E Überblick über die 13. Übung

- Objekt-Serialisierung
- serveraktivierte Objekte
- vom Client aktivierte Objekte
- lease-based Garbage Collection
- Metadaten-Assemblies

E.1 Serialisierung (2)

- Wie soll ein Objekt serialisiert werden?
- **`IFormatter`** Schnittstelle stellt Methoden bereit um ein Objekt zu serialisieren:
  - **`void Serialize(Stream outStream, object graph);`**
  - **`object Deserialize(Stream inStream);`**
- Implementierungen von **`IFormatter`**:
  - **`Formatter`**: abstrakte Basisklasse für eigene Implementierungen
  - **`SoapFormatter`**: Ausgabe im ASCII-Format (SOAP)
  - **`BinaryFormatter`**: kompakte, binäre Ausgabe

E.1 Objekt-Serialisierung

- Attribut **`[Serializable]`** markiert serialisierbare Klassen (transitiv)
  - Instanzvariablen werden automatisch serialisiert
  - Variablen, die mit **`[NonSerialized]`** markiert sind werden nicht serialisiert
- Beispiel:

```
[Serializable]
public class Account {
    private int value = 0;

    [NonSerialized]
    private Bank currentBank;

    public Account (int cash) {
        value = cash;
    }
}
```


E.1 Serialisierung (3) - Beispiel

- Serialisierung mittels **`Serialize`**:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;

Account myAccount = new Account(100);
FileStream myFile = File.Create("Account.txt");

new SoapFormatter().Serialize(myFile, myAccount);

myFile.Close();
```

- Deserialisierung mittels **`Deserialize`**:

```
FileStream myFile = File.Open("Account.txt", FileMode.Open);

Account myAccount =
       (Account) new SoapFormatter().Deserialize(myFile);

myFile.close();
```

E.1 Serialisierung (4)

- Was soll serialisiert werden?
- Beeinflussen der Serialisierung durch implementieren der Schnittstelle **`ISerializable`**:
  - Methode um das Objekt zu serialisieren, wird von der CLR aufgerufen
    `public void GetObjectData (SerialisationInfo info, StreamingContext context);`
  - Deserialisierungskonstruktor mit folgenden Parametern:
    `(SerializationInfo info, StreamingContext context);`

E.1 Serialisierung (5) - Beispiel

- Beispiel

```
[Serializable]
public class Account : ISerializable{
    private int amount;
    private Signatur mySig;

    [NonSerializable]
    private Bank currentBank;

    public void GetObjectData (SerializationInfo info,
                               StreamingContext context) {
        info.AddValue("amount", amount);
        info.AddValue("MySignatur", mySig);

        if (ctx.State == StreamingContextStates.CrossProcess)
            ;// Objekt wird an einem anderen Prozess übertragen
    }

    public Account(SerializationInfo info,
                               StreamingContext ctx){
        amount = info.GetInt32("amount");
        mySig = (Signatur) info.GetValue("mySignatur",
                                         typeof(Signatur));
    }
}
```


E.2 Marshalling

- Standard: Marshal by Value (MBV)
- Marshal by Referenz (MBR) durch ableiten von **`MarshalByRefObject`**
- Objektreferenz = **`ObjRef`**-Objekt
  - enthält Informationen über:
    - Name des Objekts incl. Name des Assembly
    - Typeninformationen über alle Basisklassen des Objekts
    - Typinformationen über alle implementierten Schnittstellen
    - die Adresse (URI) des Objekts incl. Informationen über den Kanal des Servers

E.3 Proxies

- implementieren die Methoden und Attribute des entfernten Objekts
- werden dynamisch erzeugt
- transparenter Proxy:
  - z.B.: durch **`Activator.GetObject`**
  - die Laufzeitumgebung kann damit die Anzahl und den Typ der Parameter prüfen
  - kann nicht verändert werden
  - leitet Methodenaufruf an echtes Objekt weiter
  - lokales Objekt: Methodenaufruf
  - entferntes Objekt:
    - Argumente in **`IMessage`** Objekt packen
    - mittels **`Invoke`** an den **`RealProxy`** übergeben

E.3 Proxies (2)

- **`RealProxy`**
  - ebenfalls dynamisch erzeugt
  - kann erweitert und verändert werden
  - Standardimplementierung von **`Invoke`**:
    - **`IMessage`** Objekt an Kanal weitergeben
- **`Activator.GetObject(...)`**
  - erzeugt einen transparenten Proxy für einen serveraktiviertes Objekt
  - keine Netzwerkinteraktion bei der Erstellung
    erst wenn eine Methode am Proxy aufgerufen wird

E.4 Channels und Formatter

- Channel:
  - verantwortlich für den Transport eines Methodenaufrufes und dessen Rückgabewertes über das Netzwerk
  - Beispiel:
    - **`TCPChannel`**: jeder Kanal benötigt einen eigenen Port
    - **`HTTPChannel`**: flexibler, mehrere Verbindungen können über Port 80 laufen
- Formatter:
  - Serialisiert die Objekte und legt somit das Format eines serialisierten Objekts fest
  - Beispiel.:
    - **`BinaryFormatter`**: Standard für TCPChannel
    - **`SOAPFormatter`**: Standard für HTTPChannel

E.4 Channels und Formatter (2)

- Kanäle müssen bei der Laufzeitumgebung registriert werden
  - Klasse **`ChannelServices`**
  - pro *Application Domain* kann immer nur ein Kanal des selben Typs registriert werden (Bsp: 1xTCP,1xHTTP, aber nicht 2xTCP)
- Kanal ist als Kette von Senken organisiert:
  im Kanal wird eine Kette von *Senken* durchlaufen. Durch einfügen einer eigenen Senke kann man einfluss auf die Übertragung nehmen
  - erste Senke ist üblicherweise der Formatter
  - letzte Senke ist der Transportdienst

E.5 Serveraktivierte vs. Clientaktivierte Objekte

- Einstiegspunkte in "Server"
- serveraktiviert Objekte, bekannte Typen
  (server-activated / well-known Objects)
  - besitzen einen eindeutgen, bekannten Namen
  - Laufzeitumgebung instantiiert das Objekt mittels Standardkonstruktor
  - Aktivierungsmodus bestimmt die Lebenszeit:
    - *SingelCall*: wird nach jedem Aufruf zerstört
    - *Singleton*: bleibt nach einem Aufruf am Leben
- clientaktivierte Objekte/Typen
  - können vom Client mit beliebigem Konstruktor erstellt werden
  - Lebenszeit: wird durch Lease-based Garbage Collection vorgegeben

E.6 Serveraktivierte Objekte

1 Singleton vs. SingleCall

- Singleton:
  - Lebenszeit: lease-based Lifetime
  - Anforderungen werden in separaten Threads ausgeführt
  - bei modifizierenden Zugriffen ist evtl. Synchronisation notwendig! (**`System.Threading`** oder Schlüsselwort **`lock`**)
- SingleCall:
  - für jede Anforderung wird ein eigenes Objekt erzeugt
  - keine Zustandsübermittlung zwischen Clients/Aufrufen

E.6 Serveraktivierte Objekte (2) - Server

- **`System.Runtime.Remoting.dll`** wird benötigt
  ➔ eine Referenz auf das entsprechende Assembly hinzufügen
- Klasse von **`MarshalByRefObject`** ableiten und implementieren
- Kanal auswählen und mittels **`ChannelService.RegisterChannel`** registrieren
- Die Klasse als "well-known object" registrieren, mittels **`RemotingConfiguration.RegisterWellKnownServiceType`**.
- Auf Anfragen von Clients warten

E.6 Serveraktivierte Objekte (3) - Server - Beispiel

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace BankServer {

   class ServerMain {

     static void Main (String[] args) {
       HttpChannel channel = new HttpChannel(4711);
       ChannelServices.RegisterChannel(channel);

       RemotingConfiguration.RegisterWellKnownServiceType(
                          typeof(BankLibrary.Bank),
                          "MyURI.soap",
                          WellKnownObjectMode.Singleton);

       Console.WriteLine("server startet");
       Console.ReadLine();
     }
   }
}
```

- hier wird noch kein Objekt erstellt

E.6 Serveraktivierte Objekte (4) - Client

- benötigte Informationen vom Server
  - Name des Servers
  - Typ des verwendeten Kanals
  - Port-Nummer an welcher der Server wartet
  - Die URI des entfernten Objekts
- **`System.Runtime.Remoting.dll`** wird benötigt
  ➔ eine Referenz auf das entsprechende Assembly hinzufügen
- BankLibrary Assembly wird benötigt, da es die Metadaten (z.B. Methodensignaturen) enthält, die benötigt werden um einen Proxy zu erstellen
- Kanal, vom selben Typ wie der des Servers registrieren
- mittels **`Activator.GetObject(...)`** einen Proxy erzeugen.
- Den Proxy in den entsprechenden Typ umwandeln und verwenden

E.6 Serveraktivierte Objekte (5) - Client - Beispiel

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using BankLibrary;

namespace BankClient {

    class ClientMain {
        public static void Main(String[] args) {
            HttpChannel channel = new HttpChannel();
            ChannelServices.RegisterChannel(channel);

            Object remoteObj = Activator.GetObject(
                        typeof(BankLibrary.Bank),
                        "http://localhost:4711/MyURI.soap"
                        );

            Bank bank = (Bank)remoteObj;
            Account account = new Account();
            bank.deposit(account, 3);

    }
   }
```


E.6 Serveraktivierte Objekte (6) - Client

- Alternative zu **`Activator.GetObject`**: Als entfernter Typ registrieren:

```
RemotingConfiguration.RegisterWellKnownClientType(
                    typeof ( BankLibrary.Bank ),
                    "http://localhost:4711/MyURI.soap");

new Bank();
```


E.7 Clientaktivierte Objekte

- Serveraktivierte Objekte werden immer mit dem Standardkonstruktor erstellt

```
class ClientMain {

    public static void Main(String[] args) {

        RemotingConfiguration.RegisterWellKnownClientType(
                    typeof ( BankLibrary.Bank ),
                    "http://localhost:4711/MyURI.soap");

        new Bank("Sparkasse");       // runtime exception!!
    }
}
```

- Clientaktivierte Objekte: für jeden Client wird eine eigene Instanz am Server erzeugt
- die Instanz bleibt über mehrere Aufrufe hinweg aktiv.

E.7 Clientaktivierte Objekte (2) - Server

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace BankServer {

   class ServerMain {

     static void Main (String[] args) {
       HttpChannel channel = new HttpChannel(4711);
       ChannelServices.RegisterChannel(channel);

       RemotingConfiguration.RegisterAcivatedServiceType(
                                  typeof(BankLibrary.Bank));

       Console.WriteLine("server startet");
       Console.ReadLine();
     }
   }
}
```

E.7 Clientaktivierte Objekte (2) - Client

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using BankLibrary;

namespace BankClient {

    class ClientMain {
        public static void Main(String[] args) {
            HttpChannel channel = new HttpChannel();
            ChannelServices.RegisterChannel(channel);

            RemotingConfiguration.RegisterActivatedClientType(
                        typeof ( BankLibrary.Bank ),
                        "http://localhost:4711");

            Bank bank = new Bank("Sparkasse");
        }
    }
}
```


E.8 Remotekonfigurationsdatei

- zur einfachen Konfiguration der Kanäle und registrierten Objekte
- Format: XML
- Position und Name:beliebig
  meist jedoch in Anwendungskonfigurationsdatei

```
<configuration>
    <system.runtime.remoting>
    ....
    </system.runtime.remoting>
</configuration>
```

- Einlesen:

```
RemoteingConfiguration.Configure("Customer.exe.config");
```


E.8 Remotekonfigurationsdatei (2) - Beispiel

- Server:

```
<application>
    <service>
        ...
    </service>

    <channels>
        <channel port = 4711
            type="System.Runtime.Remoting.Channels.Http.HttpChannel,
               System.Runtime.Remoting, Version=1.0.3300.0,
               Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    </channels>
</application>
```

- Client:

```
<application>
    <client>
        ...
    </client>
    <channels>
        <channel type="..." />
    </channels>
</application>
```


E.8 Remotekonfigurationsdatei (3)

- Kanal-Vorlagen:

```
<configuration>
    <system.runtime.remoting>
        <application>
            <channels>
                <channel port=4711 ref="MyChannel" />
            </channels>
        </application>
        <channels>
            <channel id="MyChannel" type="..." />
        </channels>
    </system.runtime.remoting>
</configuration>
```

- vordefinierte Vorlagen z.B.: **`http`**, **`tcp`**

E.8 Remotekonfigurationsdatei (2) - Beispiel

- serveraktiviertes Objekt: Server

```
<application>
    <service>
        <wellknown mode="Singleton"
                   type="BankLibrary.Bank, Bank"
                   objectUri="MyURI.soap" />
    </service>
    <channels>
        <channel port = 4711 type="http" />
    </channels>
</application>
```

- serveraktiviertes Objekt: Client

```
<application>
    <client displayName="BankClient">
        <wellknowntype="BankLibrary.Bank, Bank"
                  url="http://localhost:4711/MyURI.soap" />
    </client>
    <channels>
        <channel type="http" />
    </channels>
</application>
```


E.8 Remotekonfigurationsdatei (2) - Beispiel

- clientaktiviertes Objekt: Server

```
<application>
    <service>
        <activated type="BankLibrary.Account, Bank" />
    </service>
    <channels>
        <channel port=4711 id="http" />
    </channels>
</application>
```

- clientaktiviertes Objekt: Client

```
<application>
    <client url="http://localhost:4711" >
      <activated type="BankLibrary.Account, Bank" />
    </client>
    <channels>
        <channel type="http" />
    </channels>
</application>
```


E.8 lease-based Garbage Collector

- DCOM:
  - Verweiszähler
  - durch "pings" werden nicht erreichbare Clients entdeckt
- .NET: lease-based GC:
  - jedem Objekt wird eine bestimmte Zeit (*Lease*) eingeräumt, während der es nicht gelöscht wird
  - durch die Nutzung des Objekts kann die Lease verlängert werden
- lease-based GC wird für Singleton und clientaktivierte Objekte verwendet
- **`MarshalByRefObject.GetLifetimeService()`**
  liefert ein Objekt vom Typ **`ILease`** mit Informationen zur Lebenszeit:

```
public class myClass : MarshalByRefObject {
    public example() {
        ILease myLease = (ILease)this.GetLeaseTime();
    }
}
```


E.8 lease-based Garbage Collector (2)

- Eigenschaften des **`ILease`** Interfaces:
  - **`TimeSpan CurrentLeaseTime`**
    (nur lesbare) verbleibende Zeit, bis das Objekt gelöscht werden kann
  - **`TimeSpan InitialLeaseTime`**
    Zeit, die ein Objekt nach der Erzeugung bekommt.
    (Standard: 5 Minuten)
  - **`TimeSpan RenewOnCallTime`**
    Zeit, die ein Objekt nach einem Aufruf mindestens noch am Leben bleibt
    (Standard: 2 Minuten)
  - **`LeaseState CurrentState`**
    Zustand der Lease: **`Active`**, **`Expired`**, **`Initial`**, **`Null`** oder **`Renewing`**
- Methoden:
  - **`void Register(ISponsor)`**
    einen Sponsor registrieren

E.8 lease-based Garbage Collector (3)

- Konfiguration der Lease-Zeiten
  - für alle Objekte einer Anwendung: durch Remotekonfigurationsdatei

```
<application>
    <lifetime leaseTime="10s" renewOnCallTime="5s" />
    <service>
        <activated type="BankLibrary.Account, Bank" />
    </service>
    <channels>
        <channel port = 4711 type="http" />
    </channels>
</application>
```

  - für einzelne Objekte: InitializeLifetimeService überschreiben

```
public override object InitializeLifetimeService() {
    ILease leaseInfo =
                  (ILease)base.InitializeLifetimeService();
    leaseInfo.InitialLeaseTime = TimeSpan.FromSeconds(7);
    leaseInfo.RenewOnCallTime = TimeSpan.FromSeconds(3);
    // unendliche Laufzeit: InitialLeaseTime = TimeSpan.Zero
    //                oder: return null;
    return leaseInfo;
}
```


E.8 lease-based Garbage Collector (4) - Sponsoren

- ist die Lease abgelaufen, so werden alle Sponsoren gefragt, ob die Lease verlängert werden soll
- Sponsor implementiert das Interface **`ISponsor`**
  - **`public TimeSpan Renew (ILease leaseInfo)`**
    wenn 0 zurückgegeben wird, so wird der Sponsor von der Liste entfernt
- registrieren mittels **`ILease.Register(ISponsor)`**
- weitere Einstellungsmöglichkeiten:

```
<application>
    <lifetime  leaseTime="10s"
               renewOnCallTime="5s"
               sponsorshipTimeout="5s"
               leaseManagerPollTime="10s"/>
</application>
```


E.9 Metadaten-Assembly

- Problem: Client benötigt Metadaten zur Erstellung des Proxies
- bisher: Assembly muss auch auf Clientseite vorhanden sein
- Alternativen:
  - entferntes Objekt von Interface ableiten
    Interface Assembly ist sowohl beim Client als auch beim Server vorhanden
    - Nachteil: von einem Interface kann man kein Objekt erzeugen
    - → keine clientaktivierte Objekte
    - → serveraktivierte Ojekte müssen mittels **`Activator.GetObject`** angesprochen werden
  - Metadaten Assembly
    Assembly mit gleichem Namen und gleichen Klassen incl. Methoden wie auf Serverseite, jedoch ohne Implementierung

E.9 Metadaten-Assembly (2) - soapsuds auf Clientseite

- Tool: soapsuds.exe erzeugt Metadaten-Assemblies
- auf Clientseite:

```
> soapsuds -url:http://localhost:4711/MyURI.soap?wsdl
                -oa BankProxy.dll
```

- erzeugt ein Proxy-Assembly, Proxy kann mittels **`new`** instantiiert werden, Serveradresse ist fest codiert .
- Alternative Ausgabeformate:
  - C#:

```
> soapsuds -url:http://localhost:4711/MyURI.soap?wsdl
                -gc BankProxy.cs
```

  - WSDL (Web Services Description Language):

```
> soapsuds -url:http://localhost:4711/MyURI.soap?wsdl
                -os bank.wsdl
```

E.9 Metadaten-Assembly (3) - soapsuds auf Clientseite

- bei clientactivierten Objekten:

```
> soapsuds -url:http://localhost:4711/
                        RemoteApplicationMetadata.rem?wsdl
              -oa BankProxy.dll
```

- liefert Metadaten für alle verwendbaren Objekte incl. "well-known" Objekte
- Nachteil: zu den Objekten werden keine Informationen über mögliche Konstruktoren erzeugt .

E.9 Metadaten-Assembly (4) - soapsuds auf Serverseite

- auf Serverseite:

```
> soapsuds -ia Bank -oa BankProxy.dll
```

- oder nur einzelne Typen:

```
> soapsuds
      -types:BankLibrary.Bank,Bank;BankLibrary.Account,Bank
      -oa BankProxy.dll
```

- Proxy mit fest codierter URL möglich:

```
> soapsuds
      -types:BankLibrary.Bank,Bank,http://localhost:4711/
                        MyURI.soap;BankLibrary.Account,Bank
      -oa BankProxy.dll
```