```csharp
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Windows.Forms;
using System.Xml;
using HelloWorldApp.BusinessObjects;
using Polenter.Serialization;


namespace HelloWorldApp
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void serializeXmlButton_Click(object sender, EventArgs e)
        {
            // create fake obj
            var obj = RootContainer.CreateFakeRoot();

            // create instance of sharpSerializer
            // with the standard constructor it serializes to xml
            var serializer = new SharpSerializer();


            // *************************************************************************************
            // For advanced serialization you create SharpSerializer with an overloaded constructor
            //
            //  SharpSerializerXmlSettings settings = createXmlSettings();
            //  serializer = new SharpSerializer(settings);
            //
            // Scroll the page to the createXmlSettings() method for more details
            // *************************************************************************************


            // *************************************************************************************
            // You can alter the SharpSerializer with its settings, you can provide your custom readers
            // and writers as well, to serialize data into Json or other formats.
            //
            // var serializer = createSerializerWithCustomReaderAndWriter();
            //
            // Scroll the page to the createSerializerWithCustomReaderAndWriter() method for more details
            // *************************************************************************************


            // set the filename
            var filename = "sharpSerializerExample.xml";

            // serialize
            serialize(obj, serializer, filename);
        }


        private void serialize(object obj, SharpSerializer serializer, string shortFilename)
        {
            // Serializing the first object
            var file1 = getFullFilename(shortFilename, "1");
            serializer.Serialize(obj, file1);

            // Deserializing to a second object
            var obj2 = serializer.Deserialize(file1);

            // Serializing the second object
            var file2 = getFullFilename(shortFilename, "2");
            serializer.Serialize(obj2, file2);

            // Comparing two files
            compareTwoFiles(file1, file2);

            // Show files in explorer
            showInExplorer(file1);
        }

        private void compareTwoFiles(string file1, string file2)
        {
            // comparing
            var fileInfo1 = new FileInfo(file1);
            var fileInfo2 = new FileInfo(file2);


            if (fileInfo1.Length > 0 && fileInfo1.Length == fileInfo2.Length)
            {
                byte[] content1 = File.ReadAllBytes(file1);
                byte[] content2 = File.ReadAllBytes(file2);

                for(int i = 0; i < content1.Length; i++)
                    if (content1[i] != content2[i])
                    {
                        MessageBox.Show(string.Format("Files differ at offset {0}", i));
                        return;
                    }

                MessageBox.Show(string.Format("Both files have the same length of {0} bytes and the same content", fileInfo1.Length));
            }
            else
            {
                MessageBox.Show(string.Format("Length of file1: {0}, Length of file2: {1}", fileInfo1.Length,
                                              fileInfo2.Length));
            }
        }

        private void showInExplorer(string filename)
        {
            if (!string.IsNullOrEmpty(filename) && File.Exists(filename))
            {
                string arguments = string.Format("/n, /select, \"{0}\"", filename);
                Process.Start("explorer", arguments);
            }
        }

        private static string getFullFilename(string shortFilename, string nameSufix)
        {
            var folder = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
            var filenameWithoutExtension = string.Format("{0}{1}", Path.GetFileNameWithoutExtension(shortFilename),
                                                         nameSufix);
            var filenameWithExtension = Path.ChangeExtension(filenameWithoutExtension, Path.GetExtension(shortFilename));
            return Path.Combine(folder, filenameWithExtension);
        }


        private SharpSerializerXmlSettings createXmlSettings()
        {
            // create the settings instance
            var settings = new SharpSerializerXmlSettings();

            // Bare instance of SharpSerializerXmlSettings is enough for SharpSerializer to know,
            // it should serialize data as xml.

            // However there is more you can influence.


            // Culture
            // All float numbers and date/time values are serialized as text according to the Culture.
            // The default Culture is InvariantCulture but you can override this settings with your own culture.
            settings.Culture = System.Globalization.CultureInfo.CurrentCulture;


            // Encoding
            // Default Encoding is UTF8. Encoding impacts the format in which the whole Xml file is stored.
            settings.Encoding = System.Text.Encoding.ASCII;


            // AssemblyQualifiedName
            // During serialization all types must be converted to strings.
            // Since v.2.12 the type is stored as an AssemblyQualifiedName per default.
            // You can force the SharpSerializer to shorten the type descriptions
            // by setting the following properties to false
            // Example of AssemblyQualifiedName:
            // "System.String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
            // Example of the short type name:
            // "System.String, mscorlib"
            settings.IncludeAssemblyVersionInTypeName = true;
            settings.IncludeCultureInTypeName = true;
            settings.IncludePublicKeyTokenInTypeName = true;



            // ADVANCED SETTINGS
            // Most of the classes needed to alter these settings are in the namespace Polenter.Serialization.Advanced


            // PropertiesToIgnore
            // Sometimes you want to ignore some properties during the serialization.
            // If they are parts of your own business objects, you can mark these properties with ExcludeFromSerializationAttribute.
            // However it is not possible to mark them in the built in .NET classes
            // In such a case you add these properties to the list PropertiesToIgnore.
            // I.e. System.Collections.Generic.List<string> has the "Capacity" property which is irrelevant for
            // the whole Serialization and should be ignored
            // serializer.PropertyProvider.PropertiesToIgnore.Add(typeof(List<string>), "Capacity")
            settings.AdvancedSettings.PropertiesToIgnore.Add(typeof (List<string>), "Capacity");


            // PropertyTypesToIgnore
            // Sometimes you want to ignore some types during the serialization.
            // To ignore a type add these types to the list PropertyTypesToIgnore.
            settings.AdvancedSettings.PropertyTypesToIgnore.Add(typeof(List<string>));


            // RootName
            // There is always a root element during serialization. Default name of this element is "Root",
            // but you can change it to any other text.
            settings.AdvancedSettings.RootName = "MyFunnyClass";


            // SimpleValueConverter
            // During xml serialization all simple values are converted to their string representation.
            // Float values, DateTime are default converted to format of the settings.Culture or CultureInfo.InvariantCulture
            // if the settings.Culture is not set.
            // If you want to store these values in your own format (Morse alphabet?) create your own converter as an instance of ISimpleValueConverter.
            // Important! This setting overrides the settings.Culture
            settings.AdvancedSettings.SimpleValueConverter = new MyCustomSimpleValueConverter();



            // TypeNameConverter
            // Since the v.2.12 all types are serialized as AssemblyQualifiedName.
            // To change this you can alter the settings above (Include...) or create your own instance of ITypeNameConverter.
            // Important! This property overrides the three properties below/above:
            //    IncludeAssemblyVersionInTypeName, IncludeCultureInTypeName, IncludePublicKeyTokenInTypeName
            settings.AdvancedSettings.TypeNameConverter = new MyTypeNameConverterWithCompressedTypeNames();


            return settings;
        }

        private SharpSerializerBinarySettings createBinarySettings()
        {
            // create the settings instance
            var settings = new SharpSerializerBinarySettings();

            // bare instance of SharpSerializerBinarySettings tells SharpSerializer to serialize data into binary format in the SizeOptimized mode

            // However there is more possibility to influence the serialization


            // Encoding
            // Default Encoding is UTF8.
            // Changing of Encoding has impact on format in which are all strings stored (type names, property names and string values)
            settings.Encoding = System.Text.Encoding.ASCII;


            // AssemblyQualifiedName
            // During serialization all types must be converted to strings.
            // Since v.2.12 the type is stored as an AssemblyQualifiedName per default.
            // You can force the SharpSerializer to shorten the type descriptions
            // by setting the following properties to false
            // Example of AssemblyQualifiedName:
            // "System.String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
            // Example of the short type name:
            // "System.String, mscorlib"
            settings.IncludeAssemblyVersionInTypeName = true;
            settings.IncludeCultureInTypeName = true;
            settings.IncludePublicKeyTokenInTypeName = true;


            // Mode
            // The default mode, without altering the settings, is BinarySerializationMode.SizeOptimized
            // Actually you can choose another mode - BinarySerializationMode.Burst
            //
            // What is the difference?
            // To successfully restore the object tree from the serialized stream, all objects have to be serialized including their type information.
            // Both modes differ in the art the type information is stored.
            //
            // BinarySerializationMode.Burst
            // In the burst mode, type of every object is serialized as a string as part of this object.
            // It doesn't matter if all serialized objects are of the same type, their types are serialized as text as many times as many objects.
            // It increases the file size especially when serializing collections. Type information is duplicated.
            // It's ok for single, simple objects, as it has small overhead. BurstBinaryWriter supports this mode.
            //
            // BinarySerializationMode.SizeOptimized
            // In the SizeOptimized mode all types are grouped into a list. All type duplicates are removed.
            // Serialized objects refer only to this list using index of their type. It's recommended approach for serializing of complex
            // objects with many properties, or many items of the same type (collections). The drawback is - all objects are cached,
            // then their types are analysed, type list is created, objects are injected with indexes and finally the data is written.
            // Apart from types, all property names are handled the same way in the SizeOptimized mode.
            // SizeOptimizedBinaryWriter supports this mode.
            settings.Mode = BinarySerializationMode.SizeOptimized;



            // ADVANCED SETTINGS
            // Most of the classes needed to alter these settings are in the namespace Polenter.Serialization.Advanced


            // PropertiesToIgnore
            // Sometimes you want to ignore some properties during the serialization.
            // If they are parts of your own business objects, you can mark these properties with ExcludeFromSerializationAttribute.
            // However it is not possible to mark them in the built in .NET classes
            // In such a case you add these properties to the list PropertiesToIgnore.
            // I.e. System.Collections.Generic.List<string> has the "Capacity" property which is irrelevant for
            // the whole Serialization and should be ignored
            // serializer.PropertyProvider.PropertiesToIgnore.Add(typeof(List<string>), "Capacity")
            settings.AdvancedSettings.PropertiesToIgnore.Add(typeof(List<string>), "Capacity");


            // PropertyTypesToIgnore
            // Sometimes you want to ignore some types during the serialization.
            // To ignore a type add these types to the list PropertyTypesToIgnore.
            settings.AdvancedSettings.PropertyTypesToIgnore.Add(typeof(List<string>));


            // RootName
            // There is always a root element during the serialization. Default name of this element is "Root",
            // but you can change it to any other text.
            settings.AdvancedSettings.RootName = "MyFunnyClass";


            // TypeNameConverter
            // Since the v.2.12 all types are serialized as AssemblyQualifiedName.
            // To change this you can alter the settings above (Include...) or create your own instance of ITypeNameConverter.
            // Important! This property overrides the three properties below/above:
            //    IncludeAssemblyVersionInTypeName, IncludeCultureInTypeName, IncludePublicKeyTokenInTypeName
            settings.AdvancedSettings.TypeNameConverter = new MyTypeNameConverterWithCompressedTypeNames();


            return settings;
        }

        private SharpSerializer createSerializerWithCustomReaderAndWriter()
        {
            // ***************************************************************************************
            // SERIALIZATION
            // The namespace Polenter.Serialization.Advanced contains some classes which are indispensable during the serialization.


            // ***************************************************************************************
            // XmlPropertySerializer
            // serializes objects into elements and their attributes.
            // Each element has its begin and end tag.
            // XmlPropertySerializer self is not responsible for the serializing to xml, it doesn't reference the built in .NET XmlWriter.
            // Instead it uses an instance of IXmlWriter to control the element writing.
            // DefaultXmlWriter implements IXmlWriter and contains the build in .NET XmlWriter which is responsible for writing to the stream.

            // To make your own node oriented writer, you need to make a class which implements IXmlWriter
            Polenter.Serialization.Advanced.Xml.IXmlWriter jsonWriter = new MyJsonWriter();

            // this writer is passed to the constructor of the XmlPropertySerializer
            Polenter.Serialization.Advanced.Serializing.IPropertySerializer serializer =
                new Polenter.Serialization.Advanced.XmlPropertySerializer(jsonWriter);

            // in such a was, the default XmlPropertySerializer can store data in any format which is node oriented (contains begin/end tags)


            // ***************************************************************************************
            // BinaryPropertySerializer
            // serializes objects into elements which have known length and fixed position in the stream.
            // It doesn't write directly to the stream. Instead, it uses an instance of IBinaryWriter.
            // Actually there are two writers used by the SharpSerializer: BurstBinaryWriter and SizeOptimizedBinaryWriter

            // To make your own binary writer you make a class which implements the IBinaryWriter.
            Polenter.Serialization.Advanced.Binary.IBinaryWriter compressedWriter = new MyVeryStrongCompressedAndEncryptedBinaryWriter();

            // this writer is passed to the constructor of the BinaryPropertySerializer
            serializer = new Polenter.Serialization.Advanced.BinaryPropertySerializer(compressedWriter);

            // Changing only the writer and not the whole serializer allows an easy serialization of data to any binary format



            // ***************************************************************************************
            // DESERIALIZATION
            // The namespace Polenter.Serialization.Advanced contains classes which are counterparts of the above serializers/writers
            // XmlPropertySerializer -> XmlPropertyDeserializer
            // DefaultXmlWriter -> DefaultXmlReader
            // BurstBinaryWriter -> BurstBinaryReader
            // SizeOptimizedBinaryWriter -> SizeOptimizedBinaryReader

            // Deserializers are constructed analog or better say - symmetric to the Serializers/Writers, i.e.

            Polenter.Serialization.Advanced.Binary.IBinaryReader compressedReader =
                new MyVeryStrongCompressedAndEncryptedBinaryReader();

            Polenter.Serialization.Advanced.Deserializing.IPropertyDeserializer deserializer =
                new Polenter.Serialization.Advanced.BinaryPropertyDeserializer(compressedReader);

            // If you have created serializer and deserializer, the next step is to create SharpSerializer.


            // ************************************************************************************
            // Creating SharpSerializer
            // Both classes - serializer and deserializer are passed to the overloaded constructor

            var sharpSerializer = new SharpSerializer(serializer, deserializer);


            // there is one more option you can alter directly on your instance of SharpSerializer

            // ************************************************************************************
            // PropertyProvider
            // If the advanced setting PropertiesToIgnore or PropertyTypesToIgnore are not enough there is possibility to create your own PropertyProvider
            // As a standard there are only properties serialized which:
            // - are public
            // - are not static
            // - does not contain ExcludeFromSerializationAttribute
            // - have their set and get accessors
            // - are not indexers
            // - are not in PropertyProvider.PropertiesToIgnore
            // - are not in PropertyProvider.PropertyTypesToIgnore
            // You can replace this functionality with an inheritor class of PropertyProvider

            sharpSerializer.PropertyProvider = new MyVerySophisticatedPropertyProvider();

            // Override its methods GetAllProperties() and IgnoreProperty to customize the functionality

            return sharpSerializer;
        }


        private void serializeSizeOptimizedBinary_Click(object sender, EventArgs e)
        {
            // create fake obj
            var obj = RootContainer.CreateFakeRoot();

            // create instance of sharpSerializer
            var serializer = new SharpSerializer(true);


            // ************************************************************************************
            // For advanced serialization you create SharpSerializer with an overloaded constructor
            //
            //  SharpSerializerBinarySettings settings = createBinarySettings();
            //  serializer = new SharpSerializer(settings);
            //
            // Scroll the page to the createBinarySettings() method for more details
            // ************************************************************************************


            // set the filename
            var filename = "sharpSerializerExample.sizeOptimized";

            // serialize
            serialize(obj, serializer, filename);
        }

        private void serializeBurstBinary_Click(object sender, EventArgs e)
        {
            // create fake obj
            var obj = RootContainer.CreateFakeRoot();

            // create instance of sharpSerializer
            var settings = new SharpSerializerBinarySettings(BinarySerializationMode.Burst);
            var serializer = new SharpSerializer(settings);


            // ************************************************************************************
            // For advanced serialization you create SharpSerializer with an overloaded constructor
            //
            //  SharpSerializerBinarySettings settings = createBinarySettings();
            //  serializer = new SharpSerializer(settings);
            //
            // Scroll the page to the createBinarySettings() method for more details
            // ************************************************************************************


            // set the filename
            var filename = "sharpSerializerExample.burst";

            // serialize
            serialize(obj, serializer, filename);
        }
    }
}
```