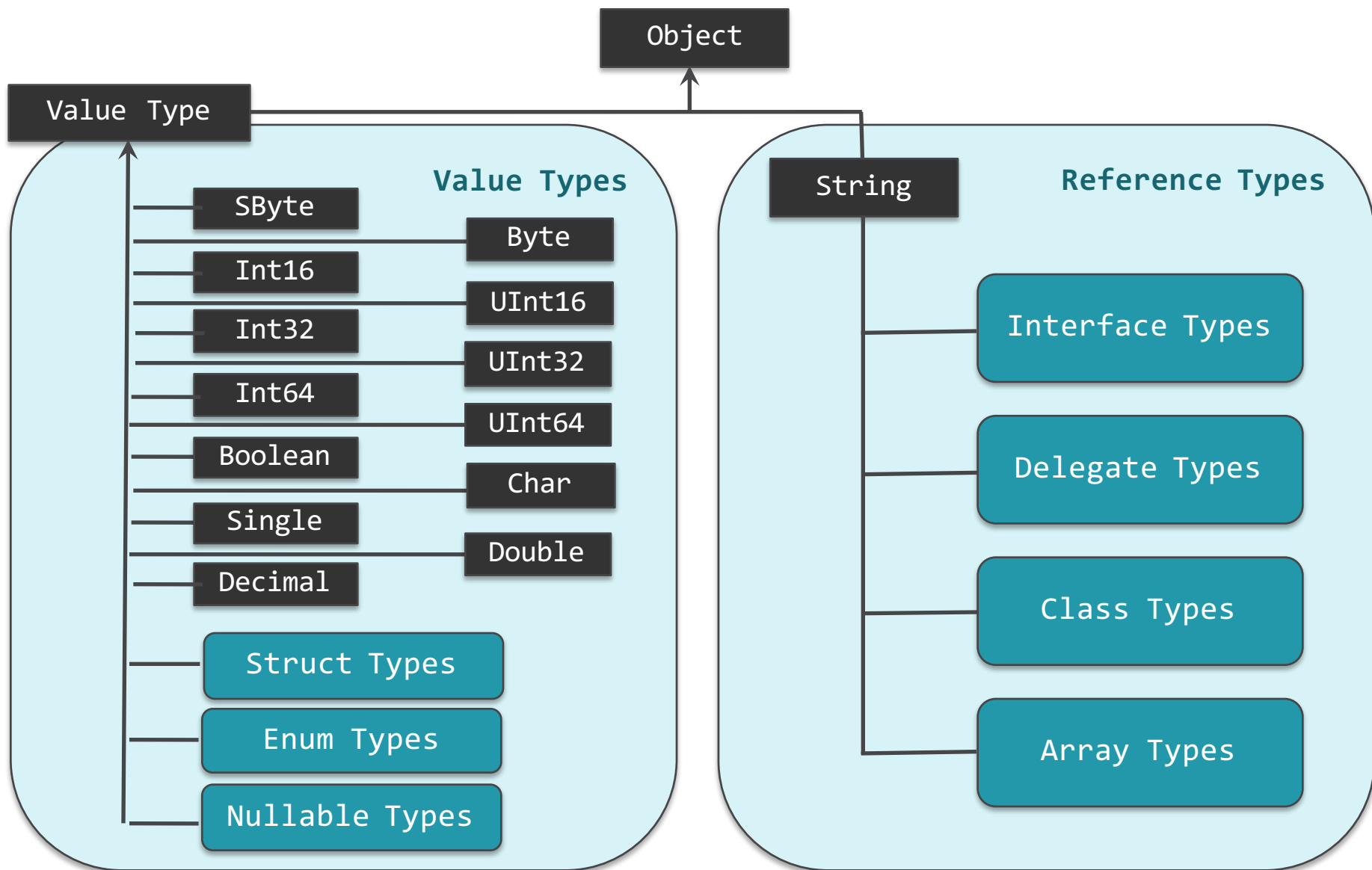




# КЛАССЫ, СТРУКТУРЫ И ПЕРЕЧИСЛЕНИЯ В C#

**.NET & JS LAB**

# Классификация типов



# Что такое класс?

**Класс** – это способ описания сущности, определяющий состояние и поведение, зависящее от этого состояния, а также правила для взаимодействия с данной сущностью (контракт)

```
class House
{
    ...
}
```

oneHouse

twoHouse

Класс определяется с ключевым словом **class**

**Объект (экземпляр класса)** – это отдельный представитель класса, имеющий конкретное состояние и поведение, полностью определяемое классом

С точки зрения программирования класс можно рассматривать как набор данных и функций для работы с ними

С точки зрения структуры программы, класс является сложным типом данных

## Класс, объект, ссылка

Объект – это понятие времени выполнения, любой объект является экземпляром класса, создается во время выполнения системы и представляет набор полей

Ссылка - это понятие времени выполнения. Значение ссылки либо null, либо она присоединена к объекту, который она однозначно идентифицирует

Сущность - это статическое понятие (времени компиляции), применяемое к программному тексту, идентификатор в тексте класса, представляющий значение или множество значений в период выполнения. Сущностями являются обычные переменные, именованные константы, аргументы и результаты функций

Определение ссылки не привязано к аппаратно-программной реализации – присоединенная к объекту она может рассматриваться как его абстрактное имя. Отличие ссылки от указателя в ее строгой типизации

Ссылка в действительности реализована в виде небольшой порции данных, которая содержит информацию, используемую CLR, чтобы точно определить объект, на который ссылается ссылка

# Что такое класс?

[Атрибуты класса]

[Модификаторы класса] **class** **ClassName** [Параметры обобщенных типов,  
базовый класс, интерфейсы]

{

Члены класса – методы, свойства, индексы, события, поля,  
конструкторы, перегруженные операторы,  
сложные типы, финализатор

}

*public, internal, abstract, sealed, static, unsafe, partial*

## Члены класса

В класс могут добавляться поля и методы, определяющие состояние и поведение класса соответственно

О поле можно думать как о переменной, которая имеет область видимости класс

Статический модификатор	static
Модификатор доступа	public internal private protected
Модификатор наследования	new
Модификатор небезопасного кода	unsafe
Модификатор доступа только для чтения	readonly
Модификатор многопоточности	volatile

# Члены класса

Метод это процедура или функция, определенная внутри класса

Статический модификатор	static
Модификатор доступа	public internal private protected
Модификатор наследования	new virtual abstract override sealed
Модификатор неуправляемого кода	unsafe extern

# Добавление элементов в классы

```
public class Residence  
{
```

```
    private ResidenceType type;  
    private int numberOfBedrooms;  
    private bool hasGarage;  
    private bool hasGarden;
```

Поля

```
    public int CalculateSalePrice()  
    {
```

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

```
    public int CalculateRebuildingCost()  
    {
```

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

```
}
```

```
public enum ResidenceType  
{
```

```
    House,  
    Flat,  
    Bungalow,  
    Apartment
```

```
};
```

Методы



# Определение конструкторов и инициализация объектов

Для обеспечения того, чтобы объект был полностью инициализирован и все его поля имели значимые значения, в классе следует определить один или несколько конструкторов

```
public class Residence
{
    public Residence(ResidenceType type, int numberOfBedrooms)
    {
    }
    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage)
    {
    }
    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage, bool hasGarden)
    {
    }
}
```

При создании объекта CLR вызывает конструктор автоматически



# Модификаторы конструктора

Статический модификатор	static
Модификатор доступа	public internal private protected
Модификатор неуправляемого кода	unsafe extern

# Определение конструкторов и инициализация объектов

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

Конструкторы, как правило, объявляются с модификатором доступа `public`, чтобы любая часть приложения имела доступ к ним для создания и инициализации объектов

Конструкторы обычно инициализируют некоторые или все поля объекта, а также могут выполнять любые дополнительные задачи инициализации, требуемые классу

# Определение конструкторов и инициализация объектов

```
public class Residence
{
    private ResidenceType type;
    private int numberOfBedrooms;
    private bool hasGarage;
    private bool hasGarden;
    private Residence(ResidenceType type, int numberOfBedrooms, bool
hasGarage, bool hasGarden)
    {
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
        this.hasGarage = hasGarage;
        this.hasGarden = hasGarden;
    }
    public Residence() : this(ResidenceType.House, 3, true, true{ }
...
}
```

Реализация конструктора по умолчанию, вызывающего параметризованный конструктор с множеством значений по умолчанию для каждого параметра

## Создание объектов

Для использования переменной класса необходимо создать экземпляр соответствующего класса и присвоить его ссылочной переменной

```
Residence flat = new Residence(ResidenceType.Flat, 2);  
  
Residence house = new Residence(ResidenceType.House, 3, true);  
  
Residence bungalow = new Residence(ResidenceType.Bungalow, 2, true, true);
```

Если при вызове new не указать параметры, сработает конструктор по умолчанию

Объект может иметь большое количество полей, и не всегда возможно или целесообразно предусматривать конструкторы, которые могут инициализировать их все возможные комбинации

## Создание объектов

- ЕЕ выделяет память под объект
- ЕЕ инициализирует указатель на таблицу методов - фактически после этого этапа объект является полноценным живым объектом
- ЕЕ закладывает указатель на объект в регистр есх и передает управление конструктору, указанному в инструкции newobj, породившей генерацию кода создания объекта
- Если во время работы конструктора не произошло необработанных исключений, то ссылка на объект помещается в ту или иную переменную области видимости, из которой вызывался код создания объектов

# Создание объектов

```
public class Employee
{
    private int id;
    private string name;
    private static CompanyPolicy policy;

    public virtual void Work()
    {
        Console.WriteLine("Zzzz...");
    }

    public void TakeVacation(int days)
    {
        Console.WriteLine("Zzzz...");
    }

    public static void SetCompanyPolicy(CompanyPolicy plc)
    {
        policy = plc;
    }
}
```

**Instance fields**

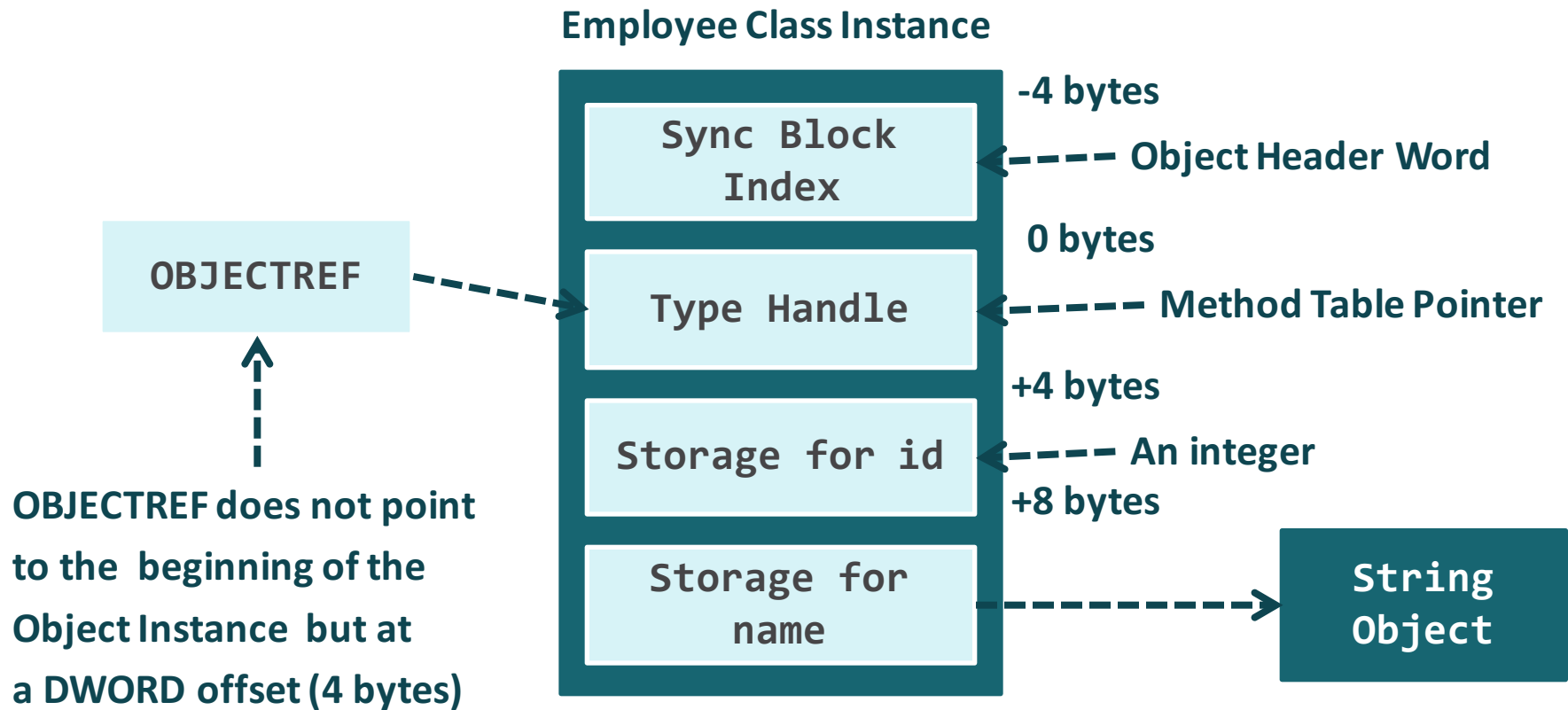
**Static field**

**Instance virtual method**

**Instance method**

**Static method**

# Создание объектов





## Доступ к членам класса

Для доступа к членам экземпляра используется имя экземпляра с последующей точкой, за которой следует имя члена класса

`InstanceName.MemberName`

При доступе к членам экземпляра применяются следующие правила:

При получении доступа к методу используется имя метода с последующими круглыми скобками

При получении доступа к `public` полю используется имя поля - таким образом можно получить значение поля или установить его новое значение

```
Residence house = new Residence(ResidenceType.House, 3);  
int salePrice = house.CalculateSalePrice();  
int rebuildCost = house.CalculateRebuildingCost();
```

# Использование разделяемых классов и разделяемых методов

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
    {
    }
}
```

# Использование разделяемых классов и разделяемых методов

При определении разделяемого класса применяются следующие правила:

Каждая часть класса должны быть доступна при компиляции приложения

Каждая часть класса должна начинаться с ключевого слова **partial**

Разделяемый класс не может быть разбит на несколько сборок

Ключевое слово **partial** должно быть префиксом ключевого слова **class**

# Использование разделяемых классов и разделяемых методов

При определении разделяемого класса в нем можно определить один или несколько разделяемых методов

```
public partial class FrameworkClass
{
    partial void DoWork(int data);

    public void FrameworkMethod()
    {
        DoWork(99);
    }
}
```

Определение разделяемого метода

Вызов разделяемого метода

Реализация разделяемого метода

```
public partial class FrameworkClass
{
    partial void DoWork(int data)
    {
        . . .
    }
}
```

# Использование разделяемых классов и разделяемых методов

При определении разделяемых методов необходимо соблюдать следующие правила:

Разделяемые методы не могут возвращать значение

Разделяемые методы неявно **private**

Объявления разделяемых методов должны начинаться с ключевого слова **partial**

Разделяемые методы могут иметь **ref** параметры, но не могут **out** параметры

# Что такое структура?

Данные в переменных структурного типа хранятся своим значением

Структуры используются для моделирования элементов, которые содержат относительно небольшое количество данных

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

# Что такое структуры?

Структура может содержать поля и методы реализации

```
int x = 99;  
string xAsString = x.ToString();
```

Для структурных типов нельзя использовать по умолчанию многие из общих операций, таких как `==` и `!=`, если для них не предоставлены определения этих операций



# Определение и использование структуры

Для объявления структуры используется ключевое слово **struct**

```
struct Currency
{
    public string currencyCode;    // The ISO 4217 currency code
    public string currencySymbol; // The currency symbol ($,£,...)
    public int fractionDigits;     // The number of decimal places
}
```

Синтаксис при определении членов в структурах аналогичен синтаксису в классах

Для создания экземпляра типа структура **необязательно** использовать оператор **new**, однако структура в этом случае считается неинициализированной

```
Currency unitedStatesCurrency;
unitedStatesCurrency.currencyCode = "USD";
unitedStatesCurrency.currencySymbol = "$";
unitedStatesCurrency.fractionDigits = 2;
```



# Инициализация структуры

Если при создании экземпляра необходимо инициализировать поля структуры, можно определить один или несколько конструкторов

```
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", "£");
```

Правила обязательной инициализации всех полей структуры, аналогичные правилам для локальных переменных (definite assignment rules)

*Сколько значимых типов из .NET Framework содержит конструкторы по умолчанию?*

# Инициализация структуры

```
struct SomeStruct
{
    private int _i;
    private double _d;
    public SomeStruct(int i) : this()
    {
        _i = i;
        // Поле _d инициализировано неявно!
    }
}
```

Вызов **this()** превращается в инструкцию **initobj**, используемую для получения значения по умолчанию экземпляра структуры

Смешивание понятий конструктора по умолчанию с получением значения по умолчанию для значимых типов является общепринятым на платформе .NET, но не является обязательным. Некоторые языки, как например, «голый» IL или Managed C++, поддерживают полноценные пользовательские конструкторы по умолчанию для значимых типов, которые позволяют инициализировать состояние структуры произвольным образом, а не только значениями по умолчанию.

# Инициализация структуры

Существуют следующие различия между конструкторами структур и классов:

Для структуры нельзя определить конструктор по умолчанию

Все конструкторы структуры должны явно инициализировать каждое поле в структуре

Конструктор в структуре не может вызывать другие методы до присваивания значений всем ее полям

Если при создании экземпляра структуры не используется конструктор (либо default), структура считается неинициализированной



# Что такое перечисление?

```
d = 5;
```



```
d = DayOfWeek.Friday;
```



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

перечислимые типы подвергаются строгой проверке типов



# Что такое перечисление?

- ✓ Каждый перечислимый тип прямо наследует System.Enum, производному от System.ValueType, а тот в свою очередь — System.Object
- ✓ Перечислимые типы относятся к значимым типам и могут выступать как в упакованной, так и в упакованной формах

Перечисления создаются с помощью ключевого слова **enum**

```
public enum Color
{
    White,
    Red,
    Green,
    Blue,
    Orange
}
```

```
//psevdocode
public struct Color : System.Enum
{
    public const Color White = (Color) 0;
    public const Color Red = (Color) 1;
    public const Color Green = (Color) 2;
    public const Color Blue = (Color) 3;
    public const Color Orange = (Color) 4;

    public Int32 value__;
}
```

# Что такое перечисление?

Color

- ... .class enum nested private auto ansi sealed
- ... extends [mscorlib]System.Enum
- ... Blue : public static literal valuetype Enums.Enums/Color
- ... Green : public static literal valuetype Enums.Enums/Color
- ... Orange : public static literal valuetype Enums.Enums/Color
- ... Red : public static literal valuetype Enums.Enums/Color
- ... White : public static literal valuetype Enums.Enums/Color
- ... value\_\_ : public specialname rtspecialname int32

Color::Orange : public static literal valuetype Enums.Enums/Color

Find Find Next

`.field public static literal valuetype Enums.Enums/Color Orange = int32(0x00000004)`

Color::Green : public static literal valuetype Enums.Enums/Color

Find Find Next

`.field public static literal valuetype Enums.Enums/Color Green = int32(0x00000002)`

# Создание новых типов перечисления

Перечисления можно объявить в классе или пространстве имен, но нельзя в методе

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
};
```

```
enum Season
{
    Spring = 1,
    Summer,
    Autumn,
    Winter
};
```

```
enum Season
{
    Spring = 1,
    Summer,
    Autumn = 3,
    Fall = 3,
    Winter
};
```



базовый класс FCL (Int32)

```
enum Season : short
{
    Spring,
    Summer,
    Autumn,
    Winter
};
```

byte sbyte short ushort int uint long ulong

# Инициализация и присваивание переменных перечисления

Объявление переменных перечисления и присваивание им значений выполняется аналогично другим типам в C#

```
enum Day
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
};

static void Main(string[] args)
{
    Day dayOff = Day.Sunday;
}
```

`[EnumType] variableName = [EnumValue]`



# Инициализация и присваивание переменных перечисления

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

```
for(Day dayOfWeek = Day.Monday; dayOfWeek <= Day.Sunday; dayOfWeek++)  
{  
    Console.WriteLine(dayOfWeek);  
}
```

Monday  
Tuesday  
Wednesday  
Thursday  
Friday  
Saturday  
Sunday

Переменные перечисления можно сравнивать

Для переменных перечисления можно выполнять целочисленные операции, такие как инкремент и декремент

«==», «!=», «<», «>», «<=», «>=»

Day.Monday + Day.Wednesday



# Упаковка и распаковка

```
Residence house = new Residence(...);  
object obj = house;
```

←..... Класс

```
Currency currency = new Currency(...);  
object o = currency;
```

←..... Структура  
←..... Упаковка

## Упаковка (boxing)

CLR выделяет кусок памяти в куче

Копирует значение переменной в эту часть памяти, а затем связывает объект с копией

# Упаковка и распаковка

```
Currency currency = new Currency(...);  
object o = currency;  
...  
Currency anotherCurrency = (Currency)o;
```

Для получения значения упакованной копии необходимо использовать приведение типов

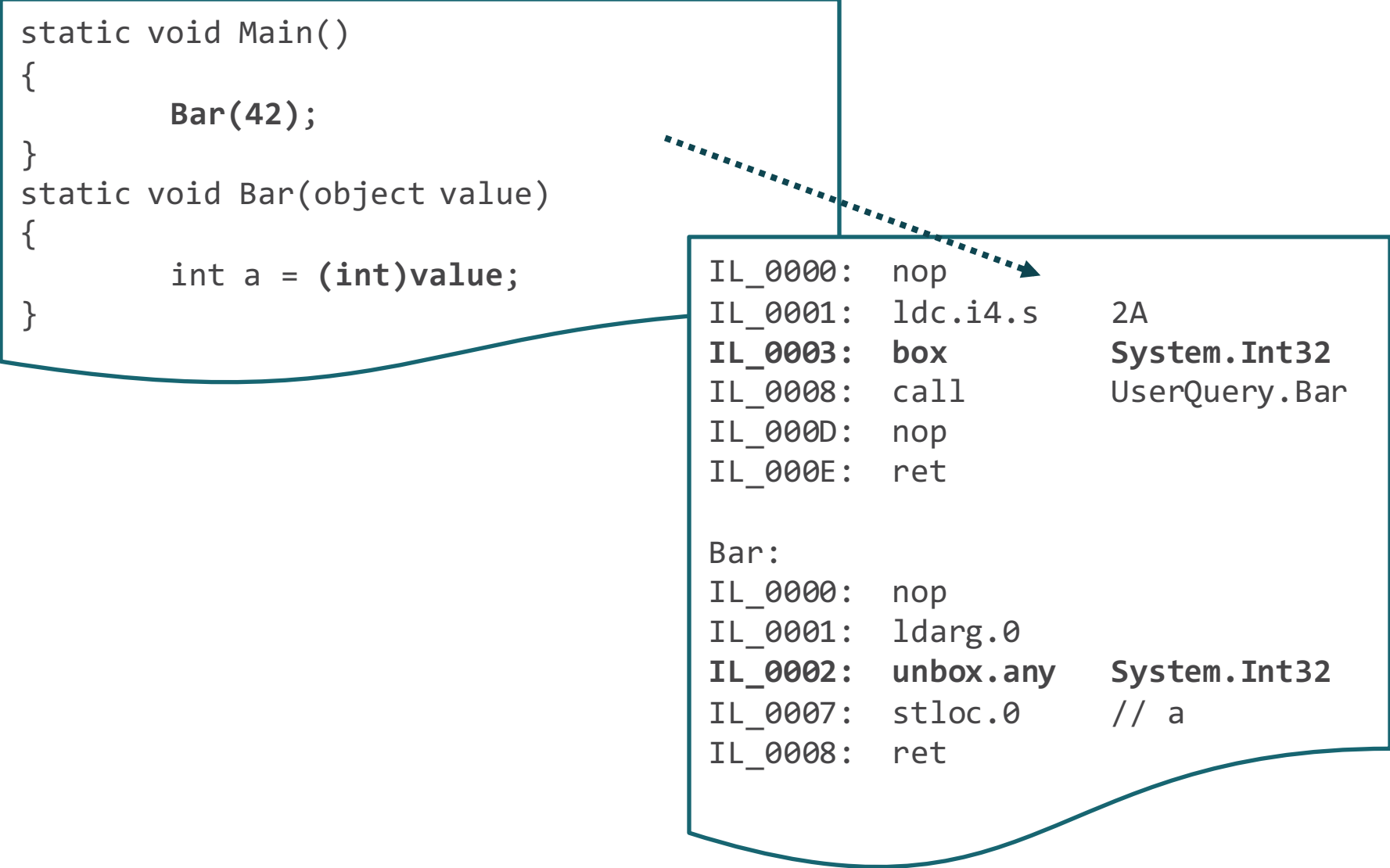
## Распаковка (unboxing)

CLR проверяет тип объекта

Если типы совпадают, извлекает значение из упакованного объекта в куче и копирует его в переменную в стеке

# Упаковка и распаковка

```
static void Main()  
{  
    Bar(42);  
}  
static void Bar(object value)  
{  
    int a = (int)value;  
}
```

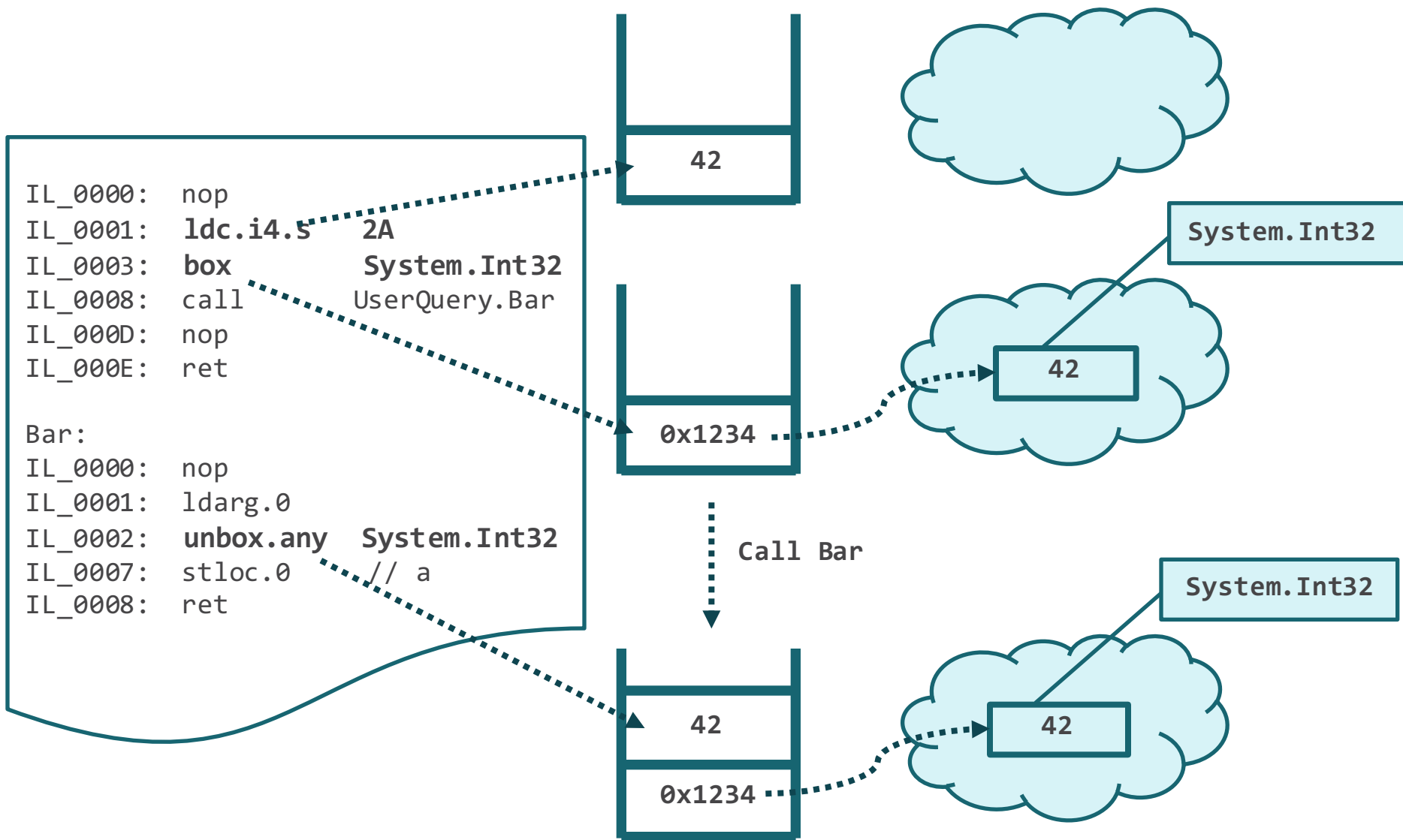


IL\_0000: nop  
IL\_0001: ldc.i4.s 2A  
IL\_0003: **box** **System.Int32**  
IL\_0008: call UserQuery.Bar  
IL\_000D: nop  
IL\_000E: ret

Bar:

IL\_0000: nop  
IL\_0001: ldarg.0  
IL\_0002: **unbox.any** **System.Int32**  
IL\_0007: stloc.0 // a  
IL\_0008: ret

# Упаковка и распаковка



# Обнуляемые типы

При объявлении ссылочной переменной можно установить ее значение в null, чтобы указать, что она не инициализирована

```
Residence house = null;  
...  
if (house == null)  
{  
    house = new Residence(...);  
}
```

```
Currency currency = null;
```



СТЕ

Чтобы указать, что тип значения является обнуляемым, используется знак вопроса «?»

```
Currency? currency = null;  
...  
if (currency == null)  
{  
    currency = new Currency(...);  
}
```

# Обнуляемые типы

Типы, допускающие значения `null`, по сути являются экземплярами структуры `System.Nullable<T>`

`Nullable<Int32>`

любое значение от -2 147 483 648 до 2 147 483 647 или значение `null`

`Nullable<bool>`

значения `true`, `false` или `null`

```
Currency? currency = null;
...
if (currency.HasValue)
{
    Console.WriteLine(currency.Value);
}
```

Свойство `HasValue` указывает, содержит ли обнуляемый тип значение или `null`

Свойство только для чтения `Value` содержит значение переменной

# Обнуляемые типы

```
int? i = null;  
int j = 99;  
i = 100
```



```
i = j;
```



```
j = i;
```



Нуль-коалесцирующая операция (операция поглощения) «??» используется для определения значения по умолчанию для обнуляемых значимых типов, а также ссылочных типов. Он возвращает левый операнд, если он не является нулевым, в противном случае он возвращает правый.

```
int x = (b.HasValue) ? b.Value : 123;
```

```
int x = b ?? 123;
```

```
string temp = GetFilename();  
filename = (temp != null) ? temp : "Untitled";
```

```
string filename = GetFilename() ?? "Untitled";
```





**Спасибо за внимание!**





**Надеюсь, что Вы найдете этот материал полезным.**

**Если Вы нашли ошибки или неточности в этом материале или знаете, как его улучшить, пожалуйста, сообщите по электронному адресу: [anzhelika.kravchuk@epam.com](mailto:anzhelika.kravchuk@epam.com) с пометкой [ASP.MVC Training Course Feedback]**

**Спасибо.**