C Überblick über die 12. Übung

- Objekt-Serialisierung
- Marshalling, Messages, Proxies
- Channels & Formatters
- Objekterzeugung und Fernaufruf
  - Aktivierung durch den Server
  - Aktivierung durch den Client

C.1 Objekt-Serialisierung

1 Attribut `[Serializable]`

- Markierung serialisierbarer 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;
    }
}
```


2 Wie soll ein Objekt serialisiert werden?

- `IFormatter` Schnittstelle stellt Methoden bereit, um ein Objekt zu (de)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)
    - Vorteil: Lesbarkeit
    - Nachteil: Konvertierung aufwändig, große Datenmenge
  - `BinaryFormatter`: kompakte, binäre Ausgabe

3 Objekt-Serialisierung - 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();
```

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);
```


4 Was soll serialisiert werden?

- 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));
   }
}
```


C.2 Marshalling

- Wie werden Objekte übertragen? 2 Möglichkeiten:
  - Objekt fernaufrufbar: Fernverweis wird übertragen
  - Objekt serialisierbar: Objektkopie wird übertragen
  - sonst: Exception
- Standard: Marshal by Value (MBV)
  - Serialisierung: Attribut `[Serializable]`
- Marshal by Referenz (MBR)
  - durch Ableiten von `MarshalByRefObject`
  - übertragen wird eine Objektreferenz = `ObjRef`-Objekt
  - das Referenz-Objekt selbst wird serialisiert (MBV)

C.2 Marshalling

- `ObjRef`-Objekt enthält Informationen über:
  - Name des Objekts inkl. Name des Assembly
  - Typinformationen über alle Basisklassen des Objekts
  - Typinformationen über alle implementierten Schnittstellen
  - die Adresse (URI = "Uniform Resource Identifier") des Objekts
  - Informationen über den Kanal des Servers
- Client muss Typ des entfernten Objekts kennen!

C.3 Messages

- Analogie in der objekt-orientierten Programmierung:
  Methodenaufruf = Nachricht an ein Objekt
- .NET: "Message objects" realisieren Kommunikation zwischen entfernten Objekten
- Interface **`IMessage`** definiert ein Property vom Typ **`IDictionary`**
  - enthält URI des entfernten Objekts
  - Name der aufzurufenden Methode
  - Parameter der Methode
  - Elemente des Dictionaries werden beim Aufruf serialisiert

C.4 Proxies

- implementieren die Methoden und Attribute des entfernten Objekts
- werden dynamisch erzeugt!
- Dabei Aufteilung des Proxies in zwei Teile
- transparenter Proxy
  - plattform-abhängig, kann nicht verändert werden
- **`RealProxy`**
  - abstrakte Basisklasse, erweiterbar durch eigene Implementierung
  - default-Implementierung der .NET-Architektur:
    `System.Runtime.Remoting.Proxies.RemotingProxy`

1 Transparenter Proxy

- wird beim Client transparent erzeugt, z.B.: durch **`Activator.GetObject`**
- besitzt nach außen dasselbe Interface wie das "richtige" Objekt
- die Laufzeitumgebung kann die Anzahl und den Typ der Parameter prüfen
- leitet Methodenaufruf an echtes Objekt weiter
- lokales Objekt: Methodenaufruf
- entferntes Objekt:
  - Argumente in **`IMessage`** Objekt packen
  - mittels **`Invoke`** an den **`RealProxy`** übergeben

2 RealProxy

- **`RealProxy`**
  - ebenfalls dynamisch erzeugt
  - kann erweitert und verändert werden

```
public class MyProxy : RealProxy {
    MarshalByRefObject _target;
    public MyProxy(Type type, MarshalByRefObject t) :
        base(type) {
        _target = target;
    }

    public override IMessage Invoke(IMessage msg) {
        // Standardimplementierung: msg 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

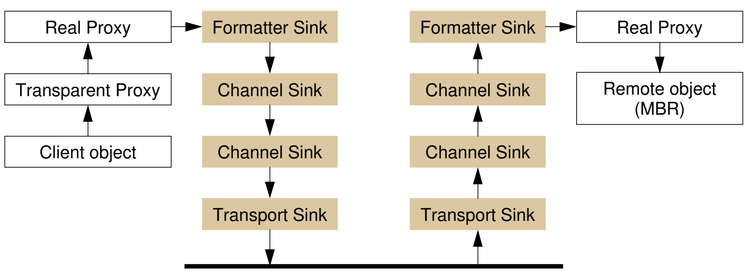
C.5 Channels und Formatter

1 Channels

- verantwortlich für den Transport eines Methodenaufrufs und dessen Rückgabewert über das Netzwerk
- Beispiel:
  - `TCPChannel`: jeder Kanal benötigt einen eigenen Port
  - `HTTPChannel`: flexibler, mehrere Verbindungen können über Port 80 laufen
- Kanäle müssen bei der Laufzeitumgebung registriert werden
  - Klasse `ChannelServices`
  - pro *Application Domain* kann immer nur ein Kanal des selben Typs registriert werden (Bsp: 1x TCP, 1x HTTP, aber nicht 2x TCP)
- Kanal ist als Kette von Senken (*Sinks*) organisiert, die der Reihe nach durchlaufen werden.

1 Channels

- Durch Einfügen einer eigenen Senke kann man Einfluss auf die Übertragung nehmen
  - erste Senke ist üblicherweise der Formatter
  - letzte Senke ist der Transportdienst
  - Beispiele: Verschlüsselungs-Senke oder Logging-Senke



2 Formatter

- Serialisiert die Objekte und legt somit das Format eines serialisierten Objekts fest
- Beispiel:
  - `BinaryFormatter`: Standard für TCPChannel
    binäre Übertragung: kompaktes Datenformat
  - `SOAPFormatter`: Standard für HTTPChannel
    XML-basiertes Format: größere Datenmenge, aber flexibler
  - Interface `IFormatter` und abstrakte Basisklasse `Formatter` für eigene Implementierungen (s. Kapitel über Serialisierung)

C.6 Objekterzeugung und Fernaufruf

- Einstiegspunkte in "Server"
- Bei MBR-Objekten Aktivierung notwendig
- Bekanntmachung eines öffentlichen Objekts
  - Objekt wird lokal erzeugt und dann "veröffentlicht"
- Fernerzeugung eines öffentlichen Objekts
  - Erzeugung wird lokal vorbereitet
  - beim ersten Zugriff durch einen Client wird Objekt erzeugt
- Fernerzeugung eines privaten Objekts
  - Objekt wird auf anderen Rechner erzeugt
  - Erzeuger erhält Referenz darauf

1 Vorgehensweise

- Als Beispiel dient die Klasse **RemoteBank** (s. **/proj/i4mw/pub/net**)
- **System.Runtime.Remoting.dll** wird benötigt
  ➔ eine Referenz auf das entsprechende Assembly hinzufügen

```
mcs -r:/local/mono/lib/System.Runtime.Remoting.dll ...
```

- Klasse von **MarshalByRefObject** ableiten und implementieren

```
namespace BankLibrary {
    public class RemoteBank : MarshalByRefObject { ... }
}
```

- Kanal auswählen und mittels **ChannelService.RegisterChannel** registrieren
- Die Klasse registrieren / aktivieren (s. nächste Folien)
- Auf Anfragen von Clients warten

1 Vorgehensweise

- Client muss Typ des entfernten Objekts kennen
  - ◆ einfachster Fall: Client hat Zugang zur Implementierung (**RemoteBank.dll**)
  - ◆ Stand-In class:

```
public class RemoteBank : MarshalByRefObject {
    public RemoteBank() {
        throw new System.NotImplementedException();
    }
    public RemoteBank(String name) {
        throw new System.NotImplementedException();
    }
    public int deposit(int amount) {
        throw new System.NotImplementedException();
    }
}
```

  - ◆ Verwendung eines Interfaces (z.B. **IRemoteBank**) als Remote-Objekt

```
interface IRemoteBank {
    int deposit(int amount);
}
```


2 Bekanntmachung eines öffentlichen Objekts

- Objekt erzeugen und mit **RemotingServices.Marshal** bekanntgeben

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

namespace BankServer {

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

           RemoteBank bank = new RemoteBank();
           RemotingServices.Marshal(bank,
               "TestBank.soap");

           Console.WriteLine("Server started.");
           Console.ReadLine();
       }
   }
}
```


3 Fernerzeugung öffentlicher und privater Objekte

- "Serveraktivierte" oder öffentliche Objekte (server-activated / well-known objects)
  - ◆ besitzen einen eindeutigen, bekannten Namen (URI): *well-known object*
  - ◆ Lebenszeit: Singleton (stateful) oder SingleCall (stateless)
  - ◆ Laufzeitumgebung instantiiert das Objekt mittels Standardkonstruktor (kein anderer Konstruktor möglich)
- "Clientaktivierte" oder private Objekte (client-activated / private objects)
  - ◆ für jeden Client wird eine Instanz mit eigenem Identifier (URI) erzeugt
  - ◆ können vom Client mit beliebigem Konstruktor erstellt werden
  - ◆ Lebenszeit: wird durch Lease-based Garbage Collection vorgegeben
  - ◆ wenn Lease abgelaufen, kann das Objekt vom GC zerstört werden (statefull)

4 Serveraktivierte Objekte

- Aktivierungsmodus bestimmt die Lebenszeit:
  - *SingleCall*: wird bei jedem Aufruf neu erzeugt und anschließend wieder zerstört (stateless)
  - *Singleton*: bleibt nach einem Aufruf am Leben (statefull)
- Singleton:
  - Lebenszeit: lease-based Lifetime
  - Anforderungen werden in separaten Threads ausgeführt (CLR stellt ThreadPool zur effizienten Ausführung bereit)
  - bei modifizierenden Zugriffen ist evtl. Synchronisation notwendig! (**`System.Threading`** oder Schlüsselwort **`lock`**)
- SingleCall:
  - für jede Anforderung wird ein eigenes Objekt erzeugt (Overhead!)
  - keine Zustandsübermittlung zwischen Clients/Aufrufen

4 Serveraktivierte Objekte - Client

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

4 Serveraktivierte Objekte - Server

- Beispiel (es wird noch kein Objekt erstellt!)

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

namespace BankServer {

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

      RemotingConfiguration.RegisterWellKnownServiceType(
                           typeof(BankLibrary.RemoteBank),
                           "TestBank.soap",
                           WellKnownObjectMode.Singleton);
      // analog: WellKnownObjectMode.SingleCall

      Console.WriteLine("Server started.");
      Console.ReadLine();
    }
  }
}
```


4 Serveraktivierte Objekte - 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) {
           TcpChannel channel = new HttpChannel();
           ChannelServices.RegisterChannel(channel);

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

           RemoteBank bank = (RemoteBank)remoteObj;
           Console.WriteLine("Balance after deposit: {0}",
               bank.deposit(3));
       }
   }
}
```

4 Serveraktivierte Objekte - Client

- Alternative zu **Activator.GetObject**: als entfernter Typ registrieren:

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

RemoteBank b = new RemoteBank();
```

- Weitere Alternative:

```
RemoteBank b = (RemoteBank) RemotingServices.Connect(
                  typeof( BankLibrary.RemoteBank ),
                  "http://localhost:4711/TestBank.soap");
```


5 Clientaktivierte Objekte - Server

- Beispiel 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.RemoteBank ) );

         Console.WriteLine("Server started.");
         Console.ReadLine();
      }
   }
}
```


5 Clientaktivierte Objekte

- Serveraktivierte Objekte werden immer mit dem Standardkonstruktor erstellt.

```
class ClientMain {

    public static void Main(string[] args) {

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

        RemoteBank bank = new RemoteBank("Sparkasse");
        // runtime exception!!
    }
}
```

- Clientaktivierte Objekte: für jeden Client wird eine eigene Instanz am Server erzeugt, beliebige Konstruktoren möglich
- Die Instanz bleibt über mehrere Aufrufe hinweg aktiv.
  - ◆ damit sind zustandsabhängige Aufrufe möglich
  - ◆ Leased-based Distributed Garbage Collector bestimmt Lebenszeit

5 Clientaktivierte Objekte - Client

- Beispiel 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.RemoteBank ),
             "http://localhost:4711");
         // alternativ: Activator.CreateInstance()

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

5 Clientaktivierte Objekte - Client

- Alternative

```
Object[] attr =
    { new UrlAttribute("http://localhost:4711") };
Object[] args = { "Sparkasse" };

RemoteBank bank = (RemoteBank)Activator.CreateInstance(
    typeof(BankLibrary.RemoteBank), args, attr);
```


6 Zusammenfassung

- Methoden der Klasse `Runtime.Remoting.RemotingConfiguration`:
  - ◆ Zum Registrieren serveraktivierter Objekte
    - Client: `RegisterWellKnownClientType()`
    - Server: `RegisterWellKnownServiceType()`
  - ◆ Zum Registrieren clientaktivierter Objekte
    - Client: `RegisterActivatedClientType()`
    - Server: `RegisterActivatedServiceType()`
- Klasse `System.Runtime.Remoting.RemotingService` zum Bekanntmachen öffentlicher Objekte
  - Server: `Marshal()`, wieder Entfernen mit `Disconnect()`, Client: `Connect()`
- Klasse `System.Activator`
  - ◆ Proxy erzeugen für öffentliche Objekte: `GetObject()`
  - ◆ für private Objekte: `CreateInstance()`

C.7 Parameterübergabe

- Wie werden Parameter übergeben?
  - ◆ Wertparameter (call-by-value)

```
void op(int n) { ... }
```

bei Fernaufrufen: ebenfalls call-by-value (Argumenten serialisieren)

  - ◆ Variablenparameter (call-by-reference)

```
void op(ref int n) { ... }
```

bei Fernaufrufen: *call-by-value-result*

  - ◆ Variablenparameter, evtl. nicht belegt (call-by-reference)

```
void op(out int n) { ... }
```

bei Fernaufrufen: *call-by-result*

C.7 Parameterübergabe

- Stolperfallen
  - ◆ Beispiel

```
void inc(ref int x, ref int y) {
    ... x++; ...
    ... y++; ...
}
```

Was passiert bei `x.inc(a[i], a[k])`, falls `i==k`?

  - ◆ Beispiel

```
interface IStack { ... }
class RemoteStack : MarshalByRefObject, IStack { ... }
class Stack       : IStack { ... }

... void op(IStack stack) { ... }
```

Man sieht dem Objekt nicht an, ob es `MarshalByRefObject` ist und wenn ja, ob es woanders liegt als der Aufrufer

➜ Unvorhersagbare Aufrufsemantik!

C.8 Lease-based Garbage Collector

- DCOM:
  - Verweiszähler
  - durch "pings" werden nicht erreichbare Clients entdeckt
  - Finden sog. "greedy clients", die absichtlich oder versehentlich eine Referenz auf das Objekt behalten
- .NET: lease-based GC:
  - jedem (marshal by reference)-Objekt wird eine bestimmte Zeit (*Lease*) eingeräumt, während der es nicht gelöscht wird
  - durch die Nutzung des Objekts kann die Lease verlängert werden
  - wenn nach Ablauf der Lease eine Methode an dem Objekt aufgerufen wird:
    - Singleton: neues Objekt wird erzeugt
    - clientaktiviertes Objekt: RemotingException

C.8 Lease-based Garbage Collector

- 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.GetLifetimeService();
   }
}
```


C.8 Lease-based Garbage Collector

- 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

C.8 Lease-based Garbage Collector

- 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;
}
```

C.8 Lease-based Garbage Collector - 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>
```

  - ◆ **`sponsorshipTimeout`**: Wartezeit auf Antwort eines Sponsors (2 Min.)
    = **`TimeSpan.Zero`**: die Lease nimmt keine Sponsoren an
  - ◆ **`leaseManagerPollTime`**: Überprüfung der Leases (10 Sekunden)