Урок №9
Сериализация объектов

1. Понятие атрибутов.

Перед знакомством с темой “Сериализация объектов” необходимо познакомиться с понятием атрибутов. Атрибуты используются при создании классов, которые могут быть сериализированы.

Компилятор .NET при компиляции компоновочного блока генерирует метаданные для всех типов, использующихся в сборке. Кроме этих стандартных метаданных платформа .NET дает возможность программисту встроить в компоновочный блок дополнительные метаданные, используюя атрибуты. Атрибуты представляют собой аннотации программного кода, которые могут применяться к заданному типу (классу, интерфейсу, структуре и т.д.), а так же к полям типов. Сами атрибуты в платформе .NET являются типами (классами) расширяющими абстрактный базовый класс System.Attribute. Следует понимать, что при использовании атрибутов в программном коде, метаданные остаются не используемыми до тех пор пока другой фрагмент программного кода не будет явно использовать отображение данной информации.

Безусловно, в платформе .NET существует целый ряд атрибутов, использование которых дает возможность получать определенные результаты.

Но чтобы получить представление о том, что собой представляет программирование с помощью атрибутов, рассмотрим следующий пример:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleAppAttributes
{
    /*
    ;
    ; Интерфейс "Человек"
    ; ------------------------------------------------------------------
    */
    interface IHuman
    {
        string getFName { get; }            // Получить Фамилию Человека
        string getSName { get; }            // Получить Имя Человека
    }

    /*
    ;
    ; Пользовательский (Наш) атрибут трансформации текста
    ; в верхний/нижний регистр
    ; ------------------------------------------------------------------
    */
    public sealed class TextTransformAttribute : System.Attribute
    {
        public bool isUpperCase;            // true - в верхний регистр,
                                            // false - в нижний
    }

    /*
    ;
    ; Класс "Простой человек"
    ; Ничего сложного - просто хранит Фамилию и Имя
    ; ------------------------------------------------------------------
    */
```

```
class SimpleHuman : IHuman
{
	string FName, SName;

	public SimpleHuman(string FNm, string SNm)
	{
		FName = FNm;
		SName = SNm;
	}

	public string getFName
	{
		get { return FName; }
	}

	public string getSName
	{
		get { return SName; }
	}
}

/*
;
; Класс "Студент"
; Также хранит Фамилию и Имя
; С помощью атрибута TextTransform формируем метаданные
; с информацией о том, что Фамилия и Имя должны переводиться
; в верхний регистр.
; ------------------------------------------------------------------
*/
[TextTransform(isUpperCase = true)]
class Student : IHuman
{
	string FName, SName;

	public Student(string FNm, string SNm)
	{
		FName = FNm;
		SName = SNm;
	}

	public string getFName
	{
		get { return FName; }
	}

	public string getSName
	{
		get { return SName; }
	}
}

/*
;
; Класс HumanPrinter - выводит в консоль Фамилию и Имя
; объектов, реализующих IHuman.
; При выводе поисходит проверка на необходимость трансформации
; выводимого текста в верхний/нижний регистр
; ------------------------------------------------------------------
*/
class HumanPrinter
{
	public void Show(IHuman H)
	{
```

```
            string firstName = H.getFName;
            string surrName = H.getSName;

        // ----- Проверка, на атрибут TextTransform ---------------------
            Type T = H.GetType();
            TextTransformAttribute[] TTA =
                        (TextTransformAttribute[])T.GetCustomAttributes(
                        typeof(TextTransformAttribute), false);

            if (TTA.Length != 0)
            {
        // ----- TextTransform обнаружен, переводим в нужный регистр ----
                firstName = (TTA[0].isUpperCase) ?
                        firstName.ToUpper() : firstName.ToLower();
                surrName = (TTA[0].isUpperCase) ?
                            surrName.ToUpper() : surrName.ToLower();
            }

        // ----- Вывод на экран -----------------------------------------
            Console.WriteLine("----------");
            Console.WriteLine("Фамилия  : {0}", firstName);
            Console.WriteLine("Имя      : {0}", surrName);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            SimpleHuman Petr = new SimpleHuman("Петр", "Иванов");
            Student     Ivan = new Student("Иван", "Немиров");

            HumanPrinter      HP = new HumanPrinter();
                HP.Show(Petr);          // Объект Petr выводится без
                                // трансформации текста
            HP.Show(Ivan);           // Объект Ivan выводится с
                                // трансформацией текста

        }
    }
}
```

Листинг 1.1. Пример программирования с атрибутами.

Результат работы программы:

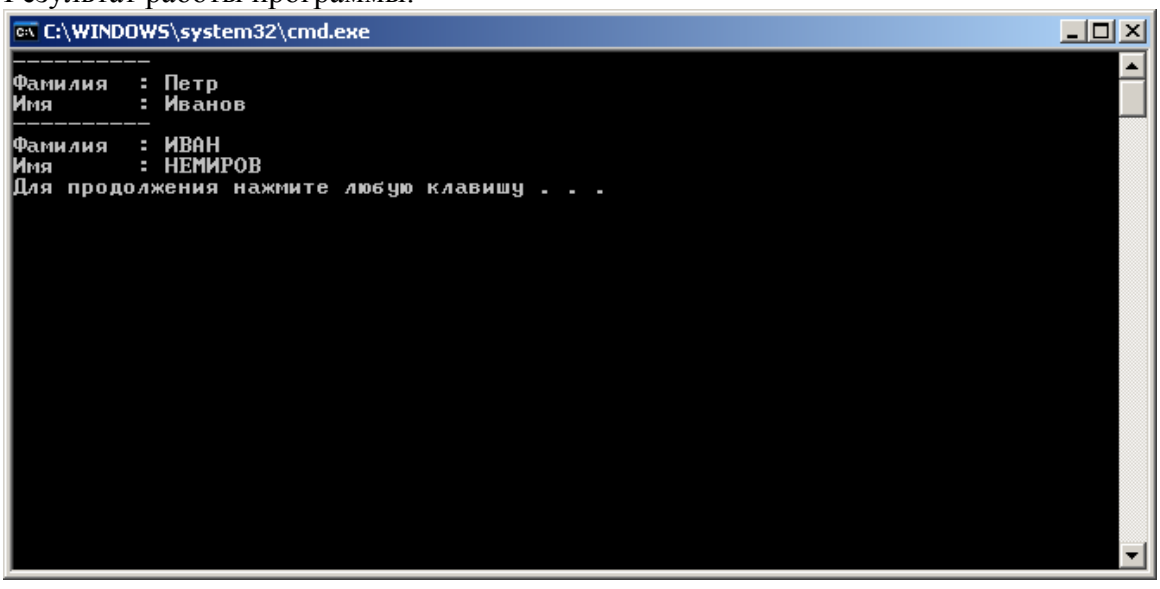

Рис. 1.1 Результат работы примера.

2. Что такое сериализация?

Сериализация это сохранение состояния объекта в байтовый поток, с целью его (объекта) последующего восстановления. Сохраненная последовательность байт содержит всю необходимую информацию для восстановления объекта.

С помощью сериализации очень просто сохранять огромные объемы данных (в самых разных форматах). Во многих случаях сохранение данных приложения с помощью сервиса сериализации оказывается намного удобным, чем прямое использование средств чтения/записи, предлагаемых в рамках пространства имен System.IO.

Использование сериализации также играет ключевую роль при копировании объекта на удаленный компьютер:

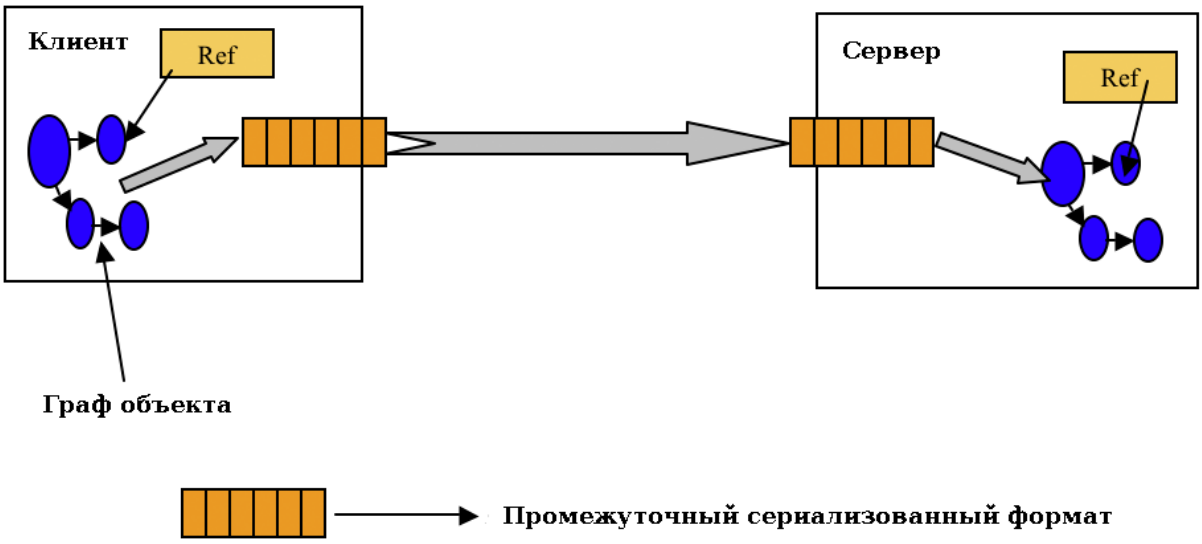

Рис. 2.1. Передача сериализированного объекта на удаленный компьютер.

Итак, в чем же все-таки проявляются преимущества сериализации? Ведь если рассматривать задачу сохранения состояния объекта, то можно обойтись и без сериализации, а воспользоваться, например, экземпляром System.IO.BinaryWriter. Хорошо. Давайте попробуем сохранить в файл экземпляр класса Auto:

```
class Auto
{
    private     int         speed;      // Скорость движения автомобиля
    private     double      weight;     // Перевозимый автомобилем груз
    private     string      marka;      // Марка автомобиля

    public Auto(int S, double W, string M)
    {
          speed       = S;
          weight      = W;
          marka       = M;
    }

    ...
}
```

```
class Program
{
     static      void Main()
     {
           Auto              A     = new Auto(100, 350, "Opel");
           BinaryWriter      BW    = new BinaryWriter(
                                       new FileStream("auto.dat",
                                     FileMode.Create, FileAccess.Write));

           BW.Write(A);              // Ошибка компиляции!!!
                                  // Нет метода Write(Object)
           BW.Write(A.speed);        // Ошибка компиляции!!!
                                  // Нет доступа к закрытому полю класса
     }
}
```

Листинг 2.1. Попытка сохранения состояния объекта в файловый поток.

Итак, первая попытка не удалась. Конечно же, можно открыть поля класса Auto и тогда можно сохранить объект путем сохранения по отдельности каждого его поля. Но при этом мы нарушим принцип инкапсуляции, что с точки зрения правильного подхода к программированию не является корректным.

Хорошо, давайте не нарушая принципа инкапсуляции, добавим в класс Auto необходимый функционал, который позволит сохранить объект этого класса в поток:

```
class Auto
{
     private     int         speed;      // Скорость движения автомобиля
     private     double      weight;     // Перевозимый автомобилем груз
     private     string      marka;      // Марка автомобиля

     public Auto(int S, double W, string M)
     {
           speed       = S;
           weight      = W;
           marka       = M;
     }

     public Auto()
     {
     }

     public override string ToString()
     {
           return string.Format(
                        "Скорость : {0}км/ч, Груз : {1}кг, Марка : {2}",
                      speed, weight, marka);
     }

     public void Write(BinaryWriter BW)
     {
           BW.Write(speed);
           BW.Write(weight);
           BW.Write(marka);
     }

     public void Read(BinaryReader BR)
     {
           speed       = BR.ReadInt32();
```

```
            weight      = BR.ReadDouble();
            marka       = BR.ReadString();
      }

      ...
}

class Program
{
      static      void Main()
      {
            Auto              A     = new Auto(100, 350, "Opel");

      // ----- Сохраняем ---------------------------------------------
                                      BinaryWriter     BW    = new
                                    BinaryWriter(
                                      new FileStream("auto.dat",
                           FileMode.Create, FileAccess.Write));
            A.Write(BW);
            BW.Close();

      // ----- Восстанавливаем ранее сохраненный объект ----------------
            Auto              B     = new Auto();
                  BinaryReader      BR    = new BinaryReader(
                                      new FileStream("auto.dat",
                                      FileMode.Open, FileAccess.Read));
            B.Read(BR);
            BR.Close();

      // ----- Убеждаемся в правильности сохранения/восстановления -------
            Console.WriteLine(B);
      }
}
```

Листниг 2.2. Удачное сохранение состояние объекта в файловый поток.

Теперь мы можем смело и самостоятельно, без использования сериализации сохранять/восстанавливать состояние объекта в поток. Но только при условии, что у объекта есть методы, подобные методам Read/Write класса Auto. Но таких методов в подавляющем большинстве случаев у классов нет. И вот здесь на помощь приходит сериализация.

Простой пример сохранения/восстановления состояния объекта в поток с использованием механизма сериализации (обратите внимание, что в этом примере в классе Auto нет методов Read/Write):

```
[Serializable]
class Auto
{
      private     int         speed;      // Скорость движения автомобиля
      private     double      weight;     // Перевозимый автомобилем груз
      private     string      marka;      // Марка автомобиля

      public Auto(int S, double W, string M)
      {
            speed       = S;
            weight      = W;
            marka       = M;
      }
```

```
        public Auto()
        {
        }

        public override string ToString()
        {
              return string.Format(
                           "Скорость : {0}км/ч, Груз : {1}кг, Марка : {2}",
                           speed, weight, marka);
        }
        ...

}

class Program
{
        static      void Main()
        {
              Auto        A     = new Auto(100, 350, "Opel");
              FileStream  FS    = new FileStream("auto.dat",
                                  FileMode.Create, FileAccess.ReadWrite);

        // ----- Сериализируем объект Auto в байтовый файловый поток -------
              BinaryFormatter   BF    = new BinaryFormatter();
              BF.Serialize(FS, A);

        // ----- Десериализируем объект Auto из потока ---------------------
              FS.Seek(0, SeekOrigin.Begin);
              Auto        B     = (Auto)BF.Deserialize(FS);
              FS.close();

        // ----- Убеждаемся в правильности примера -------------------------
              Console.WriteLine(B);

        }
}
```

Листинг 2.3. Пример сохранения состояния объекта в файловый поток с помощью сериализации.

Как видите из примера, использование механизма сериализации удобно и не сложно.

3. Отношения между объектами.

Как вы уже убедились, сохранять объекты с помощью средств сериализации .NET очень просто. Однако следует понимать, что процессы, происходящие при этом в фоновом режиме, являются достаточно сложными сложными. Например, когда объект сохраняется в потоке, все соответствующие данные (базовые классы, вложенные объекты и т.п.) автоматически сохраняются тоже. Таким образом, при попытке выполнить сериализацию производного класса в этом процессе будет задействована вся цепочка наследования. Как вы сможете убедиться позже, множество взаимосвязанных объектов можно представить в виде объектного графа (см. следующий раздел). Сервис сериализации .NET позволяет сохранить и объектный граф, причем в самых разных форматах. В предыдущем примере программного кода использовался тип BinaryFormatter, поэтому состояние объекта Auto s сохранялось в компактном двоичном формате. Если использовать другие типы, то объектный граф можно сохранить в формате SOAP (Simple Object Access Protocol —

простой протокол доступа к объектам) или в формате XML. Эти форматы могут быть полезны тогда, когда необходимо гарантировать, что ваши сохраненные объекты легко смогут быть перенесены на другие компьютеры с другими операционными системами, языками и архитектурами. Наконец, следует понимать, что объектный граф можно сохранить в любом производном от System.I0.Stream типе. В предыдущем примере объект Auto сохранялся в локальном файле с помощью типа FileStream. Но если бы требовалось сохранить объект в памяти, следовало бы использовать тип MemoryStream. Это необходимо для того, чтобы последовательность данных корректно представляла состояния объектов соответствующего графа.

4. Графы отношений объектов.

Как уже упоминалось, при сериализации объекта среда CLR учитывает состояния всех связанных объектов. Множество связанных объектов представляется объектным графом. Объектные графы обеспечивают простой способ учета взаимных связей (отношений) в множестве объектов. Каждому объекту в объектном графе назначается средой CLR уникальное числовое значение. Эти числовые значения, приписываемые членам в объектном графе, произвольны и не имеют никакого смысла вне графа. После назначения всем объектам числового значения объектный граф может начать запись множества зависимостей каждого объекта.

Для примера предположим, что мы создали множество классов, моделирующих типы автомобилей. У нас есть базовый класс, названный Auto (автомобиль), который имеет радио (Radio). Другой класс, SportAuto (спортивный автомобиль), расширяет базовый тип Auto. На рис. 4.1 показан возможный объектный граф. моделирующий указанные взаимосвязи.

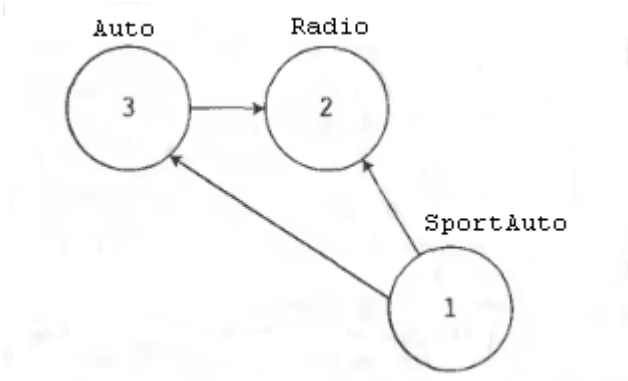

Рис. 4.1. Пример графа отношения объектов.

Взаимосвязи, указанные в диаграмме, представляются средой CLR математической формулой, которая выглядит примерно так.

```
[Auto 3, ref 2], [Radio 2], [SportAuto 1, ref 3, ref 2]
```

Проанализировав эту формулу, вы снова увидите, что объект 3 (Auto) имеет зависимость в отношении объекта 2 (Radio). Объект 2 (Radio) является объектом, которому никто не требуется. Наконец, объект 1 (SportAuto) имеет зависимость в отношении как объекта 3, так и объекта 2.

Когда выполняется сериализация или реконструкция экземпляра SportAuto, объектный граф дает гарантию того, что типы Radio и Auto тоже будут участвовать в процессе.

Особенностью процесса сериализации является то, что граф, изображающий взаимосвязи объектов, создается автоматически в фоновом режиме. При желании мы можем участвовать в построении объектного графа.

5. Атрибуты для сериализации [Serializable] и [NonSerialized].

Чтобы сделать объект доступным сервису сериализации .NET, достаточно пометить каждый связанный класс атрибутом [Serializable].

Если необходимо, чтобы некоторые члены данного класса не должны (или не могут) участвовать в процессе сериализации, то нужно обозначить соответствующие поля атрибутом [NonSerialized]. Это может быть полезно тогда, когда в классе, предназначенном для сериализации, есть члены-переменные, которые запоминать не нужно.

В примере ниже приведен класс MP3Song (класс «музыкальная композиция» в формате mp3), обозначенный атрибутом [Serializable]. Поля этого класса (название, исполнитель, битрейт, размер файла, продолжительность, url и т.д.) будут сериализированы, за исключением поля URL, поскольку это поле помечено атрибутом [NonSerialized]:

```
[Serializable]
class MP3Song
{
    string            Title;      // Название песни
    string            Author;     // Исполнитель
    int               BitRate;    // Скорость потока данных
                                  // при воспроизведении
    int               FileSize;   // Размер файла в байтах
    double            Duration;   // Продолжительность

    [NonSerialized]
    string            URL;        // URL подробной информации о песне
    ...
}
```

Листинг 5.1. Пример использования атрибутов [Serializable] и [NonSerialized]

Следует знать, что атрибут [Serializable] не наследуется. Таким образом, если вы получаете класс из типа, обозначенного атрибутом [Serializable], дочерний класс тоже следует обозначить атрибутом [Serializable], иначе он при сериализации сохраняться не будет. На самом деле все объекты в объектном графе должны обозначаться атрибутом [Serializable]. При попытке выполнить сериализацию объекта, не подлежащего сериализации, в среде выполнения генерируется исключение SerializationException.

6. Форматы сериализации.

6.1 Форматы сериализации.

Платформа .NET предоставляет возможность осуществлять сериализацию в трех различных форматах. Каждому формату сериализации соответствует свой класс .NET:

- Сериализация в двоичный формат. Осуществляется объектами класса BinaryFormatter, который находится в пространстве имен System.Runtime.Serialization.Formatters.Binary

- Сериализация в формат xml. Осуществляется объектами класса XmlSerializer, который находится в пространстве имен System.Xml.Serialization

- Сериализация в формат SOAP (Simple Object Access Protocol – простой протокол доступа к объекту). Осуществляется объектами класса SoapFormatter, который находится в пространстве имен System.Runtime.Serialization.Formatters.Soap

Сутью различий указанных трех форматов является то, как именно объектный граф переводится в поток (в двоичном формате, формате SOAP или XML). Также сущесвуют и другие отличия между перечисленными выше форматами сериализации.

При использовании BinaryFormatter будут сохраняться не только поля данных объектов из объектного графа, но и абсолютное имя каждого типа, а также полное имя определяющего тип компоновочного блока. Это очень удобно как в случае, когда необходимо передать значение объектов на другой компьютер, или в случае, если необходимо сохранить состояние объектов в байтовый поток. BinaryFormatter сериализирует как открытые, так и закрытые поля типа.

Если BinaryFormatter сохраняет тип абсолюно точно, то SoapFormatter и XmlSerializer, с другой стороны, не являются форматами сериализации, которые сохраняют тип точно - они не записывают абсолютные имена типов и компоновочных блоков. Эти форматы сериалихации предназначены для сохранения состояния объектов так, чтобы они могли использоваться в любой операционной системе для любого каркаса приложений (.NET, Java, QT и т.д.), в любом языке программирования. Поэтому нет необходимости поддерживать абсолютную точность для типа, поскольку нет гарантии, что все возможные получатели смогут понять типы данных, специфичные для .NET.

6.2 Двоичное форматирование. Класс BinaryFormatter.

В классе BinaryFormatter определены два метода, с помощью которых и выполняется сериализация/десериализация объектных графов в двоичный поток:

```
public void Serialize (Stream serializationStream, Object graph);
```

Сохраняет объектный граф graph в указанном потоке serializationStream в виде последовательности байтов.

```
public Object Deserialize (Stream serializationStream);
```

Преобразует сохраненную последовательность байтов из потока serializationStream в объектный граф.

BinaryFormatter сериализирует как открытые, так и закрытые поля и свойства объектов.

Рассмотрим пример:

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

namespace WinAppTest
{

      /*
      ;
      ; Класс Деньги - инкапсулирует в себе денежную сумму.
      ; ------------------------------------------------------------
      */
```

```
[Serializable]
class Money
{
      private     int   grn;        // Гривны
      private     int   kop;        // Копейки

      public Money(int G, int K)
      {
            kop   = K % 100;
            grn   = G + K / 100;
      }

      public override string ToString()
      {
            return string.Format("{0}.{1}грн", grn, kop);
      }
}

/*
;
; Класс "шоколадний батончик"
; ------------------------------------------------------------------
*/
[Serializable]
class ChocoTab
{
      private     DateTime    Expire;     // Срок годности батона
                                          // Кстати - DateTime объявлен
                                          // с атрибутом [Serializable],
                                          // а значит сериализации
                                          // подлежит

      private     String      Name;       // Название батончика
      private     Money       Price;      // Стоимость
      private     double      Weight;     // Вес батончика

      public ChocoTab(String N, double W, Money P, DateTime E)
      {
            Name        = N;
            Weight      = W;
            Price       = P;
            Expire      = E;
      }

      public override string ToString()
      {
            return string.Format(
                        "Батончик {0}, весом {1}г, " +
                        "стоимостью {2}, нужно съесть до {3}",
                        Name, Weight, Price, Expire);
      }
}


class Program
{
      static void Main(string[] args)
      {

// ----- Объект, который будем сериализировать ------------------
            ChocoTab    Snickers    = new ChocoTab("Snickers", 100.00,
                              new Money(5, 70),
                              new DateTime(2010, 12, 31, 23, 59, 59));
```

```
            // ----- Поток, в который сериализируем -------------------------
                        FileStream        FS          = new FileStream("choco.dat",
                                          FileMode.Create, FileAccess.Write);

            // ----- Создаем объект двоичной сериализации -------------------
                        BinaryFormatter   BF          = new BinaryFormatter();

            // ----- Сериализируем ------------------------------------------
                        BF.Serialize(FS, Snickers);

            // ----- Закрываем поток ----------------------------------------
                        FS.Close();

            // ----- Теперь восстановим сохраненное состояние объекта -------
                        ChocoTab    RealSnick   = null;

            // ----- Поток, из которого десериализируем ---------------------
                        FS                =     new FileStream("choco.dat",
                                          FileMode.Open, FileAccess.Read);

            // ----- Десериализация здесь -----------------------------------
                        RealSnick   = (ChocoTab)BF.Deserialize(FS);
                        FS.Close();

            // ----- Посмотрим что получилось -------------------------------
                        Console.WriteLine(RealSnick);
                  }
            }
}
```

Листинг 6.2.1. Пример сериализации графа объектов в двоичный формат.

В результате исполнения данного примера увидим на экране:

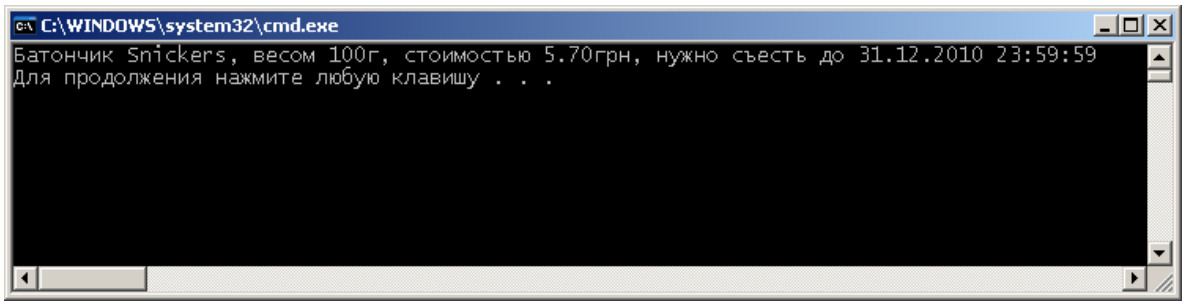

Рис. 6.2.1. Результат работы примера

Можно увидеть двоичные данные, сформированные BinaryFormatter'ом. Обратите внимание, что, как уже говорилось выше, при сериализации в двоичный формат, осуществляется сохранение не только значений полей объекта, а и информации о типа этих полей, названии, названии класса, пространства имен, сборки. Так же сериализируются и все связанные с сериализируемым объектом объекты. В нашем примере это объект экземпляр Money (стоимость). Двоичные данные, полученные в результате работы примера можно увидеть на рисунке 6.2.2:

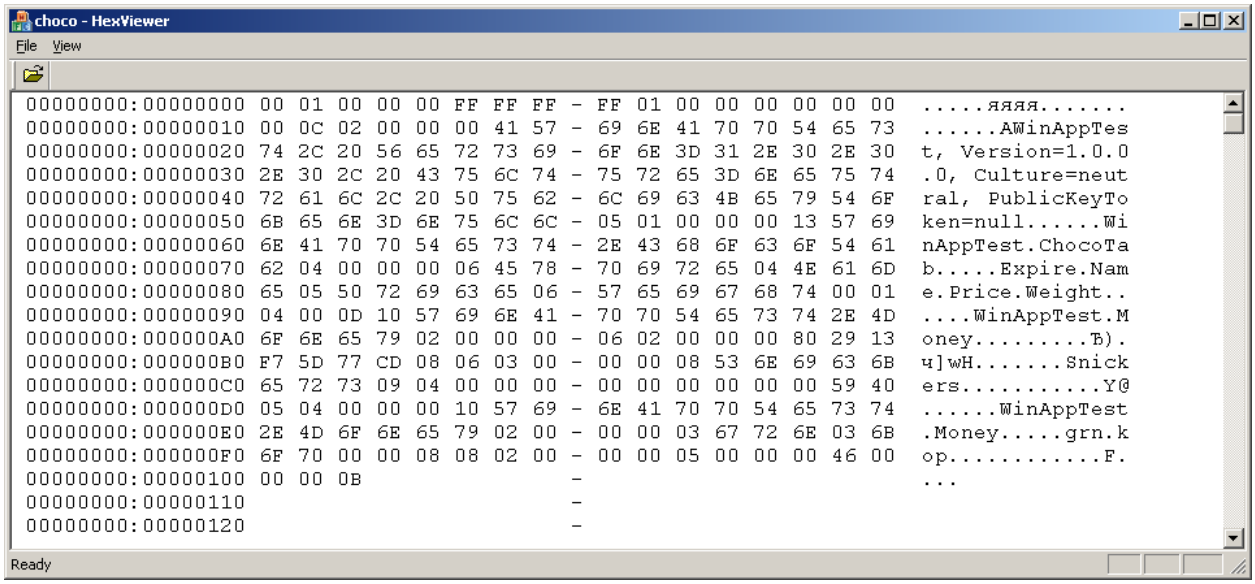

Рис. 6.2.2. Образ двоичного формата сериализации

В данном примере, сериализация выполнялась в байтовый файловый поток. Однако, при необходимости, можно выполнять сериализацию, например, в память. Для сериализации в память нужно использовать объект класса MemoryStream. Классы FileStream и MeoryStream являются классами, производными от System.IO.Stream, а в .NET сериализация возможна в любой байтовый поток, производный от System.IO.Stream.

Тот же пример, но сериализация выполняется в память:

```
class Program
{
      static void Main(string[] args)
      {
            ChocoTab          Snickers    = new ChocoTab(
                                    "Snickers", 100.00,
                                    new Money(5, 70),
                                    new DateTime(2010, 12, 31, 23, 59, 59));

            MemoryStream            MS          = new MemoryStream();
            BinaryFormatter         BF          = new BinaryFormatter();

            BF.Serialize(MS, Snickers);

// ----- Для десериализации вернемся в начало байтового потока --
            MS.Seek(0, SeekOrigin.Begin);

            ChocoTab    RealSnick   = null;
            RealSnick               = (ChocoTab)BF.Deserialize(MS);
            MS.Close();

      // ----- Посмотрим что получилось -------------------------------
            Console.WriteLine(RealSnick);
      }
}
```

Листинг 6.2.2. Пример двоичной сериализации в байтовый поток, находящийся в памяти

Результат исполнения этого примера будет такой же (см. рис. 6.2.1).

6.3 Soap форматирование. Класс SoapFormatter.

SoapFormatter сохраняет объектный граф в сообщении SOAP (Simple Object Access Protocol — простой протокол доступа к объектам), что делает этот вариант форматирования прекрасным выбором при передаче объектов средствами удаленного взаимодействия по протоколу HTTP. В сущности. SOAP определяет стандартный процесс, с помощью которого можно вызывать методы не зависящим от платформы и ОС способом для Web-сервисов XML.

Чтобы выполнить сериализацию / десериализацию объекта в формате SOAP, нужно выполнить те же действия, что и для двоичной сериализации (то есть, нет никаких отличий между двоичной сериализацией и SOAP сериализацией в требованиях к созданию класса, объекты которого будут сериализироваться).

Пример сериализации в SOAP:

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

namespace WinAppTest
{
      …
      class Program
      {
            static void Main(string[] args)
            {
                  ChocoTab          Snickers    = new ChocoTab(
                                    "Snickers", 100.00,
                                    new Money(5, 70),
                                    new DateTime(2010, 12, 31, 23, 59, 59));

                  FileStream              FS    = new FileStream("choco.xml",
                                    FileMode.Create,
                                    FileAccess.ReadWrite);

                  SoapFormatter           SF    = new SoapFormatter();

                  SF.Serialize(FS, Snickers);

      // ----- Для десериализации вернемся в начало байтового потока --
                  FS.Seek(0, SeekOrigin.Begin);

                  ChocoTab    RealSnick   = null;
                  RealSnick                   =
(ChocoTab)SF.Deserialize(FS);
                  FS.Close();

      // ----- Посмотрим что получилось ------------------------------
                  Console.WriteLine(RealSnick);
            }
      }
}
```

Листинг 6.3.1. Сериализация в SOAP-формат

В приведенном выше примере, сериализация осуществлялась в файл choco.xml. Почему у файла расширение xml? Потому что SOAP является xml-языком.

Посмотрим в файл choco.xml, на результаты сериализации (рис 6.3.1):

```
<SOAP-ENV:Envelope
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <a1:ChocoTab id="ref-1"
    xmlns:a1="http://schemas.microsoft.com/clr/nsassem/WinAppTest/WinAppTest
    %2C%20Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToke
    n%3Dnull">
                <Expire>2010-12-31T23:59:59.0000000+02:00</Expire>
                <Name id="ref-3">Snickers</Name>
                <Price href="#ref-4" />
                <Weight>100</Weight>
        </a1:ChocoTab>
        <a1:Money id="ref-4"
    xmlns:a1="http://schemas.microsoft.com/clr/nsassem/WinAppTest/WinAppTest
    %2C%20Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToke
    n%3Dnull">
                <grn>5</grn>
                <kop>70</kop>
                </a1:Money>
    </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

Рис. 6.3.1. Образ SOAP-сериализации

Домашнее задание:

Модифицировать консольную игру "21" (создание игры было заданием одной из предыдущих домашних работ) таким образом, чтобы в программе была возможность вести одновременно 10 игр (предположим, что это небольшое казино на 10 столов). В программе необходимо реализовать класс "Игра" (или "Стол"). В программе должен быть создан всего лишь один объект "Игра" на все 10 одновременно ведущихся игр. Текущее состояние каждой игры необходимо сохранять с помощью сериализации в обьекты MemoryStream. Процесс переключения между играми состоит в сериализации текущей игры в память, и десериализации игры, к которой происходит переключение, из памяти в объект "Игра". Переключение между играми осуществлять с помощью клавиш Alt+Fномер_игры.