

Разширеното разбиране за интерфейси (вкл. в .NET)

- * Когато не искаме да реализираме наследство на типове, но общ договор за поведение.
- * CLR поддържа единично наследяване на имплементация и множествено на интерфейси
- Интерфейсите в C# могат да дефинират и събития, свойства без и с параметри, тъй като те са всъщност методи
- според CLR интерфейсите могат да притежават и статични методи, полета и конструктори (на типа), но някои езици (C#) не го позволяват.
- * интерфейсите се маркират с `public`, `protected`, `private`, `internal` (както класовете или структурите)
- * ако не укажете метод в имплементацията му с `virtual`, той се счита `virtual`, `sealed`
- * типът трябва да имплементира всички методи на наследените интерфейси. В противен случай той е абстрактен (незавършен) тип.
- * обекти от даден тип могат да се третират и като някой от неговите интереси:

```
String s = "Student";  
//сега могат да се викат методите на интерфейсите имплементирани  
// в типа String и Object: IComparable; ICloneable; IConvertible, IEnumerable  
IComparable comparable = s;  
// с горната пром. могат да се викат само методите на IComparable  
ICloneable cloneable = s;  
// променлива от тип интерфейс може да се преобразува в друг  
// интерфейс, ако и двата се поддържат в типа  
IEnumerable enumerable = (IEnumerable) cloneable;
```

- Всички действия изпълнение с comparable, cloneable, enumerable или s, всъщност влияят на обекта "s". Типът на променливата определя само допустимите действия, които се изпълняват с нея
- Реализацията на наследен интерф.метод отразява спецификата на конкретния обект, но при задължителна сигнатура на метода. Ако е необходимо, кодът прави нужните преобразувания в типа (напр. за IComparable())

пример на 3 дефинирани в FCL (Framework Class Library) интерфейса:

```
public interface System.Collections.IEnumerable
{
    IEnumerator GetEnumerator();
}
public interface System.Collections.IEnumerator
{
    Boolean MoveNext();
    void Reset();
    Object Current { get; }           //свойство за четене
}
public interface System.IComparable
{
    Int32 CompareTo (Object object);
}
```

Тип или интерфейс? За и против:

- кое е важното: IS-A? или CAN DO (интерфейс)?
- Лесно използване: ако работите с типове – наследявате готовата функционалност на типа
- Устойчива имплементация при тип. Интерфейсът всеки път се имплементира с възможност за грешки
- Относно нови версии – промени в базов тип се наследяват без необходимост от промени в производните. Дори не е нужна прекомпиляция. Това не е така при промяна в интерфейсна дефинция.

1.

Проблем с промяна на поле в boxed value тип и решаването му посредством интерфейси

ето проблема:

```
using System;
struct Point
{
    public Int32 x,y;
    public void Change(Int32 x, Int32 y) { this.x = x; this.y = y; }
    public override String ToString()
        { return String.Format( "{0}, {1}", x, y); }
}
class App
{ static void Main()
    {
        Point p = new Point();           // в стек - point е стойностен тип
        p.x = p.y = 1;
        Console.WriteLine(p);             // (1,1)
        p.Change(2, 2);
        Console.WriteLine(p);             // (2, 2)
        Object o = p;                     // boxing
        Console.WriteLine( o );            // (2, 2)
        (( Point) o).Change(3, 3);         // защото Change() е метод на Point
// “ о “ се разупакова. Полетата му се копират в нов value type Point в стека
//и се променят на 3
        Console.WriteLine( o );           // (2, 2) – това е “пакетираният” обект
// C++ позволява промяна в полета на опакован тип, но C# - не. Решението:
```

ето решение на проблема (въвеждаме интерфейс):

```
using System;
```

```
interface IChangeBoxedPoint {  
    void Change(Int32 x, Int32 y);  
}
```

```
struct Point : IChangeBoxedPoint  
{  
    public Int32 x,y;  
    public void Change(Int32 x, Int32 y) { this.x = x; this.y = y;}  
    public override String ToString()  
        { return String.Format( "{0}, {1}", x, y);  
        }  
}
```

```
class App  
{ static void Main()  
    {  
        Point p = new Point();  
        p.x = p.y = 1;  
        Console.WriteLine(p);           // (1,1)  
        p.Change(2, 2);  
        Console.WriteLine(p);           // (2, 2)  
        Object o = p;                    // boxing  
        Console.WriteLine( o );         // (2, 2)  
        (( Point) o).Change(3, 3); // защото Change() е метод на Point  
// “ о “ се разупакова. Полетата му се копират във value type Point в стека и се  
// променят на 3
```

```
Console.WriteLine( o );  
// (2, 2) – това е “пакетираният” обект ‘o’
```

//опитваме така:

```
((IChangeBoxedPoint) p).Change(4,4);  
// “p” се преобразува до интерфейс.  
// Променят се стойности на 4,4, но обектът не се реферира  
// повече и се маркира за излишен (garbage collector го унищожава)
```

```
Console.WriteLine(p);           // 2,2
```

// сега опитваме така:

```
((IChangeBoxedPoint) o).Change(5,5);  
Console.WriteLine( o );         // (5,5)
```

```
// имаме преобразуване, не пакетиране или препакетиране!!!  
// обръщението е към пакетирания вече point  
// извикването на Change() се отнася към него и всичко е ОК
```

2.

Искаме да имплементираме няколко интерфейса, имащи сигнатурно-еднакъв метод

```
public interface IWindow  
    { Object GetMenu(); }  
public interface IRestaurant  
    { Object GetMenu(); }
```

дефинираме тип:

```
public class MyPizza : IWindow, IRestaurant  
    {  
        Object IWindow.GetMenu() { ..... }  
        Object IRestaurant.GetMenu() { ..... }
```

**//обърнете внимание – интерфейсите методи не са public – дефинирани.
// Те са понякога public, понякога – private (както ще видим в обръщенията
// към тях)**

```
        public Object GetMenu() { ..... }  
    }
```

използваме така дефинирания тип в някакъв наш метод:

```
static void MyMethod()  
    {  
        MyPizza mp = new MyPizza(); // инстанция на типа  
        Object menu;
```



```
menu = mp.GetMenu(); //това е единствения явно дефиниран public  
menu = ((IWindow) mp).GetMenu();  
menu = ((IRestaurant) mp).GetMenu();  
}
```

3.

двойна имплементация на интерфейсни членове

когато базов интерфейс работи с параметри от даден тип, наследяващият го следва да работи с същите типове. Имаме сандартен интерфейс:

```
public interface IComparable
{ Int32 CompareTo( Object other); } // сравнява обекти
```

създаваме **наш тип**:

```
struct SomeValuetype : IComparable
{
    private Int32 x;
    public SomeValueType(Int32 x) {this.x = x;}
    public Int32 CompareTo( Object other) {return(x-((SomeValueType)other).x);}
}
```

/* можем да сравняваме обекти, а не нашия тип. Лошото е че компилаторът не открива грешката ако напишем:*/

```
SomeValueType v = new SomeValueType(0);
```

```
Object o = new Object();
```

```
Int32 n = v.CompareTo(o); // грешка, но неоткрита!
```

къде в 'o' има член 'x'? 

- решението на проблема е през **“двойна имплементация на член на интерфейс”**:
- В нашия тип следва да имаме 2 имплементации на CompareTo(). Едната сравнява обекти (както е в сигнатурата на метода на наследения интерфейс), но не е public. Извършва нужните преобразувания в типовете и препраща към собствена за типа реализация на CompareTo(). В нея се сравняват исканите (конкретни) стойности.
- Така компилаторът “хваща” грешката при подаден неправилен тип: 

```
struct SomeValueType : IComparable {  
    private Int32 x;  
    public SomeValueType(Int32 x) {this.x = x;}
```

```
    public Int32 CompareTo(SomeValueType other) {  
        return(x - other.x);}
```

```
// явна имплементация на интерфейския метод  
Int32 IComparable.CompareTo(Object other) //името е пълно!!  
{ return CompareTo((SomeValueType) other); }
```

Използването вече е типово-обезопасено:

```
SomeValueType x1 = new SomeValueType(1);  
SomeValueType x2 = new SomeValueType(2);  
Int32 n;
```

```
n = x1.CompareTo(new Object());
```

```
n = x1.CompareTo(x2);
```

```
// вика с етипово-обезопасения метод
```

```
IComparable comp = (IComparable) x1; // опакова се x1
```

```
n = comp.CompareTo(x2); // x2 също се опакова и всичко е ОК
```

вече не е нужно пакетиране.
Това е добре!!

//грешка при компилация

//ОК.

Бележка за обмисляне:

Имплементацията `Comparable.CompareTo()` не може да има модификатор за достъп `public` или `private`. Тя е понякога `public` (когато работим с обекти `Comparable`), понякога `private` (когато работим с `SomeValueType`)!!

Целта е осигуряване на типова безопасност!

Друга възможна цел на двойните имплементации е избягване на нежелани операции по пакетиране, като се вика подходящата имплементация

Съвместимост на COM-IDL дефиниции с CLR (Common Language Runtime) в .NET

Damien 99

- (терминът е: COM Interop) за съвместимост на COM и CLR
- работи се с метаданните
- създават се прокси-обекти в клиентския и в сървърен процес. Той подменя работата с QueryInterface(), AddRef() или Release() от COM технологията.
- tools подпомагащи прехода COM $\leftrightarrow$ components in CLR:
 - **Type Library Importer** (tlbimp.exe): от type library информация $\rightarrow$ създава assembly с метаданново описание за компонента
 - **Type Library Exporter** (tlbexp.exe): type library $\leftarrow$ CLR assembly
 - **Register Assembly** (regasm.exe): запис в Registry на информация за асемблито, така че COM компонентите могат да виждат и ползват обекти от него.

пример:

имаме COM компонент HelloATL с IDL файлова дефиниция:

```
import "oaidl.idl";
[ object, uuid (FF03 .....),
]
interface IHelloATL : IDispatch
{
    HRESULT Hello([in] BSTR bstr);
}
library HELLOATLLib
{
    importlib("stdole32.tlb");
    coclass HelloATL
    {
        [default] interface IHelloATL;
    };
};
```

- компилираме idl файла. Получаваме типова библиотека.
- type library importer от тази библиотека създава нужния CLR файл с метаданни: 2 CLR интерфейсни описания (HelloATL (за класа) и IHelloATL(за интерфейса)) и CLR клас – HelloATLClass, съобразно COM описанието. Асемблито е в DLL – HELLOATLLib.dll
- всъщност е създадено проху за COM компонента – обръщенията ще са през него

ако разгледаме типовото описание от метаданните, например така:

```
HelloATL h = new HelloATL();  
Type t = h.GetType();  
MethodInfo[ ] ma = t.GetMethods();
```



за новосъздадения тип – HelloATL ще видим следните методи:

Void Hello(System.String) // нашият метод от idl интерфейса

System.Runtime.Remoting.ObjRef CreateObjRef(System.Type)
// за създаване на проху от дадения тип

Boolean Equals(System.Object) // стандартен – за сравняване на типове

System.String ToString() // стандартен – за връщане името на типа

System.Type GetType() // стандартен – връща обект от тип 'Type' с
// описания на метаданните на създадения тип

Dynamic Dispatch

perhaps the biggest change in C# 4.0 (Visual studio 2010) are in the interop features. C# now supports dynamic late-binding. The language has always been strongly typed, and it continues to be so in version 4.0. But there are times when you need to communicate with systems not based on .NET- using controls for example. Traditionally, there were at least two approaches to this. The first was simply to **import the foreign model directly into .NET as a proxy. COM Interop provides one example**. Since the original release of the .NET Framework, it has used this strategy with a tool - TLBIMP, which creates new .NET proxy types you can use directly from C#.

The second approach abandons the C# type system entirely (to embed strings and data in your code). This is what you do whenever you write code that, say, **invokes a method** on a JScript object or when you embed a SQL query in your ADO.NET application.

The **dynamic keyword** in C# is a response.

Let's start with a simple example. Normally, **without 'dynamic'**, it requires a lot of infrastructure code, such as:

```
object o = GetObject();  
Type t = o.GetType();  
object result = t.InvokeMember("MyMethod", BindingFlags.InvokeMethod, o,...);  
int i = Convert.ToInt32(result);
```

With the **dynamic keyword**, instead of calling a method MyMethod on some object , you can now tell the compiler to please treat 'o' as dynamic and delay all analysis until run time.

Code that does that looks like this:

```
dynamic o = GetObject();  
int i = o.MyMethod();
```

It works, and it accomplishes the same thing.

This is shortened & simplified C# syntax. Somewhere inside in a surrounding ScriptObject class, the **InvokeMember method** exists.

In C# 4.0, dynamic is a built-in type.

Note, however, that dynamic is different from var.

Variables declared with var actually do have a strong type,
but the programmer has left it up to the compiler to figure it out.

When the programmer uses dynamic, the compiler doesn't know what type is being used—the programmer leaves figuring it out up to the runtime.

C# Dynamic

IronPython

IronRuby

Dynamic APIs

Dynamic Language Runtime (DLR)

Rest of the
.NET Framework

Common Language Runtime (CLR)

Embedding COM Interop Types

This is more of a C# compiler feature than a C# language feature, but now, having dynamic, you can use a **COM interop assembly** without that assembly having to be present at run time. The goal is to reduce the burden of deploying COM interop assemblies with your application.

When COM interop was introduced in the original version of the .NET Framework, the notion of a **Primary Interop Assembly (PIA)** was created. This was an attempt to solve the problem of **sharing COM objects among components**. If you had different interop assemblies that defined an Excel Worksheet, we wouldn't be able to share these Worksheets between components, because they would be different .NET types. The PIA fixed this by existing only once—all clients used it, and the .NET types always matched.

Though a fine idea on paper, in practice deploying a PIA turns out to be a headache, because there's only one, and multiple applications could try to install or uninstall it. Matters are complicated because PIAs are often large, Office doesn't deploy them with default Office installations, and users can circumvent this single assembly system easily just by using TLBIMP to create their own interop assembly.

The practice now:

even without knowledge of the details, it's another feature—dynamic—you should be able to use without a problem. So, you **tell the compiler to embed interop types for you in Visual Studio by setting the Embed Interop Types property on your reference to true.**

From theory to practice: Easy Access to COM Objects from .NET structure

An object is said to be **dynamic** when its structure and behavior aren't fully described by a statically defined type that the compiler knows thoroughly. Let's look at a simple example. In a scripting language such as VBScript, the following code runs successfully:

```
Set word = CreateObject("Word.Application")
```

The CreateObject() assumes that the string it gets as an argument is the progID of a registered COM object. It creates an instance of the component and returns its IDispatch automation interface.

The details of the IDispatch interface are never visible at the level of the scripting language.

What matters is that you can write code such as:

```
Set word = CreateObject("Word.Application")  
word.Visible = True  
Set doc = word.Documents.Add()  
Set selection = word.Selection  
selection.TypeText "Hello, world"  
selection.TypeParagraph()  
doc.SaveAs(fileName)
```

In this code, you first create a reference to a component that automates the behavior of the underlying Microsoft Office Word application. Next, you make the Word main window visible, add a new document, write some text into it and then save the document somewhere. The code is clear, reads well and, more importantly, works just fine.

The reason this works, however, is due to a particular capability offered by VBScript—late binding.

In Visual Basic, the `CreateObject()` exists for (strong) compatibility reasons. The point is that .NET Framework-based languages were designed with early binding in mind. COM interoperability is a scenario addressed by the .NET Framework but never specifically supported by languages with keywords and facilities—not until C# 4.0.

C# 4.0 (and Visual Basic) has **dynamic lookup capabilities** that indicate late binding is now an approved practice for .NET Framework developers. With dynamic lookup, you can code access to methods, properties, indexer properties and fields in a way that bypasses static type checking to be resolved at run time.

***** Dynamic Language Runtime – what is happening inside?

Let's see what happens when the compiler encounters the following code:

```
class Program
{
    static void Main(string[] args)
    {
        dynamic x = 1;
        Console.WriteLine(x);
    }
}
```

*****Within the DLR, the compiler-provided expression is encapsulated in a dynamically updated site object. **A site object is responsible for binding methods to objects** on the fly.

The Real Implementation of a Dynamic Variable:

```
internal class Program
```

```
{  
private static void Main(string[] args)  
{  
    object x = 1;  
    if (MainSiteContainer.site1 == null)  
    { MainSiteContainer.site1 = CallSite< Action<CallSite, Type, object>>  
        .Create(Binder.InvokeMember( "WriteLine", null, typeof(Program),  
            new CSharpArgumentInfo[] { CSharpArgumentInfo.Create(...) }));  
    }  
}
```

```
    MainSiteContainer.site1.Target.Invoke( site1, typeof(Console), x);  
}
```

```
private static class MainSiteContainer
```

```
{  
    public static CallSite<Action<CallSite, Type, object>> site1;  
}
```

To invoke the method Console.WriteLine on a dynamic variable, you invoke the site and pass the target object (Console) and its parameters (in this case the dynamic variable). Internally, the site will check whether the target object really has a member WriteLine that can accept a parameter like the object currently stored in the variable x

The **dynamic variable** is mapped to a System.Object instance and then **a site is created for the program** in the DLR. The site manages a binding between the WriteLine method with its parameters and the target object. The binding holds within the context of the type Program.

The **dynamic** keyword and other COM interop features introduced in C# 4.0 don't make a piece of code necessarily faster, but it enables you to write C# code as if it were script.

Since the beginning of the .NET Framework, you could wrap a COM object into a managed class and use it from a .NET-based application.

For this to happen, you need to use using a **primary interop assembly (PIA)**.

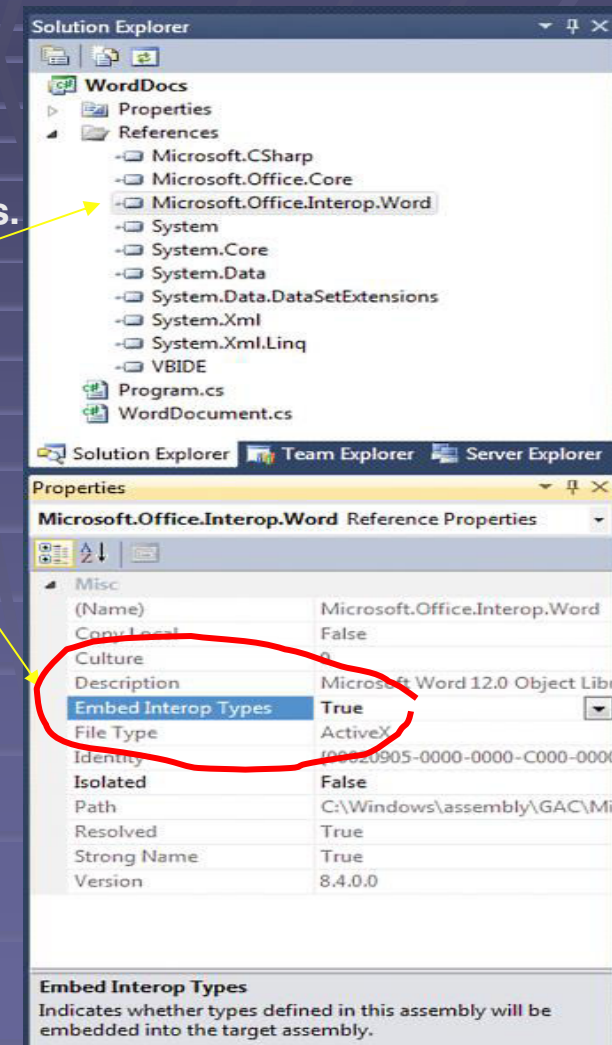
PIAs are necessary and must be deployed along with client applications. However, PIAs are too big and wrap up an entire COM API.

Visual Studio 2010 offers the **no-PIA option**.

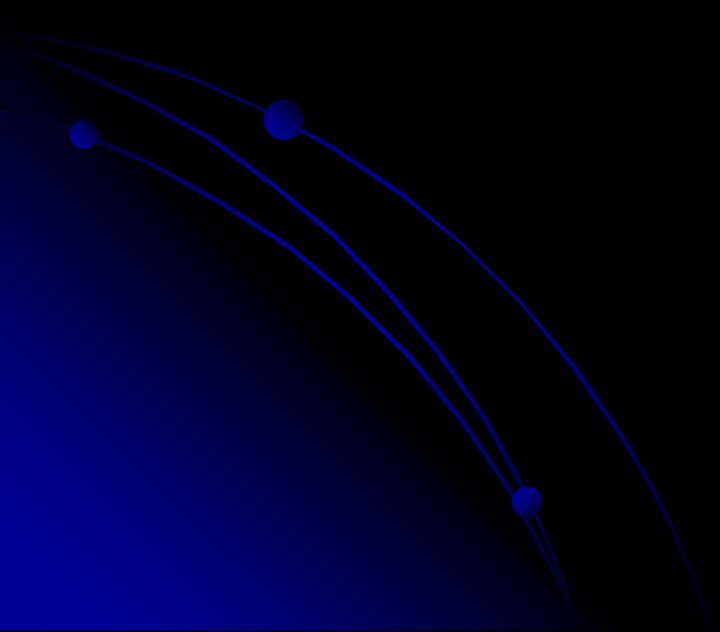
No-PIA refers to the compiler's ability to embed required definitions you'd get from a PIA in the current assembly. As a result, only definitions that are really needed are found in the final assembly and there's no need for you to pack vendor's PIAs in your setup.

In summary, working with COM objects can still be expensive, but the COM interop support in C# 4.0 makes the code you write far simpler.

COM is a necessary evil in the .NET Framework, but dynamic makes it a bit less so.



ActiveX



ActiveX контроли

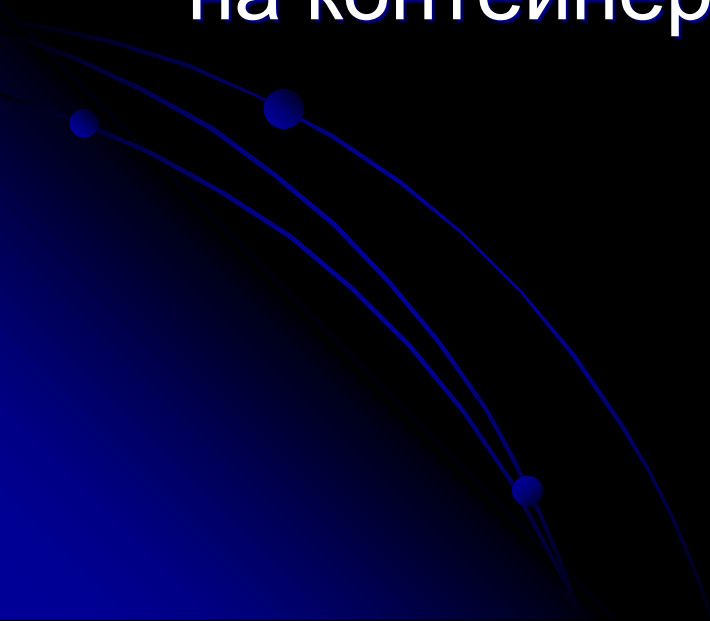
- ActiveX контролите са най-популярната за момента част от ActiveX технологията. Освен тях се използват:
 - ActiveX Data Object (ADO)
 - Active Server Pages (ASP)
- Те са COM базирани и следователно се подчиняват на правилата и концепциите в COM модела.

ActiveX контроли

- ActiveX контрол - функционално завършено парче код, пакетирано за многократно използване и предоставящо определен интерфейс за взаимодействие с клиентите си.
- ActiveX контролът е COM обект, но задължен да имплементира определени интерфейси, които го карат да се държи като контрол.
- ActiveX контролите задължително имплементират само интерфейса IUnknown.
- Повечето контроли имплементират интерфейса IDispatch, което ги прави Automation сървъри (експонират методи и свойства).

ActiveX контроли

- ActiveX контролите могат да са както видими (поддържат свой прозорец), така и невидими, при което те нямат свой прозорец – рисуват се в прозореца на контейнера.



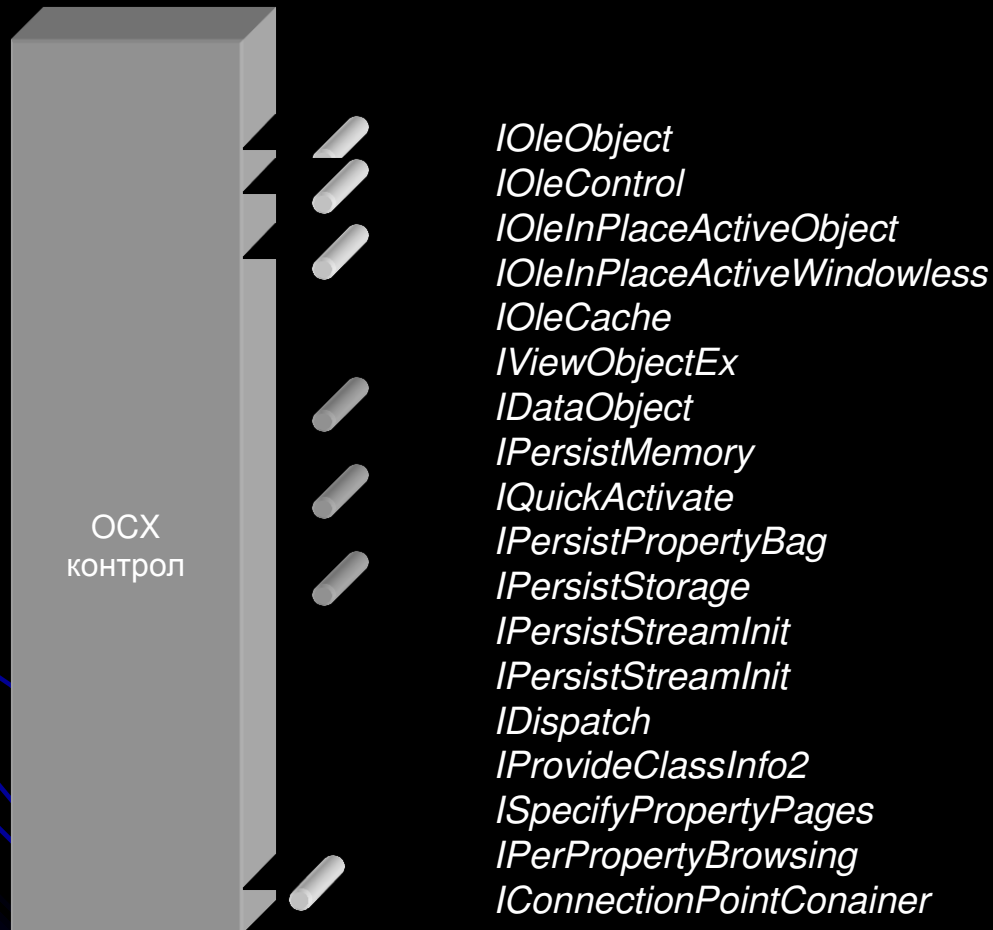
ActiveX контроли

- Контролът може да е активен (с или без прозорец) или неактивен. Тогава контейнерът го изобразява в своя прозорец от изображение, взето от метафайл. Така се спестява ресурс. ActiveX контролът се активира при кликване с бутона на мишката върху неговия образ. За целта се използва интерфейсът `IPointerInactive` на контрола, който следи движенията на мишката.

ActiveX контроли

- Най-често контролите се реализират като in-process сървъри (т.е те са в DLL).
 - Някои от интерфейси, които контролите имплементират нямат реализиран marshalling механизъм за настройка при междупроцесен обмен. Реализацията на такъв е доста обемна задача.
 - По-бърза реакция отколкото при реализация в EXE сървър.
 - В миналото OLE контролите бяха само in-process.

Интерфейси на ActiveX контролите



Интерфейси на ActiveX контролите

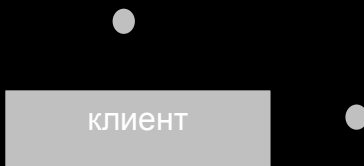
- `IConnectionPointContainer` - Използва се при нотификационни съобщения
- `IDataObject` - Предоставя за достъп данни от контрола
- `IDispatch` - Поддържа автоматизация. Предоставя по определен начин за достъп методи и данни на контрола.
- `IObjectCache` - Управява кеш памет за обменяните данни.
- `IObjectControl` - Базов интерфейс, който прави контрола ActiveX.
- `IObjectInPlaceActiveObject` - Позволява in-place активиране.
- `IPerPropertyBrowsing` - Позволява контейнер да извлича информация за свойства на контрола, например имената им.
- `IPersistMemory` - Прави данните на контрола персистентни – те могат да се съхраняват и впоследствие да се използват отново.

ActiveX контролите като Connectable обекти

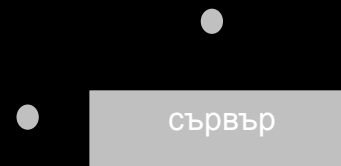
- ActiveX контролите комуникират със своите контейнери по дефиниран протокол, чрез специфични интерфейси.
 - IConnectionPoint – реализиран е в контрола и позволява клиента да заяви желанието си за event notifications.
 - IConnectionPointContainer - използва се за запитване на обект дали поддържа първия интерфейс. Притежава функции връщащи указател към IConnectionPoint или повече указатели към няколко такива интерфейса (ако има такива реализирани в обекта).

ActiveX контролите като Connectable обекти

IPROPERTYNOTIFYSink



ICONNECTIONPOINT



ICONNECTIONPOINTCONTAINER

- Клиентът, желаещ ползване на определен `ICorrelationPoint`, извиква функцията **QueryInterface()** за интерфейса `ICorrelationPointContainer`. Ако това е успешно, клиентът подава указател към своя notification sink интерфейс към сървъра чрез `ICorrelationPoint`. Притежавайки този указател, сървърът подава нотификационни съобщения на клиента.

ActiveX контролите като Connectable обекти

- Реализацията на тези интерфейси е тежка задача. MFC предоставя подразбираща се имплементация за тях в класовете CConnectionPoint и CConnectionPointContainer.
- Механизмът на интерфейсите:
 - интерфейсът IConnectionPointContainer се използва, за да предостави на контрола IDispatch интерфейсен указател за контейнера. Следователно контейнерът също е Automation обект.
 - С помощта на този указател контролът извиква методи от контейнера в реакция на свои събития.
 - От своя страна, за да се информира какви събития може да предизвика контрола, както и за да съберат информация за него, контейнерите могат да ползват методи от интерфейсите IProvideClassInfo2 и ITypeInfo на контрола, даващи достъп до информация за типовете на контрола.

Пропъртита, събития и методи на ActiveX контрола

- Взаимодействие с контролите става чрез пропърти (атрибути), методи (експонирани функции, които могат да се приложат към контрола чрез интерфейса IDispatch) и събития (съобщения, предавани към контейнера от контрола, както беше описано по-горе).

Атрибути (Properties)

- ActiveX контролите имплементират свойства, с което позволяват да бъдат модифицирани в зависимост от нуждите на дадено приложение.
- Видове свойства:
 - управляващи средата за съвпадане с поведението на клиента - например цвят на фона, шрифт, и др.
 - разширени – реализират се в клиента, но се ползват и от контрола. Например подредба на бутони в OLE контрола, съобразно настройки в клиента.
 - стандартни - те се придават на контрола, но се реализират от ActiveX Development Kit.
 - потребителски.
- И за стандартните и за потребителските свойства е необходимо да се знаят имената и DISPID (dispatch) идентификаторите, определени в спецификацията на контрола.

Методи (Methods)

- Методите са функции, реализирани точно така, както и в OLE Automation при използване на интерфейса IDispatch. Те могат да бъдат извиквани за изпълнение на някаква полезна работа.
- Методите могат да се разделят на потребителски и стандартни.

Събития (Events)

- ActiveX контролът поддържа и събитийно множество. Събитията се използват за изпращане на нотификационни съобщения от контрола към контейнера, който съдържа контрола - например кликване с мишка.
- Видове събития:
 - стандартни – реализирани са от развойната среда и се извикват подобно на стандартните функции, например генериране съобщение при грешка - `FireError()`.
 - потребителски – за дефинирането им са предвидени облекчаващи средства в средата. За тях, както в автоматизацията, трябва да използваме за повикване име и `dispatch` идентификатор.

Амбиентни свойства

- Свойства, които контейнерът предоставя за проверка или настройка на контрола.
- Пример: контролът може да се настрои да изглежда като използващия го в момента контейнер (например цвят на фона).

Амбиентни свойства

- Принцип на реализация: ползване на автоматизацията (амбиентното свойство е automation свойство), и посредничеството на интерфейса Idispatch, реализиран в контейнера.
- Амбиентните свойства са точно специфицирани в стандарта като имена и dispatch идентификатори.

Амбиентни свойства

- COleControl опростява извличането на амбиентни свойства, като реализира обвиващи функции, които извикват функцията **IDispatch::Invoke()**.
- Видове свойства:

Свойство	Dispatch идентификатор	COleControl метод
Font	DISPID_AMBIENT_FONT	AmbinetFont()
BackColor	DISPID_AMBIENT_BACKCOLOR	AmbinetBackColor()
ForeColor	DISPID_AMBIENT_FORECOLOR	AmbientForeColor()

Амбиентни свойства

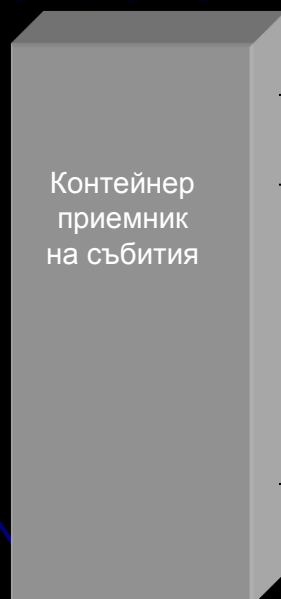
- Амбиентно свойство, за което липсва специална функция, може да се прочете с помощта на функцията **GetAmbientProperty()** с подаден dispatch идентификатор на свойството.
- Ако някое амбиентно свойство се промени в контейнера, то за това следва да се уведоми контролът.

Амбиентни свойства

- Това става в предефинираната функция **COleControl::OnAmbientPropertyChange()** в класа на контрола. Тя се извиква, когато контейнерът трябва да уведоми контрола, че някое амбиентно свойство се е променило. Като параметър на повикването се подава **dispatch** идентификатор на амбиентното свойство. В кода на тази функция контролът трябва да реализира реакцията си. Най-често тя се свежда до прерисуване на контрола (прави се обръщение към метода **InvalidateControl()**).

Контейнери на ActiveX контроли

- Контейнерът е COM обект. Това се налага поради необходимостта да се комуникира с контрола, което може да стане с посредничеството на интерфейси.

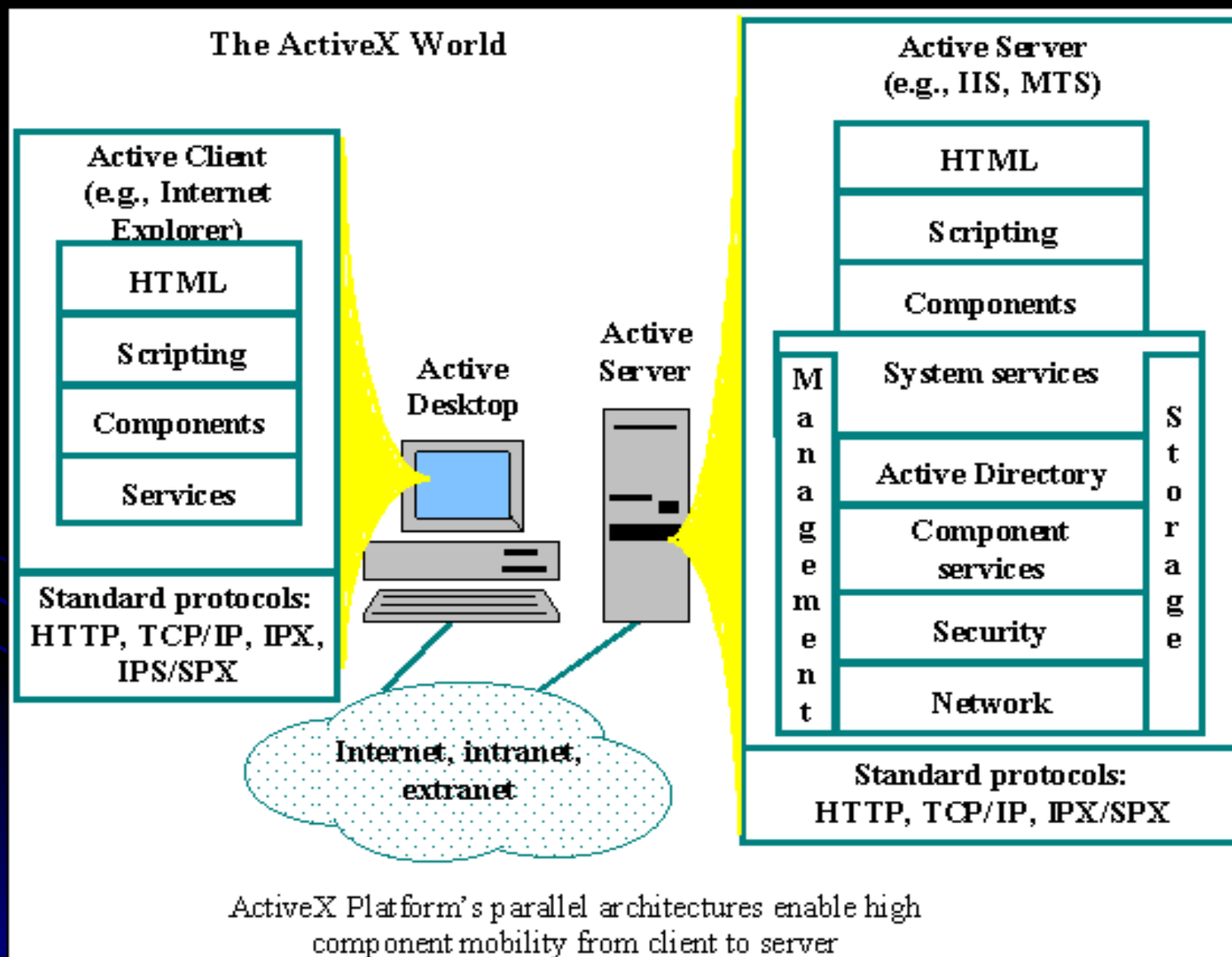


- *IOleContainer*
IOleInPlaceFrame
- *IOleClientSite*
IOleControlSite
IDispatch
IDispatch
IPropertyNotifySink
-

Контейнери на ActiveX контроли

- IOleInPlaceFrame - Използва се за in-place активиране.
- IPropertyNotifySink - Използва се при нотификационни съобщения от контрола към контейнера.
- IDispatch - Предоставя амбиентните свойства на контейнера.
- IDdpatch - Използва се за отработване на събития (events) от контейнера.
- IOleControlSite - Базов интерфейс на сайт с ActiveX контрол.

Архитектура

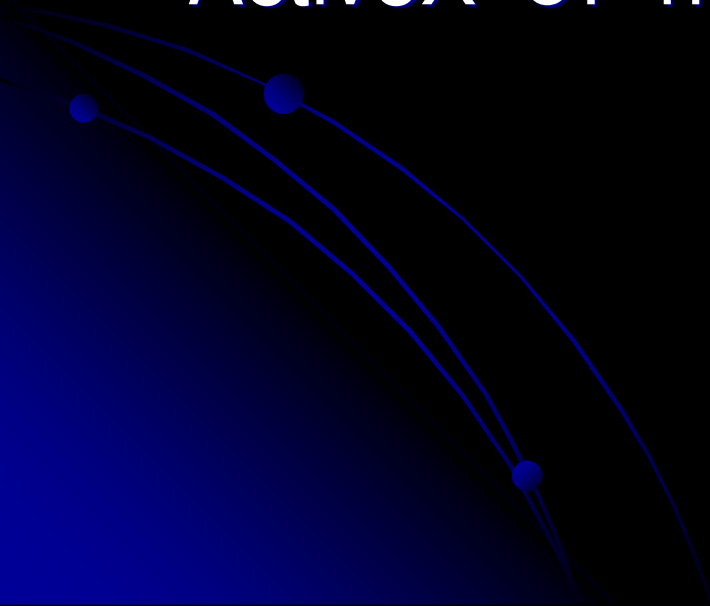


Други ActiveX технологии

- ActiveX Data Objects (ADO)
- Active Server Pages
- ActiveMovie >>>>> DirectShow
- Active Messaging >>>>> Collaboration Data Objects
- Active Scripting
- ActiveX Streaming Format (ASF)

Пример .NET

Ще разгледаме пример за създаване на ActiveX control, който използваме в html страница. Ще демонстрираме как чрез Java Script се предават параметри на ActiveX от html контроли.



Security Settings



Settings:

 ActiveX controls and plug-ins

 Automatic prompting for ActiveX controls

☐ Disable

☒ **Enable**

 Binary and script behaviors

☐ Administrator approved

☐ Disable

☒ Enable

 Download signed ActiveX controls

☐ Disable

☐ Enable

☒ Prompt

 Download unsigned ActiveX controls

☒ Disable

Reset custom settings

Reset to:

Medium

Reset

OK

Cancel

OLE Automation

цел :

Да имаме достъп до функционалност на приложение от скриптов език (Visual Basic, Java Script и т.н.) Скриптовите езици не поддържат COM технология и COM интерфейси , виртуални таблици (v-tables) и интерфейси.

Automation сървъри са и Excel, Microsoft Word, Office пакет, PowerPoint.

Основни понятия

COM обектът казваме, че експонира интерфейси и методи.

Automation обектът експонира **методи и свойства (properties)**. Скрипт език може да ги използва, все едно са негови (спомнете си, че COM обектът не дава достъп до променливи).

Script езиците с Automation поддръжка притежават run-time scripting machine (V.Basic например), посредник между script интерпретатора и Automation сървъра.

Visual C++ има обвиващи класове и прави лесен hosting (лесно вграждането) на Automation сървърен обект, както ще покажем в края на темата.

Базов интерфейс е **IDispatch**.

Чрез него OLE Autom.сървъра експонира свои properties (данни за настройка – напр. цвят на фон, шрифт и т.н.) и методи. Клиентските програми се наричат OLE Automation контролери:



OLE Automation server

OLE Automation controller

обща схема на програмната среда

Интерфейсът IDispatch

Всеки OLE Automation сървър следва да реализира този интерфейс. За да се разбере дали един COM обект поддържа OLE автоматизация, OLE контролерът следва да запита по стандартния начин с QueryInterface() за IDispatch.

Технологията е езиково независима. Следователно данните които се обменят между контролер и сървър следва да са в независим формат.

нови типове за данни

BSTR

подобен на C, като се предхожда от 4 байтово поле съдържащо дължината:

Брой байтове, не букви

UNICODE кодировка

6	a	b	c	d	e	f	\0
---	---	---	---	---	---	---	----

Windows NT предоставя алтернативни API функции за работа с BSTR:

за създаване на BSTR и попълване с низ;
за промяна;

за освобождаване;
за обработки с дължината.

MFC има също методи за преобразуване BSTR $\longleftrightarrow$ CString, напр:

```
Cstring string = (LPCWSTR) bstr;
```

или:

```
CString str = _T("Hello");
```

```
BSTR bstr = str.AllocSysString();
```

VARIANT структурата се използва при трансформиране и предаване на различни типове данни. Всъщност това е самоописващ се тип данни:

```
typedef struct tagVARIANT VARIANT;
```

```
typedef struct tagVARIANT
```

```
{
```

```
    VARTYPE vt;
```

//флагове, идентифициращи типа на данните

```
    union
```

```
    {
```

//стойността

```
    short iVal; long lVal; float fltVal; double dblVal; VARIANT_BOOL bool;
```

```
    SCODE scode; DATE date; BSTR bstrVal; IUnknown FAR * punkVal;
```

```
    IDispatch FAR* pdispVal; SAFEARRAY FAR* parray;
```

//полема, използвани при предаване на данни по указател

```
    short FAR* piVal; long FAR* plVal; float FAR* pfltVal;
```

```
    double FAR* pdblVal; VARIANT_BOOL FAR* pbool;
```

```
    DATE FAR* pdate; BSTR FAR* pbstrVal; IUnknown FAR* FAR* ppunkVal;
```

```
    IDispatch FAR* FAR* ppdispVal; VARIANT FAR* pvarVal; void FAR* byref;
```

```
};
```


За всяко от полетатата на VARIANT има съответстващ идентификатор:

VT_I4	4-byte signed int
VT_R4	4-byte real
VT_R8	8-byte real
VT_DATE	date
VT_BSTR	binary string
VT_DISPATCH	IDispatch FAR*
VT_ERROR	SCODE
VT_BOOL	True=-1, False=0
VT_UNKNOWN	IUnknown FAR*
VT_UI1	unsigned char
VT_LPSTR	null-terminated string

Има API функции за работа с VARIANT структури (например **::VariantInit()**).

При автоматизация задължително се работи с данни от представими във VARIANT типове COM управлява добре тези типове, така че не са нужни proxy / stub код в отделна DLL библиотека (както е при out of process сървърите).

VARIANT позволява предаване на всякакъв вид данни.

IDispatch използва обекти VARIANT за обмен на данни с /от OLE Automation сървъри.

чрез тази структура могат да се предават: числа без знак 1 байт, указатели към беззнакови 1 байт букви, цели, указатели към тях, 4 байта цели, указатели към тях, IEEE реални и указатели към тях (4 и 8 байта), BSTR и указател към него, булеви и указатели към тях, дата и час, указател към обект поддържащ IDispatch, или към IUnknown, масив от данни тип <anything> и т.н. Предвидени са функции за работа с обект VARIANT. Те фактически работят коректно с всякакъв тип в VARIANT и извършват коректно нужните преобразувания и обработки над данните д него.

тип **SAFEARRAY**

Тип данни за масив използван в автоматизацията.

API има множество функции (започват с SafeArray). MFC има поддържащ клас за работа с тези структури – COleSafeArray.

Ето как е дефиниран типа **SAFEARRAY** :

typedef struct tagSAFEARRAY

```
{  
    USHORT    cDims;           //размерности на масива  
    USHORT    fFeatures;       //Flags описващи масива  
    USHORT    cbElements;      // размер на всеки елемент  
    HANDLE    handle;          //HGLOBAL към масива  
    void *    pvData;  
    SAFEARRAYBOUND    rgsabound[]; // граница за всяка размерност  
} SAFEARRAY;
```

typedef struct tagSAFEARRAYBOUND

```
{  
    ULONG    cElements;        // брой на елементи във всеки размер  
    long    LBound;            //долна граница за всяка размерност  
} SAFEARRAYBOUND
```

IDispatch

Основната функция, която вика метод или пропърти, използвайки за целта идентификатори на DISPATCH (**DISPID**) е *IDispatch::Invoke*.

В момента, когато контролера получи указател към този интерфейс (`dispinterface`), то той притежава указател към реализацията на *IDispatch* с която може да има достъп до набора `dispid`, специфични за реализацията.

Когато в кода се повика *IDispatch::Invoke*, действителния преход към метод или пропърти се определя **едва в run time**.

По-долу следват методите на интерфейса:

```
interface IDispatch : public IUnknown
{
public:
HRESULT GetTypeInfoCount(....);           /* използва се да изработи броя
                                              ITypeInfo интерфейси (0 или 1)
                                              поддържани от сървъра. */

HRESULT GetTypeInfo(...);               /* използва се да изработи указател към
                                              ITypeInfo, който е нужен за определяне
                                              дали сървърът поддържа OLE Automation*/

HRESULT GetIDsOfNames(...);           /* преобразува име на метод, пропърти
                                              в DISPID (всъщност индекс към името,
                                              използван при обръщенията към IDispatch) */

HRESULT Invoke(...);                 /* същинска функция. Тя се използва за
                                              достъп до пропърти или метод
                                              експониран от OLE Automation сървъра. Чрез нея
                                              се разпитва сървъра за експонираните си интерфейси */
};
```

В автоматизацията (или чрез IDispatch) винаги се работи не с имена а с dispatch ID (dispid), които се изработват от GetIDsOfNames(), преди Invoke().

Всъщност при всяко повикване по име на метод или пропърти става следното:

1. изработва се CLSID за обекта (от името му) – за целта има API ф-ия `::CLSIDFromProgID()`;
2. вика се `CoCreateInstance()` за създаване на обекта;
3. получава се указател към `IDispatch`;
4. изработват се DISPID за данните членове с `GetIDsOfNames()`;
5. подготвя се масив с аргументите на повикване от тип `VARIANTARG`;
6. обръщение към метода `Invoke()` с подадени параметри (указва се дали метод или property);
7. резултатът, който се получава е в тип `VARIANT` структура.

Защо изобщо е необходим механизма с IDispatch , а не конвенционалния с интерфейсите. Обяснението е следното:

Скриптовите и интерпретаторните езици не поддържат работа с v-tables и интерфейси. Следователно и с COM интерфейсите.

C++ прави това като разчита интерфейсните дефиниции в заглавния файл, описващ обекта (файл с разширение .odl). Това обаче не могат скриптовите езици. Те се нуждаят от механизъм, прехвърлящ преобразуването на име на метод към указател, сочещ функцията , която го реализира.

Този механизъм се базира на универсален интерфейс IDispatch.

Следователно преобразуванията име → указател в COM се извършват от викация, през виртуални метод таблици, инстанциирани по време на компилация, докато при автоматизацията → през индексни таблици в самия обект в run-time.

Това е тъй наречения метод на късно свързване (late binding механизъм).



Късно свързване се поддържа от скриптовите и интерпретируемите езици.

Например във Visual Basic повиквания за IDispatch има неявно винаги, когато променлива се декларира от тип Object:

```
Dim myobj as Object  
set myobj = CreateObject(TV.Remote")  
myobj.interface_method
```

Същото в ASP става , когато се използва COM обектът Server:

```
Dim myobj  
set myobj = Server.Create("TV.Remote")  
myobj.interface_method
```

Забележка:

В този механизъм **не се използва файл, наречен типова библиотека (.tlb, или .odl)** . В него са описани всички интерфейси, методи и пропъртите на COM обекта, както и параметрите на повикване на всеки метод. Типовата библиотека е регистрирана в регистъра, така че клиентът черпи информация от нея за COM обекта.

Типова библиотека се създава с COM API функции, или от IDL файл. Компиляторът от MIDL (Microsoft Interface Definition Language), който е час от VC++ пакета, чете такъв файл и при компилация изработва типова библиотека. VISUAL C++ създава файл ODL към проекта при указание за създаване на Automation сървър и там описва обекта и интерфейсите му. Помощна програма MTypeLib компилира този файл и генерира типова библиотека.

Другият метод на свързване е **ранното свързване (early binding)** – типичен за C++. Там преобразуванията за достигане до указател за метод (работи се с интерфейси) се изпълняват по време на компилация.

Работи се с v-tables. Обектът може да не поддържа IDispatch, а само интерфейси.

Ако има и IDispatch, то следователно имаме **двоен интерфейс** (в един интерфейс има и методи на IUnknown и на IDispatch + специфичните му методи). Не се създава DISPID за достъп до методи и свойства, а се работи с виртуални таблици. Може да се работи и с IDispatch и ако са нужни DISPID, се работи с информацията от типовата библиотека. Обекти с двойни интерфейси поддържат ранно и късно свързване.

В MFC такива интерфейси се създават трудно. Препоръчва се използването на алтернативната класова йерария ATL (Active Template Library).

Има и трети начин на свързване (междинен) – ID binding:

Всички DISPID към свойства и методи могат да се получат от типовата библиотека с която се свързваме в compile time. Следователно не е нужно повикване на GetIDsOfNames() за всяко повикване. Имаме само 1 повикване на GetIDsOfNames(), както в early binding за изработване на CLSID за свързване с обекта (всъщност с типовата му библиотека) в compile time. Invoke() повикванията са в run time и изискват GetIDsOfNames() обръщения, но от типовата библиотека.

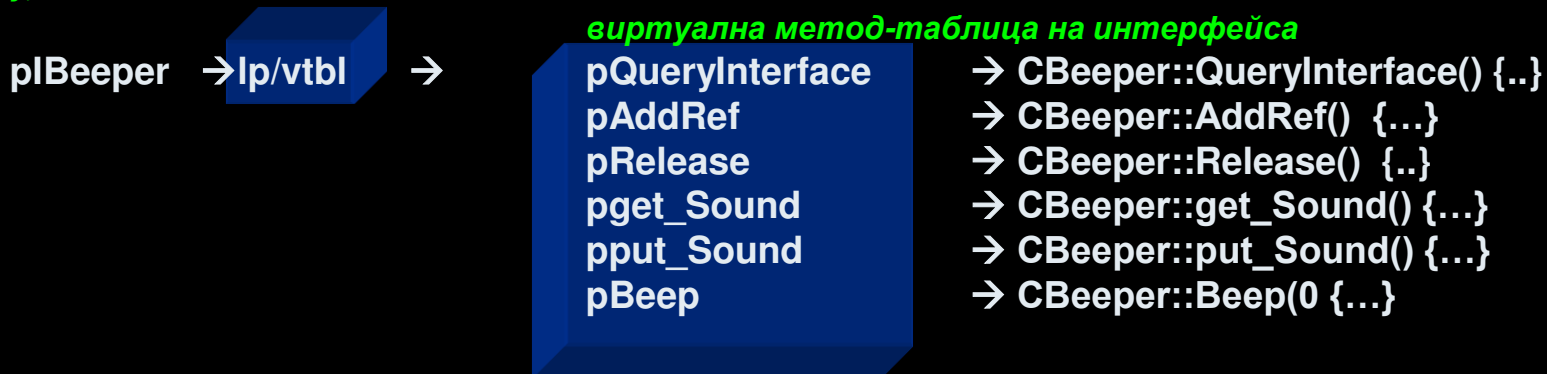
Методът е поне 2 пъти по-бърз от late binding.

Разликите между ранното и късното свързване могат да се пояснят в следния пример: Нека имаме прост обект именован Beeper (клаксон). Той притежава един даннов елемент (property) и метод с име Beep(). Реализацията на този обект нека е на C++, като експонираната му функционалност става с помощта на OLE механизмите:

```
class CBeeper
{ public:
    long      m_ISound;           // това е данновия експониран член
public:
    long      Beep(void);         // това е метода на класа
};
```

дефинираме интерфейс, например с име IBeeper.

```
interface IBeeper : IUnknown
{
    long get_Sound( void);
    void put_Sound( long lSound);
    long Beep( void);
};
```



ограниченията с **виртуалните таблици**, каквато е тази за `IBeeper`, са че всеки клиент на този обект следва да се привърже (binding) към интерфейсите му методи през указатели.

Т.е. ред от клиентски код с обръщение `pIBeeper → Beep()` например, трябва да може да се компилира в `call` инструкция на процесора с параметър – отместване спрямо началото на виртуалната таблица (или все едно спрямо `pIBeeper`).

Тази техника работи лесно и бързо за компилируем код, но не и за интерпретируем.

В този случай достъп до функционалността на обекта става на основата на късното свързване (**late binding**) **чрез посредничеството на `dispinterface`**. В неговото описание (по-долу има пример на такова) в `odl` файл, пропъртите и методи се съпоставят на собствени идентификатори (заградените в прави скоби), използвани пряко за връзката. Следва дефиниция на `dispinterface` за нашия обект:

```
dispinterface DIBeeper
{
properties:
    [id(0)]    long Sound;
methods:
    [id(1)]    long Beep(void);
};
```

За пропъртито `Sound` беше съпоставен `dispID` нула, а за метода `Beep()` – `dispID` единица. Контролерът (клиентът) използва тези `dispID` в `run-time` за да насочва точно повикванията към методи и пропъртите към мястото на тяхната реализация в обекта:

флаг определя
операция над пропъртито
`get` или `set`



Изработването на интерфейсия указател към `IDispatch` се подчинява на общите механизми в `COM` (през `QueryInterface`). Тогава се реализира и преобразуването от `pDisp` към `dispatch` таблицата (т.е. всичко това става както в `early binding`). Късното свързване намира място едва при намиране съответствие за пропърти или метод от `dispatch` таблицата, което се решава в `run-time`.

Процедура на създаване на OLE Automation server с използване на MFC класове

В MFC поддръжката за Automation (интерфейс и специфичните типове се конструират автоматично) е предвиден клас COleDispatchImpl, чиято инстанция се реализира още в CCmdTarget::EnableAutomation(). Т.е всеки MFC клас, поддържащ автоматизация следва да наследи CCmdTarget и да повика горната функция още в конструктора.

Преобразуванията:

име	→	DISPID
метод/свойство		

нужни за повикванията през Invoke() стават с **dispatch карта**:

BEGIN_DISPATCH_MAP (клас, родител)

DISP_FUNCTION (клас, име на dispatch метода, име на функция с която е реализиран, ID (от полетата на VARIANT) за параметрите с които тя се вика)

DISP_PROPERTY_EX (клас, име на свойството, методи Get.. Set.. на класа с които се обработва свойството)

END_DISPATCH_MAP()

В картата (генерирана от Class Wizard) липсват Dispid. Всъщност има позиционна зависимост: място в картата → номер на Dispid в ODL генерирания файл.

примерен OLE сървър (AutoBub), който изобразява в прозорец окръжности

(този OLE сървър има свое изображение).

1. В AppWizard създавате ново SDI приложение с това име. Указвате “support OLE Automation”. При диалозите на AppWizard, в поле File Type ID , въвеждате ProgID на Automation обекта.
2. избирате CAutoBubDoc (само 1 клас има поддръжка на automation).
3. правите добавки в класа, нужни за нормалната функционалност: структура с данните за отделна окръжност, функциите за добавяне, броене и извличане на окръжност (AddBubble(), GetBubble(), GetCount()) като прототипи и реализация.
4. Създава се диалогов прозорец (с ресурсен редактор) за описване параметри на нова окръжност (.rc). Задават се идентификатори на диалоговия прозорец и на отделните контроли в него.

5. Създава се CAddBubDlg, наследник на CDialog, свързан с идентификатора на ресурса на създадения диалогов прозорец.

6. добавя се меню опция “Add bubble”. Свързва се с обработваща меню-опцията функция – тя очевидно е метод на CAutoBubView:

```
void CAutoBubView :: OnMenuAddBubble()
{
// взима от въведените в ресурсните полета стойности и инициализира
// променливи от структурата на окръжността. След това вика: pDoc→AddBubble() ;
    Invalidate();
....
}
```

прерисуване на окръжности:

```
void CAutoBubView::OnDraw(CDC* pDC)
{
...
}
```

Дотук приложението е готово за компилация като самостоятелно приложение.

Процедура на добавяне на OLE Automation поддръжка към приложението - сървър

1. Class Wizard → Automation → Add Property с цел добавяне на нови даннови (get/set) променливи на automation обекта (CAutoBubDoc):

Implementation	тип	external name	Get функция	Set функция
Get/Set	long	Count	CountIs()	none

Функцията става метод на документния клас и следва да се опише.

Свойствата могат да бъдат не само get/set , а и прости член-променливи. Тогава даденото automation свойство става достъпно като даннов елемент на класа. Променливата ще бъде добавена в картата. В поле Notification function (ако сме избрали member-variable в диалог на развойната среда) може да се укаже функция, която се активира при присвояване от клиент на нова стойност на пропъртито. ClassWizard автоматично добавя записи в картата, декларации на функциите в класа и промени в ODL файла (както по-долу). **За get/set – записът е DISP_PROPERTY_EX; за член променливи – DISP_PROPERTY.**

2. добавяме и методи към класа. За всеки метод имаме: External Name – името на Automation метода и Intenal Name – името на съответната member функция. В полето Return Type може да се постави всеки Automation съвместим даннов тип. Параметрите на метода се указват в Parameter List. MFC се занимава с преобразуването им в VARIANTARG и обратно за да е коректно повикването на Invoke().

(За случая такива методи могат да са напр. Clear, Add. Те стават методи на документния клас и следва да се опишат функционално.)

За всеки добавен метод Class Wizard прави:

- 1 добавя декларация в .h файла на класа**
- 2. реализира празна функция във файла**
- 3. добавя ред в dispatch картата на класа**
- 4. добавя метода и disp_id в ODL файла на проекта.**

Код за тези ф-ии следва да се добави в CAutoBubDoc.

Файлът .odl се създава автоматично от Class Wizard:

```

// AutoBub.odl : type library source for AutoBub.exe
[ uuid(06A6EDEB-A2FD-11CF-9C7D-000000000000),
  version(1.0) ]
library AutoBub
{
    importlib("stdole32.tlb");
    // Primary dispatch interface for CAutoBubDoc
    [ uuid(06A6EDEC-A2FD-11CF-9C7D-000000000000) ]
    dispinterface IAutoBub
    {
        properties:
            [id(1)]    {{{AFX_ODL_PROP(CAutoBubDoc)
                        long Count;
                        }}}AFX_ODL_PROP

        methods:
            [id(2)]    {{{AFX_ODL_METHOD(CAutoBubDoc)
                        void Clear();
                        [id(3)] void Add(long clr, short nYPos, short nXPos, short nRadius);
                        }}}AFX_ODL_METHOD

    };
    // Class information for CAutoBubDoc
    [ uuid(06A6EDEA-A2FD-11CF-9C7D-000000000000) ]
    coclass CAutoBubDoc
    {
        [default] dispinterface IAutoBub;
    };
    {{{AFX_APPEND_ODL}}}
};

```

Последната промяна в кода на Automation сървъра е с цел изобразяване при инстанциране от страна на контролер.

Търсим CAutoBubApp::InitInstance().

там където се проверява дали приложението е стартирано като OLE server или като самостоятелно приложение се добавя код за регистриране на временен сървър , което позволява външни приложения да поискат поддръжка от OLE библиотеките за да създават OLE обекти от този тип в рамките на своето приложение. Ето част от кода:

```
BOOL CAutoBubApp::InitInstance()  
{  
.....  
// проверка дали е стартирано като OLE сървър  
if(cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated)  
{           COleTemplateServer::RegisterAll();           // регистрира като running всички OLE  
                                                    // factories за сървъри. Това позволява  
                                                    // създаване на обекти от такъв тип  
                                                    // от други приложения.  
  
// следва ваш код проверяващ дали има опция за /Automation при стартиране на  
// програмата. Ако да – изобразява главния прозорец на този клас  
if( !ProcessShellCommand(cmdInfo))  
    return FALSE;  
return TRUE;} /* следва автоматично добавен код */.....  
}
```

С това OLE Automation сървър е готов.

Създаване на Automation controller

1. На Visual C++

С MFC Automation клиент се пише лесно. COleDispatchDriver осигурява извличане на dispatch указатели и методи за достъп до методи и свойства.

Създават се производни на горния класове. С тях достъпът до automation методите и свойства е елементарен.

За целта в средата на **ClassWizard** избираме **AddClass**, после **From a Type Library**. Там указваме типовата библиотека за конкретния сървър. Генерира се клас съдържащ функции за извикване на методите на сървъра (със същите имена) и Get/Set функции за достъп до пропъртитата му.

Например ако сървърът е CAutoMath с Prog ID “Math.Object”, то в контейнера следва да има код от вида:

```
CAutoMath math;  
math.CreateDispatch(_T(“Math.Object”));  
// създава Automation обект и IDispatch указател в променливата m_lpDispatch  
// оттук обръщания към методи и свойства автоматично се превеждат през IDispatch  
// повиквания:  
int sum = math.Sum(2,4);           // достъп до метод  
double pi = math.GetPi();          // достъп до пропърти
```

2. На Visual Basic

Може да се създаде приложение, което управлява AutoBub през неговия OLE Automation интерфейс.

1. Създаваме си начален диалог за въвеждане на данни за окръжностите, които се предават към сървъра след активирането му.

2. Декларираме обект – носител на сървъра (напр. обект с име BubbleMachine):

Dim BubbleMachine As Object

3. Създава се инстанция на AutoBub сървъра:

set BubbleMachine = CreateObject("AutoBub.Document")

след края на използването на сървъра той се освобождава:

Set BubbleMachine = Nothing

4. Описва се процедура, активираща се например от VB меню-опция “Add Bubble”, която приема стойностите въведени чрез диалога и ги подава на BubbleMachine:

Private Sub IDC_ADD_Click()

‘ чете въведените в полета стойности например за цвят, координати и радиус

‘ формира

ColorRef = RGB(x,y,z)

‘ и други

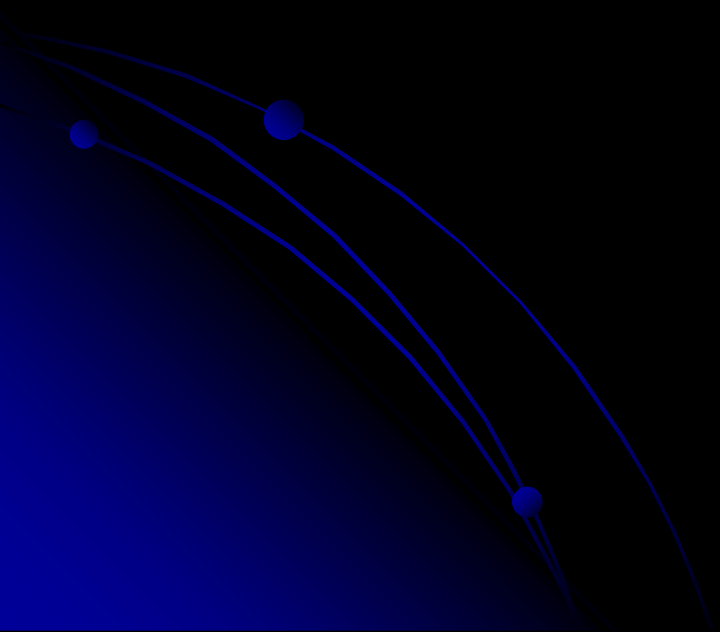
‘ създава новата окръжност:

BubbleMachine.Add ColorRef, коорд_х, коорд_у, радиус

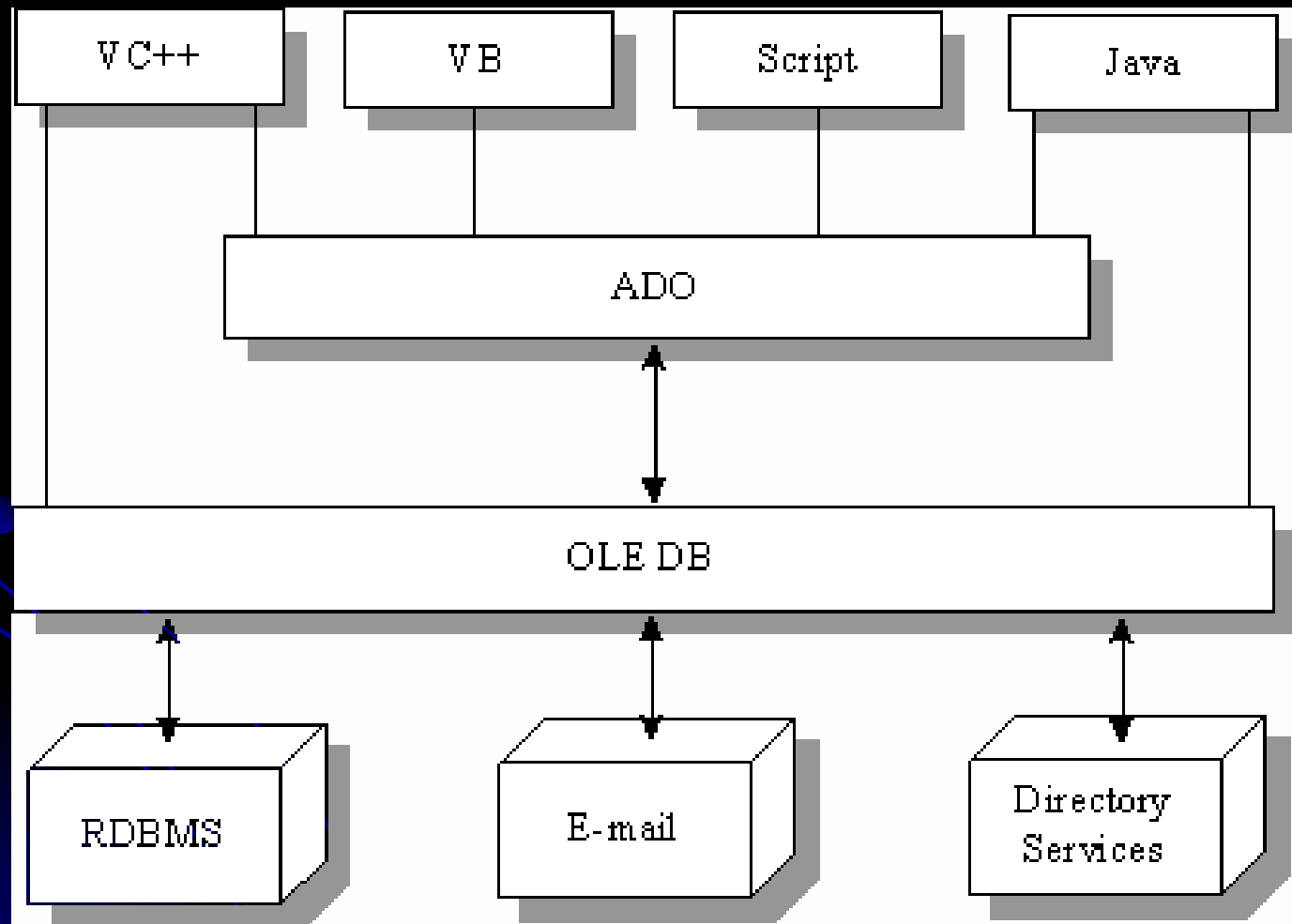
използва се функция от сървъра.

End Sub

OLE Automation II



OLE DB и ADO



Пример за използване на OLE DB

```
using System;  
using System.Data;  
using System.Data.OleDb;  
using System.Xml.Serialization;
```

```
public class MainClass {  
    public static void Main ()  
    {  
        // Set Access connection and select strings.  
        // The path to BugTypes.MDB must be changed if you build  
        // the sample from the command line:  
#if USINGPROJECTSYSTEM  
        string strAccessConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=..\..\BugTypes.MDB";  
#else  
        string strAccessConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=BugTypes.MDB";  
#endif  
        string strAccessSelect = "SELECT * FROM Categories";  
  
        // Create the dataset and add the Categories table to it:  
        DataSet myDataSet = new DataSet();  
        OleDbConnection myAccessConn = null;  
        try  
        {  
            myAccessConn = new OleDbConnection(strAccessConn);  
        }  
        catch(Exception ex)  
        {  
            Console.WriteLine("Error: Failed to create a database connection. \n{0}", ex.Message);  
            return;  
        }  
    }  
}
```

```
try
{
    OleDbCommand myAccessCommand = new OleDbCommand(strAccessSelect,myAccessConn);
    OleDbDataAdapter myDataAdapter = new OleDbDataAdapter(myAccessCommand);

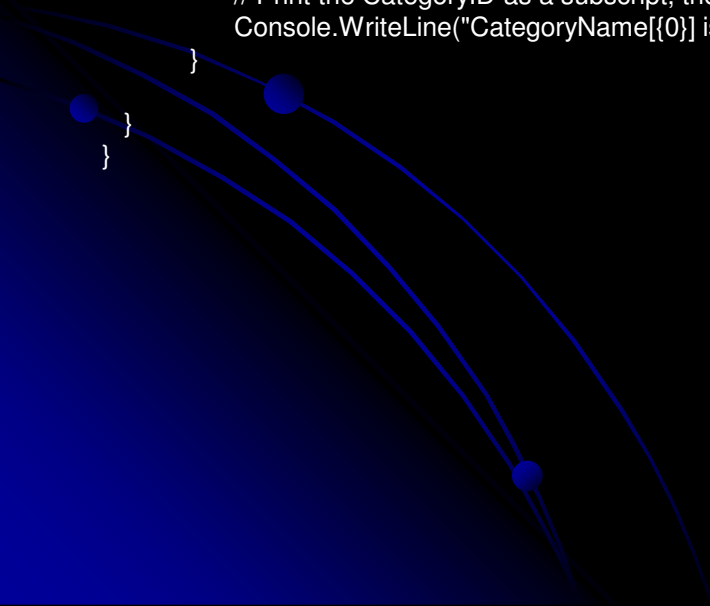
    myAccessConn.Open();
    myDataAdapter.Fill(myDataSet,"Categories");
}
catch (Exception ex)
{
    Console.WriteLine("Error: Failed to retrieve the required data from the DataBase.\n{0}",
ex.Message);
    return;
}
finally
{
    myAccessConn.Close();
}

// A dataset can contain multiple tables, so let's get them
// all into an array:
DataTableCollection dta = myDataSet.Tables;
foreach (DataTable dt in dta)
{
    Console.WriteLine("Found data table {0}", dt.TableName);
}
```

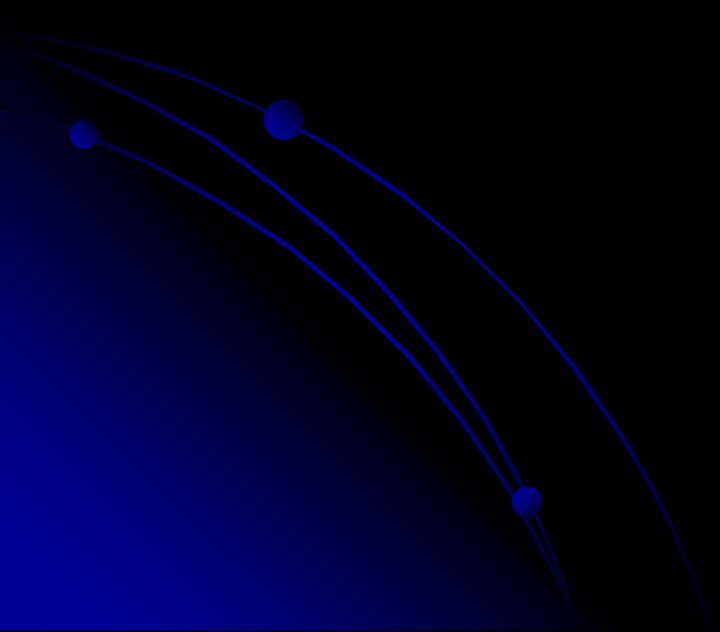
```

// The next two lines show two different ways you can get the
// count of tables in a dataset:
Console.WriteLine("{0} tables in data set", myDataSet.Tables.Count);
Console.WriteLine("{0} tables in data set", dta.Count);
// The next several lines show how to get information on
// a specific table by name from the dataset:
Console.WriteLine("{0} rows in Categories table", myDataSet.Tables["Categories"].Rows.Count);
// The column info is automatically fetched from the database,
// so we can read it here:
Console.WriteLine("{0} columns in Categories table", myDataSet.Tables["Categories"].Columns.Count);
DataColumnCollection drc = myDataSet.Tables["Categories"].Columns;
int i = 0;
foreach (DataColumn dc in drc)
{
    // Print the column subscript, then the column's name
    // and its data type:
    Console.WriteLine("Column name[{0}] is {1}, of type {2}", i++, dc.ColumnName, dc.DataType);
}
DataRowCollection dra = myDataSet.Tables["Categories"].Rows;
foreach (DataRow dr in dra)
{
    // Print the CategoryID as a subscript, then the CategoryName:
    Console.WriteLine("CategoryName[{0}] is {1}", dr[0], dr[1]);
}
}
}

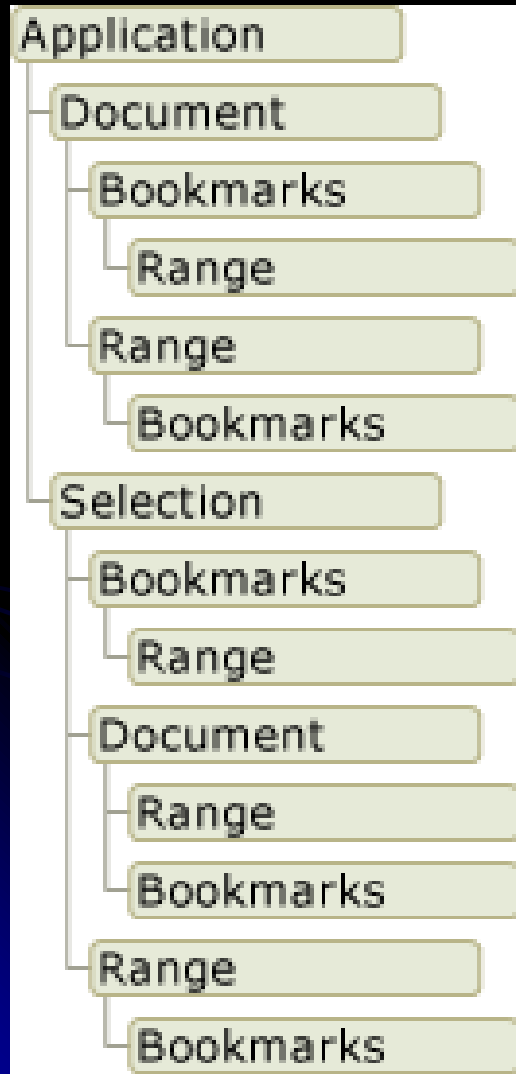
```



OLE & Office



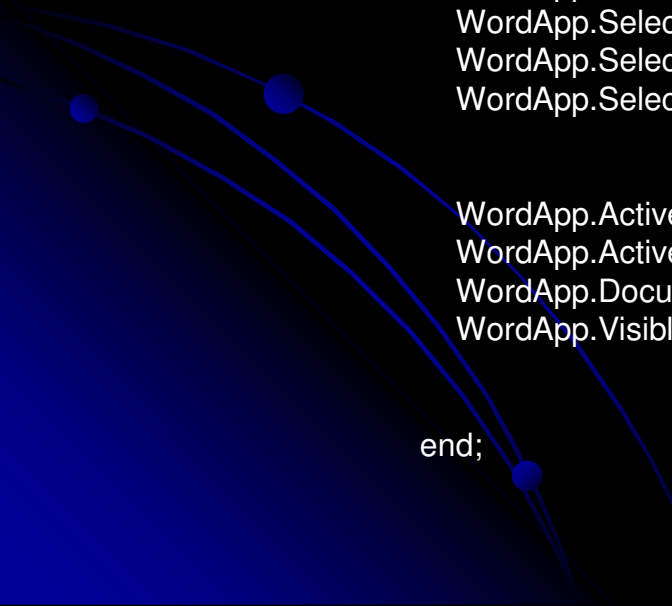
Word Object Model



Пример - Word

Replace - Delphi

```
var WordApp:Olevariant;  
    yourDocFileName:String;  
    re:Variant;  
    cnt,i:integer;  
  
begin  
  
    yourDocFileName:=Edit4.text;  
    WordApp := CreateOLEObject('Word.Application');  
    WordApp.Documents.Open(yourDocFileName);  
  
    WordApp.Selection.Find.ClearFormatting;  
    WordApp.Selection.Find.Text :='#pNAME#';  
    WordApp.Selection.Find.Replacement.Text :=Edit1.Text;  
    WordApp.Selection.Find.Execute(Replace := wdReplaceAll);  
  
    WordApp.ActiveDocument.SaveAs(Edit5.Text);  
    WordApp.ActiveDocument.Close;  
    WordApp.Documents.Open(Edit5.Text);  
    WordApp.Visible := True;  
  
end;
```



Пример Word - Macros

VB

```
Dim myStoryRange As Range

For Each myStoryRange In ActiveDocument.StoryRanges
    With myStoryRange.Find
        .Text = "#pName#"
        .Replacement.Text = "Petur"
        .Wrap = wdFindContinue
        .Execute Replace:=wdReplaceAll
    End With
    Do While Not (myStoryRange.NextStoryRange Is Nothing)
        Set myStoryRange = myStoryRange.NextStoryRange
        With myStoryRange.Find
            .Text = "Edit2"
            .Replacement.Text = "Lisabon"
            .Wrap = wdFindContinue
            .Execute Replace:=wdReplaceAll
        End With
    Loop
Next myStoryRange
End Sub
```


Excel Object Model



Outlook



Пример Excel

Създаване на файл

```
Excel.Application xlApp ;
    Excel.Workbook xlWorkbook ;
    Excel.Worksheet xlWorkSheet ;
    object misValue = System.Reflection.Missing.Value;

    xlApp = new Excel.ApplicationClass();
    xlWorkbook = xlApp.Workbooks.Add(misValue);

    xlWorkSheet = (Excel.Worksheet)xlWorkbook.Worksheets.get_Item(1);
    xlWorkSheet.Cells[1, 1] = "http://csharp.net-informations.com";

    xlWorkbook.SaveAs("csharp-Excel.xls", Excel.XlFileFormat.xlWorkbookNormal,
        misValue, misValue, misValue, misValue, Excel.XlSaveAsAccessMode.xlExclusive,
        misValue, misValue, misValue, misValue, misValue);
    xlWorkbook.Close(true, misValue, misValue);
    xlApp.Quit();

    releaseObject(xlWorkSheet);
    releaseObject(xlWorkbook);
    releaseObject(xlApp);

    MessageBox.Show("Excel file created , you can find the file c:\\csharp-Excel.xls");
```

форматиране

```
Excel.Application xlApp ;  
    Excel.Workbook xlWorkBook ;  
    Excel.Worksheet xlWorkSheet ;  
    object misValue = System.Reflection.Missing.Value;  
    Excel.Range chartRange ;  
  
    xlApp = new Excel.ApplicationClass();  
    xlWorkBook = xlApp.Workbooks.Add(misValue);  
    xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(1);  
  
    //add data  
    xlWorkSheet.Cells[4, 2] = "";  
    xlWorkSheet.Cells[4, 3] = "Student1";  
    xlWorkSheet.Cells[4, 4] = "Student2";  
    xlWorkSheet.Cells[4, 5] = "Student3";  
  
    xlWorkSheet.Cells[5, 2] = "Term1";  
    xlWorkSheet.Cells[5, 3] = "80";  
    xlWorkSheet.Cells[5, 4] = "65";  
    xlWorkSheet.Cells[5, 5] = "45";  
  
    xlWorkSheet.Cells[6, 2] = "Term2";  
    xlWorkSheet.Cells[6, 3] = "78";  
    xlWorkSheet.Cells[6, 4] = "72";  
    xlWorkSheet.Cells[6, 5] = "60";
```



.....

```
xlWorksheet.get_Range("b2", "e3").Merge(false);
```

```
chartRange = xlWorksheet.get_Range("b2", "e3");  
chartRange.FormulaR1C1 = "MARK LIST";  
chartRange.HorizontalAlignment = 3;  
chartRange.VerticalAlignment = 3;
```

```
chartRange = xlWorksheet.get_Range("b4", "e4");  
chartRange.Font.Bold = true;  
chartRange = xlWorksheet.get_Range("b9", "e9");  
chartRange.Font.Bold = true;
```

```
chartRange = xlWorksheet.get_Range("b2", "e9");  
chartRange.BorderAround(Excel.XlLineStyle.xlContinuous, Excel.XlBorderWeight.  
    xlMedium, Excel.XlColorIndex.xlColorIndexAutomatic, Excel.  
    XlColorIndex.xlColorIndexAutomatic);
```

```
xlWorkbook.SaveAs("csharp.net-informations.xls", Excel.XlFileFormat.  
    xlWorkbookNormal, misValue, misValue, misValue, misValue,  
Excel.XlSaveAsAccessMode.  
    xlExclusive, misValue, misValue, misValue, misValue, misValue);  
xlWorkbook.Close(true, misValue, misValue);  
xlApp.Quit();
```

Вмъкване на изображение

```
Excel.Application xlApp ;
    Excel.Workbook xlWorkBook ;
    Excel.Worksheet xlWorkSheet ;
    object misValue = System.Reflection.Missing.Value;

    xlApp = new Excel.ApplicationClass();
    xlWorkBook = xlApp.Workbooks.Add(misValue);
    xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(1);

    //add some text
    xlWorkSheet.Cells[1, 1] = "http://csharp.net-informations.com";
    xlWorkSheet.Cells[2, 1] = "Adding picture in Excel File";

    xlWorkSheet.Shapes.AddPicture("C:\\csharp-xl-picture.JPG", Microsoft.Office.
        Core.MsoTriState.msoFalse, Microsoft.Office.Core.MsoTriState.msoCTrue, 50, 50,
300, 45);

    xlWorkBook.SaveAs("csharp.net-informations.xls", Excel.XlFileFormat.
        xlWorkbookNormal, misValue, misValue, misValue, misValue,
Excel.XlSaveAsAccessMode.
        xlExclusive, misValue, misValue, misValue, misValue, misValue);
    xlWorkBook.Close(true, misValue, misValue);
    xlApp.Quit();

    releaseObject(xlApp);
    releaseObject(xlWorkBook);
    releaseObject(xlWorkSheet);

    MessageBox.Show ("File created !");
```

Създаване на графика

```
Excel.Application xlApp ;  
Excel.Workbook xlWorkBook ;  
Excel.Worksheet xlWorkSheet ;  
object misValue = System.Reflection.Missing.Value;
```

```
xlApp = new Excel.ApplicationClass();  
xlWorkBook = xlApp.Workbooks.Add(misValue);  
xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(1);
```

//add data

```
xlWorkSheet.Cells[1, 1] = "";  
xlWorkSheet.Cells[1, 2] = "Student1";  
xlWorkSheet.Cells[1, 3] = "Student2";  
xlWorkSheet.Cells[1, 4] = "Student3";
```

```
xlWorkSheet.Cells[2, 1] = "Term1";  
xlWorkSheet.Cells[2, 2] = "80";  
xlWorkSheet.Cells[2, 3] = "65";  
xlWorkSheet.Cells[2, 4] = "45";
```

```
xlWorkSheet.Cells[3, 1] = "Term2";  
xlWorkSheet.Cells[3, 2] = "78";  
xlWorkSheet.Cells[3, 3] = "72";  
xlWorkSheet.Cells[3, 4] = "60";
```

```
xlWorkSheet.Cells[4, 1] = "Term3";  
xlWorkSheet.Cells[4, 2] = "82";  
xlWorkSheet.Cells[4, 3] = "80";  
xlWorkSheet.Cells[4, 4] = "65";
```

```
xlWorkSheet.Cells[5, 1] = "Term4";  
xlWorkSheet.Cells[5, 2] = "75";  
xlWorkSheet.Cells[5, 3] = "82";  
xlWorkSheet.Cells[5, 4] = "68";
```

```
Excel.Range chartRange ;
```

```
Excel.ChartObjects xlCharts =  
(Excel.ChartObjects)xlWorkSheet.ChartObjects(Type.Missing);  
Excel.ChartObject myChart = (Excel.ChartObject)xlCharts.Add(10, 80,  
300, 250);  
Excel.Chart chartPage = myChart.Chart;
```

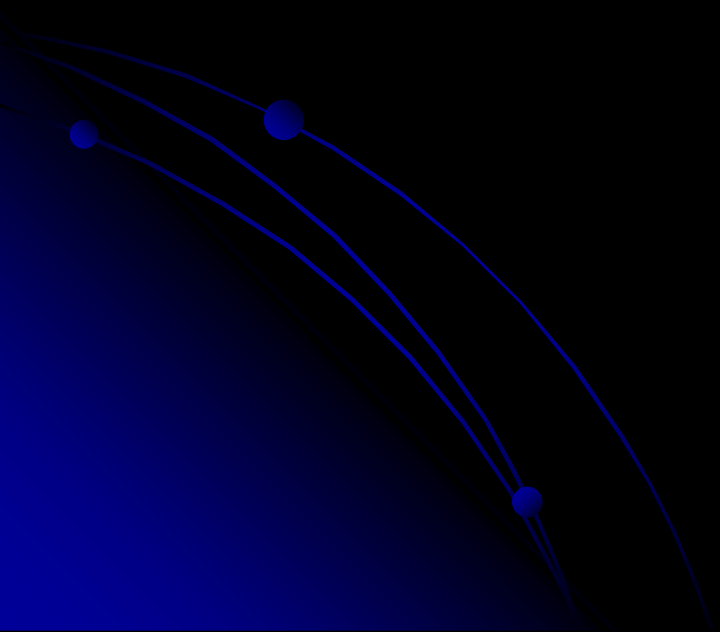
```
chartRange = xlWorkSheet.get_Range("A1", "d5");  
chartPage.SetSourceData(chartRange, misValue);  
chartPage.ChartType = Excel.XlChartType.xlColumnClustered;
```

```
xlWorkBook.SaveAs("csharp.net-informations.xls", Excel.XlFileFormat.  
xlWorkbookNormal, misValue, misValue, misValue,  
misValue, Excel.XlSaveAsAccessMode.  
xlExclusive, misValue, misValue, misValue, misValue,  
misValue);  
xlWorkBook.Close(true, misValue, misValue);  
xlApp.Quit();
```

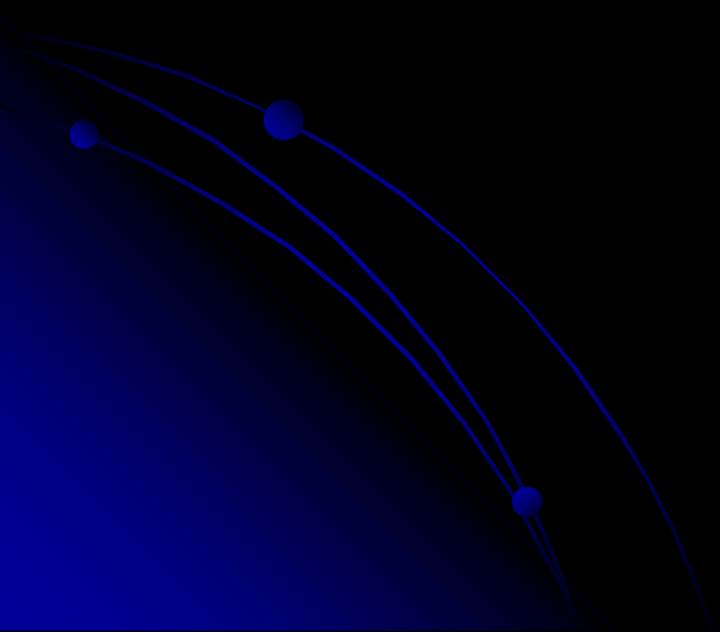
```
releaseObject(xlWorkSheet);  
releaseObject(xlWorkBook);  
releaseObject(xlApp);
```

```
MessageBox.Show("Excel file created , you can find the file  
c:\csharp.net-informations.xls");
```

VM & Ole Automation



.NET Remoting



Съдържание

- Какво е .NET Remoting?
- Remoting инфраструктурата
- Remoting канали и форматери. Регистрация на канал
- Активация на обекти. Активация от сървъра. Активация от клиента
- Маршализация (Marshaling). Marshal-by-Value обекти. Marshal-by-Reference обекти
- Живот на обектите (Lifetime). `ILease`
- Създаване на Remoting сървър и клиент
- Проблемът с общите типове
- Remoting конфигурационен файл

Разпределени приложения

- Повечето днешни приложения са разпределени
 - Състоят се от няколко раздалечени компонента, които си взаимодействат
- Модели за разпределени приложения
 - Модел "клиент/сървър"
 - Модел "разпределени обекти"
 - DCOM – ползва се в Microsoft Windows
 - CORBA – отворен стандарт, доста сложен
 - Java RMI – базира се на Java технологията
 - .NET Remoting – ползва се в .NET Framework
 - Модел "Web услуги"

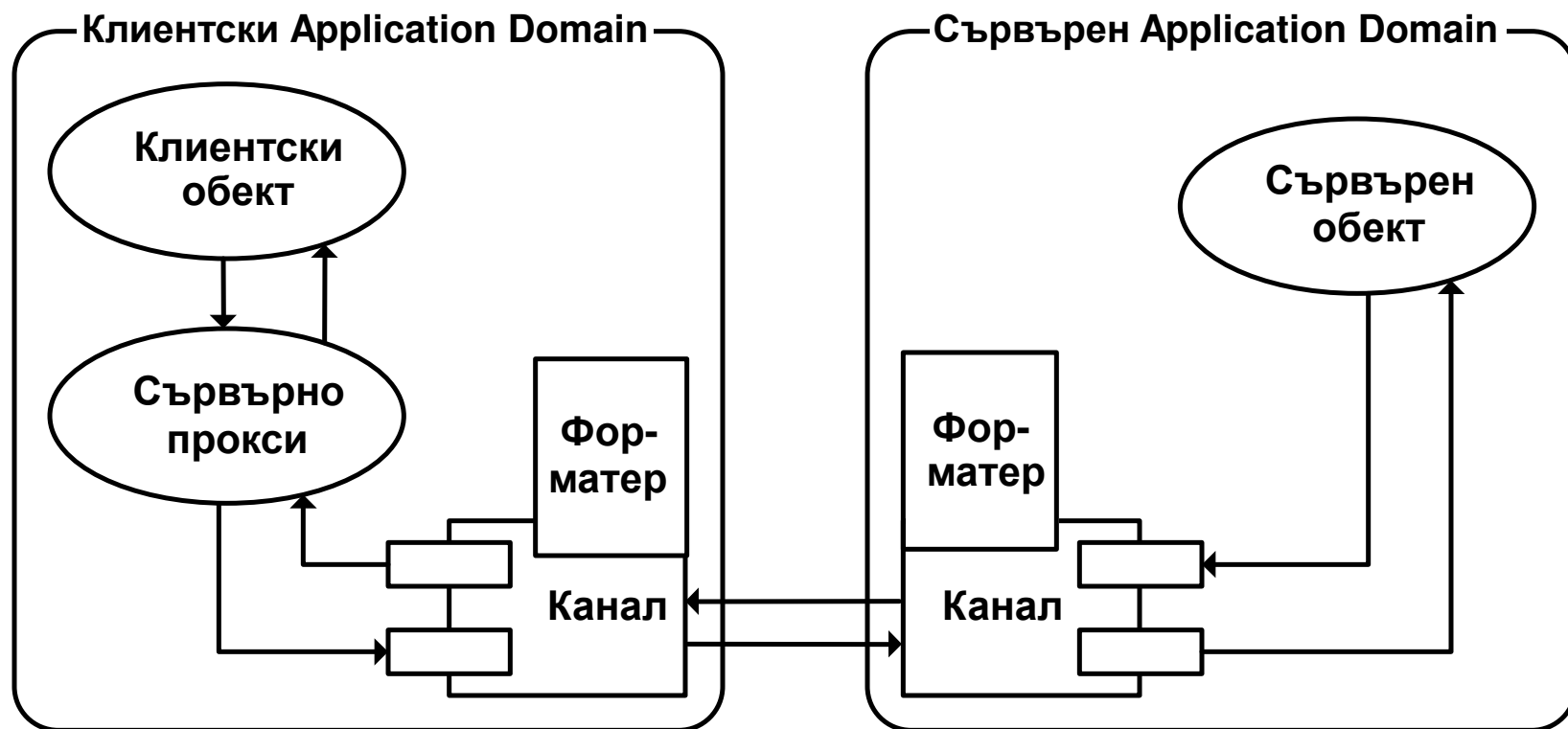
Какво е .NET Remoting?

- Remoting технологията в .NET Framework осигурява прозрачен достъп до отдалечени обекти
 - Обекти от друг application domain
 - Обекти от друг процес
 - Обекти на отдалечена машина
- Използва се за улесняване на комуникацията при разпределени приложения
- .NET Framework предоставя специализирана инфраструктура за

Remoting инфраструктурата

- .NET Remoting инфраструктурата се състои от:
 - Канали – пренасят съобщения от и към отдалечени обекти
 - Форматери – кодират и декодират съобщенията в някакъв формат
 - Прокси класове – предават извикванията на методи към отдалечените обекти
 - Механизми за активация – осигуряват отдалечено инстанциране на обекти
 - Маршализация – осигурява пренос на обекти, техните свойства, полета и т.н.

Remoting инфраструктурата (2)



Remoting канали

- Каналите (channels) в .NET Remoting
 - Транспортират съобщения от и към отдалечени обекти
 - Могат да бъдат:
 - TCP – използват чист TCP сокет
 - HTTP – използват протокола HTTP
 - Други – дефинирани от потребителя
 - Използват се както от клиентските, така и от сървърните приложения
 - Използват TCP портове
 - Трябва да бъдат регистрирани преди

Регистрация на канал

- Регистрацията на Remoting канал може да стане по 2 начина:

- Чрез класа `System.Runtime.Remoting.`

```
using System.Runtime.Remoting.Channels;  
using System.Runtime.Remoting.Channels.Tcp;  
...  
TcpChannel channel = new TcpChannel(3333);  
ChannelServices.RegisterChannel(channel);
```

- `RemotingConfiguration.Configure(`
 "`MyClient.exe.config`");

Форматери

- Форматерите (formatters) форматираят съобщенията, които се предават по Remoting каналите
- Форматерите биват:
 - Бинарен форматер
 - Голяма производителност
 - Слаба съвместимост с други технологии (специфичен за .NET Remoting)
 - SOAP форматер
 - XML базиран формат – добра съвместимост с други технологии

Форматери (2)

- Форматери по подразбиране:
 - HTTP каналите използват по подразбиране SOAP форматер
 - Това позволява използване на отдалечения метод като Web услуга
 - TCP каналите използват по подразбиране бинарен форматер

```
SoapServerFormatterSinkProvider provider =  
    new SoapServerFormatterSinkProvider();  
IDictionary props = new Hashtable();  
props["port"] = 12345;  
TcpChannel chan = new TcpChannel(props, null, provider);
```

Активация на обекти

- "Активация" означава създаване на отдалечен обект (инстанциране) и подготвяне на обекта за използване
- Активацията бива два вида:
 - Активация от страна на сървъра (server-side activation)
 - Активация от страна на клиента (client-side activation)
- Начини за активация:
 - Чрез оператора `new`
 - Чрез `Activator.GetObject(...)`
 - Чрез `Activator.CreateInstance(...)`

Активация на обекти (2)

- При активация на отдалечен обект Remoting инфраструктурата динамично създава прокси клас при клиента
- Прокси класът изглежда като истинския клас (или интерфейс), но препраща обръщанията към отдалечения обект:
 - достъп до полета, свойства, събития и други членове
 - извикване на методи
 - възникналите изключения
- За програмиста отдалечеността на обекта е прозрачна

Активация от сървър

- При активация от сървър (server-side activation)
 - Сървърът създава автоматично обекта при клиентска заявка
- Има два режима на сървърна активация:
 - Single-Call – за всяка клиентска заявка се създава отделна инстанция
 - Singleton – една и съща инстанция обслужва всички клиенти
 - Инстанцията се създава само веднъж
 - Данните на инстанцията са общи (споделени) за всички клиенти

Активация от сървъра (2)

- Сървърът задава за всеки обект, че изисква сървърна активация:
 - Регистриране на Single-Call server activated

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    typeof(CommonTypes.Library), "Library",  
    WellKnownObjectMode.SingleCall);
```

- Регистриране на Singleton server activated

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    typeof(CommonTypes.Library), "Library",  
    WellKnownObjectMode.Singleton);
```

Активация от клиента

- При активация от клиента (client-side activation)
 - Обектите се създават по заявка от клиента (ръчно, изрично)
 - Всяка заявка за създаване на обект създава нова, отделна инстанция
 - Всеки клиент работи със свои собствени обекти на сървъра

```
RemoteConfiguration.RegisterActivatedServiceType(  
    typeof(CommonTypes.Library));
```


Маршализация (Marshaling)

- Маршализацията осигурява пренос на обекти между Remoting клиента и сървъра. Използва при:
 - предаване на параметри
 - връщане на стойност
 - достъп до свойства и полета
- Има 3 типа обекти:
 - Marshal-by-Value обекти – пренасят се по стойност (сериализирани)
 - Marshal-by-Reference обекти – пренасят се по отдалечена референция (ObjRef)
 - Not-Marshaled обекти – не се пренасят

Marshal-by-Value обекти

- Marshal-by-Value обекти
 - Всички сериализирани типове ([Serializable], ISerializable)
 - Пренасят се по стойност (копие от тях се предават до отдалечената машина)
 - При промяна се променя само локалното копие на обекта
 - Всичките им полета и свойства се предават наведнъж – работи бързо

```
public struct Book {  
    public string mAuthor;  
    public string mTitle;  
}
```

Marshal-by-Reference обекти

- Marshal-by-Reference обекти
 - Всички типове, които наследяват класа `System.MarshalByRefObject` (включително сериализирани типове)
 - Пренасят се по отдалечена референция, по която се създава прокси клас
 - Всички операции се пренасят към отдалечения обект през прокси класа
 - Всяко четене или промяна на поле или

свойство предизвиква отдалечено
`public class Book : MarshalByRefObject { ... }`
извикване – лоша производителност

Живот на обектите (Lifetime)

- В .NET Remoting се използва техниката "lease-based lifetime" ("живот, отпуснат назаем")
 - Използва се само при Singleton Server-Activated обекти и Marshal-by-Reference Client-Activated обекти
 - При създаване на обект му се дава "време на живот"
 - След изтичане на това време обектът се унищожава от Garbage Collector-а
 - Времето за живот се настройва при създаване на обекта и не може да се променя след това

Интерфейсът ILease

- InitialLeaseTime – начален живот (5 мин. по подразбиране)
- RenewOnCallTime – удължаване на живота при извикване – (2 мин.)
- SponsorshipTimeout – настройва на т. нар. "спонсори" (2 мин.)
- CurrentLeaseTime – оставащо време
- CurrentState – състояние на времето за живот (Initial, Active, Expired, Renewing)

```
ClientActivatedType caObject = new ClientActivatedType();  
ILease serverLease = (ILease)  
    RemotingServices.GetLifetimeService(caObject);  
serverLease.RenewOnCallTime =  
    new TimeSpan(0, 10, 0); // 10 min.
```

Създаване на Remoting сървър

1. Регистриране на Remoting канал

```
TcpChannel channel = new TcpChannel(12345);  
ChannelServices.RegisterChannel(channel);
```

2. Регистриране на обектите, които ще
бъдат достъпни отдалечено

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    typeof(CommonTypes.Library), "Library",  
    WellKnownObjectMode.Singleton);
```

3. Блокиране на основната нишка (за да не
завърши приложението)

```
Console.ReadLine();  
// или  
System.Threading.Thread.CurrentThread.Suspend();
```

Създаване на Remoting клиент

1. **Регистриране на Remoting канал**

```
TcpChannel channel = new TcpChannel();  
ChannelServices.RegisterChannel(channel);
```

2. **Активиране на отдалечен обект:**

```
Library remoteLibrary = (Library)  
    Activator.GetObject(typeof(Library),  
        "tcp://localhost:12345/Library");
```

или така:

```
RemotingConfiguration.RegisterWellKnownClientType(  
    typeof(CommonTypes.Library),  
    "tcp://localhost:12345/Library");  
Library remoteLibrary = new Library();
```

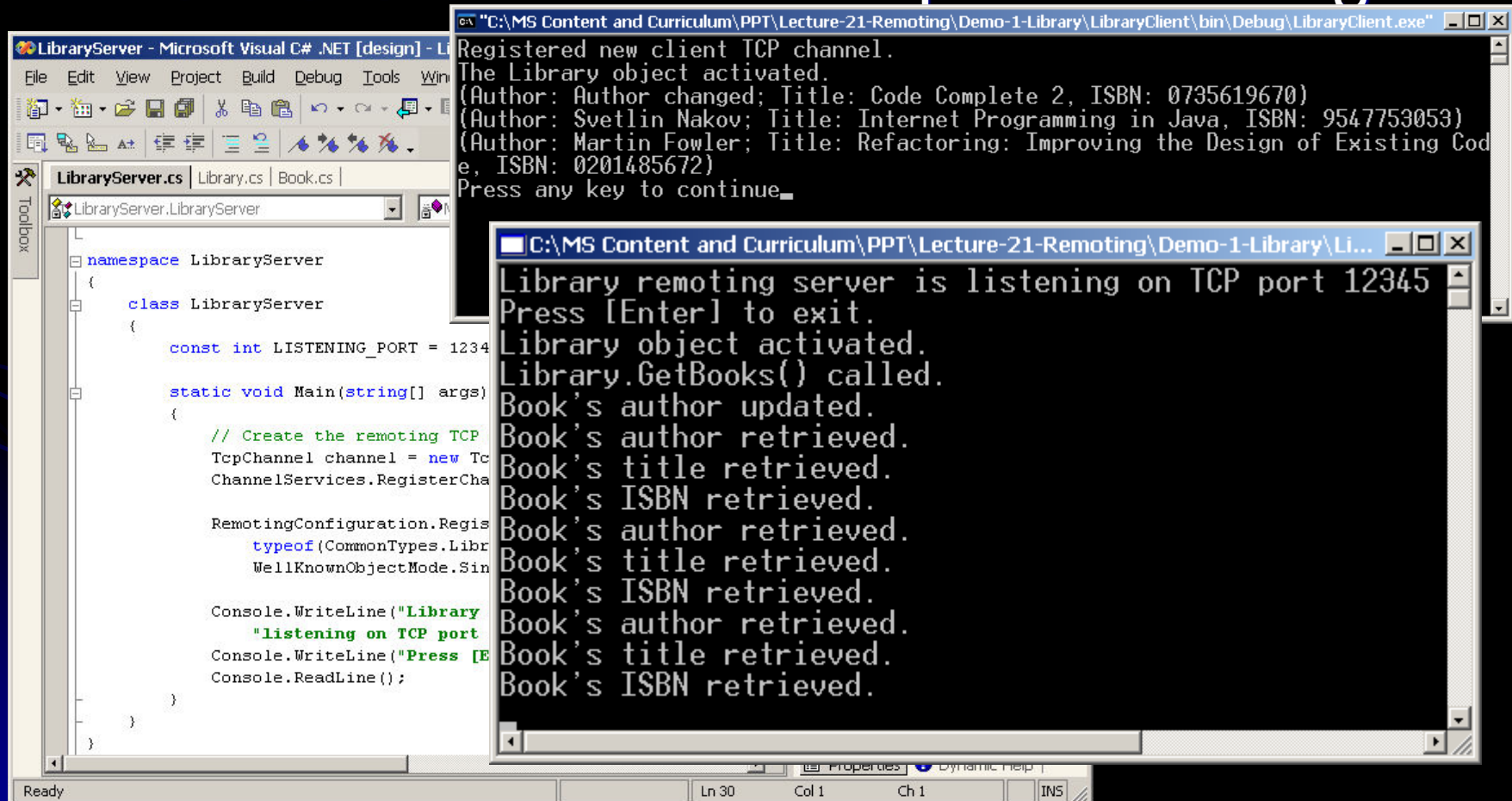
3. **Използване на отдалечения обект**

Проблемът с общите типове

- Използваните при Remoting типове трябва да са достъпни едновременно за клиента и за сървъра
- Има няколко подхода:
 - Общите типове се изнасят в отделно асембли (.DLL файл)
 - В Class Library проект във VS.NET
 - Добавя се референция към този DLL
 - Общите типове се достъпват през интерфейси, които се отделят в отделно асембли

Демонстрация #1

● Библиотека – нашето първо Remoting



Външно конфигуриране

- Конфигурирането на Remoting инфраструктурата може да става с външен за приложението файл
- Конфигурационният файл може да съдържа:
 - Канали и форматери
 - Схема на живот (lease-based lifetime)
 - Регистрация на клиентски обекти
 - Регистрация на сървърните обекти
- Зарежда се програмно (изрично):

```
RemotingConfiguration.Configure("MyConfig.xml");
```

Съдържание

- Многозадачност
- Нишки
- Синхронизация
- Асинхронни извиквания

● Забележка: Под някои слайдове в полето забележки има подробно описание на съдържанието на слайда

Многозадачност – съдържание

- Многозадачност

- Проблемът – защо многозадачност?
- Ползите от многозадачността
- Решението – процеси и нишки
- Прилики и разлики
- Какво предлагат?
- Кога са удобни нишките?
- Многозадачност – видове
- Имплементации на многозадачност
- Application Domains

- Нишки

- Синхронизация

- Асинхронни извиквания

Проблемът

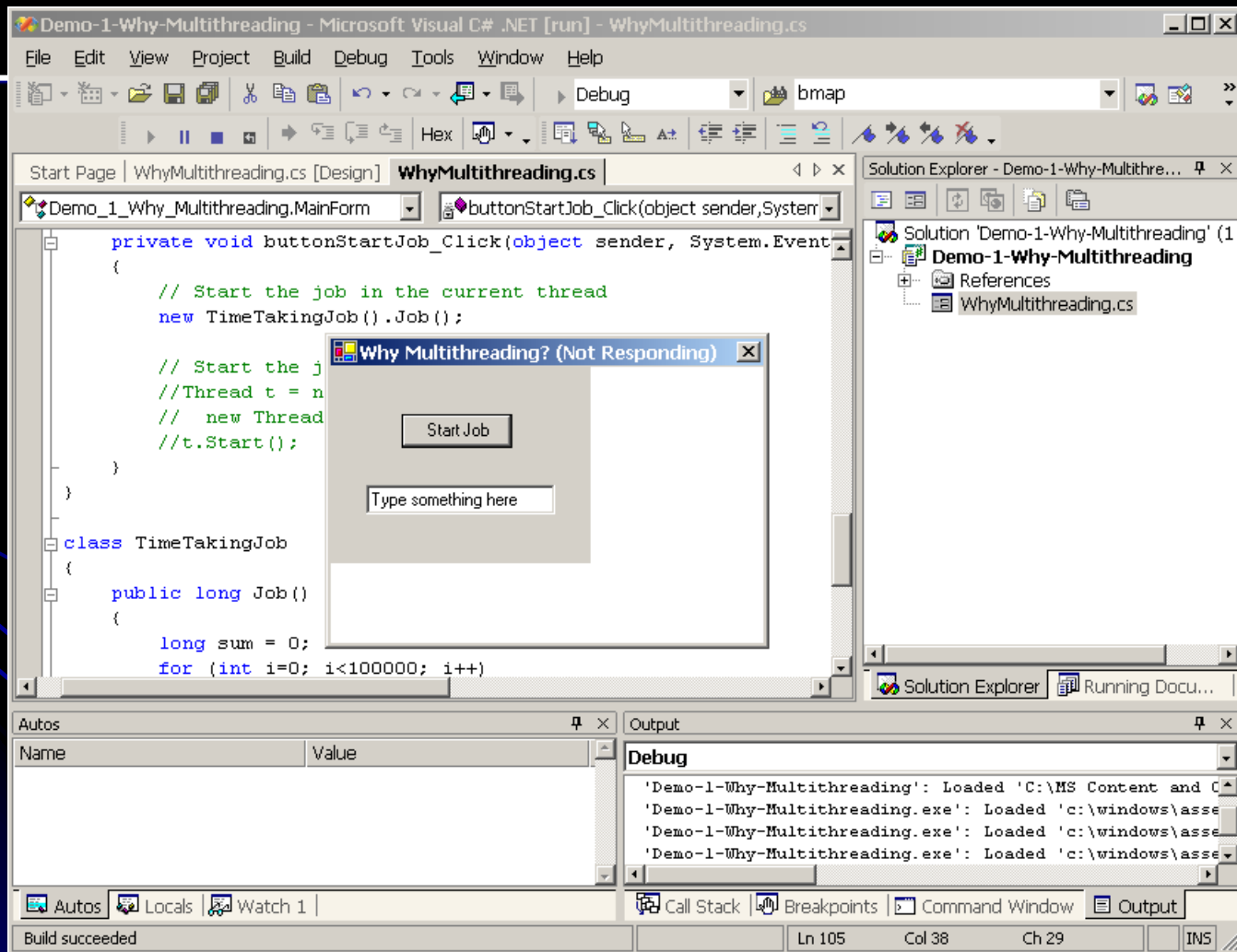
- Има случаи, в които вашето приложение трябва да изпълни времеотнемащи операции или да чака за освобождаването на ресурс
- Има случаи, в които вашето приложение трябва да извършва операция на заден план
- Нужен е механизъм, който да позволява няколко операции да се извършват едновременно

Ползите от многозадачността

- Performance – на машината с няколко процесора работи по-бързо
- Responsiveness – системата отговаря максимално бързо при интерактивна работа
- Throughput – подобряване на производителността
 - Пример със супермаркета – обслужване на няколко каси едновременно
- Обслужване на много потребители едновременно

Демонстрация #1

За



фдгдфгсдф

Решението – процеси и нишки

- Процесът представлява съвкупността от памет, стек и кода на приложението
- OS използват процеси (process), за да разграничават различните приложения
- Нишката (thread) е основната единица, на която OS може да заделя процесорно време
- В един процес различните задачи се изпълняват от отделни нишки
- В един процес има поне една нишка

Процес vs. Нишка

- Прилики

- собствен стек (stack)
- приоритет
- exception handlers

- Разлики

- Процесите са изолирани един от друг по отношение на памет и данни
- Нишките в един процес споделят паметта (променливите и данните) на този процес
- Процесите съдържат изпълнимия код, а нишките го изпълняват

Какво предлагат нишките?

- Използването на няколко нишки
 - Създава впечатление за извършване на няколко задачи едновременно
 - Причината: Времето, през което една нишка държи процесора е много кратко
- Например
 - Потребителят може да въвежда данни в текстообработваща програма докато се печатат данни на принтер
 - Би било недопустимо ако потребителят трябваше да изчака принтерът да свърши работата си

Кога са удобни нишките?

- Случаи, в които е удобно да се използват нишки:
 - Обслужване на много потребители едновременно, например Web сървър
 - Комуникация през мрежа (sockets)
 - Извършване на времеотнемаща операция
 - Различаване на задачите по приоритет
 - За да може потребителският интерфейс да продължи да “откликва” на потребителски заявки, докато на заден план се извършва друга задача

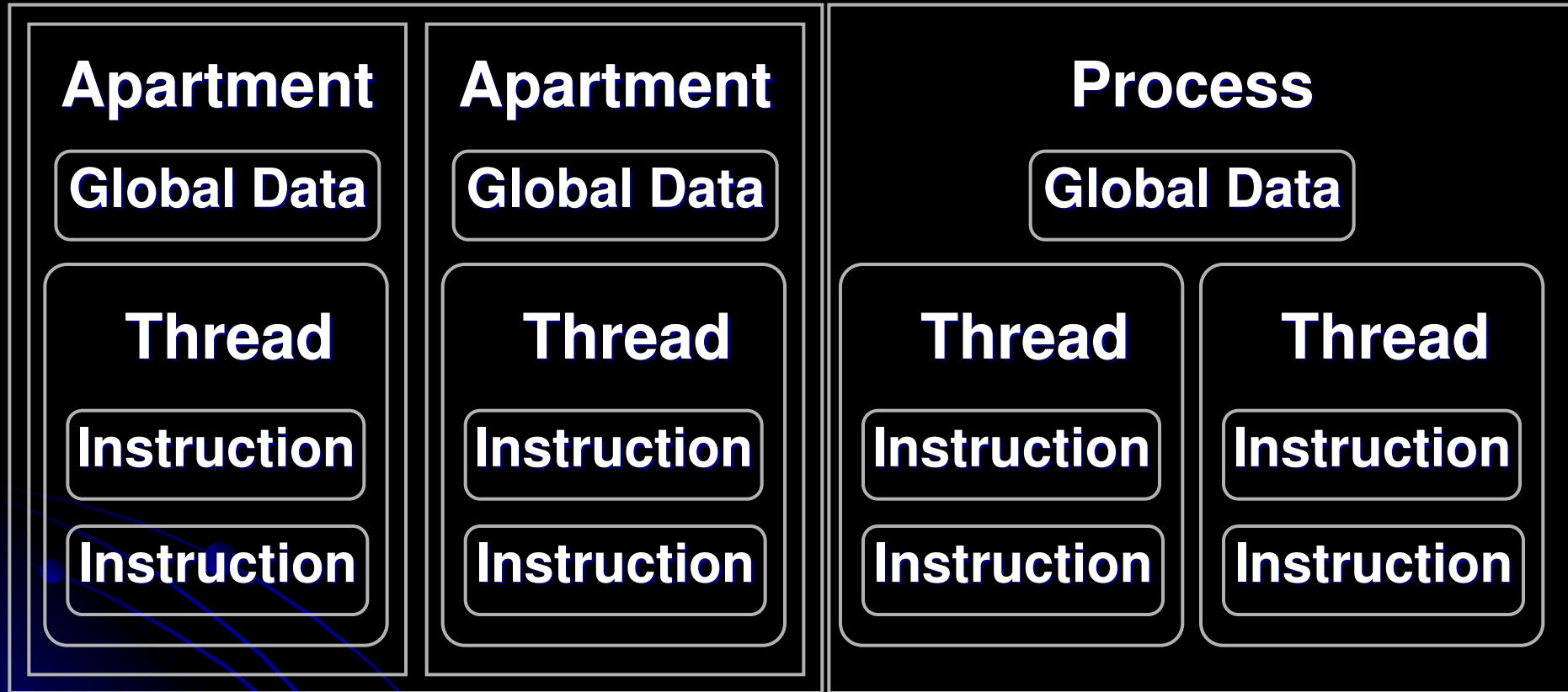
Многозадачност – Изпреварваща и кооперативна

- Кооперативна многозадачност (cooperative multitasking)
 - Нишката сама решава колко процесорно време ѝ е необходимо
 - Сваля се от процесора само ако е свършила или чака за някакъв ресурс
- Изпреварваща многозадачност (preemptive multitasking)
 - Планировчикът (task scheduler) заделя предварително някакво процесорно време
 - Без значение дали е приключила, нишката се сменя от процесора

Многозадачност – Изпреварваща и кооперативна

- При кооперативната многозадачност:
 - Една нишка може дълго време да държи процесора
 - Заради нея останалите нишки могат да чакат недопустимо дълго
- Някои системи използват комбиниран вариант
 - Нишки с висок приоритет се държат кооперативно спрямо нишките с по-нисък приоритет

Имплементации на многозадачност



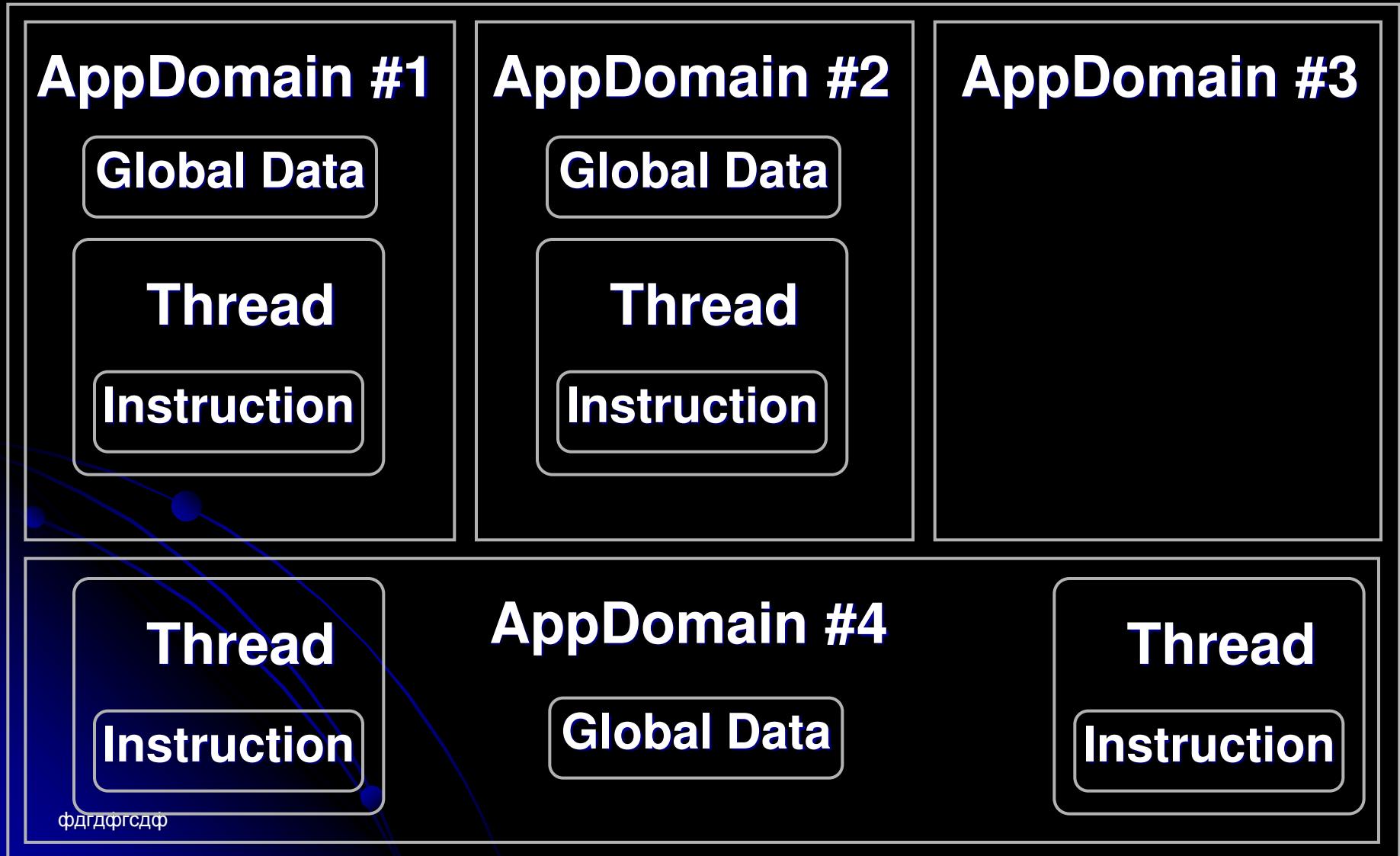
Apartment Threading

Free Threading

Application Domain

- .NET Framework добавя още едно ниво на абстракция между нишка и процес – Application Domain
- AppDomain-ът дава логическа изолация
 - При процесите изолацията е физическа, тъй като е имплементиран на ниво OS
- Предимства
 - Подобрена производителност
 - Ниво на изолация – като при процеса
 - Връзка м/у нишки без проху
 - Извършва type checking
- System.AppDomain

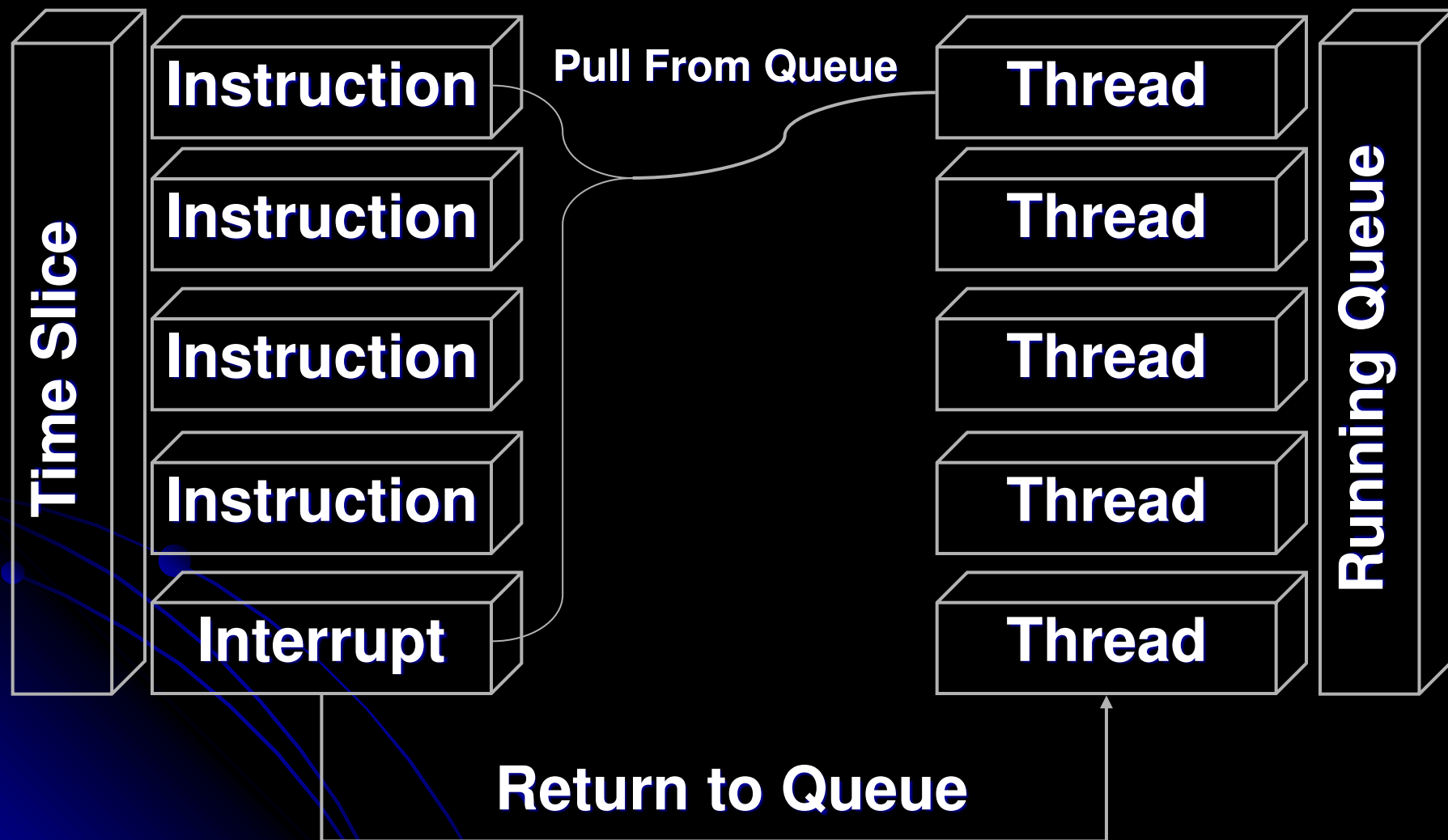
System.AppDomain



Нишки – съдържание

- Многозадачност
- Нишки
 - Как работят нишките
 - Живот на нишките
 - По-важни членове
 - Приоритет
 - Състояния
 - Thread Local Storage
 - Thread-Relative Static Fields
 - Прекратяване
 - Неудобства
 - Повреждане на данни
- Синхронизация
- Асинхронни извиквания

Как работи многопотоковостта



Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
} фдгдфгсдф
```

Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
} фдгдфгсдф
```

Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
} фдгдфгсдф
```

Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
} фдгдфгсдф
```

Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
} фдгдфгсдф
```

Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
} фдгдфгсдф
```

Малък пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter {
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Start();
        t2.Start();
    }
}
} фдгдфгсдф
```


Малък пример

//Резултат:

Thread1: job (1)

Thread1: job (2)

Thread2: job (1)

Thread1: job (3)

...

...

Thread1: job (53)

Thread2: job (46)

Thread2: job (47)

Thread1: job (54)

...

...

Thread1: job (97)

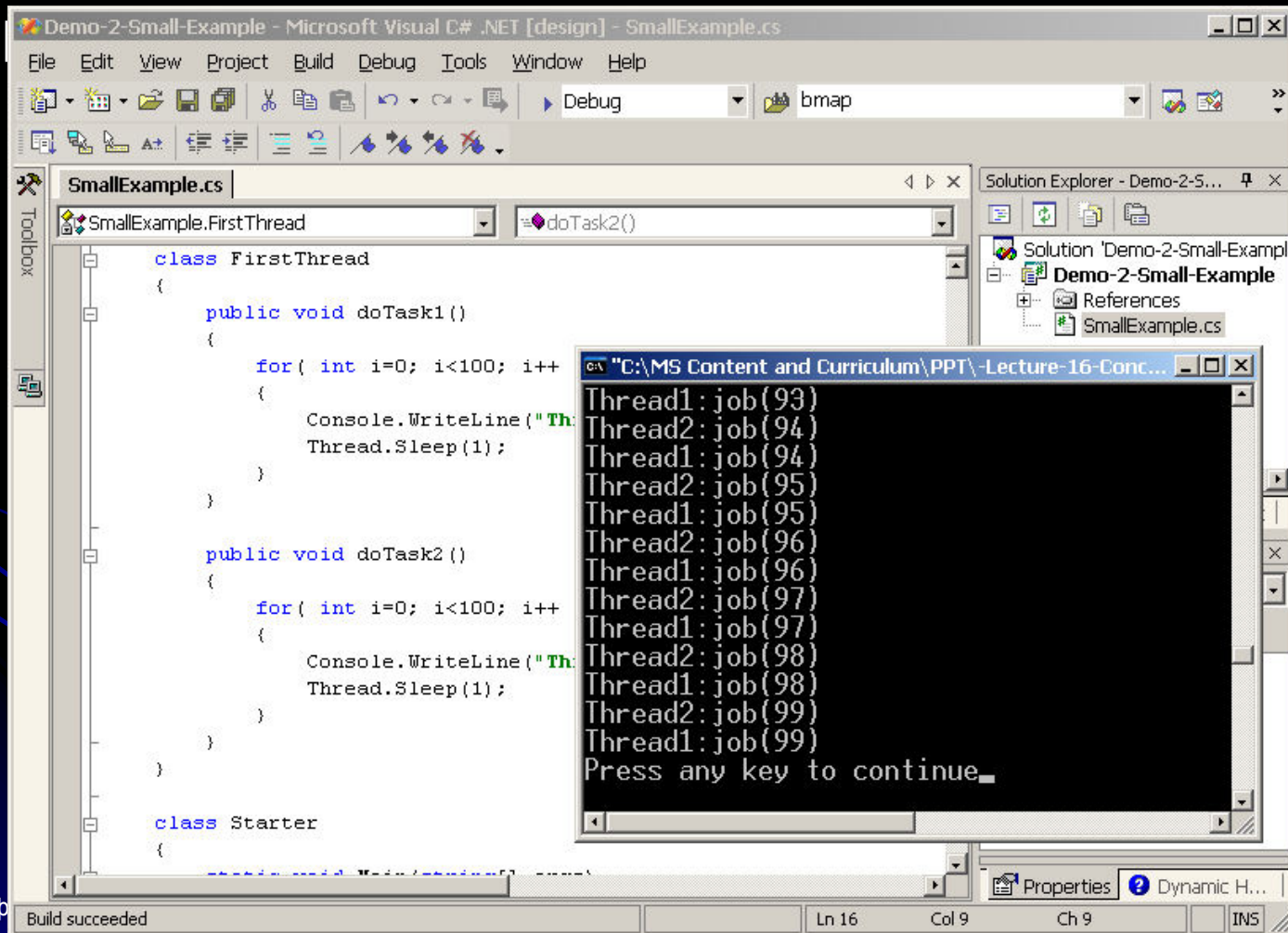
Thread2: job (99)

Thread1: job (98)

Thread1: job (99)

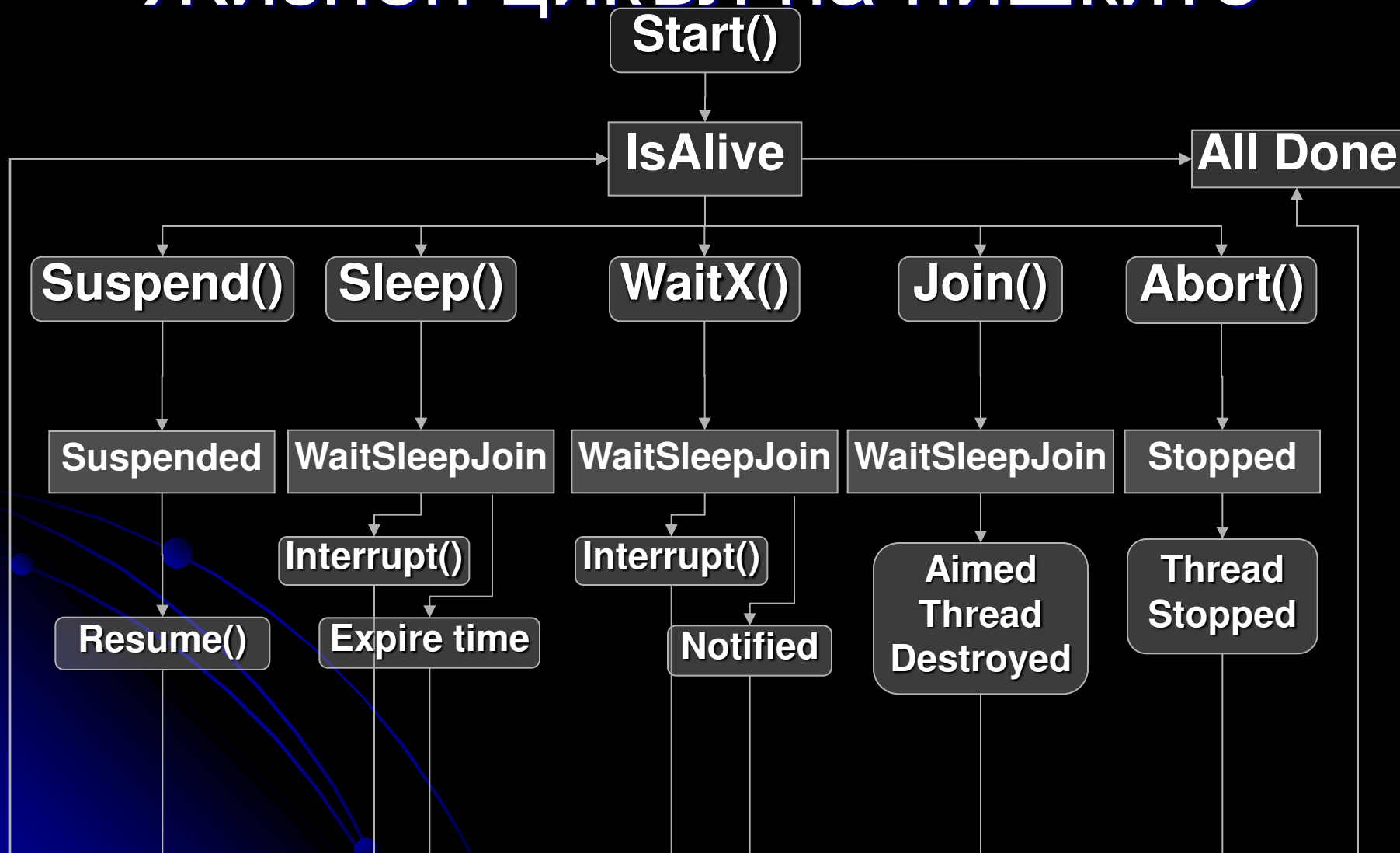
Демонстрация #2

Н



фдгдфгсдф

Жизнен цикъл на нишките



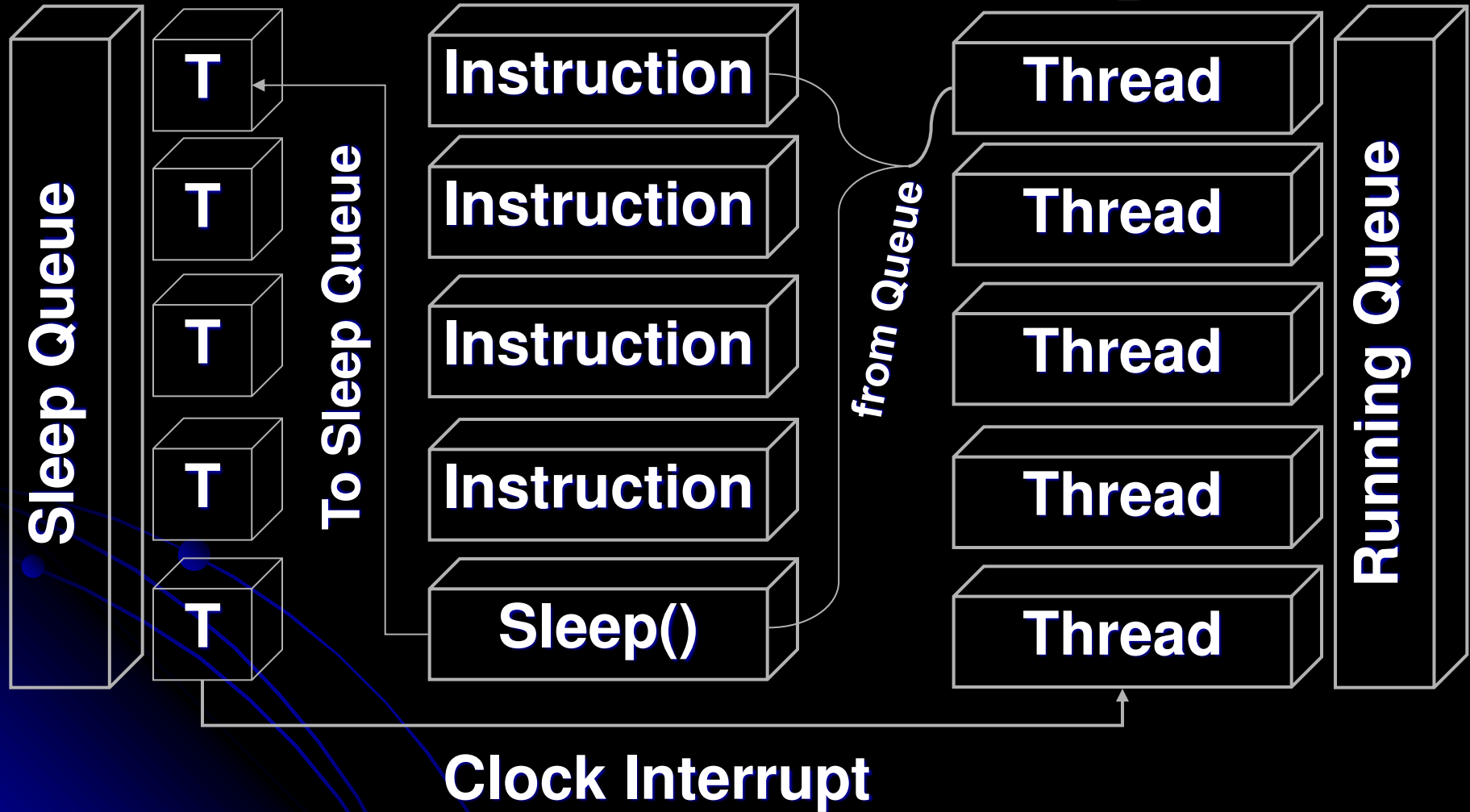
По-важните членове на Thread

- `public Thread( ThreadStart );`
 - Създава инстанция
 - Подава се делегат с метод, който да се изпълни при стартиране
- `Sleep()`
- `Suspend()`
- `Resume()`
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`
- `Name`
- `Priority`
- `ThreadState`
- `Abort()`
- `Interrupt()`
- `Join()`
- `Start()`

По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
 - “Приспива” текущата нишка за указания брой милисекунди (и наносекунди)
 - Извиква се само от самата нишка
- `Suspend()`
- `Resume()`
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`
- `Name`
- `Priority`
- `ThreadState`
- `Abort()`
- `Interrupt()`
- `Join()`
- `Start()`

Малко повече за Sleep ()



Приспиване на нишка

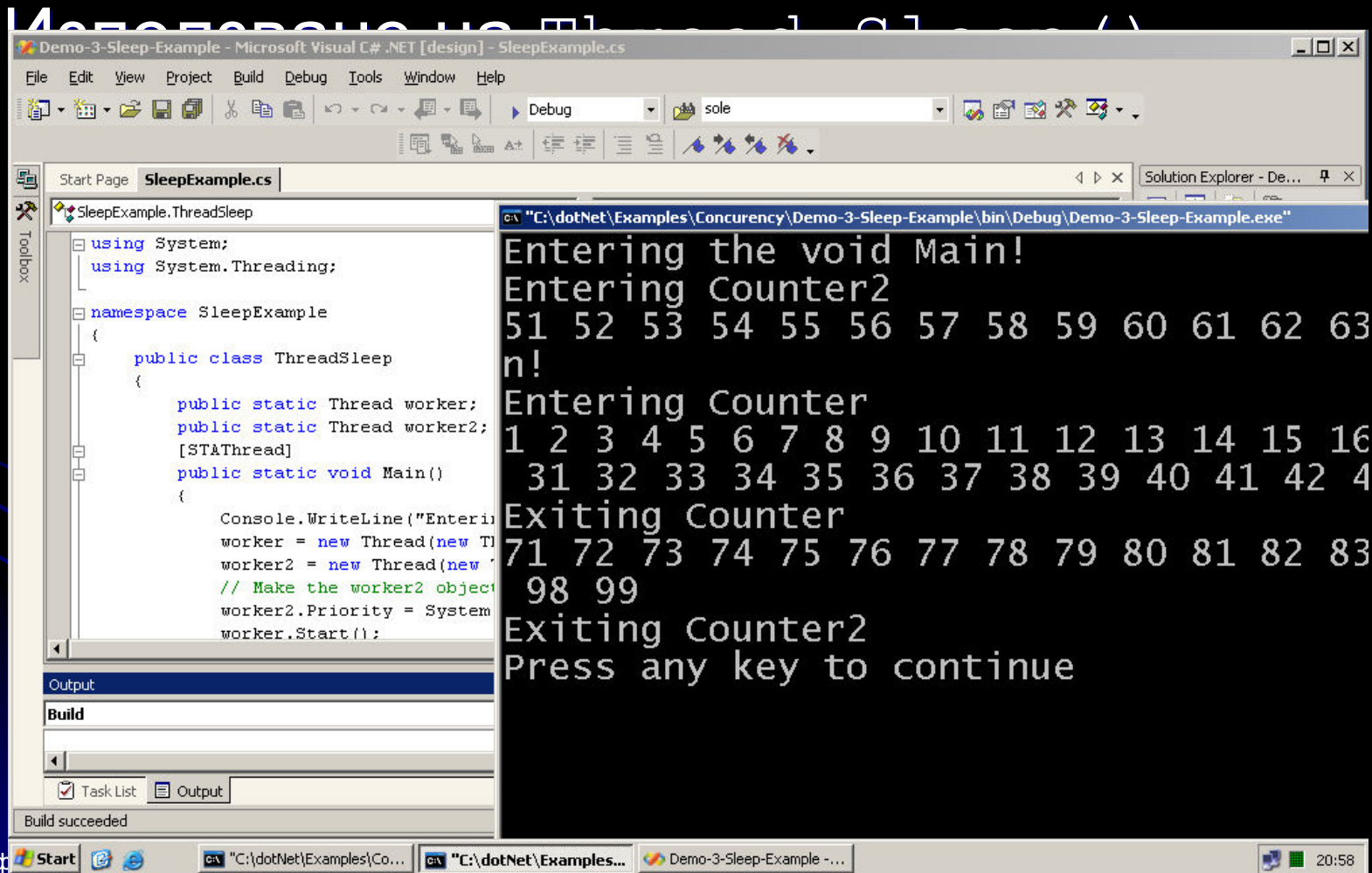
```
using System;
using System.Threading;
public class ThreadSleep
{
    public static Thread worker;
    public static Thread worker2;
    public static void Main()
    {
        Console.WriteLine("Entering void Main!");
        worker = new Thread(new
            ThreadStart(Counter));
        worker2 = new Thread(new
            ThreadStart(Counter2));
        worker.Start();
        worker2.Start();
        Console.WriteLine("Exiting void Main!");
    }
}
```

(примерът продължава)

Приспиване на нишка

```
public static void Counter()  
{  
    Console.WriteLine("Entering Counter");  
    for(int i = 1; i < 50; i++)  
    {  
        Console.Write(i + " ");  
        if(i == 10)  
            Thread.Sleep(1000);  
    }  
    Console.WriteLine("\nExiting Counter");  
}  
public static void Counter2()  
{  
    Console.WriteLine("Entering Counter2");  
    for(int i = 51; i < 100; i++)  
    {  
        Console.Write(i + " ");  
        if( i == 70 )  
            Thread.Sleep(5000);  
    }  
    Console.WriteLine("\nExiting Counter2");  
}  
}
```


Демонстрация #3



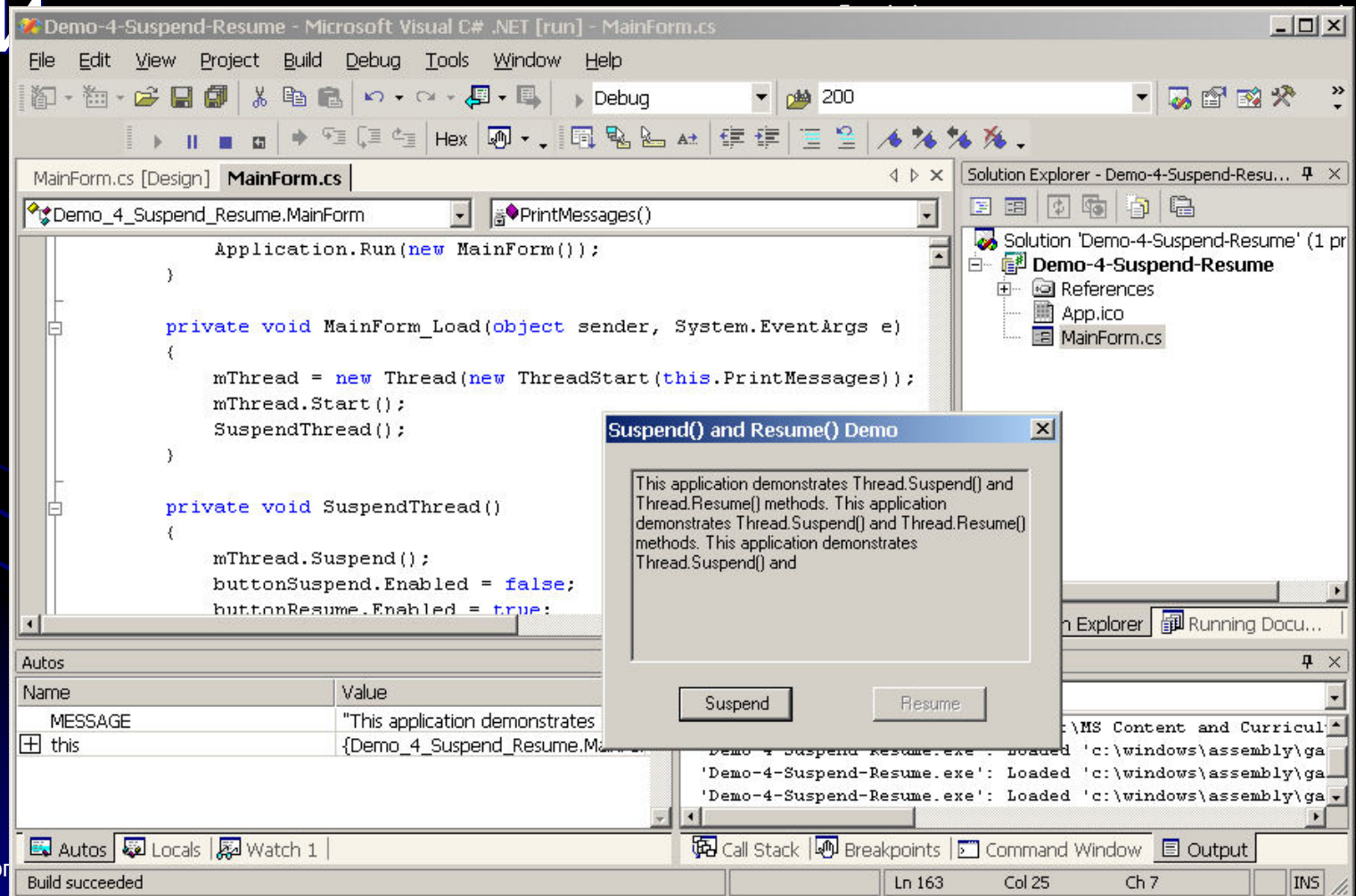
По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
- `Suspend()`
 - Ако нишката е в състояние `Running`, я преустановява временно (`suspend`)
 - ако е преустановена, не се случва нищо
- `Resume()`
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`
- `Name`
- `Priority`
- `ThreadState`
- `Abort()`
- `Interrupt()`
- `Join()`
- `Start()`

По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
- `Suspend()`
- `Resume()`
 - Подновява нишка, която е била преустановена (`suspended`)
 - Ако нишката работи, не прави нищо
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`
- `Name`
- `Priority`
- `ThreadState`
- `Abort()`
- `Interrupt()`
- `Join()`
- `Start()`

Демонстрация #4



По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
- `Suspend()`
- `Resume()`
- `IsAlive`
 - `true`, ако е стартирана и не е спряна прекъсната или прекратена
 - Повече информация дава `ThreadState`
- `IsBackground`
- `IsThreadPoolThread`
- `Name`
- `Priority`
- `ThreadState`
- `Abort()`
- `Interrupt()`
- `Join()`
- `Start()`

По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
- `Suspend()`
- `Resume()`
- `IsAlive`
- `IsBackground`
 - Преден план (foreground) и фонов режим (background)
 - Свойство за смяна/извличане
- `IsThreadPoolThread`
- `Name`
- `Priority`
- `ThreadState`
- `Abort()`
- `Interrupt()`
- `Join()`
- `Start()`

По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
- `Suspend()`
- `Resume()`
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`
 - Свойство за смяна/извличане
 - `true`, ако нишката принадлежи на `managed thread pool`, иначе `false`
- `Name`
- `Priority`
- `ThreadState`
- `Abort()`
- `Interrupt()`
- `Join()`
- `Start()`

По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
- `Suspend()`
- `Resume()`
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`

• Name

- Всяка нишка в .NET Framework може да има име

• Свойство за смяна/извличане на името

- `Priority`
- `ThreadState`
- `Abort()`
- `Interrupt()`
- `Join()`
- `Start()`

По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
- `Suspend()`
- `Resume()`
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`
- `Name`
- `Priority`
 - `Lowest, BelowNormal, Normal (по подразбиране), AboveNormal, Highest`
 - **Свойство за промяна/извличане**
- `ThreadState`
- `Abort()`
- `Interrupt()`
- `Join()`
- `Start()`

По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
- `Suspend()`
- `Resume()`
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`
- `Name`
- `Priority`

• ThreadState

- **Съдържа състоянието на нишката – зависи от това дали нишката работи**

• Свойство само за извличане

- `Abort()`
- `Interrupt()`
- `Join()`
- `Start()`

По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
- `Suspend()`
- `Resume()`
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`
- `Name`
- `Priority`
- `ThreadState`
- `Abort()`
 - Обикновено “убива” нишката
 - Хвърля `ThreadAbortException` в извиканата нишка
- `Interrupt()`
- `Join()`
- `Start()`

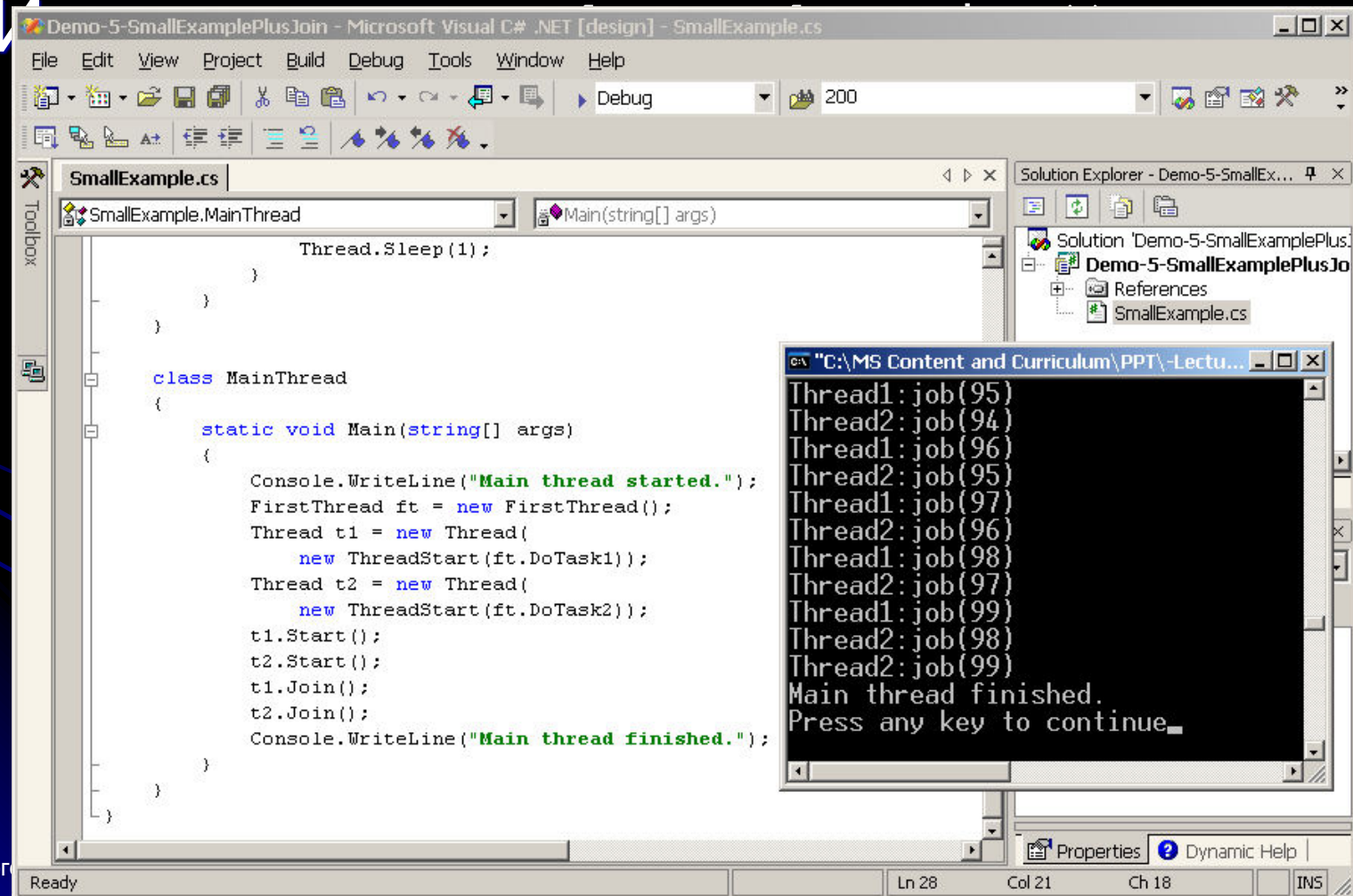
По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
- `Suspend()`
- `Resume()`
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`
- `Name`
- `Priority`
- `ThreadState`
- `Abort()`
- `Interrupt()`
 - Събужда нишка ако е в състояние `WaitSleepJoin`, иначе не прави нищо
 - Хвърля `ThreadInterruptedException`
- `Join()`
- `Start()`

По-важните членове на Thread

- `public Thread( ThreadStart start);`
- `Sleep()`
- `Suspend()`
- `Resume()`
- `IsAlive`
- `IsBackground`
- `IsThreadPoolThread`
- `Name`
- `Priority`
- `ThreadState`
- `Abort()`
- `Interrupt()`
- `Join()`
 - Извикващата нишка изчаква, докато извиканата приключи
 - Може да се укаже таймаут (timeout)
- `Start()`

Демонстрация #5



По-важните членове на Thread

- `public Thread( ThreadStart start);`

- `Sleep()`

- `Suspend()`

- `Resume()`

- `IsAlive`

- `IsBackground`

- `IsThreadPoolThread`

- `Name`

- `Priority`

- `ThreadState`

- `Abort()`

- `Interrupt()`

- `Join()`

- `Start()`

- Стартира посочената нишка

- Операцията не е блокираща (връща управлението веднага)

Приоритет на нишка

- Повечето имплементации на многонишковост включват отличителното качество "приоритет"
 - Приоритетът указва колко важна е нишката (колко време да ѝ се отделя)
 - Важен е за планировчика (task scheduler)
- Приоритетът в .NET Framework
 - Възможни стойности
 - Lowest, BelowNormal, Normal (по подразбиране), AboveNormal, Highest
 - В Win32 има и приоритет RealTime
 - OS не е длъжна да се съобразява с приоритета на

Приоритет – пример

```
class FirstThread {
    public void DoTask1() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread1:job({0})",i);
    }
    public void DoTask2() {
        for( int i=0; i<100; i++ )
            Console.WriteLine("Thread2:job({0})",i);
    }
}
class Starter
{
    static void Main(string[] args) {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(
            new ThreadStart(ft.DoTask1));
        Thread t2 = new Thread(
            new ThreadStart(ft.DoTask2));
        t1.Priority = ThreadPriority.Highest;
        t1.Start();
        t2.Start();
    }
}
```

Приоритет

Результат:

Thread1: job (1)

Thread1: job (2)

Thread2: job (1)

Thread1: job (3)

...

...

Thread1: job (97)

Thread1: job (98)

Thread2: job (85)

Thread1: job (99)

Thread2: job (86)

...

...

Thread2: job (97)

Thread2: job (98)

Thread2: job (99)

ThreadState

- Всяка нишка във всеки един момент е в някое от състоянията на енумерацията ThreadState
- Допуска се да бъде в няколко състояния едновременно
- Енумерацията има атрибут FlagsAttribute
 - Това позволява побитово комбиниране на стойностите ѝ:

```
if ((state & (Unstarted | Stopped)) == 0)  
    // implies Running
```

- Когато се създаде нишка е в състояние Unstarted

фдгдфгсдф

- Остава така докато не се извика Start()

СЪСТОЯНИЯТА В ThreadState

- Aborted (256)
 - Извикан е метода Abort ()
 - Нишката е в състояние Stopped, едновременно с Aborted
- AbortRequested (128)
- Background (4)
- Running (0)
- Stopped (16)
- StopRequested (1)
- Suspended (64)
- SuspendRequested (2)
- Unstarted (8)
- WaitSleepJoin (32)

СЪСТОЯНИЯТА В ThreadState

- Aborted(256)
- AbortRequested(128)
 - Методът Abort() е бил извикан
 - Още не е получила ThreadAbortException, което ще се опита да я прекрати
- Background(4)
- Running(0)
- Stopped(16)
- StopRequested(1)
- Suspended(64)
- SuspendRequested(2)
- Unstarted(8)
- WaitSleepJoin(32)

СЪСТОЯНИЯТА В ThreadState

- Aborted(256)
- AbortRequested(128)
- Background(4)
 - Нишката е във фонов режим
 - Променя се със свойството `Thread.IsBackground`
- Running(0)
- Stopped(16)
- StopRequested(1)
- Suspended(64)
- SuspendRequested(2)
- Unstarted(8)
- WaitSleepJoin(32)

СЪСТОЯНИЯТА В ThreadState

- Aborted(256)
- AbortRequested(128)
- Background(4)
- Running(0)
 - Нишката е стартирана и не е блокирана
 - Няма чакащо изключение
ThreadAbortedException
- Stopped(16)
- StopRequested(1)
- Suspended(64)
- SuspendRequested(2)
- Unstarted(8)
- WaitSleepJoin(32)

СЪСТОЯНИЯТА В ThreadState

- Aborted(256)
- AbortRequested(128)
- Background(4)
- Running(0)
- Stopped(16)
 - Нишката е отговорила на заявка от Abort() или
 - Прекратила е работата си доброволно
- StopRequested(1)
- Suspended(64)
- SuspendRequested(2)
- Unstarted(8)
- WaitSleepJoin(32)

СЪСТОЯНИЯТА В ThreadState

- Aborted(256)
- AbortRequested(128)
- Background(4)
- Running(0)
- Stopped(16)
- StopRequested(1)
 - Поискано е било от нишката да спре работа
 - Само за вътрешна употреба
- Suspended(64)
- SuspendRequested(2)
- Unstarted(8)
- WaitSleepJoin(32)

СЪСТОЯНИЯТА В ThreadState

- Aborted(256)
- AbortRequested(128)
- Background(4)
- Running(0)
- Stopped(16)
- StopRequested(1)
- Suspended(64)
 - Нишката е била преустановена
 - Независимо колко пъти е извикан Suspend(), един Resume() е достатъчен
- SuspendRequested(2)
- Unstarted(8)
- WaitSleepJoin(32)

СЪСТОЯНИЯТА В ThreadState

- Aborted(256)
- AbortRequested(128)
- Background(4)
- Running(0)
- Stopped(16)
- StopRequested(1)
- Suspended(64)
- SuspendRequested(2)
 - Извикан е метода Suspend()
 - Изчаква се да стигне до стабилно състояние, за да се преустанови
- Unstarted(8)
- WaitSleepJoin(32)

СЪСТОЯНИЯТА В ThreadState

- Aborted (256)
- AbortRequested (128)
- Background (4)
- Running (0)
- Stopped (16)
- StopRequested (1)
- Suspended (64)
- SuspendRequested (2)
- Unstarted (8)
 - Не е стартирана, стартира се със `Start()`
 - След като се стартира нишката никога повече не може пак да е в това състояние
- WaitSleepJoin (32)

СЪСТОЯНИЯТА В ThreadState

- Aborted(256)
- AbortRequested(128)
- Background(4)
- Running(0)
- Stopped(16)
- StopRequested(1)
- Suspended(64)
- SuspendRequested(2)
- Unstarted(8)
- WaitSleepJoin(32)
 - Нишката е блокирана
 - Става с един от методите Thread.Wait(), Thread.Sleep(), Thread.Join()

Прекратяване на нишки

- За да се “убие” една нишка, се използва методът `Thread.Abort()`
- `Thread.Abort()` **хвърля специалното изключение** `ThreadAbortedException`
 - Хвърля `ThreadStateException`, ако нишката вече е прекратена
- Изпълняват се `catch` и `finally` на прекратената нишка
- `Thread.ResetAbort()` **не позволява на CLR да хвърли изключението отново**

Прекратяване на нишки – пример

```
public class BackgroundThread {  
    public void DoBackgroundJob() {  
        try {  
            // Нишката извършва някаква работа  
        }  
        catch (ThreadInterruptedException) {  
            MessageBox.Show("Thread interrupted.");  
        }  
        catch (ThreadAbortException) {  
            MessageBox.Show("Thread aborted.");  
        }  
        finally {  
            MessageBox.Show("Finally block.");  
        }  
        MessageBox.Show("After finally block.");  
    }  
}
```

}рдгдфгсдф

(примерът продължава)

Прекратяване на нишки – пример

```
public class InterruptAbortDemo
{
    private Thread mBgThread;

    public InterruptAbortDemo() {
        bg = new BackgroundThread();
        mBgThread = new Thread(new
            ThreadStart(bg.DoBackgroundJob));
        mBgThread.IsBackground = true;
        mBgThread.Start();
    }

    private void btnSuspend_Click(...)
    {
        mBgThread.Suspend();
    }
}
```

(примерът продължава)

Прекратяване на нишки – пример

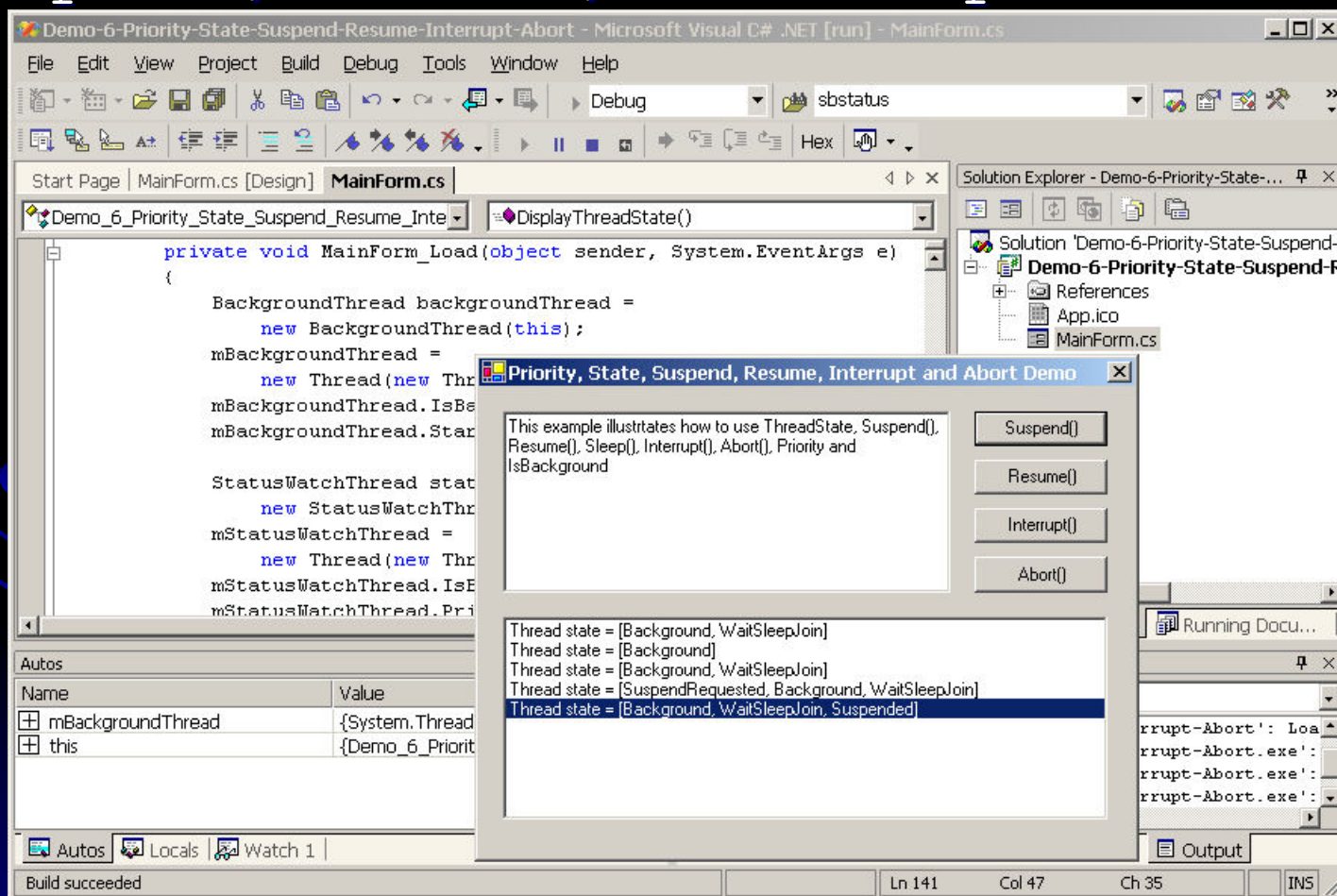
```
private void btnResume_Click(...)
{
    mBgThread.Resume();
}

private void btnInterrupt_Click(...)
{
    mBgThread.Interrupt();
}

private void btnAbort_Click(...)
{
    mBgThread.Abort();
}
}
```

Демонстрация #6

- Използване на Priority, State, Suspend, Resume, Interrupt и Abort



Thread Local Storage

- Контейнер, в който нишките могат да съхраняват данни
- Нишка не може да достъпи данните на друга нишка
- `Thread.AllocateNamedDataSlot`
 - Именуван контейнер
 - Проблеми – уникални имена
- `Thread.AllocateDataSlot`
 - Няма име, само reference
- Проблеми: две парчета код от една и съща нишка използват общ слот

Thread Local Storage – пример

```
class Threads {  
    public void CreateDataThread() {  
        LocalDataStoreSlot slot =  
            Thread.AllocateNamedDataSlot("mySlot");  
        //Записваме важни данни в NamedSlot  
        Thread.SetData(slot, "IMPORTANT DATA");  
        // Suspend-ваме  
        Thread.CurrentThread.Suspend();  
        // Прочитаме отново данните  
        object myData = Thread.GetData("mySlot");  
    }  
    public void ReadDataThread() {  
        LocalDataStoreSlot slot =  
            Thread.GetNamedDataSlot("mySlot");  
        // Опитваме се да променим информацията  
        Thread.SetData(slot, "BAD DATA");  
    }  
}
```

Thread Local Storage – пример

```
class TLSDemo
{
    static void Main(string[] args)
    {
        Threads threads = new Threads();
        Thread createData = new Thread( new
            ThreadStart(threads.createDataThread) );
        createData.Start();

        Thread readData = new Thread( new
            ThreadStart(threads.readDataThread) );
        readData.Start();
        readData.Join();
        createData.Resume();
    }
}
```

фдгдф



Thread-Relative Static Fields

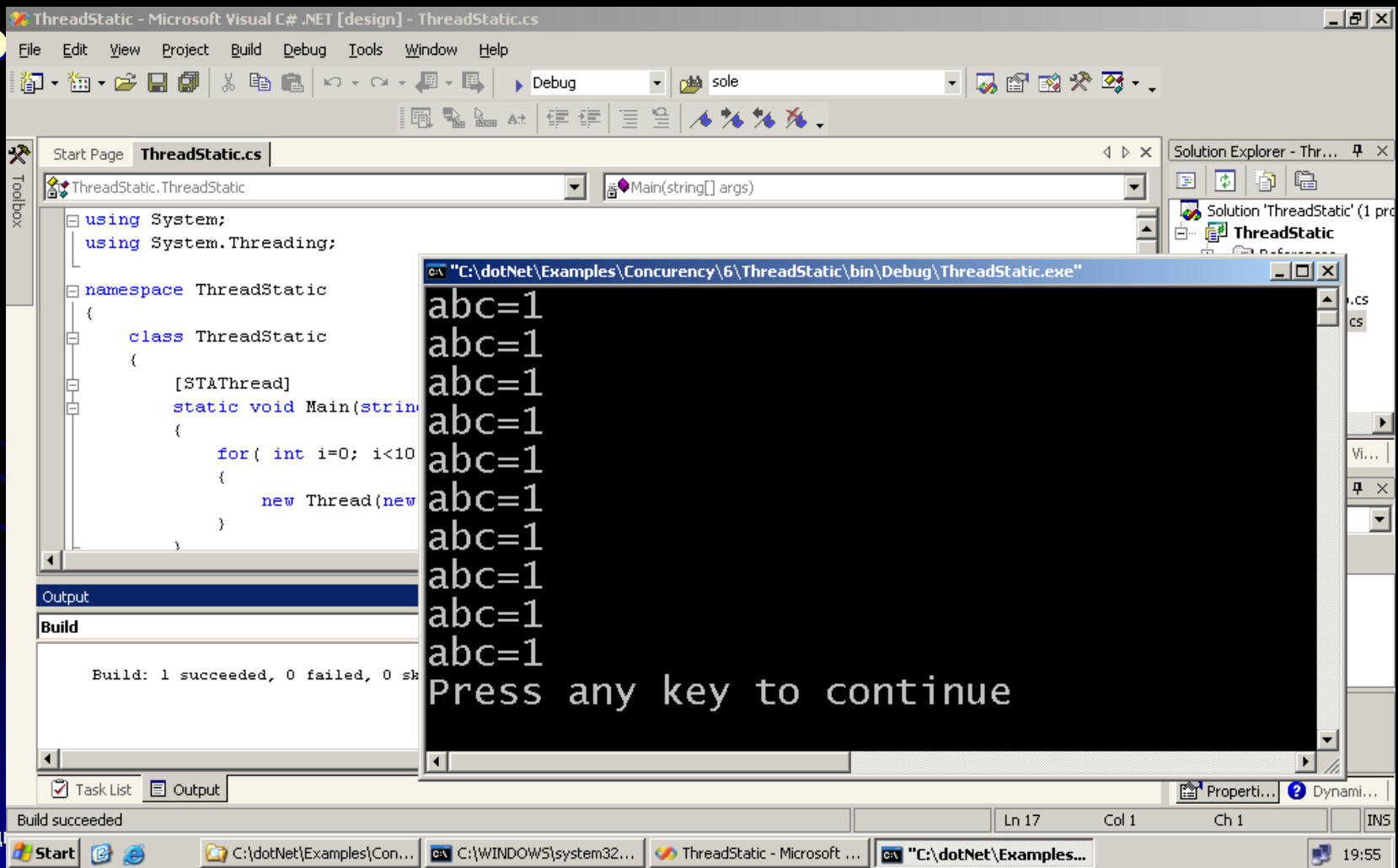
- Като статична променлива, но има по една инстанция за всяка нишка на програмата
- Декларира се като статична променлива, но с атрибута `[ThreadStatic]`
- Всяка нишка, която я използва трябва да я инициализира
- Не разчитайте на конструктора да инициализира такива променливи, защото той се изпълнява в родителската нишка!

Thread-Relative Static Fields – пример

```
class ThreadStatic {
    static void Main(string[] args) {
        for( int i=0; i<10; i++ ) {
            new Thread(new ThreadStart(new
                MyThread().DoTask)).Start();
        }
    }
}

class MyThread {
    [ThreadStatic] public static int abc;
    public MyThread() {
        abc=42; // This runs in the main app. thread
    }
    public void DoTask() {
        abc++;
        Console.WriteLine("abc={0}", abc);
    }
}
```


Демонстрация #8



Неудобства на нишките

- Самата имплементация на многонишковост изисква ресурси
- Добра практика е да не прекалявате с използването на нишки
 - Възможно е процесорът да използва повече ресурси за управлението на нишките отколкото за изпълнението им
- Препоръчва се да използвате ThreadPool когато е възможно
- Следенето на много нишки е трудно

Споделени данни – проблеми

- Race condition

- Две нишки едновременно достъпват едни и същи данни
- Непредвидими резултати
- Например две нишки едновременно изпълняват $i++$ на обща променлива i

- Deadlock

- Състояние, в което две нишки се чакат взаимно за освобождаване на взаимно заети ресурси
- А заема ресурса X, В заема ресурса Y и чака за X, а А започва да чака за Y

Race Condition – пример

```
using System;
using System.Threading;

class Account
{
    public int mBalance;

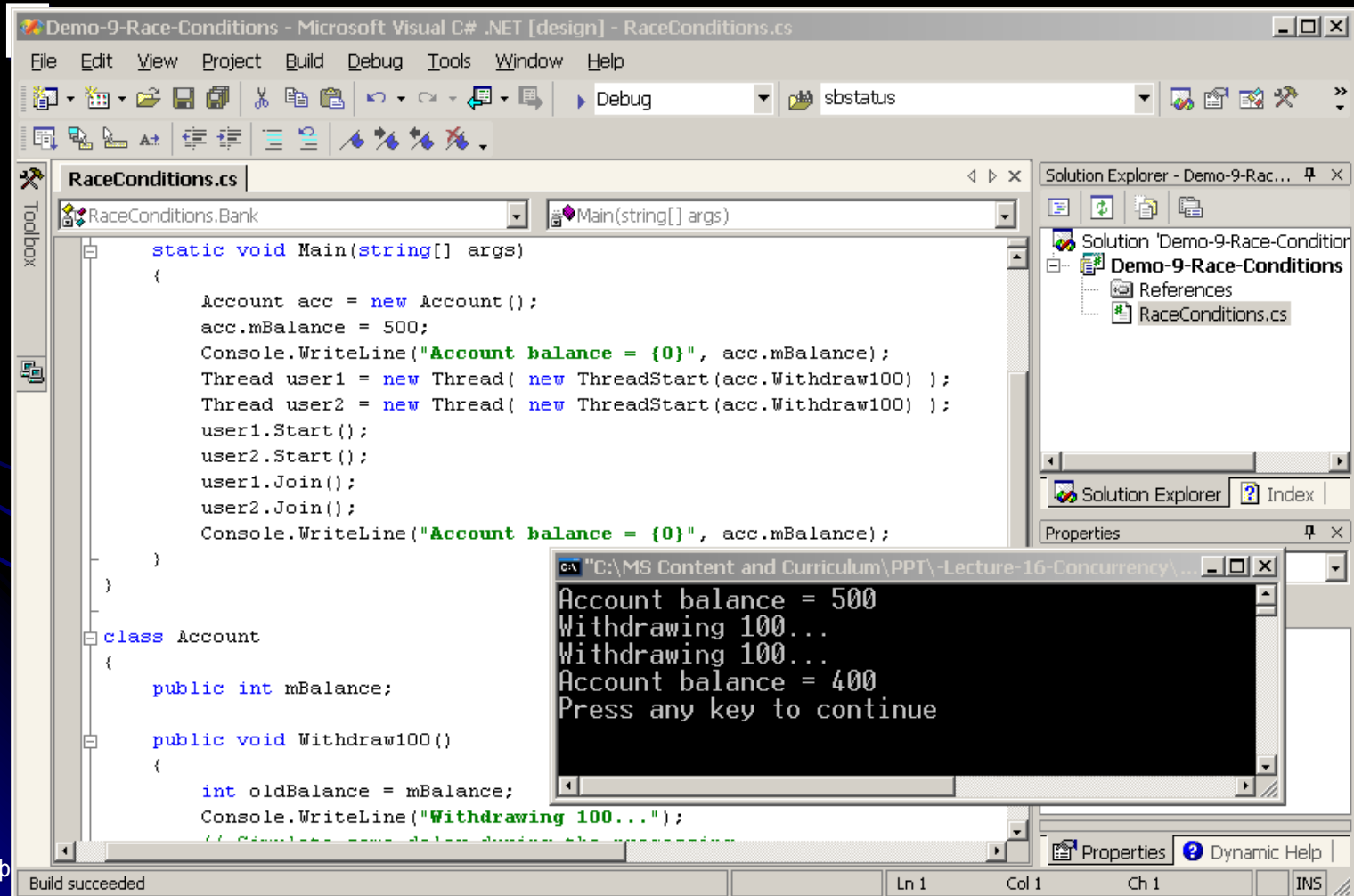
    public void Withdraw100()
    {
        int oldBalance = mBalance;
        Console.WriteLine("Withdrawing 100...");
        Thread.Sleep(100);
        int newBalance = oldBalance - 100;
        mBalance = newBalance;
    }
}
```

(примерът продължава)

Race Condition – пример

```
static void Main(string[] args)
{
    Account acc = new Account();
    acc.mBalance = 500;
    Console.WriteLine("Account balance = {0}",
        acc.mBalance);
    Thread user1 = new Thread(
        new ThreadStart(acc.Withdraw100) );
    Thread user2 = new Thread(
        new ThreadStart(acc.Withdraw100) );
    user1.Start();
    user2.Start();
    user1.Join();
    user2.Join();
    Console.WriteLine("Account balance = {0}",
        acc.mBalance);
}
```

Демонстрация #9



Синхронизация – съдържание

- Многозадачност
- Нишки
- Синхронизация
 - Най-доброто решение
 - “Стратегии” за синхронизация
 - Synchronized Contexts
 - Synchronized code regions
 - MethodImplAttribute
 - Unmanaged Synchronization – WaitHandle
 - Специални класове
 - Класически синхронизационни проблеми
 - ThreadPool
- Асинхронни извиквания

Най-доброто решение

- Най-добра сигурност при споделени данни в многонишкова среда е липсата на споделени данни
- Препоръчва се
 - Да се капсулират данните и ресурсите в инстанции на обекти
 - Тези обекти да не се споделят между нишки
- Когато това е невъзможно, .NET Framework, предоставя механизми за синхронизация на достъпа до общи данни

“Стратегии” за синхронизация

- Синхронизирани контексти (Synchronized Contexts)
- Синхронизирани “пасажи” код (Synchronized code regions)
- `[MethodImplAttribute (MethodImplOptions.Synchronized) ]`
над имплементацията на метод
- Unmanaged синхронизация
 - Класове за фина, ръчно настроена (custom) синхронизация с обекти на OS

Synchronized Contexts

- Използва се `SynchronizationAttribute` за обекти, наследяващи `ContextBoundObject`
 - обекти, които оперират в един контекст
- Всички нишки в този контекст достъпват методите на инстанции на такива обекти последователно
- Статичните членове не са предпазени
- Не се поддържа синхронизиране на специфични “отрязъци” от код – синхронизира цял клас

Synchronized Contexts – пример

```
[SynchronizationAttribute]
class CBO : ContextBoundObject
{
    public void DoSomeTask1()
    {
        Console.WriteLine("Job1 started.");
        Thread.Sleep(2000);
        Console.WriteLine("Job1 finished.\n");
    }

    public void DoSomeTask2()
    {
        Console.WriteLine("Job2 started.");
        Thread.Sleep(1500);
        Console.WriteLine("Job2 finished.\n");
    }
}
```

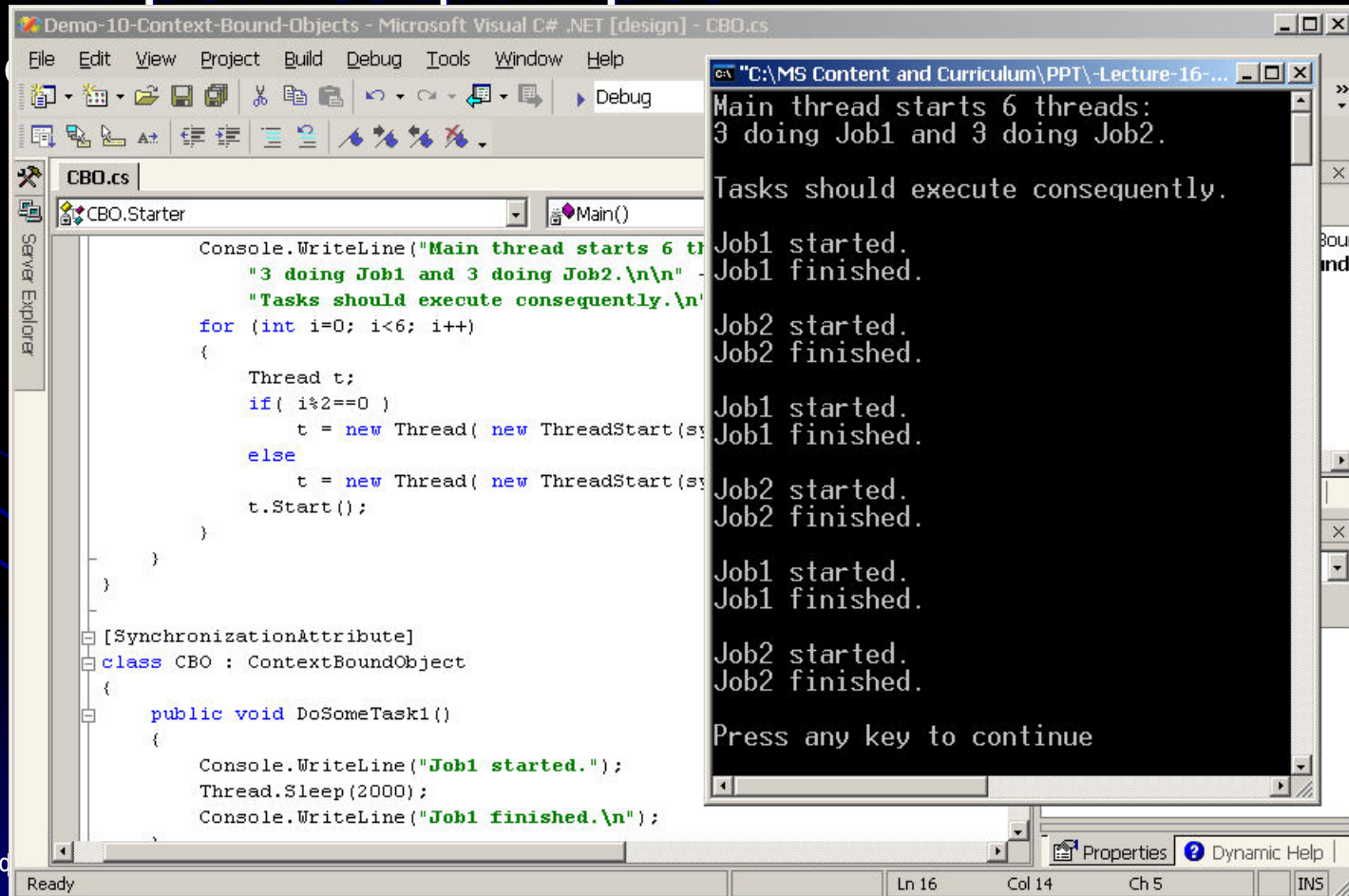
(примерът продължава)

Synchronized Contexts – пример

```
static void Main()
{
    CBO syncClass = new CBO();
    Console.WriteLine("Started 6 threads:\n" +
        "3 doing Job1 and 3 doing Job2.\n\n");
    for (int i=0; i<6; i++)
    {
        Thread t;
        if (i%2==0)
            t = new Thread( new ThreadStart(
                syncClass.DoSomeTask1) );
        else
            t = new Thread( new ThreadStart(
                syncClass.DoSomeTask2) );
        t.Start();
    }
}
```

Демонстрация #10

- Синхронизация чрез



The screenshot shows the Microsoft Visual C# .NET [design] - CBO.cs window. The code in CBO.cs is as follows:

```
using System;
using System.Threading;

class CBO : ContextBoundObject
{
    public void DoSomeTask1()
    {
        Console.WriteLine("Job1 started.");
        Thread.Sleep(2000);
        Console.WriteLine("Job1 finished.\n");
    }
}

class CBO.Starter
{
    static void Main()
    {
        Console.WriteLine("Main thread starts 6 threads:");
        Console.WriteLine("3 doing Job1 and 3 doing Job2.\n");
        Console.WriteLine("Tasks should execute consequently.\n");
        for (int i=0; i<6; i++)
        {
            Thread t;
            if ( i%2==0 )
                t = new Thread( new ThreadStart(someTask1) );
            else
                t = new Thread( new ThreadStart(someTask2) );
            t.Start();
        }
    }
}
```

The console output shows the execution of the program:

```
Main thread starts 6 threads:
3 doing Job1 and 3 doing Job2.

Tasks should execute consequently.

Job1 started.
Job1 finished.

Job2 started.
Job2 finished.

Job1 started.
Job1 finished.

Job2 started.
Job2 finished.

Job1 started.
Job1 finished.

Job2 started.
Job2 finished.

Press any key to continue
```

Критични секции

```
lock (obj) {  
    //..code..  
}
```

```
Monitor.Enter(obj);  
try { /*..code..*/ }  
finally { Monitor.Exit(obj); }
```

- ◆ **Управлявана (managed) реализация**
- ◆ **Може да се ползва и за статични членове на класовете**
- ◆ **Синхронизира се изпълнението само частта, която е застрашена**
- ◆ **Нарича се критична секция**
- ◆ **Monitor.TryEnter();**

КРИТИЧНИ СЕКЦИИ – пример

```
public class MonitorEnterExit
{
    private int mCounter;

    public void CriticalSection()
    {
        Monitor.Enter(this);
        mCounter = 0;
        try
        {
            for(int i = 1; i <= 5; i++) {
                Console.WriteLine(++mCounter);
                Thread.Sleep(1000);
            }
        }
        finally
        {
            Monitor.Exit(this);
        }
    }
}
```

(примерът продължава)

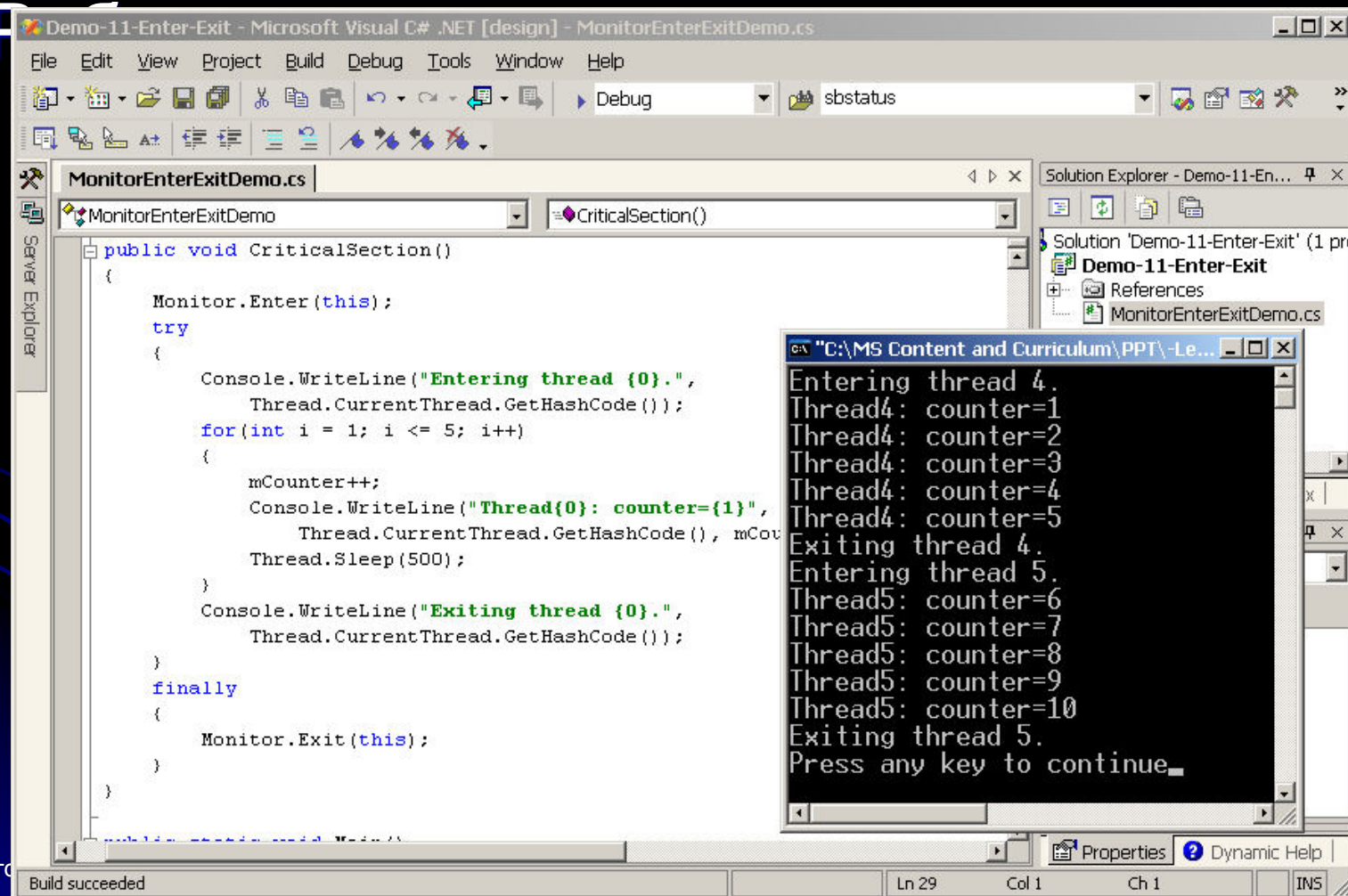
Критични секции – пример

```
static void Main() {  
    MonitorEnterExit mee = new MonitorEnterExit();  
    Thread thread1 = new Thread(new  
        ThreadStart(mee.CriticalSection));  
    thread1.Start();  
    Thread thread2 = new Thread(new  
        ThreadStart(mee.CriticalSection));  
    thread2.Start();  
}
```

// Резултат: 1234512345

// Резултат без синхронизация: 1123456789

Демонстрация #11



Няколко важни метода

- `Monitor.Wait(object)`
 - Освобождава монитора на обекта и блокира нишката докато не го получи
 - Може да се задава `Timeout`
 - Нишката се нарежда на `waiting queue`
 - Чака `Pulse(object)`, `PulseAll(object)`
- `Monitor.Pulse(object)`
 - Нишката преминава в `ready queue` (има право да вземе монитора на обект)
 - Вика се само от текущия собственик на монитора на обекта (от критична секция)
- `Monitor.PulseAll(object)`

Wait() и Pulse() – пример

- Имаме две нишки, които съвместно изпълняват някаква задача
- Когато едната прави нещо, другата я

изчаква и обратното

```
public class WaitPulse
{
    private object mSync;
    private string mName;

    public WaitPulse(string aName,
        object aSync)
    {
        mName = aName;
        mSync = aSync;
    }
}
```

фдгдфсдф

(примерът продължава)

Wait () и Pulse () – пример

```
public void DoJob()  
{  
    lock (mSync)  
    {  
        Monitor.Pulse(mSync);  
        for(int i = 1; i <= 3; i++)  
        {  
            Console.WriteLine("{0}: Pulse", mName);  
            Monitor.Pulse(mSync);  
  
            Console.WriteLine("{0}: Wait", mName);  
            Monitor.Wait(mSync);  
  
            Console.WriteLine("{0}: WokeUp", mName);  
            Thread.Sleep(1000);  
        }  
    }  
}
```

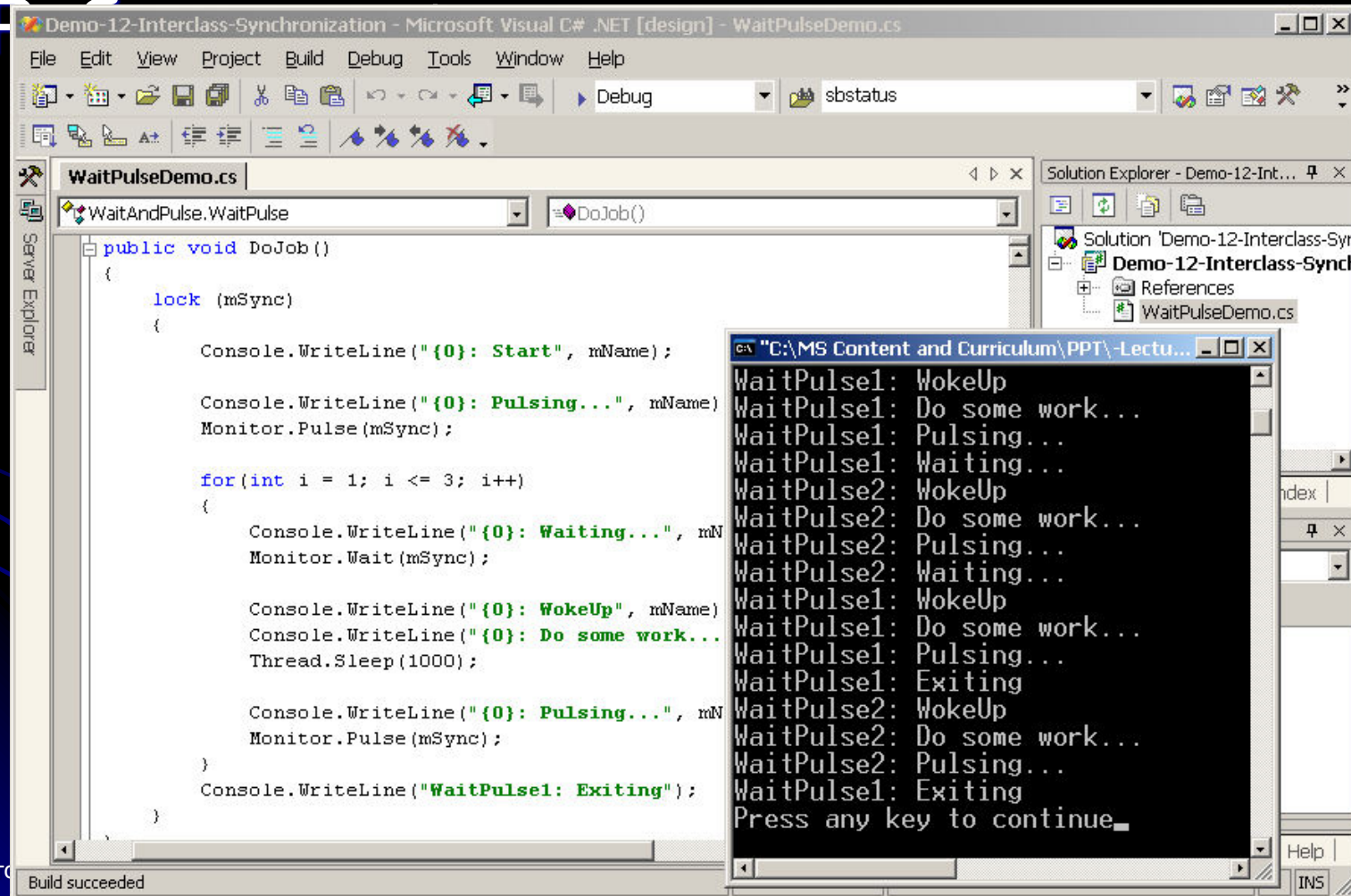
Wait() и Pulse() – пример

```
public class WaitPulseDemo
{
    public static void Main(String[] args)
    {
        object sync = new object();

        WaitPulse wp1 = new WaitPulse(
            "WaitPulse1", sync);
        Thread thread1 = new Thread(
            new ThreadStart(wp1.DoJob));
        thread1.Start();

        WaitPulse wp2 = new WaitPulse(
            "WaitPulse2", sync);
        Thread thread2 = new Thread(
            new ThreadStart(wp2.DoJob));
        thread2.Start();
    }
}
```

Демонстрация #12




```
[MethodImplAttribute  
(MethodImplOptions.Synchronized)]
```

- Прилича на lock върху цял метод
- Може да синхронизира и static

ЧЛЕНОВЕ

```
[MethodImpl(MethodImplOptions.Synchronized)]  
public void DoSomeTask1()  
{  
    Console.WriteLine("job1 started");  
    for (int i=0; i<1000000000; i++)  
        Math.Sqrt(i);  
    Console.WriteLine("Job1 done.\n");  
}
```

фдгдфгсдф

Unmanaged Synchronization

- `WaitHandle`
 - Unmanaged механизъм за синхронизация
 - Използва обекти на операционната система – за изчакване на ресурси
 - Предоставя възможности отвъд тези на CLR – `WaitAll()`, `WaitAny()`
 - Не се препоръчва – не е managed
- Има няколко наследника:
 - `Mutex`
 - `AutoResetEvent`
 - `ManualResetEvent`

Класът `WaitHandle`

- Неговите методи се използват за изчакване на събития
- Състояния: `signaled`, `nonsignaled`
- `WaitAll()` (`static`)
 - Изчаква сигнал от всички събития от масив
- `WaitAny()` (`static`)
 - Изчаква първото получило сигнал
- `WaitOne()`
 - Изчаква текущото събитие да приключи
- Прилики и разлики с класа `Monitor`

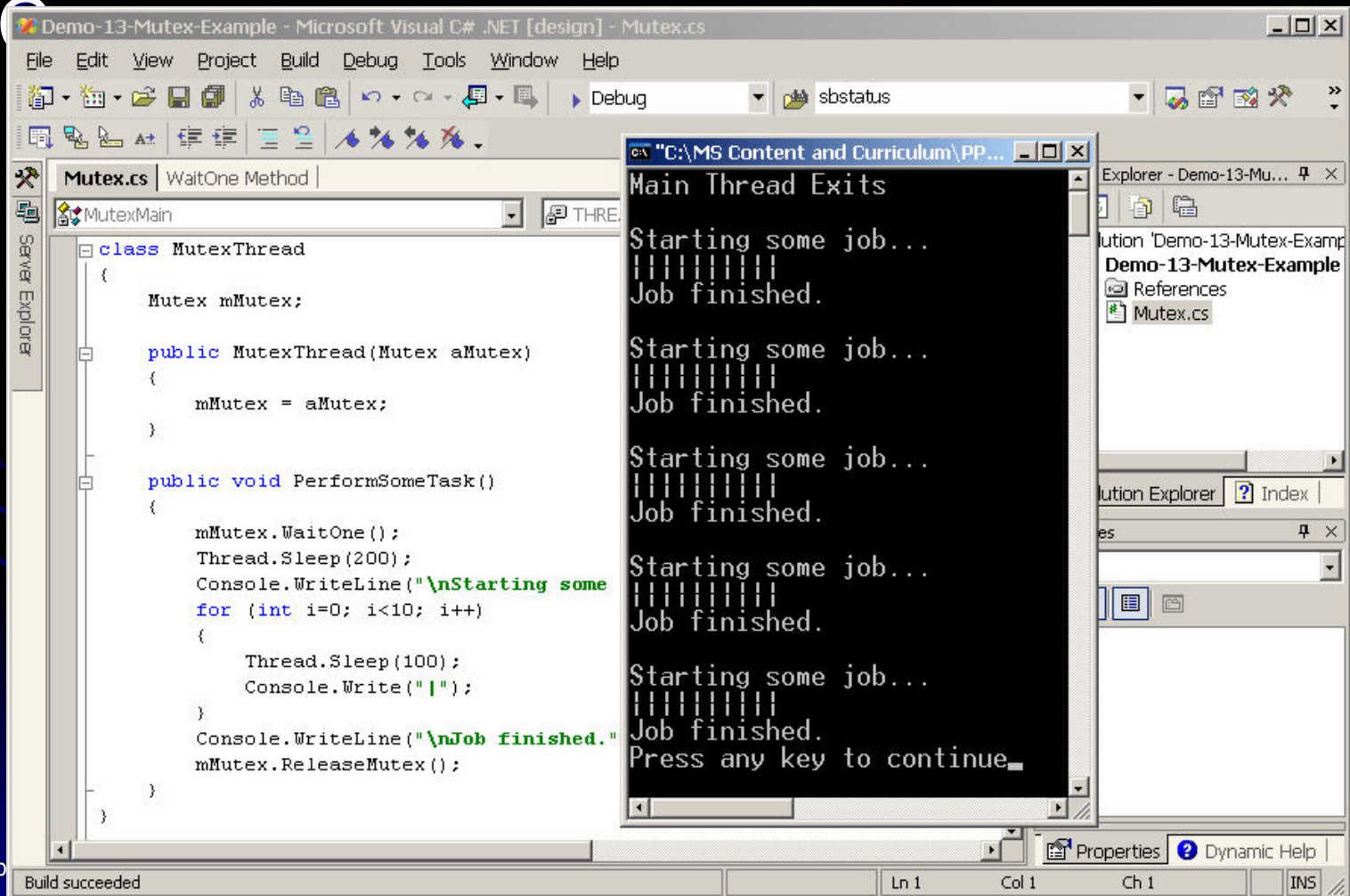
Mutex

- Наследник на `WaitHandle`
- Примитив за синхронизация на OS
- Енкапсулира Win32 synchronization handles
- `Mutex.WaitOne()` ;
- `Mutex.ReleaseMutex()` ;
 - Трябва да се извика, същия брой пъти като `Mutex.WaitOne()` ;
- `WaitHandle.WaitAll()` – изчаква два или няколко handle-a

Mutex – пример

```
class MutexDemo
{
    Mutex mMutex;
    public MutexDemo(Mutex aMutex)
    {
        mMutex = aMutex;
    }
    public void PerformSomeTask()
    {
        m.WaitOne();
        Console.WriteLine("\nJob started...");
        for( int i=0; i<10; i++)
        {
            Thread.Sleep(100);
            Console.Write("|");
        }
        Console.WriteLine("\nJob finished.");
        m.ReleaseMutex();
    }
}
```

Демонстрация #13



AutoResetEvent, ManualResetEvent

- Наследници на `WaitHandle`
- Примитиви за синхронизация
- Енкапсулират Win32 synchronization handles
- Сигнализиране със `Set()`
 - `AutoResetEvent` сигнализира само първия манипулатор(`handle`)
 - Минава автоматично в nonsignaled state
 - `ManualResetEvent()` сигнализира всички чакащи манипулатори
 - Остава в signaled state, докато някой не го промени

AutoResetEvent/ManualResetEvent

```
using System;
using System.Threading;
class OneWhoWaits
{
    WaitHandle mWaitHandle;
    int mWaitTime;
    public OneWhoWaits(WaitHandle aWaitHandle,
        int waitTime )
    {
        mWaitHandle = aWaitHandle;
        mWaitTime = mWaitTime;
    }
    public void performSomeTask()
    {
        Thread.Sleep(mWaitTime);
        Console.WriteLine("Thread {0} waiting",
            Thread.CurrentThread.GetHashCode());
        mWaitHandle.WaitOne();
    }
}
```

(примерът продължава)

AutoResetEvent/ManualResetEvent

```
class MainClass
{
    static void Main()
    {
        ManualResetEvent evnt = new
            ManualResetEvent(false);
        for (int i=0; i<10; i++ )
        {
            OneWhoWaits oww =
                new OneWhoWaits(evnt, (i+1)*500);
            Thread thread = new Thread(
                new ThreadStart(oww.performSomeTask));
            thread.Start();
        }
        for (int i=0; i<10; i++)
        {
            Console.ReadLine();
            evnt.Set();
        }
    }
}
```

фддфгсдф

Демонстрация #14

- Синхронизация с AutoResetEvent и

Ма

Demo-14-Reset-Event-Example - Microsoft Visual C# .NET [design] - ResetEvent.cs

File Edit View Project Build Debug Tools Window Help

sbstatus

ResetEvent.cs

```
class OneWhoWaits
{
    WaitHandle mWaitHandle;
    int mWaitTime;

    public OneWhoWaits(WaitHandle aWaitHandle, int aWaitTime)
    {
        mWaitHandle = aWaitHandle;
        mWaitTime = aWaitTime;
    }

    public void PerformSomeTask()
    {
        Console.WriteLine("Thread {0} started and sleeps.", Thread.CurrentThread.GetHashCode());
        Thread.Sleep(mWaitTime);
        Console.WriteLine("Thread {0} woke up and is now waiting.", Thread.CurrentThread.GetHashCode());
        mWaitHandle.WaitOne();
        Console.WriteLine("Thread {0} was signaled and exits.", Thread.CurrentThread.GetHashCode());
    }
}
```

Thread 4 started and sleeps.
Thread 5 started and sleeps.
Thread 6 started and sleeps.
Thread 7 started and sleeps.
Thread 8 started and sleeps.
Thread 4 woke up and is now waiting.
Thread 4 was signaled and exits.
Thread 5 woke up and is now waiting.
Thread 6 woke up and is now waiting.
Thread 5 was signaled and exits.
Thread 7 woke up and is now waiting.
Thread 6 was signaled and exits.
Thread 8 woke up and is now waiting.
Thread 7 was signaled and exits.
Thread 8 was signaled and exits.
Press any key to continue.

Build succeeded

Ln 1 Col 1 Ch 1

Специални класове

- `System.Threading.Interlocked`
 - Атомарни операции
- `System.Threading.ThreadPool`
 - Бърз достъп до нишки
 - Преизползваемост на нишките
 - Програмистът не се грижи за живота на нишките (създаване / унищожаване)
- `System.Threading.ReaderWriterLock`
 - Класически синхронизационни проблеми – Reader/Writer Problem
- `Synchronized Wrappers`

System.Threading.Interlocked

- Осигурява изпълнение на атомарни операции – увеличение с 1, намаление с 1, размяна, сравнение и др.
- Няма нужда синхронизация на споделените данни
- В по-простите случаи синхронизацията може да се избегне с тези методи
- Методите не хвърлят изключения

System.Threading.Interlocked

- Increment/Decrement
 - Атомарна операция за разлика от $i++/i--$
- Exchange
 - Записва стойността на вторият параметър в първия, връща оригинала
- CompareExchange
 - Има три параметъра
 - Проверява дали първият и третият са равни, ако да, записва втория в първия
 - Използва се при работа с временни променливи

Interlocked – пример

```
class TestInterlockedIncrement
{
    static long mUnsafeCounter = 0;
    static long mSafeCounter = 0;

    private static void DoTask()
    {
        while (true)
        {
            mUnsafeCounter++;
            Interlocked.Increment(ref mSafeCounter);

            if (mSafeCounter % 10000000 == 0)
            {
                Console.WriteLine("Safe={0}, Unsafe={1}",
                    mSafeCounter, mUnsafeCounter);
            }
        }
    }
}
```

фдгдфгсдф

Interlocked – пример

```
static void Main(string[] args)
{
    for (int i=0; i<5; i++)
    {
        Thread thread = new Thread(
            new ThreadStart(DoTask));
        thread.Start();
    }
}
```

// Результат:

```
// Safe=10000000, Unsafe=5846325
// Safe=20000000, Unsafe=15846326
// Safe=30000000, Unsafe=25846326
// Safe=40000000, Unsafe=35846325
// Safe=50000000, Unsafe=41356463
// ...
```

Thread Pooling

- Много нишки прекарват по-голямата част от живота си спейки (`ThreadState.WaitSleepJoin`) или чакайки някакво събитие
- Други се “събуждат” само за малки периоди, за да проверят истинността на някакво условие или за да свършат нещо дребно
- Поддържането на много неактивни нишки е излишно, неефективно и консумира ресурси

Thread Pooling

- ТР е подход за намаляване на товара при създаване и унищожаване на нишки
- При ТР се създават група нишки в началото на многонишково приложение
- Наричат се “работни нишки” (worker threads)
- Всяка нишка “живее” в т.нар. Thread Pool
- При нова задача, се използва worker thread, след това се връща обратно в пула
- ТР се грижи за живота и разпределението на задачите върху нишките
 - Използващият нишки се освобождава от този товар

System.Threading.ThreadPoo

1

- .NET Framework имплементира механизма Thread Pooling в класа ThreadPool
- Използва се, когато трябва да се свършат много на брой кратки задачи, които могат да работят паралелно
- Задачите се нареждат на опашка и се изпълняват по няколко едновременно (според броя worker threads)
- По подразбиране в Thread Pool-ът има лимит от 25 нишки на процесор
- Един процес може да има само един Thread Pool – той е общ за всички App Domains

System.Threading.ThreadPoo

- Thread Pool-ът за даден процес се създава се при първото:
 - извикване на `QueueUserWorkItem`
 - регистриране на таймер
 - регистриране на изчакваща операция с `callback` метод
- .NET Framework използва Thread Pooling за:
 - асинхронни извиквания
 - асинхронен вход/изход
 - работа с таймери
 - работа със сокети
 - изчакващи операции

System.Threading.ThreadPoo

- Ако дадена задача е ¹ в Thread Pool, не може да се премахне от него
- Thread Pooling позволява на OS да оптимизира използването на нишки, тъй като броят им е постоянен
- Случаи, в които НЕ се препоръчва използването на Thread Pool
 - Ако е нужна контролираща нишка
 - При задачи, отнемащи много време – може да се блокират другите задачи
 - Ако има нужда от синхронизация

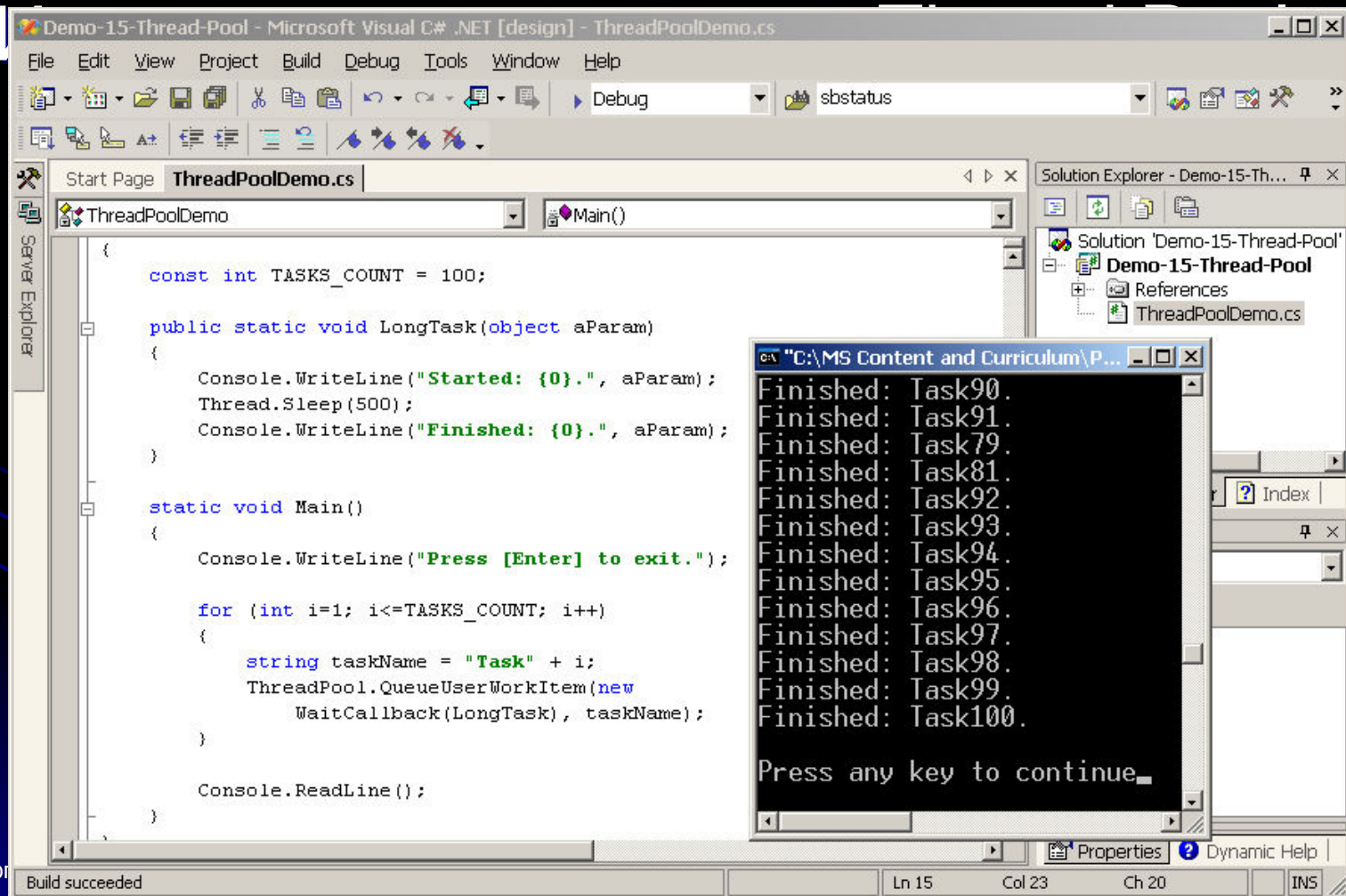
ThreadPool.QueueUserWorkItem

```
class ThreadPoolDemo
{
    public static void LongTask(object aParam)
    {
        Console.WriteLine("Started: {0}.", aParam);
        Thread.Sleep(500);
        Console.WriteLine("Finished: {0}.", aParam);
    }

    static void Main()
    {
        for (int i=1; i<=100; i++)
        {
            string taskName = "Task" + i;
            ThreadPool.QueueUserWorkItem(new
                WaitCallback(LongTask), taskName);
        }

        Console.ReadLine();
    }
}
```

Демонстрация #15



ThreadPool.RegisterWaitForSingleObject

- Регистрира делегат, който чака за събитие (да бъде сигнализиран)
- Активира се при:
 - сигнализиране
 - при настъпване на зададен timeout
- Процесът по изчакване и извикване се управлява се от Thread Pool-а
- Може да настрои да се изпълнява делегатът многократно

ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev = new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes; cause =
{0}", cause);
    }
}
```


ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev = new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes; cause =
{0}", cause);
    }
}
```

ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev = new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes; cause =
{0}", cause);
    }
}
```


ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev = new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes; cause =
{0}", cause);
    }
}
```

фдг,фсдф

ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev = new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes; cause =
{0}", cause);
    }
}
```

ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev = new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes; cause =
{0}", cause);
    }
}
```

ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev = new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes; cause =
{0}", cause);
    }
}
```

ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev = new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes; cause =
{0}", cause);
    }
}
```


ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev = new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes; cause =
{0}", cause);
    }
}
```

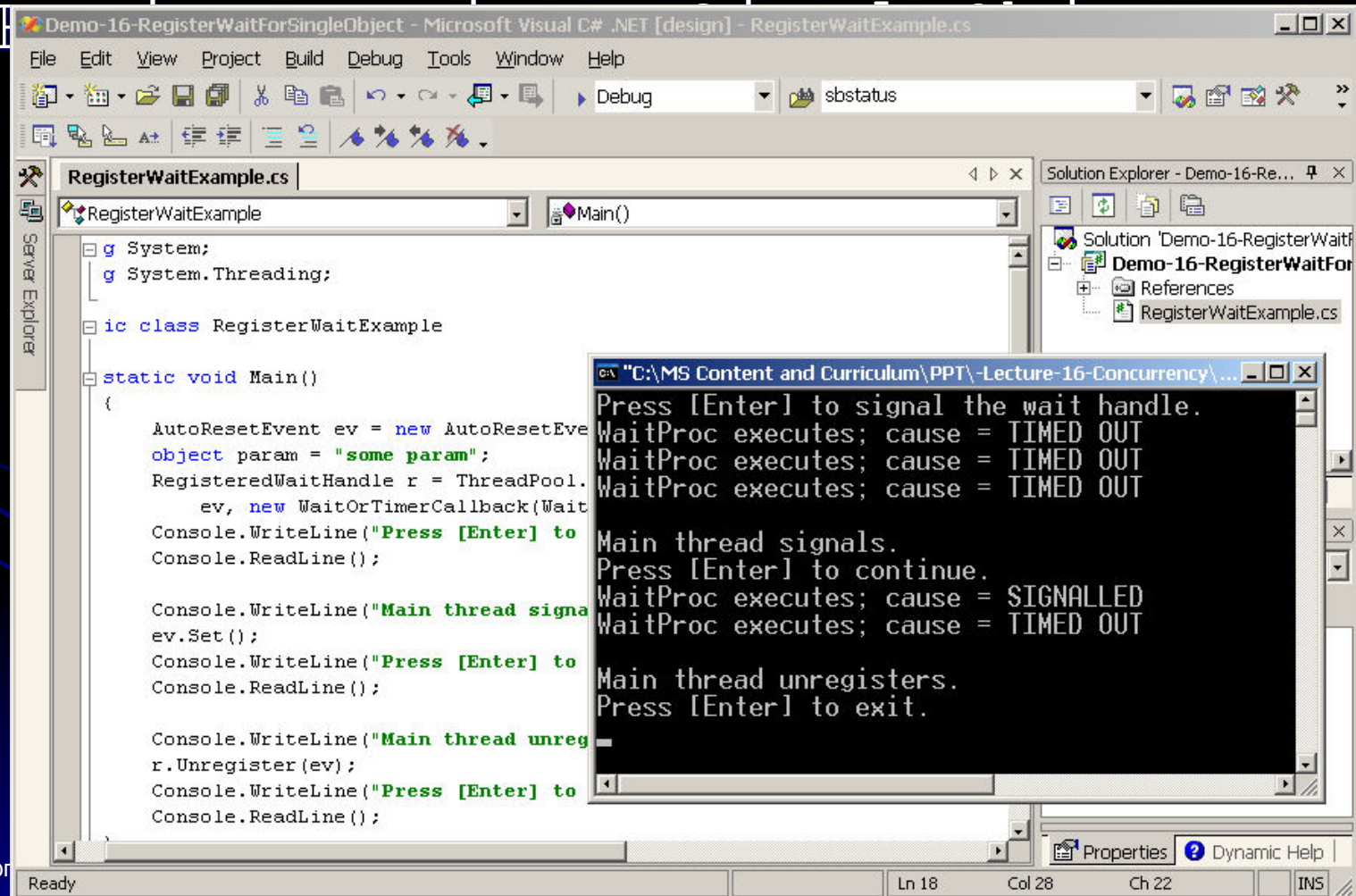
ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev = new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes; cause =
{0}", cause);
    }
}
```

ThreadPool.RegisterWaitForSingleObject

```
using ...
public class Example {
    public static void Main(string[] args) {
        AutoResetEvent ev = new AutoResetEvent(false);
        object param = "some param";
        RegisteredWaitHandle r =
            ThreadPool.RegisterWaitForSingleObject(
                ev, new WaitOrTimerCallback(WaitProc),
                param, 1000, false );
        Console.ReadLine();
        Console.WriteLine("signaling.");
        ev.Set();
        Console.ReadLine();
        Console.WriteLine("unregister wait");
        r.Unregister(ev);
        Console.ReadLine();
    }
    public static void WaitProc(object param, bool
timedOut) {
        string cause = "SIGNALLED";
        if (timedOut)
            cause = "TIMED OUT";
        Console.WriteLine("WaitProc executes; cause =
{0}", cause);
    }
}
```


Демонстрация #16



Класически синхр. проблеми

- The Producer-Consumer Problem
 - Един консуматор и един производител споделят обща опашка (shared queue)
 - Изисквания:
 - Ако опашката е празна, консуматорът блокира докато се появи нещо в нея
 - Ако опашката е пълна, производителят блокира докато не се опразни място
 - Производителят и консуматорът не могат да достъпват опашката едновременно
 - Пример: пощенска кутия
 - Нарича се още *bounded buffer problem*
- Няма стандартен клас в .NET

Класически синхр. проблеми

- The Readers-Writers Problem

- Един или повече четеща и един или повече писеща достъпват общ ресурс
- Пример: достъп до общ файл в OS
- Условия на Бернщайн:
 - Няколко четци могат да достъпват общия ресурс
 - Когато писец достъпва ресурса, нито четец нито друг писец може да го достъпва
 - Очаква се нито един писец/четец да не чака безкрайно дълго за споделения ресурс

- В .NET – `ReaderWriterLock`

System.Threading.ReaderWriterLock

- Имплементира най-популярния синхронизационен проблем – Reader/Writer Problem

- IsReaderLockHeld

- IsWriterLockHeld

- AcquireReaderLock

- AcquireWriterLock

- ReleaseReaderLock

- ReleaseWriterLock

System.Threading.ReaderWriterLock

```
class Resource {  
    ReaderWriterLock rwl = new ReaderWriterLock();  
    public void Read() {  
        rwl.AcquireReaderLock(Timeout.Infinite);  
        try {  
            // Many can read, writers are blocked  
        }  
        finally {  
            rwl.ReleaseReaderLock();  
        }  
    }  
    public void Write() {  
        rwl.AcquireWriterLock(Timeout.Infinite);  
        try {  
            // One can write, readers blocked  
        }  
        finally {  
            rwl.ReleaseWriterLock();  
        }  
    }  
}
```

Класически синхр. проблеми

- The Dining Philosophers Problem

- Няколко философа стоят около кръгла маса и се хранят или мислят
- Когато се хранят, се нуждаят и от двете клечки от двете им страни
- Условия:
 - Не трябва никой философ да умре от глад
 - Не трябва да се позволява deadlock – когато всеки философ има по една клечка и чака за другата



CustomReaderWriter

```
class CustomReaderWriter
{
    private int mReaders = 0;
    private bool mIsWriting = false;
    public void Read()
    {
        lock( this )
        {
            while (mIsWriting) Monitor.Wait( this );
            mReaders++;
        }

        //...READING TAKES PLACE HERE...

        lock( this )
        {
            mReaders--;
            if( mReaders == 0 ) Monitor.Pulse(this);
        }
    }
}
```

фдгафгсдф

(примерът продължава)

CustomReaderWriter

```
class CustomReaderWriter
{
    private int mReaders = 0;
    private bool mIsWriting = false;
    public void Read()
    {
        lock( this )
        {
            while (mIsWriting) Monitor.Wait( this );
            mReaders++;

            //...READING TAKES PLACE HERE...

            lock( this )
            {
                mReaders--;
                if( mReaders == 0 ) Monitor.Pulse(this);
            }
        }
    }
}
```

фдгдфгсдф

(примерът продължава)

CustomReaderWriter

```
class CustomReaderWriter
{
    private int mReaders = 0;
    private bool mIsWriting = false;
    public void Read()
    {
        lock( this )
        {
            while (mIsWriting) Monitor.Wait( this );
            mReaders++;

            //...READING TAKES PLACE HERE...

            lock( this )
            {
                mReaders--;
                if( mReaders == 0 ) Monitor.Pulse(this);
            }
        }
    }
}
```

фдгдфгсдф

(примерът продължава)

CustomReaderWriter

```
class CustomReaderWriter
{
    private int mReaders = 0;
    private bool mIsWriting = false;
    public void Read()
    {
        lock( this )
        {
            while (mIsWriting) Monitor.Wait( this );
            mReaders++;

            //...READING TAKES PLACE HERE...

            lock( this )
            {
                mReaders--;
                if( mReaders == 0 ) Monitor.Pulse(this);
            }
        }
    }
}
```

фдгдфгсдф

(примерът продължава)

CustomReaderWriter

```
public void Write()  
{  
    lock( this )  
    {  
        while( mReaders != 0 )  
            Monitor.Wait( this );  
        mIsWriting = true;  
    }  
  
    //...WRITING TAKES PLACE HERE...  
  
    lock( this )  
    {  
        mIsWriting = false;  
        Monitor.PulseAll( this );  
    }  
}  
}
```

фдгдфгсдф

CustomReaderWriter

```
public void Write()
{
    lock( this )
    {
        while( mReaders != 0 )
            Monitor.Wait( this );
        mIsWriting = true;

        //...WRITING TAKES PLACE HERE...

        lock( this )
        {
            mIsWriting = false;
            Monitor.PulseAll( this );
        }
    }
}
} фдгдфгсдф
```

CustomReaderWriter

```
public void Write()
{
    lock( this )
    {
        while( mReaders != 0 )
            Monitor.Wait( this );
        mIsWriting = true;

        //...WRITING TAKES PLACE HERE...

        lock( this )
        {
            mIsWriting = false;
            Monitor.PulseAll( this );
        }
    }
}
} фдгдфгсдф
```

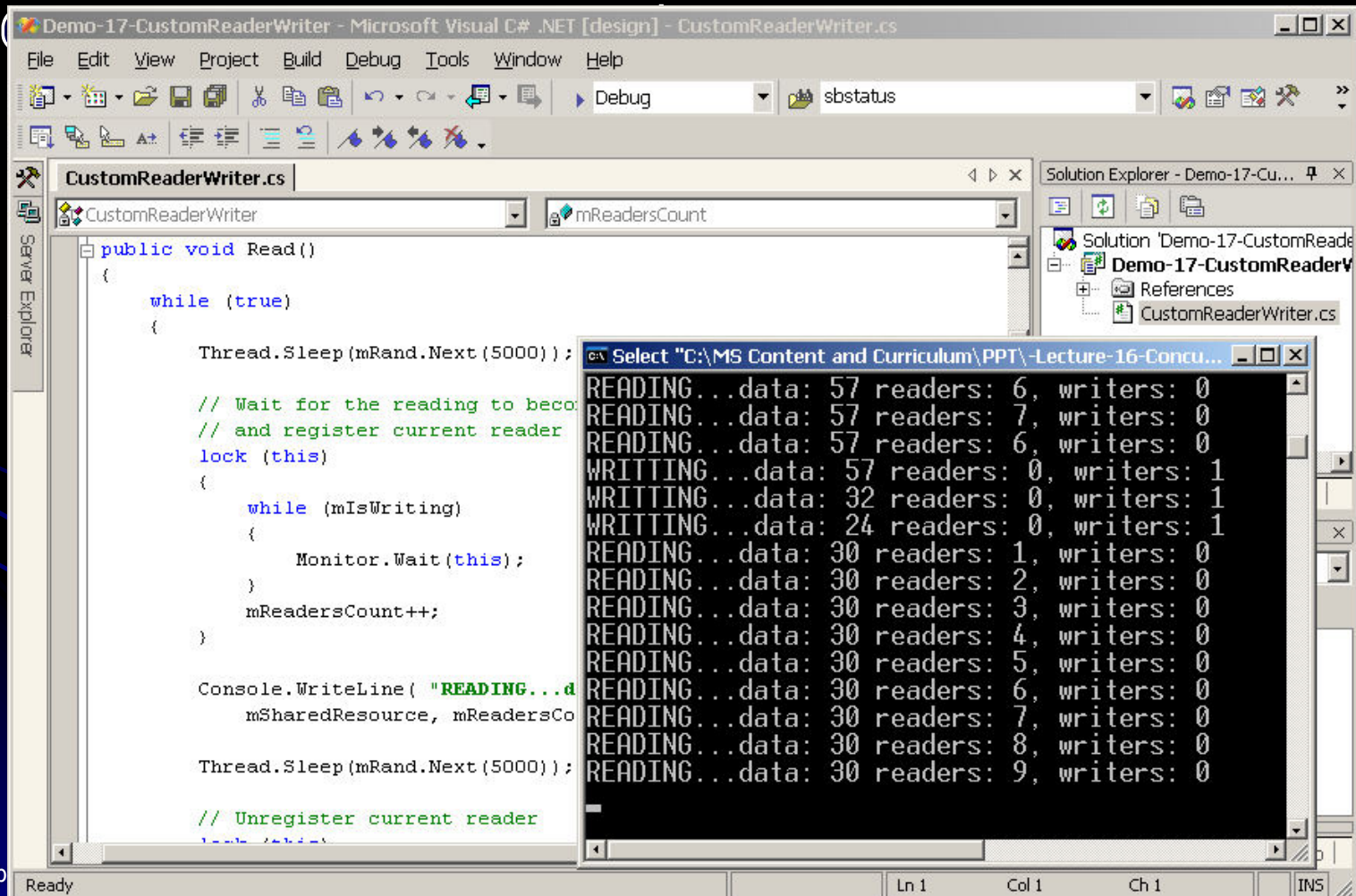
CustomReaderWriter

```
public void Write()
{
    lock( this )
    {
        while( mReaders != 0 )
            Monitor.Wait( this );
        mIsWriting = true;

        //...WRITING TAKES PLACE HERE...

        lock( this )
        {
            mIsWriting = false;
            Monitor.PulseAll( this );
        }
    }
}
} фдгдфгсдф
```

Демонстрация #17



CustomReaderWriter.cs

```
public void Read()  
{  
    while (true)  
    {  
        Thread.Sleep(mRand.Next(5000));  
  
        // Wait for the reading to become available  
        // and register current reader  
        lock (this)  
        {  
            while (mIsWriting)  
            {  
                Monitor.Wait(this);  
            }  
            mReadersCount++;  
        }  
  
        Console.WriteLine("READING...data: {0} readers: {1}, writers: {2}",  
            mSharedResource, mReadersCount, mWritersCount);  
  
        Thread.Sleep(mRand.Next(5000));  
  
        // Unregister current reader  
        lock (this)  
        {  
            mReadersCount--;  
        }  
    }  
}
```

Console Output:

```
READING...data: 57 readers: 6, writers: 0  
READING...data: 57 readers: 7, writers: 0  
READING...data: 57 readers: 6, writers: 0  
WRITING...data: 57 readers: 0, writers: 1  
WRITING...data: 32 readers: 0, writers: 1  
WRITING...data: 24 readers: 0, writers: 1  
READING...data: 30 readers: 1, writers: 0  
READING...data: 30 readers: 2, writers: 0  
READING...data: 30 readers: 3, writers: 0  
READING...data: 30 readers: 4, writers: 0  
READING...data: 30 readers: 5, writers: 0  
READING...data: 30 readers: 6, writers: 0  
READING...data: 30 readers: 7, writers: 0  
READING...data: 30 readers: 8, writers: 0  
READING...data: 30 readers: 9, writers: 0
```

Synchronized Wrappers

- По подразбиране стандартните колекции (ArrayList, Queue, Stack) не са thread safe
- Hashtable е thread safe (1 писач, N четеца)
- Synchronized Wrapper е синхронизирана обвивка около колекцията

- Връща синхронизирана версия на колекцията

```
Stack safeStack = Stack.Synchronized(stack) ;
```

- При Hashtable – threadsafe (N писача, N четеца)

- СВОЙСТВОТО ICollection.SyncRoot

```
lock (bitArray.SyncRoot)  
{ // Perform thread unsafe operations here }
```


Нишки – препоръчвани практики

- Избягвайте нуждата от синхронизация, когато е възможно
- Използвайте `Interlocked` вместо синхронизация, където е възможно
- Пазете се от `deadlocks`
- Избягвайте `lock (..) { }`, ако е възможно
- Правете критичните секции възможно най-кратки (за да намалите изчакването)
- Член-променливите не е нужно да са `thread safe`, ако не се достъпват от други нишки
- Статичните членове е нужно да са `thread safe`
- Използвайте статични данни само за четене (`read-only`)

Асинхронни извиквания – съдържание

- Многозадачност
- Нишки
- Синхронизация
- Асинхронни извиквания
 - Асинхронни извиквания на методи
 - Къде се поддържа?
 - Асинхронно извикване чрез делегат
 - Design Pattern за асинхр. извиквания
 - Интерфейсът `IAsyncResult`

Асинхронни извиквания на методи

- По подразбиране един метод се дефинира като синхронен (synchronous)
- Синхронно извикване на метод (synchronous method call)
 - Изчаква се неговото приключване и след това се преминава към следващия оператор
- Асинхронно извикване на метод (asynchronous method call)
 - Без да се чака приключването на метода, се минава на следващия оператор
- Механизмът на асинхронното извикване използва нишки (ThreadPool)

Къде се поддържа?

- Вход/изход: File IO, Stream IO, Socket IO
- Мрежови класове: HTTP, TCP
- Remoting канали (HTTP, TCP) и проксита
- ASP.NET XML Web-услуги
- ASP.NET Web-приложения
- При работа с MSMQ (Microsoft Message Queue)
- Асинхронни делегати
- Потребителски класове, дефиниращи асинхронен интерфейс за извикване

Асинхронно извикване чрез делегат

- Делегатите предоставят функционалност за асинхронно извикване на синхронен метод
 - Автоматично се генерират `BeginInvoke()` и `EndInvoke()` с правилния брой параметри
 - Генерирането става при създаването на делегат със съответната сигнатура
- Добавя се функционалност без да се изисква нито ред допълнителен код

Асинхронно извикване чрез делегат

```
using System;
using System.Threading;

class AsyncCall
{
    public delegate int MyDelegate(int a, int b);
    public int Sum(int a, int b)
    {
        Thread.Sleep(3000);
        return a+b;
    }

    static void Main(string[] args)
    {
        MyDelegate asyncCall = new MyDelegate(new
            AsyncCall().Sum);
        Console.WriteLine("Starting method async.");
    }
}
```

Асинхронно извикване чрез делегат

```
IAAsyncResult status =  
    asyncCall.BeginInvoke(5, 6, null, null);  
Console.WriteLine("Async method is working");  
  
Console.WriteLine("Calling EndInvoke()");  
int result = asyncCall.EndInvoke(status);  
Console.WriteLine("EndInvoke() returned");  
Console.WriteLine("Result={0}", result);  
}
```

```
}
```

// Резултат:

Starting method asynchronously

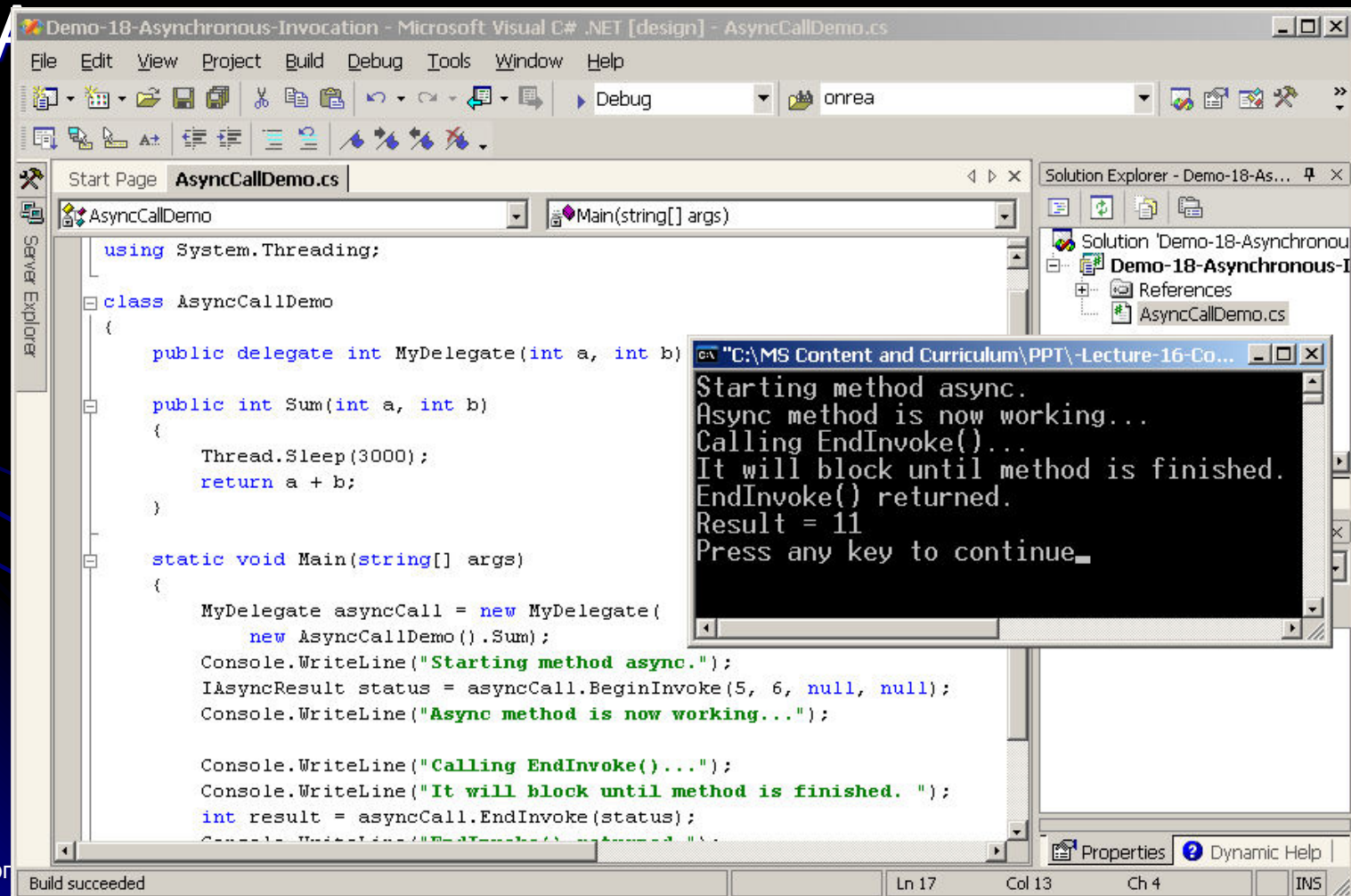
Asynchronous method is working now

Calling EndInvoke() // тук има пауза, заради Sleep()

EndInvoke() returned

Result=11

Демонстрация #18



Design Pattern за асинхронно програмиране

- Случаи, в които се налага изричното имплементиране на асинхронно извикване на метод
 - Ако изричното имплементиране е по-бързо от извикване с делегат
 - Ако методът трябва да се извиква само асинхронно
- В такива случаи строго се препоръчва следването на design pattern-a

Асинхронно програмиране

- Нека имаме синхронен метод

```
int Sum(int a, int b);
```

- Искаме да предоставим асинхронна версия
- В .NET Framework асинхронното извикване се характеризира с използването на методите BeginXXXXX и EndXXXXX, където XXXXX е синхронната версия на метода
- Асинхронната версия на метода Sum е:

```
IASyncResult BeginSum(int a, int b,  
    AsyncCallback requestCallback,  
    object stateObject);
```

```
int EndSum(IASyncResult ar);
```

Асинхронно програмиране

- Нека имаме синхронен метод

```
int Sum(int a, int b);
```

- Искаме да предоставим асинхронна версия
- В .NET Framework асинхронното извикване се характеризира с използването на методите BeginXXXXX и EndXXXXX, където XXXXX е синхронната версия на метода
- Асинхронната версия на метода Sum е:

```
IASyncResult BeginSum(int a, int b,  
    AsyncCallback requestCallback,  
    object stateObject);
```

```
int EndSum(IASyncResult ar);
```

Асинхронно програмиране

- Нека имаме синхронен метод

```
int Sum(int a, int b);
```

- Искаме да предоставим асинхронна версия
- В .NET Framework асинхронното извикване се характеризира с използването на методите BeginXXXXX и EndXXXXX, където XXXXX е синхронната версия на метода
- Асинхронната версия на метода Sum е:

```
IASyncResult BeginSum(int a, int b,  
    AsyncCallback requestCallback,  
    object stateObject);
```

```
int EndSum(IASyncResult ar);
```

Асинхронно програмиране

- Нека имаме синхронен метод

```
int Sum(int a, int b);
```

- Искаме да предоставим асинхронна версия
- В .NET Framework асинхронното извикване се характеризира с използването на методите BeginXXXXX и EndXXXXX, където XXXXX е синхронната версия на метода
- Асинхронната версия на метода Sum е:

```
IASyncResult BeginSum(int a, int b,  
    AsyncCallback requestCallback,  
    object stateObject);
```

```
int EndSum(IASyncResult ar);
```

Асинхронно програмиране

- Нека имаме синхронен метод

```
int Sum(int a, int b);
```

- Искаме да предоставим асинхронна версия
- В .NET Framework асинхронното извикване се характеризира с използването на методите BeginXXXXX и EndXXXXX, където XXXXX е синхронната версия на метода
- Асинхронната версия на метода Sum е:

```
IASyncResult BeginSum(int a, int b,  
    AsyncCallback requestCallback,  
    object stateObject);
```

```
int EndSum(IASyncResult ar);
```

```
delegate void AsyncCallback(IASyncResult ar);
```

Асинхронно програмиране

- Нека имаме синхронен метод

```
int Sum(int a, int b);
```

- Искаме да предоставим асинхронна версия
- В .NET Framework асинхронното извикване се характеризира с използването на методите BeginXXXXX и EndXXXXX, където XXXXX е синхронната версия на метода
- Асинхронната версия на метода Sum е:

```
IASyncResult BeginSum(int a, int b,  
    AsyncCallback requestCallback,  
    object stateObject);
```

```
int EndSum(IASyncResult ar);
```


Асинхронно програмиране

- Нека имаме синхронен метод

```
int Sum(int a, int b);
```

- Искаме да предоставим асинхронна версия
- В .NET Framework асинхронното извикване се характеризира с използването на методите BeginXXXXX и EndXXXXX, където XXXXX е синхронната версия на метода
- Асинхронната версия на метода Sum е:

```
IASyncResult BeginSum(int a, int b,  
    AsyncCallback requestCallback,  
    object stateObject);
```

```
int EndSum(IASyncResult ar);
```


IAsyncResult

```
public interface IAsyncResult
{
    object AsyncState {get;}
    WaitHandle AsyncWaitHandle {get;}
    bool CompletedSynchronously {get;}
    bool IsCompleted {get;}
}
```

- Свойството AsyncState, връща същия обект подаден като stateObject на BeginInvoke()
- Предоставя се, за да се използва от потребителя, като начин за следене на статуса
- Не се използва или променя по какъвто и да било начин от извиквания метод

IAsyncResult

```
public interface IAsyncResult
{
    object AsyncState {get;}
    WaitHandle AsyncWaitHandle {get;}
    bool CompletedSynchronously {get;}
    bool IsCompleted {get;}
}
```

- Свойството AsyncWaitHandle се използва като параметър на методите WaitAll(), WaitOne() или WaitAny() на класа WaitHandle за изчакване приключването на асинхронния метод
- Гореописаният механизъм е един от начините за приключване на асинхронен метод

IAsyncResult

```
public interface IAsyncResult
{
    object AsyncState {get;}
    WaitHandle AsyncWaitHandle {get;}
    bool CompletedSynchronously {get;}
    bool IsCompleted {get;}
}
```

- CompletedSynchronously връща true ако асинхронният метод е приключил работа преди края на извикването на BeginInvoke метода (т.е. ако работи прекалено бързо)

IAsyncResult

```
public interface IAsyncResult
{
    object AsyncState {get;}
    WaitHandle AsyncWaitHandle {get;}
    bool CompletedSynchronously {get;}
    bool IsCompleted {get;}
}
```

- IsCompleted връща true ако асинхронният метод е приключил работа
- Може да се използва чрез механизма "polling" – през определено време да се проверява истинността на IsCompleted докато върне true

Изчакване на асинхронен метод

- Има 4 начина да се провери дали е приключил един асинхронен метод
 1. Механизмът "polling" – проверява се `IAsyncResult.IsCompleted` през определено време
 2. Чрез `WaitHandle.WaitOne()` / `WaitAll()` / `WaitAny()` върху `IAsyncResult.AsyncWaitHandle`
 - Може да се задава таймаут, за да не се чака безкрайно дълго
 3. Извикване на `EndInvoke()`, който блокира докато асинхронният метод не свърши работата си

Изчакване на асинхронен метод

- Има 4 начина да се провери дали е приключил един асинхронен метод

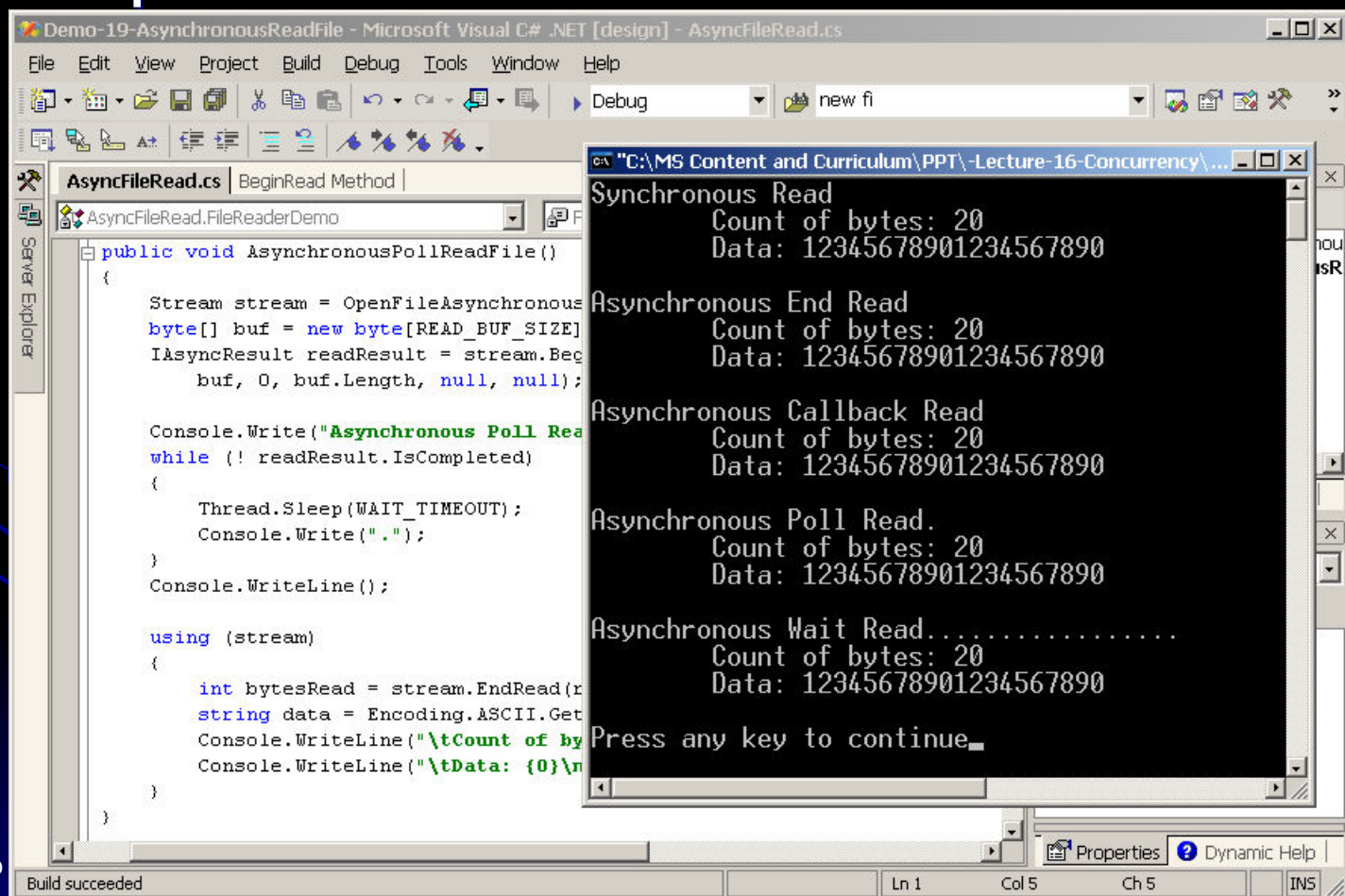
4. Да се подаде callback метод на `BeginInvoke()` чрез делегата `AsyncCallback`

```
delegate void AsyncCallback(IAsyncResult ar)
```

- Подаденият метод се извиква, когато асинхронният метод приключи работа
- От него може да се извика `EndInvoke()` за да се извлече резултата

Демонстрация #19

- 4 начина да изчакаме приключването на асинхронно извикване



The screenshot shows the Visual Studio IDE with the file AsyncFileRead.cs open. The code implements a method AsynchronousPollReadFile that demonstrates four different ways to wait for an asynchronous file read to complete. The output window shows the results of these four methods, all of which successfully read 20 bytes of data: "12345678901234567890".

```
public void AsynchronousPollReadFile()
{
    Stream stream = OpenFileAsynchronous();
    byte[] buf = new byte[READ_BUF_SIZE];
    IAsyncResult readResult = stream.BeginRead(buf, 0, buf.Length, null, null);

    Console.WriteLine("Asynchronous Poll Read");
    while (!readResult.IsCompleted)
    {
        Thread.Sleep(WAIT_TIMEOUT);
        Console.WriteLine(".");
    }
    Console.WriteLine();

    using (stream)
    {
        int bytesRead = stream.EndRead(readResult);
        string data = Encoding.ASCII.GetString(buf, 0, bytesRead);
        Console.WriteLine("\tCount of bytes: {0}\n\tData: {0}\n", bytesRead, data);
    }
}
```

Output:

```
Synchronous Read
Count of bytes: 20
Data: 12345678901234567890

Asynchronous End Read
Count of bytes: 20
Data: 12345678901234567890

Asynchronous Callback Read
Count of bytes: 20
Data: 12345678901234567890

Asynchronous Poll Read.
Count of bytes: 20
Data: 12345678901234567890

Asynchronous Wait Read.....
Count of bytes: 20
Data: 12345678901234567890

Press any key to continue.
```

Web Service

Web услуги

2010/2011

Съдържание

- ◆ **Инфраструктурата на Web услугите**
 - ❖ **Разпределени приложения**
 - ❖ **Нуждата от Web услуги**
 - ❖ **Услуги и Web услуги**
 - ❖ **UDDI директории за Web услуги**
 - ❖ **Откриване на Web услуги (DISCO)**
 - ❖ **WSDL описания на услуги**
 - ❖ **SOAP – формат на заявките**
 - ❖ **Протоколен стек на Web услугите**
 - ❖ **Сценарии за използване на Web услуги**
 - ❖ **.NET Enterprise приложения**

Съдържание (2)

- ◆ **Web услугите в ASP.NET**
 - ❖ Архитектура
 - ❖ Създаване на Web услуги
 - ❖ Публикуване на Web услуги
 - ❖ Използване на Web услуги. Генериране на прокси клас
 - ❖ Web услугите и VS.NET – създаване и консумиране
 - ❖ Атрибути за Web услугите – [WebService], [WebMethod]
 - ❖ Прехвърляне на типове

Разпределени приложения

- ◆ Повечето днешни приложения са разпределени
 - ❖ Състоят се от няколко раздалечени компонента, които си взаимодействат
- ◆ Модели за разпределени приложения
 - ❖ Модел "клиент/сървър"
 - ❖ Модел "разпределени обекти"
 - ❖ DCOM – ползва се в Microsoft Windows
 - ❖ CORBA – отворен стандарт, доста сложен
 - ❖ Java RMI – базира се на Java технологията
 - ❖ .NET Remoting – ползва се в .NET Framework
 - ❖ Модел "Web услуги"

Нуждата от Web услуги

- ◆ Моделът "разпределени обекти не е подходящ за употреба в Интернет
 - ❖ Силна свързаност между услуга и клиент
 - ❖ Трудности при хетерогенна инфраструктура
 - ❖ Проблеми с версиите
 - ❖ Защитните стени (firewalls) силно затрудняват комуникацията
 - ❖ COM/DCOM се поддържа само в Windows
 - ❖ CORBA не работи добре през Интернет
 - ❖ Java RMI / EJB се поддържат само в Java
- ◆ Решението: Web услуги

Какво е услуга?

- ◆ В реалния свят услугата е:
 - ❖ Единица работа, извършвана от доставчик на услуги
 - ❖ Предоставя някакъв желан резултат на клиента (консуматор на услуги)
 - ❖ С ясно дефинирани входни параметри и изходни резултати
 - ❖ Лесна за използване
 - ❖ Винаги достъпна
 - ❖ Има характеристики за качество (цена, време за извършване и др.)

Web-услуги

- ◆ **Web-услугите моделират услугите от реалния свят**
 - ❖ **Програмни компоненти, достъпни отдалечено през Web**
 - ❖ **Модел за изпълнение “заявка-отговор”**
 - ❖ **Клиентът поръчва, услугата изпълнява поръчката и връща резултат**
 - ❖ **Използват отворени стандарти за комуникация – HTTP, XML и SOAP**
 - ❖ **Сами описват интерфейса си за достъп чрез езика WSDL**
 - ❖ **Могат да бъдат търсени чрез UDDI**

Web-услуги

- ◆ Web-услугите моделират услугите от реалния свят
 - ❖ Работят на принципа на обмяна на SOAP съобщения
 - ❖ Съобщенията съдържат структурирана информация – данни + метаданни (описание на данните)
 - ❖ Независими от операционната система, платформата и езика за програмиране
 - ❖ Функционално-независими (loosely coupled)
 - ❖ Слабо-обвързани с клиента

Инфраструктурата

- ◆ Инфраструктурата на Web услугите се състои от следните компоненти
 - ❖ Директория (Directory)
 - ❖ UDDI
 - ❖ Откриване (Discovery)
 - ❖ DISCO
 - ❖ Описание (Description)
 - ❖ WSDL
 - ❖ Формат на заявките (Wire Format)
 - ❖ SOAP
 - ❖ XML, XSD, HTTP

Директории за Web услуги

- ◆ Директориите за Web услуги
 - ❖ Позволяват търсене на Web услуги
 - ❖ Организирани са по категории
 - ❖ Може да се търси по различни параметри
 - ❖ Използват UDDI стандарта
 - ❖ UDDI (Universal Description, Discovery and Integration)
 - ❖ Отворен стандарт на OASIS
 - ❖ Представява Web услуга, предлагаща регистрация и търсене на други Web услуги
 - ❖ Пример за UDDI директория:
 - ❖ <http://uddi.microsoft.com>

Директории – пример

Microsoft UDDI Business Registry (UBR) node - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Home Search Favorites

Address <http://uddi.microsoft.com/search/frames.aspx?frames=true&search=28494621-3ebe-470a-a9cb-828a1613d7c> Go Links

Microsoft
uddi
Business Registry Node

MS Products | MS Search | MSDN Home | Microsoft Home

Sign In

Home Search Quick Help UDDI Help Go

Search

Results Explorer

- Cape Clear Software
 - Andy Grove
 - USA Headquarters
 - European Headquarters
 - Airport Weather Check
 - <http://live.capescience.co>
 - Airport Weather
 - CapeConnect SOAPBuilder "
 - Corporate Web Site
 - <http://www.capeclear.com>
 - Developer Web Site
 - www.capescience.com
 - PolyCalc Web Service
 - <http://live.capescience.co>

Cape Clear Software
Airport Weather Check

An XML Web service uses standard Web protocols to share data with one or more applications, regardless of how each service is implemented in its native environment. View details, categorizations, and access points (bindings) for a service using the tabs below.

Details Bindings Categories

Details provide the service name and a brief description. The service key is unique and is intended for use in programmatic queries.

Service Key:
55c48719-52dc-49d2-9a74-7c01383771b7

Name

(en)
Airport Weather Check
1 record(s) found.

Descriptions

(en)
Web Service to provide weather information for international airports
1 record(s) found.

Internet

Откриване на Web услуги

◆ Откриването на Web услуги

- ❖ Позволява да се извлече описанието на Web услуга от даден сървър
- ❖ Използват се т. нар. "discovery" файлове (*.disco)
- ❖ DISCO (Discovery of Web Services)
 - ❖ XML базиран език
 - ❖ Описва Web услугата – връзка към WSDL файла, връзка към самата услуга и връзка към документация за услугата
- ❖ Обикновено disco файлът стои в главната директория на услугата
 - ❖ Напр. <http://www.mysite.com/math/math.disco>

DISCO файлове – пример

```
<?xml version="1.0" encoding="utf-8"?>
<discovery
  xmlns="http://schemas.xmlsoap.org/disco/">

  <contractRef
    ref="http://www.myserver.com/Demo-5-Service-
Types/TypesService.asmx?wsdl"
    docRef="http://www.myserver.com/Demo-5-Service-
Types/TypesService.asmx"
    xmlns="http://schemas.xmlsoap.org/disco/scl/" />

  <soap
    address="http://www.myserver.com/Demo-5-Service-
Types/TypesService.asmx"
    xmlns:q1="http://www.myserver.com/schemas/
Demo_5_Service_Types/"
    binding="q1:TypesServiceSoap"
    xmlns="http://schemas.xmlsoap.org/disco/soap/" />

</discovery>
```

WSDL описания на услуги

- ◆ WSDL (Web Services Description Language)
 - ❖ Описва интерфейса за достъп до дадена Web услуга
 - ❖ имената на поддържаните методи
 - ❖ входни и изходни параметри
 - ❖ използвани типове данни
 - ❖ XML базиран отворен стандарт на W3C
 - ❖ ASP.NET Web услугите връщат своя WSDL при извикване с параметър ?wsdl
 - ❖ <http://localhost/MyService.asmx?wsdl>

WSDL описание – пример

```
<?xml version="1.0" encoding="utf-8"?>
<definitions
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:s0="http://www.devgb.org/ws/MathService"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  targetNamespace="http://www.devgb.org/ws/MathService"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types> ... </types>

  <message name="AddSoapIn"> ... </message>

  <portType name="MathServiceSoap"> ... </portType>

  <binding name="MathServiceSoap" ... > ... </binding>

  <service name="MathService"> ... </service>

</definitions>
```

SOAP – формат на заявките

- ◆ SOAP (Simple Object Access Protocol)
 - ❖ Дефинира XML базиран формат за обмяна на съобщения
 - ❖ Отворен стандарт на W3C
- ◆ Едно SOAP съобщение се състои от:
 - ❖ SOAP хедър – описва параметри на съобщението (метаданни)
 - ❖ SOAP тяло – съдържа самото съобщение (данни – изпратената или върнатата заявка)
- ◆ Обикновено SOAP съобщенията се предават по протокол HTTP
 - ❖ Това позволява да преминават през firewall

SOAP заявка – пример

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CalcDistance xmlns="http://www.devbgo.org/Calc">
      <p1>
        <x>4</x>
        <y>5</y>
      </p1>
      <p2>
        <x>7</x>
        <y>-3</y>
      </p2>
    </CalcDistance>
  </soap:Body>
</soap:Envelope>
```


SOAP отговор – пример

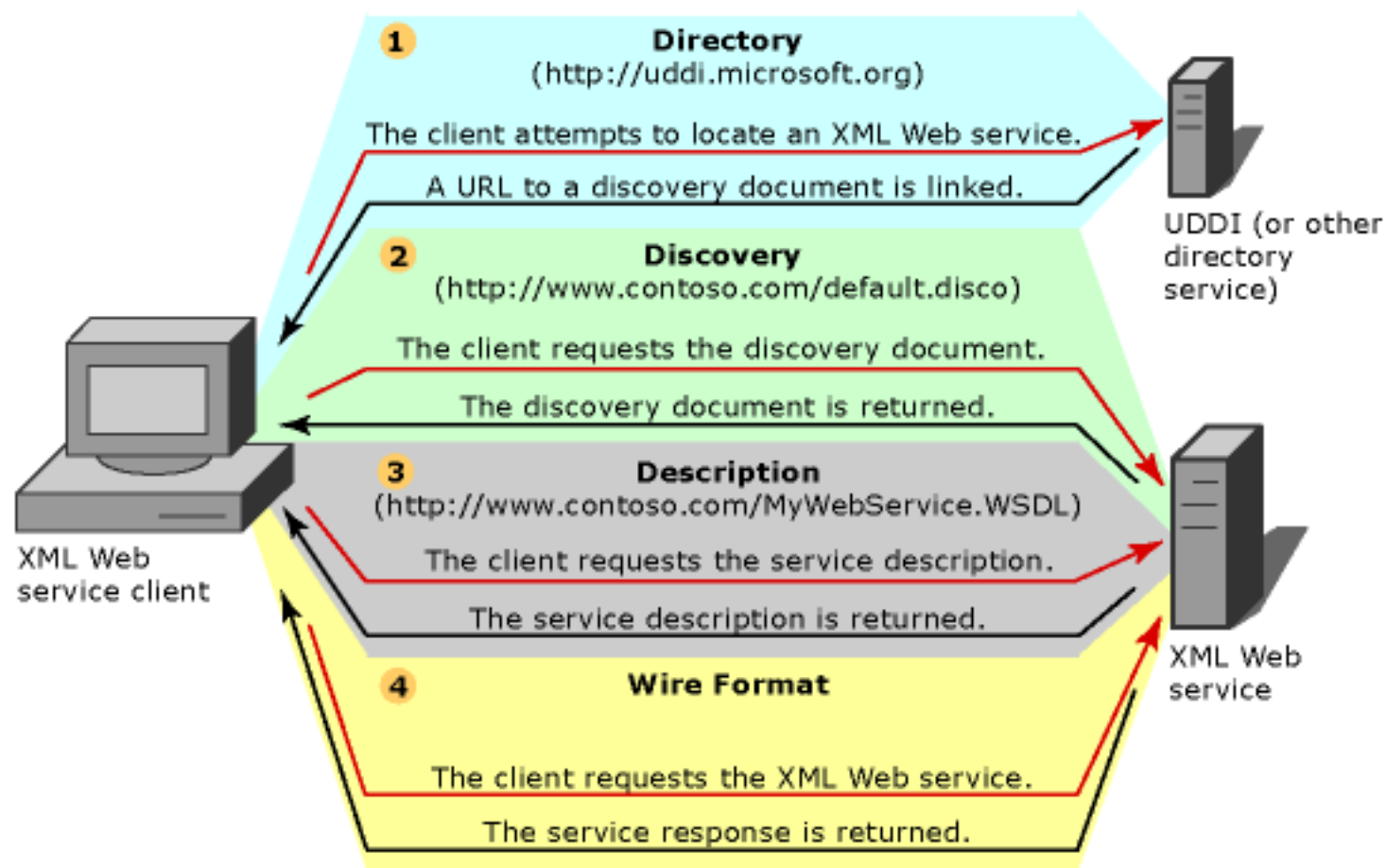
```
<?xml version="1.0" encoding="utf-8"?>

<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

  <soap:Body>
    <CalcDistanceResponse
      xmlns="http://www.devbg.org/Calc/">
      <CalcDistanceResult>
        8,54400374531753
      </CalcDistanceResult>
    </CalcDistanceResponse>
  </soap:Body>

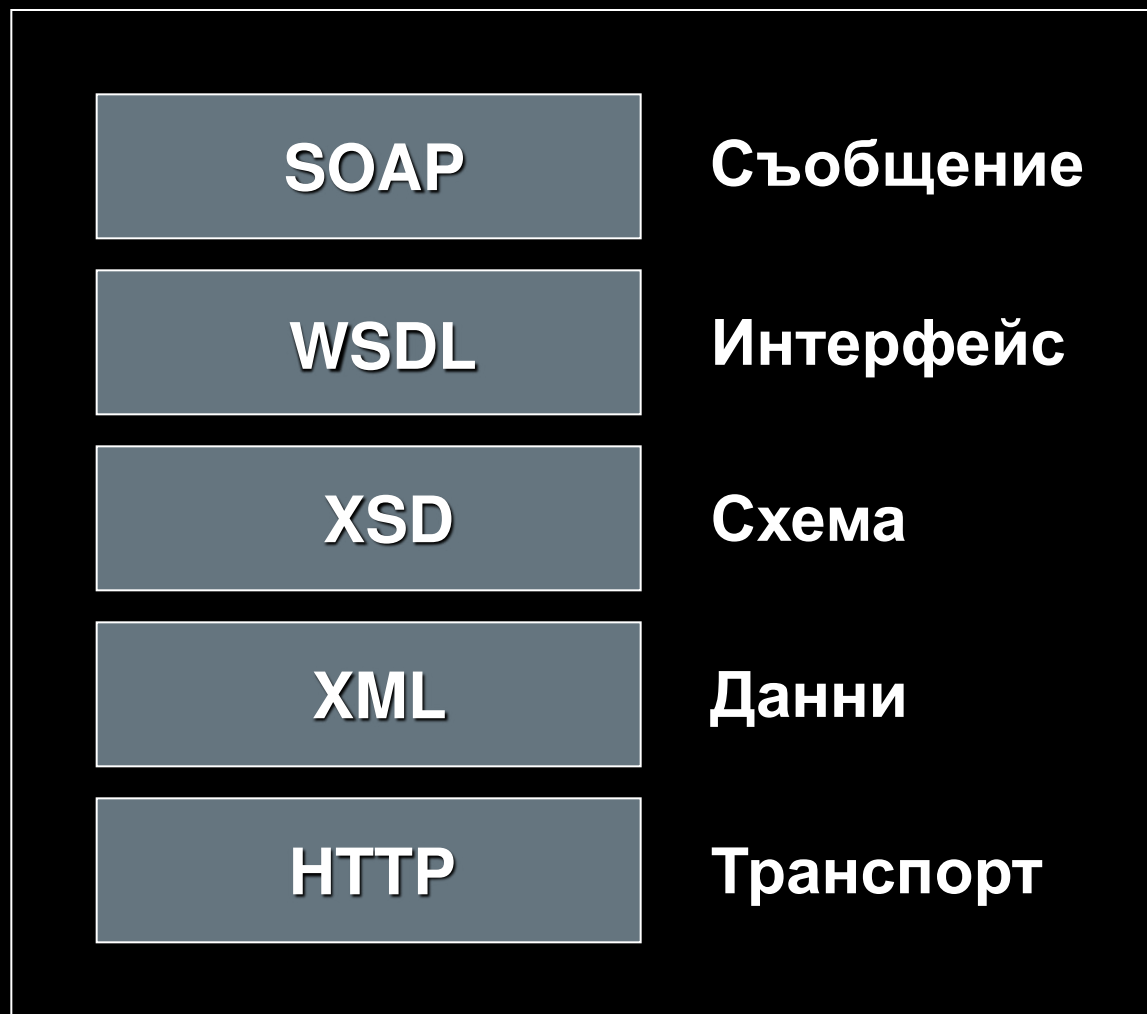
</soap:Envelope>
```

Инфраструктурата



Източник: MSDN

Протоколен стек на Web услугите



HTTP, XML, XSD

◆ HTTP

- ❖ Протокол за обмяна на информация между Web сървър и Web браузър
- ❖ Използва модела "заявка-отговор"
- ❖ Не поддържа стандартно сесия (stateless)
- ❖ Отворен стандарт на W3C

◆ XML

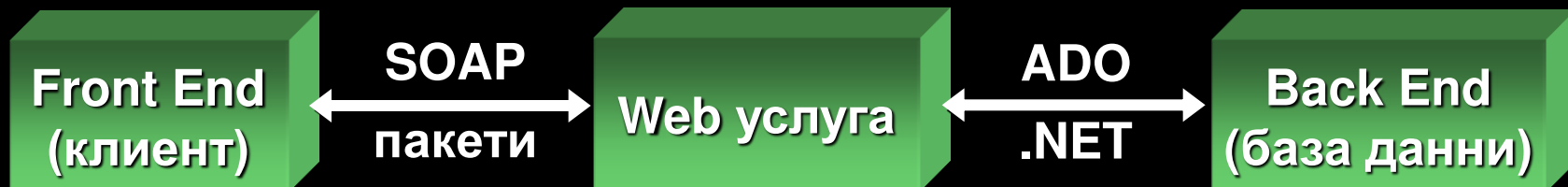
- ❖ Прост, текстово-базиран език за структурирано описание на информация
- ❖ Отворен стандарт на W3C

◆ XSD

- ❖ Мощен XML-базиран език за описание структурата (формата) на XML документи
- ❖ Отворен стандарт на W3C

Сценарии за използване

- ◆ Типични сценарии за употреба на Web услуги
 - ❖ Услуги към клиентски приложения
 - ❖ Могат да се консумират директно от клиентите, например от MS Office
 - ❖ Интеграция на приложения
 - ❖ Взаимодействие между приложения, работещи на различни платформи и ОС
 - ❖ Връзка между отделните компоненти при Enterprise приложения

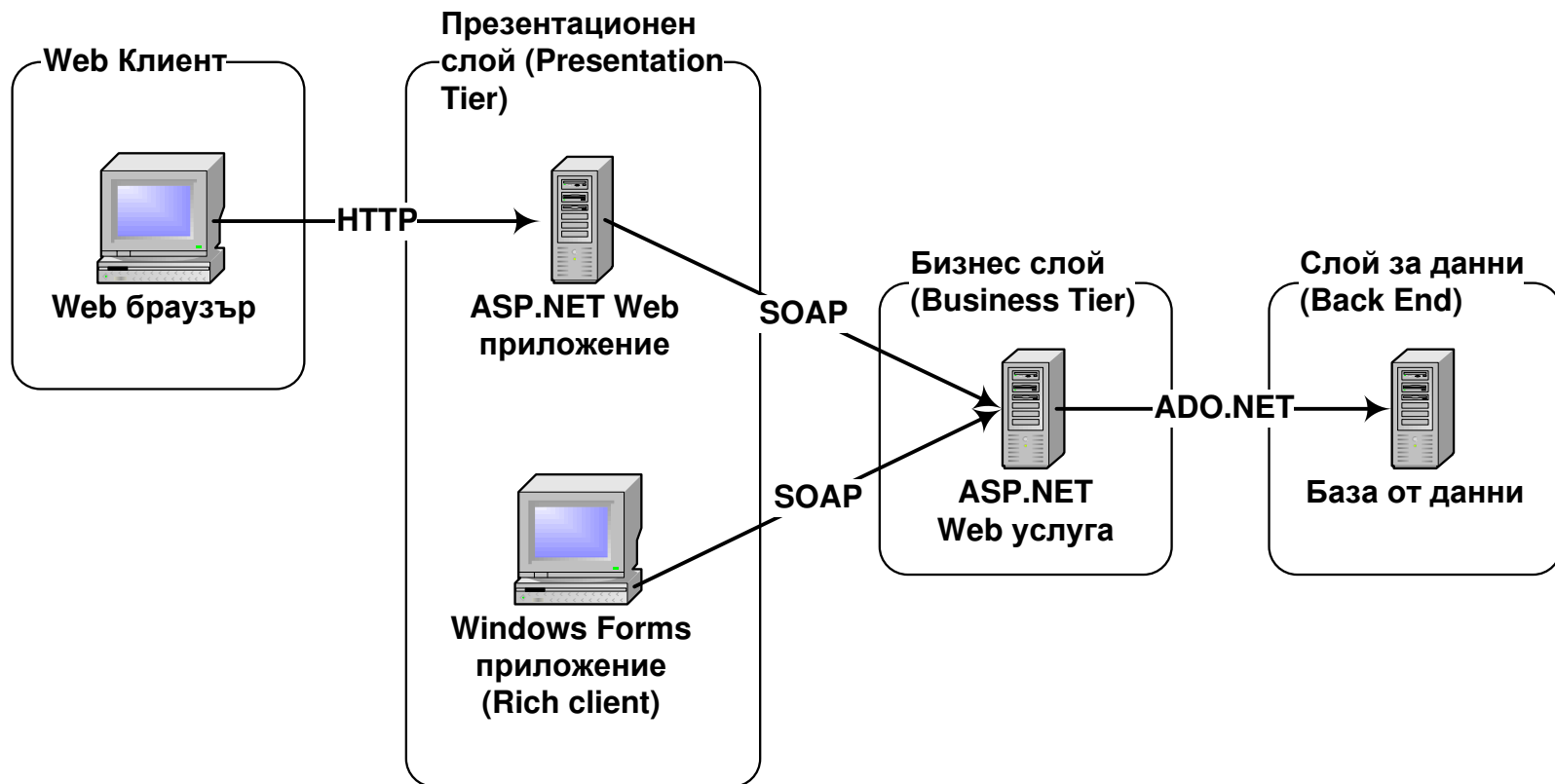


Enterprise приложения

- ◆ Какво са "Enterprise приложенията"?
 - ❖ Многослойни разпределени приложения, отговарящи на нуждите на големите корпоративни клиенти
 - ❖ Надеждни
 - ❖ Скалируеми
 - ❖ Лесно разширяеми
 - ❖ Устойчиви на срывове
- ◆ "Enterprise" е buzzword – понятие, за което много се говори, много се спекулира с него, но няма точна дефиниция какво означава

.NET Enterprise приложения

- ◆ Класическата Enterprise архитектура е трислойна:



.NET Enterprise приложения

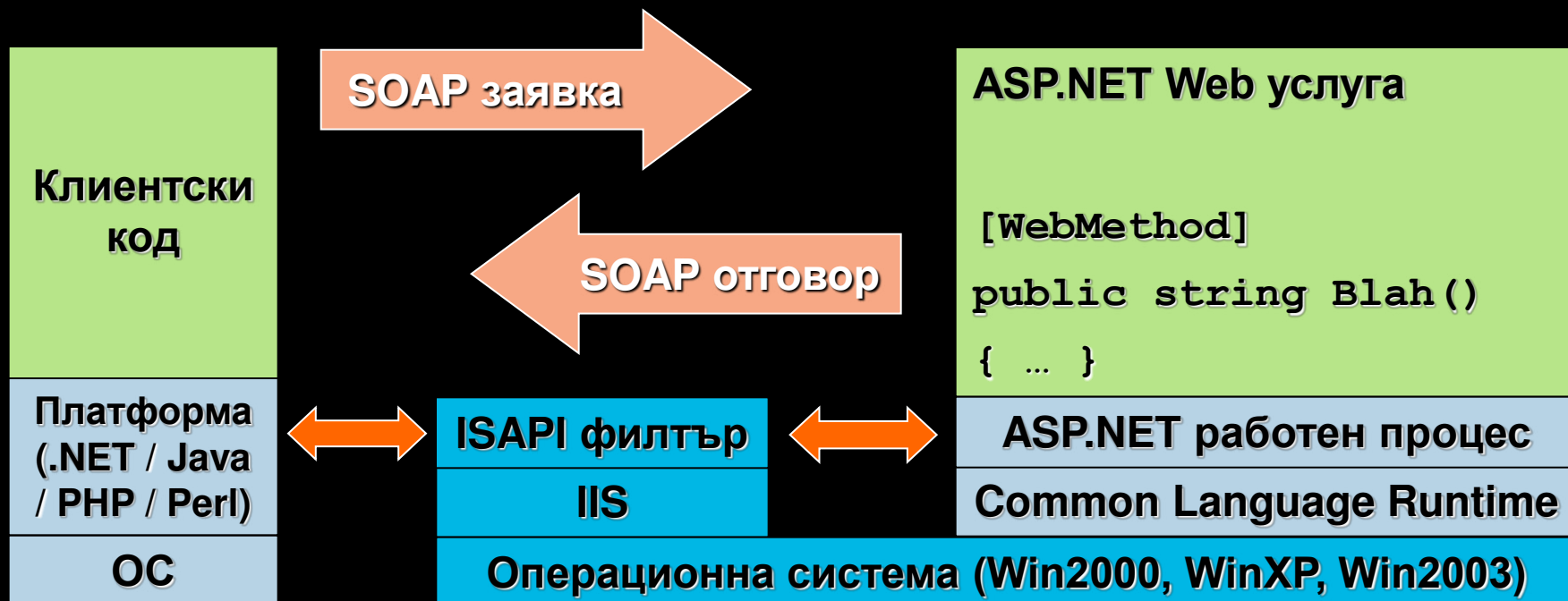
- ◆ **Слоеве в трислойната архитектура:**
 - ❖ **Презентационен слой (Front End)**
 - ❖ Управлява взаимодействието с потребителя (потребителски интерфейс)
 - ❖ Windows Forms и ASP.NET Web приложения
 - ❖ **Бизнес слой (Business Tier)**
 - ❖ Реализира работните процеси и операциите над данните (бизнес логиката)
 - ❖ ASP.NET Web услуги
 - ❖ **Слой за данните (Back End)**
 - ❖ Управлява данните на системата
 - ❖ База от данни (MS SQL Server, Oracle, DB2)

Web услугите в ASP.NET

- ◆ Реализират се в пространството `System.Web.Services`
- ◆ Имплементират предаването и приемането на SOAP съобщения
- ◆ Осигуряват преобразуване на типовете от XML в .NET типове и обратното
- ◆ Автоматично генериране на WSDL описания
- ◆ Автоматично генериране на C# прокси класове по WSDL описание

ASP.NET Web услуги – архитектура

- ◆ Web услугите се хостват в IIS (Internet Information Server)
- ◆ Изпълняват се от ASP.NET работния процес (aspnet_wp)



Създаване на Web услуги

- ◆ В .NET Framework за създаване на Web услуга се създава .asmx файл
 - ❖ Задава се специален таг, указващ, че файлът е Web услуга:

```
<%@ WebService Language="C#" Class="SomeService" %>
```

- ❖ Наследява се класът `System.Web.Services.WebService` (не е задължително)
- ❖ Имплементират се 1 или няколко Web метода:

```
[WebMethod] public void SomeMethod(...) { ... }
```

Web-услуга – пример

Demo-1-AddService.asmx

```
<%@ WebService Language="C#" Class="AddService" %>
```

```
using System;
```

```
using System.Web.Services;
```

```
public class AddService : WebService  
{
```

```
    [WebMethod]
```

```
    public int Add(int a, int b)
```

```
    {
```

```
        return a + b;
```

```
    }
```

```
}
```

Публикуване на Web услуги

- ◆ Web услугите се изпълняват от ASP.NET като Web-приложения
- ◆ Могат да се настройват както Web-приложенията с файловете
 - ❖ `Web.config` и `Global.asax`
- ◆ Обикновено се хостват от IIS (Internet Information Server)
- ◆ Публикуване на Web услуга
 - ❖ Копира се директорията на услугата в `C:\Inetpub\wwwroot`
 - ❖ Създава се Web Application в IIS
 - ❖ Компилира се услугата (ако е необходимо)

Демонстрация #1

◆ Нашата първа Web услуга – публикуване и тестване

The screenshot displays a Windows XP desktop with two main windows open.

Left Window: C:\Inetpub\wwwroot
This is a Windows Explorer window showing the file system. The 'Folders' pane on the left lists various directories including 'Inetpub', 'AdminScripts', 'ftproot', 'iissamples', 'mailroot', 'Scripts', 'sdk', and 'wwwroot'. Under 'wwwroot', there are subfolders like '_private', '_vti_crl', '_vti_log', '_vti_pvt', '_vti_script', '_vti_bin', and 'aspnet_client'. The 'Demo-1-AddService' folder is selected. The 'Address' bar shows 'C:\Inetpub\wwwroot'. The '1 objects selected' status bar is visible at the bottom.

Right Window: Demo-1-AddService Properties
This is a dialog box with tabs for 'Directory', 'Documents', 'Directory Security', 'HTTP Headers', and 'Custom Errors'. The 'Directory' tab is active. It contains settings for 'When connecting to this resource, the content should come from:' with radio buttons for 'The designated directory' (selected), 'A share located on another computer', and 'A redirection to a URL'. Below this is the 'Local Path' field set to '\\Demo-1-AddService'. There are checkboxes for 'Script source access', 'Read', 'Write', 'Directory browsing', 'Log visits', and 'Index this resource'. The 'Application Settings' section includes 'Application name' (Default Application), 'Starting point' (<Default Web Site>), 'Execute Permissions' (Scripts only), and 'Application Protection' (Medium (Pooled)). A black arrow points to the 'Create' button. At the bottom are 'OK', 'Cancel', 'Apply', and 'Help' buttons.

Right Window: AddService Web Service - Microsoft Internet Explorer
This is a Microsoft Internet Explorer window showing the 'AddService' web service. The address bar shows 'http://localhost/Demo-1-AddService/Demo-1-AddService.asmx?op=Add'. The page title is 'AddService'. Below the title, there is a link 'Click [here](#) for a complete list of operations.' The 'Add' section has a 'Test' subsection with the instruction: 'To test the operation using the HTTP POST protocol, click the 'Invoke' button.' Below this is a table with two rows: 'a:' with value '3' and 'b:' with value '5'. An 'Invoke' button is at the bottom right of the table. The 'SOAP' section contains a sample SOAP request and response, with placeholders like 'length' and 'length' highlighted in blue. The status bar at the bottom shows 'Done' and 'Local intranet'.

Използване на Web услуги

- ◆ В .NET Framework за консумиране на Web услуги се генерират т. нар. прокси класове
- ◆ Прокси класовете
 - ❖ Генерират се по WSDL дефиницията на Web услугата (<http://.../service?wsdl>)
 - ❖ Представяват сорс код на C# (VB.NET, ...)
 - ❖ Преобразуват заявките от .NET към SOAP заявки и SOAP отговорите към .NET
 - ❖ Преобразуват .NET типовете в XML и обратното (marshalling)



Генериране на прокси клас

- ◆ Прокси класовете могат да се генерират с инструмента `wsdl`

- ❖ Пример:

```
wsdl http://localhost/Demo-1-AddService/Demo-1-AddService.asmx?wsdl
```

- ❖ Резултатът е прокси класът `AddService.cs`
- ◆ За всеки метод `XXX` се генерират 3 метода с подходящите параметри:
 - ❖ `XXX(...)` – за синхронно извикване
 - ❖ `BeginXXX(...)` и `EndXXX(...)` – за асинхронно извикване

Използване на прокси клас

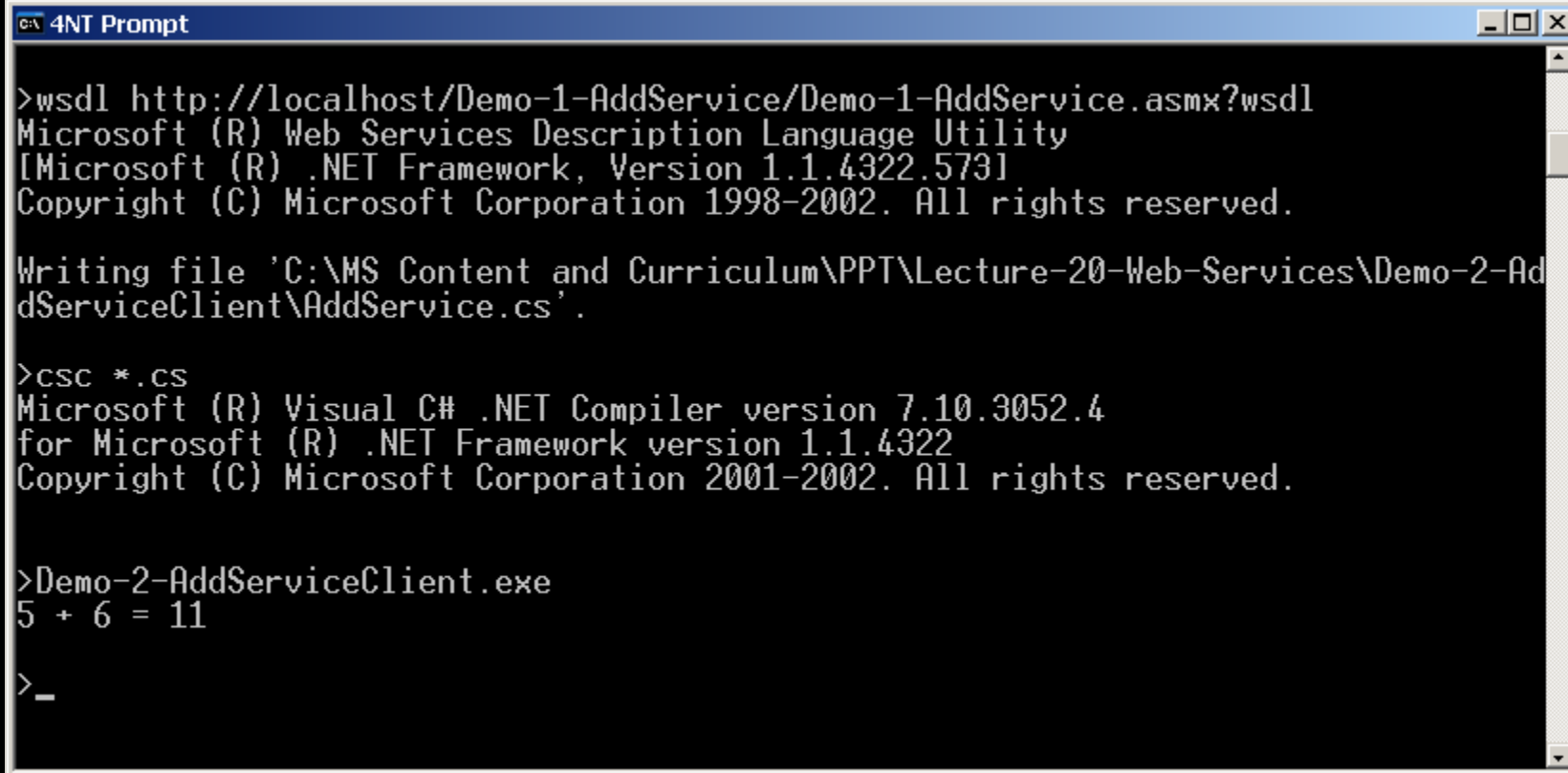
- ◆ Използването на прокси класовете е като обикновените класове
 - ❖ Инстанцира се прокси класът
 - ❖ Извикват му се методите

```
AddService addService = new AddService();  
int sum = addService.Add(5, 6);
```

- ❖ Методите могат да предизвикат `SoapException` или `WebException`
- ◆ При промяна на интерфейса на Web услугата трябва да се генерира отново прокси класът

Демонстрация #2

◆ Използване на Web услуга



```
C:\ 4NT Prompt

>wsdl http://localhost/Demo-1-AddService/Demo-1-AddService.asmx?wsdl
Microsoft (R) Web Services Description Language Utility
[Microsoft (R) .NET Framework, Version 1.1.4322.573]
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

Writing file 'C:\MS Content and Curriculum\PPT\Lecture-20-Web-Services\Demo-2-AddServiceClient\AddService.cs'.

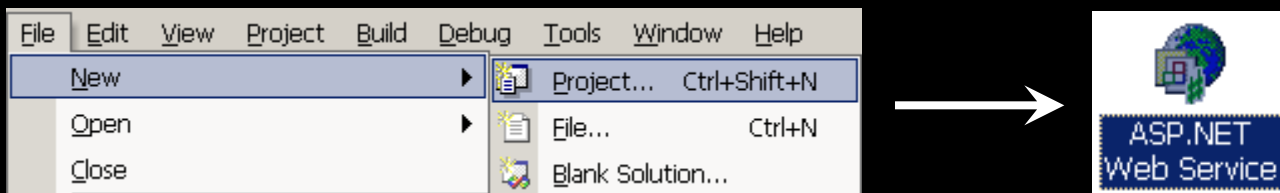
>csc *.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

>Demo-2-AddServiceClient.exe
5 + 6 = 11

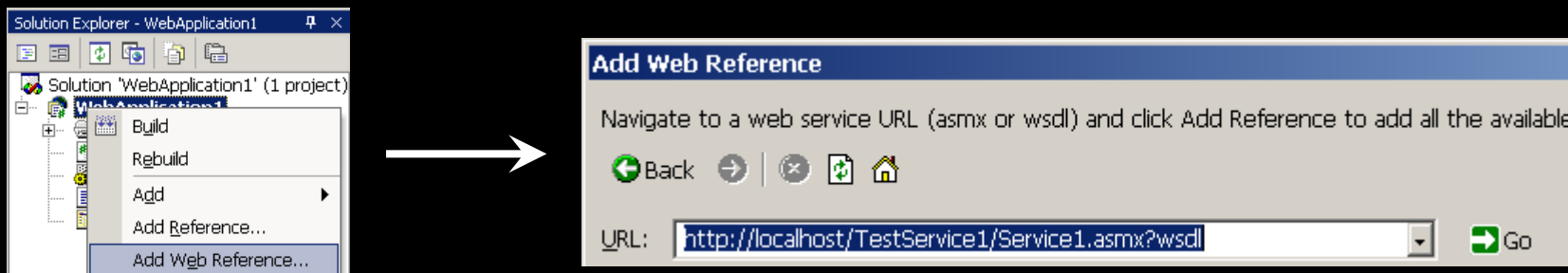
>_
```

Web услугите и VS.NET

- ◆ Visual Studio .NET има силна поддръжка за Web услуги
 - ❖ Създаване на Web услуги



- ❖ Консумиране на Web услуги
 - ❖ Автоматично генериране на прокси клас по WSDL описанието



Създаване на услуга с VS.NET

- ◆ При създаване на нова Web-услуга VS.NET 2003 прави следното:

1. Създава .sln файл:

```
C:\Documents and Settings\<your user name>\  
My Documents\Visual Studio Projects\  
<WebServiceName>\<WebServiceName>.sln
```

2. Създава виртуална директория в IIS:

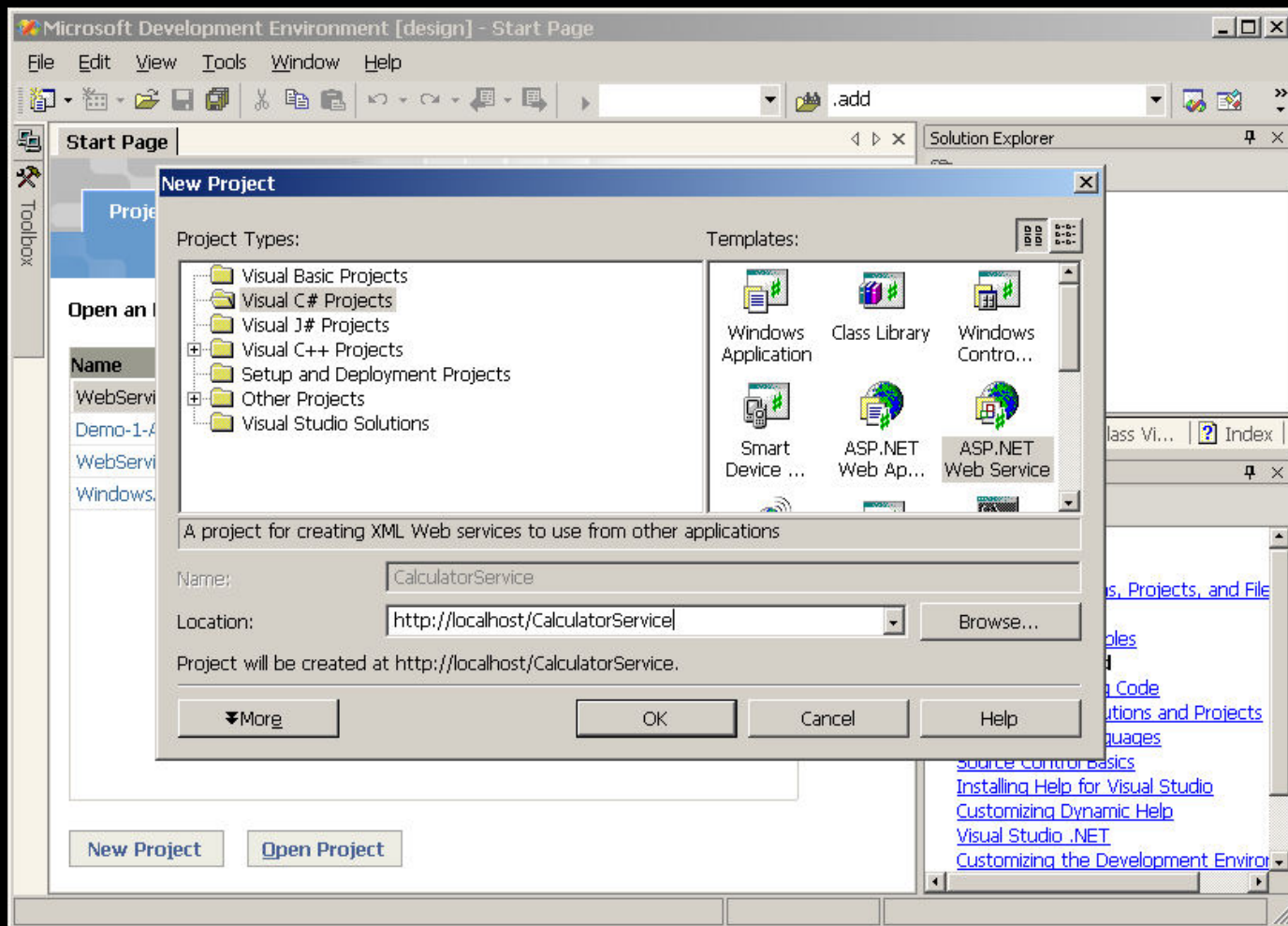
```
C:\Inetput\wwwroot\<WebAppName>
```

3. Създава в нея файловете на проекта:

```
<WebServiceName>.csproj, <WebServiceName>.asmx,  
<WebServiceName>.asmx.cs, <WebServiceName>.resx,  
Web.config, Global.asax, Global.asax.cs
```

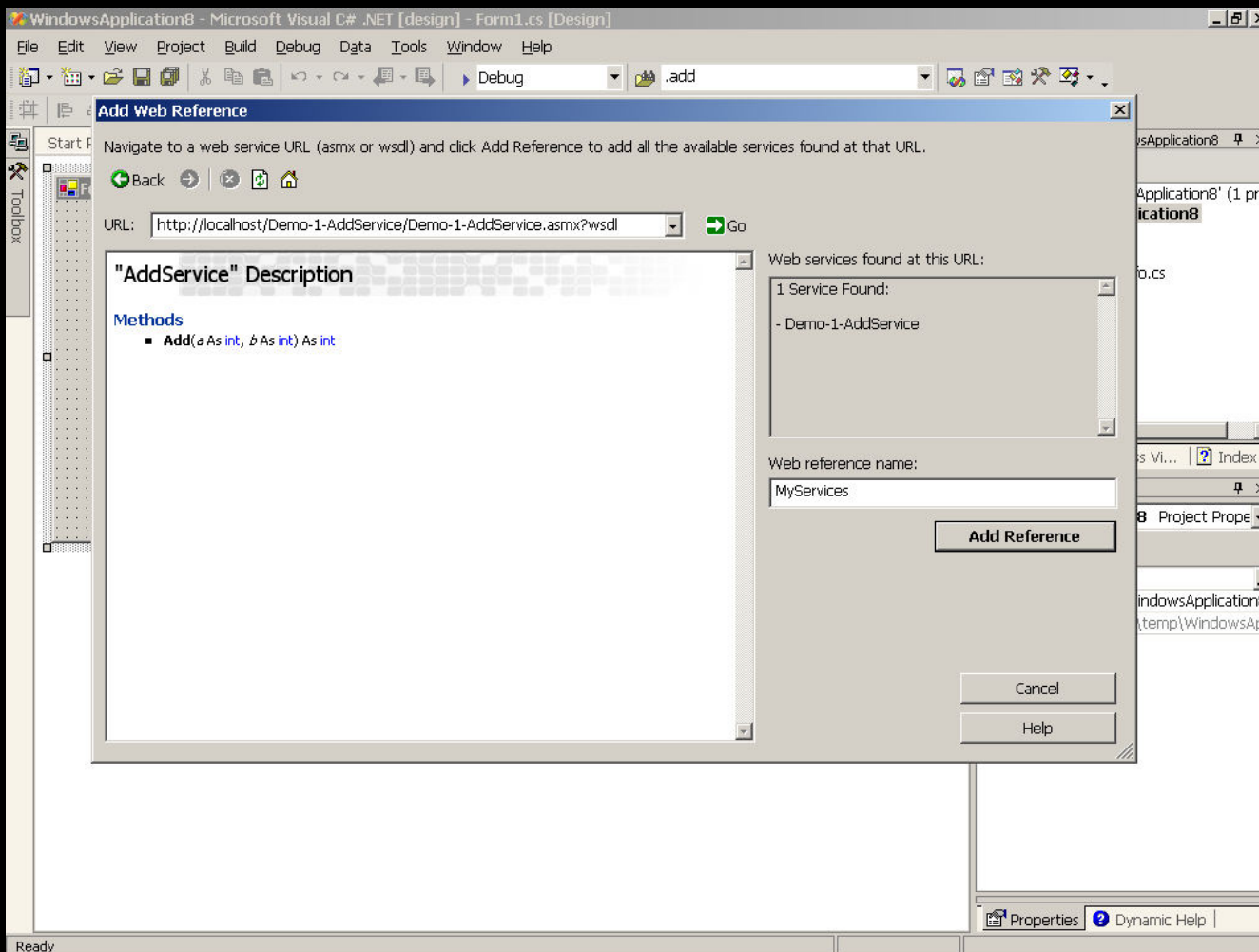
Демонстрация #3

◆ Създаване на Web услуга с VS.NET



Демонстрация #4

◆ Консумиране на Web услуга с VS.NET



Атрибути за Web услугите

◆ Атрибутът [WebService]

- ❖ Задава описание на Web услугата
- ❖ Namespace – задава XML пространство, което идентифицира уникално услугата
- ❖ Name – име на услугата
- ❖ Description – текстово описание на услугата

◆ Атрибутът [WebMethod]

- ❖ Указва, че даден метод е достъпен за клиентите на Web услугата
- ❖ EnableSession=true – позволява използване на сесия (Session обекта в ASP.NET)
- ❖ Description – текстово описание на метода

Атрибути на услугите – пример

```
[WebService(  
    Description="Demo Web service - " +  
        " demonstrates complex type marshalling.",  
    Namespace="http://www.devbg.org/services/" +  
        "Demo_5_Service_Types/2004/12/17",  
    Name="Demo 5 - Service Types")]  
public class TypesService :  
    System.Web.Services.WebService  
{  
    ...  
  
    [WebMethod(  
        Description="Returns the list of " +  
            "currently active issues.",  
        EnableSession=true)]  
    public Issue[] GetIssues() { ... }  
    ...  
}
```


Прехвърляне на типове

- ◆ Прехвърлянето на типовете (type marshalling) в .NET Framework се извършва автоматично
 - ❖ Използва се вградената XML сериализация
- ◆ Като параметри и връщана стойност на методите могат да бъдат използвани:

Тип	Описание
Примитивни типове	Например: <code>string</code> , <code>char</code> , <code>byte</code> , <code>bool</code> , <code>sbyte</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>short</code> , <code>ushort</code> , <code>float</code> , <code>double</code> , <code>decimal</code> , ...
Изброени типове	Например: <code>enum Color {Red, Blue}</code> Предават се като низове. Ползват се имената на изброените стойности.

Прехвърляне на типове

- ◆ Като параметри и връщана стойност на методите могат да бъдат използвани:

Тип	Описание
Класове и структури	Предават се само публичните полета и свойства на класовете и структурите. Поддържат се вложени типове и дървовидни структури, но не и цикли. Трябва да е дефиниран конструктор без параметри! Ползва се XML сериализация. Примери: <pre>public struct Point { public int x, y; } public class Student { public int Age { get {...} set {...} }</pre>

Прехвърляне на типове

- ◆ Като параметри и връщана стойност на методите могат да бъдат използвани:

Тип	Описание
Масиви	Масиви от примитивни типове, изброени типове, класове или структури Например: <code>string[]</code> , <code>Color[]</code> , <code>Point[]</code>
Колекции	Колекции от примитивни типове, изброени типове, класове или структури. Пример: <code>ICollection</code> , <code>ArrayList</code> , ... Предават се като масиви.
DataSet обекти	ADO.NET DataSet обекти. Предават се заедно с XSD схемата, която описва структурата им.
XmlNode	XML фрагменти. Предават се в чист вид.

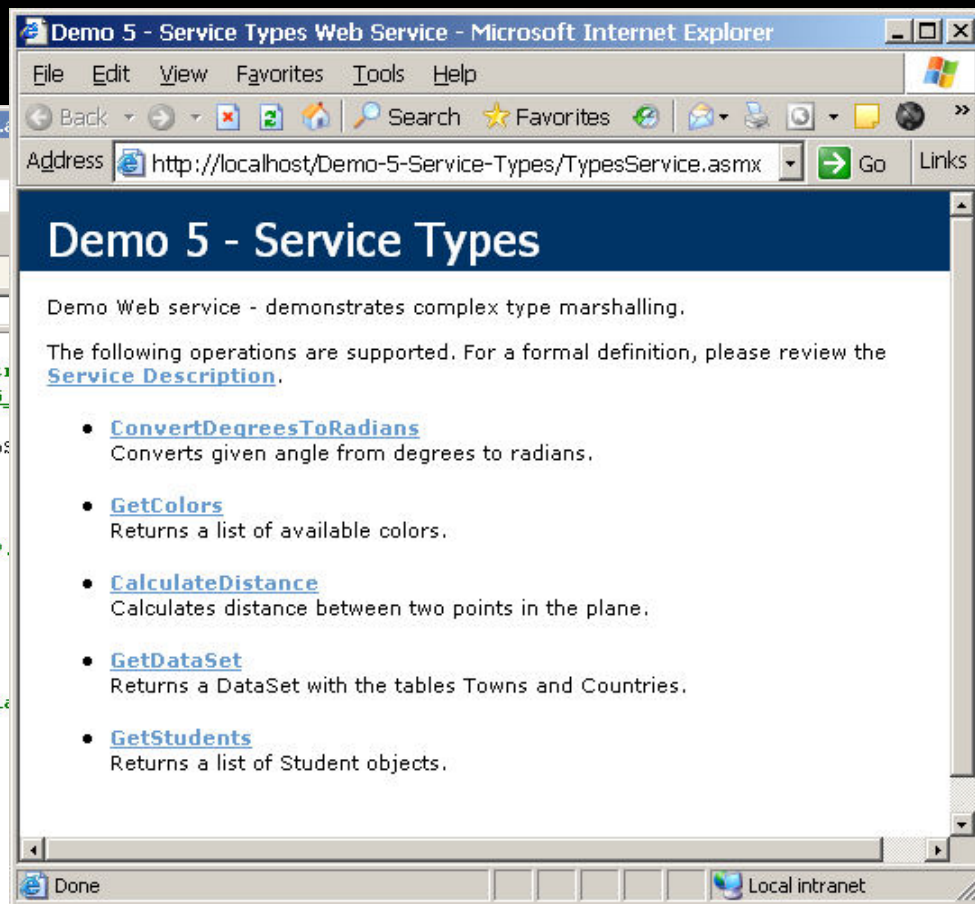
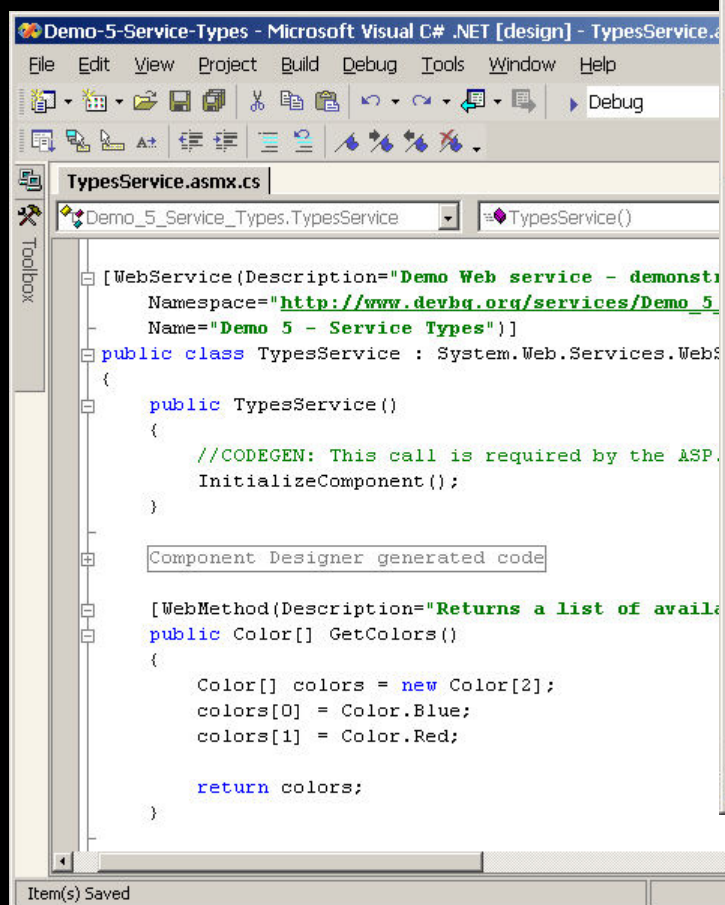
Deployment на Web услуга

- ◆ Искаме да публикуваме Web-услуга MyService, която имаме на сорс-код
 1. Копираме всички файлове на услугата в C:\Inetpub\wwwroot\MyService
 2. От IIS административната конзола създаваме Web приложение за виртуалната директория MyService
 3. Компилираме услугата с VS.NET
- ◆ Ако няма да дебъгваме (например при Release), изтриваме сорс-кода:

```
del *.cs *.csproj *.resx *.webinfo ...
```

Демонстрация #5

◆ Прехвърляне на сложни типове – Web услугата



Демонстрация #6

- ◆ Прехвърляне на сложни типове – клиент за Web услугата

Demo-6-TypesService-Client - Microsoft Visual C# .NET [design] - TypesServiceClient.cs

File Edit View Project Build Debug Tools Window Help

TypesServiceClient.cs

```
// Invoke CalculateDistance() web method
MyServices.Point p1 = new MyServices.Point();
p1.x = 5;
p1.y = 5;
MyServices.Point p2 = new MyServices.Point();
p2.x = 7;
p2.y = -3;
double distance = service.CalculateDistance(p1,p2);
Console.WriteLine("\nDistance = {0}", distance);

// Invoke ConvertDegreesToRadians(double angle)
double angle = 90;
service.ConvertDegreesToRadians(ref angle);
Console.WriteLine("\nAngle in radians = {0}", angle);

// Invoke GetStudents()
MyServices.Student[] students = service.GetStudents();
Console.WriteLine("\nStudents:");
foreach (MyServices.Student student in students)
{
    Console.WriteLine("{0} : {1}", student.Name, student.Score);
}
```

Colors:
Blue
Red

Distance = 8,24621125123532

Angle in radians = 1,5707963267949

Students:
Иван : 20
Мария : 19
Жоро : 21

Towns:
1 : София
2 : Варна

Countries:
1 : България

Press any key to continue.

Build succeeded

Ln 32 Col 16 Ch 7 INS

Демонстрация #7

◆ Дебъгване на Web услуги

The image displays two Visual Studio windows side-by-side, illustrating the debugging process of a web service.

Top Window: Demo-5-Service-Types - Microsoft Visual C# .NET [run] - TypesService.asmx.cs

- File:** TypesService.asmx.cs
- Method:** CalculateDistance(Point p1, Point p2)
- Code:**

```
[WebMethod]
public double CalculateDistance(Point p1, Point p2)
{
    int dx = p1.x - p2.x;
    int dy = p1.y - p2.y;
    double distance = Math.Sqrt(dx*dx + dy*dy);
    return distance;
}

[WebMethod]
public void ConvertDegreesToRadians(ref double aAngle)
{
    aAngle = (double) aAngle * Math.PI / 180;
}
```
- Autos:** Shows 'students' with value '<undefined value>' and 'this' with value '{Demo_5_Service_Types.TypesService}'.
- Output:** Shows the path to the assembly: '/LM/W3SVC/1/Root/Demo_5/TypesService.asmx'. The status bar indicates 'Build succeeded'.

Bottom Window: Demo-6-TypesService-Client - Microsoft Visual C# .NET [break] - TypesServiceClient.cs

- File:** TypesServiceClient.cs
- Method:** Main(string[] args)
- Code:**

```
foreach (MyServices.Color color in colors)
{
    Console.WriteLine(color);
}

// Invoke CalculateDistance() web method
MyServices.Point p1 = new MyServices.Point();
p1.x = 5;
p1.y = 5;
MyServices.Point p2 = new MyServices.Point();
p2.x = 7;
p2.y = -3;
double distance = service.CalculateDistance(p1, p2);
Console.WriteLine("\nDistance = {0}", distance);
```
- Autos:** Shows 'distance' (0.0, double), 'p1' and 'p2' (MyServices.Point), 'p2.y' (-3, int), and 'service' (MyService Demo_6).
- Output:** Shows the path to the assembly: '/LM/W3SVC/1/Root/Demo_6/TypesServiceClient.exe'. The status bar indicates 'Build succeeded'.

Моделът на изпълнение

- ◆ За всяка заявка към Web услуга ASP.NET прави следното:
 1. Инстанцира класа на Web-услугата
 2. Заделя отделна нишка от пула
 3. Изпълнява заявката в тази нишка
 4. Връща резултата към клиента
 5. Връща нишката в обратно пула
 6. Остава инстанцията на класа на услугата да бъде унищожена от GC
- ◆ Заявките се подреждат в опашка и се изпълняват по няколко едновременно

Асинхронно извикване

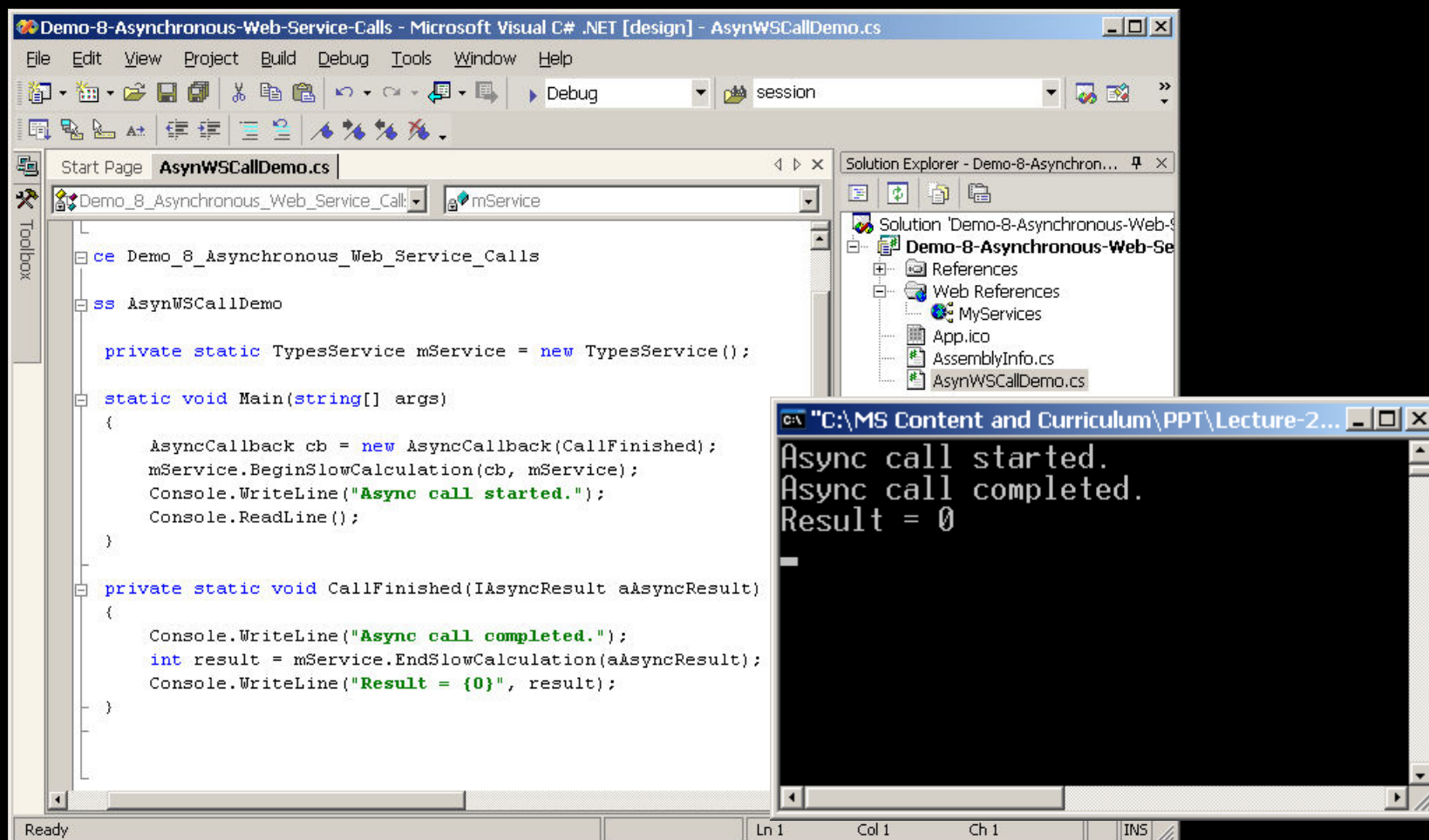
- ◆ Асинхронното извикване се използва при времеотнемачи операции с Web услуги
- ◆ За асинхронни извиквания в прокси класа има методи `BeginXXX(...)` и `EndXXX(...)`
- ◆ При завършване услугата извиква зададен метод за обратно извикване (callback)

```
public void static Main()
{
    AsyncCallback cb = new AsyncCallback(CallFinished);
    service.BeginLongProcess(params, cb, service);
}

private void CallFinished(IAsyncResult asyncResult)
{
    Console.WriteLine("Async call completed.");
}
```

Демонстрация #8

◆ Асинхронно извикване на Web услуга



The screenshot shows the Microsoft Visual Studio IDE with the following components:

- File Explorer:** Shows the project structure for "Demo-8-Asynchronous-Web-Service-Calls".
- Solution Explorer:** Displays the project files, including "References", "Web References", "MyServices", "App.ico", "AssemblyInfo.cs", and "AsynWSCallDemo.cs".
- Code Editor:** Shows the source code for "AsynWSCallDemo.cs". The code defines a static class "AsynWSCallDemo" with a "Main" method and a "CallFinished" callback method. The "Main" method creates an "AsyncCallback" object, calls "BeginSlowCalculation" on "mService", and prints "Async call started.". The "CallFinished" method calls "EndSlowCalculation" and prints "Async call completed." and "Result = 0".
- Console Window:** Displays the output of the application, showing the sequence of events: "Async call started.", "Async call completed.", and "Result = 0".

```
private static TypesService mService = new TypesService();

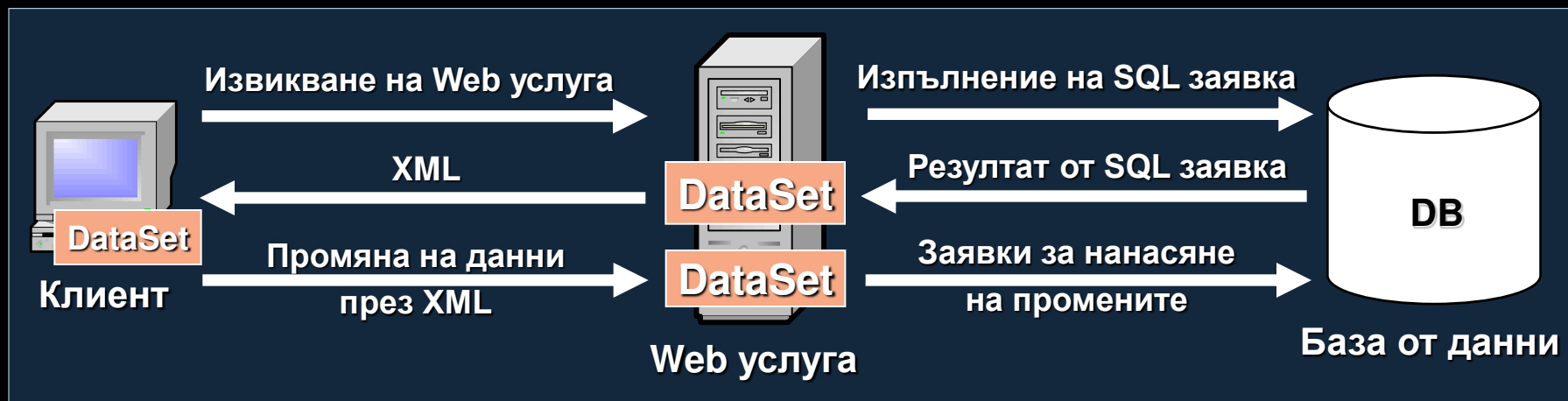
static void Main(string[] args)
{
    AsyncCallback cb = new AsyncCallback(CallFinished);
    mService.BeginSlowCalculation(cb, mService);
    Console.WriteLine("Async call started.");
    Console.ReadLine();
}

private static void CallFinished(IAsyncResult aAsyncResult)
{
    Console.WriteLine("Async call completed.");
    int result = mService.EndSlowCalculation(aAsyncResult);
    Console.WriteLine("Result = {0}", result);
}
```

Ready Ln 1 Col 1 Ch 1 INS

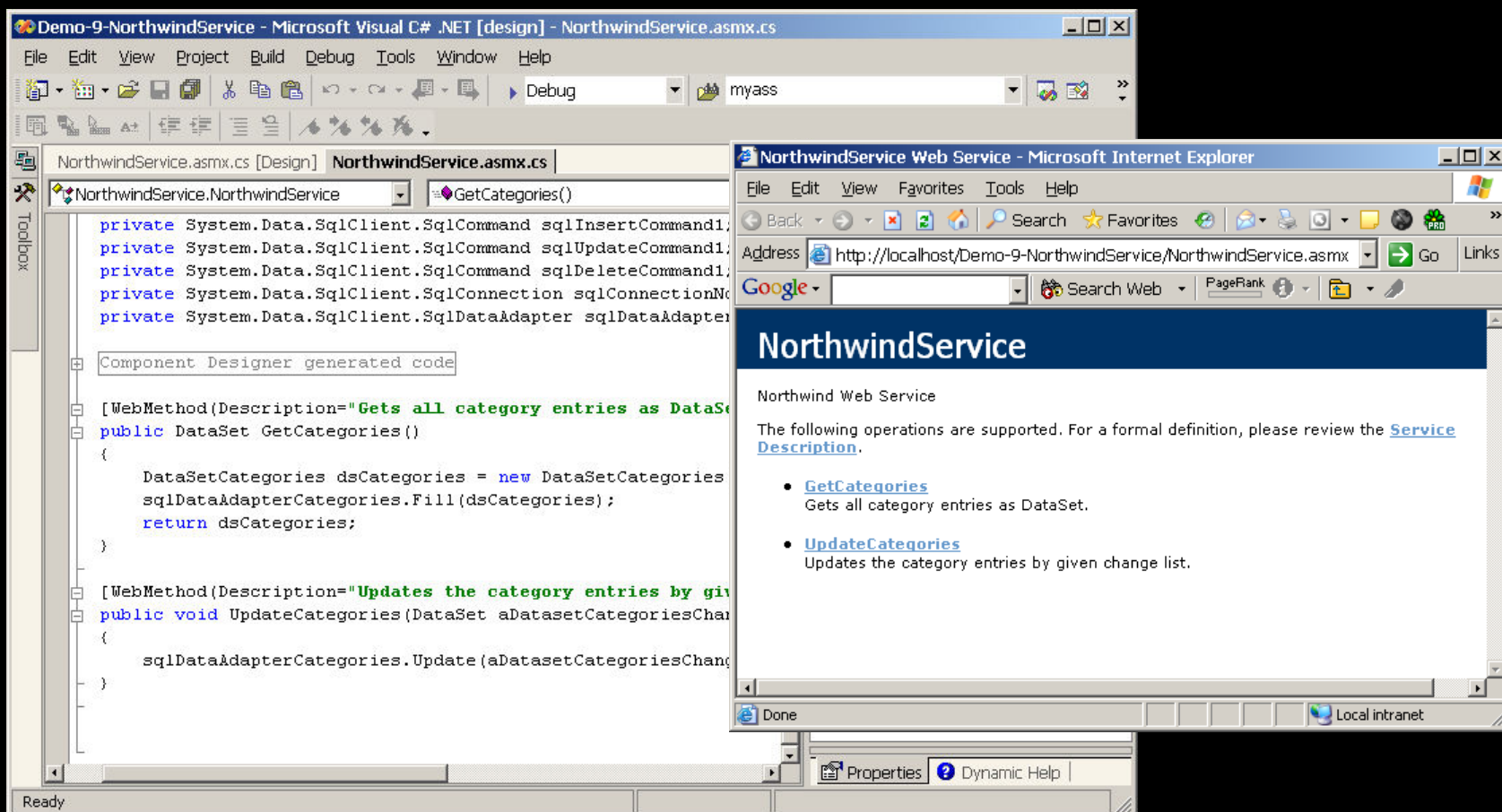
Web услуги и работа с данни

- ◆ Web услугите често се използват за достъп до данни
 - ❖ Обикновено се прехвърлят DataSet обекти или масиви с данни
 - ❖ Web услугата реализира:
 - ❖ Извличането на данни
 - ❖ Промяната на данни



Демонстрация #9

- ◆ Доступ до данни през Web услуга – услугата



Демонстрация #10

- ◆ Доступ до данни през Web услуга – клиентското приложение

The screenshot shows the Visual Studio .NET IDE with the following components:

- Visual Studio .NET IDE:** The title bar reads "Demo-10-NorthwindClient - Microsoft Visual C# .NET [design] - MainForm.cs". The menu bar includes File, Edit, View, Project, Build, Debug, Tools, Window, and Help. The toolbar shows various icons for file operations and development. The status bar at the bottom indicates "Ready".
- MainForm.cs [Design]:** The design view shows a single control, `SaveData()`.
- MainForm.cs [Code]:** The code view shows the following C# code:

```
private void SaveData()
{
    try
    {
        DataSet dsCategoriesChanges = mDsCategories.GetChanges()
        if (dsCategoriesChanges != null)
        {
            // Commit the changes to the Web service
            mNorthwindService.UpdateCategories(dsCategoriesChanges)

            // Refresh the categories table
            LoadData();

            MessageBox.Show("Categories updated succesfully.", "Success")
        }
    }
    catch (SoapException)
    {
        MessageBox.Show(
            "Can not update the categories.", "Error");
    }
}
```
- Northwind Categories Dialog:** A modal dialog box titled "Northwind Categories" is open. It contains a table with the following data:

	CategoryID	CategoryName	Description	Picture
	1	Beverages	Soft drinks, c	Byte[] Array
	2	Condiments	Sweet and sa	Byte[] Array
	3	Confections	Desserts, can	Byte[] Array
	4	Dairy Product	Cheeses	Byte[] Array
	5	Grains/Cereal	Breads, crack	Byte[] Array
	6	Meat/Poultry	Prepared me	Byte[] Array
	7	Produce	Dried fruit an	Byte[] Array
	8	Seafood	Seaweed and	Byte[] Array
	9	бира	Заропка	(null)
*				

The dialog also features "Load" and "Save" buttons at the bottom.

Поддръжка на сесии

- ◆ ASP.NET Web услугите поддържат сесии, както всички Web приложения
 - ❖ `Session` – съдържа текущата сесия
 - ❖ `Application` – съдържа контекста на Web приложението
- ◆ Разрешаване на сесията при услугата:

```
[WebMethod(EnableSession=true)]
```

- ◆ Разрешаване на сесията при клиента:

```
MyService myService = new MyService();  
myService.CookieContainer =  
    new System.Net.CookieContainer();
```


Демонстрация #11

◆ Използване на сесии – Web услуга

The screenshot displays two windows from a Windows XP environment. The top window is Microsoft Visual Studio, showing the code for `SessionService.asmx.cs`. The code defines a `GetSessionCounter()` method that increments a session counter and returns its value. The bottom window is Microsoft Internet Explorer, displaying the `SessionService` web service page. The page lists the supported operations: `GetSessionCounter` and `GetApplicationCounter`.

Visual Studio Code:

```
counter = (int) Application["counter"];
}
counter++;
Application["counter"] = counter;
return counter;
}
finally
{
    Application.Unlock();
}
}

[WebMethod(EnableSession=true, Description="Returns t
public int GetSessionCounter()
{
    int counter = 0;
    if (Session["counter"] != null)
    {
        counter = (int) Session["counter"];
    }
    counter++;
    Session["counter"] = counter;
    return counter;
}
```

Internet Explorer Page:

SessionService

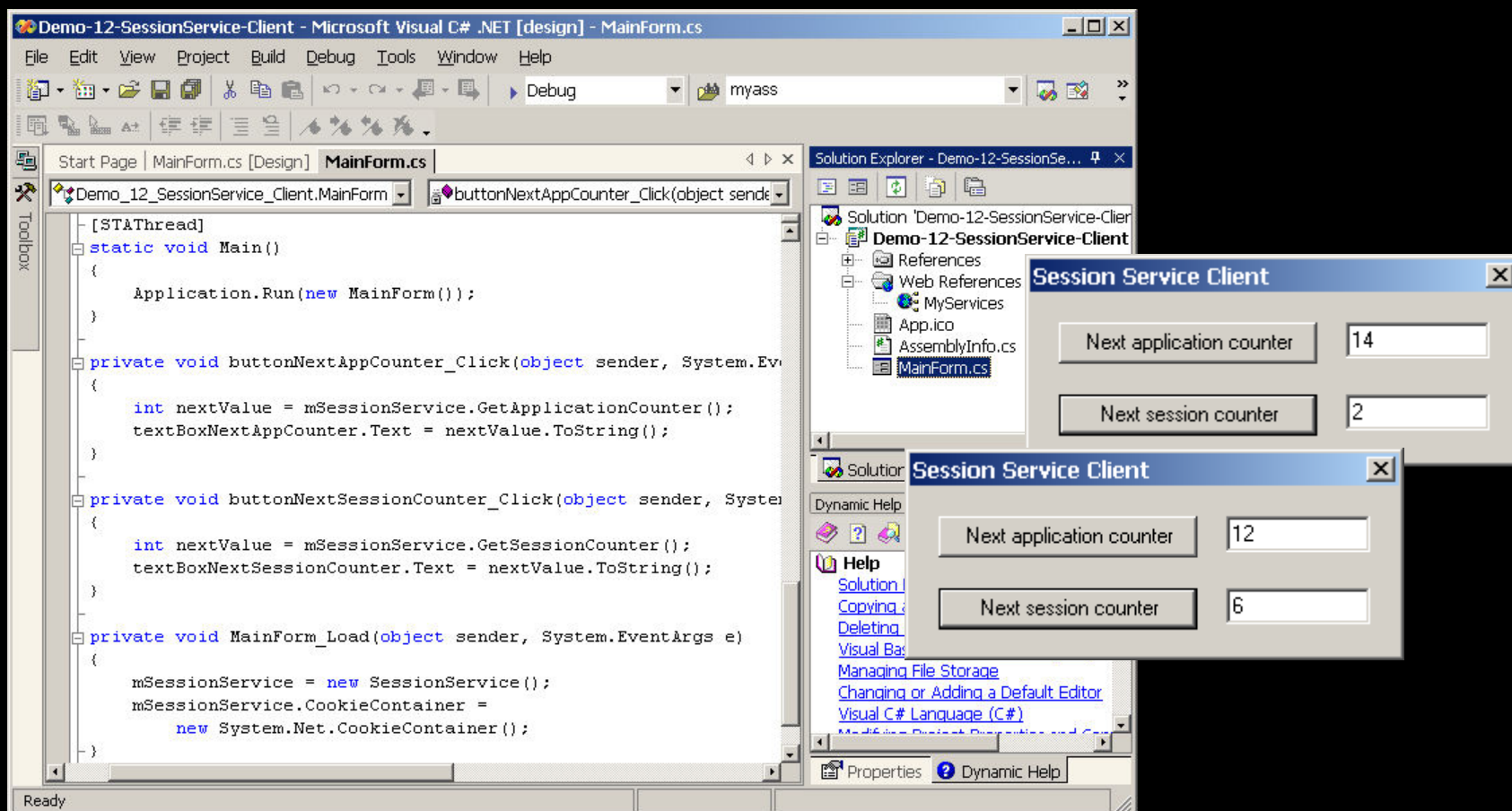
Session Web Service

The following operations are supported. For a formal definition, please review the [Service Description](#).

- [GetSessionCounter](#)
Returns the next value of the local session counter.
- [GetApplicationCounter](#)
Returns the next value of the global application counter.

Демонстрация #12

- ◆ Използване на сесии – клиент за Web услугата



Сигурност на Web услугите

- ◆ Често пъти се изисква някои методи да не могат да се викат без автентикация
- ◆ Защитата от неоторизиран достъп може да се направи по няколко начина:
 - ❖ Всеки метод да има допълнителни параметри (потребител, парола и др.)
 - ❖ Чрез използване на сесия
 - ❖ Чрез средствата на Web Services Enhancements (WSE). WSE поддържат:
 - ❖ Криптиране на трафика
 - ❖ Цифрово подписване
 - ❖ Автентикация чрез цифрови сертификати

Сигурност чрез сесии

1. Добавяме метод за автентикация на потребител:

```
[WebMethod(EnableSession = true, Description =  
    "Checks given login and creates a session.")]  
public void Login(string aUser, string aPassword)  
{  
    if ( user/pass is valid )  
    {  
        Session["user"] = aUser;  
    }  
    else  
    {  
        throw new Exception("Invalid login!");  
    }  
}
```

Сигурност чрез сесии

2. Във всеки защитен метод проверяваме дали има автентикиран потребител:

```
[WebMethod(EnableSession = true, Description =  
    "This method requires active session.")]  
public string GetProtectedData()  
{  
    string user = (string) Session["user"];  
    if (user != null)  
    {  
        return "This is protected data!";  
    }  
    else  
    {  
        throw new Exception("Access denied!");  
    }  
}
```

Исключенията в Web услугите

- ◆ Когато Web услуга хвърли изключение, клиентът получава `SoapException`
 - ❖ Оригиналният тип на изключението се губи
 - ❖ Губят се вложените (inner) изключения
 - ❖ Запазва се текстът (свойството `Message`)
- ◆ За да връщаме информация за проблем, възникнал в Web метод:
 - ❖ Дефинираме изброен тип (`enum`), съдържащ възможните грешки
 - ❖ Връщаме от Web метода код на грешка – инстанция на изброения тип
- ◆ Клиентът проверява кода за грешка

Исключенията в Web услугите

```
[WebMethod]
public UserError AddUser(string aUser, string aPass)
{
    if ( aUser is invalid )
        return UserError.InvalidUser;
    if ( aPass is invalid )
        return UserError.InvalidPassword;
    ...
    return UserError.Success;
}

public enum UserError
{
    Success,
    UserAlreadyExists,
    InvalidUser,
    InvalidPassword,
    DatabaseError
}
```

Пример

Service.vb

```
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols
```

```
<WebService(Namespace:="http://tempuri.org/")> _
<WebServiceBinding(ConformsTo:=WsiProfiles.BasicProfile1_1)> _
<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> _
```

```
Public Class Service
    Inherits System.Web.Services.WebService
```

```
<WebMethod()> _
Public Function HelloWorld() As String
    Return "Hello World"
End Function
```

```
<WebMethod()> Public Function FahrenheitToCelsius(ByVal Fahrenheit As String) As String
    Dim fahr
    fahr = Trim(Replace(Fahrenheit, ",", "."))
    If fahr = "" Or IsNumeric(fahr) = False Then Return "Error"
    Return (((fahr) - 32) / 9) * 5
End Function
```

```
<WebMethod()> Public Function CelsiusToFahrenheit(ByVal Celsius As String) As String
    Dim cel
    cel = Trim(Replace(Celsius, ",", "."))
    If cel = "" Or IsNumeric(cel) = False Then Return "Error"
    Return (((cel) * 9) / 5) + 32
End Function
```

```
End Class
```

Използване от HTML страница #1

```
<form target="_blank"
action=
'http://localhost:14572/WebSite10/Service.asmx/FahrenheitToCelsius
method="POST">
<table>
  <tr>
    <td>Fahrenheit to Celsius:</td>
    <td><input class="frmInput" type="text"
      size="30" name="Fahrenheit"></td>
  </tr>
  <tr>
    <td></td>
    <td align="right"> <input type="submit"
      value="Submit" class="button"></td>
  </tr>
</table>
</form>
```

Използване от HTML страница

```
<form target="_blank" action=
'http://localhost:14572/WebSite10/Service.asmx/CelsiusToFahrenheit'
method="POST">
<table>
  <tr>
    <td>Celsius to Fahrenheit:</td>
    <td><input class="frmInput" type="text"
      size="30" name="Celsius"></td>
  </tr>
  <tr>
    <td></td>
    <td align="right"> <input type="submit"
      value="Submit" class="button"></td>
  </tr>
</table>
</form>
```


Service

The following operations are supported. For a formal definition, please review the [Service Description](#).

- [CelsiusToFahrenheit](#)
- [FahrenheitToCelsius](#)

This web service is using `http://tempuri.org/` as its default namespace.

Recommendation: Change the default namespace before the XML Web service is made public.

Each XML Web service needs a unique namespace in order for client applications to distinguish it from other Web services that are under development, but published XML Web services should use a more permanent namespace.

Your XML Web service should be identified by a namespace that you control. For example, you can use your company name. Although many XML Web service namespaces look like URLs, they need not point to actual resources on the Internet.

For XML Web services created using ASP.NET, the default namespace can be changed using the `WebServiceNamespace` attribute applied to the class that contains the XML Web service methods. Below is a code example that shows how to change the default namespace.

Address  http://localhost:1036/WebSite1/Service.asmx?op=CelsiusToFahrenheit

Service

Click [here](#) for a complete list of operations.

CelsiusToFahrenheit

Test

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Parameter	Value
Celsius:	
Invoke	

test

Fahrenheit to Celsius:		<input type="submit" value="Submit"/>
Celsius to Fahrenheit:		<input type="submit" value="Submit"/>

```
<form target="_blank" action='http://localhost:1036/WebSite1/Service.asmx/FahrenheitToCelsius'
method="POST">
<table>
<tr>
<td>Fahrenheit to Celsius:</td>
<td><input class="frmInput" type="text"
size="30" name="Fahrenheit"></td>
</tr>
<tr>
<td></td>
<td align="right"> <input type="submit"
value="Submit" class="button"></td>
</tr>
</table>
</form>
```

wsdl

```
<?xml version="1.0" encoding="utf-8" ?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:tns="http://tempuri.org/"
  xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" targetNamespace="http://tempuri.org/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  - <wsdl:types>
    - <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
      - <s:element name="FahrenheitToCelsius">
        - <s:complexType>
          - <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="Fahrenheit" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      - <s:element name="FahrenheitToCelsiusResponse">
        - <s:complexType>
          - <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="FahrenheitToCelsiusResult" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      - <s:element name="CelsiusToFahrenheit">
        - <s:complexType>
          - <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="Celsius" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>

```

DISCO

```
<?xml version="1.0" encoding="utf-8" ?>
<discovery xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/disco/">
  <contractRef ref="http://localhost:1036/WebSite1/Service.asmx?wsdl"
    docRef="http://localhost:1036/WebSite1/Service.asmx" xmlns="http://schemas.xmlsoap.org/disco/scl/" />
  <soap address="http://localhost:1036/WebSite1/Service.asmx" xmlns:q1="http://tempuri.org/" binding="q1:ServiceSoap"
    xmlns="http://schemas.xmlsoap.org/disco/soap/" />
  <soap address="http://localhost:1036/WebSite1/Service.asmx" xmlns:q2="http://tempuri.org/" binding="q2:ServiceSoap12"
    xmlns="http://schemas.xmlsoap.org/disco/soap/" />
</discovery>
```

WEB SERVICE - 2

Поддръжка на сесии

- ◆ ASP.NET Web услугите поддържат сесии, както всички Web приложения
 - ❖ `Session` – съдържа текущата сесия
 - ❖ `Application` – съдържа контекста на Web приложението
- ◆ Разрешаване на сесията при услугата:

```
[WebMethod(EnableSession=true)]
```

- ◆ Разрешаване на сесията при клиента:

```
MyService myService = new MyService();  
myService.CookieContainer =  
    new System.Net.CookieContainer();
```


Демонстрация #11

◆ Използване на сесии – Web услуга

The image shows a screenshot of a development environment and a web browser. The top window is Microsoft Visual Studio, displaying the code for `SessionService.asmx.cs`. The code defines a `GetSessionCounter()` method that increments a session counter and returns it. The bottom window is Microsoft Internet Explorer, showing the `SessionService` web service page. The page lists the supported operations: `GetSessionCounter` and `GetApplicationCounter`.

Visual Studio Code:

```
counter = (int) Application["counter"];
}
counter++;
Application["counter"] = counter;
return counter;
}
finally
{
    Application.Unlock();
}
}

[WebMethod(EnableSession=true, Description="Returns t
public int GetSessionCounter()
{
    int counter = 0;
    if (Session["counter"] != null)
    {
        counter = (int) Session["counter"];
    }
    counter++;
    Session["counter"] = counter;
    return counter;
}
```

Internet Explorer Page:

SessionService

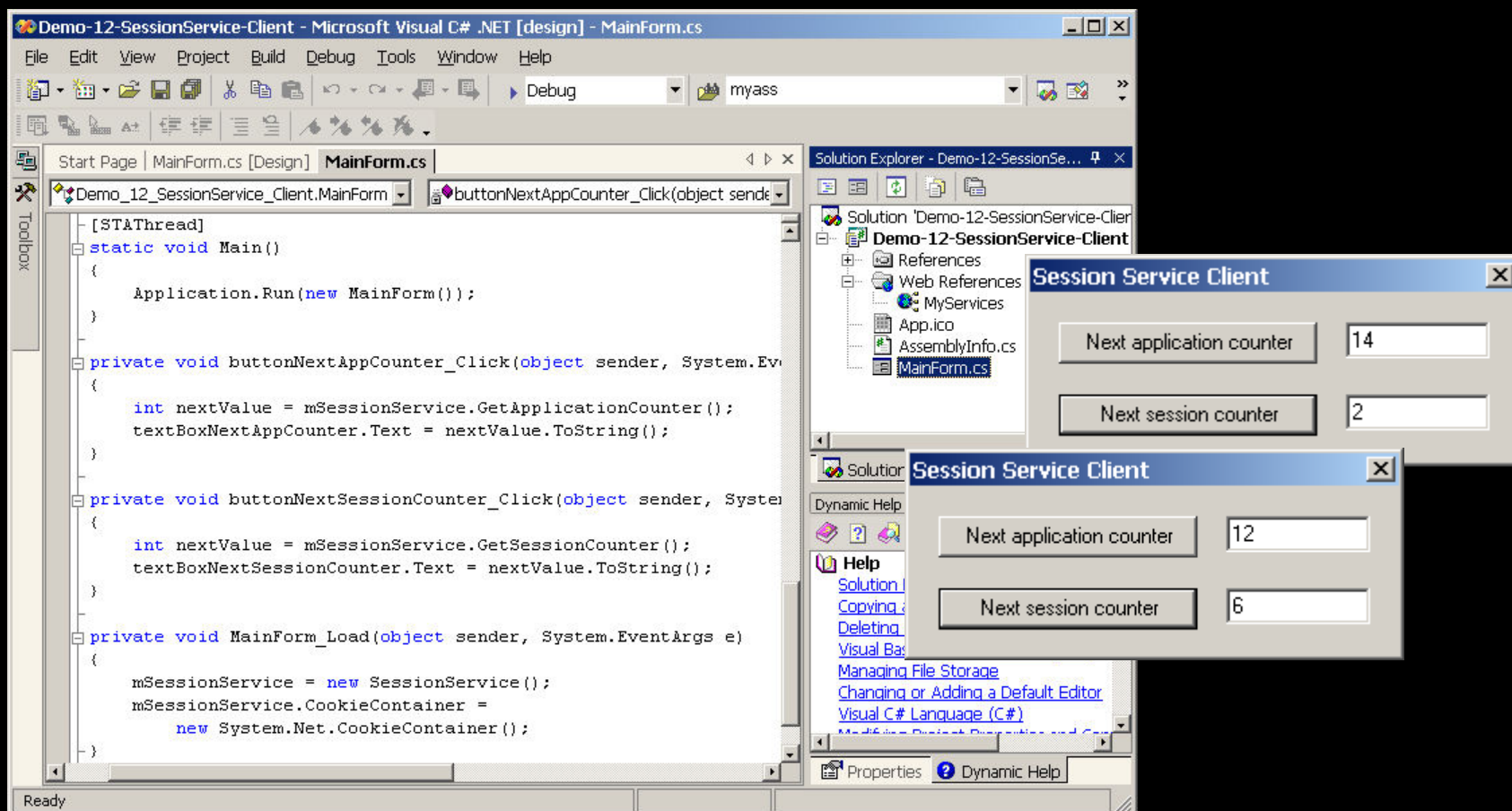
Session Web Service

The following operations are supported. For a formal definition, please review the [Service Description](#).

- [GetSessionCounter](#)
Returns the next value of the local session counter.
- [GetApplicationCounter](#)
Returns the next value of the global application counter.

Демонстрация #12

- ◆ Използване на сесии – клиент за Web услугата



Сигурност на Web услугите

- ◆ Често пъти се изисква някои методи да не могат да се викат без автентикация
- ◆ Защитата от неоторизиран достъп може да се направи по няколко начина:
 - ❖ Всеки метод да има допълнителни параметри (потребител, парола и др.)
 - ❖ Чрез използване на сесия
 - ❖ Чрез средствата на Web Services Enhancements (WSE). WSE поддържат:
 - ❖ Криптиране на трафика
 - ❖ Цифрово подписване
 - ❖ Автентикация чрез цифрови сертификати

Сигурност чрез сесии

1. Добавяме метод за автентикация на потребител:

```
[WebMethod(EnableSession = true, Description =  
    "Checks given login and creates a session.")]  
public void Login(string aUser, string aPassword)  
{  
    if ( user/pass is valid )  
    {  
        Session["user"] = aUser;  
    }  
    else  
    {  
        throw new Exception("Invalid login!");  
    }  
}
```

Сигурност чрез сесии

2. Във всеки защитен метод проверяваме дали има автентикиран потребител:

```
[WebMethod(EnableSession = true, Description =  
    "This method requires active session.")]  
public string GetProtectedData()  
{  
    string user = (string) Session["user"];  
    if (user != null)  
    {  
        return "This is protected data!";  
    }  
    else  
    {  
        throw new Exception("Access denied!");  
    }  
}
```

Демонстрация #13

◆ Защита на достъпа до Web услуга чрез сесия – Web услугата

The image shows a development environment with two windows. The left window is Microsoft Visual C# .NET [design] - SecuredService.asmx.cs, displaying the source code for the SecuredService.asmx.cs file. The code defines three web methods: Login, Logout, and GetProtectedData. The Login method checks if the credentials are valid and establishes a session. The Logout method terminates the active session. The GetProtectedData method returns some protected data, requiring an active session. The right window is Microsoft Internet Explorer, displaying the SecuredService Web Service. The page title is 'SecuredService' and the subtitle is 'Secured Web Service'. The page content states: 'The following operations are supported. For a formal definition, please review the [Service Description](#).' The supported operations are:

- [GetNotProtectedData](#): Returns some data. Does not require a session.
- [Login](#): Checks given credentials and establishes a session.
- [GetProtectedData](#): Returns some protected data. Requires active session.
- [Logout](#): Terminates the active session (if any).

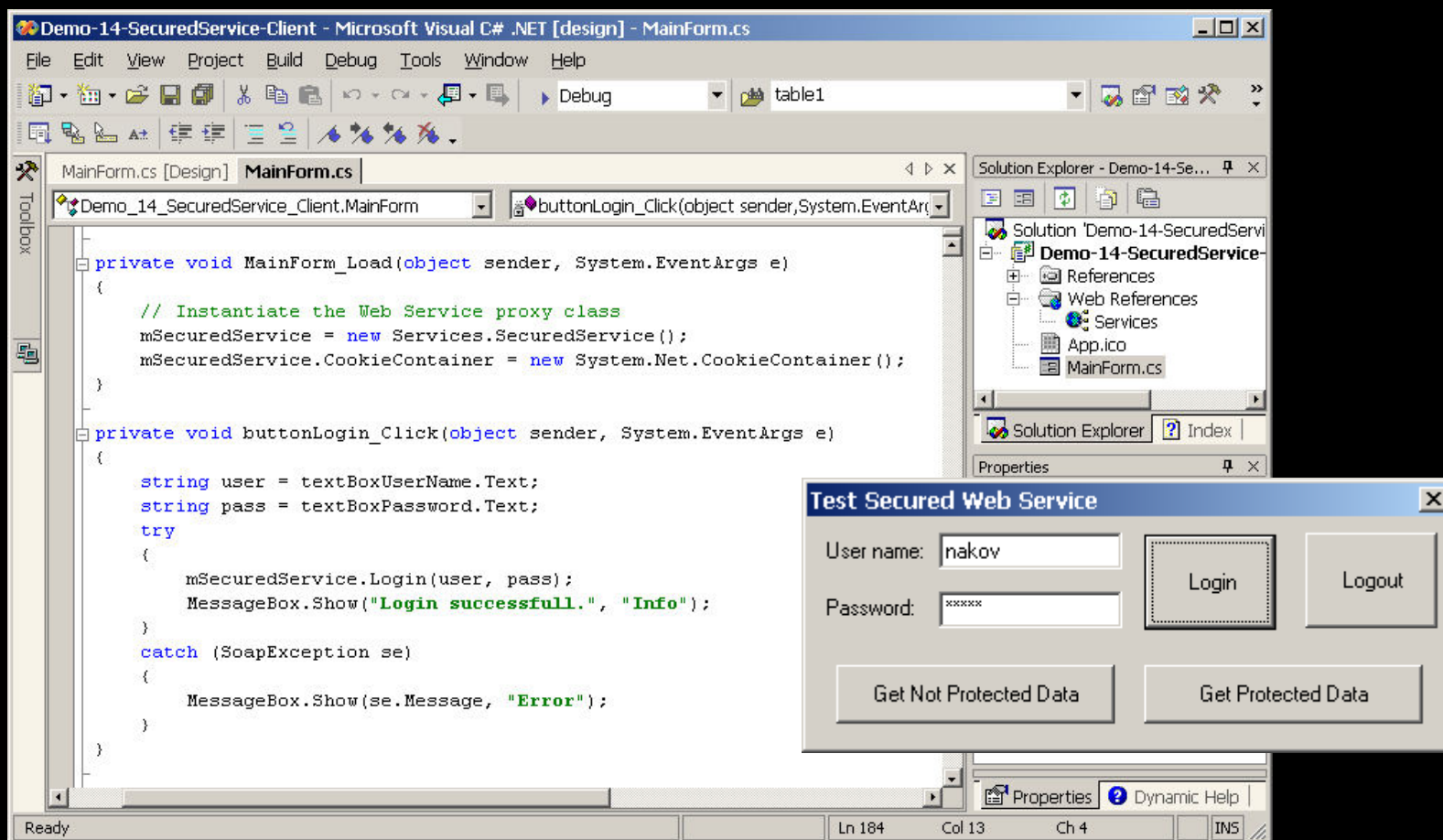
```
[WebMethod(EnableSession=true,
Description="Checks given credentials and establishes a session.")
public void Login(string aUserName, string aPassword)
{
    if (IsLoginValid(aUserName, aPassword))
    {
        Session[USER_NAME_SESSION_KEY] = aUserName;
    }
    else
    {
        throw new AuthenticationException("Invalid credentials!");
    }
}

[WebMethod(EnableSession=true,
Description="Terminates the active session (if any).")
public void Logout()
{
    Session.Abandon();
}

[WebMethod(EnableSession=true,
Description="Returns some protected data. Requires active session.")
public string GetProtectedData()
{
    return "Protected Data";
}
```

Демонстрация #14

- ◆ Защита на достъпа до Web услуга чрез сесия – клиент за Web услугата



Исключенията в Web услугите

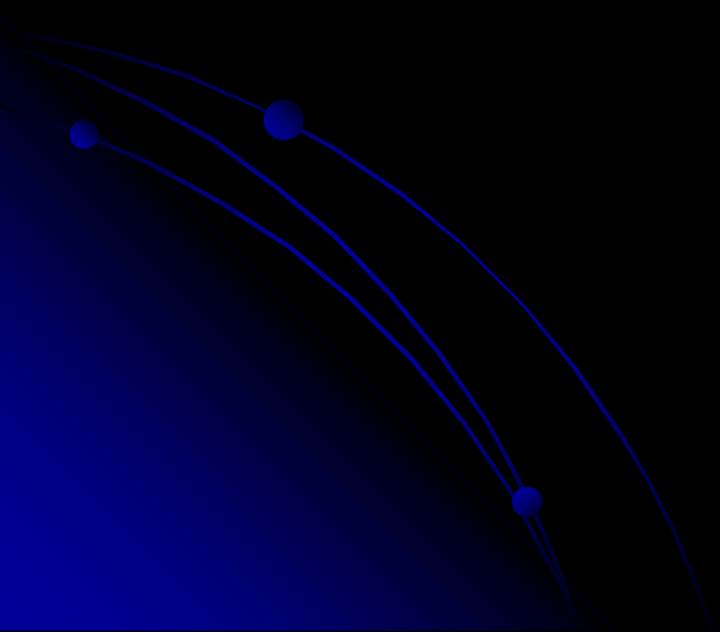
- ◆ Когато Web услуга хвърли изключение, клиентът получава `SoapException`
 - ❖ Оригиналният тип на изключението се губи
 - ❖ Губят се вложените (inner) изключения
 - ❖ Запазва се текстът (свойството `Message`)
- ◆ За да връщаме информация за проблем, възникнал в Web метод:
 - ❖ Дефинираме изброен тип (`enum`), съдържащ възможните грешки
 - ❖ Връщаме от Web метода код на грешка – инстанция на изброения тип
- ◆ Клиентът проверява кода за грешка

Исключенията в Web услугите

```
[WebMethod]
public UserError AddUser(string aUser, string aPass)
{
    if ( aUser is invalid )
        return UserError.InvalidUser;
    if ( aPass is invalid )
        return UserError.InvalidPassword;
    ...
    return UserError.Success;
}

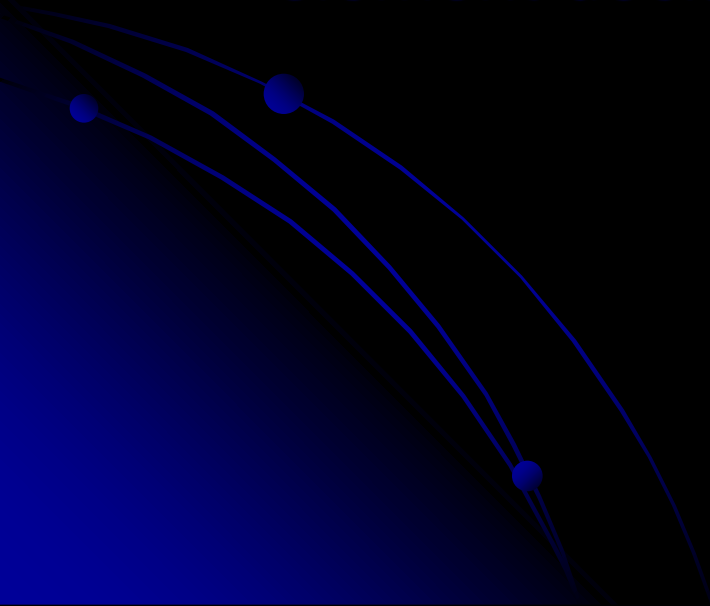
public enum UserError
{
    Success,
    UserAlreadyExists,
    InvalidUser,
    InvalidPassword,
    DatabaseError
}
```


XML SCHEMA



Schema

- A schema is a collection of:
 - type definitions
 - simple type
 - complex type (contains element or attribute)
 - element declarations



```

<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>

```

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Purchase order schema for Example.com.
      Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items" type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>

  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN"
      fixed="US"/>
  </xsd:complexType>

  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity">
              <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                  <xsd:maxExclusive value="100"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="USPrice" type="xsd:decimal"/>
            <xsd:element ref="comment" minOccurs="0"/>
            <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="partNum" type="SKU" use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  <!-- Stock Keeping Unit, a code for identifying products -->
  <xsd:simpleType name="SKU">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-[A-Z]{2}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>

```

Complex Type Definitions

Defining the USAddress Type

```
<xsd:complexType name="USAddress" >
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>
```

- The elements must appear in the same sequence
- The attribute must contain "US"

Complex Type Definition

Defining PurchaseOrderType

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```

- The “ref” attribute is for referencing an existing element
- The value of this attribute must reference a global element (declared directly under schema)

Occurrence Constraints

- For elements:
 - minOccurs: default 1
 - maxOccurs: default 1
- For attributes the “use” attribute has the following value:
 - required
 - optional (default)
 - prohibited
- “default” attribute is for both attribute and element definitions
- default attribute values apply when attributes are missing, default element values apply when elements are empty

Global Elements

- The declaration enables the element to appear at the top-level of an instance document
- In the example, both `purchaseOrder` and `comment` can be the root
- cardinality constraints can not be put to global declarations

Simple Types

- string
- normalizedString
- token
- byte
- unsignedByte
- base64Binary
- hexBinary
- integer
- positiveInteger
- negativeInteger
- nonNegativeInteger
- nonPositiveInteger
- int
- unsignedInt
- long
- unsignedLong
- decimal
- float
- double
- boolean
- time
- dateTime
- duration
- date
- anyURI
- language
-

New simple types

- by deriving from existing simple types (restricting)

Defining myInteger, Range 10000-99999

```
<xsd:simpleType name="myInteger">  
  <xsd:restriction base="xsd:integer">  
    <xsd:minInclusive value="10000"/>  
    <xsd:maxInclusive value="99999"/>  
  </xsd:restriction>  
</xsd:simpleType>
```

Defining the Simple Type "SKU"

```
<xsd:simpleType name="SKU">  
  <xsd:restriction base="xsd:string">  
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>  
  </xsd:restriction>  
</xsd:simpleType>
```

Enumeration

Using the Enumeration Facet

```
<xsd:simpleType name="USState">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="AK"/>
    <xsd:enumeration value="AL"/>
    <xsd:enumeration value="AR"/>
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>
```

List Types

Creating a List of myInteger's

```
<xsd:simpleType name="listOfMyIntType">  
  <xsd:list itemType="myInteger"/>  
</xsd:simpleType>
```

```
<listOfMyInt>20003 15037 95977 95945</listOfMyInt>
```

List Type for Six US States

```
<xsd:simpleType name="USStateList">  
  <xsd:list itemType="USState"/>  
</xsd:simpleType>  
  
<xsd:simpleType name="SixUSStates">  
  <xsd:restriction base="USStateList">  
    <xsd:length value="6"/>  
  </xsd:restriction>  
</xsd:simpleType>
```

```
<sixStates>PA NY CA NY LA AK</sixStates>
```

Union Type

Union Type for Zipcodes

```
<xsd:simpleType name="zipUnion">  
  <xsd:union memberTypes="USState listOfMyIntType"/>  
</xsd:simpleType>
```

```
<zip>CA</zip>
```

```
<zip>95630 95977 95945</zip>
```

```
<zip>AK</zip>
```

Anonymous Type

Two Anonymous Type Definitions

```
<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName" type="xsd:string"/>
          <xsd:element name="quantity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="USPrice" type="xsd:decimal"/>
          <xsd:element ref="comment" minOccurs="0"/>
          <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="partNum" type="SKU" use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Complex Types from Simple Types

```
<internationalPrice currency="EUR">423.46</internationalPrice>
```

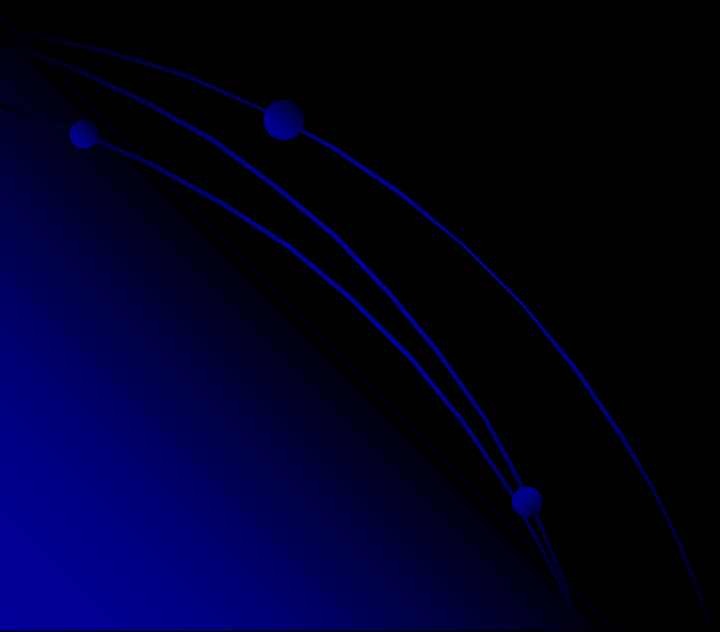
```
<xsd:element name="internationalPrice">  
  <xsd:complexType>  
    <xsd:simpleContent>  
      <xsd:extension base="xsd:decimal">  
        <xsd:attribute name="currency" type="xsd:string"/>  
      </xsd:extension>  
    </xsd:simpleContent>  
  </xsd:complexType>  
</xsd:element>
```

Empty Content

```
<xsd:element name="internationalPrice">  
  <xsd:complexType>  
    <xsd:attribute name="currency" type="xsd:string"/>  
    <xsd:attribute name="value" type="xsd:decimal"/>  
  </xsd:complexType>  
</xsd:element>
```

Content Structure

- all
- sequence
- choice



Choice & Group

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:group ref="shipAndBill"/>
      <xsd:element name="singleUSAddress" type="USAddress"/>
    </xsd:choice>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

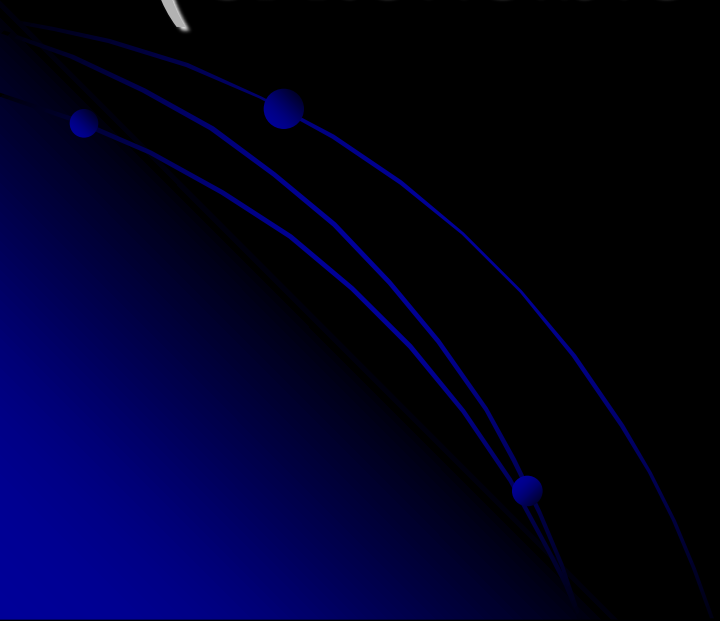
<xsd:group name="shipAndBill">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
  </xsd:sequence>
</xsd:group>
```

All

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:all>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:all>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```

XML

(eXtensible Markup Language)



Съдържание

- Какво е XML?
- XML и HTML
- Кога се използва XML?
- Пространства от имена
- Схеми и валидация – DTD, XSD и XDR схеми
- Редакторът за схеми на VS.NET
- XML парсери
- XML поддръжка в .NET Framework
- Работа с DOM парсера – класовете XmlNode и XmlDocument

Съдържание

- SAX парсери и класът `XmlReader`
- Кога да използваме DOM и кога SAX?
- Създаване на XML документи с `XmlWriter`
- Валидация на XML по схема
- Работа с XPath – класовете `XPathNavigator` и `XPathDocument`
- XSL трансформации в .NET Framework

Какво е XML?

- XML е:
 - универсален език (нотация) за описание на структурирани данни
 - данните се съхраняват заедно с метаянформация за тях
 - прилича на HTML – текстово-базиран, използва тагове и атрибути
 - с него се описват други езици (формати) за представяне на данни
 - световно-утвърден стандарт, поддържан от W3C (www.w3c.org)
 - независим от платформата, езиците за програмиране и операционната система

XML – пример

```
<?xml version="1.0"?>
<library name=".NET Developer's Library">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

XML – р

заглавна
част
(пролог)

атрибут

```
<?xml version="1.0"?>
```

```
<library name=".NET Developer's Library">
```

```
<book>
```

отварящ
таг

```
<title>Programming Microsoft .NET</title>
```

```
<author>Jeff Prosise</author>
```

```
<isbn>0-7356-1376-1</isbn>
```

```
</book>
```

елемент

```
<book>
```

затварящ
таг

```
<title>Microsoft .NET for Programmers</title>
```

```
<author>Fergal Grimes</author>
```

```
<isbn>1-930110-19-7</isbn>
```

```
</book>
```

стойност на елемент

```
</library>
```


XML и HTML

- Прилики между езиците XML и HTML:
 - и двата са текстово базирани
 - използват тагове и атрибути
- Разлики между езиците XML и HTML:
 - HTML е език, а XML е синтаксис за описание на други езици
 - HTML описва форматирането на информацията, а XML описва структурирана информация
 - XML изисква документите да са добре дефинирани (well-formed)

Добре дефинирани документи

- XML изисква документите да са добре дефинирани (well-formed):
 - таговете винаги да се затварят и то в правилния ред (да не се застъпват)
 - атрибутите винаги да се затварят
 - да има само един основен root елемент
 - ограничения върху имената на таговете и атрибутите

● Пример за лошо-дефиниран документ:

```
<xml>  
  <button bug! value="OK name="b1">  
    <animation source="demo1.avi"> 1 < 2 < 3  
  </click-button>  
< / xml >
```

Кога се използва XML?

- XML се използва:

- за обмяна на информация между различни системи
- за съхранение на структурирани данни
- за създаване на собствени езици за описание на информация

- Недостатъци на XML:

- данните са по-обемисти
- намалена производителност
- често пъти е необходима много памет
- увеличаване на мрежовия трафик

Пространства от имена

- Пространствата от имена (namespaces) в XML документите позволяват дефиниране и

използване на такова съдържание:

```
<?xml version="1.0" encoding="UTF-8"?>
<country:towns xmlns:country="urn:koko-com:country"
  xmlns:town="http://www.koko.com/town">
  <town:town>
    <town:name>Sofia</town:name>
    <town:population>1 200 000</town:population>
    <country:name>Bulgaria</country:name>
  </town:town>
  <town:town>
    <town:name>Plovdiv</town:name>
    <town:population>700 000</town:population>
    <country:name>Bulgaria</country:name>
  </town:town>
</country:towns>
```

Пространства от имена

```
<?xml version="1.0" encoding="UTF-8"?>
<country:towns xmlns:country="urn:koko-com:country"
  xmlns:town="http://www.koko.com/town">
  <town:town>
    <town:name>
    <town:popul
    <country:name>Bulgaria</country:name>
  </town:town>
  <town:town>
    <town:na
    <town:population>700 000</town:population>
    <country:name>Bulgaria</country:name>
  </town:town>
</country:towns>
```

пространство с префикс "country" и
URI идентификатор "urn:koko-
com:country"

таг с име "name" от пространството
"country", т.е. пълното име на тага е
"urn:koko-com:country:name"

Пространства от имена

- Позволява се използването на пространства

```
<?xml version="1.0" encoding="windows-1251"?>
<order xmlns="http://www.hranitelni-stoki.com/orders">
  <item>
    <name>бира "Загорка"</name>
    <ammount>8</ammount>
    <measure>бутилка</measure>
    <price>3.76</price>
  </item>
  <item>
    <name>кебапчета</name>
    <ammount>6</ammount>
    <measure>60</measure>
    <price>1.20</price>
  </item>
</order>
```

пространство по
подразбиране

ПЪЛНОТО ИМЕ НА ТАГА "item" е
"http://www.hranitelnik-stoki.com:item"

Схеми и валидация

- Структурата на XML документите се описва със схеми
- Схемите описват:
 - допустимите тагове
 - допустимите атрибути на таговете
 - допустимите стойности за атрибутите и на елементите
 - реда на поставяне на таговете
 - стойности по подразбиране
- Има няколко стандарта за XML схеми:
 - DTD – Document Type Definition
 - XSD – XML Schema Definition Language
 - XDR – XML -Data Reduced

Езикът DTD

- DTD (Document Type Definition) е:
 - формален език за описание структурата на XML документи
 - съдържа съвкупност от правила за таговете в документа и техните атрибути
 - текстово-базиран език, но не е базиран на XML
 - използва се рядко, защото е заместен от XSD

Пример:

```
<!ELEMENT library (book+)>
<!ATTLIST library
  name CDATA #REQUIRED
>
<!ELEMENT book (title, author, isbn)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT isbn (#PCDATA)>
```


XSD схеми

- XSD (XML Schema Definition Language) е:
 - мощен XML-базиран език за описание структурата на XML документи
 - съдържа съвкупност от правила за таговете в

`<?xml version="1.0" encoding="UTF-8"?>`

`<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">`

`<xs:element name="library">`

`<xs:complexType>`

`<xs:sequence>`

`<xs:element ref="book"
 maxOccurs="unbounded"/>`

`</xs:sequence>`

`<xs:attribute name="name"
 type="xs:string" use="optional"/>`

`</xs:complexType>`

`</xs:element>`

(примерът продължава)

XSD схеми

```
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="title"/>
      <xs:element ref="author"/>
      <xs:element ref="isbn"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="title" type="xs:string"/>
<xs:element name="author" type="xs:string"/>
<xs:element name="isbn" type="xs:string"/>

</xs:schema>
```

- XSD схемите имат по-голяма изразителна мощ от DTD
- XSD постепенно измества DTD

XDR схеми

- XDR (XML-Data Reduced) е:
 - Език за описание на структурата на XML документи
 - Компактен вариант на XML-Data схемите
 - XML базиран език, подобен на XSD
 - По-мощен от DTD
 - По-слабо изразителен от XSD
 - Въведен от Microsoft, използван често в техни продукти и технологии (напр. в SQL Server 2000, BizTalk, ...)
 - Може да описва съответствия между структурата на XML документи и релационни бази данни

XDR схеми – пример

```
<?xml version="1.0" encoding="UTF-8"?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
  xmlns:dt="urn:schemas-microsoft-com:datatypes">

  <ElementType name="author" model="closed"
    content="textOnly" dt:type="string"/>

  <ElementType name="title" model="closed"
    content="textOnly" dt:type="string"/>

  <ElementType name="isbn" model="closed"
    content="textOnly" dt:type="string"/>

  <ElementType name="book" model="closed"
    content="eltOnly" order="seq">
    <element type="title" minOccurs="1" maxOccurs="1"/>
    <element type="author" minOccurs="1" maxOccurs="1"/>
    <element type="isbn" minOccurs="1" maxOccurs="1"/>
  </ElementType>
```

(примерът продължава)

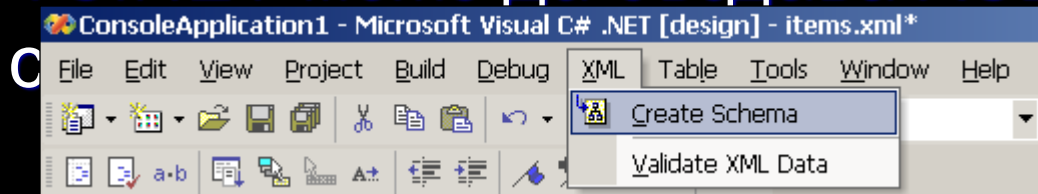
XDR схеми – пример

```
<ElementType name="library" model="closed"
  content="eltOnly" order="seq">
  <AttributeType name="name" dt:type="string"
    required="yes"/>
  <attribute type="name"/>
  <AttributeType name="xmlns" dt:type="string"/>
  <attribute type="xmlns"/>
  <element type="book" minOccurs="1" maxOccurs="*" />
</ElementType>
```

```
</Schema>
```

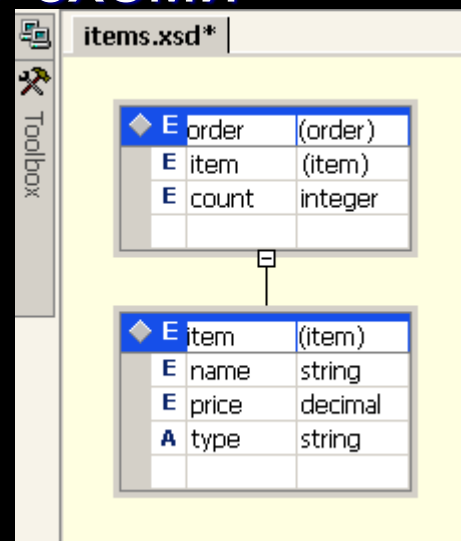
Редакторът за схеми на VS.NET

- VS.NET може да създава XSD схема по документ



- VS.NET има редактор за XSD схеми

- С него можем да редактираме схемите
- Можем да създаваме, променяме и изтриваме:
 - елементи
 - атрибути на елементи
 - ТИПОВЕ



Демонстрация #0

• Средствата на VS.NET за създаване и

The screenshot displays the Microsoft Visual Studio .NET environment with two windows open:

- order.xsd**: Shows an XML Schema design view. The schema defines three elements: `order` (root), `items`, and `item`.
 - `order` contains attributes `customer-na` (string), `customer-ad` (string), and `date` (dateTime), and a child element `items`.
 - `items` contains a child element `item`.
 - `item` contains attributes `name` (string) and `price` (decimal).
- ConsoleApplication1 - Microsoft Visual C# .NET [design] - items.xml***: Shows the XML data view for `items.xml`. The `XML` tab is active, displaying the `Create Schema` and `Validate XML Data` buttons.

The **Toolbox** on the left lists XML Schema components: Pointer, element, attribute, attributeGroup, complexType, simpleType, group, any, anyAttribute, facet, key, and Relation. The **Properties** window on the right shows the `Dataset` and `Misc` properties for the selected element.

XML парсери

- XML парсерите са програмни библиотеки, които улесняват работата с XML
- Служат за:
 - извличане на данни от XML документи
 - построяване на XML документи
 - валидация на XML документи по дадена схема
- По начин на работа биват:
 - DOM (Document Object Model) – представя XML документите като дърво в паметта и позволява лесна обработка
 - SAX (Simple API for XML Processing) – чете XML документите последователно като поток и

XML и .NET Framework

- В .NET Framework:
 - Средствата за работа с XML се намират в пространството `System.Xml`
 - Има пълна реализация на DOM модела
 - чрез класовете `XmlDocument`, `XmlNode`, ...
 - XML документът се зарежда целият като дърво в паметта и след това се обработва
 - Има класове, с функционалност подобна на SAX, но няма чиста SAX имплементация
 - класовете `XmlReader` и `XmlWriter` четат и пишат XML документи последователно елемент по елемент

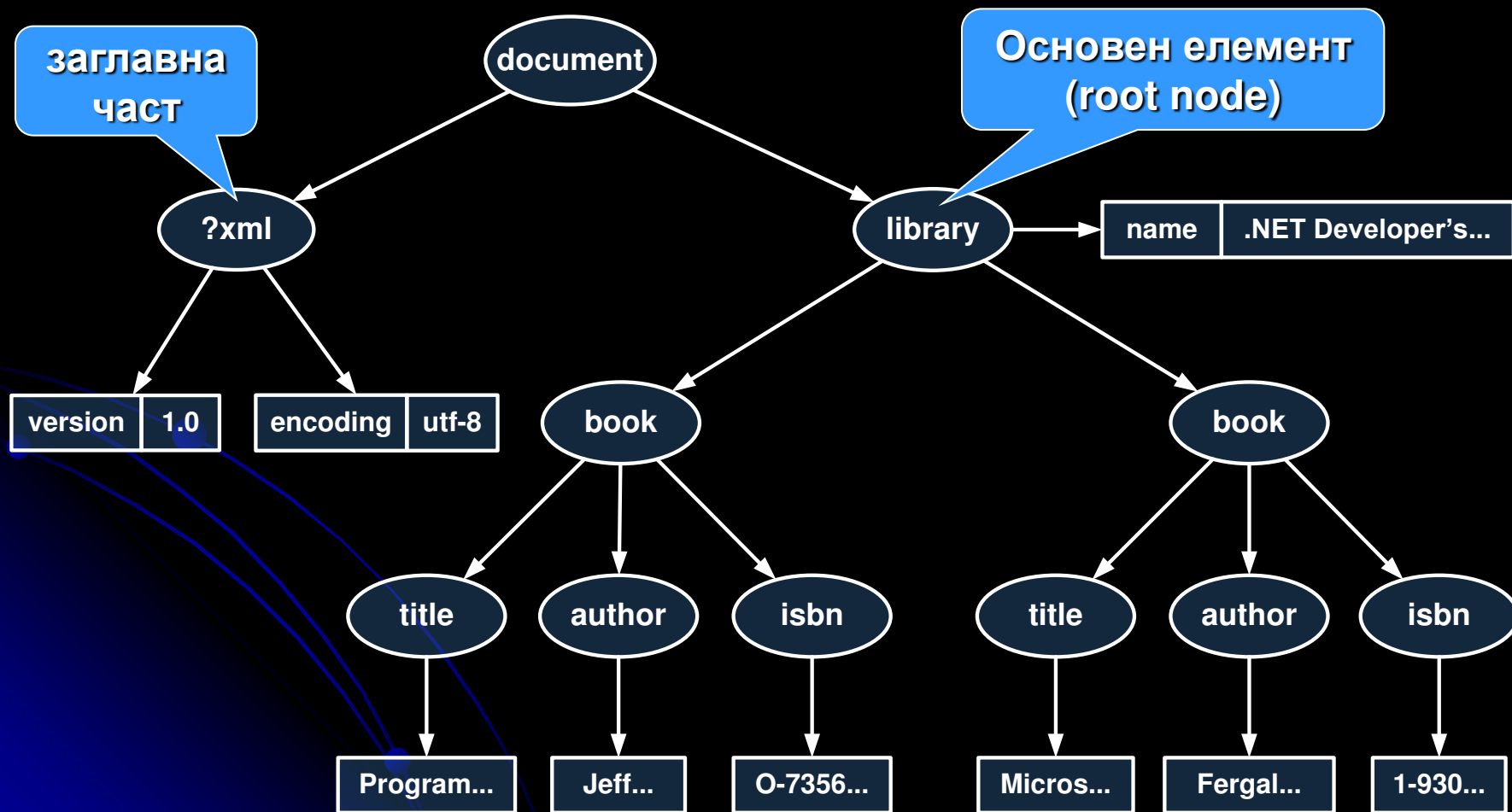
Работа с DOM парсера

- Даден е следния XML документ:

```
<?xml version="1.0"?>
<library name=".NET Developer's Library">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

Работа с DOM парсера

- Този документ се представя в паметта като DOM дърво по следния начин:



DOM парсер – пример

```
XmlDocument doc = new XmlDocument();  
doc.Load("library.xml");  
  
XmlNode rootNode = doc.DocumentElement;  
Console.WriteLine("Root node: {0}", rootNode.Name);  
  
foreach (XmlAttribute atr in rootNode.Attributes)  
{  
    Console.WriteLine("Attribute: {0}={1}",  
        atr.Name, atr.Value);  
}  
  
foreach (XmlNode node in rootNode.ChildNodes)  
{  
    Console.WriteLine("\nBook title = {0}",  
        node["title"].InnerText);  
    Console.WriteLine("Book author = {0}",  
        node["author"].InnerText);  
    Console.WriteLine("Book isbn = {0}",  
        node["isbn"].InnerText);  
}
```

Демонстрация #1

• Парсинг XML DOM

The screenshot displays the Microsoft Visual Studio IDE with the following components:

- File Explorer:** Shows the project structure for 'Demo-1-DOM-Parser'.
- Code Editor:** Contains the `DOMParserDemo.cs` file. The code is as follows:

```
XmlNode rootNode = doc.DocumentElement;
Console.WriteLine("Root node: {0}", rootNode.Name);

foreach (XmlAttribute atr in rootNode.Attributes)
{
    Console.WriteLine("Attribute: {0}={1}",
        atr.Name, atr.Value);
}

Console.WriteLine();

foreach (XmlNode node in rootNode.ChildNodes)
{
    Console.WriteLine("Book title = {0}",
```
- Debugger:** A breakpoint is set at the line `Console.WriteLine("Attribute: {0}={1}", atr.Name, atr.Value);`.
- Autos Window:** Displays the current state of variables:

Name	Value	Type
atr	{System.Xml.XmlAttribute}	System.Xml.XmlAttribute
atr.Name	"name"	string
atr.Value	".NET Developer's Library"	string
rootNode	{System.Xml.XmlElement}	System.Xml.XmlNode
rootNode.Attributes	{System.Xml.XmlAttributeCollection}	System.Xml.XmlAttributeCollection
- Call Stack:** Shows the current method call: `Demo-1-DOM-Parser.exe!DOM C.`

Класовете за работа с DOM

- За работа с DOM се използват класовете:
 - XmlNode – абстрактен базов клас за всички възли в едно DOM дърво
 - XmlDocument – съответства на корена на DOM дърво, обикновено съдържа два наследника:
 - заглавна част (пролог) на XML документа
 - елемент-корен на XML документа
 - XmlElement – представя XML елемент
 - XmlAttribute – представя атрибут на XML елемент (двойка име-стойност)
 - XmlAttributeCollection – списък от XML атрибути
 - XmlNodeList – списък от възли в DOM дърво

Класът XmlNode

- Класът `System.Xml.XmlNode`:
 - е основополагащ при работата с DOM
 - представлява базов възел, а неговите наследници са типовете DOM възли:
 - `XmlDocument`, `XmlElement`, `XmlAttribute`, ...
 - позволява навигация в DOM дървото:
 - `ParentNode` – връща възела-родител (или `null` ако няма)
 - `PreviousSibling` / `NextSibling` – връща левия / десния съсед на текущия възел
 - `FirstChild` / `LastChild` – връща първия / последния наследник на текущия възел
 - `Item` (индексатор в C#) – връща наследник на текущия възел по името му

Класът XmlNode

- Класът `System.Xml.XmlNode`:
 - позволява работа с текущия възел:
 - `Name` – връща името на възела (име на елемент, атрибут, ...)
 - `Value` – стойността на възела
 - `Attributes` – списък от атрибутите на възела (като `XmlAttributeCollection`)
 - `HasChildNodes` – дали има наследници
 - `InnerXml`, `OuterXml` – връща частта от XML документа, която описва съдържанието на възела съответно с и без него самия
 - `InnerText` – конкатенация от стойностите на възела и наследниците му рекурсивно
 - `NodeType` – връща типа на възела (вж. изброяения тип `XmlNodeType` в MSDN)

Класът XmlNode

- Класът `System.Xml.XmlNode`:
 - позволява промяна на текущия възел:
 - `AppendChild(...)` / `PrependChild(...)` – добавя нов наследник след / преди всички други наследници на текущия възел
 - `InsertBefore(...)` / `InsertAfter(...)` – вмъква нов наследник преди / след указан наследник
 - `RemoveChild(...)` / `ReplaceChild(...)` – премахва / заменя указания наследник
 - `RemoveAll()` – изтрива всички наследници на текущия възел (атрибути, елементи, ...)

Класът XmlDocument

- Класът `System.Xml.XmlDocument`:
 - съдържа XML документ във вид на DOM дърво
 - позволява зареждане и съхранение на XML документи от/във файл, поток или символен низ (вж. `Load(...)`, `LoadXml(...)`, `Save(...)`)
- По-важни свойства, методи и събития:
 - `DocumentElement` – извлича елемента-корен
 - `PreserveWhitespace` – указва дали празното пространство да бъде запазено при зареждане/записване на документа
 - `CreateElement(...)`, `CreateAttribute(...)`, `CreateTextNode(...)` – създава нов XML елемент, атрибут или стойност на елемент
 - `NodeChanged`, `NodeInserted`, `NodeRemoved` – събития за следене за промени в документа

Работа с DOM – пример

- Удвояване на цените на бирата в следния

XML документ

```
<?xml version="1.0" encoding="windows-1251"?>
<items culture="en-US">
  <item type="beer">
    <name>Загорка</name>
    <price>0.54</price>
  </item>
  <item type="food">
    <name>кебапчета</name>
    <price>0.48</price>
  </item>
  <item type="beer">
    <name>Каменица</name>
    <price>0.56</price>
  </item>
</items>
```

Работа с DOM – пример

```
static void Main()
{
    XmlDocument doc = new XmlDocument().Load("items.xml");
    string culture = doc.DocumentElement.
        Attributes["culture"].Value;
    CultureInfo numberFormat = new CultureInfo(culture);
    foreach (XmlNode node in doc.DocumentElement)
    {
        if (node.Attributes["type"].Value == "beer")
        {
            string currentPriceStr =
                node["price"].InnerText;
            decimal currentPrice = Decimal.Parse(
                currentPriceStr, numberFormat);
            decimal newPrice = currentPrice * 2;
            node["price"].InnerText =
                newPrice.ToString(numberFormat);
        }
    }
    doc.Save("itemsNew.xml");
}
```

Демонстрация #2

Пр

The screenshot displays the Microsoft Visual Studio IDE with a C# project named "Demo-2-Modify-XML". The main code file, "ModifyXMLDemo.cs", contains a `Main` method that iterates through XML nodes. It specifically targets nodes with the attribute `type = "beer"` and updates their `price` by doubling the current value. The current price is 0.54, and the new price is 1.08. The `node["price"].InnerText` property is highlighted in yellow in the original image.

```
foreach (XmlNode node in doc.DocumentElement)
{
    if (node.Attributes["type"].Value == "beer")
    {
        string currentPriceStr =
            node["price"].InnerText;
        decimal currentPrice = Decimal.Parse(
            currentPriceStr, numberFormat);
        decimal newPrice = currentPrice * 2;
        node["price"].InnerText =
            newPrice.ToString(numberFormat);
    }
}

Console.WriteLine(doc.OuterXml);
```

The Solution Explorer on the right shows the project structure, including "References" and "ModifyXMLDemo.cs". The Properties window is empty. The Autos window at the bottom left shows the current state of variables:

Name	Value	Type
currentPrice	0.54	System.Decimal
newPrice	1.08	System.Decimal
node	{System.Xml.XmlElement}	System.Xml.XmlNode
numberFormat	{System.Globalization.CultureInfo}	System.Globalization.CultureInfo

The Call Stack window at the bottom right shows the current execution point in the `Demo-2-Modify-XML.exe!Main` method. The status bar at the bottom indicates "Build succeeded" and the current cursor position is at Line 23, Column 17, Character 5.

SAX парсери и XmlReader

- В .NET Framework няма имплементация на класически SAX парсер
- Класическите SAX парсери:
 - обхождат документа последователно
 - извикват callback функции при срещане на определени възли
- В .NET последователната обработка на XML документи се извършва с XmlReader
- XmlReader е абстрактен клас, който:
 - предоставя еднопосочен достъп само за четене до XML данни
 - работи като поток, но чете XML документи
 - прочетените на всяка стъпка данни могат да се извличат и анализират от програмиста

Класът XmlReader

- Класът `XmlReader` предоставя следните методи и свойства:
 - `Read()` – прочита следващия възел от XML документа или връща `false` ако няма следващ
 - `NodeType` – връща типа на прочетения възел
 - `Name` – връща името на прочетения възел (име на елемент, на атрибут, ...)
 - `HasValue` – връща дали възелът има стойност
 - `Value` – връща стойността на възела
 - `ReadElementString()` – прочита стойността (текста) от последния прочетен елемент
 - `AttributeCount`, `GetAttribute(...)` – за извличане на атрибутите на XML елемент

Класът XmlReader

- XmlReader е абстрактен клас
- За работа с него се използват неговите наследници:
 - XmlTextReader – за четене от файл или поток
 - XmlNodeReader – за четене от възел в DOM дърво
 - XmlValidatingReader – за валидация по XSD, DTD или XDR схема при четене от друг XmlReader

Начин на използване.

```
XmlTextReader reader = new XmlTextReader("some-file.xml");  
while (reader.Read()) {  
    // Analyze the read node  
}
```


XmlReader – пример

- Даден е XML документ, съдържащ списък от

КНИГИ:
<?xml version="1.0"?>

<library name=".NET Developer's Library">

<book>

<title>Programming Microsoft .NET</title>

<author>Jeff Prosise</author>

<isbn>0-7356-1376-1</isbn>

</book>

<book>

<title>Microsoft .NET for Programmers</title>

<author>Fergal Grimes</author>

<isbn>1-930110-19-7</isbn>

</book>

</library>

XmlReader – пример

- Извличане на всички заглавия:

```
XmlTextReader reader = new XmlTextReader("library.xml");
while (reader.Read())
{
    if ((reader.NodeType == XmlNodeType.Element) &&
        (reader.Name == "title"))
    {
        Console.WriteLine(reader.ReadElementString());
    }
}
```

- Извличане на имената на всички елементи:

```
XmlTextReader reader = new XmlTextReader("library.xml");
while (reader.Read())
{
    if (reader.NodeType == XmlNodeType.Element)
        Console.WriteLine(reader.Name);
}
```

Демонстрация #3

Demo-3-XmlReader - Microsoft Visual C# .NET [break] - XmlReaderDemo.cs

File Edit View Project Build Debug Tools Window Help

Debug main

XmlReaderDemo.cs

XmlReaderDemo

```
class XmlReaderDemo
{
    static void Main(string[] args)
    {
        Console.WriteLine("Book titles in the library:");
        XmlTextReader reader = new XmlTextReader("../..");
        while (reader.Read())
        {
            if ((reader.NodeType == XmlNodeType.Element)
                (reader.Name == "title"))
            {
                Console.WriteLine(reader.ReadElementString());
            }
        }
    }
}
```

Solution Explorer - Demo-3-XmlR...

Solution 'Demo-3-XmlReader' (1 p...

Demo-3-XmlReader

References

XmlReaderDemo.cs

Autos

Name	Value	Type
reader	{System.Xml.XmlTextReader}	System.Xml.XmlTextReader
reader.Name	"book"	string
reader.NodeType	Element	System.Xml.XmlNodeType

Watch 1

Call S... Brea... Task...

Build succeeded

Ln 12 Col 13 Ch 4 INS

"C:\MS Content and Curriculum\PPT\Lecture-12-Working-with-XML\Demo-3-...

Book titles in the library:
Programming Microsoft .NET
Microsoft .NET for Programmers

Element names in the XML file:
library
book
title
author
isbn
book
title
author
isbn
Press any key to continue.

Кога да използваме DOM и SAX?

- Моделът за обработка на XML документи DOM (`XmlDocument`) е подходящ когато:
 - обработваме малки по обем документи
 - нуждаем се от гъвкавост при навигацията
 - имаме нужда от пряк достъп до отделните възли на документа
 - желаем да променяме документа
- Моделът за обработка на XML документи SAX (`XmlReader`) е подходящ когато:
 - обработваме големи по обем документи
 - скоростта на обработка е важна

Работа с XmlWriter

- Класът `XmlWriter` позволява създаване на XML документи
- Работи като поток, но пише в XML документи
- `XmlWriter` предлага следните методи:
 - `WriteStartDocument()` – добавя пролог частта в началото на документа (`<?xml ...`)
 - `WriteStartElement(...)` – добавя отварящ таг
 - `WriteEndElement()` – затваря последния таг
 - `WriteAttributeString(...)` – добавя атрибут в текущия елемент
 - `WriteElementString(...)` – добавя елемент по зададено име и текстова стойност
 - `WriteEndDocument()` – затваря всички тагове и изпразва вътрешните буфери (чрез `Flush()`)

Работа с XmlWriter

- `XmlWriter` е абстрактен клас и не се инстанцира директно
- `XmlTextWriter` пише XML във файлове и потоци
 - Има допълнителни методи и свойства:
 - `Format`, `Indentation`, `IndentChar` – задават настройките за отместване навътре на вложените тагове
 - `QuoteChar` – задава символа за отделяне на стойностите на атрибутите
 - В конструктора може да се задава кодирането, което да се използва
 - По подразбиране се използва UTF-8

Работа с XmlWriter – пример

```
public static void Main()
{
    XmlTextWriter writer = new XmlTextWriter("lib.xml",
        Encoding.GetEncoding("windows-1251"));
    writer.Formatting = Formatting.Indented;
    writer.IndentChar = '\t';
    writer.Indentation = 1;
    try
    {
        writer.WriteStartDocument();
        writer.WriteStartElement("library");
        writer.WriteAttributeString("name", "My Library");
        WriteBook(writer, "Code Complete",
            "Steve McConnell", "155-615-484-4");
        WriteBook(writer, "Интернет програмиране с Java",
            "Светлин Наков", "954-775-305-3");
        WriteBook(writer, "Writing Solid Code",
            "Steve Maguire", "155-615-551-4");
        writer.WriteEndDocument();           (примерът продължава)
```


Работа с XmlWriter – пример

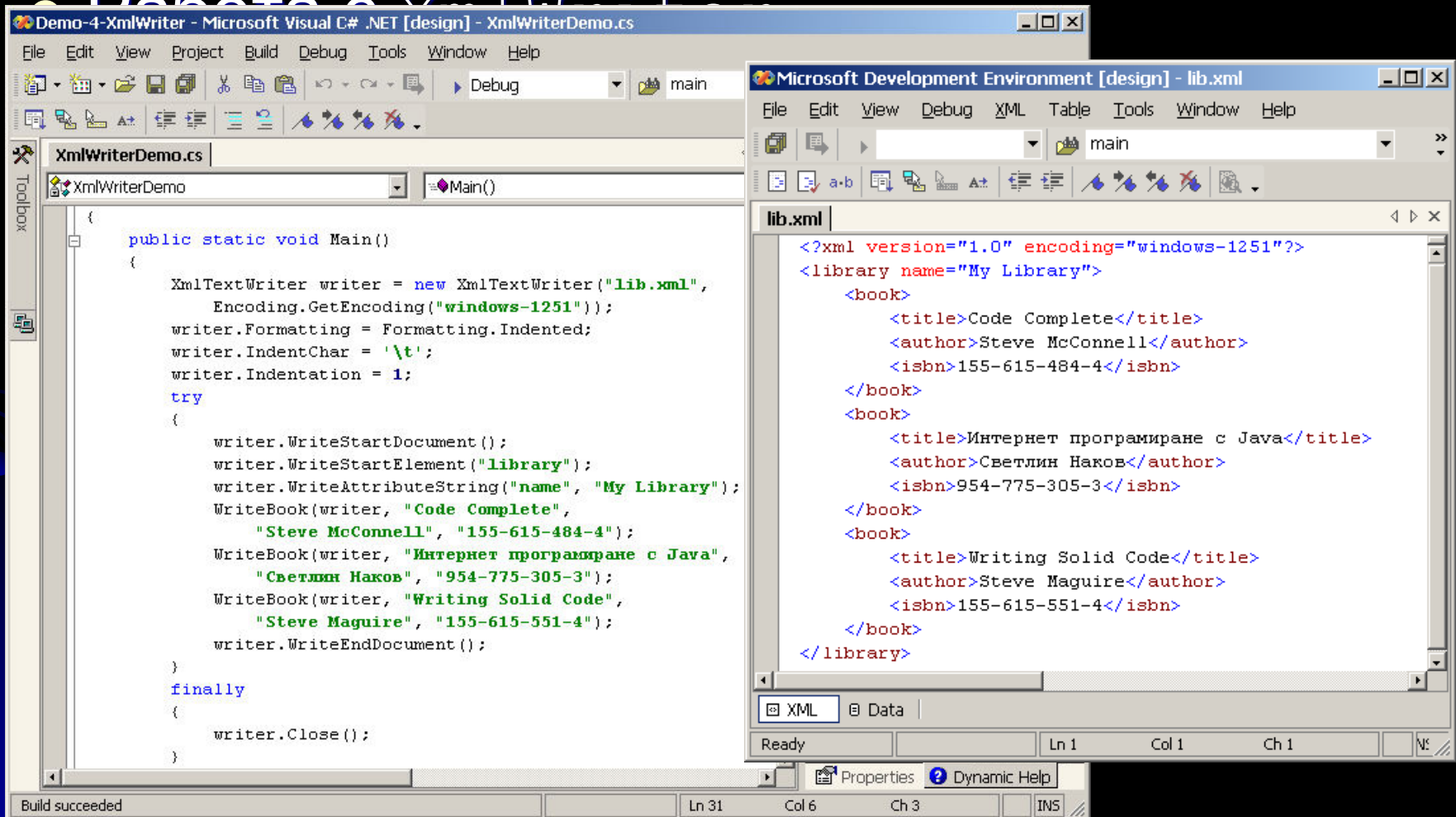
```
    }  
    finally  
    {  
        writer.Close();  
    }  
}  
  
private static void WriteBook(XmlWriter aWriter,  
    string aTitle, string aAuthor, string aIsbn)  
{  
    aWriter.WriteStartElement("book");  
    aWriter.WriteElementString("title", aTitle);  
    aWriter.WriteElementString("author", aAuthor);  
    aWriter.WriteElementString("isbn", aIsbn);  
    aWriter.WriteEndElement();  
}
```


Работа с XmlWriter – пример

- Резултатен XML документ:

```
<?xml version="1.0" encoding="windows-1251"?>
<library name="My Library">
  <book>
    <title>Code Complete</title>
    <author>Steve McConnell</author>
    <isbn>155-615-484-4</isbn>
  </book>
  <book>
    <title>Интернет програмиране с Java</title>
    <author>Светлин Наков</author>
    <isbn>954-775-305-3</isbn>
  </book>
  <book>
    <title>Writing Solid Code</title>
    <author>Steve Maguire</author>
    <isbn>155-615-551-4</isbn>
  </book>
</library>
```

Демонстрация #4



Валидация на XML по схема

- Схемите (XSD, DTD и XDR) описват правила и ограничения за съставяне на XML документи:
 - позволените тагове и начини на влагане
 - позволените атрибути и стойности
 - реда на таговете
- Валидацията на XML документ по дадена схема в .NET Framework се извършва с валидиращи парсери
 - Използва се класът `XmlValidatingReader`
 - Поддържа се XSD, DTD и XDR

Валидация на XML – пример

Имаме следния XML файл:

```
<?xml version="1.0"?>
<library xmlns="http://www.koko.com/schemas/library"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.koko.com/schemas/library
    http://www.koko.com/schemas/library.xsd"
  name=".NET Developer's Library">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

Валидация на XML – пример

В него е указана използваната XSD схема:

```
<?xml version="1.0"?>
<library xmlns="http://www.koko.com/schemas/library"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.koko.com/schemas/library
    http://www.koko.com/schemas/library.xsd"
  name=".NET Developer's Library">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-713-0376-1</isbn>
  </book>
</library>
```

URL адрес (в Интернет),
където е публикувана
XSD схемата

URI на
използваното
пространство
от имена

Валидация на XML – пример

● XSD схемата, по която ще валидираме:

```
<?xml version="1.0" encoding="UTF-8"?>  
  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"  
  xmlns="http://www.koko.com/schemas/library"  
  targetNamespace="http://www.koko.com/schemas/library">
```

```
  <xs:element name="lib" >
```

```
    <xs:complexType>
```

```
      <xs:sequence>
```

```
        <xs:element
```

```
          max
```

```
        </xs:sequence>
```

```
      <xs:attribute name="name" type="xs:string"  
        use="optional"/>
```

```
    </xs:complexType>
```

```
  </xs:element>
```

Указано е, че схемата описва
елементите от пространството
<http://www.koko.com/schemas/library>

(примерът продължава)

Валидация на XML – пример

- XSD схемата, по която ще валидираме:

```
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="title"/>
      <xs:element ref="author"/>
      <xs:element ref="isbn"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="title" type="xs:string"/>
<xs:element name="author" type="xs:string"/>
<xs:element name="isbn" type="xs:string"/>

</xs:schema>
```

Валидация на XML – пример

- Кодът, който валидира XML документа:

```
using System;
using System.Xml;
using System.Xml.Schema;
...
static void Main()
{
    XmlTextReader tr = new XmlTextReader("library.xml");
    XmlValidatingReader vr =
        new XmlValidatingReader(tr);
    vr.Schemas.Add("http://www.koko.com/schemas/library",
        "library.xsd");
    vr.ValidationType = ValidationType.Schema;
    vr.ValidationEventHandler +=
        new ValidationEventHandler (ValidationHandler);
    mValid = true;
```

(примерът продължава)

Валидация на XML – пример

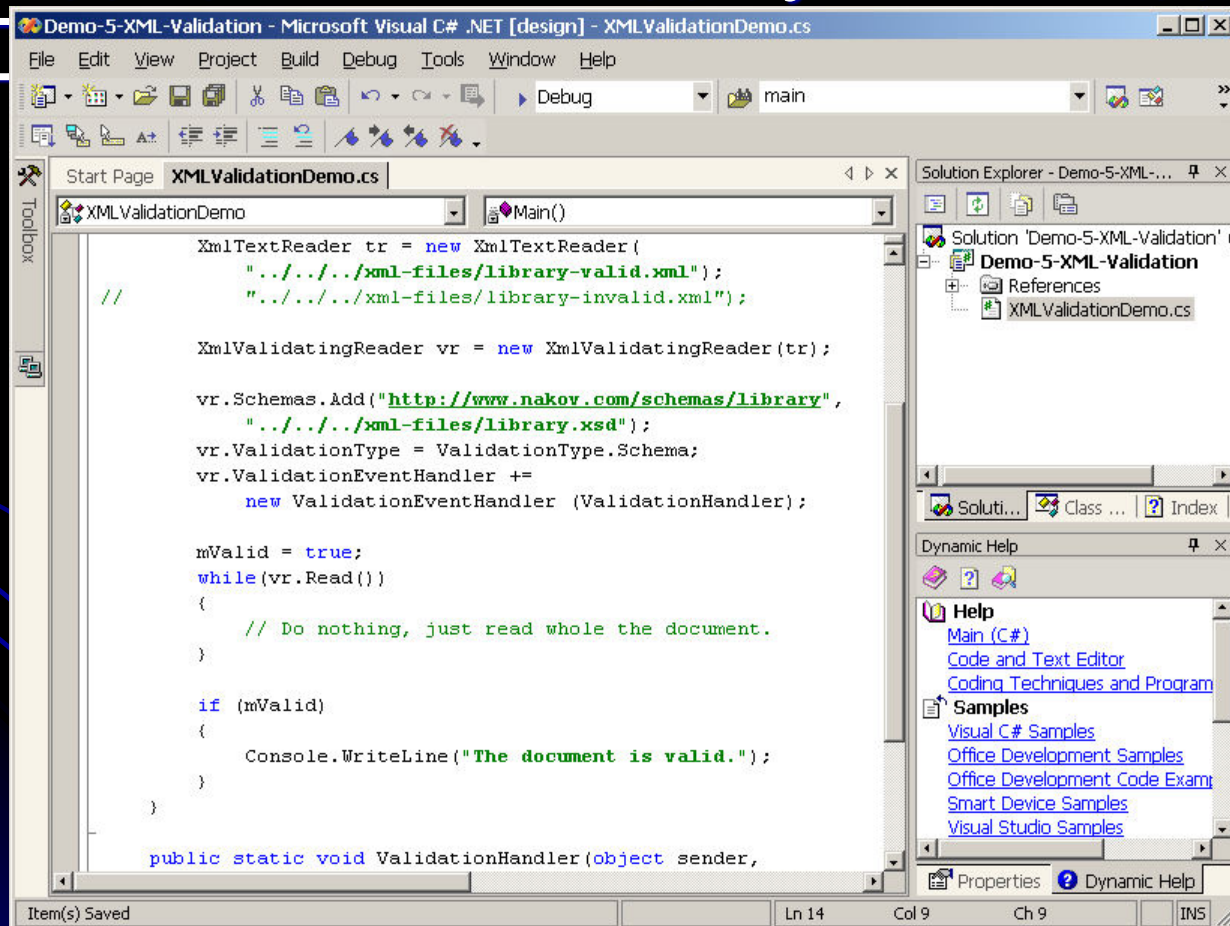
- Кодът, който валидира XML документа:

```
while (vr.Read())  
{  
    // Do nothing, just read the whole document.  
}  
  
if (mValid)  
{  
    Console.WriteLine("The document is valid.");  
}  
}
```

```
public static void ValidationHandler(  
    object aSender, ValidationEventArgs aArgs)  
{  
    mValid = false;  
    Console.WriteLine("***Validation error");  
    Console.WriteLine("\tSeverity:{0}", aArgs.Severity);  
    Console.WriteLine("\tMessage:{0}", aArgs.Message);  
}
```

Демонстрация #5

- Валидация на XML документ по дадена XSD



Още за XML валидацията

- Валидацията може да се извършва и по време

на зареждане на DOM дърво:

```
XmlDocument doc = new XmlDocument();  
XmlTextReader tr = new XmlTextReader("Sample.xml");  
XmlValidatingReader valReader =  
    new XmlValidatingReader(tr);  
valReader.ValidationType = ...;  
valReader.ValidationEventHandler += ...;  
doc.Load(valReader);
```

- При валидация по DTD схема от локален DTD файл трябва да се използва `XmlResolver` за извличане на външните XML ресурси
- Предварителното зареждане и кеширане на схемите в `XmlValidatingReader.Schemas` е

XPath

- XPath (XML Path Language) е език за адресиране на части от XML документи
- XPath изразите съдържат описание на пътища до възли и критерии, на които възлите трябва да отговарят
- XPath често пъти се използва заедно с XSLT и XPointer

- Примери за XPath изрази:

```
/library/book[isbn='1-930110-19-7']
```

```
/catalog/cd[@price<10.80]
```

```
/book/chapter[3]/para[last()]
```

XPath изразите

- / – адресира корена на документа
- /someNode – адресира всички възли с име "someNode", преки наследници на корена
- /books/book – адресира всички възли "book", наследници на възела "books"
- /books/book[price<"10"]/author – адресира всички автори (/books/book/author), чиито книги имат цена < "10"
- /items/item[@type="food"] – адресира всички възли с име item, които имат атрибут "type" със стойност "food" и са наследници на възел "items", закачен за корена на документа

XPath в .NET Framework

- XPath е много удобен за търсене на информация в XML документи
- Класовете и интерфейсите за работа с XPath се намират в пространството `System.Xml.XPath`
- XPath може да се ползва директно от класа `XmlNode` (и всички негови наследници) чрез методите:
 - `SelectNodes(string xpathQuery)` – връща списък от всички възли, които съответстват на зададения XPath израз
 - `SelectSingleNode(string xpathQuery)` – връща първия възел, който съответства на зададения XPath израз

XPath и XmlNode – пример

Даден е следния XML документ:

```
<?xml version="1.0" encoding="windows-1251"?>
<items>
  <item type="beer">
    <name>Загорка</name>
    <price>0.54</price>
  </item>
  <item type="food">
    <name>кебапчета</name>
    <price>0.48</price>
  </item>
  <item type="beer">
    <name>Каменица</name>
    <price>0.56</price>
  </item>
</items>
```


XPath и XmlNode – пример

- Търсим в него имената на всички стоки от тип “бира”:

```
<?xml version="1.0" encoding="windows-1251"?>
<items>
  <item type="beer">
    <name>Загорка</name>
    <price>0.54</price>
  </item>
  <item type="food">
    <name>кебапчета</name>
    <price>0.48</price>
  </item>
  <item type="beer">
    <name>Каменица</name>
    <price>0.56</price>
  </item>
</items>
```

Можем да ползваме XPath израза
/items/item[@type='beer']/name

XPath и XmlNode – пример

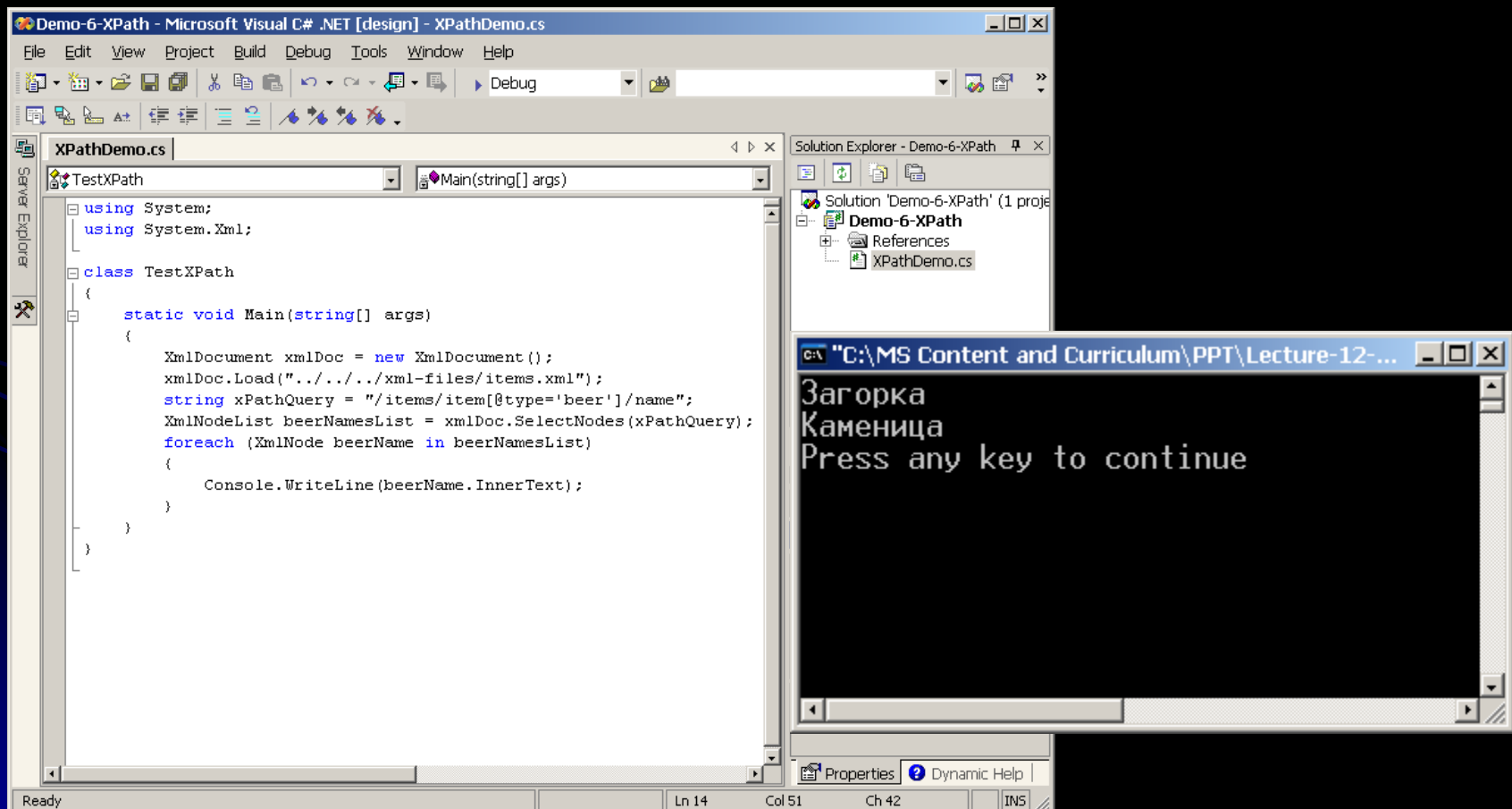
- Можем да използваме следния код:

```
using System;
using System.Xml;

class TestXPath
{
    static void Main(string[] args)
    {
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.Load("items.xml");
        string query = "/items/item[@type='beer']/name";
        XmlNodeList beerNamesList =
            xmlDoc.SelectNodes(query);
        foreach (XmlNode beerName in beerNamesList)
        {
            Console.WriteLine(beerName.InnerText);
        }
    }
}
```

Демонстрация #6

- Търсене с XPath в XML документи чрез



Пространството `System.Xml.XPath`

- `XPathNavigator` – позволява обхождане на дървовидно-организирани данни и изпълнение на `XPath` заявки върху тях
- `XPathNodeIterator` – позволява навигация по резултата от `XPath` заявки получени от `XPathNavigator.Select(...)`
- `IXPathNavigable` – интерфейс, с който получаваме `XPathNavigator`. Поддържа се от класовете `XmlNode` и `XPathDocument`
- `XPathDocument` – осигурява високоскоростна еднопосочна работа с `XPath` и `XSLT` в режим само за четене (за обемни XML документи)
- `XPathExpression` – съдържа компилиран `XPath` израз (по-добра производителност)

XPathNavigator – пример

- Промоция – намаляване на цената на

бирата с 20%

```
using System;
using System.Xml;
using System.Xml.XPath;
using System.Globalization;

class XPathNavigatorDemo
{
    public const decimal DISCOUNT = (decimal) 0.20;

    static void Main()
    {
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.Load("items.xml");
        CultureInfo numberFormat =
            new CultureInfo("en-US");
```

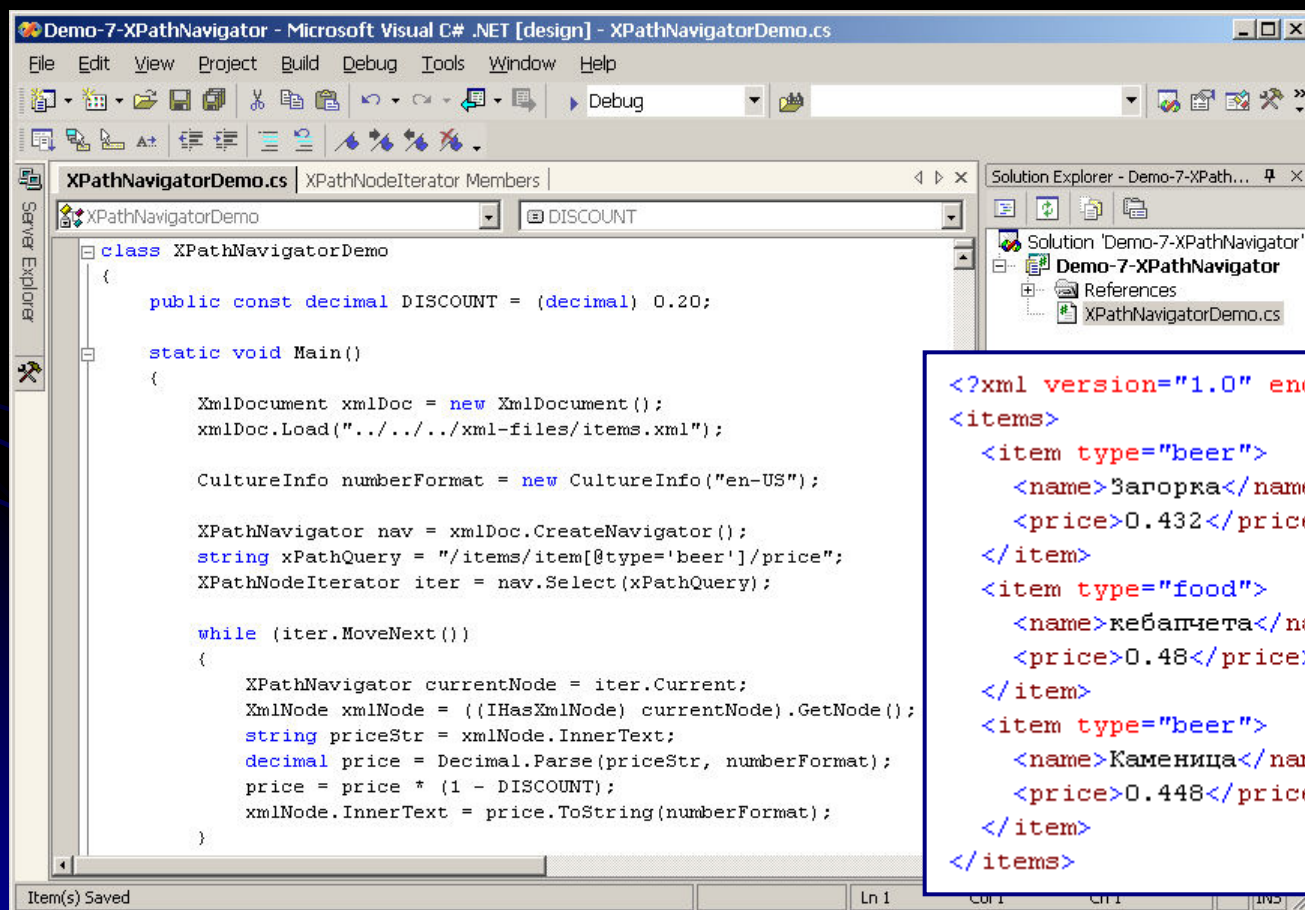
(примерът продължава)

XPathNavigator – пример

```
XPathNavigator nav = xmlDoc.CreateNavigator();
string xpathQuery =
    "/items/item[@type='beer']/price";
XPathNodeIterator iter = nav.Select(xpathQuery);
while (iter.MoveNext())
{
    XPathNavigator currentNode = iter.Current;
    XmlNode xmlNode =
        ((IHasXmlNode) currentNode).GetNode();
    string priceStr = xmlNode.InnerText;
    decimal price =
        Decimal.Parse(priceStr, numberFormat);
    price = price * (1 - DISCOUNT);
    xmlNode.InnerText =
        price.ToString(numberFormat);
}
xmlDoc.Save("itemsNew.xml");
}
```

Демонстрация #7

- Обработка на XML документ с



```
class XPathNavigatorDemo
{
    public const decimal DISCOUNT = (decimal) 0.20;

    static void Main()
    {
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.Load("../..../xml-files/items.xml");

        CultureInfo numberFormat = new CultureInfo("en-US");

        XPathNavigator nav = xmlDoc.CreateNavigator();
        string xpathQuery = "/items/item[@type='beer']/price";
        XPathNodeIterator iter = nav.Select(xpathQuery);

        while (iter.MoveNext())
        {
            XPathNavigator currentNode = iter.Current;
            XmlNode xmlNode = ((IHasXmlNode) currentNode).GetNode();
            string priceStr = xmlNode.InnerText;
            decimal price = Decimal.Parse(priceStr, numberFormat);
            price = price * (1 - DISCOUNT);
            xmlNode.InnerText = price.ToString(numberFormat);
        }
    }
}
```

```
<?xml version="1.0" encoding="windows-1251"?>
<items>
  <item type="beer">
    <name>Загорка</name>
    <price>0.432</price>
  </item>
  <item type="food">
    <name>кебапчета</name>
    <price>0.48</price>
  </item>
  <item type="beer">
    <name>Каменица</name>
    <price>0.448</price>
  </item>
</items>
```

XPathDocument – пример

```
using System;
using System.Xml;
using System.Xml.XPath;

class XPathNavigatorDemo
{
    static void Main()
    {
        // Create fast forward-only XPath processor
        XPathDocument doc = new XPathDocument(
            "../../../xml-files/items.xml");
        XPathNavigator nav = doc.CreateNavigator();

        XPathExpression expr = nav.Compile(
            "/items/item[@type='beer']/name");
        XPathNodeIterator iter = nav.Select(expr);
        while (iter.MoveNext())
        {
            XPathNavigator currentNode = iter.Current;
            Console.WriteLine(currentNode.Value);
        }
    }
}
```


XSL трансформации

- XSL трансформациите (XSLT) позволяват преобразуване на един XML документ в друг XML документ с различна структура
- XSLT разчита на XPath за извличане на части от входния документ, които се използват в изходния документ
- В частност XSLT може да се използва и за преобразуване на XML документи в XHTML

```
<?xml version="1.0" encoding="windows-1251"?>
```

```
<xsl:stylesheet version="1.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:template match="/">
```

(примерът продължава)

XSL трансформации

```
<html>
<body>
<h1>Моята библиотека</h1>
<table bgcolor="#E0E0E0" cellspacing="1">
  <tr bgcolor="#EEEEEE">
    <td><b>Заглавие</b></td>
    <td><b>Автор</b></td>
  </tr>
  <xsl:for-each select="/library/book">
    <tr bgcolor="white">
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="author"/></td>
    </tr>
  </xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Езикът XSL – конструкции

- `<xsl:template match="XPath expr."> ...`
`</xsl:template>` – замества зададената с XPath израз част от документа с тялото на конструкцията
- `<xsl:for-each select="XPath expr."> ...`
`</xsl:for-each>` – замества всеки възел, отговарящ на дадения XPath израз с тялото на конструкцията
- `<xsl:value-of select="XPath expr."/>` – извлича стойността на зададения XPath израз (само първото намерено съответствие)
- `<xsl:sort select="XPath expr."/>` – в `<xsl:for-each ... >` конструкции сортира по стойността на даден XPath израз
- XSL шаблоните са валидни XML документи!

XSLT и .NET Framework

- В .NET Framework можем да извършваме XSL трансформации чрез класа `System.Xml.Xsl.XslTransform`
- Класът `XslTransform` поддържа методите:
 - `Load(...)` – зарежда XSL шаблон, по който ще се извършва трансформация
 - `Transform(...)` – извършва трансформация на даден XML документ
 - Приема като вход име на XML файл, `XPathNavigator` или `IXPathNavigable`
 - Записва изхода в XML файл, поток или `XmlWriter`

XSL трансформации – пример

- Преобразуване на XML документ по

даден XSL шаблон:

```
using System;
```

```
using System.Xml.Xsl;
```

```
class XSLTransformDemo
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        XsltTransform xslt = new XsltTransform();
```

```
        xslt.Load("library-xml2html.xsl");
```

```
        xslt.Transform("library.xml",  
            "library.html", null);
```

```
    }
```

```
}
```

Демонстрация #8

- Преобразуване на XML документ по

The screenshot illustrates the XSLT transformation process in a Visual Studio .NET environment. The main window, titled "Demo-8-XSL-Transform - Microsoft Visual C# .NET [design] - XSLTransformDemo.cs", shows the following code:

```
using System;
using System.Xml.Xsl;

class XSLTransformDemo
{
    static void Main()
    {
        XslTransform xslt = new XslTransform();
        xslt.Load("../xml-files/lib.xslt");
        xslt.Transform("../xml-files/lib.xml", null, null);
    }
}
```

The Solution Explorer on the right shows the project structure, including the XSLTransformDemo.cs file.

A smaller window displays the resulting HTML output, which is a table titled "Моята библиотека" (My Library). The table contains the following data:

Заглавие	Автор
Programming Microsoft .NET	Jeff Prosise
Microsoft .NET for Programmers	Fergal Grimes

The bottom window shows the original XML file, which is transformed into the HTML output shown in the middle window.

```
<?xml version="1.0"?>
<library name=".NET Developer's Library">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

Clipboard & Drag & Drop в .NET

1. работа с Clipboard

-клипборда винаги поддържа само един даннов обект и обикновено, но вече незадължително в 1 формат. Напр. ралично форматиранни данни в колони на електр. таблица

-базов е класът от пространството: **System.Windows.Forms Clipboard**. Той е sealed, няма public конструктори → не може да се наследява и няма инстанции. Има 2 статични метода:

```
void SetDataObject(...);  
IDataObject GetData(..);
```

-ако поставяме в клипборд данни само в 1 формат, всичко е елементарно:

```
Clipboard.SetDataObject( str, true);  
Clipboard.SetDataObject( bitmap, true);
```

второто повикване подменя данните
true – данните са достъпни и след
приключване на програмата

но може и:

```
Button btn = new Button();  
btn.Text = "OK";  
Clipboard.SatDataObject(btn, false);
```

можем да поставяме и всякакъв
обект.
вторият арг. е false, защото обект
не може да се предхвърля м/ду
програми

1. извличат се не данните, а инстанция на класа, който съдържа информация за тях:

IDataObject data = Clipboard.GetDataObject();

2. **`GetDataObject()`** дефинира 4 базови (12 с предефинициите) метода:

bool GetDataPresent(Type type)

object GetData(Type type)

пример:

проверка за наличност:

dataobject.GetDataPresent(typeof(String));

//върща true ако има низ в clipboard

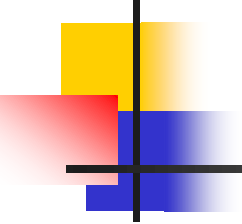
dataobject.GetDataPresent(typeof(Bitmap));

dataobject.GetDataPresent(typeof(Button));

и извличане на данни:

string str = (string) dataobject.GetData(typeof(string));

Bitmap bitmap = (Bitmap) dataobject.GetData(typeof(Bitmap));



Формати на данни, прехвърляни през clipboard

-всякакви обекти и типове, съответстващи на ползвани в Windows Forms

Но какво става с Windows приложения (и съответни данни), работещи с Win32 API или MFC ?

*А проблемите с **текст**:*

*може да е в **Unicode** или не (ANSI или OEM – за MS DOS и конзоли) ?*

*или какво става с **форматиран текст**? Ако е създаден от WORD и се вмъква в Notepad или друга неразбираща форматните спецификации? – може би трябва подразбиращо се преобразуване в клипборда към RTF или HTML ?*

*или какво става с копиран в клипборд **текст от browser**? HTML кодировките следва да се запазят ако се вмъква пак в browser, но не и към приложения, неразбиращи HTML тагове.*

*каково става с '**данни извлечени от файл с разделители**' (т.нар. CSV формат -Comma Separated Value) ?*

В КЛИПБОРДА СЛЕДВА ДА СЕ ЗАДАВА И ИНФОРМАЦИЯ ЗА ТИПА ТЕКСТ И ИЗОБЩО ИДЕНТИФИКАТОРИ ЗА ФОРМАТИТЕ НА ДАННИТЕ



Формати на данни, прехвърляни през clipboard

-Полезен е методът **GetFormats()** на интерфейса **IDataObject**:

```
string[] asstr = dataobject.GetFormats();           // връща ID за всички
                                                    // формати, достъпни в clipboard'а
```

```
string str = (string) dataobject.GetData(typeof(string))
string str = (string) dataobject.GetData("System.String")
string str = (string) dataobject.GetData("UnicodeText")
```

} връщат низ от
клипборда, с възможно
вътрешно преобразуване

-в класа **DataFormats** има 21 статични полета за четене, връщащи форматите, които се поддържат за низове в .NET. По-интересни са:

"Text" "OEMText"

за 8 битови кодировки

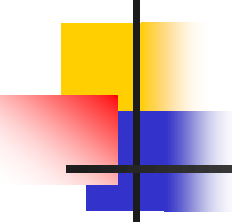
"Bitmap" "DeviceIndependentBitmap" "Palette" "TaggedImageFileFormat"

за картинки и съпътстващи палитри

за аудио файлове

за списък от имена на файлове (напр. селектиран файл от Explorer се копира в clipboard

за HTML, CSV, RTF - които се извличат като низ, но са предвидени
методи за подходящото им преобразуване до HTML,...



Clipboard и повече от 1 формати в него едновременно

1. как да зададем формата на данните в клипборда?

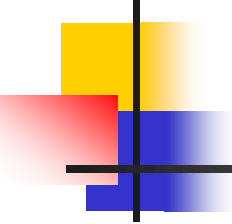
```
Clipboard.SetDataObject(strText);  
Clipboard.SetDataObject(new DataObject(strText));  
Clipboard.SetDataObject(new DataObject(DataFormats.Html, strText));
```

еквивалентни

2. как да зададем множество формати в клипборд (SetData())?

```
DataObject data = new DataObject();  
data.SetData(strText);  
data.SetData(DataFormats.Html, strHtml);  
data.SetData(DataFormats.Rtf, strRtf);  
Clipboard.SetDataObject(data, True);
```

// пъхна в клипборда с конверсия
// до съвместими формати



За механизмите : Drag & Drop

1. възможни действия:

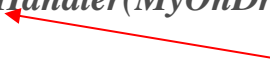
състояние на клавиши по време на операцията	действие
-	Move
Ctrl	Copy
Shift + Ctrl	Link (източник и цел получават препратки към общ обект)

2. за разрешаване на операция drop в приложение (контрол или форма) следва наследеното от Control свойство **AllowDrop** на обекта да е **true**

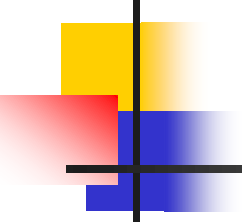
3. 4-те събития, достъпни в целта(DargEnter, DragOver, DragDrop, DragLeave) са свързани с методи On... и могат да се реализират като делегати, напр:

```
txtbutton.AllowDrop = true;
```

```
txtbutton.DragOver +=new DragEventHandler(MyOnDragOver);
```



делегатния тип



За механизмите: Drag & Drop

4. Към всички методи, свързани със събития в целта се подава аргумент **DragEventArgs**, съдържащ информация за:

- клавиатура и мишка по време на събитието;
 - обект **DataObject**, имплементиращ **IDataObject** (от него могат да се ползват методите за извличане на данните);
 - какви операции са разрешени над предавания обект (зададено от изт.);
 - свойство, което приемникът установява след като анализира дали може и иска да приеме данните, както и каква операция се извършва (copy, move..).
- Това променя вида на курсора и информира източника (свойството е достъпно при него, защото **DoDragDrop()** привършва за източника едва когато обекта е в целта или Esc)
-

A. Едно приложение става източник като наследява **Control** и инициира операция Drag с повикване на

DragDropEffects DoDragDrop(object dataobject, DragDropEffects dde);

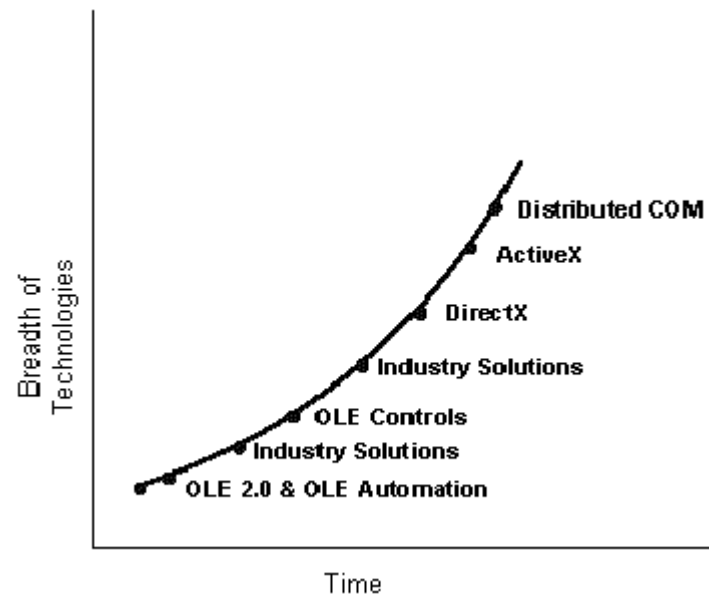


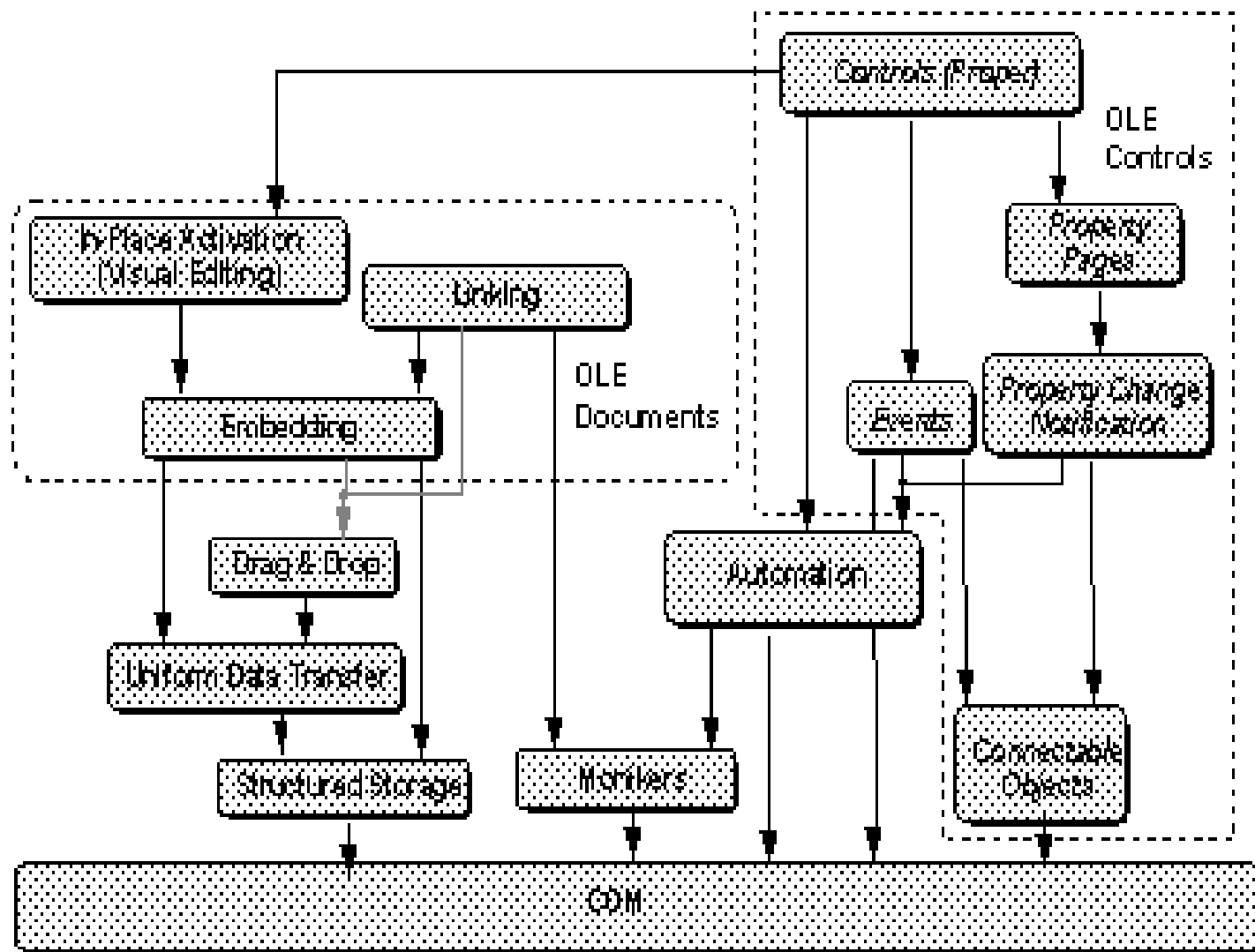
данновия обект



позволените операции

COM технология за изграждане на съставни обекти

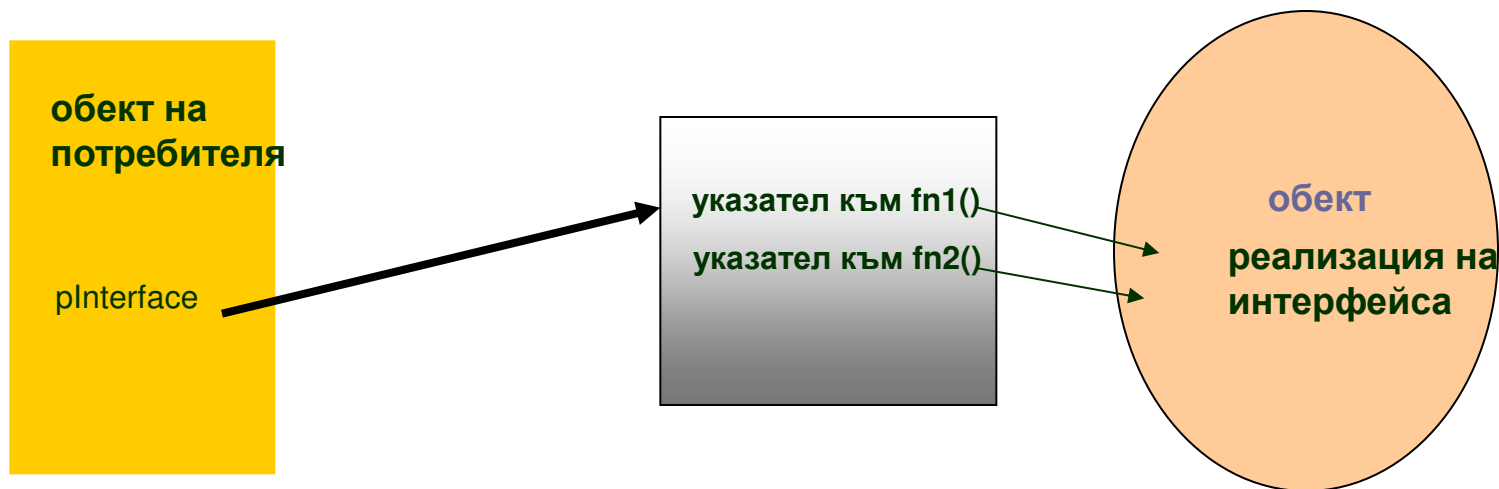




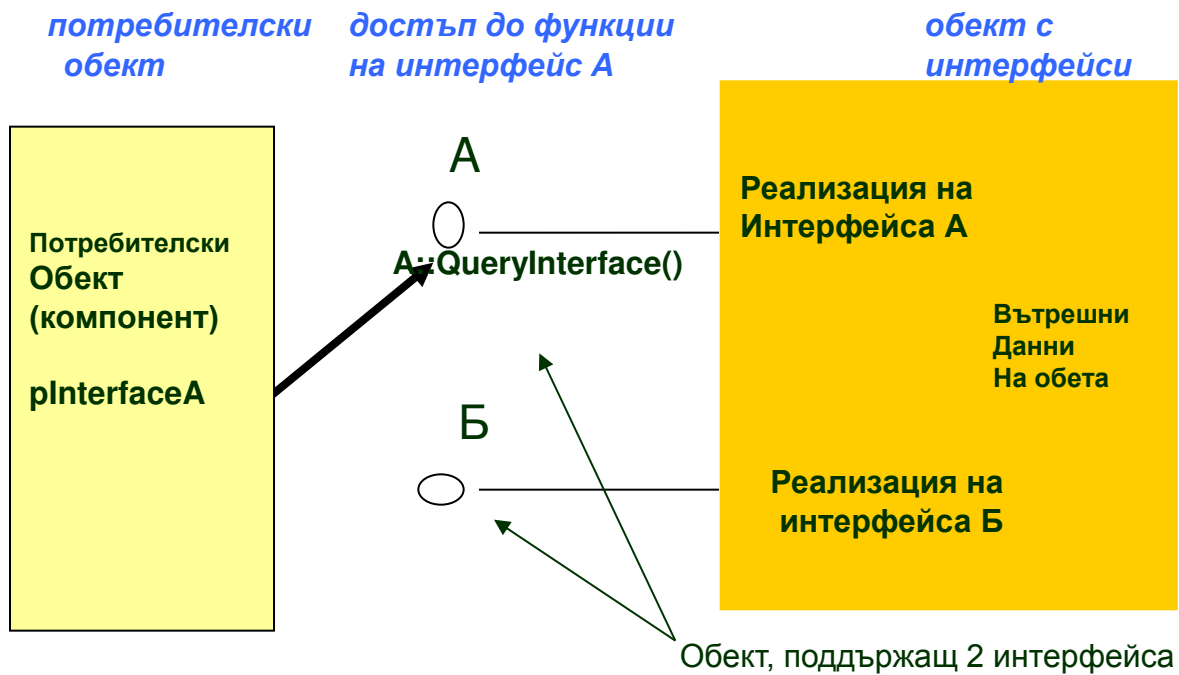
Обекти и интерфейси

Интерфейсът е набор от семантично свързани функции, реализирани в определен обект. Това включва прототип или сигнатура.

Използването (instantiation) става, когато интерфейсът е реализиран и е изработен указател към масив от указатели към функциите му:



обекти и интерфейси



Намираме се в 1 компонент

Базов интерфейс

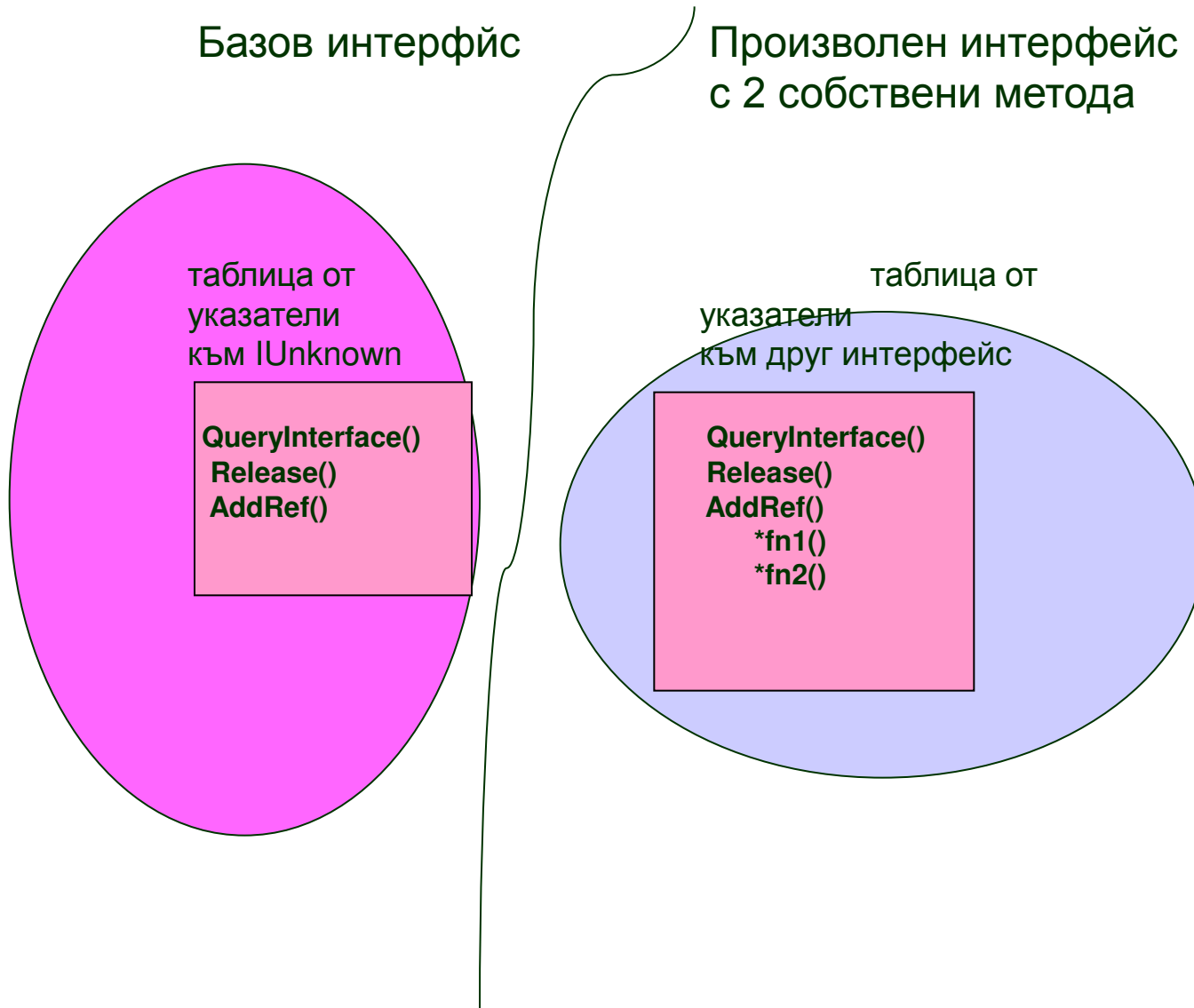
таблица от
указатели
към IUnknown

QueryInterface()
Release()
AddRef()

Произволен интерфейс
с 2 собствени метода

таблица от
указатели
към друг интерфейс

QueryInterface()
Release()
AddRef()
 *fn1()
 *fn2()



Uniform Data Transfer (UDT)& Notification подсистеми

OLE има 2 въведения:

- **не ограничава формата** до този на clipboard, а добавя указания за типа данни, устройство на обмена и препоръчителна среда на обмен.
- **разделят се протокола от обмена на данните.**

Данновият обект е COM обект и поддържа интерфейс: **IDataObject** с два съществени метода:

SetData се използва за поставяне на данните в данновия обект и **GetData** – за извличане на данните.

Самият даннов обект с поставени в него данни стои в clipboard

поставянето става с

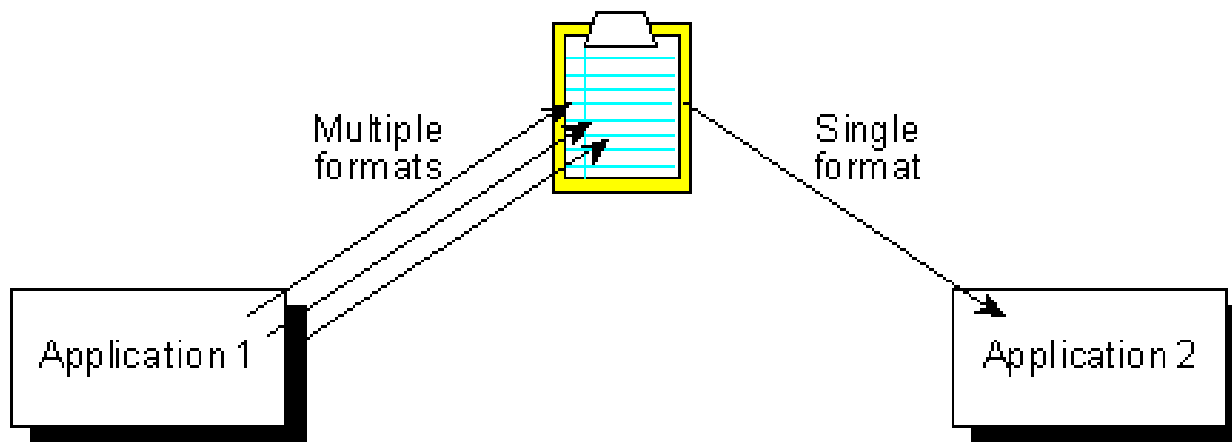
`::OleSetClipboard()`

а консуматорът на данните получава указател към IDataObject с

`::OleGetClipboard.`

OLE Clipboard и данновия обект (DO)

1. OLE clipboard е COM базиран. Работи се с методи, част от COM интерфейси, а не с API функции.
2. Данните се прехвърлят не единствено през глобалната памет, както в класическия едноименен механизъм.





Данновият обект и интерфейса IDataObject работят основно с 2 структури:

FORMATETC:

описва обекта, който ще се трансферира –
*обобщения Clipboard формат на данните,
вида и начина на изобразяване на данни,
възможни места за съхранение на данните,
тип и х-ки на средата към която е възможен обмен.
данни за устройството, което съхранява обекта,
възможните начини за предаване на данни
(компресия, като икона, в пълен образ),
какво става с образа ако е извън изобразяваното поле и др.*

Тази структура, след инициализация в източника се подава в данновия обект с COleDataSource::CacheData().

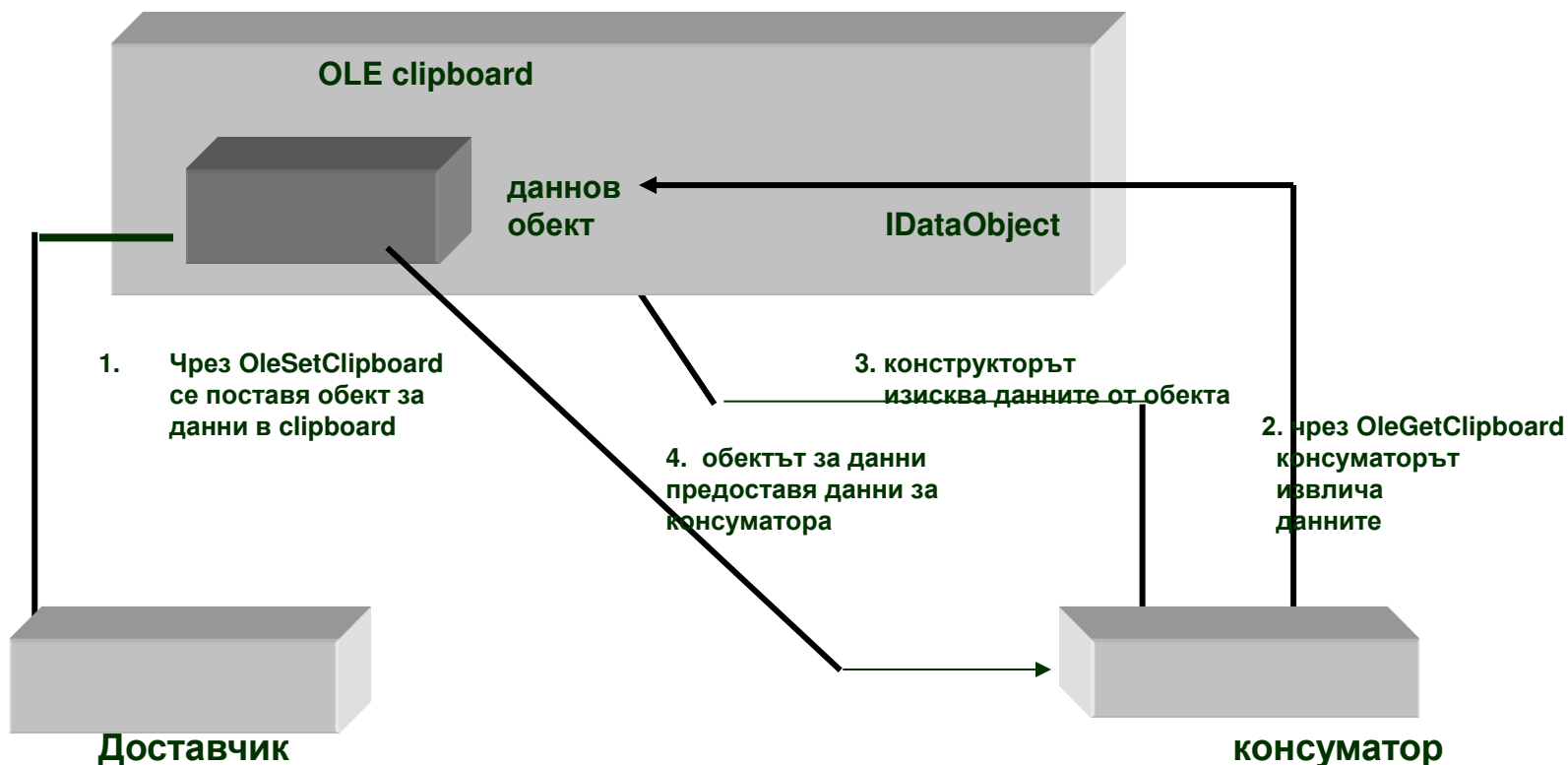
STGMEDIUM

Описва текущото местоположение на данните за трансфер.
Например, при глобална памет, тук се поддържа HGLOBAL,
при файл – низ с името на файла и т.н.

Всеки Windows прозорец може да стане източник на данни за OLE clipboard.

За целта е достатъчно в описващия го клас да се инстанциира обект от тип **COleDataSource** (той поддържа прехвърляните данни реализира интерфейса **IDataObject**).

Всъщност в клипборда са не самите данни, а COM обект, капсулиращ ги (**IDataObject**). Следва обобщена схема на процеса:



Схемата по-горе, но представена като последователност от обръщания към функции изглежда така:

- | | |
|---------------------------|---|
| 1. IDataObject::SetData() | // данните са в обекта DataObject. |
| 2. ::OleSetClipboard() | //данновият обект е в clipboard. |
| 3. ::OleGetClipboard() | //конс. взима указател към IDataObject. |
| 4. IDataObject::GetData() | //извличат се данните. |
-

MFC поддържа OLE clipboard с помощта на два мощни класа:
COleDataSource (в източника) и COleDataObject (в консуматора).

Ето примерна последователност от действия, които следва да се реализират в консуматора при извличане на дановия обект и същинските данни:

1. Създава се COleDataObject обект;
2. С:
COleDataObject::AttachClipboard()
се свързва нашия COleDataObject с OLE clipboard.

Оттук нататък всички повиквания за данни и методи на COleDataObject се трансферират към методи на IDataObject (указател за него е получен с ::OleGetClipboard()), чиято обвивка е AttachClipboard();)

3. С повиквания от вид: COleDataObject::GetGlobalData() се извличат данни.
4. Освобождава се блока заета глобална памет.

Действия, извършвани в източника:

1. създава се обект в heap (не в стек) от тип : COleDataSource
 2. COleDataSource::CacheGlobalData() //данните се кешират в обекта
 3. COleDataSource::SetClipboard() // обекта се поставя в клипборда
-

друга, полезна абстракция с OLE clipboard:

COleDataObject::GetFileData() асоциира клипборд с файл.

Това ни позволява от този момент да работим с функции на CFile (например метода: GetFileData(), който връща указател към CFile).

Тази абстракция работи за обекти в глобалната памет, файлове, както и с обекти от тип CStream.

Drag&Drop механизмите и данновия обект

- Работи както с цял документ, така и с части от него;
- Прехвърля произволен тип данни.

1. *източникът на операция-drag подава data object (указател към IDataObject на **::DoDragDrop()**), определя началото и края на операцията и управлява интерфейса (мишката).*
2. *Всеки прозорец, който може да бъде целеви , следва да се регистрира в системата (**::RegisterDragDrop()**).*
3. *Приемникът на операцията-drop получава data object , определя използвания в него формат и решава какво следва да се направи при drop в целевия прозорец*

Drag&Drop интерфейс

Действие

интерфейс IDropSource	имплементиран от обекта – източник на данни. Има 2 полезни метода: за обновяване на курсора и за промени в хода на операцията, вследствие клавиш или мишка. Те се викат директно от COM средата.
интерфейс IDropTarget	имплементиран в обекта – приемник на данните (цел). Има 4 полезни метода: 3 свързани със събития на курсора, когато той е в целта и 1 се вика при пускане – Drop()
Интерфейс IDataObject	В източника

Операции в източника

1. детектира се drag
2. източникът подава с повикването на `::DoDragDrop()`:
IDataObject, IDropSource, стойност показваща типа позволени операции над данните (напр. местене, копиране), адрес за попълване от `DoDragDrop()` как е привършила операцията. Така източникът знае какво се случва в другия край и напр. при успешен drop може да изтрие данните от документа-източник
3. `DoDragDrop()` държи управлението, до
 1. drop в целта;
 2. прекратяване (Esc).

През това време COM подсистемата (към ОС) вика методи на IDropSource (напр. за обновяване вида на курсор при някакви условия, или източникът да прекрати операцията при други..)

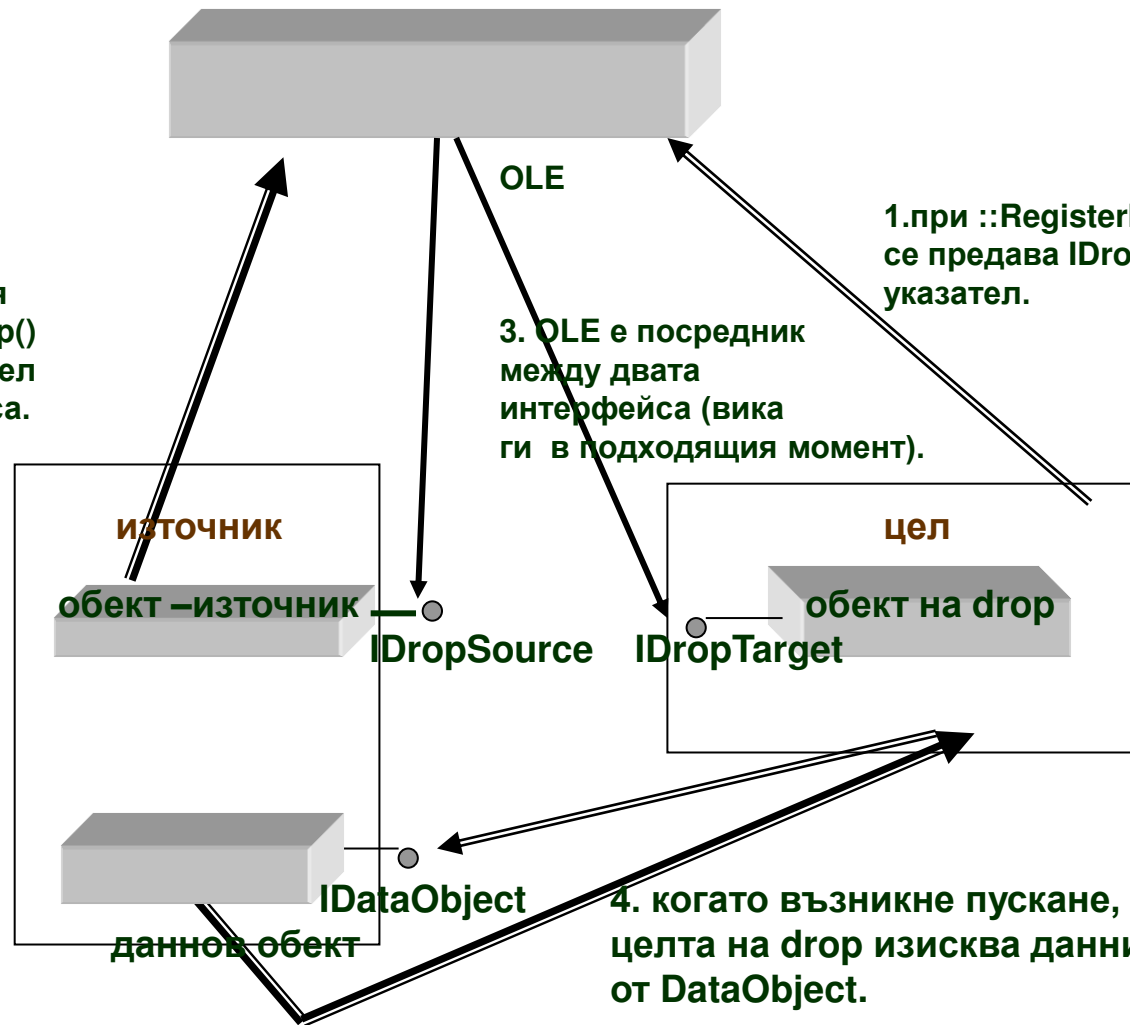
Операции в drop целта

1. `::RegisterDragDrop(hwnd, указател към IDropTarget);`
2. когато курсор попадне в целта, системата вика `IDropTarget` методи (вкл. и `Drop()` – описващ какво се случва при пускане, с подаден `IDataObject` и * към място за съобщения към източника). Те се пишат от програмиста!!!
3. Попълва се параметър, връщан към източника
4. извличат се данните (напр. `pDataObject --> GetData()`)

методи на COleDropTarget

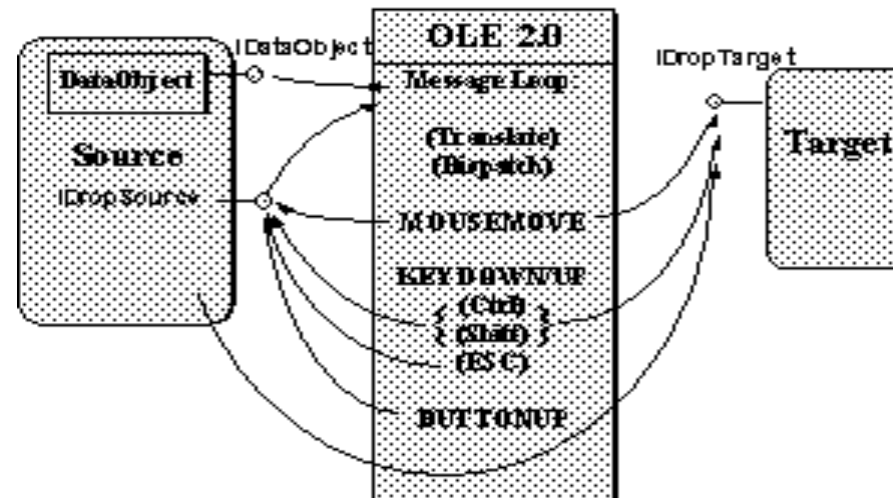
Методи за припокриване	цел
OnDragEnter	Вика се когато курсор навлезе в прозореца –цел.
OnDragLeave	Определя поведение, когато drag операцията напусне прозореца без пускане.
OnDragOver	Курсорът се влечи в рамките на прозорец.
OnDrop	Тук следва да се поемат за обработване данните – предмет на операцията (това е в прозореца – цел).
OnScrollBy	Определя поведение в случаите, когато в целевия прозорец се налага скролиране.

2. при започване на drag операция към ::DoDragDrop() се подава указател към 2 интерфейса.



5. DataObject подава данни към целта на Drop().

DoDragDrop функцията изпълнява цикъл, следящ събитията от мишка или клавиатура и реализиращ повиквания към методи на IDropSource или IDropTarget :



MFC и OLE drag & drop

MFC имплементира класове за всички COM интерфейси към обекти, необходими за целта:

COleDataSource – обект за данни;
COleDropSource – обект източник;
COleDropTarget – обект цел (негова инстанция се поставя в изгледа на приложението – цел).

Примерна извадка от код:

*// в CMyView например, добавяте променлива (говорим за приложение- drop цел.
//Тази даннова променлива е от тип COleDropTarget или производен.*

COleDropTarget m_oledroptarget;

// вика се методът Register() за създадения тип

m_oledroptarget.Register(this);

*// следва предефиниране на методите на IDropTarget,
//особено на OnDrop().*

....

какво следва да се извърши в източника:

HANDLE hdata = ::GlobalAlloc(....., указател към данните);

COleDataSource ods;

ods::CacheGlobalData(CF_TEXT, hdata);

DROPEFFECT de = ods.DoDragDrop(.....);

/ методът DoDragDrop() освен че обвива едноименната функция на OLE, поставя обекта в clipboard и подава указател към IDropSource на ::DoDragDrop(). */*

Съставни документи. Идея за процеса Object linking и обекта Moniker

Документите могат да съдържат части, създавани или обработвани от други.

Тогава говорим за съставни док. В този случай те се явяват източник на данни и трябва да съдържат информация за връзка с обработващото прил (. linking).

Данните за връзка се поддържат в отделен файл.

Така и що се касае до взаимното местоположение на обработващите приложения.

За решаване на този проблем се въвежда обект – moniker. Той съдържа информация как да се осъществи свързването и в какво обкръжение.

Moniker е също COM обект, капсулиращ както името на данновия файл, така и информация, дефинираща обработките над данни с това име. Тази информация е достъпна чрез интерфейс наричан IMoniker.

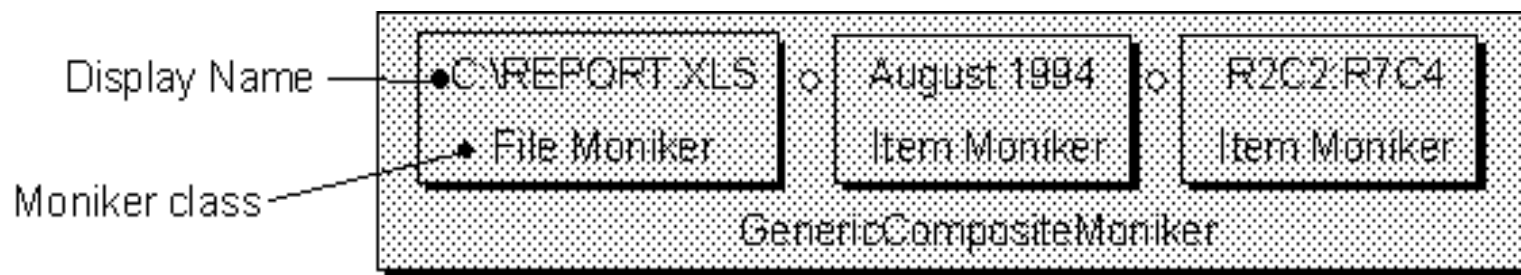
Потребителите използват обекта moniker, когато искат да работят с това име.

В интерфейса IMoniker се дефинират операциите, които могат да се изпълнят при свързване на обект с данни, носещи даденото име.

1. **File moniker** съдържа пътеката и името на файла с данни (абсолютна и относителна пътека). Когато свързан обект се активира, данните му се взимат от файл по указания абсолютен път, активира продукта с който е бил създаден (по разширението в името), получава обкръжението в което се зарежда за обработка файла (от item moniker) и зарежда файла. Ако не успее опитва с относителния път. Код за свързване върши всичко това.

2. **Item moniker** за формиране на обкръжение за обработка. Съдържа име на обект с данни за обкръжението и код за свързване , който изисква приложението-сървър на тези данни да върне указател към обекта Item moniker. Съставен moniker включва верига от горните съставки, както и код за свързване, който прохожда последователно всички monikers в списъка и ги активира за връзка със своите обекти и обработващи приложения.

Показана е структура на съставен moniker:



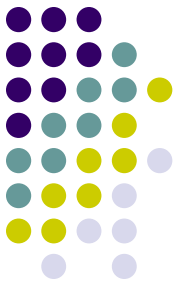
Windows - обекти (WO) и интерфейси

Интерфейсите се идентифицират по номер – 128 битово число (IID –интерфейсен идентификатор). **Uuidgen** – от команден ред и **Guidgen** от средата. IID се генерира и с API функция **CoCreateGuid()**.

как да изработим първи указател към наличен COM object?

1. Чрез API функции, **създаващи обекти от определен тип** - те връщат указател към обекта или негов интерфейс.
2. Чрез API функции, **създаващи обекти, базирани на указан клас-идентификатор**, Те също връщат указател.
3. Чрез **методи на интерфейс**, връщащи указател към друг интерфейс.
4. Чрез реализирани интерфейсни функции, които изискват другите обекти при обръщение да подават указатели към свои интерфейс





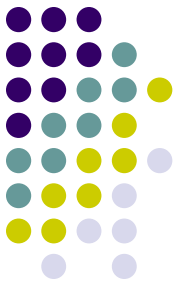
Разликите м/ду WO и C++ обектите са в:

- **декларирането на класа:** WO се дефинира чрез интерфейсите, които поддържа
- **инстанцирането на обекта:** class factory (с действие, подобно на new). Обектът class factory се изработва чрез специални OLE функции и поддържа интерфейс IClassFactory. В този интерфейс има функция CreateInstance() на която се подава идентификатора на интерфейса от който се нуждаем.

IClassFactory::CreateInstance() == new

Пример:

```
IMyInt* pMy;  
CoCreateInstance(CLSID_MyObject, NULL, ... , IID_IMyInt, (void**) &pMy);  
// създава обект чрез class factory и изработва IID чрез метод на  
IClassFactory  
  
...  
pMy->func (...); //вика се функцията на интерфейса.
```



- обръщението към обект:

WO са достъпни чрез указател, но единствено към интерфейс.
Обръщението към интерфейсни методи е тривиално:

`pISomeInterface -> MemberFunction([списък _параметри])`

- Работа с интерфейси и унищожаване на COM обект – методи `AddRef()` и `Release()`.

```
ULONG __stdcall CComClass::AddRef()           //обикновено обектът, а не
                                              // клиентът вика AddRef(), когато
                                              // подава указател към интерфейс
                                              // Клиент вика тази функция само
                                              // ако копира указателя в друг.

{ return ++m_IRef; }
```

```
ULONG __stdcall CComClass::Release()          // Release се вика от клиента

{
    if( __m_IRef == 0)
    {
        delete this;
        return 0;
    }
    return m_IRef;
}
```



брояч на интерфейсни обръщания:

1. Всяко създаване на нов интерфейс се съпътства с обръщение към `AddRef()`
 2. Унищожаване на интерфейсен указател се съпътства с `Release()`.
- И двете функции се отнасят за целия WO, а не за определен интерфейс.

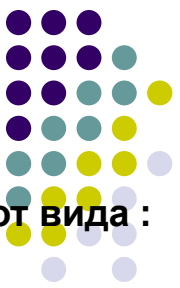
Пример:

```
LPSOMEINTERFACE pISome1;  
LPSOMEINTRFACE pISome2;  
CreateSomeObject( &pISome1);  
.....  
LPSOMEINTEFACE pCopy;  
pCopy = pISome1;  
pCopy->AddRef();           //броячът става 2  
.....  
pCopy->Release();           // броячът става 1  
pCopy = pISome2;           // PCopy може да се преинициализира след Release()  
                             //едва ако броячът стане равен на 0 обекта се унищожава.
```

броенето се подчинява на следните правила

1. Всяка функция, изработваща указател към интерфейс, трябва да се обръща към `AddRef()` преди края си.
2. Всяка функция се обръща към `Release()` преди да припокрие с нова стойност указателя .
Викащите я функции се обръщат към `AddRef()` за предавания указател (ако има такъв), ако искат да поддържат отделно копие на указателя.
3. Вика се `AddRef()` за всяко локално копие на глобален указател към интерфейс.

Реализация на COM обект и свързване с контейнер



Изпълнимият файл с реализацията на COM обект се нарича COM сървър.

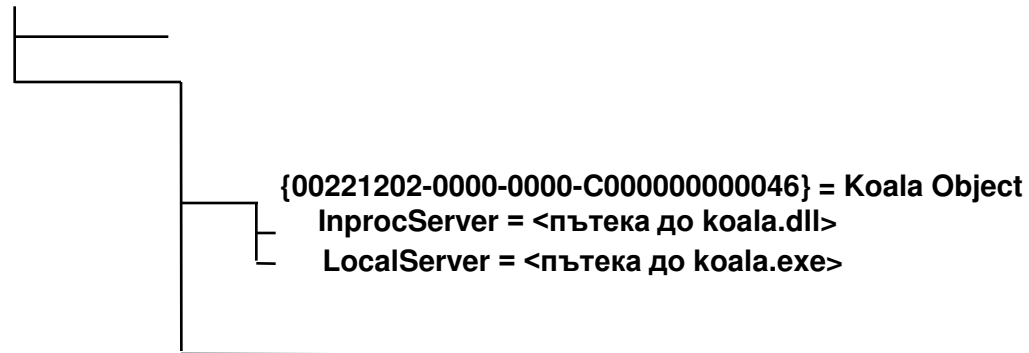
В секцията HKEY_CLASSES_ROOT\CLSID на регистъра на Windows има запис от вида :

```
HKEY_CLASSES_ROOT
  CLSID
```

```
    {E312522E-A7B7-11D1-A52E-0000F8751BA7}="Foo Class"
      InprocServer32="D:\\Examples\\Foo\\Foo.dll"
```

Например:

CLSID





Базови понятия

Локален сървър (in-proc) е сървърът, реализиран в DLL и работи в адресното пространство на клиента си.

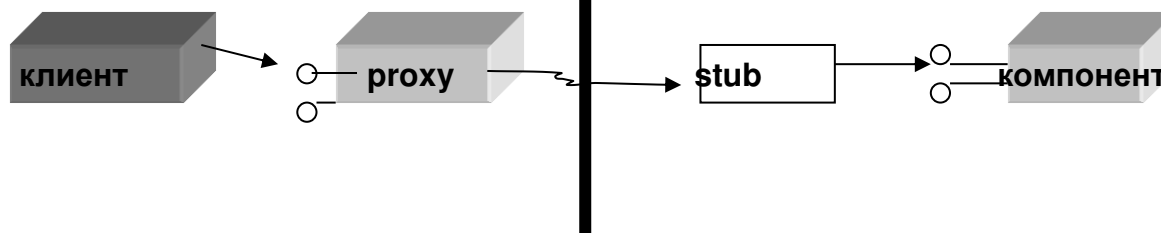
Отдалечен сървър (out-of-process) е реализиран в EXE и работи в свое адресно пространство.

Хостване в DCOM

Когато сървърът е в друг процес или компютър, COM системата прави междупроцесен marshaling (вмъква съответния код за това).

клиентско адр. пространство

адресно пространство
сървър





MIDL (Microsoft Interface Definition Language) компилатор е нужен за да се генерира правилно DLL с код и интерфейсни декларации.
Интерфейсите са описани предварително на езика **IDL (Interface Definition Language)**.

Пример:

Ако се каним да създадем нов интерфейс IFoo2 с три метода, IDL описанието изглежда така:

```
[ uuid(E312522F-A7B7-11D1-A52E-0000F8751BA7) ]  
interface IFoo2 : IUnknown  
{  
    HRESULT Func1();  
    HRESULT Func2(int in_only);  
    HRESULT Func3([in, out] int *inout);  
};
```

IDL код се създава и за описание на самия обект.

```
[ uuid(E312522E-A7B7-11D1-A52E-0000F8751BA7) ]  
coclass Foo  
{  
    [default] interface IFoo;  
};
```

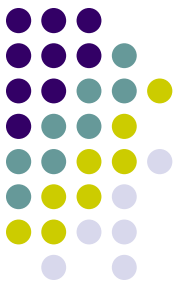


Интерфейс IUnknown (има IID-идентификатор – IID_IUnknown)

```
STDMETHODIMP QueryInterface(REFIID riid, void **ppvObject);  
STDMETHODIMP_(ULONG) AddRef();  
STDMETHODIMP_(ULONG) Release();
```

Характеристики на QueryInterface

1. Всяко обръщение към QueryInterface за един и същ обект с цел запитване относно IUnknown винаги връща един и същ указател.
Следователно за 2 интерфейса може да се определи дали принадлежат към 1 обект като се запита поотделно за IUnknown.
2. След като обект има проявление в паметта, интерфейсите му са статични.
Това значи, че ако QueryInterface успее в една точка на програмата, то ще успее при викане на същия интерфейс във всяка друга. Това не значи, обаче, че две повиквания на 1 интерфейс ще върнат еднакъв указател. Просто интерфейсът е винаги на разположение.



3. QueryInterface() е :

рефлексивен: `pInterface1->QueryInterface(IInterface1)` винаги ще успее.
симетричен ако `pInterface2` е изработен от `pInterface1->QueryInterface(IInterface2)`,
то `pInterface2->QueryInterface(IInterface1)` също ще успее.

транзитивен ако `pInterface2` е получен чрез `pInterface1->QueryInterface(IInterface2)`
`pInterface3` е получен от `pInterface2->QueryInterface(IInterface3)`,
То повикването: `pInterface3->QueryInterface(IInterface1)`
ще е успешно .

4. Докато обект съществува, всички указатели към интерфейси получени от този обект са валидни, дори ако са били освободени чрез `Release()` тъй като броячът е общ за обект, а не за отделен интерфейс. Например:

```
LPSOMEINTERFACE pSome;  
LPSOMEINTERFACE pOther;  
CreateSomeObject(&pSome);           //броячът става 1  
pSome->QueryInterface(IOtherInterface, &pOther); //броячът е 2  
pOther->Release();                   //броячът е отново 1  
//обектът съществува, pOther е още валиден, макар и освободен  
// с Release(). Това значи, че още можем да викаме функции чрез pOther  
.....  
pSome->Release();                     //броячът става 0, обектът се разрушава  
//pSome и pOther вече не са валидни
```



Нека IFoo е нашият пример за потребителски интерфейс.

За него сте дефинирали IID, например, **IID_IFoo** който има стойност, например: **"13C0205C-A753-11d1-A52D-0000F8751BA7"** получена от генератора на GUID.

Декларацията на класа има вид:

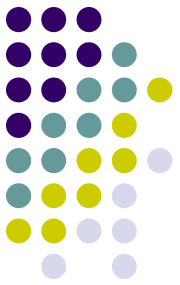
```
class IFoo
{
    virtual void Func1(void) = 0;
    virtual void Func2(int nCount) = 0;
};
```

За да стане декларацията COM-съвместима, нека леко я променим:

```
interface IFoo : IUnknown
{
    virtual HRESULT STDMETHODCALLTYPE Func1(void) = 0;
    virtual HRESULT STDMETHODCALLTYPE Func2(int nCount) = 0;
};
```

С използване на макроси, окончателно декларацията има вид:

```
interface IFoo : IUnknown
{
    STDMETHOD Func1(void) PURE;
    STDMETHOD Func2(int nCount) PURE;
};
```



Служебната дума **"interface"** изгражда служебна структура. (Спомнете си, че класовете в C++ и структурите имат еднакво вътрешно представяне, с изключение на това, че структурата има public наследяване).

STDMETHOD се дефинира като `__stdcall`, указвайки на компилатора да генерира стандартна конвенция на повикване на функциите.

Използването на тези макроси, различно дефинирани при различните платформи, допринася за преносимостта на кода.

фабрика за класове (Class Factory)

Какво става, когато се срещне `CoCreateInstance()`?

-Всяко пряко или косвено обръщение към `CoCreateInstance()`, кара COM-системата да претърсва регистъра за да открие CLSID на обекта, след което да намери и реализиращия го код в съответния DLL (или EXE).

`CoCreateInstance()` всъщност извършва следното:

```
IClassFactory *pCF;  
CoGetClassObject(rclsid, dwClsContext, NULL, IID_IClassFactory, (void **) &pCF);  
hresult = pCF->CreateInstance(pUnkOuter, riid, ppvObj)  
pCF->Release();
```

Реализация на класовия обект (Class Object)



```
class CMyClassObject : public IClassFactory
{
protected:
    ULONG m_cRef;
public:
    CMyClassObject() : m_cRef(0) { };
    //IUnknown членове
    STDMETHODCALLTYPE QueryInterface(REFIID, void **);
    STDMETHODCALLTYPE AddRef(void);
    STDMETHODCALLTYPE Release(void);

    // функции на IClassFactory
    STDMETHODCALLTYPE CreateInstance(IUnknown *, REFIID iid, void **ppv);
    STDMETHODCALLTYPE LockServer(BOOL);
};
```

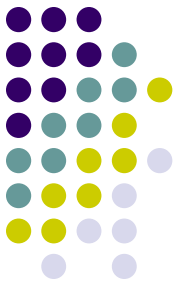
Всеки COM клас трябва да поддържа своя фабрика за класове. COM фабриките за класове съдържат предимно шаблонен код, така че тяхното създаване се автоматизира лесно от развойните среди.

В MFC библиотеката е предвиден клас от тази група - **COleObjectFactory**.

В него са реализирани два интерфейса – **IClassFactory** и **IClassFactory2** (използван в ActiveX технологиите), които вършат същинската работа по създаване инстанция на интересуващия ни клас.

Ето прототип на конструктора:

```
COleObjectFactory(REFCLSID clsid, CRuntimeClass* pRuntimeClass, BOOL bMultiInstance,
                  LPCTSTR lpszProgID );
```



Вместо да се декларира инстанция на COleObjectFactory във всяко приложение с COM обекти, VISUAL C++ дефинира макросите

DECLARE_OLECREATE

IMPLEMENT_OLECREATE

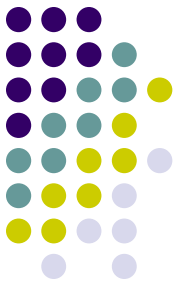
съответно в декларацията на класа и неговата реализацията.

За да се създаде копие (instance) на COM обект има 2 начина:

1. Чрез OLE функции *CoCreateInstance()* или *CoGetClassObject()*;
2. Чрез обекта - class factory и неговия интерфейсен метод:
IClassFactory::CreateInstance();

Ако е нужна само една инстанция на обекта, това може да стане с *CoCreateInstance()* с подадени CLSID и IID за искания интерфейс.

Например:



```
.....  
LPUNKNOWN pUnkOuter = NULL;
```

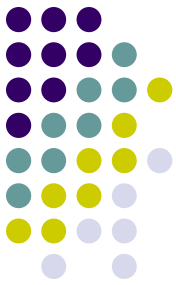
```
.....  
CoCreateInstance(CLSID_Koala, pUnkOuter, dwClsCtx, IID_IPersist,  
                (LPLPVOID) &pIPersist);
```

reference към идентификатор на класа на обекта;

указател към обхващащия обект, ако обекта е създаден като част от агрегат (виж частта, описание на процеса агрегиране)

комбинация от флагове, описващи контекста в който обекта ще работи:
в EXE - локален сървър, в DLL и т.н.

IID, който искаме да получим за този обект. Ако обектът не поддържа този интерфейс, функцията връща стойност за неуспех. указател към мястото, в което CoCreate...() ще съхрани изработения указател.



CoCreateInstance() се изпълнява на 3 стъпки:

1. Изработва се обект - class factory и негов интерфейс - IClassFactory.
2. Вика се IClassFactory::CreateInstance() за създаване на нужния обект, в примера по-горе - Koala.
3. Вика се IClassFactory::Release() за ненужния вече обект - class factory.

**ПРИМЕР: създаване и експониране на примерен, потребителски COM обект - “Koala”,
поддържащ интерфейс IPersist:**

За целта следва да създадем два файла:–

с декларации **koala.h**

с описание **koala.cpp**

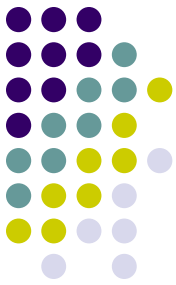
Файлът **koala.h** съдържа декларации на класа CKoala, интерфейс IUnknown и friend-клас:

CImpPersist: public IPersist

IPersist е реализация на стандартен интерфейс. Обектът Koala е агрегиращ (ще говорим за това по-късно). В този клас са декларациите на експортируемите интерфейси.

Файлът **koala.cpp** съдържа реализацията на класа.

Методите на обекта, които следва да бъдат реализирани са:



```
CKoala(LPUNKNOWN pUnkOuter,...); //конструктор
~CKoala();                        // деструктор
FInit();                          // метод, изработващ указател към
                                  // IUnknown, или UnkOuter.

// трите метода на собствения за Koala COM обекта интерфейс IUnknown
QueryInterface();
AddRef();
Release();

// следват методи на вграден (агрегиран) клас и реализация на IPersist
CImpPersist::CImpPersist();      // конструктор
~CImpPersist();                  // деструктор
// следват трите метода на IUnknown – собствена реализация
QueryInterface();
AddRef();
Release();
GetClassID();
```



Ако обектът поддържа **агрегиране**, той следва да притежава средства за откриване на всички интерфейси, които са реализирани в обхващащия го клас (какъвто е случая с интерфейса IPersist е обекта CKoala).

При агрегиране, реализацията на интерфейс изпраща повикванията на IUnknown винаги към обекта, който контролира (обхваща) настоящия.

Ето защо в конструктора на класа CKoala е предвиден указател към интерфейса IUnknown на обхващащия обект (pUnkOuter). Ако обектът “Koala” не е агрегиран (този параметър е NULL), CKoala:FInit() предава указател към собствения си IUnknown, вместо към този на обхващащия обект. Така винаги имаме достъп до интерфейсите на двата обекта: външен и агрегиран.

(Функцията FInit() е предвидена като отделна, втора фаза в конструирането на обекта “Koala”. Това е направено с цел повишаване надеждността на етапа конструкция на обекта. FInit() вика конструктора на CImpPersist, където поради алокиране на памет и търсене на агрегиращ обект може да има неуспех.)

За да се конструира коректно обекта чрез реализацията си в сървър, остава да се реализира и обекта **class factory** (както и за всеки потребителски COM обект) През който ще се реализират инстанциите (в случая за обекта “CKoala”). Обектът Class factory поддържа стандартно интерфейс IClassFactory. **В сървъра се реализира клас CKoalaClassFactory например така:**



```
class _far CKoalaClassFactory : public IClassFactory
{
    protected:
        ULONG m_cRef;           // брояч на обръщенията

    public:
        CKoalaClassFactory(void);
        ~CKoalaClassFactory(void);

        STDMETHODCALLTYPE QueryInterface(REFIID, LPLPVOID);
        // STDMETHODCALLTYPE е макрос, скриващ различията в декларации
        // на типове в C, C++, 16, 32 битови версии и Macintosh.
        STDMETHODCALLTYPE AddRef(void);
        STDMETHODCALLTYPE Release(void);

        STDMETHODCALLTYPE CreateInstance(LPIUNKNOWN, REFIID, LPLPVOID);
        STDMETHODCALLTYPE LockServer(BOOL);
}
```

Съществена тук е функцията **CreateInstance()**. Първият параметър е указател към IUnknown на обхващащ обект (ако съществува агрегиране). Целта е да се реализира връзка с външния (обхващащ) обект и той да може да запита за IUnknown на съставния си. Ако липсва агрегиране, този параметър е NULL.



Функцията **LockServer()** инкрементира или декрементира брояча в сървъра.

Чрез него сървърът може да остане в паметта, дори ако не обслужва повече обекти. Това е оптимизация за избягване на следващо зареждане.

Фабриката за класове трябва да се направи видима. Възможните за това реализации са различни според мястото на сървъра - в EXE или в DLL. В EXE се дефинира нова задача (task). Нека разгледаме по-простата процедура, когато се използва DLL. Всеки сървър реализира и експортира функция **DllGetClassObject()**, която извършва следното:

Когато потребител повика **CoCreateInstance()** за създаване на обект (или **CoGetClassObject()**), COM библиотеката претърсва регистъра за запис с търсения клас- идентификатор (CLSID) за да зареди съответния сървър в паметта, след което търси в него реализация на **DllGetClassObject()**, извикана със заявените от потребителя параметри за CLSID и IID. Чрез тази функция се създава class factory за нужния клас идентификатор CLSID и се получава указател към заявения интерфейс. Този начален интерфейс обикновено е **IClassFactory** за конкретния COM обект.

Това, всъщност, е протокола на създаване на COM обекти

Наследяемост в компонентна среда - механизми



1. съдържане (containment) – пример: Коала съдържа Животно и реализира версия на Животно, която експонира.

2. агрегиране: Коала експонира директно интерфейсите на родителя като свои. Това изисква пренасочване на повикванията към реалните родителски реализации. Т.е следва да има специфична реализация на QueryInterface(), AddRef(), Release().

3. Старата техника на множествено наследяване. Започваме с този подход:

искаме да създадем COM обект, поддържащ интерфейсите IDrawing и IOutlet, при това с използване на **множествено наследяване**:

```
class IOutlet :    public IUnknown {...} ;
class IDrawing :  public IUnknown {...} ;
class CLightBulb : public IDrawing,    public IOutlet
{
    public:
        // реализация на IUnknown
        virtual HRESULT __stdcall QueryInterface(REFIID riid, void **ppv) ;
        virtual ULONG __stdcall AddRef() ;
        virtual ULONG __stdcall Release() ;
        // реализация на интерфейса IDrawing
        virtual HRESULT __stdcall Draw(CDC* pDC, int x, int y);
        virtual HRESULT __stdcall SetPalette(CPalette* pPal);
        virtual HRESULT __stdcall GetRect(CRect* pRect);
        // реализация на интерфейса IOutlet
        virtual HRESULT __stdcall On();
        virtual HRESULT __stdcall Off();
        virtual HRESULT __stdcall GetState(BOOL* pState);
    ...// следват детайлите по реализацията
    ...}
```



Доколкото в езика C++ клас и структура се третират по еднакъв начин, един интерфейс може да наследява друг, както и клас да наследява интерфейс.

Подходът с множествено наследяване има очевидни предимства: интерфейсите, които имаме намерение да имплементираме в класа просто се изброяват в списък като от базови класове. IUnknown ще се имплементира само веднаж .

дефиницията на IUnknown е еднократна, независимо от това, че той е наследяван във всеки интерфейс (и в IDrawing и в IOutlet).

Да си спомним: чрез виртуалност може да се създаде многообразие. Например:

```
class A
{
    virtual int f();
} ;
class B
{
    virtual int f();
} ;
class C : public A, public B
{
    virtual int f() ;
} ;
```

т.е можем да мислим за различни реализации на еднакво декларираната f() !!!

Тъй като всеки отделен интерфейс се нуждае от методите на IUnknown, при множествено наследяване може да имаме обща тяхна реализация, която повторно да използваме навсякъде. Тогава, да замислим отделен клас CUnknown, който съдържа тази реализация:

```
class CUnknown : public IUnknown {...} ;  
...  
class CLightBulb : public CUnknown, public IDrawing, public IOutlet  
{  
...  
} ;
```



За съжаление, такава реализация няма да работи както се очаква. CUnknown няма да може да имплементира достатъчно пълноценно IUnknown. Той следва да е реализиран и в IDrawing и IOutlet, а и в наследника им CLightBulb и то по различен начин.

Абстрактно-виртуални функции се имплементират в (и само в) производния клас (а тук имаме и 2 родителя и наследил ги клас).

А ние искаме да имплементираме IUnknown в CLightBulb. Отделно IUnknown следва да е имплементиран и в IDrawing и в IOutlet за да разпознава и техните методи. Стана объркващо!

И така, трябва да имплементираме няколко реализации на интерфейса IUnknown: един в CUnknown, друг в CLightBulb и по 1 в наследяваните интерфейси.

IUnknown в CUnknown ще прави това което искаме от IUnknown и

IUnknown в CLightBulb ще пренасочва повикванията към горния IUnknown.

IUnknown в CLightBulb се нарича делегиращ (пренасочващ) IUnknown, а CUnknown ще съдържа не-делегиращ IUnknown.

Как обаче ще знаем предварително кой наследник ще се появи на даден клас за да предвидим пренасочването правилно? А какво ще правим при наличие на методи с еднаква сигнатура в различни , но свързани класове? ЛОШО!

Преминаваме към обсъждане на механизма на **съдържане** (**object containment**)

искаме да използваме съществуващ интерфейс (напр. IЖивотно) в наш компонент (СКоала).

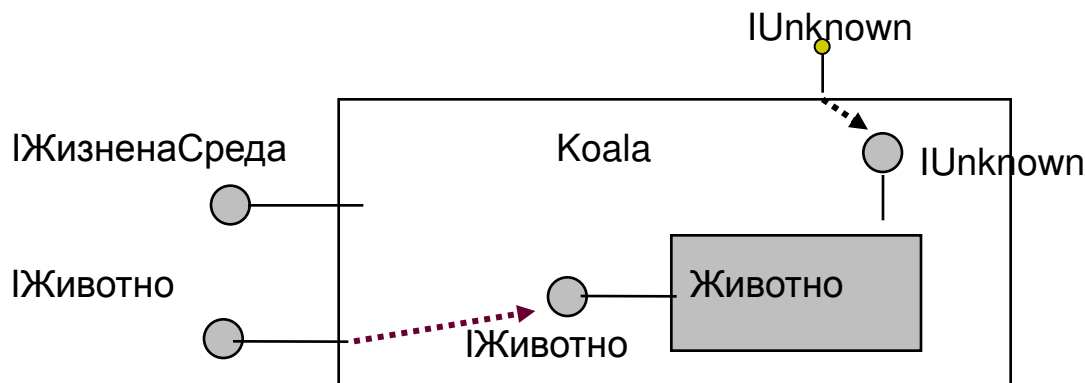
Това можем да постигнем като в нашия обект създадем инстанция

(чрез CoCreateInstance()) на другия и при необходимост викаме неговите методи.

Тогава нашият обект (СКоала) е външен, а другият е вътрешен. Ако поддържаме и подходящо сме инициализирали указател **pInnerObject**, сочещ към интерфейси на вътрешния обект, то с негова помощ можем да викаме методите им, като че са реализирани във външния (СКоала).

Например така:

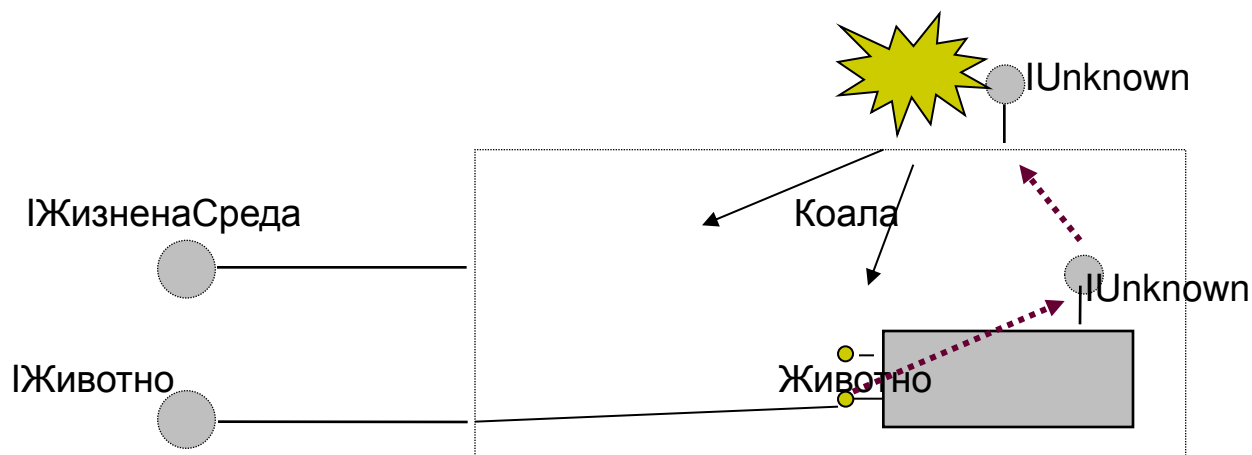
```
HRESULT __stdcall CKoala::Eating(...)\n{\n    return pInnerObject->Eating(...);    //викаме метод Eating() на вградения компонент\n                                          // от интерфейса IЖивотно\n}
```



третият подход –агрегиране (object aggregation)



Сега картинката от предния слайд би изглеждала така:



***Пренасочването на повикванията е съществена част за разбиране на агрегацията.
Тя пък е основния механизъм в компонентното програмиране***

```
// CSomeObject е агрегиращ обект, реализиращ IUnknown и ISomeInterface
class CSomeObject : public IUnknown
{ private:
    DWORD          m_cRef;          // брояч на обръщанията
    IUnknown*      m_pUnkOuter;     // Controlling IUnknown for this class
```

// вътрешен клас реализира интерфейс ISomeInterface:

```
class CImpSomeInterface : public ISomeInterface
{
    friend class CSomeObject ;
private:
    IUnknown*      m_pUnkOuter;     // Controlling IUnknown for this class
public:
    CImpSomeInterface() { m_cRef = 0; };
    ~ CImpSomeInterface(void) {};

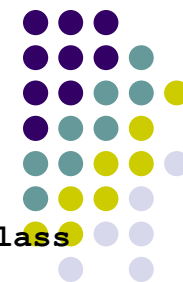
    // IUnknown методи, които насочват към външния IUnknown

    STDMETHODIMP QueryInterface(REFIID riid, void** ppv)
        { return m_pUnkOuter->QueryInterface(riid,ppv); };

    ...

    //Методи на ISomeInterface
    STDMETHODIMP SomeMethod(void)          { ... };
};
```

```
CImpSomeInterface m_ImpSomeInterface; //инстанция на вътрешния клас
```



Някой бележки по реализацията:



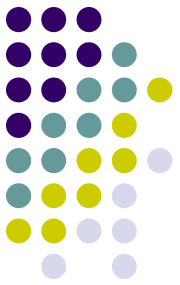
Методи
на външния
обект

```
public:
    CSomeObject(IUnknown* pUnkOuter) // конструктор на външния клас
    { инициализации
        ...
    } ;
    ~CSomeObject(void) {} ;           // деструктор

static HRESULT Create (IUnknown* pUnkOuter, REFIID riid, void **ppv)
{
    CSomeObject*    pObj;
    if (pUnkOuter != NULL && riid != IID_IUnknown)
        return CLASS_E_NOAGGREGATION;
    pObj = new CSomeObject(pUnkOuter);
    ...
}

// Методи на IUnknown на външния клас, които не пренасочват
STDMETHODIMP QueryInterface(REFIID riid, void** ppv)
{
    if (riid == IID_IUnknown)      *ppv=this;
    if (riid == IID_ISomeInterface) ....
} ;
STDMETHODIMP_(DWORD) AddRef(void) {...}
STDMETHODIMP_(DWORD) Release(void) {...}
};
```

Обърнете внимание за двойното дефиниране на методите на IUnknown. Реализацията им е различна.



предимства :

- възможност за съществуване на методи с еднаква сигнатура в различни интерфейси;
- единно броене на референциите към общия, агрегиращ обект и оттук облекчен механизъм за създаване и унищожаване;
- централизирано насочване на повикванията от единен, недеlegerащ QueryInterface().

Агрегация и MFC

MFC крие механизма в набор от макроси. Основните от тях са:

```
BEGIN_INTERFACE_PART( име_клас, име_на_интерфейс)
...
STDMETHOD(име_на_интерфейсен_метод)(списък от параметри);
STDMETHOD(име_на_интерфейсен_метод)(списък от параметри);
.....
END_INTERFACE_PART(име_клас)
```