

```
1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.IO;
5 using System.Windows.Forms;
6 using System.Xml;
7 using HelloWorldApp.BusinessObjects;
8 using Polenter.Serialization;
9
10
11 namespace HelloWorldApp
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19
20         private void serializeXmlButton_Click(object sender, EventArgs e)
21         {
22             // create fake obj
23             var obj = RootContainer.CreateFakeRoot();
24
25             // create instance of sharpSerializer
26             // with the standard constructor it serializes to xml
27             var serializer = new SharpSerializer();
28
29
30             // *****
31             // For advanced serialization you create SharpSerializer with an overloaded constructor
32             //
33             // SharpSerializerXmlSettings settings = createXmlSettings();
34             // serializer = new SharpSerializer(settings);
35             //
36             // Scroll the page to the createXmlSettings() method for more details
37             // *****
38
39
40             // *****
41             // You can alter the SharpSerializer with its settings, you can provide your custom readers
42             // and writers as well, to serialize data into Json or other formats.
43             //
44             // var serializer = createSerializerWithCustomReaderAndWriter();
45             //
46             // Scroll the page to the createSerializerWithCustomReaderAndWriter() method for more details
47             // *****
48
49
50             // set the filename
51             var filename = "sharpSerializerExample.xml";
52
53             // serialize
54             serialize(obj, serializer, filename);
55         }
56
57
58         private void serialize(object obj, SharpSerializer serializer, string shortFilename)
59         {
60             // Serializing the first object
61             var file1 = getFullFilename(shortFilename, "1");
62             serializer.Serialize(obj, file1);
63
64             // Deserializing to a second object
65             var obj2 = serializer.Deserialize(file1);
66
67             // Serializing the second object
68             var file2 = getFullFilename(shortFilename, "2");
69             serializer.Serialize(obj2, file2);
70
71             // Comparing two files
72             compareTwoFiles(file1, file2);
73
74             // Show files in explorer
75             showInExplorer(file1);
76         }
77
78         private void compareTwoFiles(string file1, string file2)
79         {
80             // comparing
81             var fileInfo1 = new FileInfo(file1);
82             var fileInfo2 = new FileInfo(file2);
83
84
85             if (fileInfo1.Length > 0 && fileInfo1.Length == fileInfo2.Length)
86             {
87                 byte[] content1 = File.ReadAllBytes(file1);
88                 byte[] content2 = File.ReadAllBytes(file2);
89
90                 for(int i = 0; i < content1.Length; i++)
91                     if (content1[i] != content2[i])
92                     {
93                         MessageBox.Show(string.Format("Files differ at offset {0}", i));
94                         return;
95                     }
96
97                 MessageBox.Show(string.Format("Both files have the same length of {0} bytes and the same content", fileInfo1.Length));
98             }
99             else
100             {
101                 MessageBox.Show(string.Format("Length of file1: {0}, Length of file2: {1}", fileInfo1.Length,
102                     fileInfo2.Length));
103             }
104         }
105
106         private void showInExplorer(string filename)
107         {
108             if (!string.IsNullOrEmpty(filename) && File.Exists(filename))
109             {
110                 string arguments = string.Format("/n, /select, \"{0}\"", filename);
111                 Process.Start("explorer", arguments);
112             }
113         }
114
115         private static string getFullFilename(string shortFilename, string nameSuffix)
116         {
117             var folder = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
118             var filenameWithoutExtension = string.Format("{0}{1}", Path.GetFileNameWithoutExtension(shortFilename),
119                 nameSuffix);
120         }
121     }
122 }
```

```

120     var filenameWithExtension = Path.ChangeExtension(filenameWithoutExtension, Path.GetExtension(shortFilename));
121     return Path.Combine(folder, filenameWithExtension);
122 }
123
124
125 private SharpSerializerXmlSettings createXmlSettings()
126 {
127     // create the settings instance
128     var settings = new SharpSerializerXmlSettings();
129
130     // Bare instance of SharpSerializerXmlSettings is enough for SharpSerializer to know,
131     // it should serialize data as xml.
132
133     // However there is more you can influence.
134
135
136     // Culture
137     // All float numbers and date/time values are serialized as text according to the Culture.
138     // The default Culture is InvariantCulture but you can override this settings with your own culture.
139     settings.Culture = System.Globalization.CultureInfo.CurrentCulture;
140
141
142     // Encoding
143     // Default Encoding is UTF8. Encoding impacts the format in which the whole Xml file is stored.
144     settings.Encoding = System.Text.Encoding.ASCII;
145
146
147     // AssemblyQualifiedName
148     // During serialization all types must be converted to strings.
149     // Since v.2.12 the type is stored as an AssemblyQualifiedName per default.
150     // You can force the SharpSerializer to shorten the type descriptions
151     // by setting the following properties to false
152     // Example of AssemblyQualifiedName:
153     // "System.String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
154     // Example of the short type name:
155     // "System.String, mscorlib"
156     settings.IncludeAssemblyVersionInTypeName = true;
157     settings.IncludeCultureInTypeName = true;
158     settings.IncludePublicKeyTokenInTypeName = true;
159
160
161
162     // ADVANCED SETTINGS
163     // Most of the classes needed to alter these settings are in the namespace Polenter.Serialization.Advanced
164
165
166     // PropertiesToIgnore
167     // Sometimes you want to ignore some properties during the serialization.
168     // If they are parts of your own business objects, you can mark these properties with ExcludeFromSerializationAttribute.
169     // However it is not possible to mark them in the built in .NET classes
170     // In such a case you add these properties to the list PropertiesToIgnore.
171     // I.e. System.Collections.Generic.List<string> has the "Capacity" property which is irrelevant for
172     // the whole Serialization and should be ignored
173     // serializer.PropertyProvider.PropertiesToIgnore.Add(typeof(List<string>), "Capacity")
174     settings.AdvancedSettings.PropertiesToIgnore.Add(typeof(List<string>), "Capacity");
175
176
177     // PropertyTypesToIgnore
178     // Sometimes you want to ignore some types during the serialization.
179     // To ignore a type add these types to the list PropertyTypesToIgnore.
180     settings.AdvancedSettings.PropertyTypesToIgnore.Add(typeof(List<string>));
181
182
183     // RootName
184     // There is always a root element during serialization. Default name of this element is "Root",
185     // but you can change it to any other text.
186     settings.AdvancedSettings.RootName = "MyFunnyClass";
187
188
189     // SimpleValueConverter
190     // During xml serialization all simple values are converted to their string representation.
191     // Float values, DateTime are default converted to format of the settings.Culture or CultureInfo.InvariantCulture
192     // if the settings.Culture is not set.
193     // If you want to store these values in your own format (Morse alphabet?) create your own converter as an instance of ISimpleValueConverter.
194     // Important! This setting overrides the settings.Culture
195     settings.AdvancedSettings.SimpleValueConverter = new MyCustomSimpleValueConverter();
196
197
198
199     // TypeNameConverter
200     // Since the v.2.12 all types are serialized as AssemblyQualifiedName.
201     // To change this you can alter the settings above (Include...) or create your own instance of ITypeNameConverter.
202     // Important! This property overrides the three properties below/above:
203     // IncludeAssemblyVersionInTypeName, IncludeCultureInTypeName, IncludePublicKeyTokenInTypeName
204     settings.AdvancedSettings.TypeNameConverter = new MyTypeNameConverterWithCompressedTypeNames();
205
206
207     return settings;
208 }
209
210 private SharpSerializerBinarySettings createBinarySettings()
211 {
212     // create the settings instance
213     var settings = new SharpSerializerBinarySettings();
214
215     // bare instance of SharpSerializerBinarySettings tells SharpSerializer to serialize data into binary format in the SizeOptimized mode
216
217     // However there is more possibility to influence the serialization
218
219
220     // Encoding
221     // Default Encoding is UTF8.
222     // Changing of Encoding has impact on format in which are all strings stored (type names, property names and string values)
223     settings.Encoding = System.Text.Encoding.ASCII;
224
225
226     // AssemblyQualifiedName
227     // During serialization all types must be converted to strings.
228     // Since v.2.12 the type is stored as an AssemblyQualifiedName per default.
229     // You can force the SharpSerializer to shorten the type descriptions
230     // by setting the following properties to false
231     // Example of AssemblyQualifiedName:
232     // "System.String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
233     // Example of the short type name:
234     // "System.String, mscorlib"
235     settings.IncludeAssemblyVersionInTypeName = true;
236     settings.IncludeCultureInTypeName = true;
237     settings.IncludePublicKeyTokenInTypeName = true;
238

```

```

239
240 // Mode
241 // The default mode, without altering the settings, is BinarySerializationMode.SizeOptimized
242 // Actually you can choose another mode - BinarySerializationMode.Burst
243 //
244 // What is the difference?
245 // To successfully restore the object tree from the serialized stream, all objects have to be serialized including their type information.
246 // Both modes differ in the art the type information is stored.
247 //
248 // BinarySerializationMode.Burst
249 // In the burst mode, type of every object is serialized as a string as part of this object.
250 // 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.
251 // It increases the file size especially when serializing collections. Type information is duplicated.
252 // It's ok for single, simple objects, as it has small overhead. BurstBinaryWriter supports this mode.
253 //
254 // BinarySerializationMode.SizeOptimized
255 // In the SizeOptimized mode all types are grouped into a list. All type duplicates are removed.
256 // Serialized objects refer only to this list using index of their type. It's recommended approach for serializing of complex
257 // objects with many properties, or many items of the same type (collections). The drawback is - all objects are cached,
258 // then their types are analysed, type list is created, objects are injected with indexes and finally the data is written.
259 // Apart from types, all property names are handled the same way in the SizeOptimized mode.
260 // SizeOptimizedBinaryWriter supports this mode.
261 settings.Mode = BinarySerializationMode.SizeOptimized;
262
263
264
265 // ADVANCED SETTINGS
266 // Most of the classes needed to alter these settings are in the namespace Polenter.Serialization.Advanced
267
268
269 // PropertiesToIgnore
270 // Sometimes you want to ignore some properties during the serialization.
271 // If they are parts of your own business objects, you can mark these properties with ExcludeFromSerializationAttribute.
272 // However it is not possible to mark them in the built in .NET classes
273 // In such a case you add these properties to the list PropertiesToIgnore.
274 // I.e. System.Collections.Generic.List<string> has the "Capacity" property which is irrelevant for
275 // the whole Serialization and should be ignored
276 // serializer.PropertyProvider.PropertiesToIgnore.Add(typeof(List<string>), "Capacity")
277 settings.AdvancedSettings.PropertiesToIgnore.Add(typeof(List<string>), "Capacity");
278
279
280 // PropertyTypesToIgnore
281 // Sometimes you want to ignore some types during the serialization.
282 // To ignore a type add these types to the list PropertyTypesToIgnore.
283 settings.AdvancedSettings.PropertyTypesToIgnore.Add(typeof(List<string>));
284
285
286 // RootName
287 // There is always a root element during the serialization. Default name of this element is "Root",
288 // but you can change it to any other text.
289 settings.AdvancedSettings.RootName = "MyFunnyClass";
290
291
292 // TypeNameConverter
293 // Since the v.2.12 all types are serialized as AssemblyQualifiedName.
294 // To change this you can alter the settings above (Include...) or create your own instance of ITypeNameConverter.
295 // Important! This property overrides the three properties below/above:
296 // IncludeAssemblyVersionInTypeName, IncludeCultureInTypeName, IncludePublicKeyTokenInTypeName
297 settings.AdvancedSettings.TypeNameConverter = new MyTypeNameConverterWithCompressedTypeNames();
298
299
300 return settings;
301 }
302
303 private SharpSerializer createSerializerWithCustomReaderAndWriter()
304 {
305 // *****
306 // SERIALIZATION
307 // The namespace Polenter.Serialization.Advanced contains some classes which are indispensable during the serialization.
308
309
310 // *****
311 // XmlPropertySerializer
312 // serializes objects into elements and their attributes.
313 // Each element has its begin and end tag.
314 // XmlPropertySerializer self is not responsible for the serializing to xml, it doesn't reference the built in .NET XmlWriter.
315 // Instead it uses an instance of IXmlWriter to control the element writing.
316 // DefaultXmlWriter implements IXmlWriter and contains the built in .NET XmlWriter which is responsible for writing to the stream.
317
318 // To make your own node oriented writer, you need to make a class which implements IXmlWriter
319 Polenter.Serialization.Advanced.Xml.IXmlWriter jsonWriter = new MyJsonWriter();
320
321 // this writer is passed to the constructor of the XmlPropertySerializer
322 Polenter.Serialization.Advanced.Serializing.IPropertySerializer serializer =
323     new Polenter.Serialization.Advanced.XmlPropertySerializer(jsonWriter);
324
325 // in such a was, the default XmlPropertySerializer can store data in any format which is node oriented (contains begin/end tags)
326
327
328 // *****
329 // BinaryPropertySerializer
330 // serializes objects into elements which have known length and fixed position in the stream.
331 // It doesn't write directly to the stream. Instead, it uses an instance of IBinaryWriter.
332 // Actually there are two writers used by the SharpSerializer: BurstBinaryWriter and SizeOptimizedBinaryWriter
333
334 // To make your own binary writer you make a class which implements the IBinaryWriter.
335 Polenter.Serialization.Advanced.Binary.IBinaryWriter compressedWriter = new MyVeryStrongCompressedAndEncryptedBinaryWriter();
336
337 // this writer is passed to the constructor of the BinaryPropertySerializer
338 serializer = new Polenter.Serialization.Advanced.BinaryPropertySerializer(compressedWriter);
339
340 // Changing only the writer and not the whole serializer allows an easy serialization of data to any binary format
341
342
343
344 // *****
345 // DESERIALIZATION
346 // The namespace Polenter.Serialization.Advanced contains classes which are counterparts of the above serializers/writers
347 // XmlPropertySerializer -> XmlPropertyDeserializer
348 // DefaultXmlWriter -> DefaultXmlReader
349 // BurstBinaryWriter -> BurstBinaryReader
350 // SizeOptimizedBinaryWriter -> SizeOptimizedBinaryReader
351
352 // Deserializers are constructed analog or better say - symmetric to the Serializers/Writers, i.e.
353
354 Polenter.Serialization.Advanced.Binary.IBinaryReader compressedReader =
355     new MyVeryStrongCompressedAndEncryptedBinaryReader();
356
357 Polenter.Serialization.Advanced.Deserializing.IPropertyDeserializer deserializer =

```



```

358         new Polenter.Serialization.Advanced.BinaryPropertyDeserializer(compressedReader);
359
360         // If you have created serializer and deserializer, the next step is to create SharpSerializer.
361
362         // *****
363         // Creating SharpSerializer
364         // Both classes - serializer and deserializer are passed to the overloaded constructor
365
366         var sharpSerializer = new SharpSerializer(serializer, deserializer);
367
368
369         // there is one more option you can alter directly on your instance of SharpSerializer
370
371         // *****
372         // PropertyProvider
373         // If the advanced setting PropertiesToIgnore or PropertyTypesToIgnore are not enough there is possibility to create your own PropertyProvider
374         // As a standard there are only properties serialized which:
375         // - are public
376         // - are not static
377         // - does not contain ExcludeFromSerializationAttribute
378         // - have their set and get accessors
379         // - are not indexers
380         // - are not in PropertyProvider.PropertiesToIgnore
381         // - are not in PropertyProvider.PropertyTypesToIgnore
382         // You can replace this functionality with an inheritor class of PropertyProvider
383
384         sharpSerializer.PropertyProvider = new MyVerySophisticatedPropertyProvider();
385
386         // Override its methods GetAllProperties() and IgnoreProperty to customize the functionality
387
388         return sharpSerializer;
389     }
390
391
392
393     private void serializeSizeOptimizedBinary_Click(object sender, EventArgs e)
394     {
395         // create fake obj
396         var obj = RootContainer.CreateFakeRoot();
397
398         // create instance of sharpSerializer
399         var serializer = new SharpSerializer(true);
400
401         // *****
402         // For advanced serialization you create SharpSerializer with an overloaded constructor
403         //
404         // SharpSerializerBinarySettings settings = createBinarySettings();
405         // serializer = new SharpSerializer(settings);
406         //
407         // Scroll the page to the createBinarySettings() method for more details
408         // *****
409
410
411         // set the filename
412         var filename = "SharpSerializerExample.sizeOptimized";
413
414         // serialize
415         serialize(obj, serializer, filename);
416     }
417
418
419     private void serializeBurstBinary_Click(object sender, EventArgs e)
420     {
421         // create fake obj
422         var obj = RootContainer.CreateFakeRoot();
423
424         // create instance of sharpSerializer
425         var settings = new SharpSerializerBinarySettings(BinarySerializationMode.Burst);
426         var serializer = new SharpSerializer(settings);
427
428         // *****
429         // For advanced serialization you create SharpSerializer with an overloaded constructor
430         //
431         // SharpSerializerBinarySettings settings = createBinarySettings();
432         // serializer = new SharpSerializer(settings);
433         //
434         // Scroll the page to the createBinarySettings() method for more details
435         // *****
436
437
438         // set the filename
439         var filename = "SharpSerializerExample.burst";
440
441         // serialize
442         serialize(obj, serializer, filename);
443     }
444 }
445 }
446 }

```