Saving And Loading A C# Object’s Data To An Xml, Json, Or Binary File

9 minute read March 14, 2014

I love creating tools, particularly ones for myself and other developers to use. A common situation that I run into is needing to save the user’s settings to a file so that I can load them up the next time the tool is ran. I find that the easiest way to accomplish this is to create a Settings class to hold all of the user’s settings, and then use serialization to save and load the class instance to/from a file. I mention a Settings class here, but you can use this technique to save any object (or list of objects) to a file.

There are tons of different formats that you may want to save your object instances as, but the big three are Binary, XML, and Json. Each of these formats have their pros and cons, which I won’t go into. Below I present functions that can be used to save and load any object instance to / from a file, as well as the different aspects to be aware of when using each method.

The follow code (without examples of how to use it) is also available here, and can be used directly from my NuGet Package.

Writing and Reading an object to / from a Binary file

- Writes and reads ALL object properties and variables to / from the file (i.e. public, protected, internal, and private).
- The data saved to the file is not human readable, and thus cannot be edited outside of your application.
- Have to decorate class (and all classes that it contains) with a **[Serializable]** attribute.
- Use the **[NonSerialized]** attribute to exclude a variable from being written to the file; there is no way to prevent an auto-property from being serialized besides making it use a backing variable and putting the [NonSerialized] attribute on that.

```
/// <summary>
/// Functions for performing common binary Serialization operations.
/// <para>All properties and variables will be serialized.</para>
/// <para>Object type (and all child types) must be decorated with the [Serializable] attribute.</para>
/// <para>To prevent a variable from being serialized, decorate it with the [NonSerialized] attribute; cannot be applied to
properties.</para>
/// </summary>
public static class BinarySerialization
{
    /// <summary>
    /// Writes the given object instance to a binary file.
    /// <para>Object type (and all child types) must be decorated with the [Serializable] attribute.</para>
    /// <para>To prevent a variable from being serialized, decorate it with the [NonSerialized] attribute; cannot be applied to
properties.</para>
    /// </summary>
    /// <typeparam name="T">The type of object being written to the XML file.</typeparam>
    /// <param name="filePath">The file path to write the object instance to.</param>
    /// <param name="objectToWrite">The object instance to write to the XML file.</param>
    /// <param name="append">If false the file will be overwritten if it already exists. If true the contents will be appended to
the file.</param>
    public static void WriteToBinaryFile<T>(string filePath, T objectToWrite, bool append = false)
    {
        using (Stream stream = File.Open(filePath, append ? FileMode.Append : FileMode.Create))
        {
            var binaryFormatter = new System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();
            binaryFormatter.Serialize(stream, objectToWrite);
        }
    }

    /// <summary>
    /// Reads an object instance from a binary file.
    /// </summary>
    /// <typeparam name="T">The type of object to read from the XML.</typeparam>
    /// <param name="filePath">The file path to read the object instance from.</param>
    /// <returns>Returns a new instance of the object read from the binary file.</returns>
    public static T ReadFromBinaryFile<T>(string filePath)
    {
        using (Stream stream = File.Open(filePath, FileMode.Open))
        {
            var binaryFormatter = new System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();
            return (T)binaryFormatter.Deserialize(stream);
        }
    }
}
```

And here is an example of how to use it:

```
[Serializable]
public class Person
{
    public string Name { get; set; }
    public int Age = 20;
    public Address HomeAddress { get; set;}
    private string _thisWillGetWrittenToTheFileToo = "even though it is a private variable.";

    [NonSerialized]
    public string ThisWillNotBeWrittenToTheFile = "because of the [NonSerialized] attribute.";
}

[Serializable]
public class Address
{
    public string StreetAddress { get; set; }
    public string City { get; set; }
}

// And then in some function.
Person person = new Person() { Name = "Dan", Age = 30; HomeAddress = new Address() { StreetAddress = "123 My St", City = "Regina"
}};
List<Person> people = GetListOfPeople();
BinarySerialization.WriteToBinaryFile<Person>("C:\person.bin", person);
BinarySerialization.WriteToBinaryFile<List<People>>("C:\people.bin", people);

// Then in some other function.
Person person = BinarySerialization.ReadFromBinaryFile<Person>("C:\person.bin");
List<Person> people = BinarySerialization.ReadFromBinaryFile<List<Person>>("C:\people.bin");
```

Writing and Reading an object to / from an XML file (Using System.Xml.Serialization.XmlSerializer in the System.Xml assembly)

- Only writes and reads the Public properties and variables to / from the file.
- Classes to be serialized must contain a public parameterless constructor.
- The data saved to the file is human readable, so it can easily be edited outside of your application.
- Use the **[XmlIgnore]** attribute to exclude a public property or variable from being written to the file.

```
/// <summary>
/// Functions for performing common XML Serialization operations.
/// <para>Only public properties and variables will be serialized.</para>
/// <para>Use the [XmlIgnore] attribute to prevent a property/variable from being serialized.</para>
/// <para>Object to be serialized must have a parameterless constructor.</para>
/// </summary>
public static class XmlSerialization
{
    /// <summary>
    /// Writes the given object instance to an XML file.
    /// <para>Only Public properties and variables will be written to the file. These can be any type though, even other classes.
</para>
    /// <para>If there are public properties/variables that you do not want written to the file, decorate them with the
[XmlIgnore] attribute.</para>
    /// <para>Object type must have a parameterless constructor.</para>
    /// </summary>
    /// <typeparam name="T">The type of object being written to the file.</typeparam>
    /// <param name="filePath">The file path to write the object instance to.</param>
    /// <param name="objectToWrite">The object instance to write to the file.</param>
    /// <param name="append">If false the file will be overwritten if it already exists. If true the contents will be appended to
the file.</param>
    public static void WriteToXmlFile<T>(string filePath, T objectToWrite, bool append = false) where T : new()
    {
        TextWriter writer = null;
        try
        {
            var serializer = new XmlSerializer(typeof(T));
            writer = new StreamWriter(filePath, append);
            serializer.Serialize(writer, objectToWrite);
        }
        finally
        {
            if (writer != null)
                writer.Close();
        }
    }

    /// <summary>
    /// Reads an object instance from an XML file.
    /// <para>Object type must have a parameterless constructor.</para>
    /// </summary>
    /// <typeparam name="T">The type of object to read from the file.</typeparam>
    /// <param name="filePath">The file path to read the object instance from.</param>
    /// <returns>Returns a new instance of the object read from the XML file.</returns>
    public static T ReadFromXmlFile<T>(string filePath) where T : new()
    {
        TextReader reader = null;
        try
        {
            var serializer = new XmlSerializer(typeof(T));
            reader = new StreamReader(filePath);
            return (T)serializer.Deserialize(reader);
        }
        finally
        {
            if (reader != null)
                reader.Close();
        }
    }
}
```

And here is an example of how to use it:

```
public class Person
{
    public string Name { get; set; }
    public int Age = 20;
    public Address HomeAddress { get; set;}
    private string _thisWillNotGetWrittenToTheFile = "because it is not public.";

    [XmlIgnore]
    public string ThisWillNotBeWrittenToTheFile = "because of the [XmlIgnore] attribute.";
}

public class Address
{
    public string StreetAddress { get; set; }
    public string City { get; set; }
}

// And then in some function.
Person person = new Person() { Name = "Dan", Age = 30; HomeAddress = new Address() { StreetAddress = "123 My St", City = "Regina"
}};
List<Person> people = GetListOfPeople();
XmlSerialization.WriteToXmlFile<Person>("C:\person.txt", person);
XmlSerialization.WriteToXmlFile<List<People>>("C:\people.txt", people);

// Then in some other function.
Person person = XmlSerialization.ReadFromXmlFile<Person>("C:\person.txt");
List<Person> people = XmlSerialization.ReadFromXmlFile<List<Person>>("C:\people.txt");
```

Writing and Reading an object to / from a Json file (using the Newtonsoft.Json assembly in the Json.NET NuGet package)

- Only writes and reads the Public properties and variables to / from the file.
- Classes to be serialized must contain a public parameterless constructor.
- The data saved to the file is human readable, so it can easily be edited outside of your application.
- Use the **[JsonIgnore]** attribute to exclude a public property or variable from being written to the file.

```
/// <summary>
/// Functions for performing common Json Serialization operations.
/// <para>Requires the Newtonsoft.Json assembly (Json.Net package in NuGet Gallery) to be referenced in your project.</para>
/// <para>Only public properties and variables will be serialized.</para>
/// <para>Use the [JsonIgnore] attribute to ignore specific public properties or variables.</para>
/// <para>Object to be serialized must have a parameterless constructor.</para>
/// </summary>
public static class JsonSerialization
{
    /// <summary>
    /// Writes the given object instance to a Json file.
    /// <para>Object type must have a parameterless constructor.</para>
    /// <para>Only Public properties and variables will be written to the file. These can be any type though, even other classes.
</para>
    /// <para>If there are public properties/variables that you do not want written to the file, decorate them with the
[JsonIgnore] attribute.</para>
    /// </summary>
    /// <typeparam name="T">The type of object being written to the file.</typeparam>
    /// <param name="filePath">The file path to write the object instance to.</param>
    /// <param name="objectToWrite">The object instance to write to the file.</param>
    /// <param name="append">If false the file will be overwritten if it already exists. If true the contents will be appended to
the file.</param>
    public static void WriteToJsonFile<T>(string filePath, T objectToWrite, bool append = false) where T : new()
    {
        TextWriter writer = null;
        try
        {
            var contentsToWriteToFile = Newtonsoft.Json.JsonConvert.SerializeObject(objectToWrite);
            writer = new StreamWriter(filePath, append);
            writer.Write(contentsToWriteToFile);
        }
        finally
        {
            if (writer != null)
                writer.Close();
        }
    }

    /// <summary>
    /// Reads an object instance from an Json file.
    /// <para>Object type must have a parameterless constructor.</para>
    /// </summary>
    /// <typeparam name="T">The type of object to read from the file.</typeparam>
    /// <param name="filePath">The file path to read the object instance from.</param>
    /// <returns>Returns a new instance of the object read from the Json file.</returns>
    public static T ReadFromJsonFile<T>(string filePath) where T : new()
    {
        TextReader reader = null;
        try
        {
            reader = new StreamReader(filePath);
            var fileContents = reader.ReadToEnd();
            return Newtonsoft.Json.JsonConvert.DeserializeObject<T>(fileContents);
        }
        finally
        {
            if (reader != null)
```

```
                reader.Close();
            }
        }
    }
```

And here is an example of how to use it:

```
public class Person
{
    public string Name { get; set; }
    public int Age = 20;
    public Address HomeAddress { get; set;}
    private string _thisWillNotGetWrittenToTheFile = "because it is not public.";

    [JsonIgnore]
    public string ThisWillNotBeWrittenToTheFile = "because of the [JsonIgnore] attribute.";
}

public class Address
{
    public string StreetAddress { get; set; }
    public string City { get; set; }
}

// And then in some function.
Person person = new Person() { Name = "Dan", Age = 30; HomeAddress = new Address() { StreetAddress = "123 My St", City = "Regina"
}};
List<Person> people = GetListOfPeople();
JsonSerialization.WriteToJsonFile<Person>("C:\person.txt", person);
JsonSerialization.WriteToJsonFile<List<People>>("C:\people.txt", people);

// Then in some other function.
Person person = JsonSerialization.ReadFromJsonFile<Person>("C:\person.txt");
List<Person> people = JsonSerialization.ReadFromJsonFile<List<Person>>("C:\people.txt");
```

As you can see, the Json example is almost identical to the Xml example, with the exception of using the [JsonIgnore] attribute instead of [XmlIgnore].

Writing and Reading an object to / from a Json file (using the JavaScriptSerializer in the System.Web.Extensions assembly)

There are many Json serialization libraries out there. I mentioned the Newtonsoft.Json one because it is very popular, and I am also mentioning this JavaScriptSerializer one because it is built into the .Net framework. The catch with this one though is that it requires the Full .Net 4.0 framework, not just the .Net Framework 4.0 Client Profile.

The caveats to be aware of are the same between the Newtonsoft.Json and JavaScriptSerializer libraries, except instead of using [JsonIgnore] you would use **[ScriptIgnore]**.

Be aware that the JavaScriptSerializer is in the System.Web.Extensions assembly, but in the System.Web.Script.Serialization namespace. Here is the code from the Newtonsoft.Json code snippet that needs to be replaced in order to use the JavaScriptSerializer:

```
// In WriteFromJsonFile<T>() function replace:
var contentsToWriteToFile = Newtonsoft.Json.JsonConvert.SerializeObject(objectToWrite);
// with:
var contentsToWriteToFile = new System.Web.Script.Serialization.JavaScriptSerializer().Serialize(objectToWrite);

// In ReadFromJsonFile<T>() function replace:
return Newtonsoft.Json.JsonConvert.DeserializeObject<T>(fileContents);
// with:
return new System.Web.Script.Serialization.JavaScriptSerializer().Deserialize<T>(fileContents);
```

Happy Coding!