Working with External Text Files and XML Data

In this chapter, we will cover:

- Loading external text files using the TextAsset public variable
- Loading external text files using C# file streams
- Saving external text files with C# file streams
- Loading and parsing external XML files
- Creating XML text data manually using XMLWriter
- Creating XML text data automatically through serialization
- Creating XML text files – saving XML directly to text files with XMLDocument.Save()

Introduction

Text-based external data is very common and very useful as it is both computer and human readable. Text files may be used to allow non-technical team members to edit written content or for recording game performance data during development and testing. There is a lot of XML-based text file data on the Web, which is why we have included several XML-specific file reading and writing methods.

In the previous chapter, three general methods for loading external resource files were demonstrated, all of which work for text files as well as textures and audio files. Several additional methods for loading text files in particular are presented in this chapter.

All the recipes in this chapter are closely related.

Loading external text files using the TextAsset public variable

A straightforward way to store data in text files and then choose between them before compiling is to use a public variable of the class `TextAsset`.

This technique is only appropriate when there will be no change to the data file after game compilation.

Getting ready

For this recipe, you'll need a text (`.txt`) file. In the `1362_bonus_01` folder, we have provided two such files:

- `cities.txt`
- `countries.txt`

How to do it...

To load external text files using `TextAsset`, perform the following steps:

1. Import the text file you wish to use into your project (for example, `cities.txt`)
2. Add the following C# script to **Main Camera**:

```
// file: ReadPublicTextAsset.cs
using UnityEngine;
using System.Collections;

public class ReadPublicTextAsset : MonoBehaviour {
    public TextAsset dataTextFile;
    private string textData = "";

    private void Start() {
        textData = dataTextFile.text;
    }

    private void OnGUI() {
        GUILayout.Label ( textData );
    }
}
```

3. With **Main Camera** selected in the **Hierarchy** view, drag the `cities.txt` file into the public string variable **dataTextFile** in the **Inspector** view.

How it works...

The contents of the text file are read from the `dataTextFile TextAsset` object and stored as a string into `textData`. Our `OnGUI()` method displays the contents of `textData` as a label.

Loading external text files using C# file streams

For standalone executable games that both read from and write to (create or change) text files, .NET data streams are often used for both reading and writing. A later recipe illustrates how to write text data to files, while this recipe illustrates how to read a text file.

This technique only works when you compile to a Windows or Mac *standalone executable*; it will not work for Web Player builds, for example.

Getting ready

For this recipe, you'll need a text file; two of these have been provided in the `1362_bonus_01` folder.

How to do it...

To load external text files using C# file streams, perform the following steps:

1. Create a new C# script called `FileReadWriteManager`:

```
// file: FileReadWriteManager.cs
using System;
using System.IO;

public class FileReadWriteManager {
    public void WriteTextFile(string pathAndName, string
stringData) {
        // remove file (if exists)
        FileInfo textFile = new FileInfo(  pathAndName );
        if( textFile.Exists )
            textFile.Delete();

        // create new empty file
        StreamWriter writer;
        writer = textFile.CreateText();

        // write text to file
```

```
            writer.Write(stringData);

            // close file
            writer.Close();
        }

        public string ReadTextFile(string pathAndName) {
            string dataAsString = "";

            try {
                // open text file
                StreamReader textReader = File.OpenText( pathAndName
);

                // read contents
                dataAsString = textReader.ReadToEnd();

                // close file
                textReader.Close();

            }
            catch (Exception e) {
                //display/set e.Message error message here if you wish

            }

            // return contents
            return dataAsString;
        }
}
```

2. Add the following C# script to **Main Camera**:

```
// file: ReadWithStream.cs
using UnityEngine;
using System.Collections;
using System.IO;

public class ReadWithStream : MonoBehaviour {
    private string filePath = "";
    private string textFileContents = "(file not found yet)";
    private FileReadWriteManager fileReadWriteManager = new
FileReadWriteManager();

    private void Start () {
        string fileName = "cities.txt";
        filePath = Path.Combine(Application.dataPath,
"Resources");
```

```
        filePath = Path.Combine(filePath, fileName);

        textFileContents = fileReadWriteManager.ReadTextFile(
filePath );
    }

    private void OnGUI() {
        GUILayout.Label ( filePath );
        GUILayout.Label ( textFileContents );
    }
}
```

3. Build your (Windows, Mac, or Linux) standalone executable. You'll need to save the current scene and then add this to the scenes in the build.
4. Copy the text file containing your data into your standalone's `Resources` folder (that is, the filename you set in the first statement in the `Start()` method—in our listing, this is the `cities.txt` file)

You will need to place the files in the `Resources` folder manually after every compilation.

For Windows and Linux users: When you create a Windows or Linux standalone executable, there is a `_Data` folder that is created with the executable application file. The `Resources` folder can be found inside this data folder.

For Mac users: A Mac standalone application executable looks like a single file, but it is actually a Mac OS "package" folder. Right-click on the executable file and select **Show Package Contents**. You will then find the standalone's `Resources` folder inside the `Contents` folder.

How it works...

When the game runs, the `Start()` method creates the `filePath` string and then calls the `ReadTextFile()` method from the `fileReadWriteManager` object, to which it passes it the `filePath` string. This method reads the contents of the file and returns them as a string, which is stored in the `textFileContents` variable. Our `OnGUI()` method displays the values of these two variables (`filePath` and `textFileContents`).

Note the need to use the `System.IO` package for this recipe. The C# script `FileReadWriteManager.cs` contains two general purpose file read and write methods that you may find useful in many different projects.

Saving external text files with C# file streams

This recipe illustrates how to use C# streams to write text data to a text file, either into the standalone project's `Data` folder or to the `Resources` folder.

This technique only works when you compile to a Windows or Mac standalone executable.

Getting ready

In the `1362_bonus_02` folder, we have provided the `FileReadWriteManager.cs` class script file:

How to do it...

To save external text files using C# file streams, follow these steps:

1. Import the C# script `FileReadWriteManager.cs` into your project.
2. Add the following C# script to **Main Camera**:

```
// file: SaveTextFile.cs
using UnityEngine;
using System.Collections;
using System.IO;

public class SaveTextFile : MonoBehaviour {
    public string fileName = "hello.txt";
    public string folderName = "Data";
    private string filePath = "(no file path yet)";
    private string message = "(trying to save data)";
    private FileReadWriteManager fileManager;

    void Start () {
        fileManager = new FileReadWriteManager();
        filePath = Application.dataPath + Path.
DirectorySeparatorChar + fileName;

        string textData = "hello \n and goodbye";
        fileManager.WriteTextFile( filePath, textData );

        message = "file should have been written now ...";
```

```
    }

    void OnGUI()
    {
        GUILayout.Label("filepath = " + filePath);
        GUILayout.Label("message = " + message);
    }
}
```

3. Build and run your (Windows or Mac) standalone executable. You'll need to save the current scene and then add this to the scenes in the build.
4. You should now find a new text file named `hello.txt` in the `Data` folder of your project's standalone files, containing the lines "hello" and " and goodbye".

It is possible to test this when running within the *Unity editor* (that is, before building a standalone application). In order to test this way, you'll need to create a `Data` folder in your project panel.

How it works...

When the game runs, the `Start()` method creates the `filePath` string from the public variables `fileName` and `folderName`, and then calls the `WriteTextFile()` method from the `fileReadWriteManager` object, to which it passes the `filePath` and `textData` strings. This method creates (or overwrites) a text file (for the given file path and filename) containing the string data received.

There's more...

Some details you don't want to miss:

Choosing the Data or the Resources folder

Standalone build applications contain both a `Data` folder and a `Resources` folder. Either of these can be used for writing (or some other folder, if desired). We generally put read-only files into the `Resources` folder and use the `Data` folder for files that are to be created from scratch or have had their contents changed.

Loading and parsing external XML files

XML is a common data exchange format; it is useful to be able to parse (process the contents of) text files and strings containing data in this format. C# offers a range of classes and methods to make such processing straightforward.

Getting ready

You'll find player name and score data in XML format in the `playerScoreData.xml` file in the `1362_bonus_04` folder. The contents of this file are as follows:

```
<scoreRecordList>
    <scoreRecord>
        <player>matt</player>
        <score>2200</score>
        <date>
            <day>1</day>
            <month>Sep</month>
            <year>2012</year>
        </date>
    </scoreRecord>

    <scoreRecord>
        <player>jane</player>
        <score>500</score>
        <date>
            <day>12</day>
            <month>May</month>
            <year>2012</year>
        </date>
    </scoreRecord>
</scoreRecordList>
```

The data is structured by a root element named `scoreRecordList`, which contains a sequence of `scoreRecord` elements. Each `scoreRecord` element contains a `player` element (which contains a player's name), a `score` element (which has the integer content of the player's score), and a `date` element, which itself contains three child elements, `day`, `month`, and `year`.

How to do it...

To load and parse external XML files, follow these steps:

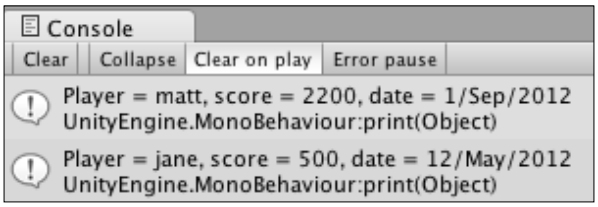
1. Add the following C# script to **Main Camera**:

```
// file: ParseXML.cs
using UnityEngine;
using System.Collections;

using System.Xml;
using System.IO;
```

```
public class ParseXML : MonoBehaviour {
    public TextAsset scoreDataTextFile;

    private void Start() {
        string textData = scoreDataTextFile.text;
        ParseScoreXML( textData );
    }

    private void ParseScoreXML(string xmlData) {
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.Load( new StringReader(xmlData) );

        string xmlPathPattern = "//scoreRecordList/scoreRecord";
        XmlNodeList myNodeList = xmlDoc.SelectNodes(
xmlPathPattern );

        foreach(XmlNode node in myNodeList)
            print ( ScoreRecordString( node ) );

    }

    private string ScoreRecordString(XmlNode node) {
        XmlNode playerNode = node.FirstChild;
        XmlNode scoreNode = playerNode.NextSibling;
        XmlNode dateNode = scoreNode.NextSibling;

        return "Player = " + playerNode.InnerXml + ", score = " +
scoreNode.InnerXml + ", date = " + DateString( dateNode );
    }

    private string DateString(XmlNode dateNode) {
        XmlNode dayNode = dateNode.FirstChild;
        XmlNode monthNode = dayNode.NextSibling;
        XmlNode yearNode = monthNode.NextSibling;

        return dayNode.InnerXml + "/" + monthNode.InnerXml + "/" +
yearNode.InnerXml;
    }
}
```

2. Import the playerScoreData.xml file into your project.
3. Select the Main Camera and set playerScoreData.xml as the Score Data Text File variable for the Parse XML component.

4. The output of the `print()` statements should be visible in the **Console** window:



How it works...

Note the need to use the `System.Xml` and `System.IO` packages for this recipe.

The `text` property of the `TextAsset` variable `scoreDataTextFile` provides the contents of the XML file as a string, which is passed to the `ParseScoreXML()` method. This method creates a new `XmlDocument` variable with the contents of this string. The `XmlDocument` class provides the `SelectNodes()` method, which returns a list of node objects for a given element path. In this example, a list of `scoreRecord` nodes is requested. A `for-each` statement prints out the string returned by the `ScoreRecordString()` method when each node object is passed to it.

The `ScoreRecordString()` method relies on the ordering of the XML elements to retrieve the player's name and score, and it gets the date as a slash-separated string by passing the node containing the three date components to the `DateString()` method.

There's more...

Some details you don't want to miss:

Retrieving XML data files from the web

You can use the `WWW` Unity class if the XML file is located on the Web rather than in your Unity project.

Creating XML text data manually using XMLWriter

One way to create XML data structures from game objects and properties is by hand-coding a method to create each element and its contents, using the `XMLWriter` class.

How to do it...

To create XML text data using `XMLWriter`, follow these steps:

1. Add the following C# script to **Main Camera**:

```
// file: CreateXMLString.cs
using UnityEngine;
using System.Collections;
using System.Xml;
using System.IO;

public class CreateXMLString : MonoBehaviour {
    private string output = "(nothing yet)";

    private void Start () {
        output = BuildXMLString();
    }

    private void OnGUI() {
        GUILayout.Label( output );
    }

    private string BuildXMLString() {
        StringWriter str = new StringWriter();
        XmlTextWriter xml = new XmlTextWriter(str);

        // start doc and root el.
        xml.WriteStartDocument();
        xml.WriteStartElement("playerScoreList");

        // data element
        xml.WriteStartElement("player");
        xml.WriteElementString("name", "matt");
        xml.WriteElementString("score", "200");
        xml.WriteEndElement();

        // data element
        xml.WriteStartElement("player");
        xml.WriteElementString("name", "jane");
        xml.WriteElementString("score", "150");
        xml.WriteEndElement();

        // end root and document
        xml.WriteEndElement();
```

```
            xml.WriteEndDocument();
            return str.ToString();
        }
    }
```

2. The XML text data should be visible when the game is run:

How it works...

The `Start()` method calls `BuildXMLString()` and stores the returned string in the `output` variable. Our `OnGUI()` method displays the contents of `output`.

The `BuildXMLString()` method creates a `StringWriter` object, into which `XMLWriter` builds the string of XML elements. The XML document starts and ends with the `WriteStartDocument()` and `WriteEndDocument()` methods. Elements start and end with `WriteStartElement(<elementName>)` and `WriteEndElement()`. String content for an element is added using `WriteElementString()`.

There's more...

Some details you don't want to miss:

Adding newlines to make XML strings more human readable

After every instance of the `WriteStartElement()` and `WriteElementString()` methods, you can add a newline character using `WriteWhiteSpace()`. These are ignored by XML parsing methods, but if you intend to display the XML string for a human to see, the presence of the newline characters makes it much more readable:

```
xml.WriteWhitespace("\n  ");
```

Making data class responsible for creating XML from list

The XML to be generated is often from a list of objects, all of the same class. In this case, it makes sense to make the class of the objects responsible for generating the XML for a list of those objects.

The `CreateXMLFromArray` class simply creates an instance of `ArrayList` containing `PlayerScore` objects, and then calls the (static) method `ListToXML()`, passing in the list of objects.

```
// file: CreateXMLFromArray.cs
using UnityEngine;
using System.Collections;

public class CreateXMLFromArray : MonoBehaviour {
    private string output = "(nothing yet)";
    private ArrayList myPlayerList;

    private void Start () {
        myPlayerList = new ArrayList();
        myPlayerList.Add (new PlayerScore("matt", 200) );
        myPlayerList.Add (new PlayerScore("jane", 150) );

        output = PlayerScore.ListToXML( myPlayerList );
    }

    private void OnGUI() {
        GUILayout.Label( output );
    }
}
```

All the hard work is now the responsibility of the `PlayerScore` class. This class has two private variables for the players' name and score and a constructor that accepts values for these properties. The `public static` method `ListToXML()` takes an `ArrayList` object as an argument, and uses `XMLWriter` to build the XML string, looping through each object in the list and calling the object's `ObjectToElement()` method. This method adds an XML element to the `XMLWriter` argument received for the data in that object.

```
// file: PlayerScore.cs
using System.Collections;
using System.Xml;
using System.IO;

public class PlayerScore {
    private string _name;
    private int _score;

    public PlayerScore(string name, int score) {
        _name = name;
        _score = score;
    }
```

```
    // class method
    static public string ListToXML(ArrayList playerList) {
        StringWriter str = new StringWriter();
        XmlTextWriter xml = new XmlTextWriter(str);

        // start doc and root el.
        xml.WriteStartDocument();
        xml.WriteStartElement("playerScoreList");

        // add elements for each object in list
        foreach (PlayerScore playerScoreObject in playerList) {
            playerScoreObject.ObjectToElement( xml );
        }

        // end root and document
        xml.WriteEndElement();
        xml.WriteEndDocument();

        return str.ToString();
    }

    private void ObjectToElement(XmlTextWriter xml) {
        // data element
        xml.WriteStartElement("player");
        xml.WriteElementString("name", _name);
        string scoreString = "" + _score; // make _score a string
        xml.WriteElementString("score", scoreString);
        xml.WriteEndElement();
    }
}
```

Creating XML text data automatically through serialization

Another way to create XML data structures from game objects and properties is by serializing the contents of an object automatically. This technique automatically generates XML for all the public properties of an object.

Getting ready

In the `1362_bonus_06` folder, you'll find the listings for all four classes described in this recipe.

How to do it...

To create XML text data through serialization, perform the following steps:

1. Create a C# script called `PlayerScore`:

```
// file: PlayerScore.cs
public class PlayerScore
{
    public string name;
    public int score;

    // default constructor, needed for serialization
    public PlayerScore() {}

    public PlayerScore(string newName, int newScore) {
        name = newName;
        score = newScore;
    }
}
```

2. Create a C# script called `SerializeManager`:

```
// file: SerializeManager.cs
//
// acknowledgements - this code has been adapted from:
//www.eggheadcafe.com/articles/system.xml.xmlserialization.asp
using System.Xml;
using System.Xml.Serialization;
using System.IO;
using System.Text;
using System.Collections.Generic;

public class SerializeManager<T> {
    public string SerializeObject(T pObject) {
        string XmlizedString = null;
        MemoryStream memoryStream = new MemoryStream();
        XmlSerializer xs = new XmlSerializer(typeof(T));
        XmlTextWriter xmlTextWriter = new
XmlTextWriter(memoryStream, Encoding.UTF8);
        xs.Serialize(xmlTextWriter, pObject);
        memoryStream = (MemoryStream)xmlTextWriter.BaseStream;
        XmlizedString = UTF8ByteArrayToString(memoryStream.
ToArray());
        return XmlizedString;
    }
```

```
    public object DeserializeObject(string pXmlizedString) {
        XmlSerializer xs = new XmlSerializer(typeof(T));
        MemoryStream memoryStream = new MemoryStream(StringToUTF8B
yteArray(pXmlizedString));
        return xs.Deserialize(memoryStream);
    }

    private string UTF8ByteArrayToString(byte[] characters) {
        UTF8Encoding encoding = new UTF8Encoding();
        string constructedString = encoding.GetString(characters);
        return (constructedString);
    }

    private byte[] StringToUTF8ByteArray(string pXmlString) {
        UTF8Encoding encoding = new UTF8Encoding();
        byte[] byteArray = encoding.GetBytes(pXmlString);
        return byteArray;
    }
}
```

3. Add the following C# script class to **Main Camera**:

```
// file: SerialiseToXML.cs
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class SerialiseToXML : MonoBehaviour {
    private string output = "(nothing yet)";

    void Start () {
        SerializeManager<PlayerScore> serializer = new SerializeMa
nager<PlayerScore>();
        PlayerScore myData = new PlayerScore("matt", 200);
        output = serializer.SerializeObject(myData);
    }

    void OnGUI() {
        GUILayout.Label( output );
    }
}
```

4. Run your game. You should see XML data displayed as a label on the screen.

How it works...

The `Start()` method creates `serializer`, a new `SerializeManager()` object for objects of the `PlayerScore` class. A new `PlayerScore` object, `myData`, is created with the values `"Matt"` and `200`. The `serializer` object is then passed the `myData` object, and the XML text data is returned as a string and stored in `output`. Our `OnGUI()` method displays the contents of `output`.

The `SerializeManager` class has been adapted from the code published by EggHeadCafe at the following URL:

`http://www.eggheadcafe.com/articles/20031219.asp`

You shouldn't need to worry too much about understanding all the code in this class—unless you want to :-). We have adapted the class so that it can be used for any data object class via C# generics; all you need to do is the following:

- Replace `<PlayerScore>` with `<YourDataClassName>` when declaring and creating an instance of `SerializationManager`; in our example, this is in the first statement of the `Start()` method
- Pass in an object of your class as the argument to `serializer.SerializeObject()`
- Ensure that your data class has a default constructor (that is public and takes no arguments)
- Make sure each property that you want included in the generated XML is public

The following site is a good place to learn more about Unity XML serialization:

`http://wiki.unity3d.com/index.php/Saving_and_Loading_Data:_XmlSerializer`

There's more...

Some details you don't want to miss:

Protecting member properties with accessor methods

Just having public properties is generally considered poor practice, because any part of an application with a reference to an object can make any kind of changes to those properties. However, if you wish to use this serialization recipe, you have to have public properties.

A good solution is to provide public accessor methods, which behave like public properties but allow you to add a validation code behind the get/set methods for each corresponding hidden private property. Here, we have provided such an improved implementation of the `PlayerScore` class, `PlayerScore2`. Data is stored in the private variables `_name` and `_score`; they are accessed via the public variables `Name` and `Score`, which have get/set statements that handle changes to the private variables, and for which the appropriate validation logic could be implemented (for example, only changing `Score` if the new score is zero or higher, to prevent negative scores):

```
// file: PlayerScore2.cs
using System.Xml.Serialization;

[XmlRoot("player_score_2")]
public class PlayerScore2
{
    private string _name;
    private int _score;

    [XmlElement("name")]
    public string Name {
        get{ return _name; }
        set{ _name = value; }
    }

    [XmlElement("score")]
    public int Score {
        get{ return _score;}
        set{ _score = value;}
    }

    // default constructor, needed for serialization
    public PlayerScore2() {}

    public PlayerScore2(string newName, int newScore) {
        Name = newName;
        Score = newScore;
    }
}
```

Variable naming styles vary between programmers and organizations. One approach to naming private member variables is to prefix them with an underscore (as we have done in the previous code snippet). An alternative, which some programmers prefer, involves using no underscore and disambiguating the argument of a setter method from the member variable with the same identifier by using keyword `this`; for example, `this.name = name`.

Creating XML text files – saving XML directly to text files with XMLDocument.Save()

It is possible to create an XML data structure and then save that data directly to a text file using the `XMLDocument.Save()` method; this recipe illustrates how.

How to do it...

To save XML data to text files directly, perform the following steps:

1. Create a new C# script named `PlayerXMLWriter`:

```
// file: PlayerXMLWriter.cs
using System.Text;
using System.Xml;
using System.IO;

public class PlayerXMLWriter {
    private string _filePath;
    private XmlDocument _xmlDoc;
    private XmlElement _elRoot;

    public PlayerXMLWriter(string filePath) {
        _filePath = filePath;
        _xmlDoc = new XmlDocument();

        if(File.Exists (_filePath)) {
            _xmlDoc.Load(_filePath);
            _elRoot = _xmlDoc.DocumentElement;
            _elRoot.RemoveAll();
        }
        else {
            _elRoot = _xmlDoc.CreateElement("playerScoreList");
            _xmlDoc.AppendChild(_elRoot);
        }
    }

    public void SaveXMLFile() {
        _xmlDoc.Save(_filePath);
    }
    public void AddXMLElement(string playerName, string
playerScore) {
        XmlElement elPlayer = _xmlDoc.
CreateElement("playerScore");
```

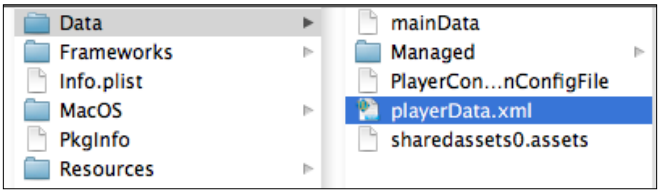
```
            _elRoot.AppendChild(elPlayer);

            XmlElement elName = _xmlDoc.CreateElement("name");
            elName.InnerText = playerName;
            elPlayer.AppendChild(elName);

            XmlElement elScore = _xmlDoc.CreateElement("score");
            elScore.InnerText = playerScore;
            elPlayer.AppendChild(elScore);
        }
}
```

2. Add the following C# script to **Main Camera**:

```
// file: CreateXMLTextFile.cs
using UnityEngine;
using System.Collections;
using System.IO;

public class CreateXMLTextFile : MonoBehaviour {
    public string fileName = "playerData.xml";
    public string folderName = "Data";

    private void Start() {
        string filePath = Application.dataPath + Path.
DirectorySeparatorChar + fileName;

        PlayerXMLWriter myPlayerXMLWriter = new
PlayerXMLWriter(filePath);
        myPlayerXMLWriter.AddXMLElement("matt", "55");
        myPlayerXMLWriter.AddXMLElement("jane", "99");
        myPlayerXMLWriter.AddXMLElement("fred", "101");
        myPlayerXMLWriter.SaveXMLFile();

        print( "XML file should now have been created at: " +
filePath);
    }
}
```

3. You can quickly test the scene in the Unity Editor if you create a folder 'Data' in the Project panel and then run the scene. After 10-20 seconds you should now find that text file playerData.xml has been created in folder Data.
4. Build and run your (Windows, Mac, or Linux) standalone executable. You'll need to save the current scene and then add this to the list of scenes in the build.

5. You should now find a new text file named `playerData.xml` in the `Data` folder of your project's standalone files, containing the XML data for the three players:



6. The contents of the `playerData.xml` file should be the XML player list data:

```
<playerScoreList>
  <playerScore>
    <name>matt</name>
    <score>55</score>
  </playerScore>
  <playerScore>
    <name>jane</name>
    <score>99</score>
  </playerScore>
  <playerScore>
    <name>fred</name>
    <score>101</score>
  </playerScore>
</playerScoreList>
```

How it works...

The `Start()` method creates `myPlayerXMLWriter`, a new object of the `PlayerXMLWriter` class, to which it passes the new, required XML text file `filePath` as an argument. Three elements are added to the `PlayerXMLWriter` object, which store the names and scores of three players. The `SaveXMLFile()` method is called and a debug `print()` message is displayed.

The `PlayerXMLWriter` class works as follows: when a new object is created, the provided file path string is stored in a private variable; at the same time, a check is made to see whether any file already exists. If an existing file is found, the content elements are removed; if no existing file is found, then a new root element, `playerScoreList`, is created as the parent for child data nodes. The `AddXMLElement()` method appends a new data node for the provided player name and score. The `SaveXMLFile()` method saves the XML data structure as a text file for the stored file path string.