|  |
| --- |
| , RD Dep. |
| Конспект и раздаточный материал  NET.C#.04 Создание новых типов в C# |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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. Создание и использование перечислений 3](#_Toc301306277)

[1.1. Что такое перечисление? 3](#_Toc301306278)

[1.2. Создание новых типов перечисления 4](#_Toc301306279)

[1.3. Инициализация и присваивание переменных перечисления 5](#_Toc301306280)

[1.4. Демонстрация: Создание и использование перечислений 7](#_Toc301306281)

[2. Урок 2. Создание и использование классов 7](#_Toc301306282)

[2.1. Что такое класс? 8](#_Toc301306283)

[2.2. Добавление элементов в классы 9](#_Toc301306284)

[2.3. Определение конструкторов и инициализация объектов 12](#_Toc301306285)

[2.4. Создание объектов 14](#_Toc301306286)

[2.5. Доступ к членам класса 15](#_Toc301306287)

[2.6. Демонстрация: Создание и инициализация объектов, доступ к элементам классам 16](#_Toc301306288)

[2.7. Использование разделяемых классов и разделяемых методов 17](#_Toc301306289)

[3. Урок 3. Создание и использование структур 18](#_Toc301306290)

[3.1. Что такое структура? 19](#_Toc301306291)

[3.2. Определение и использование структуры 20](#_Toc301306292)

[3.3. Инициализация структуры 21](#_Toc301306293)

[4. Урок 4. Сравнение ссылочных и значимых типов 22](#_Toc301306294)

[4.1. Сравнение ссылочных и значимых типов 23](#_Toc301306295)

[4.2. Передача значимого типа в метод по ссылке 26](#_Toc301306296)

[4.3. Упаковка и распаковка 27](#_Toc301306297)

[4.4. Демонстрация: Упаковка и распаковка 29](#_Toc301306298)

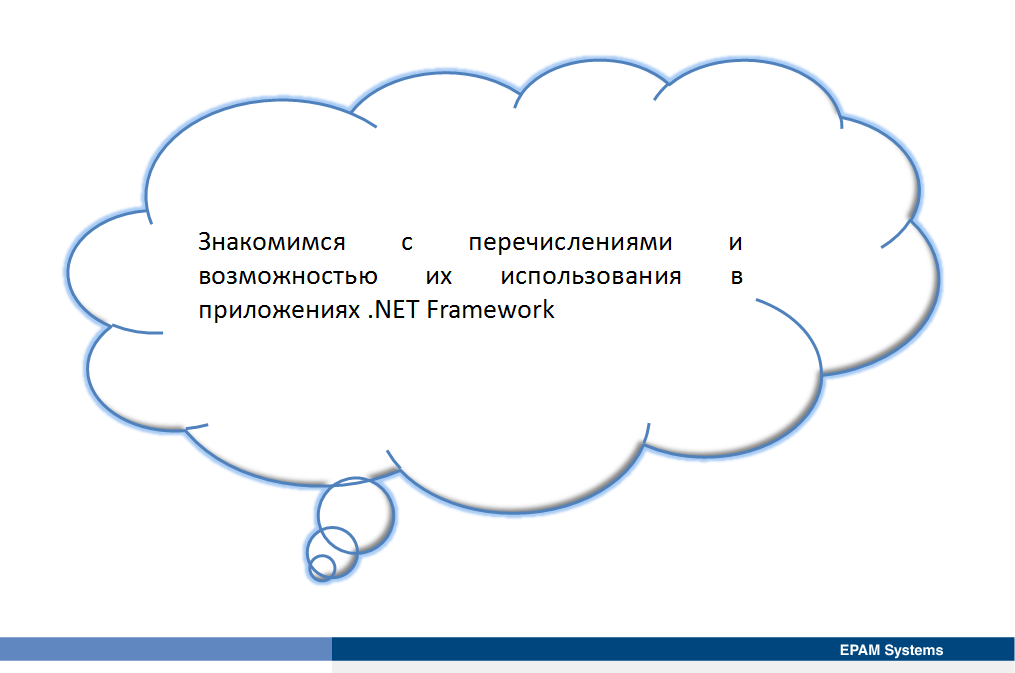
[4.5. Обнуляемые типы 29](#_Toc301306299)

[4.6. Демонстрация: Обнуляемые типы 31](#_Toc301306300)

[4.7. Тип dynamic 31](#_Toc301306301)

[4.8. Демонстрация: Тип dynamic 33](#_Toc301306302)

# Урок 1. Создание и использование перечислений



Перечисление это набор связанных постоянных значений, имеющих предопределенный порядок. Перечисления очень полезны при работе с данными, у которых есть определенный диапазон значений. Для моделирования дней недели, например, чтобы указать все дни с воскресенья по субботу, можно использовать цифры от 0 до 6, результатом будет плохо читаемый и трудно поддерживаемый код.

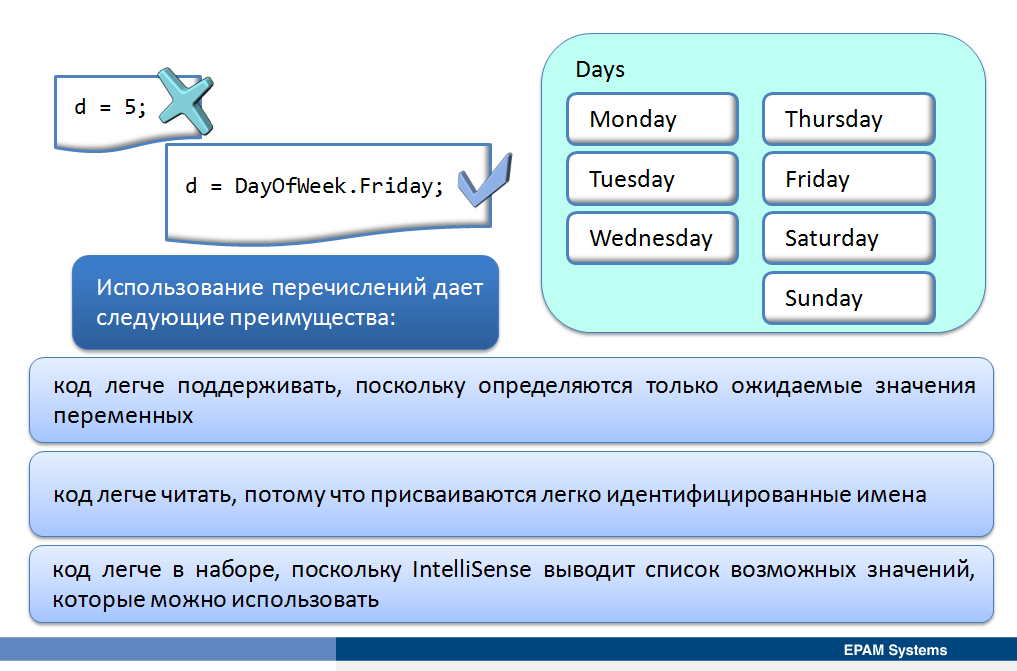
Если приложение содержит следующий оператор,

d = 5;

несложно определить, что переменной d присваивается значение 5, однако отнюдь не является очевидным, что этой переменной присаивается день недели пятница. Оператор в следующем примере кода более интуитивен, поскольку очевидно, что d ссылается на день недели пятница.

d = DayOfWeek.Friday;

## Что такое перечисление?



Тип перечисление определяет набор связанных именованых констант. Перечислимый тип является скалярным типом, имеющим определяемый пользователем диапазон значений. После создания перечисления можно объявлять переменные этого типа и присваивать значения этих переменных во многом таким же образом, как это делается, используя встроенные скалярные типы C#, такие как int или float. Перечисления можно использовать для представления множества значений из конкретной предметной области для того, чтобы сделать код более легким для чтения и поддержки.

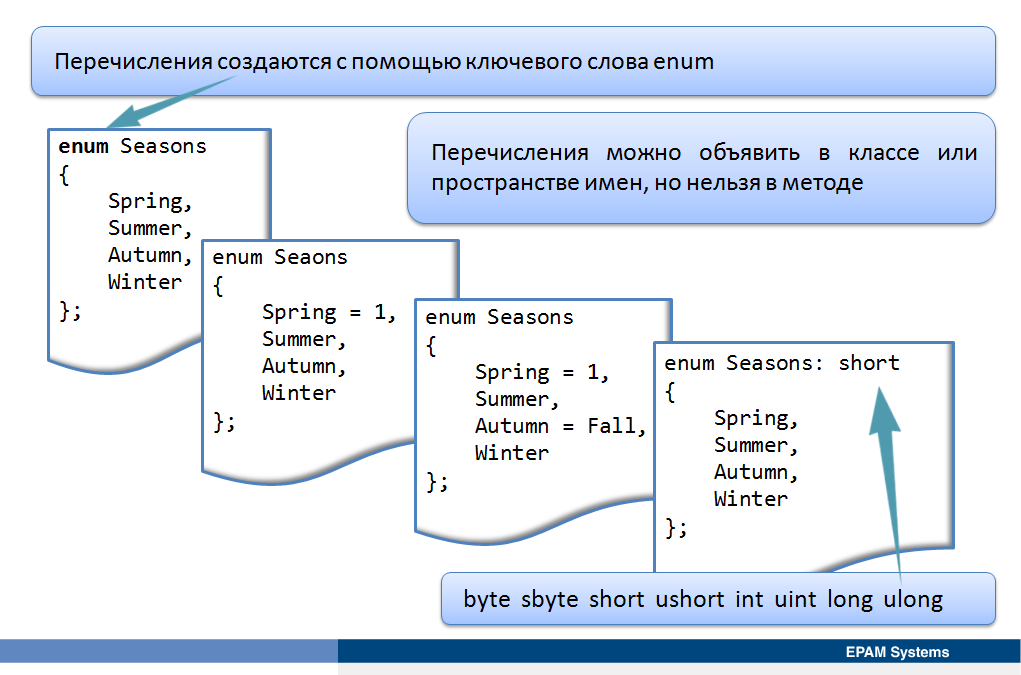
Базовая библиотека классов .NET Framework содержит различные перечисления, которые можно использовать в приложениях. Многие из классов .NET Framework используют эти перечисления как возвращаемые методом значения и параметры метода.

Основным отличием перечисления в С# от перечислений в других языках программировния, например Java , является то, что перечисления в Microsoft Visual C# основаны на целых типах данных (например, int и long).

Использование перечислений дает следующие преимущества:

* код легче поддерживать, поскольку определяются только ожидаемые значения переменных;
* код легче читать, потому что присваиваются легко идентифицированные имена;
* код легче в наборе, поскольку IntelliSense выводит список возможных значений, которые можно использовать.

## Создание новых типов перечисления



Создавать собственные перечисления можно с помощью ключевого слова enum. Для этого нужно перечислению присвоить имя, а затем перечислить список значений, которые принимает перечисление. Перечисления это типы, поэтому их можно объявить в классе или пространстве имен, но нельзя в методе.

enum Seasons

{

Spring,

Summer,

Autumn,

Winter

};

Внутренне, тип перечисление связывает целое число с каждым элементом перечисления. По умолчанию нумерация начинается с 0 для первого элемента и увеличивается с шагом 1. Можно связать конкретную целочисленную константу (например, 1) с литералом перечисления (например, Spring), как показано в следующем примере. В этом случае литералы перечисления Summer, Autumn и Winter автоматически принимают значения 2, 3 и 4.

enum Seasons

{

Spring = 1,

Summer,

Autumn,

Winter

};

Числовое значение, связанное с каждым литералом перечисления, становится существенным, если в коде осуществляется перебор возможных значений, которые может принимать переменная перечисления. Для переменных перечисления при продвижении вперед или назад по возможным значениям переменной можно использовать операции инкримента и декримента.

Существует возможность назначить более одного литерала перечисления одной и той же величине. Например, в Соединенном Королевстве, осень называется fall. При построении перечисления Season можно угодить обеим культурам.

enum Seasons

{

Spring = 1,

Summer,

Autumn = Fall,

Winter

};

При объявляении перечисления литералы перечисления базируются на значениях типа int. Можно базировать перечисления на другом типе, основанном на базовом целочисленном типе. В следующем примере объявляется, что базовым типом перечисления Season является тип short, а не тип int.

enum Seasons: short

{

Spring,

Summer,

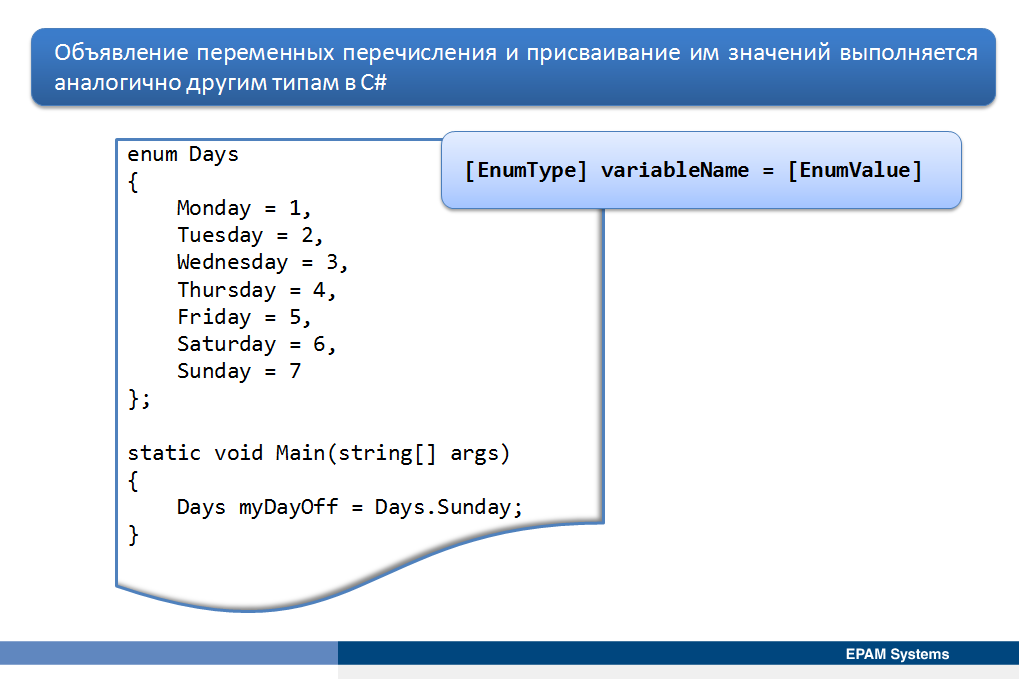
Autumn,

Winter

};

Основной причиной для переопределения базового типа перечисления является экономия памяти – переменная типа int занимает больше памяти, чем переменная типа short. Если не требуется весь спектр значений, доступных для типа int, имеет смысл использовать в качестве базового типа перечисления более короткий тип данных. Перечисления могут базироваться на любом из восьми целых типов: byte, sbyte, short, ushort, int, uint, long или ulong, при этом значения всех литералов перечисления должны поместиться внутри диапазона выбранного базового типа.

## Инициализация и присваивание переменных перечисления



Объявление переменных перечисления и присваивание им значений выполняется аналогично другим типам в C#. Тип переменной это имя перечисления и значения, которые ей можно присвоить, и являющиеся литералами, которые определяют перечисление.

enum Days

{

Monday = 1,

Tuesday = 2,

Wednesday = 3,

Thursday = 4,

Friday = 5,

Saturday = 6,

Sunday = 7

};

static void Main(string[] args)

{

Days myDayOff = Days.Sunday;

}

С переменными типа перечисления можно выполнять простые операции во многом таким же образом, как и с переменной целого типа.

enum Days

{

Monday = 1,

Tuesday = 2,

Wednesday = 3,

Thursday = 4,

Friday = 5,

Saturday = 6,

Sunday = 7

};

...

for (Days dayOfWeek = Days.Monday; dayOfWeek <= Days.Sunday;dayOfWeek++)

{

Console.WriteLine(dayOfWeek);

}

/\* Output is:

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

Sunday

\*/

Сравнивать переменные перечисления можно с помощью значения литералов, которые определяет перечисление. Сравнения выполняются с использованием базовых целочисленных значений каждого литерала. Кроме того, для переменных перечисления можно выполнять целочисленные операции, такие как инкримент и декримент. Эффектом операции инкримента является продвижение переменной перечисления до следующего значения, а декримента – откат к предыдущему значению. Наконец, при отображении значения переменной перечисления, значение, которое отображается, является соответствующим литералом перечисления. Однако, если увеличить переменную перечисления за пределы диапазона целочисленных значений, используемых перечислением, она будет отображаться соответствующим целым значением.

for (Days dayOfWeek = Days.Monday; dayOfWeek <= Days.Sunday + 1;dayOfWeek++)

{

Console.WriteLine(dayOfWeek);

}

/\* Output is:

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

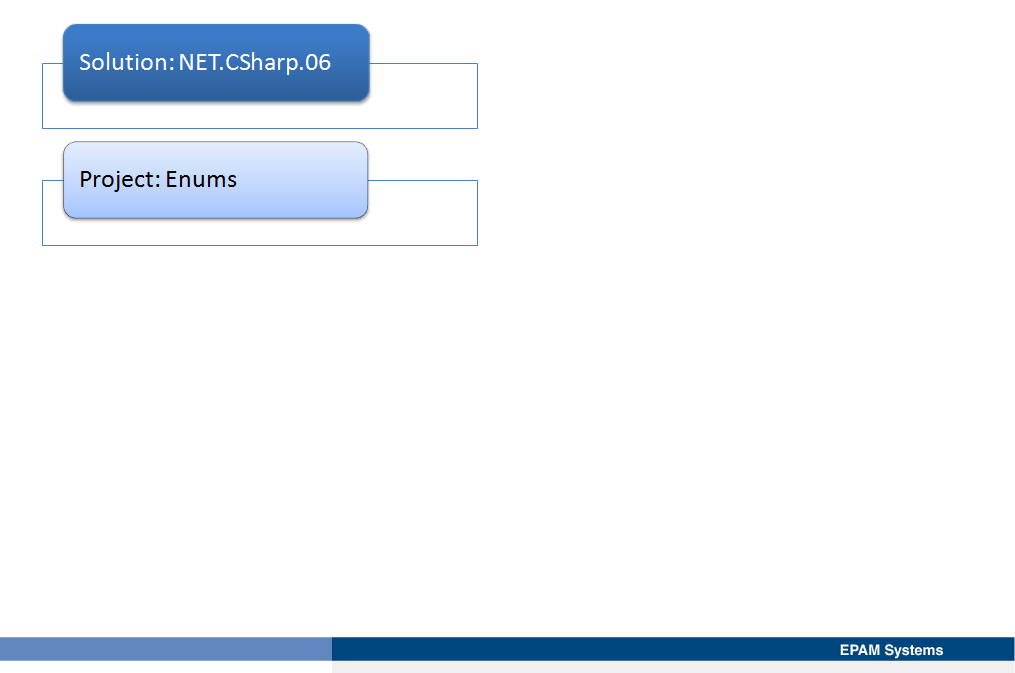
Sunday

8

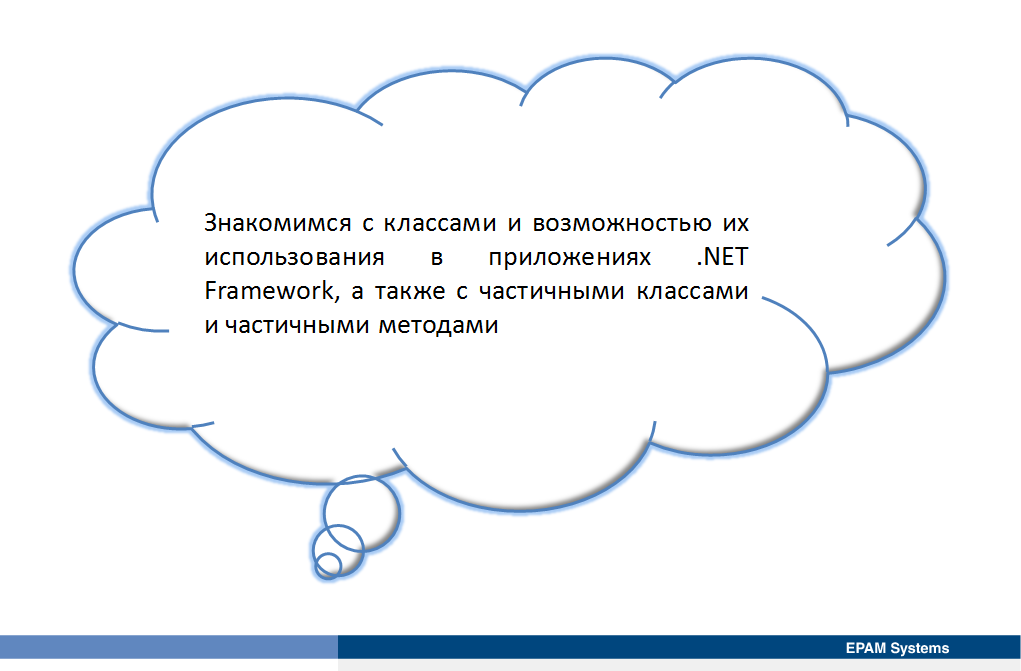
\*/

Помимо операций инкримента и декримента нельзя выполнять любые другие арифметические операции над переменными перечисления, если предварительно они не придены к переменной целого типа. В действительности это не ограничение, а в большинстве случаев семантика арифметических операций, не имеющих смысла для типа перечисления (например, что означает выражение Days.Monday + Days.Wednesday?)

## Демонстрация: Создание и использование перечислений



# Урок 2. Создание и использование классов



Visual C# является объектно-ориентированным языком программирования. Вся логика приложений C# содержится в структурах и классах. Урок объясняет, как создавать собственные классы и использовать их в приложениях .NET Framework. Урок вводит такие понятия, как частичные классы и частичные методы.

## Что такое класс?



При создании приложений C# можно использовать классы, представляющие основные типы данных в приложении. .NET Framework предлагает большое количество служебных классов, позволяя при этом определить собственные классы, инкапсулирующие данные и логику, специфичные для приложения.

О классе можно думать как о чертеже, на основании которого можно создавать объекты. Класс определяет характеристики объекта, такие как данные, которые может содержать объект и операции, которые объект может выполнять. Объект является экземпляром класса. Если класс, это чертеж, то объект это элемент, который создается с помощью этого чертежа.

В реальном мире чертеж дома это класс, а дом, который построен по этому чертежу, это объект. Можно построить много экземпляров домов, следуя одному плану. Все дома будут иметь одинаковый формат и структуру (те же комнаты), при этом они остаются разными домами. В объектно-ориентированном программировании, можно, например, определить класс House, определяющий размеры и особенности конкретного помещения, а затем создать один или несколько объектов этого класса. Каждый объект House будет иметь макет комнаты и размеры, определяемые классом, но некоторые другие аспекты каждого объекта House могут отличаться деталями, такими, например, как расположение объекта House или цвет входной двери.

Для того, чтобы добавить в проект новый класс, можно использовать Microsoft Visual Studio. Как правило, каждый класс располагается в отдельном файле, имеющим то же имя, что и имя класса. Visual Studio 2010 генерирует шаблонный код в исходном файле для нового класса. Шаблонный код обычно включает в себя директивы using для подключения постранств имен, определение пространства имен[[1]](#footnote-1), относящегося к классу, и определение самого класса.

Для добавления в проект нового класса необходимо (Рис. .):

* В Solution Explorer щелкнуть правой кнопкой мыши проект, выбрать команду Add, а затем нажать кнопку Class.
* В диалоговом окне Add New Item ввести имя исходного файла, который будет содержать новый класс, а затем нажать кнопку Add.

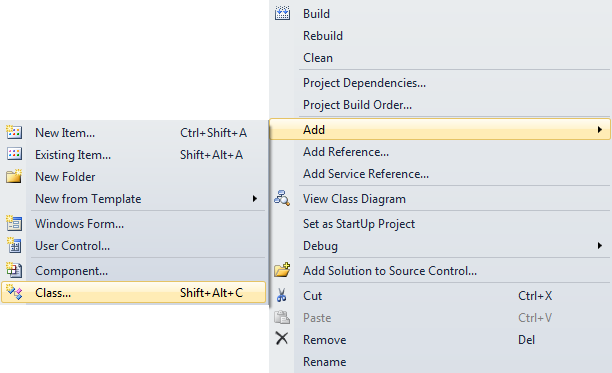


Рис. 1.

В следующем примере показано определение нового класса House.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace HouseSystem

{

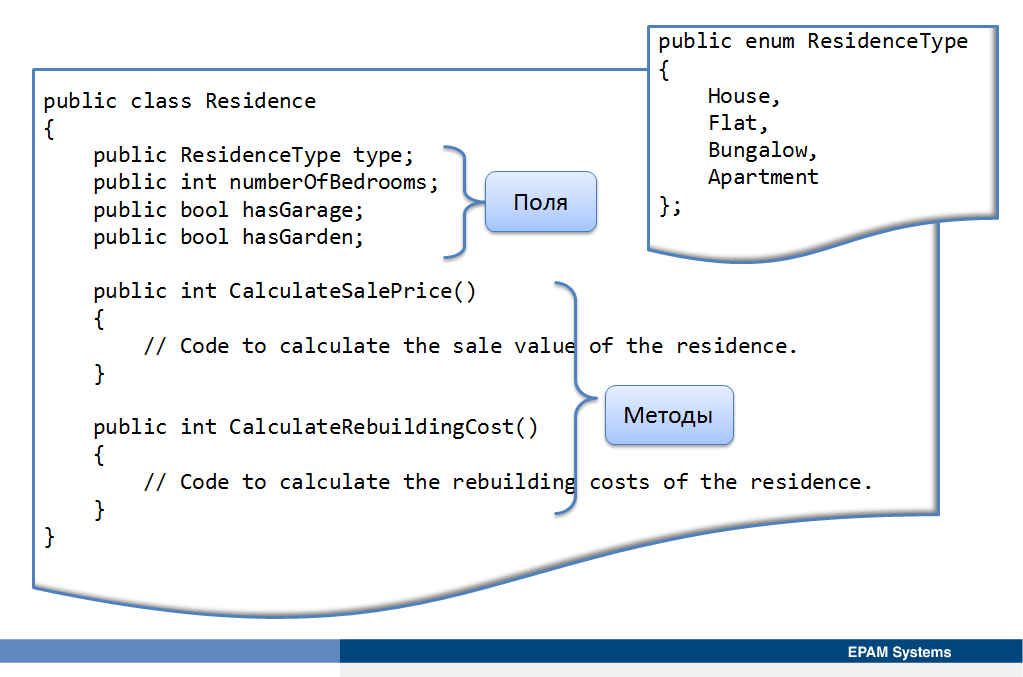
class House

{

}

}

## Добавление элементов в классы



В класс могут добавляться поля и методы, определяющие данные и поведение класса. В зависимости от цели и предназначенной функциональности класса в нем можно определить любое количество полей и методов[[2]](#footnote-2).

О поле можно думать как о переменной, которая имеет областью видимости класс. Все методы, которые определены в классе, могут получить доступ к полю. Как и переменная, каждое поле имеет имя, тип данных и модификатор доступа. Если модификатор доступа для поля явно не указан, уровень доступа по умолчанию является private. Это означает, что переменная может быть доступна только методам, определенным в классе. Если нужно сделать поля доступными методам, определенным в других классах, его необходимо пометить как public. Определения полей можно размещать в любой точке класса. Некоторые программисты предпочитают размещать определения поля вблизи начала класса, чтобы код был легко читаемым для других программистов. При определении поля можно присвоить ему значение по умолчанию, хотя при создании объекта можно изменить присвоенное полю значение, используя конструктор.

Метод это процедура или функция, определнная внутри класса. Методы используются для реализации поведения класса. Каждый метод имеет имя, список параметров, тип возвращаемого значения и модификатор доступа. Метод имеет полный и неограниченный доступ ко всем остальным членам класса. Это важный аспект объектно-ориентированного программирования, методы инкапсулируют в классе операции над полями. При ссылке в методе на поле класса можно сопроводить это поле префиксом this. Такой подход помогает устранить неоднозначность любых ссылок (например, параметр в методе может иметь то же имя, что и поле в классе, хотя это и не рекомендуется) и помогает сделать код более простым в обслуживании.

public bool hasGarage;

public void OpenGarageDoor(int doorId)

{

if(this.hasGarage)

{

// Code to run if a residence has a garage.

}

}

В следующем примере показан класс Residence, используемый как часть реального приложения. Класс состоит из четырех полей, представляющих вид проживания, количество спален в нем, а такжн наличие гаража и сада. Класс имеет методы, которые расчитывают стоимость жилья для продажи и затраты на восстановление жилья при страховании.

public enum ResidenceType

{

House,

Flat,

Bungalow,

Apartment

};

public class Residence

{

public ResidenceType type;

public int numberOfBedrooms;

public bool hasGarage;

public bool hasGarden;

public int CalculateSalePrice()

{

// Code to calculate the sale value of the residence.

}

public int CalculateRebuildingCost()

{

// Code to calculate the rebuilding costs of the residence.

}

}

Класс можно создать вручную, написав код в окне Code Editor. Однако, можно также использовать окно Class Designer для разработки класса и добавления полей и методов графически. Для использования окна Class Designer, необходимо добавить к проекту диаграмму классов (Рис. 2). Диаграмма классов автоматически включает в себя все перечисления, классы и структуры, определенные в проекте. Она также предоставляет инструментарий, который можно использовать для добавления новых элементов в схему и добавления полей и их методов (Рис. 3).

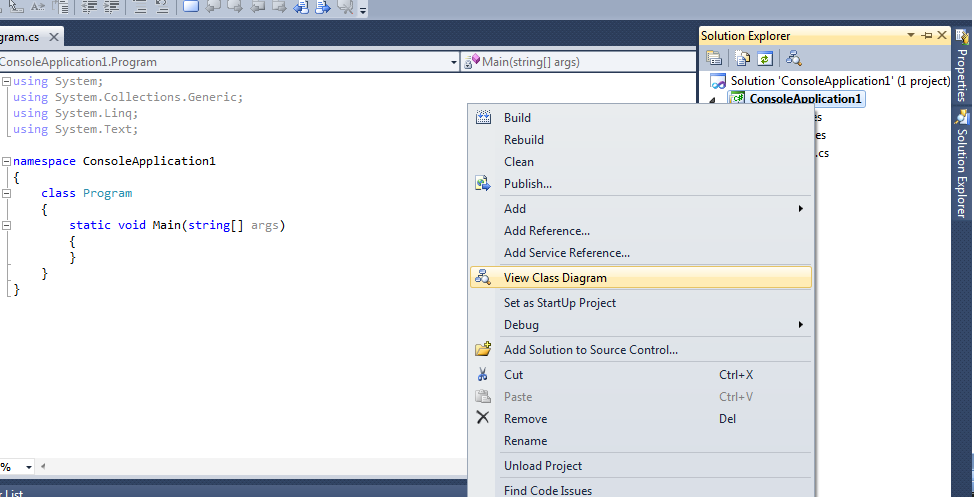


Рис. 2.

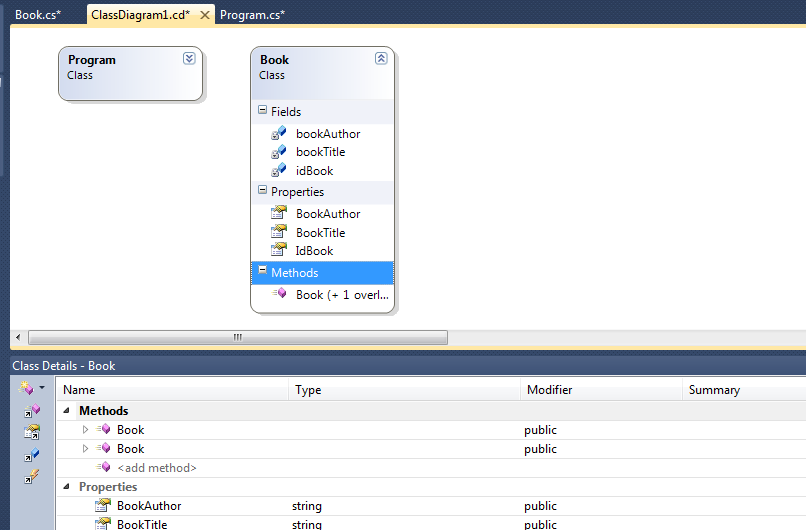
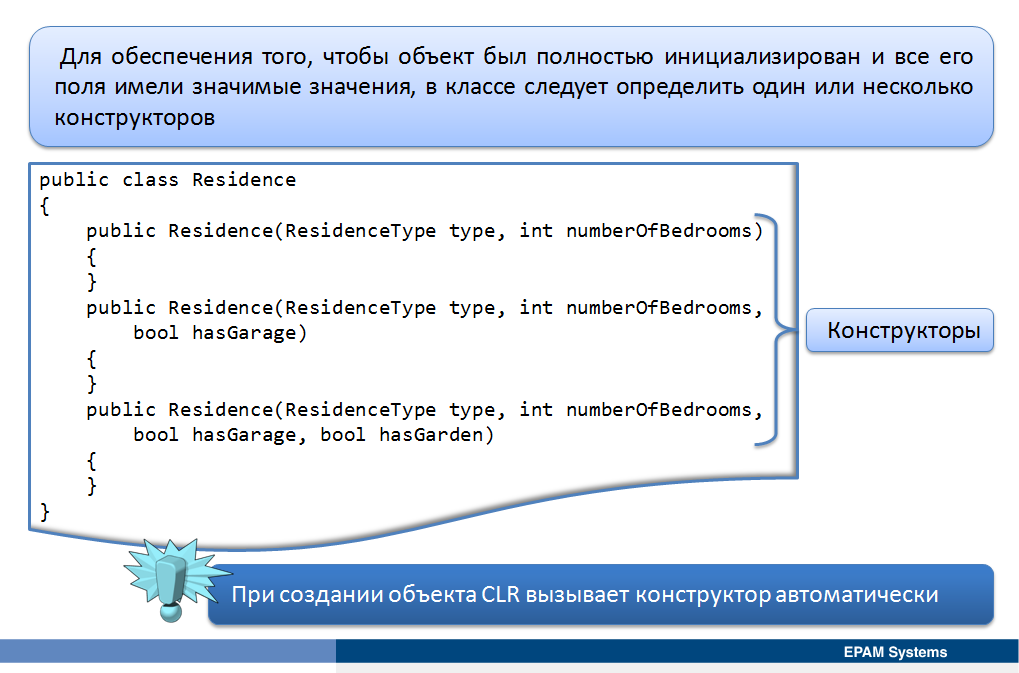


Рис. 3.

## Определение конструкторов и инициализация объектов



При создании объекта важно обеспечить, чтобы объект был полностью инициализирован и чтобы все его поля имели значимые значения. Для достижения этой цели в классе следует определить один или несколько конструкторов. При создании объекта общеязыковя среда выполнения (CLR) автоматически вызывает конструктор.

Если поле в классе не инициализировано, ему присваивается значение по умолчанию. Если поле имеет числовое значение, оно инициализируется нулем. Если поле является логическим значением, оно инициализируется значением false. Если поле является строкой, он инициализируется значением null. Если поле является классом, оно также инициализируется значением null.

Конструктор это специальный метод, автоматически вызываемый CLR при создании объекта. При определении конструктора соблюдаются следующие правила и принципы:

* Конструкторы имеют то же имя, что и класс, в котором они определены.
* Конструкторы не имеют типа возвращаемого значения (даже void), но они могут принимать параметры. Можно определить любое количество конструкторов в одном классе, при условии, что каждый конструктор имеет уникальный список параметров. Конструктор, не принимающий никаких параметров, называется конструктором по умолчанию.
* Конструкторы, как правило, объявляются с модификатором доступа public, чтобы любая часть приложения имела доступ к ним для создания и инициализации объектов. Если нужно ограничить части приложения, которые могут создавать и инициализировать объекты, для конструкторов можно определить более строгий уровень доступа.
* Конструкторы обычно инициализируют некоторые или все поля объекта, а также могут выполнять любые дополнительные задачи инициализации, требуемые классу.

Если в классе не определяется никаких конструкторов, C# компилятор автоматически создает конструктор по умолчанию (конструктор без параметров). Этот конструктор ничего не делает, но он позволяет создать экземпляр класса. Однако, если определить один и более собственных конструкторов, компилятор C# не будет создавать конструктор по умолчанию.

В следующем примере показано, как определить три конструктора для класса Residence:

* Первый конструктор принимает два параметра и задает тип жилья и количество спален в нем.
* Второй конструктор принимает три параметра и устанавливает тип жилья, количество спален и наличие у жилья гаража.
* Третий конструктор принимает четыре параметра и устанавливает тип жилья, количество спален в нем, наличие у жилья гаража и сада.

Следует обратить внимание на использование ключевого слова this для различия между полями и параметрами с одинаковым именем.

public enum ResidenceType

{

House,

Flat,

Bungalow,

Apartment

};

public class Residence

{

public ResidenceType type;

public int numberOfBedrooms;

public bool hasGarage;

public bool hasGarden;

public Residence(ResidenceType type, int numberOfBedrooms)

{

this.type = type;

this.numberOfBedrooms = numberOfBedrooms;

}

public Residence(ResidenceType type, int numberOfBedrooms, bool hasGarage)

{

this.type = type;

this.numberOfBedrooms = numberOfBedrooms;

this.hasGarage = hasGarage;

}

public Residence(ResidenceType type, int numberOfBedrooms, bool hasGarage, bool hasGarden)

{

this.type = type;

this.numberOfBedrooms = numberOfBedrooms;

this.hasGarage = hasGarage;

this.hasGarden = hasGarden;

}

...

}

Возможно вызывать один конструктор из другого с помощью ключевого слова this в рамках объявления конструктора. В этом случае будет работать конструктор с соответствующей сигнатурой. Используя эту возможность можно реализовать конструктор по умолчанию, вызывающий параметризованный конструктор с множеством значений по умолчанию для каждого параметра, как показано в следующем примере.

public class Residence

{

public ResidenceType type;

public int numberOfBedrooms;

public bool hasGarage;

public bool hasGarden;

public Residence(ResidenceType type, int numberOfBedrooms, bool hasGarage, bool hasGarden)

{

this.type = type;

this.numberOfBedrooms = numberOfBedrooms;

this.hasGarage = hasGarage;

this.hasGarden = hasGarden;

}

// Default constructor creates a 3-bedroom residence with a garage and a garden

public Residence() : this(ResidenceType.House, 3, true, true)

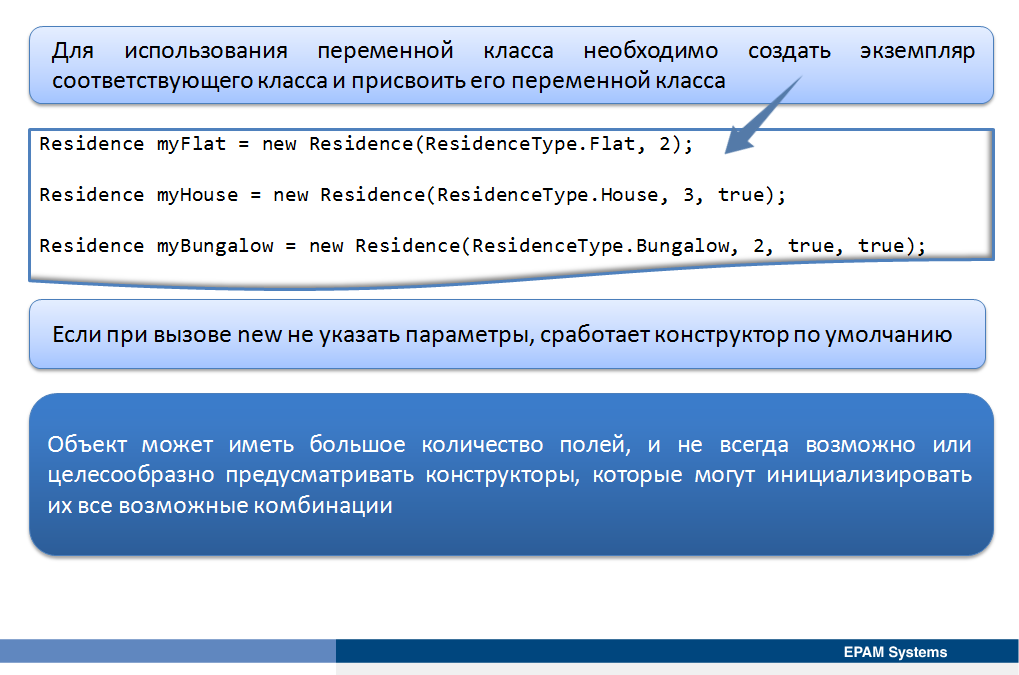
{

}

...

}

## Создание объектов



При объявлении переменной класса, она изначально не определена. Для использования переменной класса необходимо создать экземпляр соответствующего класса и присвоить его переменной класса.

В следующем примере кода показано создание и использование экземпляров класса с помощью конструкторов, определенных в предыдущем разделе.

// Create a flat with two bedrooms.

Residence myFlat = new Residence(ResidenceType.Flat, 2);

// Create a house with three bedrooms and a garage.

Residence myHouse = new Residence(ResidenceType.House, 3, true);

// Create a bungalow with two bedrooms, a garage, and a garden.

Residence myBungalow = new Residence(ResidenceType.Bungalow, 2, true, true);

Если при вызове new не указать параметры, сработает конструктор по умолчанию. Если для класса определить один или несколько конструкторов, C# компилятор автоматически не создает конструктор по умолчанию.

Экземпляр объекта создается путем вызова конструктора. Конструктор может принимать параметры, которые определяют значения для инициализации полей объекта. Однако, объект может иметь большое количество полей, и не всегда возможно или целесообразно предусматривать конструкторы, которые могут инициализировать их все возможные комбинации. В следующем примере класс Residence обеспечивает три конструктора.

public enum ResidenceType

{

House,

Flat,

Bungalow,

Apartment

};

public class Residence

{

public ResidenceType type;

public int numberOfBedrooms;

public bool hasGarage;

public bool hasGarden;

public Residence(ResidenceType type, int numberOfBedrooms)

{

this.type = type;

this.numberOfBedrooms = numberOfBedrooms;

}

public Residence(ResidenceType type, int numberOfBedrooms, bool hasGarage)

{

this.type = type;

this.numberOfBedrooms = numberOfBedrooms;

this.hasGarage = hasGarage;

}

public Residence(ResidenceType type, int numberOfBedrooms, bool hasGarage, bool hasGarden)

{

this.type = type;

this.numberOfBedrooms = numberOfBedrooms;

this.hasGarage = hasGarage;

this.hasGarden = hasGarden;

}

...

}

С помощью указанных конструкторов возможно создать объект Residence и инициализировать различные поля, но одна комбинация отсутствует: можно указать, что у жилья есть сад, если явно указано наличие гаража; не существует конструктора, который позволяет инициализировать поле hasGarden без установки свойства hasGarage. Можно попробовать определить следующий дополнительный конструктор.

// Constructor to initialize the hasGarden field without setting hasGarage.(CTE!)

public Residence(ResidenceType type, int numberOfBedrooms,bool hasGarden)

{

this.type = type;

this.numberOfBedrooms = numberOfBedrooms;

this.hasGarden = hasGarden;

}

Однако при перегрузке конструкторы следуют тем же правилам, что и методы, а, следовательно, нельзя определить два или более конструктора, имеющие одинаковые сигнатуры. В примере класс Residence не скомпилируется, поскольку два конструктора имеют одинаковые сигнатуры.

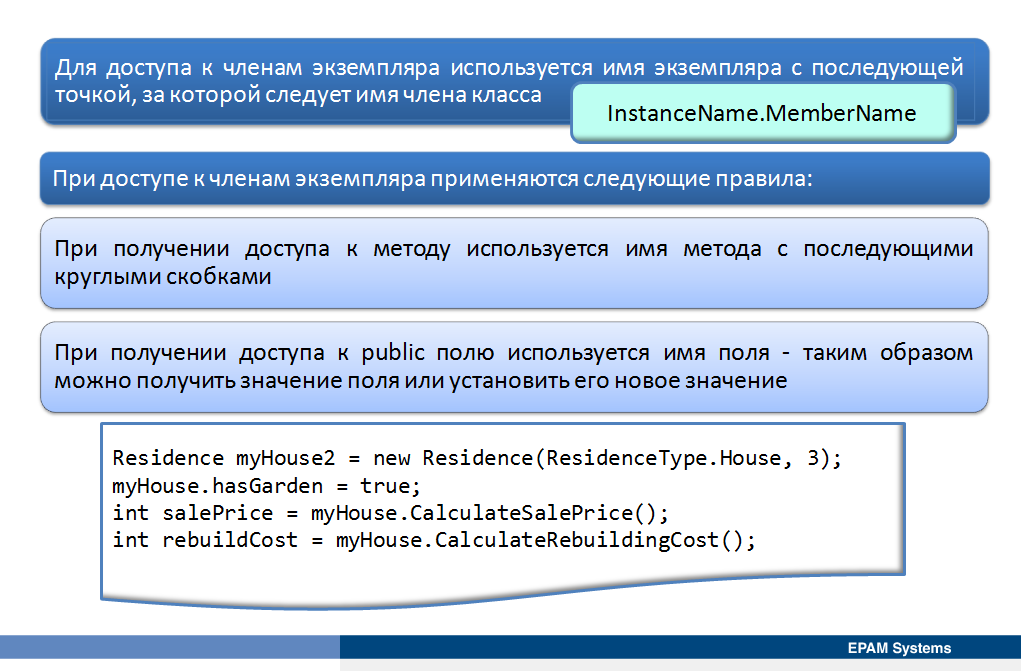
Решить эту проблему можно, используя инициализатор объекта, который позволяет при создании объекта с помощью конструктора инициализировать любые поля объекта. Для этого в скобках после вызова конструктора указываются поля для инициализации и их значения.

// Create a house with three bedrooms and a garden.

Residence myHouse1 = new Residence(ResidenceType.House, 3) { hasGarden = true };

При создании объекта с помощью инициализатора объекта работает соответствующий конструктор, а затем полям присваиваются значения. Присваивание полям значений может отменить инициализацию, выполненную в конструкторе.

## Доступ к членам класса



Для доступа к членам экземпляра используется имя экземпляра с последующей точкой, за которой следует имя члена класса. При доступе к членам экземпляра применяются следующие правила и директивы:

* При получении доступа к методу используется имя метода с последующими круглыми скобками. В скобках, передаются значения для всех параметров, требуемых методу. Если метод не принимает параметров, скобки необходимы.
* При получении доступа к public полю используется имя поля. Таким образом можно получить значение поля или установить его новое значение.

Приведенный ниже код выполняет следующие задачи:

* Создает экземпляр класса Residence с помощью конструктора, определяющего тип жилья и количество спален в нем.
* Устанавливает свойство hasGarden истинным, чтобы указать, что у жилья есть сад. (Это также можно сделать с помощью объекта инициализации при создании объекта.)
* Вызывает метод CalculateSalePrice для определения текущей рыночной стоимости жилья.
* Вызывает метод CalculateRebuildingCost для определения стоимости восстановления жилья с целью страхования.

// Create a three-bedroom house.

Residence myHouse2 = new Residence(ResidenceType.House, 3);

// Indicate that the residence has a garden.

myHouse.hasGarden = true;

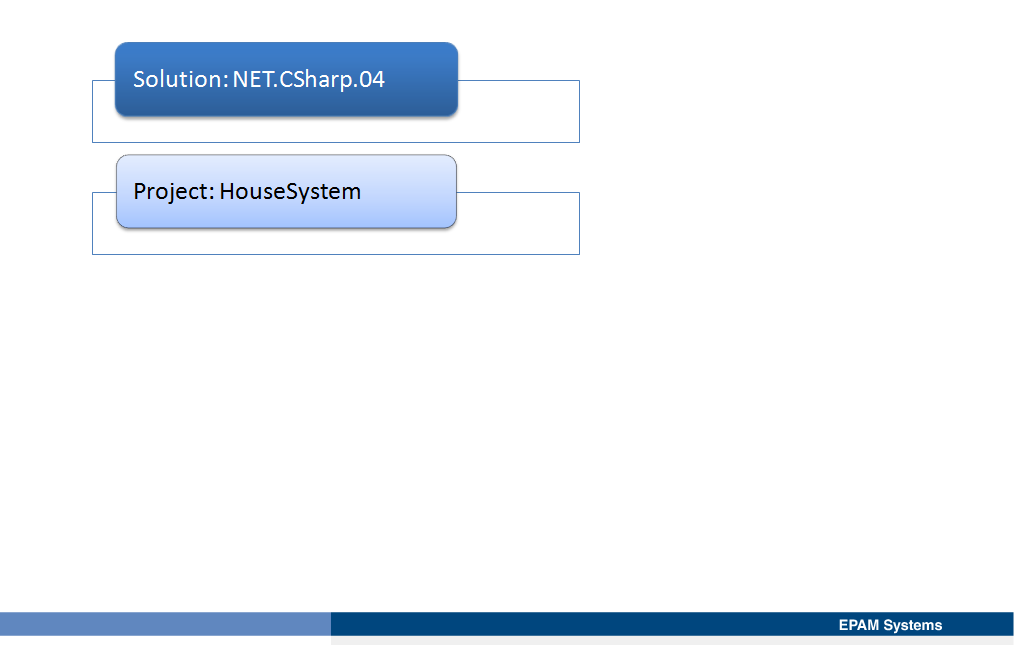
// Calculate the market value.

int salePrice = myHouse.CalculateSalePrice();

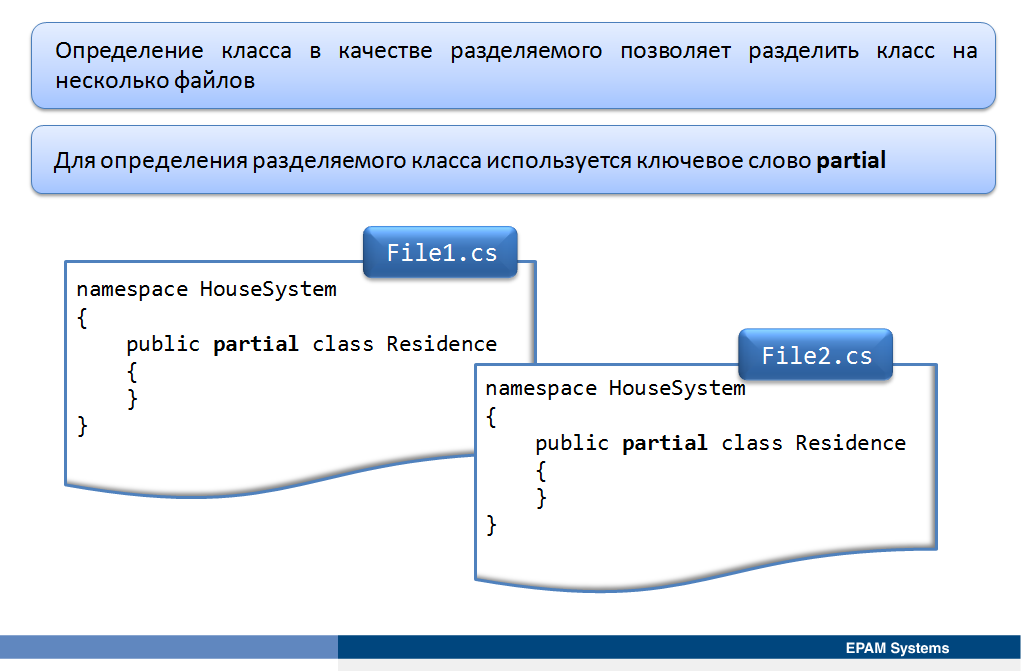
// Get the rebuilding costs.

int rebuildCost = myHouse.CalculateRebuildingCost();

## Демонстрация: Создание и инициализация объектов, доступ к элементам классам



## Использование разделяемых классов и разделяемых методов



Существуют ситуации, когда необходимо разделить определение класса по нескольким исходным файлам. Например, когда над классом одновременно работают несколько разработчиков, или имеются части класса, которые никогда не должны измениться. Для этой цели .NET Framework предоставляет концепцию разделяемого класса (partial class). Некоторые классы в .NET Framework и проектах Visual Studio используют эту концепцию. Например, приложения WPF используют разделяемые классы для отделения кода, который Visual Studio генерирует для инициализации окна, от программной логики, добавляемой в процессе ввода данных пользователем и отображения результатов.

Определение класса в качестве разделяемого позволяет разделить класс на несколько файлов. Для определения разделяемого класса используется ключевое слово partial.

//File1.cs

namespace HouseSystem

{

public partial class Residence

{

}

}

//File2.cs

namespace HouseSystem

{

public partial class Residence

{

}

}

При определении разделяемого класса применяются следующие правила:

* Каждая часть класса должны быть доступна при компиляции приложения, поскольку класс компилируется в одно целое.
* Каждая часть класса должна начинаться с ключевого слова partial.
* Разделяемый класс не может быть разбит на несколько сборок. Каждая часть разделяемого класса должна существовать в одной и той же сборке.
* Ключевое слово partial должно быть префиксом ключевого слова class.

При определении разделяемого класса в нем можно определить один или несколько разделяемых методов. Разделяемый метод определяет сигнатуру метода в одном файле, содержащем разделяемый класс, и опционально указывает код, который метод реализует в другом файле, содержащем разделяемый класс. Если разделяемый метод не реализован, следует удалить из класса и любые вызывающие его операторы. Разделяемые методы обычно используются фреймворками, где они обеспечивают механизм для вызова методов, когда разработчики вне фреймворка разрабатывают код для этих методов.

Следующие примеры кода демонстрируют разделение объявления метода от реализации посредством разделяемого файла. В коде первого файла показан разделяемый класс, реализущийся как часть класса FrameworkClass. Класс FrameworkClass определяет разделяемый метод DoWork. Метод FrameworkMethod в данном классе вызывает метод DoWork. Метод DoWork реализуется в отдельном файле. Если второй файл не реализует метод DoWork, вызов этого метода в методе FrameworkMethod будет игнорироваться компилятором.

//Code provided by the Framework.

public partial class FrameworkClass

{

partial public DoWork(int data);//The definition of the partial method

public public FrameworkMethod()

{

...

DoWork(99);//Call the partial mehhod

...

}

}

//Code provided by a developer to link into the Framework

public partial class FrameworkClass

{

partial public DoWork(int data)

{

...

//Code that implements the DoWork method

...

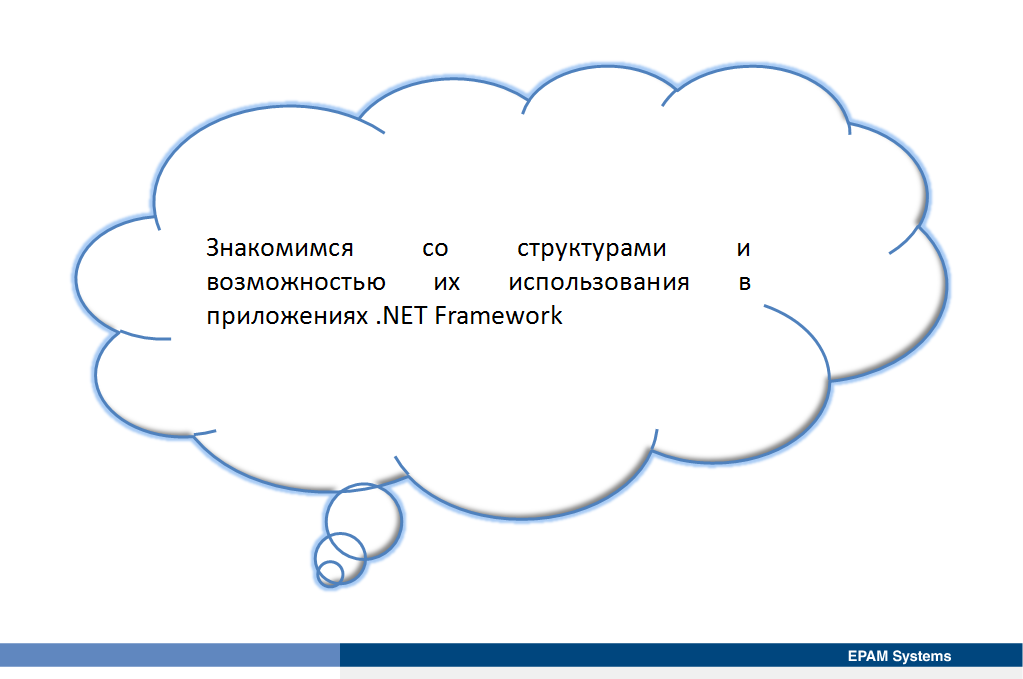
}

}

При определении разделяемых методов необходимо соблюдать следующие правила:

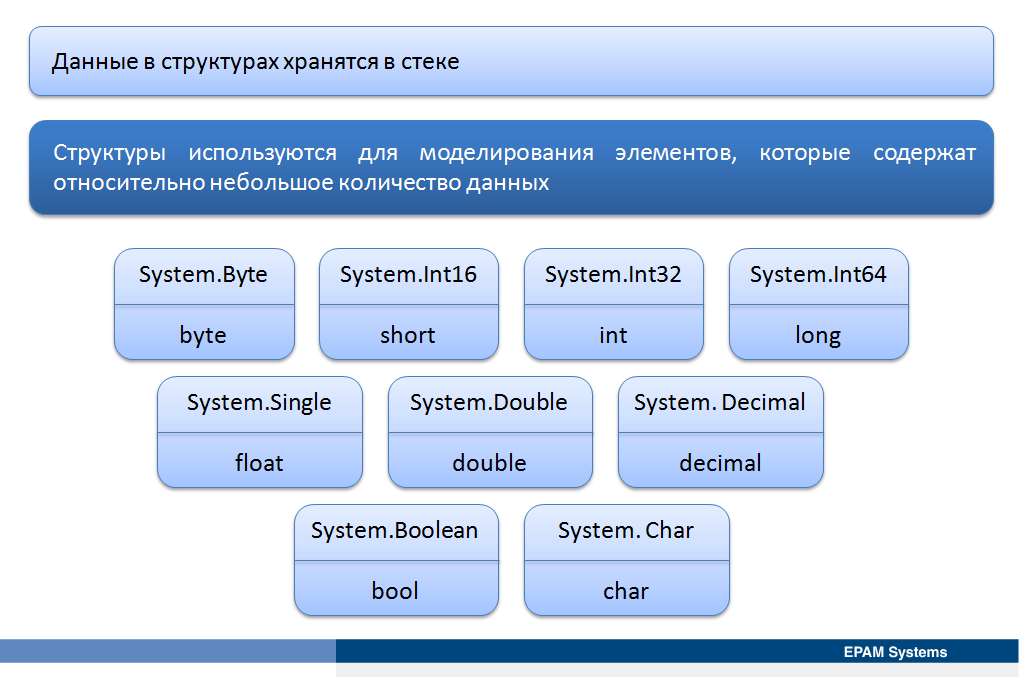
* Все разделяемые методы не могут возвращать значение. т. е. должны иметь тип void.
* Все разделяемые методы неявно private. Нельзя получить доступ к разделяемому методу вне класса, в котором он определен.
* Все объявления разделяемых методов должны начинаться с ключевого слова partial.
* Разделяемые методы могут иметь ref параметры, но не могут out параметры.

# Урок 3. Создание и использование структур



Классы очень полезны, когда в приложении необходимо моделировать сущности реального мира и инкапсулировать связанные с ними данные и бизнес-логику. Однако создание объектов приводит к возновению накладных расходов, а иногда требуется более легкое решение. Структуры имеют многие из характеристик классов, но без некоторых накладных расходов, связанных с классами, хотя их использование и связано с некорыми ограничениями. Урок описывает определение структуры, а также объясняет некоторые различия между структурами и классами.

## Что такое структура?



Структура во многих отношениях очень похожа на класс, кроме того, что она уменьшает накладные расходы из-за способа, которым CLR создает и управляет экземплярами структуры. Однако, структуры имеют некоторые ограничения. Как правило, структуры используются для моделирования элементов, которые содержат относительно небольшое количество данных. Многие из примитивных типов в языке C# только псевдонимы для некоторых структур, определяемых .NET Framework, и можно взаимозаменяемо использовать эти псевдонимы или соответствующие структуры. В следующей таблице описаны некоторые из этих структур.

|  |  |
| --- | --- |
| **Тип структуры** | **Ключевое слово в С#** |
| System.Byte | byte |
| System.Int16 | short |
| System.Int32 | int |
| System.Int64 | long |
| System.Single | float |
| System.Double | double |
| System.Decimal | decimal |
| System.Boolean | bool |
| System.Char | char |

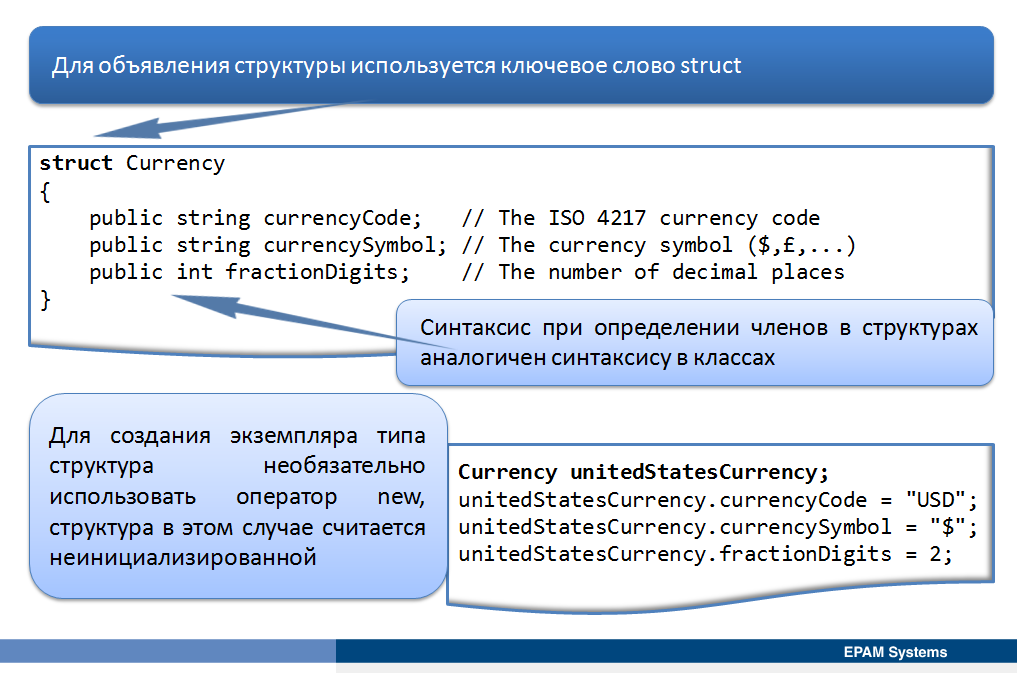
Как и класс, структура может содержать поля и методы реализации. Например, структура System.Int32 определяет метод ToString, возвращающий строковое представление целого значения.

int x = 99;

string xAsString = x.ToString();

Следует отметить, что для структурных типов нельзя использовать по умолчанию многие из общих операций, таких как == и !=, если для них не предоставлены определения этих операций. Типы, перечисленные в предыдущей таблице, предоставляют свои собственные реализации этих операций.

## Определение и использование структуры



Синтаксис, используемый при объявлении структуры, похож на синтаксис, который используется для объявления класса, за исключением использования ключевого слова struct вместо ключевого слова class. Синтаксис, который можно использовать для определения членов в структурах также очень похож на то, каким образом определяются члены в классах. Основным отличием является то, что в структуре при определении поля экземпляра ему нельзя присвоить инициализирующие значение (кроме полей static).

Следующий пример представояет структуру Currency, которая может быть использована для представления денежного выражения.

struct Currency

{

public string currencyCode; // The ISO 4217 currency code

public string currencySymbol; // The currency symbol ($,£,...)

public int fractionDigits; // The number of decimal places

}

CLR управляет структурами отличным от классов способом. При объявлении переменной структуры память для этой переменной выделяется автоматически. Следовательно, для создания экземпляра типа структура необязательно использовать оператор new, достаточно просто объявить переменную этого типа. Затем можно присвоить конкретные значения полям структуры с помощью той же точечной нотации, которая использовалась для ссылки на члены класса. Прочитать значения полей можно таким же образом. В следующем примере показано создание и использование экземпляра типа Currency.

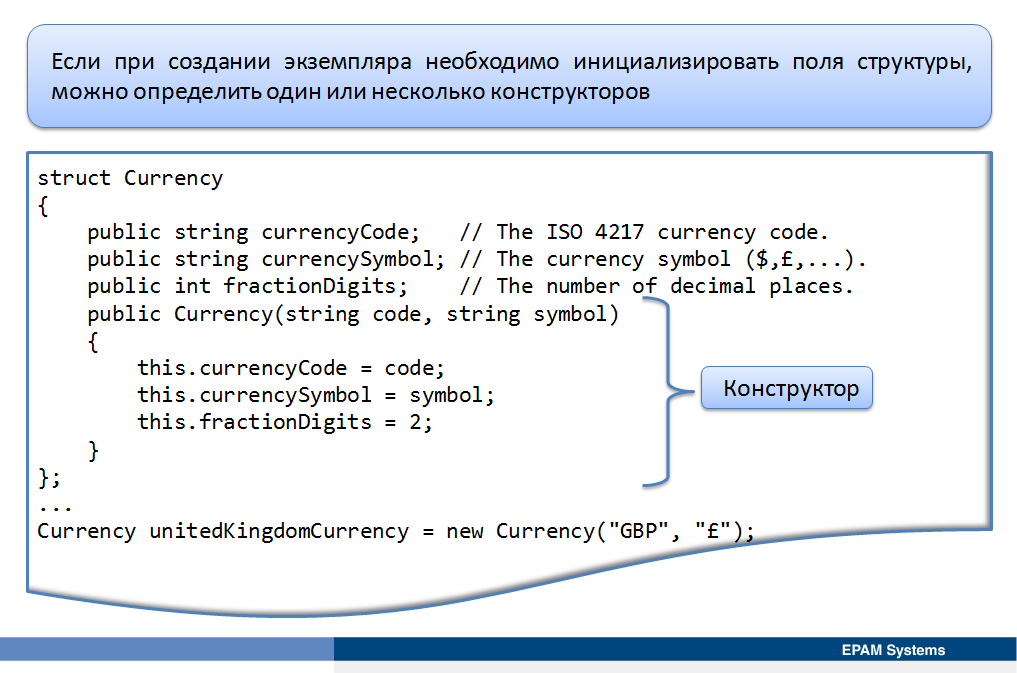
Currency unitedStatesCurrency;

unitedStatesCurrency.currencyCode = "USD";

unitedStatesCurrency.currencySymbol = "$";

unitedStatesCurrency.fractionDigits = 2;

## Инициализация структуры



При создании объекта класса оператор new используется для выделения памяти для соответствующего объекта и вызова конструктора. Для создания экземпляра структуры использование оператора new не является необходимым, поскольку память выделяется автоматически при ее объявлении. Однако, если при создании экземпляра необходимо инициализировать поля структуры, можно определить один или несколько конструкторов.

Конструкторы структур и классов синтаксически очень похожи, однако, существуют некоторые семантические различия, которые заключаются в следующем:

* Для структуры нельзя определить конструктор по умолчанию (без параметров). В отличие от класса, компилятор всегда генерирует свой собственный конструктор по умолчанию для структуры, независимо от того, определены ли любые другие конструкторы.
* Все конструкторы структуры должны явно инициализацировать каждое поле в структуре.
* Кроме того, конструктор в структуре не может вызывать другие методы до присваивания значений всем ее полям.

В следующем примере показана структура Currency, в которой определен конструктор, принимающий два параметра, определяющие код валюты и символ для ее использования. Переменная unitedKingdomCurrency инициализируется с помощью этого конструктора.

struct Currency

{

public string currencyCode; // The ISO 4217 currency code.

public string currencySymbol; // The currency symbol ($,£,...).

public int fractionDigits; // The number of decimal places.

public Currency(string code, string symbol)

{

this.currencyCode = code;

this.currencySymbol = symbol;

this.fractionDigits = 2;

}

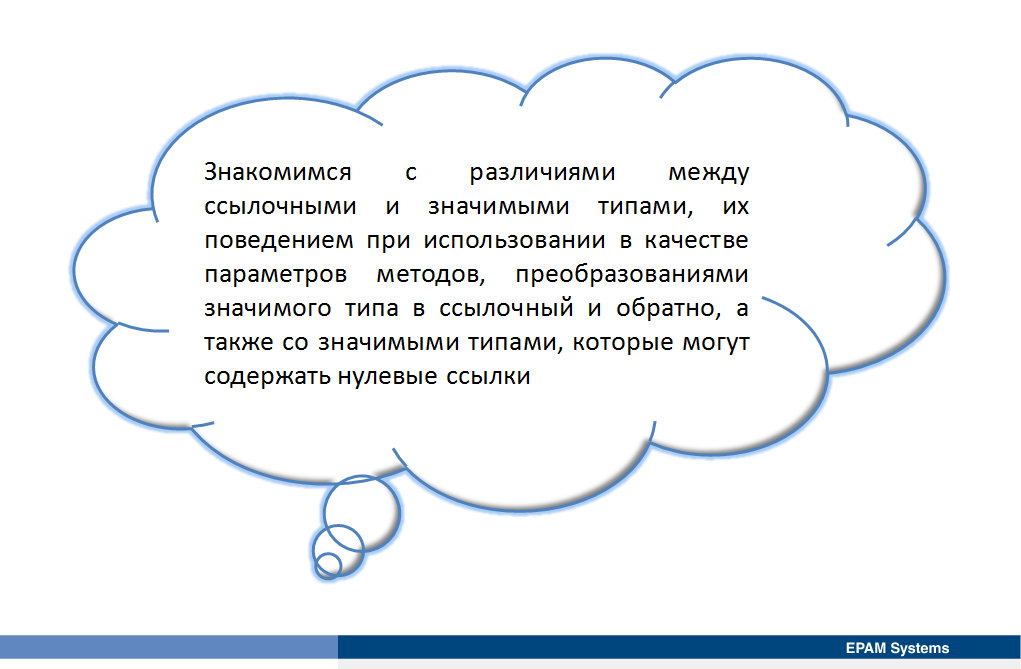
};

...

Currency unitedKingdomCurrency = new Currency("GBP", "£");

Если при создании экземпляра структуры не используется конструктор, структура считается неинициализированной. При этом можно читать и писать в отдельные поля неинициализированной структуры, однако ее нельзя использовать в качестве аргумента метода или присваивать другой переменной, пока явно не будет присвоено значение каждому полю в этой структуре. Простейший способ гарантировать, что структура полностью инициализирована, это всегда использовать конструктор. Компилятор генерирует конструктор по умолчанию автоматически, так что необязательно писать конструктор, если просто нужно, чтобы структура была заполнена значениями по умолчанию.

# Урок 4. Сравнение ссылочных и значимых типов

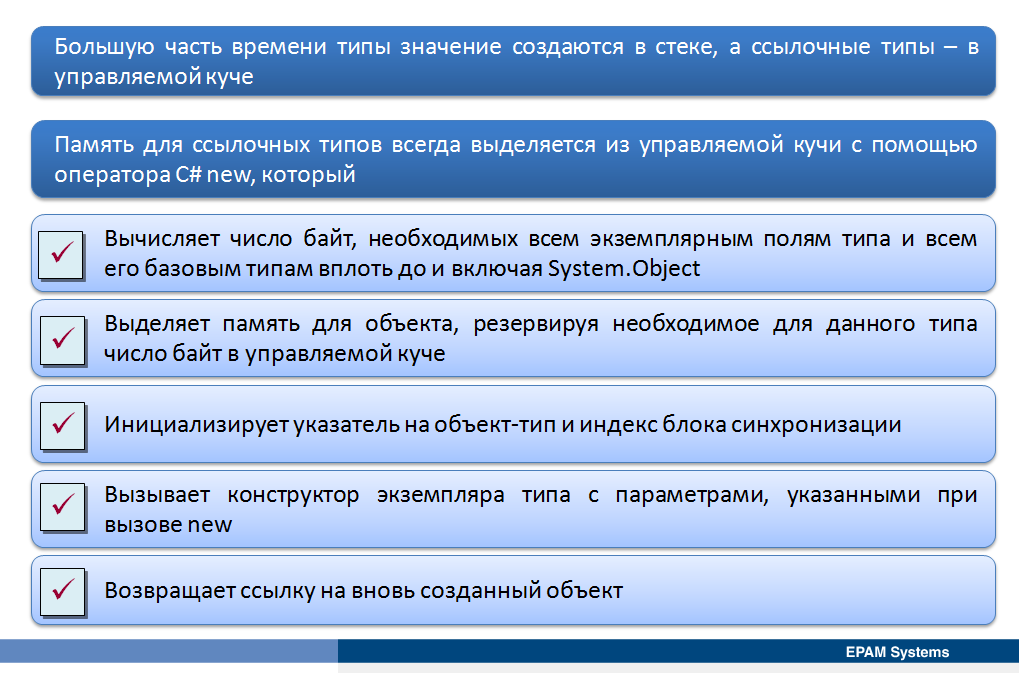


Структурные типы, такие как пользовательские структуры или примитивные типы, которые использует C#, также называются значимыми типами (value types) или типами значения. При объявлении переменной структурного типа, компилятор генерирует код, выделяющий блок памяти для хранения соответствующего значения. Например, объявление переменной типа int (System.Int32) заставляет компилятор выделить 4 байта памяти (32 бита). Оператор, присваивающий значение переменной типа int, распорядится этим блоком памяти, чтобы сохранить в него указанные в операторе данные.

Типы классы, такие как класс Residence, который был определен ранее, называются ссылочными типами (reference types). CLR обрабатывает эти типы по-другому. При объявлении переменной Residence, компилятор не генерирует код, выделящий блок памяти для сохранения в нем объекта Residence. Все, что компилятор делает, это выделяет небольшой кусок памяти, который потенциально может содержать адрес другого блока памяти (или ссылку на другой блок памяти), содержащего объекта Residence. Компилятор инициализирует эту ссылку значением null, чтобы указать, что объект еще не был инициализирован. Память для объекта Residence выделяется только при использовании ключевого слова new для вызова конструктора и создания объекта.

Урок описывает различия между ссылочным и значимыми типами и объясняет, как их поведение отличается при использовании их в качестве параметров методов. Урок также описывает, как преобразовать значимый тип в ссылочный и обратно, используя упаковку и распаковку, а также как создать значимые типы, которые могут содержать нулевые (null) ссылки.

## Сравнение ссылочных и значимых типов



CLR делит свою память на две основные области: стек и кучу. Большую часть времени типы значение создаются в стеке, а ссылочные типы – в управляемой куче. Память для ссылочных типов всегда выделяется из управляемой кучи с помощью оператора С# new, который выполняет следующие действия.

* Вычисляет число байт, необходимых всем экземплярным полям типа и всем его базовым типам вплоть до и включая System.Object (в котором отсутствуют собственные экземплярные поля). Каждый объект кучи имеет дополнительные члены: указатель на объект-тип (Type object pointer) и индекс блока синхронизации (Sync block index), используемые CLR для управления объектом. Байты этих дополнительных членов добавляются к байтам, необходимым для размещения самого объекта.
* Выделяет память для объекта, резервируя необходимое для данного типа число байт в управляемой куче байтов.
* Инициализирует указатель на объект-тип и индекс блока синхронизации.
* Вызывает конструктор экземпляра типа с параметрами, указанными при вызове new. Большинство компиляторов автоматически создает в конструкторе код вызова конструктора базового класса. Каждый конструктор выполняет инициализацию определенных в соответствующем типе полей. В частности, вызывается конструктор System.Object, который, ничего не делает и просто возвращает управление.
* Возвращает ссылку на вновь созданный объект.

Таким образом при работе с ссылочными типами следует иметь в виду следующие аспекты, относящиеся к производительности приложения:

* память для ссылочных типов всегда выделяется из управляемой кучи;
* каждый объект, размещаемый в куче, имеет некоторые дополнительные члены, подлежащие инициализации;
* незанятые полезной информацией байты объекта обнуляются (касается полей);
* размещение объекта в управляемой куче со временем инициирует сборку мусора.

А, следовательно, если бы все типы были ссылочными, эффективность приложения резко упала бы. Чтобы ускорить обработку простых, часто используемых типов, CLR предлагает «облегченные» значимые типы. Экземпляры этих типов обычно размещаются в стеке потока (хотя они могут быть встроены и в объект ссылочного типа). В представляющей экземпляр переменной нет указателя на экземпляр; поля экземпляра размещаются в самой переменной. Поскольку переменная содержит поля экземпляра, то для работы с экземпляром не нужно выполнять разыменовывание (dereference) экземпляра. Благодаря тому, что экземпляры значимых типов не обрабатываются сборщиком мусора, уменьшается интенсивность работы с управляемой кучей.

В документации по .NET Framework можно сразу увидеть, какие типы относятся к ссылочным, а какие – к значимым. Если тип называют классом (class), речь идет о ссылочном типе. Так, классы System.Object, SystemException, System.lO.FileStream и System.Random – это ссылочные типы. В свою очередь значимые типы в документации называют структурами (structure) и перечислениями (enumeration). Например, структуры System.Int32, System.Boolean, System.Decimal, System.TimeSpan и перечисления System.DayOfWeek, System.lO.FileAttributes и System.Drawing.FontStyle являются значимыми типами. Все структуры являются прямыми потомками абстрактного типа System.ValueType, который в свою очередь является производным от типа System.Object. По умолчанию все значимые типы должны быть производными от System.ValueType. Все перечисления являются производными от типа System.Enum, производного от System.ValueType (Рис. 4).

Рис. 4.

При определении собственного значимого типа нельзя выбрать произвольный базовый тип, однако значимый тип может реализовать один или несколько выбранных интерфейсов. Кроме того, в CLR значимый тип является изолированным, то есть не может служить базовым типом для какого-либо другого ссылочного или значимого типа. Поэтому, например, нельзя в описании нового типа указать в качестве базовых типы Boolean, Char, Int32, Uint64, Single, Double, Decimal и т. д.

Основное различие между значимыми и ссылочными типами просматривается при их копировании. В следующих примерах тип SomeRef это класс (reference type), а тип SomeVal это структура (value type).

// Reference type (because of 'class')

private class SomeRef

{

public int x;

}

// Value type (because of 'struct')

private struct SomeVal

{

public int x;

}

Ниже показано различие между ссылочными и значимыми типами (Рис. 5-6) при выполнении следующего кода:

public static void Do()

{

SomeRef r1 = new SomeRef(); // Allocated in heap

SomeVal v1 = new SomeVal(); // Allocated on stack

r1.x = 5; // Pointer dereference

v1.x = 5; // Changed on stack

Console.WriteLine(r1.x); // Displays "5"

Console.WriteLine(v1.x); // Also displays "5"

//==================================================

SomeRef r2 = r1; // Copies reference (pointer) only

SomeVal v2 = v1; // Allocate on stack & copies members

r1.x = 8; // Changes r1.x and r2.x

v1.x = 9; // Changes v1.x, not v2.x

Console.WriteLine(r1.x); // Displays "8"

Console.WriteLine(r2.x); // Displays "8"

Console.WriteLine(v1.x); // Displays "9"

Console.WriteLine(v2.x); // Displays "5"

}

**. . .**

**Стек потока**

**Управляемая куча**

**r1**

**v1**

**Instance fields**

**SomeRef Object**

**Type object ptr**

**Sync block index**

**x = 5**

**x = 5**

Рис. 5.

**. . .**

**Стек потока**

**Управляемая куча**

**r1**

**v1**

**Instance fields**

**SomeRef Object**

**Type object ptr**

**Sync block index**

**x = 9**

**x = 8**

**r2**

**v2**

**x = 5**

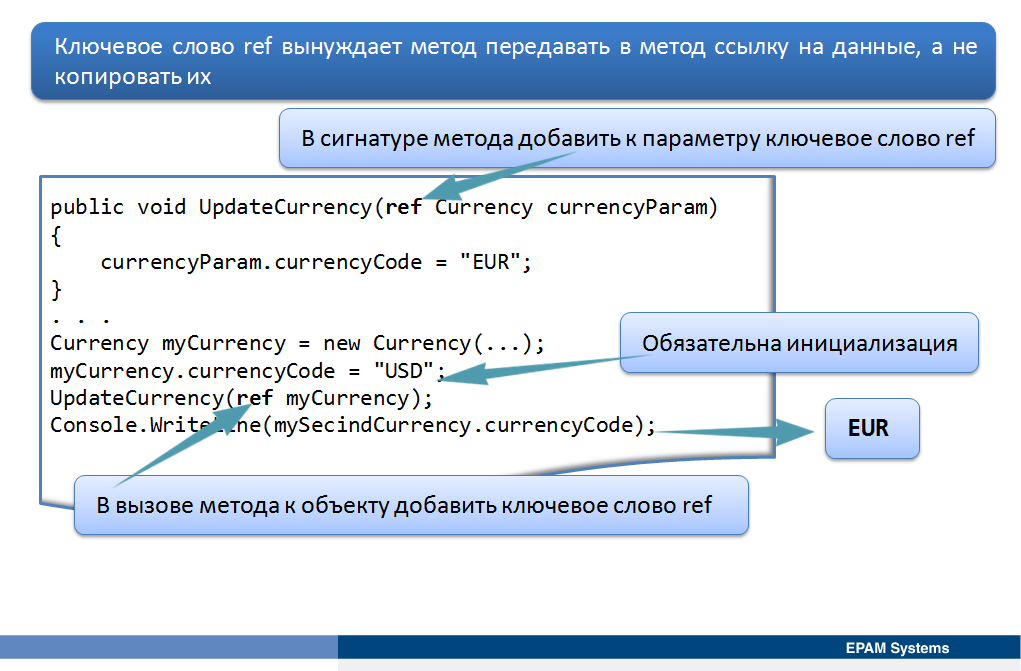
Рис. 6.

При присваивании ссылки просто устанавливается ссылка на объект в памяти. Если одну и ту же ссылку присвоить двум различным переменным, обе переменные будут ссылатся на один и тот же объект.

В примере переменная r1 ссылается на новый объект SomeRef. Переменная r2 ссылается на тот же объект. При изменении данных, на которые ссылается переменная r1, изменяется и тот объект, на который ссылается переменная r2 и наоборот. При присваивании переменной v2 переменной v1, CLR создает копию данных и присваивает ее переменной v2. Две переменные не относятся к одним и тем же данным в памяти, таким образом можно изменить значения в переменной v1, при этом информации в переменной v2 не изменится.

Перечисления являются значимыми типами и ведут себя при копировании аналогично структурам.

## Передача значимого типа в метод по ссылке



Фундаментальное различие в поведении между значимыми и ссылочными типами имеет значительное влияние на то, что происходит при передаче в метод параметров этих типов. В следующем примере показан метод UpdateCurrency, принимающий параметр Currency (value type) и изменяет его поле currencyCode.

public void UpdateCurrency(Currency currencyParam)

{

currencyParam.currencyCode = "EUR";

}

...

Currency myCurrency = new Currency(...);

myCurrency.currencyCode = "USD";

UpdateCurrency(myCurrency);

Console.WriteLine(mySecindCurrency.currencyCode);

Код создает переменную myCurrency типа Currency и присваивает полю currencyCode этой переменной значение "USD" перед вызовом метода UpdateCurrency. При вызове метода вычисляется выражение myCurrency и значение этого выражения передается в качестве параметра метода UpdateCurrency. Следует отметить, что это значение копирует данные в переменную myCurrency. Следовательно, метод UpdateCurrency изменяет данные только в этой копии. При завершении работы метода, эта копия будет потеряна. Значение в переменной myCurrency остается неизменным, таким образом, оператор Console.WriteLine выведет на экран строку "USD".

В следующем примере методу передается параметр Residence (reference type).

public void UpdateResidence(Residence residenceParam)

{

residenceParam.numberOfBedrooms = 3;

}

...

Residence myResidence = new Residence(ResidenceType.House,2);

UpdateResidence(myResidence);

Console.WriteLine(myResidence.numberOfBedrooms);

В этом случае при вызове метода UpdateResidence вычисляется выражение myResidence и становится ссылкой на объект Residence. Эта ссылка передается в качестве параметра методу UpdateResidence. Как параметр residenceParam, так и переменная myResidence, оба ссылаются на один и тот объект Residence в управляемой куче. Следовательно, когда код в методе UpdateResidence изменяет поле параметра numberOfBedrooms, он обновляет тот объект, на который ссылается переменная myResidence. Когда метод закончит выполнение, оператор Console.WriteLine выведет значение 3.

Для того, чтобы передавать значение параметра в метод по ссылке, нужно использовать ключевое слово ref. Ключевое слово ref вынуждает метод передавать в метод ссылку на данные, а не копировать их. Это значит, любые изменения этого параметра в методе выполняются над объектом и останутся, когда метод завершится. Чтобы использовать ключевое слово ref, необходимо выполнить следующие действия:

* В сигнатуре метода добавить к параметру ключевое слово ref.
* В вызове метода к объекту добавить ключевое слово ref.

В следующем примере показано использование ключевого слова ref с переменной значимого типа Currency.

public void UpdateCurrency(ref Currency currencyParam)

{

currencyParam.currencyCode = "EUR";

}

Currency myCurrency = new Currency(...);

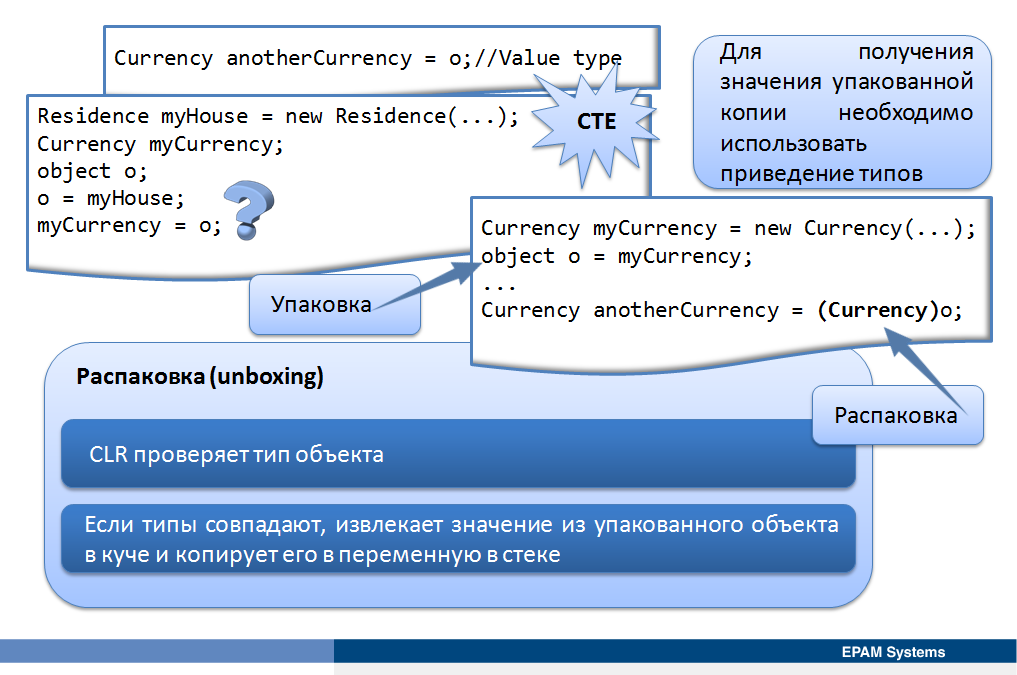
myCurrency.currencyCode = "USD";

UpdateCurrency(ref myCurrency);

Console.WriteLine(mySecindCurrency.currencyCode);

На этот раз метод UpdateCurrency принимает ссылку на переменную Currency. Аргумент, который передается в метод это ссылка на переменную myCurrency. В методе параметр currencyParam ссылается на переменную myCurrency (не копирует), и любые изменения, сделанные через эту ссылку, изменяют данные в переменной myCurrency. При завершении метода оператор Console.WriteLine выведет на экран значение "EUR".

## Упаковка и распаковка



Ссылочные типы ссылаются на объекты, а значимые типы хранят значения. Язык C# определяет особый тип, называемый object, который можно использовать для обозначения любого типа.

Residence myHouse = new Residence(...);

object obj = myHouse;

Тип object полезен, когда необходимо определить методы, принимающие параметры различных типов, и заранее неизвестно, какие это типы. Например, классы коллекций в библиотеке классов .NET Framework позволяют создавать коллекции объектов практически любого типа, и методы, которые эти классы определяют, используют тип object.

Тип object псевдоним класса System.Object. Этот класс лежит в основе всей системы типов, которую реализует .NET Framework; все типы данных в действительности являются только специализированными версиями типа object (Рис. 7).

Рис. 7.

В некоторых случаях может понадобиться преобразовывать значимый тип в ссылочный, такой как object. Добиться этого довольно просто следующим образом.

Currency myCurrency = new Currency(...);//Value type

object o = myCurrency; //Box the value type into a reference

Второй оператор требует пояснений. Следует напомнить, что myCurrency переменная значимого типа, создаваемая в стеке. Если переменная o ссылается непосредственно на переменную myCurrency, ссылка хотела бы сослаться на стек. Однако все ссылки должны относиться к объектам в куче, создание ссылок на элементы в стеке может создать серьезную угрозу надежности и безопасности CLR и поэтому не допускается. Таким образом, CLR выделяет кусок памяти в куче, копирует значение переменной myCurrency в эту часть памяти, а затем связывает объект o с этой копией. Это автоматическое копирование элемента из стека в кучу называется упаковкой (boxing).

Поскольку переменная типа object может ссылаться на упакованную копию значения, можно попытаться получить доступ к этому упакованному значению через переменную. Например, можно рассчитывать на доступ к значению упакованной переменной Currency, на которую ссылается переменная o, используя простой оператор присваивания.

Currency anotherCurrency = o;//Value type

Однако при использовании данного синтаксиса возникнет ошибка времени компиляции, поскольку переменная o может ссылаться на какой угодно тип, а не только на значение переменной Currency.

Residence myHouse = new Residence(...);

Currency myCurrency;

object o;

o = myHouse;// o refers to a Residence

myCurrency = o; // what is stored in myCurrency?

Чтобы получить значение упакованной копии, необходимо использовать приведение типов. Приведение заставляет компилятор генерировать код, который проверяет, является ли безопасным преобразование объекта в переменную указанного типа. В следующем примере кода показано, как для данного примера использовать приведение.

Currency myCurrency = new Currency(...);

object o = myCurrency;//boxing

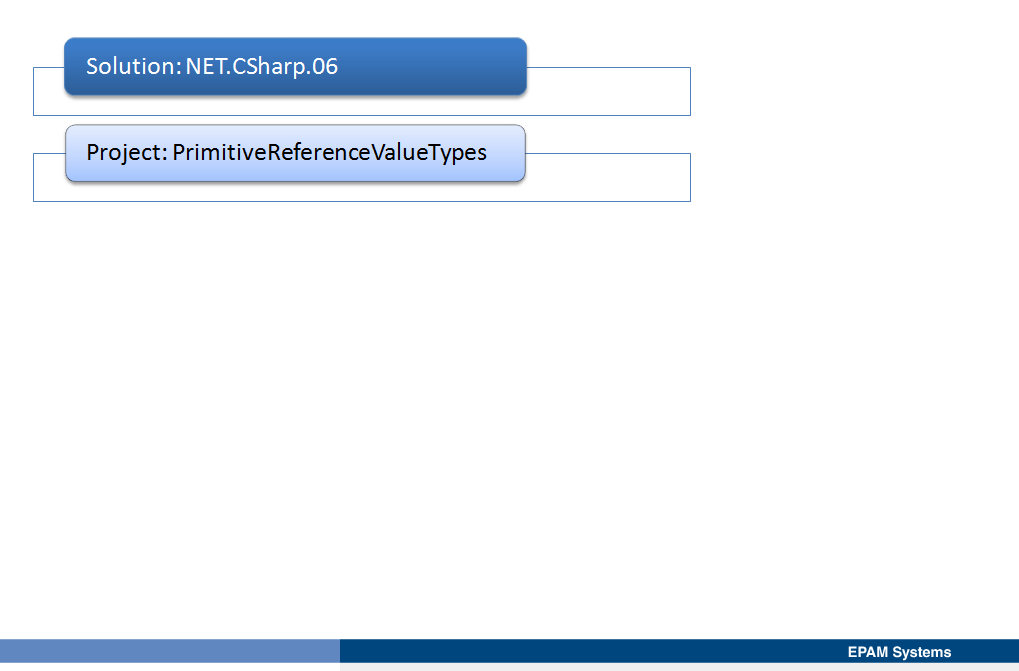
...

Currency anotherCurrency = (Currency)o;//compiles okay

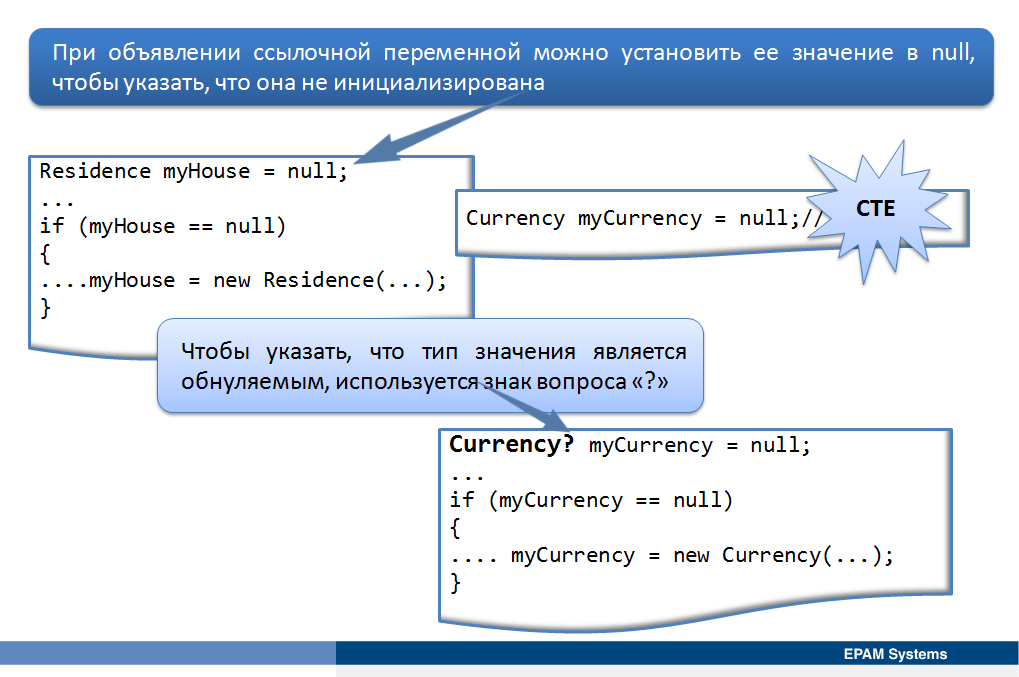
Если генерируемый компилятором код проверяет тип объекта и устанавливает, что переменная o ссылается на значение переменной Currency, этот оператор извлекает значение из упакованного объекта Currency в куче и копирует его в переменную anotherCurrency, которая хранится в стеке. Этот процесс называется распаковкой (unboxing). Однако, если o не ссылаться на упакованный объект Currency, существует несоответствие типов, что является причиной неверного приведения типов, и генерируемый компилятором код генерирует исключение времени выполнения InvalidCastException.

Упаковка и распаковка происходит только при преобразовании из значимого типа в ссылочный (например, объект) и обратно. При преобразовании из одного ссылочного типа в другой копии не создаются, и все, что происходит, это создание новой ссылки для существующего в куче объекта

## Демонстрация: Упаковка и распаковка



## Обнуляемые типы



При создании ссылочной переменной она изначально неинициализирована. Нельзя использовать ссылочную переменную, пока ей не присвоено значение. Возможно, при объявлении ссылочной переменной еще неизвестно, как ее инициализировать. В этом случае, можно установить ее значение в null, чтобы указать, что она не инициализирована. Значение null полезно, потому что можно в любой момент в коде явно проверить значение ссылочной переменной на null, и, если нужно, инициализировать ее с помощью оператора new.

Residence myHouse = null;

...

if (myHouse == null)

{

....myHouse = new Residence(...);

}

Само значение null является ссылой. Для типов значения соответствующего значения не существует, что может вызвать проблемы в коде. Например, может быть сложно определить инициализирована ли переменная значимого типа (если попытаться передать неинициализированные переменные типа значения в метод, код не будет компилироваться). В силу того, что значение null является ссылкой, следующий оператор является недействительным.

Currency myCurrency = null;//Illegal!

Однако C# определяет модификатор, который можно использовать для декларирования того, что переменная является обнуляемым типом значения. Обнуляемый тип значения ведет себя аналогично исходному типу значения, но ему можно присвоить значение null. Чтобы указать, что тип значения является обнуляемым, используется знак вопроса «?». Позднее в приложении можно убедиться в том, что обнуляемая переменная содержит null, проверив ее так же, как переменную ссылочного типа.

Currency? myCurrency = null; //Legal!

...

if (myCurrency == null)

{

.... myCurrency = new Currency(...);

}

Типы, допускающие значения null, по сути являются экземплярами структуры System.Nullable<T>. Тип, допускающий значения null, может представлять правильный диапазон значений для своего базового типа значений и дополнительное пустое значение null. Например, для Nullable<Int32>, называемого «тип Int32, допускающий значения null», можно присваивать любое значение от -2 147 483 648 до 2 147 483 647 или значение null. Переменным типа Nullable<bool> можно присвоить значения true, false или null. Возможность присваивания значения null для числовых и логических типов особенно полезна при работе с базами данных и другими типами данных, содержащими элементы, значение которых может быть не определено. Например, логическое поле в базе данных может хранить значения true или false или может быть не задано (null).

Можно связать выражение соответствующего типа значения с непосредственно обнуляемой переменной. Следующие примеры кода все являются действительными.

int? i = null;

int j = 99;

i = 100;//Copy a value-type constant to a nullable type

i = j;//Copy a value-type variable to a nullable type

Стоит отметить, что обратное утверждение неверно. Нельзя присвоить обнуляемое значение обычной переменной значимого типа. Учитывая определения переменных i и j из предыдущего примера, следующий оператор не допустим.

j = i;//Illegal!

Обнуляемые типы представляют пару свойств, которые можно использовать для определения имеет ли обнуляемая переменная значение null:

* Свойство HasValue указывает, содержит ли обнуляемый тип значение или null. Если свойство true, обнуляемая переменная имеет значение, если оно false, обнуляемая переменная содержит null.
* Свойство Value[[3]](#footnote-3) содержит значение переменной. Можно получить значение обнуляемой, когда свойство HasValue равно true, в противном случае код сгенерирует исключение.

В следующем примере кода показано использование свойств обнуляемой переменной Currency.

Currency? myCurrency = null;

...

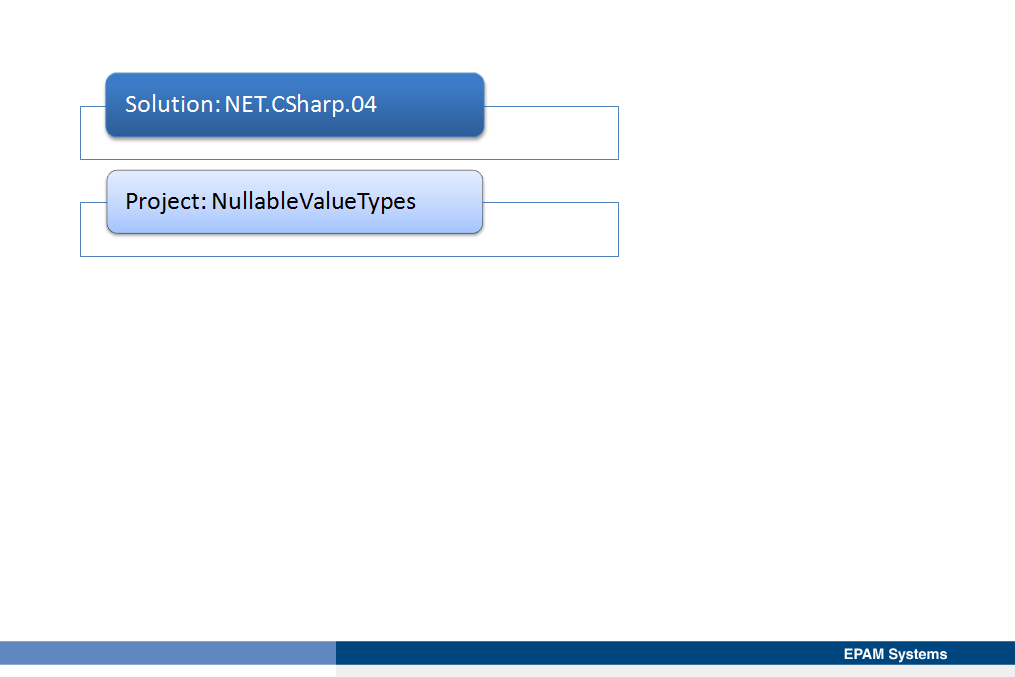
if (myCurrency.HasValue)

{

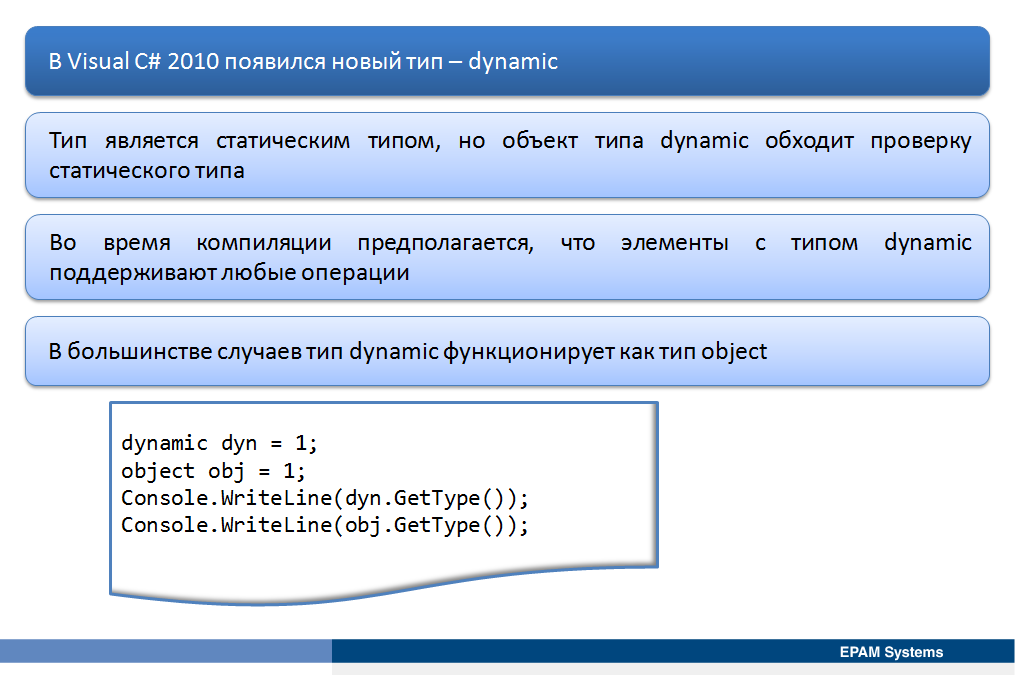
Console.WriteLine(myCurrency.Value);

}

## Демонстрация: Обнуляемые типы



## Тип dynamic



В Visual C# 2010 появился новый тип – dynamic. Тип является статическим типом, но объект типа dynamic обходит проверку статического типа. В большинстве случаев он функционирует, как тип object. Во время компиляции предполагается, что элементы с типом dynamic поддерживают любые операции. Поэтому разработчику не нужно следить за тем, откуда объект получает свое значение – из интерфейса API COM, из динамического языка, такого как IronPython, из объектной модели DOM HTML, из отражения или из любого другого места программы. Но если код является недействительным, во время выполнения перехватываются ошибки.

В большинстве случаев тип dynamic ведет себя так же, как object. Однако операции, которые содержат выражения типа dynamic, не разрешаются или выполняется проверка типа компилятором. Компилятор пакует сведения об операции, затем эти сведения используются для оценки операции во время выполнения. Как часть процесса, переменные типа dynamic компилируются в переменные типа object. Поэтому тип dynamic существует только во время компиляции, но не во время выполнения.

В следующем примере переменная типа dynamic противопоставляется переменной типа object. Чтобы проверить тип каждой переменной во время компиляции, нужно навести указатель мыши на переменные dyn или obj в операторах WriteLine. IntelliSense отображает dynamic для dyn и object для obj.

dynamic dyn = 1;

object obj = 1;

// Rest the mouse pointer over dyn and obj

//to see their types at compile time.

Console.WriteLine(dyn.GetType());

Console.WriteLine(obj.GetType());

Операторы WriteLine отображают типы времени выполнения dyn и obj. На этом этапе оба имеют одинаковый тип «целое число» (System.Int32).

В следующем примере, если у метода exampleMethod экземпляра в следующем коде имеется только один параметр, компилятор определяет, что первый вызов метода (ec.exampleMethod1(10, 4)) является недопустимым, поскольку он содержит два аргумента. Такой вызов приводит к ошибке компилятора. Второй вызов метода dynamic\_ec.exampleMethod1(10, 4) не проверяется компилятором, поскольку тип dynamic\_ec является типом dynamic. Поэтому ошибка компилятора не возникает.Однако ошибка не остается незамеченной и не приводит к неопределенному результату. Она перехватывается во время выполнения и вызывает исключение времени выполнения.

static void Main(string[] args)

{

ExampleClass ec = new ExampleClass ();

// The following line causes a compiler error if exampleMethod1

// has only one parameter.

//ec.exampleMethod1(10, 4);

dinamic dynamic\_ec = new ExampleClass ();

// The following line is not identified as an error by the

// compiler, but it causes a run-time exception.

dynamic\_ec.exampleMethod1(10, 4);

// The following calls also do not cause compiler errors, whether

// appropriate methods exist or not.

dynamic\_ec.someMethod("some argument", 7, null);

dynamic\_ec.nonexistentMethod();

}

class ExampleClass

{

public ExampleClass(){ }

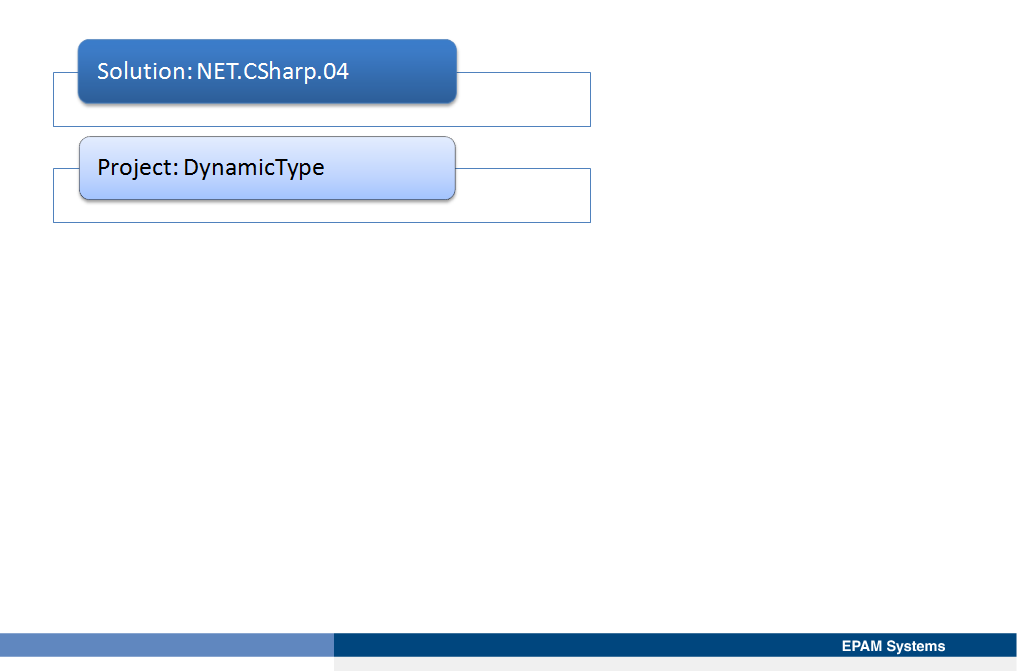
public ExampleClass(int v){ }

public void exampleMethod1(int i){ }

public void exampleMethod2(string str){ }

}

## Демонстрация: Тип dynamic



1. Пространство имен по умолчанию для класса определяется приложением, к которому добавляется класс. [↑](#footnote-ref-1)
2. Поля и методы, описанные в этом разделе, являются полями и методами экземпляра. Два объекта одного класса имеют свою собственную копию экземпляра поля. Однако, можно разделить поле между экземплярами путем создания статических (static) членов. [↑](#footnote-ref-2)
3. Свойство Value является свойством только для чтения. Можно использовать это свойство для чтения значения переменной, но не для его изменения. Для обновления значения обнуляемой переменной, используется обычный оператор присваивания. [↑](#footnote-ref-3)