

---

# **ExtendedXmlSerializer Documentation**

***Release 4.0***

**ExtendedXmlSerializer Contributors**

**Nov 14, 2017**



---

## Contents

---

<b>1</b>	<b>Information</b>	<b>3</b>
<b>2</b>	<b>The Basics</b>	<b>5</b>
<b>3</b>	<b>Xml Settings</b>	<b>7</b>
<b>4</b>	<b>Serialization</b>	<b>9</b>
<b>5</b>	<b>Deserialization</b>	<b>11</b>
<b>6</b>	<b>Fluent API</b>	<b>13</b>
<b>7</b>	<b>Serialization of dictionary</b>	<b>15</b>
<b>8</b>	<b>Custom serialization</b>	<b>17</b>
<b>9</b>	<b>Deserialize old version of xml</b>	<b>19</b>
<b>10</b>	<b>Extensibility</b>	<b>21</b>
<b>11</b>	<b>Object reference and circular reference</b>	<b>23</b>
<b>12</b>	<b>Property Encryption</b>	<b>25</b>
<b>13</b>	<b>Custom Conversion</b>	<b>27</b>
<b>14</b>	<b>Optimized Namespaces</b>	<b>29</b>
<b>15</b>	<b>Implicit Namespaces/Typing</b>	<b>31</b>
<b>16</b>	<b>Auto-Formatting (Attributes)</b>	<b>33</b>
<b>17</b>	<b>Private Constructors</b>	<b>35</b>
<b>18</b>	<b>Parameterized Members and Content</b>	<b>37</b>
<b>19</b>	<b>Tuples</b>	<b>39</b>
<b>20</b>	<b>Experimental Xaml-ness: Attached Properties</b>	<b>41</b>

<b>21</b>	<b>Experimental Xaml-ness: Markup Extensions</b>	<b>43</b>
<b>22</b>	<b>How to Upgrade from v1.x to v2</b>	<b>45</b>
<b>23</b>	<b>History</b>	<b>47</b>
<b>24</b>	<b>Authors</b>	<b>49</b>
<b>25</b>	<b>Indices and tables</b>	<b>51</b>

ExtendedXmlSerializer for .NET



Support platforms:

- .NET 4.5
- .NET Platform Standard 1.6

Support features:

- Deserialization xml from standard XMLSerializer
- Serialization class, struct, generic class, primitive type, generic list and dictionary, array, enum
- Serialization class with property interface
- Serialization circular reference and reference Id
- Deserialization of old version of xml
- Property encryption
- Custom serializer
- Support XmlElementAttribute and XmlRootAttribute
- POJO - all configurations (migrations, custom serializer...) are outside the class

Standard XML Serializer in .NET is very limited:

- Does not support serialization of class with circular reference or class with interface property.
- There is no mechanism for reading the old version of XML.
- Does not support properties that are defined with interface types.
- Does not support read-only collection properties (like Xaml does).
- Does not support parameterized constructors.
- Does not support private constructors.
- If you want create custom serializer, your class must inherit from IXmlSerializable. This means that your class will not be a POJO class.

- Does not support IoC



## CHAPTER 2

---

### The Basics

---

Everything in `ExtendedXmlSerializer` begins with a configuration container, from which you can use to configure the serializer and ultimately create it:

```
var serializer = new ConfigurationContainer()
                    // Configure...
                    .Create();
```

Using this simple subject class:

```
public sealed class Subject
{
    public string Message { get; set; }

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

The results of the default serializer will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Introduction;
↪assembly=ExtendedXmlSerializer.Samples">
  <Message>Hello World!</Message>
  <Count>6776</Count>
</Subject>
```

We can take this a step further by configuring the *Subject*'s `Type` and `Member` properties, which will effect how its Xml is emitted. Here is an example of configuring the *Subject*'s name to emit as *ModifiedSubject*:

```
var serializer = new ConfigurationContainer().ConfigureType<Subject>()
                    .Name("ModifiedSubject")
                    .Create();
```

```
<?xml version="1.0" encoding="utf-8"?>
<ModifiedSubject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Introduction;
↪assembly=ExtendedXmlSerializer.Samples">
```

```
<Message>Hello World!</Message>
<Count>6776</Count>
</ModifiedSubject>
```

Diving a bit further, we can also configure the type's member information. For example, configuring *Subject.Message* to emit as *Text* instead:

```
var serializer = new ConfigurationContainer().ConfigureType<Subject>()
    .Member(x => x.Message)
    .Name("Text")
    .Create();
```

```
<?xml version="1.0" encoding="utf-8"?>
<Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Introduction;
assembly=ExtendedXmlSerializer.Samples">
  <Text>Hello World!</Text>
  <Count>6776</Count>
</Subject>
```

## CHAPTER 3

---

### Xml Settings

---

In case you want to configure the XML write and read settings via *XmlWriterSettings* and *XmlReaderSettings* respectively, you can do that via extension methods created for you to do so:

```
var subject = new Subject{ Count = 6776, Message = "Hello World!" };
var serializer = new ConfigurationContainer().Create();
var contents = serializer.Serialize(new XmlWriterSettings {Indent = true}, subject);
// ...
```

And for reading:

```
var instance = serializer.Deserialize<Subject>(new XmlReaderSettings{IgnoreWhitespace =
↳ false}, contents);
// ...
```



## CHAPTER 4

---

### Serialization

---

Now that your configuration container has been configured and your serializer has been created, it's time to get to the serialization.

```
var serializer = new ConfigurationContainer().Create();  
var obj = new TestClass();  
var xml = serializer.Serialize(obj);
```



## CHAPTER 5

---

### Deserialization

---

```
var obj2 = serializer.Deserialize<TestClass>(xml);
```





ExtendedXmlSerializer use fluent API to configuration. Example

```
var serializer = new ConfigurationContainer()
    .UseEncryptionAlgorithm(new CustomEncryption())
    .Type<Person>() // Configuration of Person class
        .Member(p => p.Password) // First member
            .Name("P")
            .Encrypt()
        .Member(p => p.Name) // Second member
            .Name("T")
    .Type<TestClass>() // Configuration of another class
        .CustomSerializer(new TestClassSerializer())
    .Create();
```



---

Serialization of dictionary

---

You can serialize generic dictionary, that can store any type.

```
public class TestClass
{
    public Dictionary<int, string> Dictionary { get; set; }
}
```

```
var obj = new TestClass
{
    Dictionary = new Dictionary<int, string>
    {
        {1, "First"},
        {2, "Second"},
        {3, "Other"},
    }
};
```

Output XML will look like:

```
<?xml version="1.0" encoding="utf-8"?>
<TestClass xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Dictionary;
↪assembly=ExtendedXmlSerializer.Samples">
  <Dictionary>
    <Item xmlns="https://extendedxmlserializer.github.io/system">
      <Key>1</Key>
      <Value>First</Value>
    </Item>
    <Item xmlns="https://extendedxmlserializer.github.io/system">
      <Key>2</Key>
      <Value>Second</Value>
    </Item>
    <Item xmlns="https://extendedxmlserializer.github.io/system">
      <Key>3</Key>
      <Value>Other</Value>
    </Item>
  </Dictionary>
</TestClass>
```

```
</Item>
</Dictionary>
</TestClass>
```

If you use `UseOptimizedNamespaces` function xml will look like:

```
<?xml version="1.0" encoding="utf-8"?>
<TestClass xmlns:sys="https://extendedxmlserializer.github.io/system" xmlns:exs=
↪ "https://extendedxmlserializer.github.io/v2" xmlns="clr-
↪ namespace:ExtendedXmlSerializer.Samples.Dictionary;assembly=ExtendedXmlSerializer.
↪ Samples">
  <Dictionary>
    <sys:Item>
      <Key>1</Key>
      <Value>First</Value>
    </sys:Item>
    <sys:Item>
      <Key>2</Key>
      <Value>Second</Value>
    </sys:Item>
    <sys:Item>
      <Key>3</Key>
      <Value>Other</Value>
    </sys:Item>
  </Dictionary>
</TestClass>
```

---

## Custom serialization

---

If your class has to be serialized in a non-standard way:

```
public class TestClass
{
    public TestClass(string paramStr, int paramInt)
    {
        PropStr = paramStr;
        PropInt = paramInt;
    }

    public string PropStr { get; private set; }
    public int PropInt { get; private set; }
}
```

You must create custom serializer:

```
public class TestClassSerializer : IExtendedXmlCustomSerializer<TestClass>
{
    public TestClass Deserialize(XElement element)
    {
        var xElement = element.Member("String");
        var xElement1 = element.Member("Int");
        if (xElement != null && xElement1 != null)
        {
            var strValue = xElement.Value;

            var intValue = Convert.ToInt32(xElement1.Value);
            return new TestClass(strValue, intValue);
        }
        throw new InvalidOperationException("Invalid xml for class_
↪TestClassWithSerializer");
    }

    public void Serializer(XmlWriter writer, TestClass obj)
    {
    }
```

```
        writer.WriteElementString("String", obj.PropStr);
        writer.WriteElementString("Int", obj.PropInt.ToString(CultureInfo.
↪ InvariantCulture));
    }
}
```

Then, you have to add custom serializer to configuration of TestClass:

```
var serializer = new ConfigurationContainer().Type<TestClass>()
                                                    .CustomSerializer(new ↪
↪ TestClassSerializer())
                                                    .Create();
```

## CHAPTER 9

---

### Deserialize old version of xml

---

In standard XMLSerializer you can't deserialize XML in case you change model. In ExtendedXMLSerializer you can create migrator for each class separately. E.g.: If you have big class, that uses small class and this small class will be changed you can create migrator only for this small class. You don't have to modify whole big XML. Now I will show you a simple example: If you had a class:

```
public class TestClass
{
    public int Id { get; set; }
    public string Type { get; set; }
}
```

and generated XML look like:

```
<? xml version="1.0" encoding="utf-8"?>
<TestClass xmlns="clr-namespace:ExtendedXmlSerialization.Samples.MigrationMap;
↪assembly=ExtendedXmlSerializer.Samples">
    <Id>1</Id>
    <Type>Type</Type>
</TestClass>
```

Then you renamed property:

```
public class TestClass
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

and generated XML look like:

```
<? xml version="1.0" encoding="utf-8"?>
<TestClass xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:version="1"
↪xmlns="clr-namespace:ExtendedXmlSerialization.Samples.MigrationMap;
↪assembly=ExtendedXmlSerializer.Samples">
    <Id>1</Id>
```

```
<Name>Type</Name>
</TestClass>
```

Then, you added new property and you wanted to calculate a new value during deserialization.

```
public class TestClass
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Value { get; set; }
}
```

and new XML should look like:

```
<?xml version="1.0" encoding="utf-8"?>
<TestClass xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:version="2"
↪xmlns="clr-namespace:ExtendedXmlSerializer.Samples.MigrationMap;
↪assembly=ExtendedXmlSerializer.Samples">
    <Id>1</Id>
    <Name>Type</Name>
    <Value>Calculated</Value>
</TestClass>
```

You can migrate (read) old version of XML using migrations:

```
public class TestClassMigrations : IEnumerable<Action<XElement>>
{
    public static void MigrationV0(XElement node)
    {
        var typeElement = node.Member("Type");
        // Add new node
        node.Add(new XElement("Name", typeElement.Value));
        // Remove old node
        typeElement.Remove();
    }

    public static void MigrationV1(XElement node)
    {
        // Add new node
        node.Add(new XElement("Value", "Calculated"));
    }

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

    public IEnumerator<Action<XElement>> GetEnumerator()
    {
        yield return MigrationV0;
        yield return MigrationV1;
    }
}
```

Then, you must register your TestClassMigrations class in configuration

```
var serializer = new ConfigurationContainer().ConfigureType<TestClass>()
    .AddMigration(new TestClassMigrations())
    .Create();
```



## CHAPTER 10

---

### Extensibility

---

With type and member configuration out of the way, we can turn our attention to what really makes `ExtendedXmlSerializer` tick: extensibility. As its name suggests, `ExtendedXmlSerializer` offers a very flexible (but albeit new) extension model from which you can build your own extensions. Pretty much all if not all features you encounter with `ExtendedXmlSerializer` are through extensions. There are quite a few in our latest version here that showcase this extensibility. The remainder of this document will showcase the top features of `ExtendedXmlSerializer` that are accomplished through its extension system.



---

## Object reference and circular reference

---

If you have a class:

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }

    public Person Boss { get; set; }
}

public class Company
{
    public List<Person> Employees { get; set; }
}
```

then you create object with circular reference, like this:

```
var boss = new Person {Id = 1, Name = "John"};
boss.Boss = boss; //himself boss
var worker = new Person {Id = 2, Name = "Oliver"};
worker.Boss = boss;
var obj = new Company
{
    Employees = new List<Person>
    {
        worker,
        boss
    }
};
```

You must configure Person class as reference object:

```
var serializer = new ConfigurationContainer().ConfigureType<Person>()
                                                .EnableReferences(p => p.Id)
                                                .Create();
```

Output XML will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Company xmlns="clr-namespace:ExtendedXmlSerializer.Samples.ObjectReference;
↪assembly=ExtendedXmlSerializer.Samples">
  <Employees>
    <Capacity>4</Capacity>
    <Person Id="2">
      <Name>Oliver</Name>
      <Boss Id="1">
        <Name>John</Name>
        <Boss xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:entity="1" />
      </Boss>
    </Person>
    <Person xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:entity="1" />
  </Employees>
</Company>
```

## CHAPTER 12

---

### Property Encryption

---

If you have a class with a property that needs to be encrypted:

```
public class Person
{
    public string Name { get; set; }
    public string Password { get; set; }
}
```

You must implement interface `IEncryption`. For example, it will show the Base64 encoding, but in the real world better to use something safer, eg. RSA.:

```
public class CustomEncryption : IEncryption
{
    public string Parse(string data)
        => Encoding.UTF8.GetString(Convert.FromBase64String(data));

    public string Format(string instance)
        => Convert.ToBase64String(Encoding.UTF8.GetBytes(instance));
}
```

Then, you have to specify which properties are to be encrypted and register your `IEncryption` implementation.

```
var serializer = new ConfigurationContainer().UseEncryptionAlgorithm(new CustomEncryption())
    .ConfigureType<Person>()
    .Member(p => p.Password)
    .Encrypt()
    .Create();
```



# CHAPTER 13

---

## Custom Conversion

---

ExtendedXmlSerializer does a pretty decent job (if we do say so ourselves) of composing and decomposing objects, but if you happen to have a type that you want serialized in a certain way, and this type can be deconstructed into a *string*, then you can register a custom converter for it.

Using the following:

```
public sealed class CustomStructConverter : IConverter<CustomStruct>
{
    public static CustomStructConverter Default { get; } = new
    ↪ CustomStructConverter();
    CustomStructConverter() {}

    public bool IsSatisfiedBy(TypeInfo parameter) => typeof(CustomStruct).
    ↪ GetTypeInfo()

    ↪ IsAssignableFrom(parameter);

    public CustomStruct Parse(string data) =>
        int.TryParse(data, out var number) ? new CustomStruct(number) : CustomStruct.
    ↪ Default;

    public string Format(CustomStruct instance) => instance.Number.ToString();
}

public struct CustomStruct
{
    public static CustomStruct Default { get; } = new CustomStruct(6776);

    public CustomStruct(int number)
    {
        Number = number;
    }

    public int Number { get; }
}
```

Register the converter:

```
var serializer = new ConfigurationContainer().Register(CustomStructConverter.Default).
    ↪ Create();
var subject = new CustomStruct(123);
var contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<CustomStruct xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
    ↪ assembly=ExtendedXmlSerializer.Samples">123</CustomStruct>
```



---

Optimized Namespaces

---

By default Xml namespaces are emitted on an “as needed” basis:

```
<?xml version="1.0" encoding="utf-8"?>
<List xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:arguments="Object"
↪ xmlns="https://extendedxmlserializer.github.io/system">
  <Capacity>4</Capacity>
  <Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪ assembly=ExtendedXmlSerializer.Samples">
    <Message>First</Message>
  </Subject>
  <Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪ assembly=ExtendedXmlSerializer.Samples">
    <Message>Second</Message>
  </Subject>
  <Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪ assembly=ExtendedXmlSerializer.Samples">
    <Message>Third</Message>
  </Subject>
</List>
```

But with one call to the *UseOptimizedNamespaces* call, namespaces get placed at the root of the document, thereby reducing document footprint:

```
var serializer = new ConfigurationContainer().UseOptimizedNamespaces()
                                                    .Create();
var subject = new List<object>
{
    new Subject {Message = "First"},
    new Subject {Message = "Second"},
    new Subject {Message = "Third"}
};
var contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<List xmlns:ns1="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪assembly=ExtendedXmlSerializer.Samples" xmlns:exs="https://extendedxmlserializer.
↪github.io/v2" exs:arguments="Object" xmlns="https://extendedxmlserializer.github.io/
↪system">
  <Capacity>4</Capacity>
  <ns1:Subject>
    <Message>First</Message>
  </ns1:Subject>
  <ns1:Subject>
    <Message>Second</Message>
  </ns1:Subject>
  <ns1:Subject>
    <Message>Third</Message>
  </ns1:Subject>
</List>
```

## CHAPTER 15

---

### Implicit Namespaces/Typing

---

If you don't like namespaces at all, you can register types so that they do not emit namespaces when they are rendered into a document:

```
var serializer = new ConfigurationContainer().EnableImplicitTyping(typeof(Subject))
    .Create();
var subject = new Subject{ Message = "Hello World! No namespaces, yay!" };
var contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<Subject>
  <Message>Hello World! No namespaces, yay!</Message>
</Subject>
```



## CHAPTER 16

---

### Auto-Formatting (Attributes)

---

The default behavior for emitting data in an Xml document is to use elements, which can be a little chatty and verbose:

```
var serializer = new ConfigurationContainer().UseOptimizedNamespaces()
                                                .Create();
var subject = new List<object>
{
    new Subject {Message = "First"},
    new Subject {Message = "Second"},
    new Subject {Message = "Third"}
};
var contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<SubjectWithThreeProperties xmlns="clr-namespace:ExtendedXmlSerializer.Samples.
↪Extensibility;assembly=ExtendedXmlSerializer.Samples">
  <Number>123</Number>
  <Message>Hello World!</Message>
  <Time>2017-10-25T08:38:36.9033659+02:00</Time>
</SubjectWithThreeProperties>
```

Making use of the *UseAutoFormatting* call will enable all types that have a registered *IConverter* (convert to string and back) to emit as attributes:

```
<?xml version="1.0" encoding="utf-8"?>
<SubjectWithThreeProperties Number="123" Message="Hello World!" Time="2017-10-
↪25T08:38:36.9033659+02:00" xmlns="clr-namespace:ExtendedXmlSerializer.Samples.
↪Extensibility;assembly=ExtendedXmlSerializer.Samples" />
```



# CHAPTER 17

---

## Private Constructors

---

One of the limitations of the classic `XmlSerializer` is that it does not support private constructors, but `ExtendedXmlSerializer` does via its *EnableAllConstructors* call:

```
public sealed class SubjectByFactory
{
    public static SubjectByFactory Create(string message) => new
↳ SubjectByFactory(message);

    SubjectByFactory() : this(null) {} // Used by serializer.

    SubjectByFactory(string message) => Message = message;

    public string Message { get; set; }
}
```

```
var serializer = new ConfigurationContainer().EnableAllConstructors()
                                                .Create();
var subject = SubjectByFactory.Create("Hello World!");
var contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<SubjectByFactory xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↳ assembly=ExtendedXmlSerializer.Samples">
  <Message>Hello World!</Message>
</SubjectByFactory>
```





---

## Parameterized Members and Content

---

Taking this concept bit further leads to a favorite feature of ours in `ExtendedXmlSerializer`. The classic serializer only supports parameterless public constructors. With `ExtendedXmlSerializer`, you can use the `EnableParameterizedContent` call to enable parameterized parameters in the constructor that by convention have the same name as the property for which they are meant to assign:

```
public sealed class ParameterizedSubject
{
    public ParameterizedSubject(string message, int number, DateTime time)
    {
        Message = message;
        Number = number;
        Time = time;
    }

    public string Message { get; }
    public int Number { get; }
    public DateTime Time { get; }
}
```

```
var serializer = new ConfigurationContainer().EnableParameterizedContent()
    .Create();
var subject = new ParameterizedSubject("Hello World!", 123, DateTime.Now);
var contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<ParameterizedSubject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.
↪Extensibility;assembly=ExtendedXmlSerializer.Samples">
    <Message>Hello World!</Message>
    <Number>123</Number>
    <Time>2017-10-25T08:38:37.1053861+02:00</Time>
</ParameterizedSubject>
```



## CHAPTER 19

---

### Tuples

---

By enabling parameterized content, it opens up a lot of possibilities, like being able to serialize Tuples. Of course, serializable Tuples were introduced recently with the latest version of C#. Here, however, you can couple this with our member-naming functionality and provide better naming for your tuple properties:

```
var serializer = new ConfigurationContainer().EnableParameterizedContent()
    .Type<Tuple<string>>()
    .Member(x => x.Item1)
    .Name("Message")
    .Create();

var subject = Tuple.Create("Hello World!");
var contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<Tuple xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:arguments="string"
  xmlns="https://extendedxmlserializer.github.io/system">
  <Message>Hello World!</Message>
</Tuple>
```



---

## Experimental Xaml-ness: Attached Properties

---

We went ahead and got a little cute with v2 of `ExtendedXmlSerializer`, adding support for Attached Properties on objects in your serialized object graph. But instead of constraining it to objects that inherit from *DependencyObject*, every object can benefit from it. Check it out:

```
sealed class NameProperty : ReferenceProperty<Subject, string>
{
    public const string DefaultMessage = "The Name Has Not Been Set";

    public static NameProperty Default { get; } = new NameProperty();
    NameProperty() : base(() => Default, x => DefaultMessage) {}
}

sealed class NumberProperty : StructureProperty<Subject, int>
{
    public const int DefaultValue = 123;

    public static NumberProperty Default { get; } = new NumberProperty();
    NumberProperty() : base(() => Default, x => DefaultValue) {}
}
```

```
var serializer = new ConfigurationContainer().EnableAttachedProperties(NameProperty.
    ↳Default,
                                                                    NumberProperty.
    ↳Default)
    .Create();

var subject = new Subject {Message = "Hello World!"};
subject.Set(NameProperty.Default, "Hello World from Attached Properties!");
subject.Set(NumberProperty.Default, 123);

var contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
    ↳assembly=ExtendedXmlSerializer.Samples">
```

```
<Message>Hello World!</Message>  
<NameProperty.Default>Hello World from Attached Properties!</NameProperty.Default>  
<NumberProperty.Default>123</NumberProperty.Default>  
</Subject>
```

(Please note that this feature is experimental, but please try it out and let us know what you think!)

---

## Experimental Xaml-ness: Markup Extensions

---

Saving the best feaure for last, we have experimental support for one of Xaml's greatest features, Markup Extensions:

```
sealed class Extension : IMarkupExtension
{
    const string Message = "Hello World from Markup Extension! Your message is: ",
    ↪None = "N/A";

    readonly string _message;

    public Extension() : this(None) {}

    public Extension(string message)
    {
        _message = message;
    }

    public object ProvideValue(IServiceProvider serviceProvider) => string.
    ↪Concat(Message, _message);
}
```

```
var contents =
    @"<?xml version=""1.0"" encoding=""utf-8""?>
      <Subject xmlns=""clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
    ↪assembly=ExtendedXmlSerializer.Samples""
        Message=""{Extension 'PRETTY COOL HUH!!!}'"" />;
var serializer = new ConfigurationContainer().EnableMarkupExtensions()
    .Create();
var subject = serializer.Deserialize<Subject>(contents);
Console.WriteLine(subject.Message); // "Hello World from Markup Extension! Your
    ↪message is: PRETTY COOL HUH!!!"
```

(Please note that this feature is experimental, but please try it out and let us know what you think!)





## CHAPTER 22

---

### How to Upgrade from v1.x to v2

---

Finally, if you have documents from v1, you will need to upgrade them to v2 to work. This involves reading the document in an instance of v1 serializer, and then writing it in an instance of v2 serializer. We have provided the *ExtendedXmlSerializer.Legacy* nuget package to assist in this goal.

```
<?xml version="1.0" encoding="utf-8"?><ArrayOfSubject><Subject type=
↳ "ExtendedXmlSerializer.Samples.Introduction.Subject"><Message>First</Message><Count>
↳ 0</Count></Subject><Subject type="ExtendedXmlSerializer.Samples.Introduction.Subject
↳ "><Message>Second</Message><Count>0</Count></Subject><Subject type=
↳ "ExtendedXmlSerializer.Samples.Introduction.Subject"><Message>Third</Message><Count>
↳ 0</Count></Subject></ArrayOfSubject>
```

```
var legacySerializer = new ExtendedXmlSerialization.ExtendedXmlSerializer();
var content = File.ReadAllText(@"bin\Upgrade.Example.v1.xml"); // Path to your legacy_
↳ xml file.
var subject = legacySerializer.Deserialize<List<Subject>>(content);

// Upgrade:
var serializer = new ConfigurationContainer().Create();
var contents = serializer.Serialize(new XmlWriterSettings {Indent = true}, subject);
File.WriteAllText(@"bin\Upgrade.Example.v2.xml", contents);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<List xmlns:ns1="clr-namespace:ExtendedXmlSerializer.Samples.Introduction;
↳ assembly=ExtendedXmlSerializer.Samples" xmlns:exs="https://extendedxmlserializer.
↳ github.io/v2" exs:arguments="ns1:Subject" xmlns="https://extendedxmlserializer.
↳ github.io/system">
  <Capacity>4</Capacity>
  <ns1:Subject>
    <Message>First</Message>
    <Count>0</Count>
  </ns1:Subject>
  <ns1:Subject>
    <Message>Second</Message>
```

```
<Count>0</Count>
</ns1:Subject>
<ns1:Subject>
  <Message>Third</Message>
  <Count>0</Count>
</ns1:Subject>
</List>
```

## CHAPTER 23

---

### History

---

- 2017-?-?- v2.0.0 - Rewritten version



## CHAPTER 24

---

### Authors

---

- Wojciech Nagórski
- Mike-EEE



## CHAPTER 25

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`