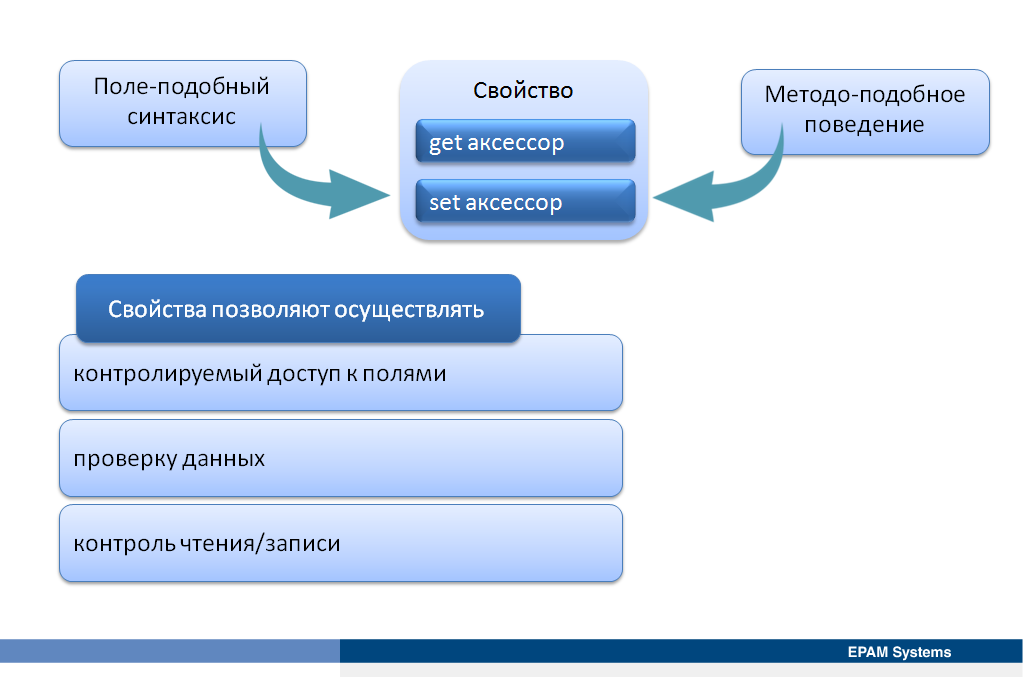
}

# Урок 2: Создание и использование свойств



Для обеспечения контролируемого доступа к данным в типе можно использовать свойства. Урок знакомит со свойствами, показывает, как их определить в типе и почему следует использовать для инкапсуляции данных.

## Что такое свойство?



Свойство это нечто среднее между полем и методом, поскольку используется поле-подобный синтаксис для доступа к свойству. Однако, поведение свойства больше похоже на метод. Свойство может содержать два элемента:

* get аксессор, который приложение использует для чтения значения свойства.
* set аксессор, который приложение использует для изменения значения свойства.

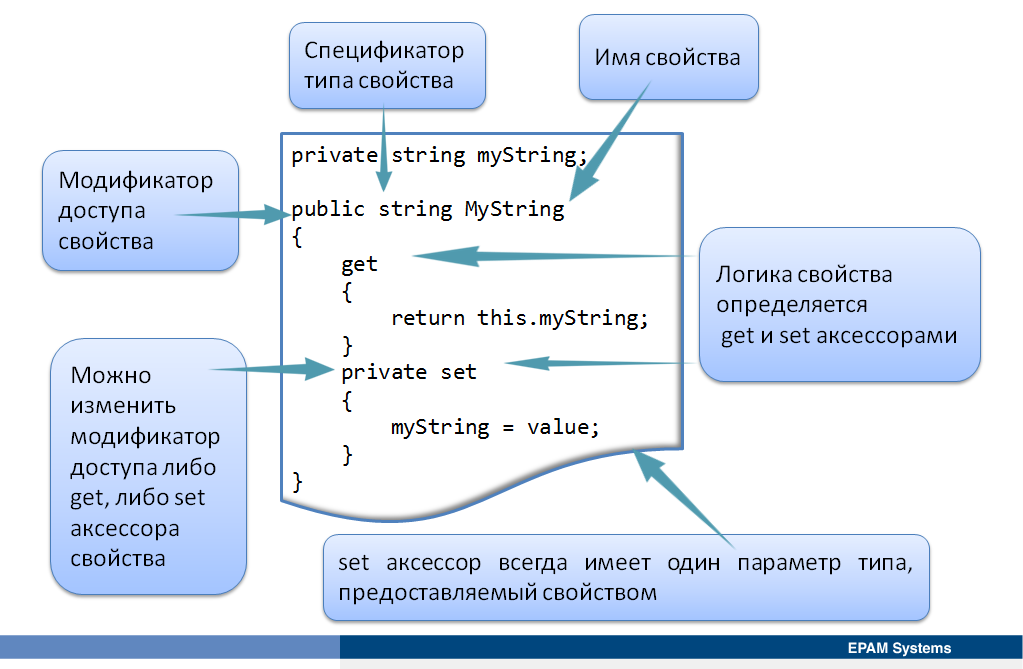
Свойства являются распространенным способом инкапсуляции данных, доступных в классе. Обычно свойства отображается на private поля типа. Поле хранит данные, а get и set аксессоры свойства обеспечивают механизм для доступа к этому поле. Свойства имеют преимущество, позволяющее осуществлять контроль над полями, предоставляя возможность сделать их только для чтения или только для записи, или для чтения и записи, что нельзя осуществить, сделав поле public.

Еще одним преимуществом использования свойств является возможность проверки данных. Если открывается поле типа, любой другой тип может читать или писать в это поле. При условии, что данные являются правильным типом, любое значение, может быть присвоено этому полю. Это не всегда логично, иногда может понадобиться ограничить диапазон допустимых значений для поля в типе. Со свойством, можно добавить логику set аксессору для проверки того, чтобы значение попадало в ожидаемый диапазон перед обновлением private поля.

Хотя свойства обычно отображаются на private поля, не существует никаких требований для того, чтобы это было именно так. Get аксессор свойства может возвращать вычисленное значение, постоянное значение или выполнять любые другие операции, применимые к приложению. Свойства часто включают дополнительную логику, например, если с помощью свойства изменяется имя файла, свойство может проверить, существует ли файл, используемый в настоящее время, и, при необходимости, переименовать файл или открыть новый файл в соответствии с требованиями приложения.

<http://go.microsoft.com/fwlink/?LinkId=192948>

## Определение свойства



Свойство имеет тип и имя так же, как и поле. Однако, логика свойства определяется get и set аксессорами. Get аксессор, как метод, может включать любой код, однако он должен возвращать объект типа, указанный свойством или генерировать исключение. Set аксессору не требуется выполнять никаких функций, хотя, как правило, для обновления private поля при выполнении некоторых операций значение передается в свойство. Для set аксессора не указывается параметр; set аксессор всегда имеет один параметр типа, предоставляемый свойством. Получить доступ к объекту, переданному в качестве параметра для set аксессора можно используя ключевое слово value.

В следующем примере показано определение простого свойство, обеспечивающего доступ к private полю. Ключевое слово get определяет блок кода, который запускается, когда приложение считывает свойство. Ключевое слово set определяет блок кода для логики, выполняемой, когда приложение присваивает значение свойству.

private string myString;

public string MyString

{

get

{

return this.myString;

}

set

{

this.myString = value;

}

}

Чтобы определить свойство только для чтения, просто опускается set аксессор. Аналогичным образом, чтобы определить свойство только для записи, не реализовывается get аксессор.

При определении свойства для него необходимо указать модификатор доступа. Модификатор доступа, указанный для свойства наследуется get и set аксессорами. Можно изменить модификатор доступа либо get, либо set аксессора, однако, нельзя сделать их более доступными, чем определено уровнем доступа свойства. Например, нельзя сделать get аксессор public, если свойство является private. В следующем примере показано, как изменить уровень доступа public get аксессора на уровне доступа private.

public string MyString

{

get

{

return this.myString;

}

private set

{

myString = value;

}

}

Использовать свойство в классе можно с помощью точечной нотации так же, как при обращении к public полю. В следующем примере показано получение доступа к свойству MyString из предыдущего примера кода. Внутренне, Visual C# компилятор преобразует все попытки прочитать свойство в вызовы get аксессоров и изменяет все попытки написать свойство в вызовы set аксессоров.

MyObject theClass = new MyObject;

// Setting the string – calls the set accessor

theClass.MyString = "Property set.";

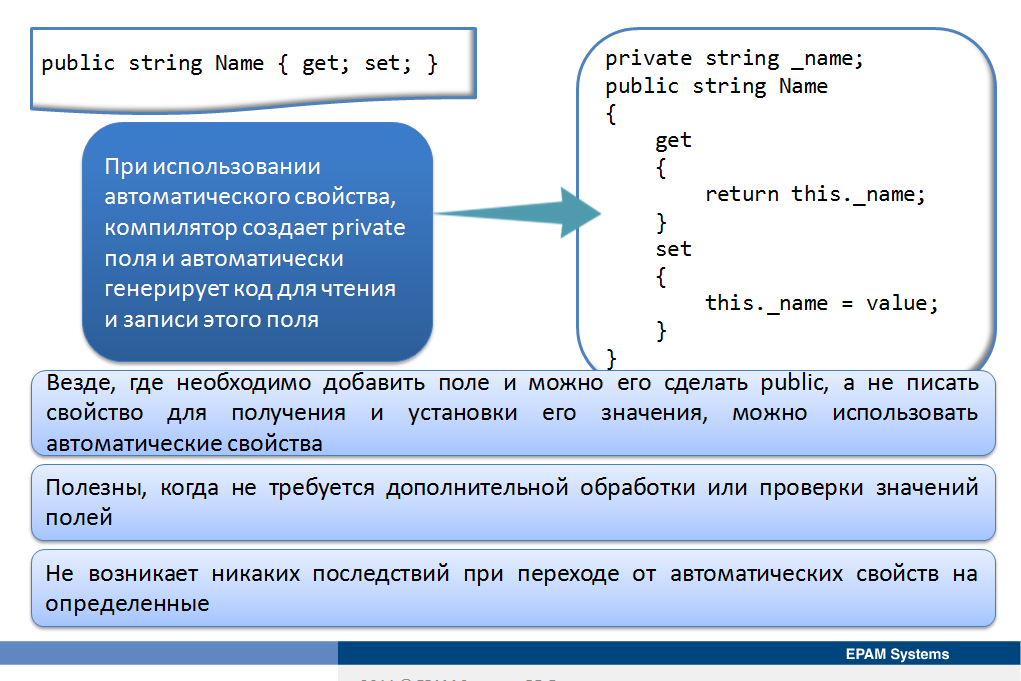
// Getting the string – calls the get accessor

Console.WriteLine(theClass.MyString);

Можно определить статические свойства, но они могут получить доступ только к статическим данным.

<http://go.microsoft.com/fwlink/?LinkId=192949>

## Автоматические свойства



При разработке нового типа, можно включать в тип поля данных, открытые для приложений. Если не требуется дополнительной обработки или проверки значений этих полей, может быть заманчиво просто открыть поля вместо добавления свойств для предоставления доступа к ним. В этом случае открытое поле может не выглядеть как проблема. Однако, следует помнить, что в поле нельзя добавить код для контроля недопустимых значений, а в свойство можно. Если при начальной разработке приложения не нужно добавлять проверку или некоторую логику в свойство, это не означает, что так будет всегда. Требования типа могут изменяться с течением жизни приложения.

На первый взгляд использование свойства аналогично использованию поля, однако для компилятора это не так. Компилятор преобразует код при обращении к свойству в вызов метода get аксессора, и аналогичным образом преобразует запись в свойство в вызов метода set аксессора (Рис. ). И если возникнет необходимо преобразовать поле в свойство на более позднем сроке, это может имеет последствия для существующих приложений. Любое приложение, которое использовало тип со значением представленным как поле, должно быть перекомпилировано с данными, предоставляемыми через свойство. Если этот тип является сборкой, используемый рядом приложений, может потребоваться много установок для восстановления и повторного развертывания.



Рис.

Этой дополнительной работы можно избежать просто представляя данные через свойство при начальной разработке типа. Тогда любые последующие изменения типа могут быть сделаны без необходимости перекомпиляции приложений его использующих.

Везде, где необходимо добавить поле и можно его сделать public, а не писать свойство для получения и установки его значения, можно использовать автоматические свойства.

Автоматические свойства обеспечивают простой встроенный синтаксис, преобразующий поле в свойство. Чтобы использовать автоматические свойства, добавляются фигурные скобки, содержащие как set, так и get аксессор, за каждым из которых следует точкой с запятой.

public string Name { get; set; }

При использовании автоматического свойства, компилятор создает private поля и автоматически генерирует код для чтения и записи этого поля.

private string \_name;

public string Name

{

get

{

return this.\_name;

}

set

{

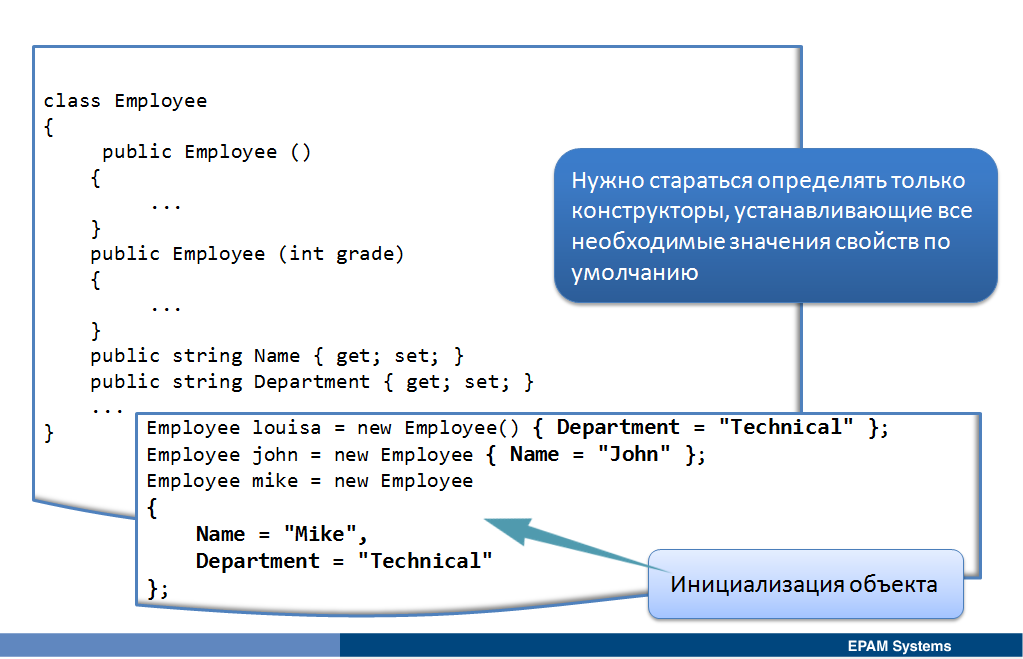
this.\_name = value;

}

}

Автоматические свойства всегда определяют как set, так и get аксессоры. Автоматические свойства предназначены для использования там, где в противном случае тип просто предоставил public поля. Если требуется более конкретный контроль над данными, следует написать свойства вручную. Для использующих классов не возникает никаких последствий при переходе от автоматических свойств на определенные вручную при более позднем создании кода, они полностью взаимозаменяемы, в отличие от свойств и полей.

## Создание экземпляра объекта с помощью свойства



Ранее было продемонстрировано использование конструктора для создания экземпляра объекта и инициализации его полей. При этом, можно объявить несколько конструкторов, с разными сигнатурами, чтобы предоставить возможность устанавливать различные комбинации полей в типе в соответствующие значения, однако, этот подход является проблематичным, если у типа существует большое количество полей или свойств одного и того же типа. В следующем примере показан простой класс с несколькими конструкторами.

class Employee

{

private string name;

private string department;

// Initialize both fields

public Employee(string empName, string empDepartment)

{

this.name = empName;

this.department = empDepartment;

}

// Initialize name only

public Employee(string empName)

{

this.name = empName;

}

// Initialize department only

public Employee(string empDepartment)

{

this.department = empDepartment

}

...

}

Намерения данных конструкторов состоит в указании значения имени сотрудника, названия отдела или и того, и другого при создании нового объекта Employee. Однако данный код не скомпилируется, поскольку компилятор не может различить два конструктора, принимающих один параметр string. При попытке создать экземпляр объекта Employee, используя следующий код, компилятор не знает, какой конструктор используется.

// Is "Fred" the name of an employee or a department?

Employee myEmployee = new Employee("Fred");

Решить эту проблему можно путем использования свойства при инициализации объекта при создании его экземпляра. Этот синтаксис известен как инициализатор объекта. С инициализатором объекта, новый объект создается с помощью конструктора, который завершается с помощью связанных пар имя\_свойства/значение, разделенных запятыми и заключенный в фигурные скобки. В следующем примере показано определение класса, поддерживающего инициализаторы объектов и создание объекта, используя инициализаторы.

class Employee

{

// Default constructor.

public Employee ()

{

...

}

// Constructor that sets the grade of an employee.

public Employee (int grade)

{

...

}

// Expose Name and Department as automatic properties.

public string Name { get; set; }

public string Department { get; set; }

...

}

// Instantiating an object and setting a single property.

Employee louisa = new Employee() { Department = "Technical" };

// Instantiating an object and setting a single property.

// You do not have to add the brackets to use the default constructor.

Employee john = new Employee{ Name = "John" };

// Instantiating an object and setting a multiple properties.

// Separate properties with a comma.

Employee mike = new Employee

{

Name = "Mike",

Department = "Technical"

};

В первом примере для создания объекта Employee louisa используется конструктор по умолчанию. После того как объект будет создан и конструктор завершен, значение «Technical» присвоится свойству Department. Если используется конструктор по умолчанию, можно опустить скобки, как иллюстрируют второй (john) и третий (mike) примеры.

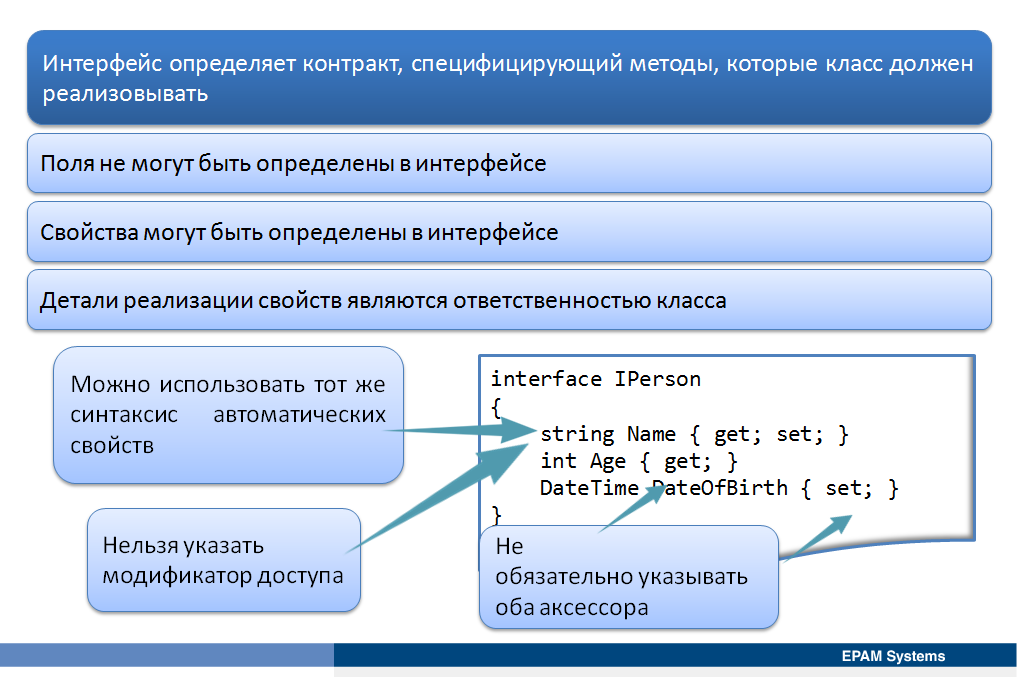
Если класс Employee имеет конструктор не по умолчанию, можно вызывать его вместе с инициализатором объекта, как показано в следующем примере. Здесь используется конструктор, устанавливающий ранг сотрудника.

Employee antony = new Employee(2){ Name = "Antony", Department = "Management" };

При использовании инициализатора объекта, вначале работает логика конструктора, а затем задаются значения свойств, указанные в инициализаторе объекта. Это означает, что если установить свойство в конструкторе, а затем установить его в инициализаторе объекта, значение инициализатора объекта будет перезаписывать значение, установленное конструктором.

Нужно стараться определять только конструкторы, устанавливающие все необходимые значения свойств по умолчанию. Тогда классы, использующие тип, смогут переопределить эти свойства в инициализаторе объекта.

## Определение свойств в интерфейсе



Интерфейс определяет контракт, специфицирующий методы, которые класс должен реализовывать. Интерфейс может также определять свойства. Однако детали реализации этих свойств (например, поля на которые они ссылаются, если таковые имеются) являются ответственностью класса. Чтобы добавить свойства в интерфейс, можно использовать тот же синтаксис автоматических свойств, за исключением того, что нельзя указать модификатор доступа. Следующий пример показывает свойства добавленные в интерфейс.

interface IPerson

{

string Name { get; set; }

int Age { get; }

DateTime DateOfBirth { set; }

}

Классы, реализующие интерфейс, который включает в себя свойства, могут реализовать свойства явно или неявно. В следующих примерах кода показаны свойства интерфейса IPerson, реализованные в неявном и явном виде соответственно.

class Person : IPerson

{

public string Name

{

get

{

throw new **NotImplementedException**();

}

set

{

throw new **NotImplementedException**();

}

}

public int Age

{

get { throw new **NotImplementedException**(); }

}

public DateTime DateOfBirth

{

set { throw new **NotImplementedException**(); }

}

}

class Person : IPerson

{

string IPerson.Name

{

get

{

throw new **NotImplementedException**();

}

set

{

throw new **NotImplementedException**();

}

}

int IPerson.Age

{

get { throw new **NotImplementedException**(); }

}

DateTime IPerson.DateOfBirth

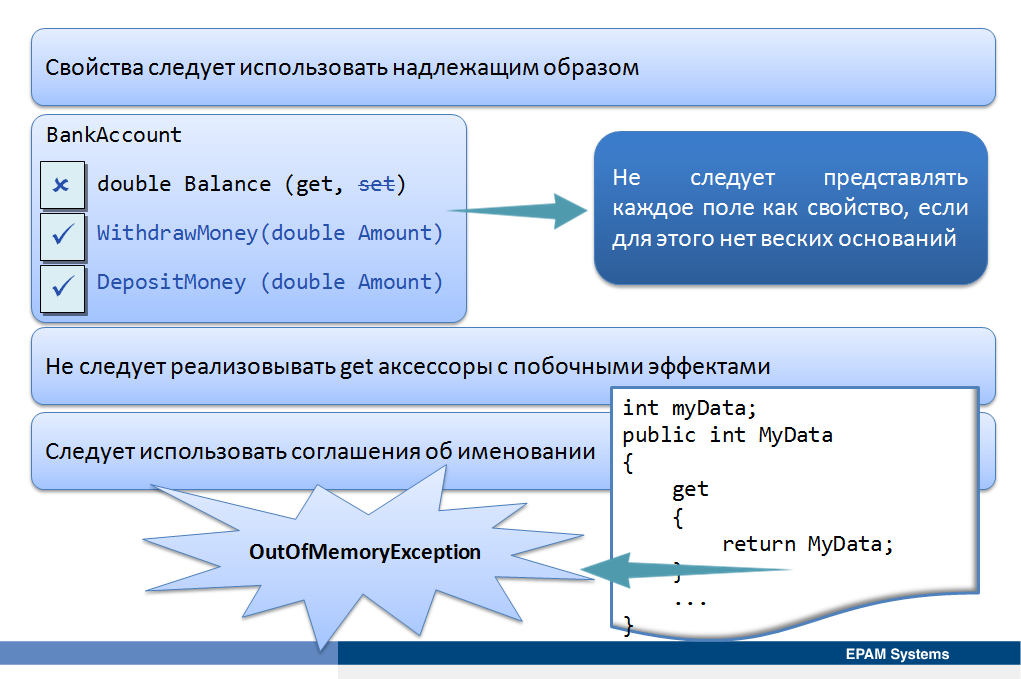
{

set { throw new **NotImplementedException**(); }

}

}

## Рекомендации по определению и использованию свойств



Свойства обеспечивают прекрасную основу для предоставления данных разрабатываемых типов, однако, если не использовать свойства надлежащим образом или просто представляя свойства, позволяющие использующих их классам выполнять нежелательное поведение, существует риск внедрения ошибок. Следуя некоторым рекомендациям можно смягчить риски.

**Свойства следует использовать надлежащим образом** Следует тщательно рассмотреть вопрос о том, является ли предоставление свойства подходящим для типов операций, которые приложение может выполнять над элементом данных.

Например, если разрабатывается тип, представляющий банковский счет, поле в классе может представлять баланс счета. Может быть заманчиво обеспечить свойство, которое позволяет приложению читать и записывать баланс счета, но это не будет отражать реальных операций, обычно реализуемых банком, для этого более целесообразно предусмотреть методы Deposit и Withdraw.

При разработке типов в приложении, следует помнить, что при разработке типов представляются функциональные возможности, необходимые для конкретного приложения. Не следует представлять каждое поле как свойство, если для этого нет веских оснований.

**Не следует реализовывать get аксессоры с побочными эффектами.** Get аксессоры должны просто получать значение и возвращать это значение использующему приложению. При реализации get аксессора, извлечение значения не должно влиять на значение или любые другие данные, хранящиеся в типе. Единственным исключением из этого правила является правило, когда пишутся приложения, которые должны придерживаться ограничений по безопасности. В этом случае, можно добавить логику get аксессора для доступа или еще большего ограничения доступа в соответствии с бизнес-требованиями.

**Следует использовать соглашения об именовании.** Согласно соглашению обертка поля использует имя, которое отличается от имени поля только начальной буквой. Например, поле, называемое myData, как правило, заключено в свойство MyData. Однако, в этом случае очень легко случайно написать код, который вызывает себя рекурсивно.

int myData;

public int MyData

{

get

{

return MyData;

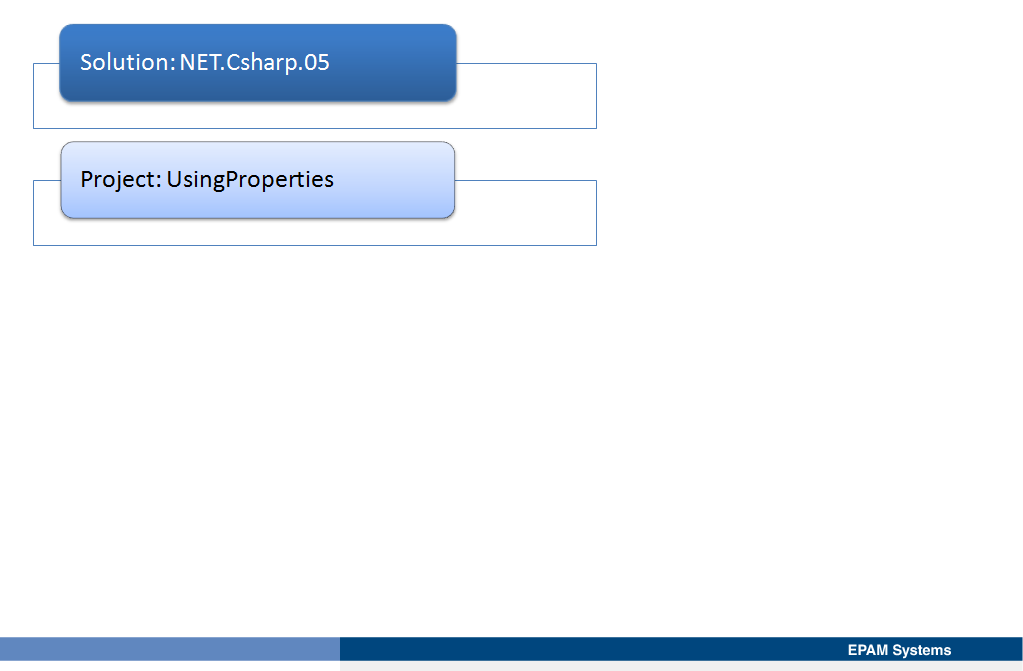
}

...

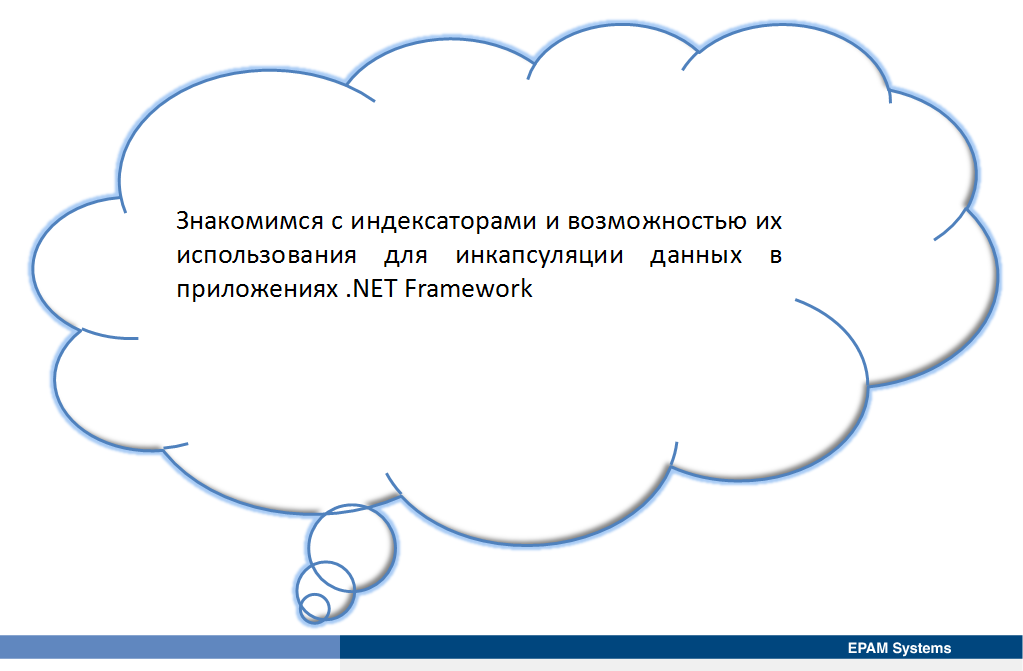
}

Код будет компилироваться, однако, в этом нем есть ошибка, которая приведет к бесконечному циклу, поскольку свойство **MyData** вызывает себя рекурсивно. Ошибки, такие как эта, может быть трудно обнаружить. Если позволить приложению с такой ошибкой выполняться достаточно долго, то в конечном итоге сгенерируется исключение **OutOfMemoryException**.

## Демонстрация: Использование свойств

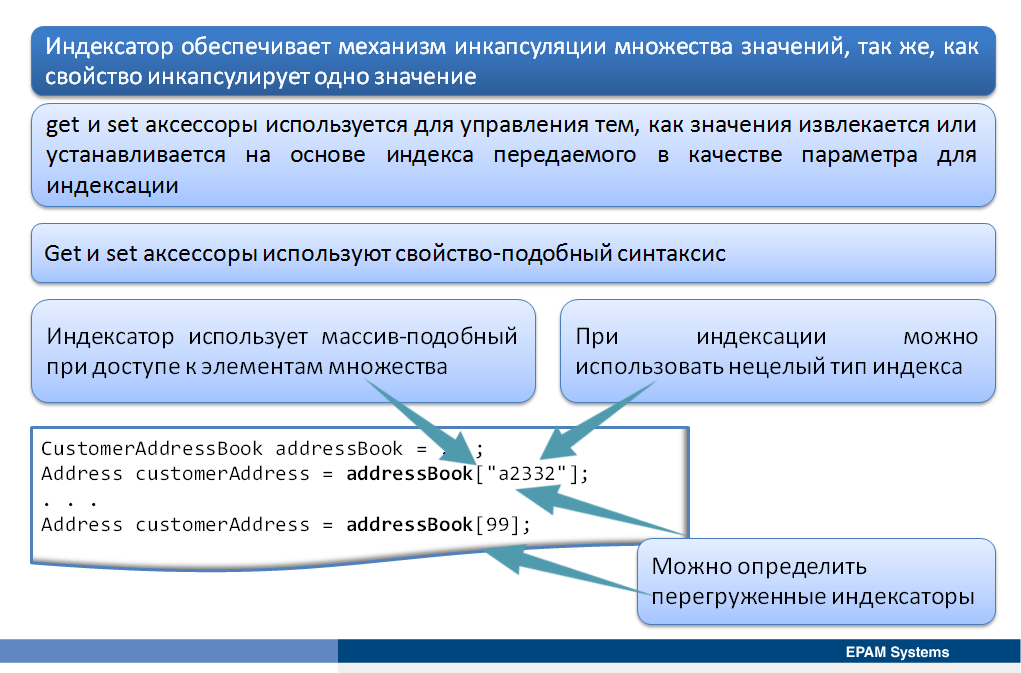


# Урок 3: Создание и использование индексаторов



Свойство предоставляет доступ к одному элементу в типе. Однако некоторые типы по своей природе многозначны, например, массивы или коллекции. Элемент может содержать вложенные элементы, к которым нужно обеспечить легкий доступ. Например, думая о строке как о наборе символов, может понадобиться обеспечить доступ к отдельным символам в строке через свойство. Наиболее естественно для доступа к элементам множества использовать массиво-подобную запись. Предоставить такой доступ возможно, определяя свойство-индексатор. Урок знакомит с индексаторами и описывает, как их можно использовать для инкапсуляции данных в приложениях.

## Что такое индексатор?



Индексатор обеспечивает механизм инкапсуляции множества значений, так же, как свойство инкапсулирует одно значение. Индексатор используется для доступа к одному значению во множестве значений, а get и set аксессоры используется для управления тем, как значения извлекается или устанавливается на основе индекса передаемого в качестве параметра для индексации. Get и set аксессоры используют свойство-подобный синтаксис.

Индексатор доступа использует тот же синтаксис, что и при доступе к элементам массива. Однако индексаторы предоставляют больше гибкости. Например, при индексации можно использовать нецелый тип индекса вместо целого, обычно используемого для доступа к элементам массива.

В следующем примере показано использование простого индексатора для типа CustomerAddressBook. Индексатор позволяет приложению получать адрес клиента, указав идентификатор клиента, который хранится в строке.

CustomerAddressBook addressBook = ...;

// Use an indexer to find the address of a customer.

Address customerAddress = addressBook["a2332"];

В типе можно определить перегруженные индексаторы, принимающие различные типы параметров. Например, тип CustomerAddressBook может обеспечить индексатор, извлекающий адреса клиента на основе целого числа.

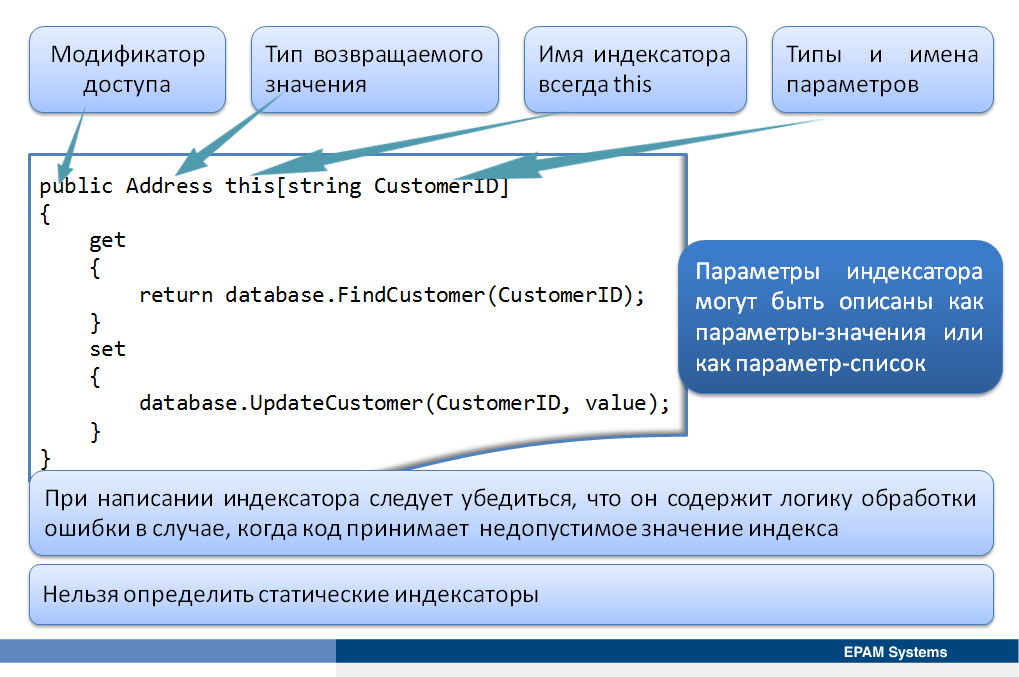
// Find the address of the customer with the specified reference.

Address customerAddress = addressBook[99];

В дополнение к определению индексаторов, принимающих различные параметры, индексаторы также могут возвращать различные типы; они не должны возвращать экземпляр типа, который определяет индексатор.

<http://go.microsoft.com/fwlink/?LinkId=192952>

## Создание индексатора



При написании индексатора используется синтаксис, напоминающий свойство: определяется модификатор доступа, тип и get и set аксессоры, но имя индексатора всегда this. При этом типы и имена параметров указываются используя массиво-подобное обозначение в квадратных скобках. Как свойство, индексатор также может быть только для чтения (имеет только get аксессор) или только для записи (имеет только set аксессор).

Получить доступ к параметрам индексатора можно по имени в аксессорах, а также в set аксессоре, можно использовать ключевое слово value для доступа к значению, передаемому индексатором.

Параметры индексатора могут быть описаны как параметры-значения или как параметр-список (с использованием ключевого слова params). Допустимо также использование опциональных параметров. Параметры, передаваемые индексатором, предназначены только для того, чтобы найти элемент данных для установки или получения. В get аксессоре возвращается найденный в этом месте элемент,а в set аксессоре хранятся данные, указанные значением параметра в этом месте.

В следующем примере кода показан простой индексатор, который позволяет приложению найти адрес клиента данного идентификатора или обновить адрес. Адрес хранится в базе данных, доступной через переменную базы данных.

class AddressBook

{

public Address this[string CustomerID]

{

get

{

return database.FindCustomer(CustomerID);

}

set

{

database.UpdateCustomer(CustomerID, value);

}

}

...

}

При написании индексатора следует убедиться, что он содержит логику обработки ошибки в случае, когда код принимает недопустимое значение индекса.

Нельзя определить статические индексаторы.

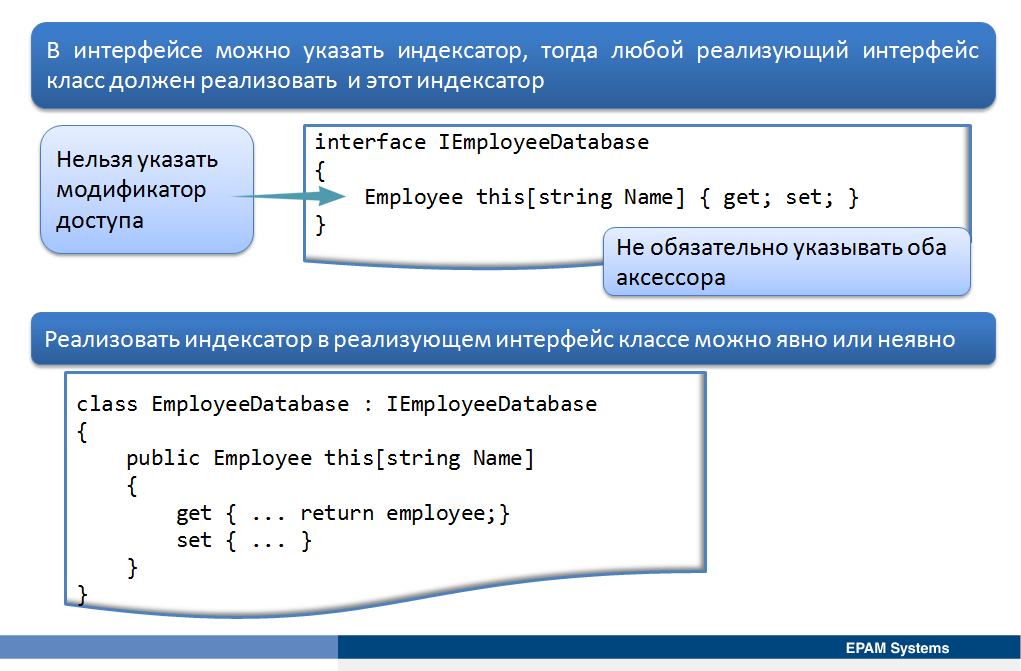
## Сравнение индексаторов и массивов



При использовании индексатора используется массиво подобный синтаксис, однако между индексатерами и массивами существует несколько важных различий.

* При использовании массива доступ к элементам этого массива осуществляется с помощью числового индекса. Например, для получения доступа к пятому элементу в массиве myArray используется следующий синтаксис myArray[4] (при условии индексации с нуля). С массивами можно использовать только числовые индексы. Индексатор предоставляет большую гибкость, поскольку предоставляется возможность использовать нечисловые индексы.
* Нельзя перегружать массивы; реализация определяется средой выполнения, и все классы, которые наследуются от какого-то класса не могут изменить поведение этого массива. Однако, существует полный контроль над поведением индексатора, и наследуемые классы могут переопределить индексаторы, обеспечив их собственную реализацию.
* При вызове метода, принимающего ref или out параметры, ему нужно передать указатель на ячейку памяти в методе. Элементы массива могут быть отображены непосредственно на ячейки памяти, поэтому они могут быть использованы в качестве параметра в методе, который принимает ref или out параметры. Индексаторы не отображаются непосредственно на ячейки памяти, поэтому нельзя использовать индексатор в качестве параметра ref или out, хотя можно передавать их в качестве значения параметров.

## Определение индексатора в интерфейсе



В интерфейсе можно указать индексатор, тогда любой реализующий интерфейс класс должен реализовать и этот индексатор. Чтобы указать индексатор в интерфейсе, следует добавить индексатор без модификатора доступа, указав get, set или оба аксессора и заменить тело аксессоров точкой с запятой.

interface IEmployeeDatabase

{

Employee this[string Name] { get; set; }

}

Реализовать индексатор в реализующем интерфейс классе можно явно или неявно. В следующем примере показан класс неявно реализующий интерфейс с индексатором.

class EmployeeDatabase : IEmployeeDatabase

{

public Employee this[string Name]

{

get

{

...

return employee;

}

set

{

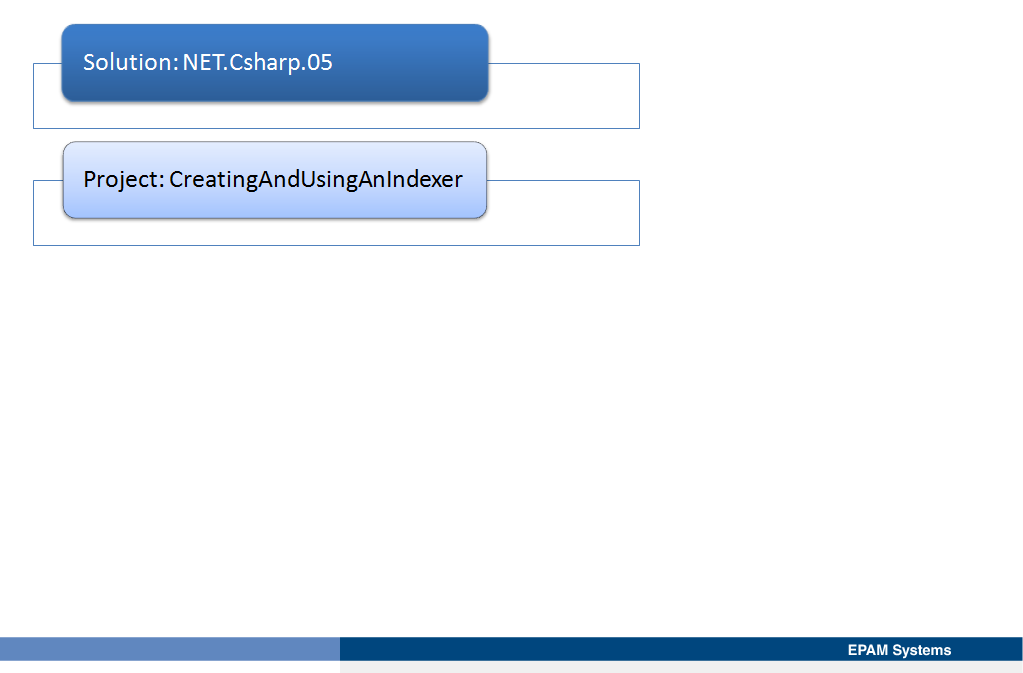
...

}

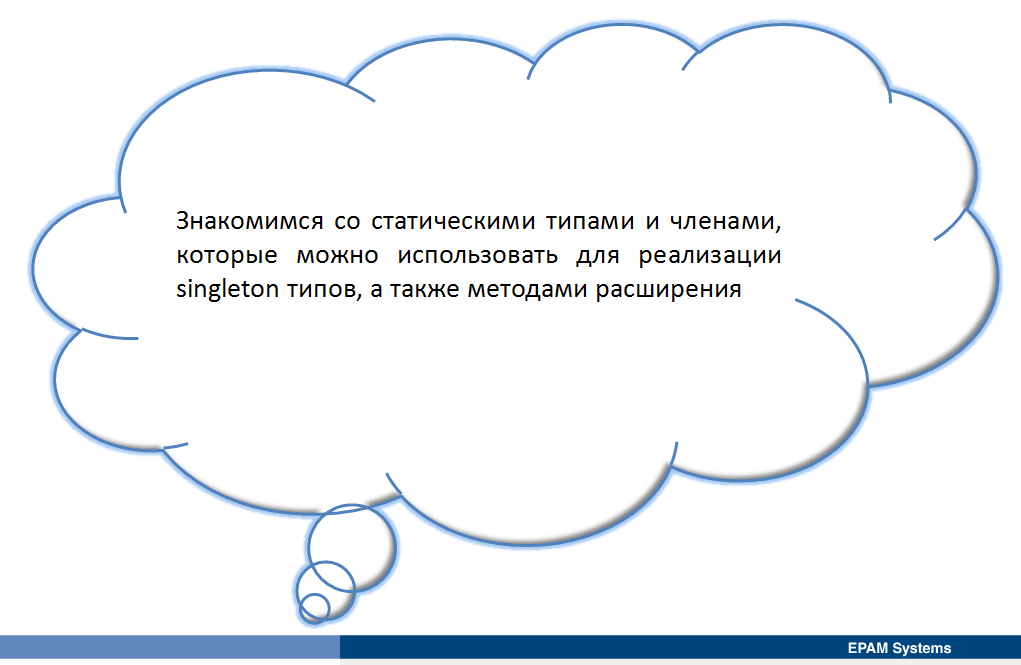
}

}

## Демонстрация: Создание и использование индексатора

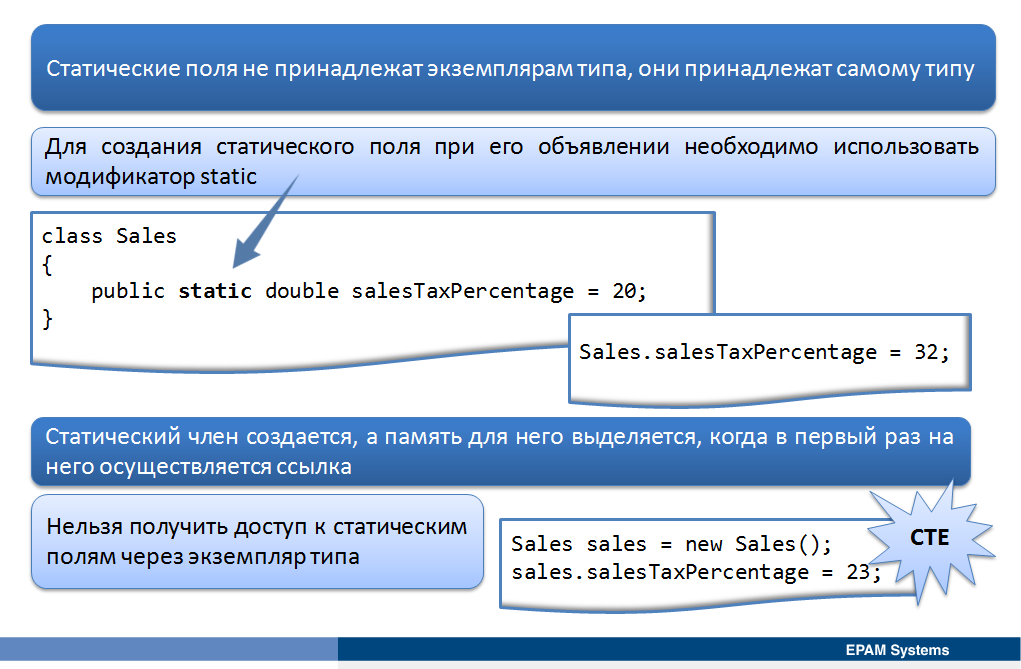


# Урок 4: Статические методы и данные



Урок вводит понятие статических типов и членов, которые можно использовать для реализации singleton типов, а также объясняет, как расширить существующие типы с помощью методов расширения (extension methods).

## Создание и использование статических полей



До этого момента, основной акцент был сделан на создании и использовании членов экземплярных типов. Члены экземплярных типов обычно содержат данные и осуществляют функции, относящиеся к конкретному экземпляру типа. При создании экземпляра типа память выделяется для каждого поля, что позволяет экземпляру сохранять уникальные данные. В следующем примере объект sales2010 является экземпляром класса Sales, и любые данные, которые хранятся в объекте sales2010, доступны только через экземпляр sales2010. При создании второго экземпляра класса Sales sales2011, он не сможет получить доступ к любому из данных экземпляра объекта sales2010, а объект sales2010 не может получить доступ к данным экземпляра sales2011.

Sales sales2010 = new Sales();

sales2010.SetMonthlyProfit(34672);

Console.WriteLine(sales2010.GetAnnualProfitForecast());

Sales sales2011 = new Sales();

sales2011.SetMonthlyProfit(98675);

Console.WriteLine(sales2011.GetAnnualProfitForecast());

...

class Sales

{

private double monthlyProfit;

public void SetMonthlyProfit(double monthlyProfit)

{

this.monthlyProfit = monthlyProfit;

}

public double GetAnnualProfitForecast()

{

return (this.monthlyProfit \* 12);

}

}

В противоположность экземплярным полям, статические поля не принадлежат экземплярам типа, они принадлежат самому типу. Память выделяется независимо от любого экземпляра, и все ссылки на статическое поле ссылаются на один и тот же участок памяти. Для доступа к статическим членам типа не требуется создавать экземпляр этого типа. Статический член создается, а память для него выделяется, когда в первый раз на него осуществляется ссылка.

Для создания статического поля при его объявлении необходимо использовать модификатор static. В следующем примере кода показано, как объявить статическое поле salesTaxPercentage и присвоить ему значение 20.

class Sales

{

public static double salesTaxPercentage = 20;

}

Статические поля можно инициализировать при объявлении, и, если поле является public, можно получить доступ к полю из другого типа[[1]](#footnote-1). При этом для доступа к полю salesTaxPercentage не нужно создавать экземпляр типа Sales, поскольку поле является статическим. Вместо этого, доступ к полю можно получить непосредственно указав имя типа Sales, за которым следует точка и имя поля.

Sales.salesTaxPercentage = 32;

Нельзя получить доступ к статическим полям через экземпляр типа, так как статические поля принадлежат самому типу. В противном случае возникнет ошибка компиляции, объясняющая, что член не может быть доступен ссылке на экземпляр. Следующий код будет генерировать ошибку компиляции.

Sales sales = new Sales();

sales.salesTaxPercentage = 23; // Compile error.

...

class Sales

{

public static double salesTaxPercentage;

}

Однако можно получить доступ к статическим полям с помощью методов экземпляра и конструкторов, позволяющим совместно использовать данные нескольким экземплярам одного и того же типа. Например, класс User в следующем примере представляет пользователя, обращающегося к веб-сайту. Каждый раз, когда новый пользователь обращается к сайту, создается новый объект User, а также увеличивается значение статического поля usersOnline. Другие экземпляры класса могут получать доступ к данным и увеличивать их, потому что поле является статическим, в итоге поле usersOnline будет представлять общее число пользователей веб-сайта в данный момент времени.

class User

{

internal static int usersOnline;

internal User()

{

usersOnline++;

}

}

User a = new User();

User b = new User();

User c = new User();

User d = new User();

int totalUsersOnline = User.usersOnline; // Returns the value 4.

Если поле usersOnline не будет статическим, данные не будут разделяться экземплярами типа, и поле usersOnline сможет достигнуть только значения 1, как показано в следующем примере.

class User

{

internal int usersOnline;

internal User()

{

usersOnline++;

}

}

User a = new User();

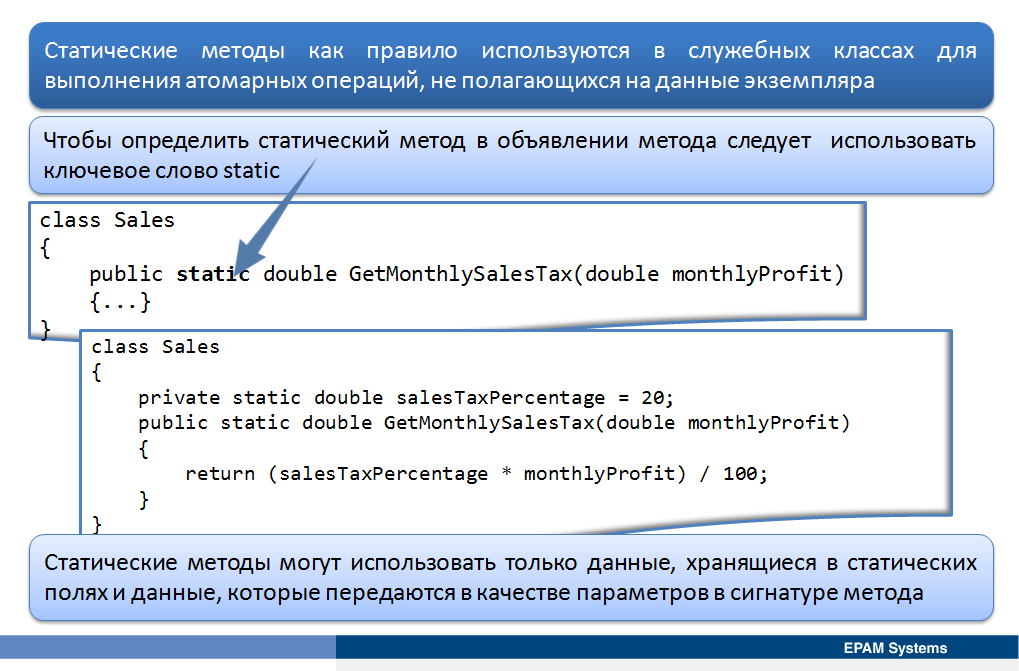
User b = new User();

User c = new User();

User d = new User();

int totalUsersOnline = d.usersOnline; // Returns the value 1.

## Создание и использование статических методов



Модификатор static также можно использовать для создания статических методов. Статические методы очень полезны и, как правило, используются в служебных классах для выполнения атомарных операций, не полагающихся на данные экземпляра. Например, класс File пространства имен System.IO содержит несколько статических методов для выполнения атомарных операций с файлами.

Чтобы определить статический метод, в объявлении метода следует использовать ключевое слово static[[2]](#footnote-2).

class Sales

{

public static double GetMonthlySalesTax(double monthlyProfit)

{

...

}

}

Статический метод не может ссылаться на членов экземпляров, объявленных в этом классе. Статические методы могут ссылаться только на другие статические члены, в отличие от методов экземпляра, которые могут получать доступ как к экземплярным, так и статическим членам. Как правило, методы экземпляра используют данные, хранящиеся в полях экземпляра, которые были собраны во время инициализации объекта и вызовов других методов. Статические методы могут использовать только данные, хранящиеся в статических полях и данные, которые передаются в качестве параметров в сигнатуре метода.

Например, класс Sales в следующем примере предоставляет метод GetMonthlySalesTax, используетющий данные, которые передаются параметром monthlyProfit, и public static поле salesTaxPercentage.

class Sales

{

private static double salesTaxPercentage = 20;

public static double GetMonthlySalesTax(double monthlyProfit)

{

return (salesTaxPercentage \* monthlyProfit) / 100;

}

}

Чтобы получить доступ к статическим методам класса, можно использовать имя класса, за которым следуют точка и имя метода. Например, в следующем примере показан доступ к методу GetMonthlySalesTax класса Sales.

double monthlySalesTax = Sales.GetMonthlySalesTax(34267);

...

class Sales

{

public static double salesTaxPercentage = 20;

public static double GetMonthlySalesTax(double monthlyProfit)

{

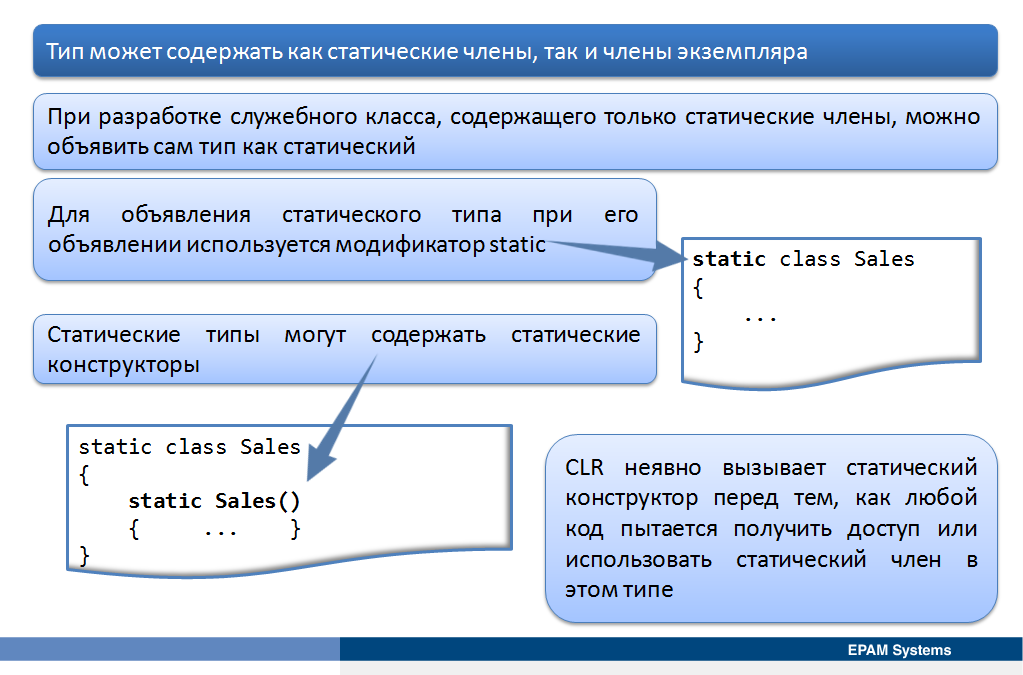
return (salesTaxPercentage \* monthlyProfit) / 100;

}

}

<http://go.microsoft.com/fwlink/?LinkId=192933>

## Создание статических типов и использование статических конструкторов



Тип может содержать как статические члены, так и члены экземпляра, которые порой могут быть очень полезны. Однако, если разрабатывается служебный класс, содержащий только статические члены, можно объявить сам тип как статический. Для объявления статического типа при его объявлении используется модификатор static. В следующем примере показано создание статического класса.

static class Sales

{

...

}

Если класс объявлен как статический, этот класс не может содержать члены экземпляров, такие как поля, методы и конструкторы. Поскольку класс не может содержать конструктор экземпляра, нельзя создавать объекты этого типа. Однако, статические типы могут содержать статические конструкторы, которые ведут себя иначе, чем конструкторы экземпляра.

При использовании статических типов, нельзя явно вызвать конструктор с помощью ключевого слова new. Как правило, общеязыковая среда выполнения (CLR) неявно вызывает статический конструктор перед тем, как любой код пытается получить доступ или использовать статический член в этом типе. Статический конструктор можно определить в статическом типе следующим образом.

static class Sales

{

static Sales()

{

...

}

}

Статический класс можно использовать как обычный контейнер для набора методов, работающих на входных параметрах, и не возвращающих или устанавливающих какие-либо внутренние поля экземпляра.

При использовании статических конструкторов типов необходимо следовать некоторым правилам, чтобы избежать ошибок компиляции.

* Можно определить только один конструктор, имеющий префикс-модификатор static.
* Нельзя явно вызывать статический конструктор, поэтому конструктор не может быть доступнен вне типа, а, следовательно, он всегда использует неявный модификатор доступа private.
* При определении сигнатуры для статического конструктора, нельзя указывать параметры.
* Статический конструктор может содержать только ссылки на другие статические члены.

Статические конструкторы имеют множество применений, например, с их помощью можно определить объекты, реализующие шаблон проектирования singleton. Шаблон проектирования singleton гласит, что код должен быть ограничен созданием одного экземпляра типа. Потом этот экземпляр можно использовать всякий раз, когда в приложении потребуется экземпляр этого типа.

static class Sales

{

static SaleData data = null;

static Sales()

{

if (SaleData.WebServerConnectionExists())

{

data = SaleData.GetWebServerData();

}

else if (SaleData.LocalDatabaseConnectionExists())

{

data = SaleData.GetDatabaseData();

}

else

{

throw new NotSupportedException(

"No data source could be found.");

}

}

public static string[] GetAllSalesRegions()

{

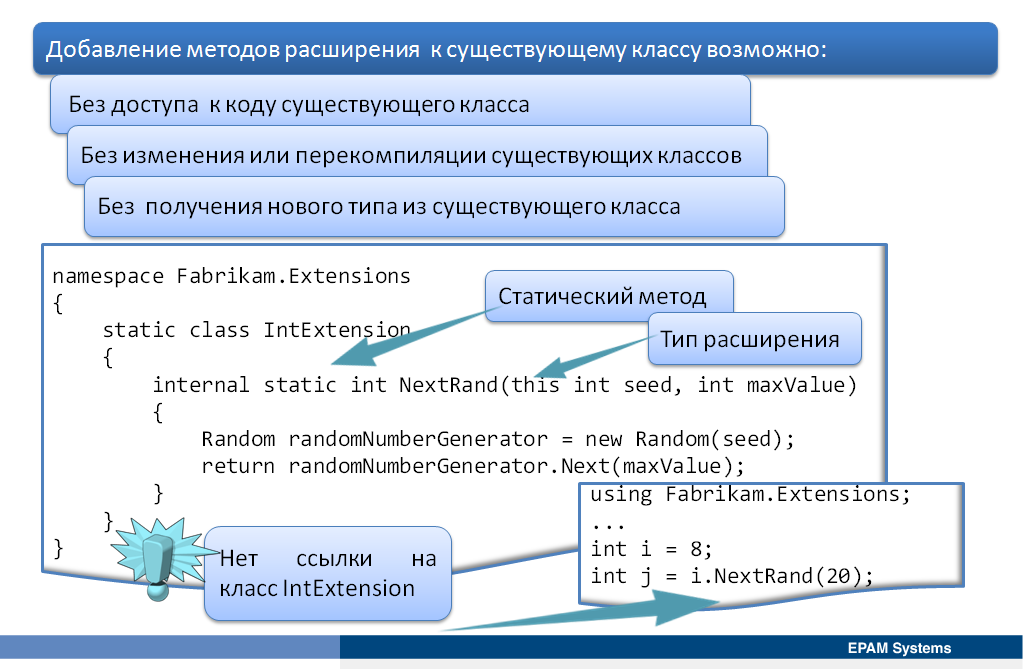
throw new NotImplementedException();

}

}

Класс Sales содержит конструктор, гарантирующий, что объект data инициализируется перед вызовом метода GetAllSalesRegions. Конструктор достигает этого некоторой условной логикой, которая либо инициализирует данные объекта, либо выбрасывает исключение. Клиентское приложение, использующее класс Sales, может быть уверено, что объект data был инициализирован до вызова метода GetAllSalesRegions.

## Создание и использование методов расширения



Методы расширения предоставляют возможность расширения новой функциональностью существующих типов таким образом, что это не влияет и не нарушает работу других приложений, уже использующих даннве типы. До введения методов расширения, необходимо было определить новый класс, чтобы обернуть существующую функциональность и обеспечить новый требуемый метод, что, однако, могло повлиять на существующий код и внести критические изменения.

При определении метода расширения, первый параметр в сигнатуре метода должен обязательно указывать на расширяемый тип и сопровождается префиксом из ключевого слова this, указывающий на метод расширения. Во время выполнения параметр заменяется экземпляром типа, и, в итоге, можно манипулировать этим экземпляром тем же способом, который требуется для реализации логики метода расширения.

Общим требованием является определение методов расширения в статических классах и создание таких классов в конкретных пространствах имен. При этом, методы расширения всегда должны быть определены как статические. В следующем примере показано определение метода расширения NextRand для типа int. Метод возвращает случайное число, использующее существующий объект int как источник для генерации. Метод также принимает параметр, представляющий максимальное значение нового случайного числа.

namespace Fabrikam.Extensions

{

static class IntExtension

{

internal static int NextRand(this int seed, int maxValue)

{

Random randomNumberGenerator = new Random(seed);

return randomNumberGenerator.Next(maxValue);

}

}

}

Клиентское приложение использует метод расширения таким же образом, как использует и стандартный метод, нет видимой разницы. Метод можно по-прежнему вызвать для экземпляра объекта. В следующем примере показан вызов метода NextRand пространства имен Fabrikam.Extensions. Следует обратить внимание, что код клиента не ссылается на класс IntExtension, в котором объявлен метод NextRand

using Fabrikam.Extensions;

namespace Fabrikam.Client

{

class Program

{

static void Main()

{

int i = 8;

int j = i.NextRand(20);

}

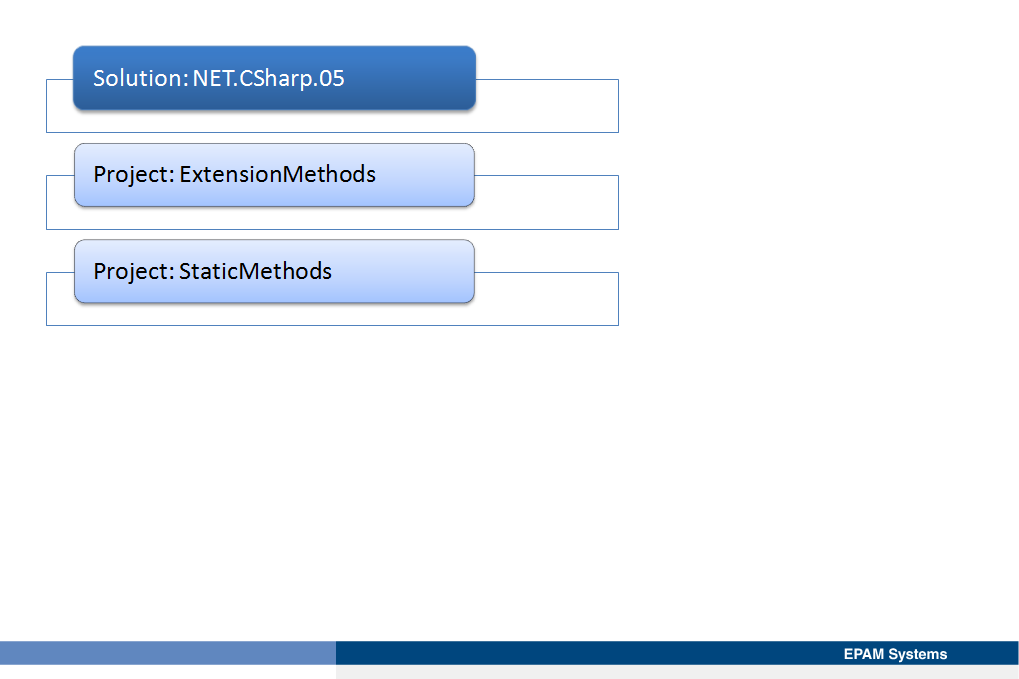
}

}

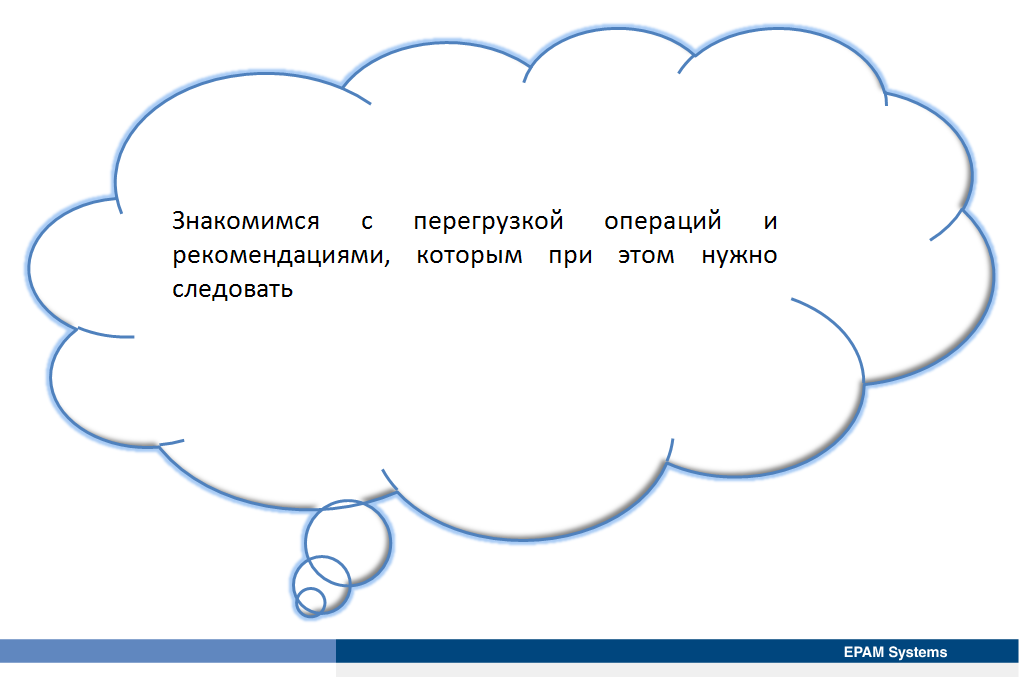
Методы расширения применимы к типу, как только импортируется пространство имен, содержащее класс с этими методами расширения. Если выполняется импорт нескольких пространств имен, содержащих классы с одинаковыми методами расширения для одного типа, то возникает ошибка компиляции.

Наиболее распространенными методами расширения являются операторы запросов LINQ, которые добавляют стандартную функциональность запросов к существующим типам System.Collections.Inumerable и System.Collections.Generic.IEnumerable<T>. Чтобы использовать стандартные операторы запросов, нужно импортировать пространство имен System.Linq. Затем каждый тип, реализущий интерфейс IEnumerable<T>, будет иметь методы экземпляра, такие как GroupBy, OrderBy, Average и т. д.

## Демонстрация

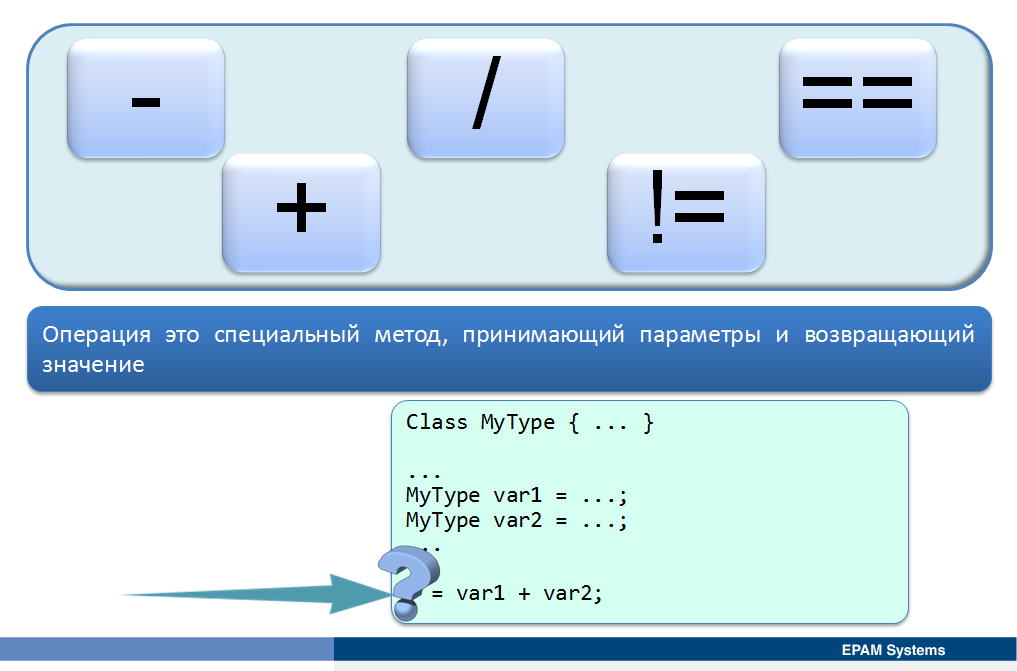


# Урок 5: Перегрузка операций



Многие из встроенных типов, определяемых Visual C#, предоставляют возможность выполнения над ними некоторых общих операций. Например, Visual C# определяет операции «+», «-» , «\*» и «/», которые выполняют вполне определенное действие над числовыми данными. Однако, операция «+» может работать и со строковым типом, и тогда ее поведение совершенно иное, она объединяет строки вместе. Это пример перегруженной операции. Для пользовательских типов также можно реализовать перегруженные операции. Урок показывает, как определять и осуществлять перегрузку операций, а также описывает некоторые рекомендации, которым нужно при этом следовать.

## Что такое перегрузка операций?



Visual C# включает в себя несколько операций, позволяющих выполнять стандартные операции над объектами. Эти операции можно использовать для построения выражений. Точное поведение каждой из операций зависит от типа объекта, над которым она выполняется.

Операция это специальный метод, который принимает множество параметров и возвращает значение. При вызове операции операнды передаются в этот метод в качестве параметров, а значение, возвращенное методом, является результат операции. При перегрузке операций предоставляется собственная реализация этого метода.

Visual C# определяет три категории операций, которые могут вызвать перегрузку:

* Унарные операции. Эти операции включают «!», «++», «--», «+» и «-». При перегрузке этих операций, необходимо указать один параметр, который должен быть того же типа, что и класс, который определяет оператор.
* Бинарные операции. Эти операторы включают «\*», «/», «+», «-» и «%». При перегрузке этих операций необходимо указать два параметра, по крайней мере один из которых должен быть того же типа, что и класс, определяющий операцию.
* Операции преобразования. Эти операции можно использовать для преобразования данных одного типа в другой. При перегрузке этих операций, необходимо указать один параметр, содержащий данные, которые нужно конвертировать. Эти данные могут быть любого допустимого типа.

Нельзя перегрузить все операции, определенные в Visual C#. Ниже приводится краткая таблица, в которой приведены операции, которые можно и нельзя перегружать.

|  |  |
| --- | --- |
| **Операции** | **Возможность перегрузки** |
| **+**, **-**, **!**, **~**, **++**, **--**, **true**, **false** | Унарные операции, которые могут быть перегружены. |
| **+**, **-**, **\***, **/**, **%**, **&**, **|**, **^**, **<<**, **>>** | Бинарные операции, которые могут быть перегружены. |
| **==**, **!=**, **<**, **>**, **<=**, **>=** | Операции сравнения, которые могут быть перегружены. |
| **&&**, **||** | Условные логические операции не могут быть перегружены, но они оцениваются с помощью **&** и **|**, которые могут быть перегружены. |
| **[ ]** | Операция индексирования массива не может быть перегружена, но можно определить индексаторы, которые могут бть перегружены. |
| **()** | Операция приведения не может быть перегружена, но можно определить новые операции преобразования. |
| **+=**, **-=**, **\*=**, **/=**, **%=**, **&=**, **|=**, **^=**, **<<=**, **>>=** | Операции присваивания не могут быть перегружены, но +=, например, оценивается с помощью операции «+», которая может быть перегружена. |
| **=**, **.**, **?:**, **->**, **new**, **is**, **sizeof**, **typeof** | Эти операции не могут быть перегружены. |

## Перегрузка операций



Для определения в разрабатываемом типе собственного поведения операции, выбранную операцию следует перегрузить. Для этого используется методо-подобный синтаксис с типом возвращаемого значения и параметрами, но именем метода при этом является ключевое слово operator вместе с символом перегружаемой операции. Например, для перегрузки операции «+» можно определить метод, называемый operator+. В следующем примере показана определяемая пользователем структура Hour, в которой бинарная операция operator+ перегружена для сложения двух экземпляров Hour.

struct Hour

{

public Hour(int initialValue)

{

this.value = initialValue;

}

public static Hour operator +(Hour lhs, Hour rhs)

{

return new Hour(lhs.value + rhs.value);

}

...

private int value;

}

Анализируя метод operator +, следует обратить внимание на следующие моменты:

* Рекомендуется модификатор доступа public.
* Все методы должны быть static. Перегруженные операции не могут быть полиморфными и использовать модификаторы virtual, abstract, override или sealed.

При объявлении очень стилизованных функциональностей (таких как, операции), полезно принять соглашения для именования параметров. Например, разработчики часто используют lhs и rhs (сокращения для левой и правой операндов, соответственно) для бинарных операторов.

При использовании операции «+» в бинарном выражении типа Hour Visual C# компилятор автоматически преобразует код в вызов метода operator +.

Hour Example(Hour a, Hour b)

{

return a + b;

}

Visual C# компилятор преобразует предыдущий код в код, напоминающий следующий (это псевдокод, а не реальный синтаксис Visual C#).

Hour Example(Hour a, Hour b)

{

return Hour.operator +(a,b); // pseudocode

}

Существует правило, которому нужно следовать при объявлении операции: по крайней мере один из параметров должен быть всегда содержащего операцию типа. В классе Hour в методе operator+ один из параметров a или b должен быть объектом Hour. В этом примере, оба параметра объекты Hour.

Операции следуют обычным правилам перегрузки, и можно перегрузить операцию в классе столько раз, сколько необходимо до тех пор, пока компилятор Visual C# сможет различить каждую перегрузку. Например, можно определить дополнительную реализацию метода operator+ для того, чтобы добавить целое число (количество часов) к объеку Hour – первый параметр может быть объектом Hour, а второй параметр может быть целочисленным объектом.

Hour Example(Hour a, int b)

{

return a + b;

}

Любой класс или структура могут перегрузить операции true и false. Операции перегружаются парой, тип возвращаемого значения операций – булевский. Если выполнена подобная перегрузка, объекты могут использоваться как условия в операторах условного перехода и циклов.

Рассмотрим следующий пример. Пусть в классе Point перегружены операции true и false:

public class Point

{

public double X { get; set; }

public double Y { get; set; }

public override string ToString()

{

return String.Format("X = {0} Y = {1}", X, Y);

}

public static Point operator +(Point a, Point b)

{

return new Point { X = a.X + b.X, Y = a.Y + b.Y };

}

. . .

public static bool operator true(Point a)

{

return (a.X > 0) || (a.Y > 0);

}

public static bool operator false(Point a)

{

// этот метод должен возвращать «true»,

// если семантика объекта соответствует false

return (a.X == 0) && (a.Y == 0);

}

}

Теперь возможно написать такой код:

var p1 = new Point { X = 10, Y = 20 };

var p2 = new Point();

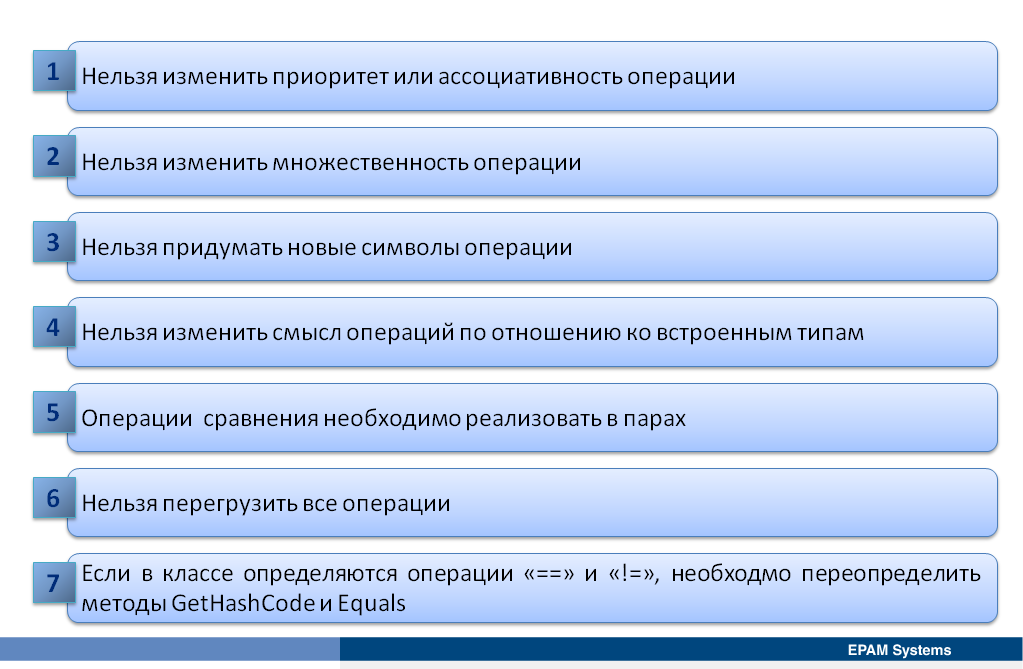
if (p2)

Console.WriteLine("Point is positive");

else

Console.WriteLine("Point has non-positive data");

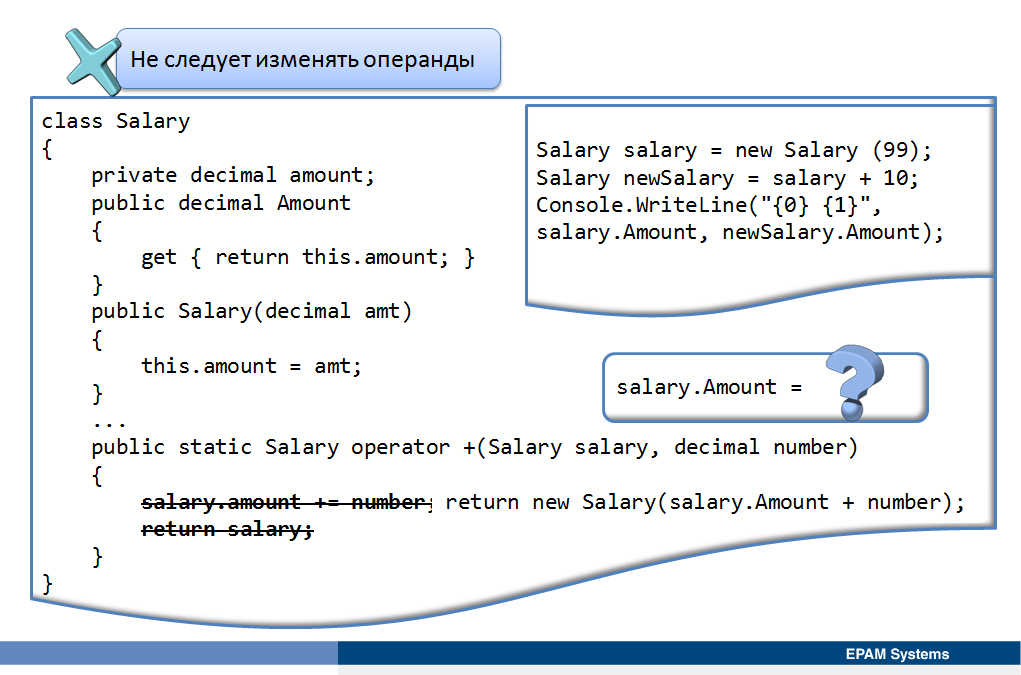
## Ограничения при перегрузке операций



При перегрузке операций можно полностью контролировать ее выполнение, однако, существует несколько правил, применяемых к операциям, которые нельзя изменить:

* Нельзя изменить приоритет или ассоциативность операции. Приоритет и ассоциативность основаны на символе операции (например, «+») и не зависят от типа (например, int), для которого используется символ операции. Следовательно, выражение a + b \* c всегда то же, что и выражение a + (b \* c), независимо от типов a, b и c.
* Нельзя изменить множественность (число операндов) операции. Например, символ умножения «\*» определяет бинарную операцию. При объявлении в типе операции «\*» она должна быть бинарной операцией. Аналогично, операция «++» является унарной операцией, и если она объявляется в типе, она должна быть унарной операцией.
* Нельзя придумать новые символы операции. Например, нельзя создать новый символ операции, такой как «\*\*», для возведения в степень. Если необходимо выполнить операцию, для которой не существует оператора, необходимо создать метод.
* Нельзя изменить смысл операций по отношению ко встроенным типам. Например, выражение  1 + 2  имеет предопределенное значения, и нельзя его переопределить. Когда в типе определяется операция, по крайней мере один из операндов для этой операции должен быть содержащего типа.
* Кроме того, операторы сравнения необходимо реализовать в парах. Например, если перегружается операция «>», необходимо также перегрузить операцию «<». Если перегружается операция «==», необходимо также перегрузить операцию «!=».
* Если в классе определяются операции «==» и «!=», также необходмо переопределить методы GetHashCode и Equals, унаследованные от System.Object (или System.ValueType при создании структуры). Метод Equals должен обладать точно таким же поведение, как оператор «==». (Следует определить одно через другое.) Метод GetHashCode используется другими классами в Microsoft .NET Framework (например, при использовании объекта в качестве ключа в хэш-таблице).

## Рекомендации при перегрузке операций



При определении перегруженных операций для типа следует придерживаться следующих рекомендаций, где возможно:

* Не изменять операнды.
* Определять симметрические операции.
* Определять только имеющие смысл операции.

**Не следует изменять операнды.** Операция никогда не должна изменять значения любого из операндов. Если любой из этих операндов ссылочного типа, используемая операция может изменять значение операнда в дополнение к возвращению результата (операция приводит к побочному эффекту). В следующем примере класс Salary предоставляет в классе operator+ для добавления значения decimal к полю amount и возвращает обновленный объект Salary как результат.

class Salary

{

private decimal amount;

public decimal Amount

{

get { return this.amount; }

}

public Salary(decimal amt)

{

this.amount = amt;

}

...

public static Salary operator +(Salary salary, decimal number)

{

salary.amount += number;

return salary;

}

}

Это плохая реализация операции, поскольку operator+ изменяет значение первого операнда. Когда operator+ завершается, поле amount в первом операнде имеет то же значение, что и результат. В следующем примере кода, значение в newSalary.Amount составляет 109, и значение salary.Amount также 109, хотя хотелось бы ожидать 99.

Если Salary является структурой, а не классом, побочного эффекта не будет, потому что параметр Salary будет передан методу operator+ по значению, а не по ссылке.

Salary salary = new Salary (99);

Salary newSalary = salary + 10;

Console.WriteLine("{0} {1}", salary.Amount, newSalary.Amount);

Output

------

109 109

Вместо этого, следует вернуть новый объект, который содержит новое значение, как показано в следующем примере кода.

class Salary

{

private decimal amount;

public decimal Amount

{

get { return this.amount; }

}

public Salary(decimal amt)

{

this.amount = amt;

}

...

public static Salary operator +(Salary salary, decimal number)

{

return new Salary(salary.Amount + number);

}

}

**Определение симметричных операций.** При определении бинарной операции следует избегать навязывания порядка операндам. Если оператор коммутативный, порядок, в котором указываются операнды не должен иметь логического различия. В примере класса Salary, выражения salary + 99 и 99 + salary должны иметь один и тот же результат.

При перегрузке операции первый операнд используется в качестве первого параметра, а второй операнд используется в качестве второго параметра. Поэтому, чтобы поддерживать обе формы сложения, класс Salary должен предоставить две реализации метода operator+, как показано в следующем примере. Компилятор автоматически не добавляет симметрические операции.

public static Salary operator +(Salary salary, decimal number)

{

return new Salary (salary.Amount + number);

}

public static Salary operator +( decimal number, Salary salary)

{

// Call the first operator – avoid code duplication.

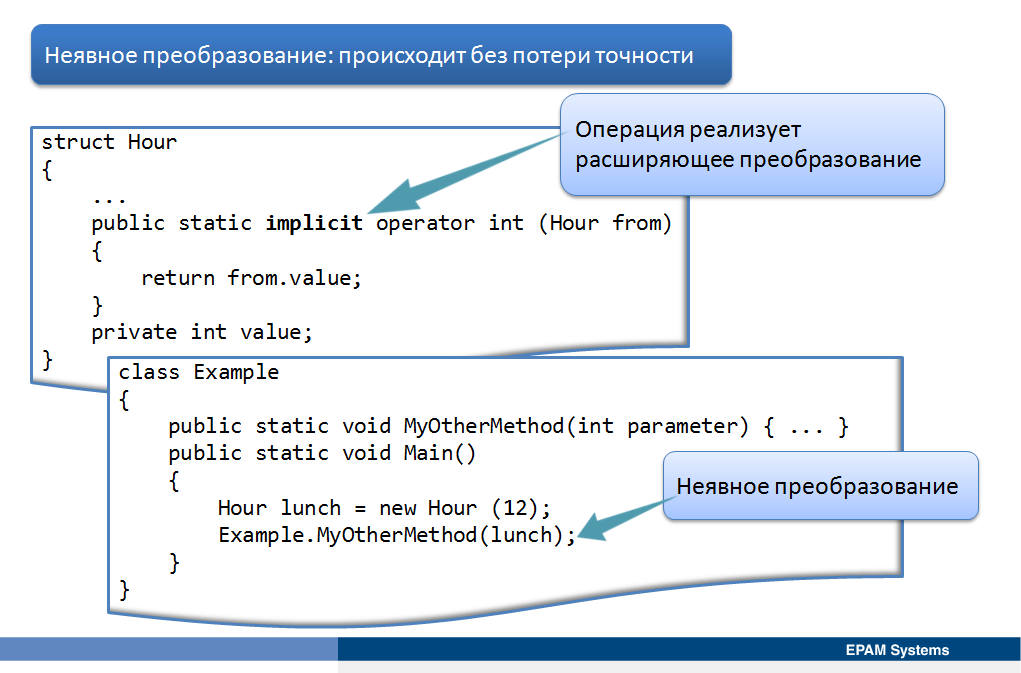
return salary + number;

}

Вторая реализация метода operator+ просто вызывает первый оператор, чтобы избежать дублирования кода. Это гарантирует, что логика определения операции приводится в одном методе и, следовательно, проще в обслуживании.

**Определение только имеющих смысл операций.** Определять операции следует только там, где они являются естественными и имеют смысл. Например, вряд ли стоит определять «+» или «-» операции в классе, моделирующем банковский счета, для добавления или удаления средств со счета; банки выполняют много дополнительных проверок при добавлении или удалении средств; это не простые операции сложения или вычитания. Вместо этого нужно предоставить методы Deposit и Withdraw, которые могут инкапсулировать эти проверки более значимым образом.

## Реализация и использование операций преобразования



Операция преобразования преобразует выражение из одного типа в другой. Преобразование может быть явным или неявным. Неявные преобразования происходят при присваиваии типа более широкому без потери точности. Это называется расширяющее преобразования (widening conversion). Оператор, реализующий расширяющее преобразование вызвается компилятором автоматически, не требуя никакого дополнительного вмешательства программиста. Например, в следующем примере, операция, которая присваивает целое выражение переменной double всегда возможна без потери данных, поскольку тип int имеет меньшие размер и точность, чем тип double.

int i = 99;

double d = i; // Safe, widening conversion from int to double

Явное преобразование происходит при переходе от типа к более специфическому типу, когда есть риск потери данных. Это называется сужающем преобразованием. Из-за потенциальной потери данных, сужающие преобразования не выполняются автоматически, а требуют, чтобы программист указал операцию приведение. Присваивание значения double значению int является примером сужающего преобразования, требующего приведения, например.

double d = 99.9;

int i = (int)d; // Data loss, narrowing conversion from double to int

Синтаксис объявления операции пользовательского преобразования аналогичен объявлению перегруженной операции. Операция преобразования должна быть public и static. Имя операции преобразования либо implicit (если она реализует расширяющее преобразование) или explicit (если он реализует сужающее преобразование).

В следующем примере кода показана операция преобразования, позволяющая неявно преобразовать объект Hour в int, что безопасно, поскольку все часы имеют эквивалентные значения целого типа.

struct Hour

{

...

public static implicit operator int (Hour from)

{

return from.value;

}

private int value;

}

При этом тип, который необходимо конвертировать (Hour), объявляется в качестве единственного параметра, а тип, в который осуществляется преобразование (int) – после ключевого слова operator.

При объявлении оператора преобразования следует указать, является ли это неявным или явным оператором преобразования с использованием ключевых слов implicit и explicit.

Вызвать неявную операцию преобразования можно без требуемого преобразования.

class Example

{

public static void MyOtherMethod(int parameter) { ... }

public static void Main()

{

Hour lunch = new Hour (12);

Example.MyOtherMethod(lunch); // implicit conversion

}

}

В следующем примере показана операция явного преобразования из объекта int в объект Hour. Здесь возвращаемый тип Hour, а параметр int. Это сужающее преобразование, поскольку не все целые значения представляют собой раельные часы. Операция преобразования строит объект Hour, используя остаток от деления целого параметра на 24.

struct Hour

{

...

public Hour(int hr)

{

this.value = hr % 24;

}

public static explicit operator Hour (int from)

{

return new Hour(from);

}

private int value;

}

Когда следует объявлять операторы преобразования как явные или неявные? Если преобразование всегда безопасно, не существует вероятности потерять информацию, и не может быть сгенерировано исключение, оно может быть определено как неявное преобразование. В противном случае, оно должно быть объявлено как явное преобразование.

Операции преобразования предоставляют альтернативный способ решения проблемы обеспечения симметрических операций. Например, предположим, определяется operator+ для добавления объектов Hour к объектам Hour, и объектов Hour к значениям int. Вместо того чтобы для структур Hour обеспечивать три версии operator+ (Hour + Hour, Hour + int и int + Hour), можно обеспечить единую версию operator+, принимающего два параметра Hour, и операцию неявного преобразования int в Hour.

struct Hour

{

public Hour(int hr)

{

this.value = hr % 24;

}

public static Hour operator +(Hour lhs, Hour rhs)

{

return new Hour(lhs.value + rhs.value);

}

public static implicit operator Hour (int from)

{

return new Hour (from);

}

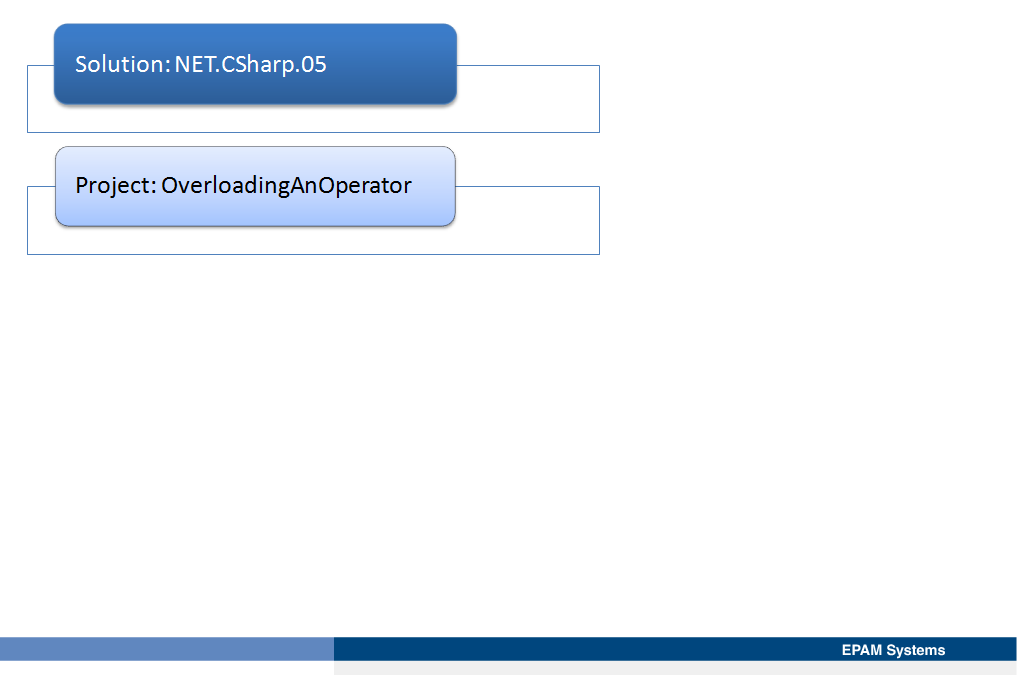
...

private int value;

}

Если нужно добавить объект Hour к объекту int (в любом порядке), Visual C# компилятор автоматически преобразует объект int в объект Hour, а затем вызовет operator+ с двумя аргументами Hour.

## Демонстрация: Перегрузка операций



1. Статические поля можно также инициализацировать из конструктора. [↑](#footnote-ref-1)
2. Общепринятой практикой является расположение модификатора static после модификатора доступа, хотя можно поместить его и перед ним. [↑](#footnote-ref-2)