1 Binäre Serialisierung

In den allermeisten Fällen ist es wichtig, dass die Daten eines Programms nicht verloren gehen, wenn man das Programm beendet. Das bedeutet, dass der Programmierer irgendwie dafür sorgen muss, dass die Daten auch nach Beenden des Programms gesichert werden.

Für die dauerhafte Speicherung von Daten gibt es die sogenannte *Serialisierung*. Dieser Prozess speichert die Zustände von Objekten und sichert sie binär in einer Datei.

Neben der binären Serialisierung gibt es auch noch weitere Serialisierungsverfahren, wir beschränken uns hier jedoch auf die binäre Serialisierung.

1.1 Das Attribut »Serializable«

Bevor wir ein Objekt serialisieren können, müssen wir es mit Hilfe des Attributes `Serializable` kennzeichnen.

Eine Klasse kann folgendermaßen mit diesem Attribut markiert werden:

```
[Serializable()]
class <Klassenname>
{
  ...
}
```

Ohne dieses Attribut kann es zur einer `SerializationException` kommen, wenn wir versuchen, ein Objekt dieser Klasse zu serialisieren.

Wichtig: Lokale Variablen und statische Member werden nicht serialisiert, alle anderen Felder der Klasse hingegen schon.

1.2 Der Serialisierungsprozess

Sobald das Attribut Serializable nun einer Klasse zugewiesen wurde, können wir Objekte dieser Klasse serialisieren. Dazu erstellen wir uns zunächst eine Klasse und implementieren einige Felder.

```
[Serializable()]
class Rechteck
{
  public int Breite { get; set; }
  public int Hoehe { get; set; }
  private int flaeche;

  public Rechteck(int breite, int hoehe)
  {
    Breite = breite;
```

```
    Hoehe = hoehe;
    flaeche = Breite * Hoehe;
  }

  public int Flaeche { get { return flaeche; } }
}
```

Nun können wir Objekte der Klasse `Rechteck` folgendermaßen serialisieren:

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

namespace Serialisierung
{
  class Program
  {
    static void Main(string[] args)
    {
      Rechteck rechteck = new Rechteck(5, 10);

      try
      {
        FileStream stream =
            new FileStream(@"rechteck.dat", FileMode.Create);

        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(stream, rechteck);

        stream.Close();
      }
      catch (Exception e)
      {
        Console.WriteLine(e);
      }

      Console.ReadKey();
    }
  }

  [Serializable()]
  class Rechteck
  {
    public int Breite { get; set; }
    public int Hoehe { get; set; }
    private int flaeche;

    public Rechteck(int breite, int hoehe)
    {
      Breite = breite;
      Hoehe = hoehe;
      flaeche = Breite * Hoehe;
    }
```

```
    public int Flaeche { get { return flaeche; } }
  }
}
```

Der Serialisierungsprozess ist dank der Klasse `BinaryFormatter` leicht umzusetzen. Wir benötigen lediglich ein Objekt dieser Klasse, über welches wir dann die Methode zur Serialisierung aufrufen können. Dabei übergeben wir zwei Parameter:

- Einen `FileStream`, der angibt, in welcher Datei die Daten gespeichert werden sollen (Dateiendung ist hier `.dat`, da es sich um eine binäre Datei handelt).
- Das Objekt, welches serialisiert werden soll.

1.3 Der Deserialisierungsprozess

Nun wissen wir, wie wir Daten eines Objektes speichern können. Das hilft uns natürlich nur weiter, wenn wir die gespeicherten Daten auch wieder abrufen können. Dieser Vorgang nennt sich *Deserialisierung*.

Der Deserialisierungsprozess ist genau so einfach wie der Serialisierungsprozess.

```
Rechteck rechteck;

FileStream stream = new FileStream(@"rechteck.dat", FileMode.Open);

BinaryFormatter formatter = new BinaryFormatter();
rechteck = (Rechteck)formatter.Deserialize(stream);

stream.Close();
```

Wir benutzen ebenfalls ein Objekt der Klasse `BinaryFormatter`, um die zuvor gespeicherten Daten wieder abzurufen. Die Methode zur Deserialisierung gibt uns ein Objekt vom Datentyp `object` zurück, weshalb wir noch eine Typkonvertierung durchführen.

Nachfolgend gibt es noch ein Beispiel, welches sowohl Serialisierung als auch Deserialisierung beinhaltet.

Auch wenn in den folgenden Beispielen die `try...catch`-Blöcke fehlen (weil der Programmcode sonst zu lang gewesen wäre), sollte man stets bedenken, dass Ausnahmen auftreten können, wenn wir mit Dateien arbeiten.

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

namespace Serialisierung
{
  class Program
```

```
{
  static BinaryFormatter formatter;

  static void Main(string[] args)
  {
    formatter = new BinaryFormatter();

    Rechteck rechteck = new Rechteck(5, 5);
    Serialisieren(rechteck);

    Rechteck rechteck2 = Deserialisieren();
    Console.WriteLine("Breite: " + rechteck2.Breite);
    Console.WriteLine("Hoehe: " + rechteck2.Hoehe);
    Console.WriteLine("Flaeche: " + rechteck2.Flaeche);

    Console.ReadKey();
  }

  static void Serialisieren(object obj)
  {
    FileStream stream =
        new FileStream(@"unserObjekt.dat", FileMode.Create);

    formatter.Serialize(stream, obj);
    stream.Close();
  }

  static Rechteck Deserialisieren()
  {
    FileStream stream =
        new FileStream(@"unserObjekt.dat", FileMode.Open);

    Rechteck rechteck = (Rechteck)formatter.Deserialize(stream);
    stream.Close();

    return rechteck;
  }
}

[Serializable()]
class Rechteck
{
  public int Breite { get; set; }
  public int Hoehe { get; set; }
  private int flaeche;

  public Rechteck(int breite, int hoehe)
  {
    Breite = breite;
    Hoehe = hoehe;
    flaeche = Breite * Hoehe;
  }
```

```
    public int Flaeche { get { return flaeche; } }
  }
}
```

In diesem Beispiel sind die Methoden auf Objekte der Klasse `Rechteck` angepasst.

Zudem ist zu beachten, dass Ausnahmen auftreten können, wenn wir mit den Klassen `FileStream` und `BinaryFormatter` arbeiten.

1.4 Die Serialisierung mehrerer Objekte

Möchte man mehrere Objekte serialisieren, so kann man beispielweise die Objekte in einem Array speichern und dieses Array dann serialisieren. Die Objekte können auch unterschiedlichen Datentypen angehören, wenn wir Collections anstelle eines Arrays verwenden.

Wenn wir nun auf diese Weise serialisierte Objekte wieder deserialisieren möchten, dann ist zu bedenken, dass das zuerst serialisierte Objekt auch wieder zuerst deserialisiert wird (FIFO-Prinzip).

Das folgende Beispiel zeigt, wie wir mehrere Objekte serialisieren können.

```
static void Serialisieren(ArrayList list)
{
  FileStream stream = new FileStream(@"Objektliste.ifn", FileMode.Create);
  BinaryFormatter formatter = new BinaryFormatter();
  formatter.Serialize(stream, list);
  stream.Close();
}

static void Deserialisieren(ref ArrayList list)
{
  FileStream stream = new FileStream(@"Objektliste.ifn", FileMode.Open);
  BinaryFormatter formatter = new BinaryFormatter();
  list = (ArrayList)formatter.Deserialize(stream);
  stream.Close();
}
```

Dabei hat sich nicht viel im Vergleich zu den bisherigen Beispielen verändert. Wir nutzen nun einfach ein Objekt vom Typ `ArrayList` statt ein einzelnes Objekt zu übergeben. Zudem verwenden wir jetzt eine andere Dateiendung (`.ifn`) zur Speicherung der Daten.

Man hätte bei der Methode zur Deserialisierung natürlich auch einfach eine `ArrayList` zurückgeben können. Wenn man es so macht, wie in diesem Beispiel, sollte man jedoch daran denken, das Schlüsselwort `ref` zu verwenden, damit die Daten auch tatsächlich in der als Parameter übergebenen `ArrayList` gespeichert werden.

Im kommenden Beispiel wird noch einmal demonstriert, wie wir diese Methoden nun verwenden können.

```
using System;
using System.Collections;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace Serialisierung
{
  class Program
  {
    static void Main(string[] args)
    {
      Rechteck rechteck = new Rechteck(5, 5);
      Kreis kreis = new Kreis(5);

      ArrayList beispielListe = new ArrayList();
      beispielListe.Add(rechteck);
      beispielListe.Add(kreis);
      Serialisieren(beispielListe);

      ArrayList neueBeispielListe = new ArrayList();
      Deserialisieren(ref neueBeispielListe);

      Console.ReadKey();
    }

    static void Serialisieren(ArrayList list)
    {
      FileStream stream =
          new FileStream(@"Objektliste.ifn", FileMode.Create);

      BinaryFormatter formatter = new BinaryFormatter();
      formatter.Serialize(stream, list);

      stream.Close();
    }

    static void Deserialisieren(ref ArrayList list)
    {
      FileStream stream =
          new FileStream(@"Objektliste.ifn", FileMode.Open);

      BinaryFormatter formatter = new BinaryFormatter();
      list = (ArrayList)formatter.Deserialize(stream);

      stream.Close();
    }
  }

  [Serializable()]
  class Rechteck
  {
```

```
    public int Breite { get; set; }

    public Rechteck(int breite, int hoehe)
    {
      Breite = breite;
    }
  }

  [Serializable()]
  class Kreis
  {
    public int Radius { get; set; }

    public Kreis(int radius)
    {
      Radius = radius;
    }
  }
}
```

Für die Übersichtlichkeit sind die Klassen `Rechteck` und `Kreis` so stark vereinfacht, wie es nur möglich war.

Nach der Deserialisierung können wir nun auch den Inhalt der neuen `ArrayList` ausgeben, um zu prüfen, ob die Daten tatsächlich gespeichert worden sind.

```
foreach (object obj in neueBeispielListe)
{
  if (obj is Rechteck)
  {
    Console.WriteLine("Rechteck mit Breite " + ((Rechteck)obj).Breite);
  }
  else if (obj is Kreis)
  {
    Console.WriteLine("Kreis mit Radius " + ((Kreis)obj).Radius);
  }
}
```

Sobald wir unterschiedliche Objekttypen serialisiert haben, kann die Verwendung des Operators `is` Sinn ergeben. Dieser überprüft, ob das Objekt einem angegebenen Typ angehört.

Wenn wir dann auf Felder der serialisierten Objekte zugreifen möchten, dürfen wir nicht vergessen, eine Konvertierung durchzuführen, da alle Elemente einer `ArrayList` vom Typ *object* sind.

1.5 Serialisierung von Feldern verhindern

Sobald wir eine Klasse mit dem Attribut `Serializable` kennzeichnen, sind alle darin enthaltenen Daten (außer statische Felder und lokale Variablen) Teil des

Serialisierungsprozesses.

Wenn man nun bestimmte Felder nicht serialisieren möchte, kann man diese mit dem Attribut `NonSerialized` kennzeichnen. Das folgende Beispiel zeigt, wie das funktioniert.

```
[Serializable()]
class Rechteck
{
  public int Breite { get; set; }

  [NonSerialized()]
  private int Flaeche;

  public Rechteck(int breite)
  {
    Breite = breite;
  }
}
```

1.6 Serialisierung einer abgeleiteten Klasse

Wenn wir eine abgeleitete Klasse serialisieren möchten, dann müssen sowohl Basisklasse als auch abgeleitete Klasse mit dem Attribut `Serializable` gekennzeichnet sein.

Verweise

- »Rheinwerk Computing :: Visual C# 2010 - 13 Binäre Serialisierung«, `http://openbook.rheinwerk-verlag.de/visual_csharp_2010/visual_csharp_2010_13_001.htm#mj6daa9120efc53601bc38de9631d5d353`, 01.10.2017
- »Rheinwerk Computing :: Visual C# 2010 - 13.2 Serialisierung mit ›BinaryFormatter‹«, `http://openbook.rheinwerk-verlag.de/visual_csharp_2010/visual_csharp_2010_13_002.htm`, 01.10.2017