# 8

# External text files and XML data

In this chapter, we will cover:

## Introduction

Text-based external data is very common and very useful, since it is both computer and human readable. Text files may be used to allow non-technical team members to edit written content, or may be useful for recording game performance data during development and testing. There is a lot of XML-based text file data on the web, and so 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 of loading text files in particular are presented in this chapter.

All the recipes in this chapter are closely related.

## Loading external text files – by 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 class TextAsset.

Note: 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 folder 0423_08_01 we have provided two:

- cities.txt
- countries.txt

### How to do it...

1 – Import into your project the text file you wish to use (e.g. *cities.txt*)

2 – Add the following C# script to the *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 the *Main Camera* selected in the *Hierarchy*, drag the *cities.txt* text file into the public string variable *dataTextFile* in the *Inspector*

## How it works...

The text contents of the text file are read from the *dataTextFile* TextAsset object, and stored as a string into *textData*. Our OnGU() method displays the contents of *textData* as a Label().

## Loading external text files – by C# file streams

For standalone executable games that both read from and write (create or change) the contents of text files, the use of .net data streams is 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.

NOTE: This technique only works when you compile to a Windows or Mac *stand alone executable* – it will not work for Web Player builds for example.

### Getting ready

For this recipe, you'll need a text (.txt) file; two were provided in folder 0423_08_01.

### How to do it...

1 – Create a new C# script *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 the 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 );
    }
}
```

2 – Build your (Windows or Mac or Linux) standalone executable. You'll need to save the current scene, and add this to the scenes in the build.

3 – Copy the text file containing your text data into your standalone's *Resources* folder (i.e. the file whose name you set in the first statement in method *Start*() – in the listing above this is file *cities.txt*)

> NOTE: You will need to place the files in the Resources folder manually after **every** compilation.
>
> WINDOWS and LINUX: When you create a Windows or Linux standalone executable, there is also a _Data folder created with executable application file. The Resources folder can be found inside this data folder.
>
> MAC: A Mac standalone application executable looks like single file, but it is actually a MacOS 'package' folder. Right-click the executable file and select Show Package Contents. You will then find the standalone's Resources folder inside the Contents folder.

## How it works...

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, which you may find useful in many different projects.

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

## There's more...

Some details you don't want to miss:

## Saving external text files with C# file streams

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

NOTE: This technique only works when you compile to a Windows or Mac *stand alone executable*.

### Getting ready

In folder 0423_08_02 we have provided the class script file:

- FileReadWriteManager.cs

### How to do it...

1 – Import into your project the C# script: *FileReadWriteManager.cs*

2 – Add the following C# script to the *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 = Path.Combine(Application.dataPath,
folderName);
      filePath = Path.Combine(filePath, 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 add this to the scenes in the build.

2 – Build your (Windows or Mac or Linux) standalone executable. You'll need to save the current scene, and 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"

NOTE:

It is possible to test this when running within **the Unity editor** (i.e. 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 frpom public variables *fileName* and *folderName*, and then calls method WriteTextFile() from the *fileReadWriteManager* object, passing it the *filePath* and *textData* strings. This method creates (or overwrites) a text file (for the given file path and file name) containing the string data received.

## There's more...

Some details you don't want to miss:

### Data or Resources folder?

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

## Loading and parsing external XML files

XML is a common data exchange format, so 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 file *playerScoreData.xml* in folder 0423_08_04. The contents of this file are as follows:

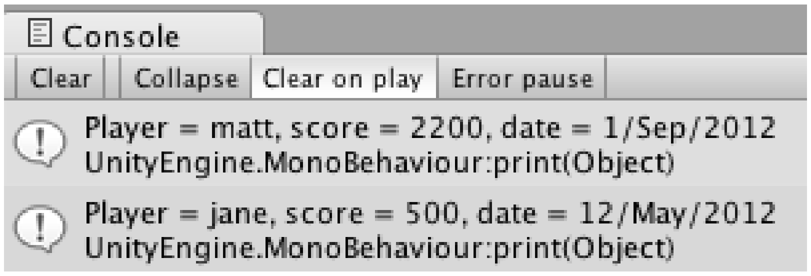
```
<scoreRecordList>
   <scoreRecord>
      <player>matt</player>
      <score>2200</score>
      <date>
         <day>1</day>
         <month>Sep</month>
         <year>2012</year>
      </date>
   </scoreRecord>

   <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 (with text content of player's name), a *score* element (with integer text content of player's score), and a *date* element, which itself contains 3 child-elements of *day*, *month* and *year*.

### How to do it...

1 – Add the following C# script to the *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;
   }
}
```



```
Player = matt, score = 2200, date = 1/Sep/2012
UnityEngine.MonoBehaviour:print(Object)
Player = jane, score = 500, date = 12/May/2012
UnityEngine.MonoBehaviour:print(Object)
```

0423_08_01.png

## 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 method *ParseScoreXML*(). This method creates a new XmlDocument variable with the contents of this string. The XmlDocument class provides the method SelectNodes() which returns a list of node objects for a given element path. In this example a list of *scoreRecord* nodes is requested. A ‘foreach’ statement prints out the string returned by the *ScoreRecordString*() method when passed each node object.

The *ScoreRecordString*() method relies on the ordering of the XML elements, to retrieve the player’s name and score, and gets the date as a slash-separated string by passing method *DateString*() the node containing the 3 date components.

## There's more...

Some details you don’t want to miss:

### Retrieving XML data files from the web

Of course 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 with XMLWriter

One method 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.

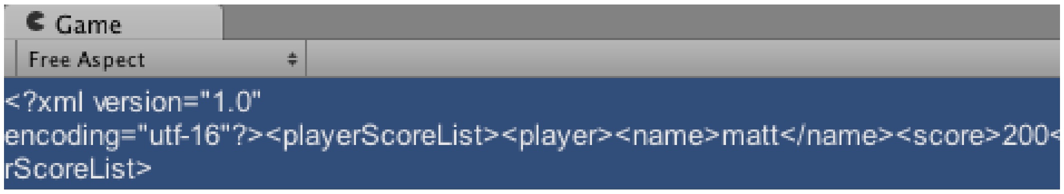
## Getting ready

## How to do it...

1 – Add the following C# script to the *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();
```

```
    }
}
```



```
<?xml version="1.0"
encoding="utf-16"?><playerScoreList><player><name>matt</name><score>200
rScoreList>
```

0423_08_02.png

## How it works...

Method Start() calls BuildXML() and stores the returned string in variable *output*. Our OnGUI() method displays the contents of *output*.

Method BuildXML() creates a StringWriter *str*, into which an XMLWriter will build the string of XML elements. The XML document is started and ended with WriteStartDocument() and WriteEndDocument(). Elements are started and ended with WriteStartElement( <elementName> ) and WriteEndElement(). String content for an element is added with WriteElementString().

## There's more...

Some details you don't want to miss:

### Adding newlines to make XML strings more human readable

After each WriteStartElement() and WriteElementString() you can add a newline character with WriteWhiteSpace(). These are ignored by XML parsing methods, but if displaying the XML string for a human to see, the newlines make it much more readable.

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

### Make data class responsible for creating XML from list

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

Class *CreateXMLFromArray* simply creates an ArrayList of *PlayerScore* objects, and then calls the class (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 name and score, and a constructor that accepts values for these properties. The public static method *ListToXML*() takes an ArrayList as an argument, and uses the XMLWriter to build the XML string, looping through for 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 method 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 folder 0423_08_06 you'll find the listings for all 4 classes described in this recipe.

### How to do it...

1 – Create C# script *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 *SerializeManager:*

```
// file: SerializeManager.cs
//
// acknowledgements - this code has been adapted from:
//www.eggheadcafe.com/articles/system.xml.xmlserializatio
n.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(StringToUTF8ByteArray(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 the *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
SerializeManager<PlayerScore>();
      PlayerScore myData = new PlayerScore("matt", 200);
      output = serializer.SerializeObject(myData);
   }

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

3 – Run your game. You should see XML data displayed as a Label on screen.

## How it works...

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

NOTE: The SerializationManager class has been adapted from the code published by the EggHeadCafe from the following URL.

- www.eggheadcafe.com/articles/system.xml.xmlserialization.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 your data class has a default constructor (i.e. public and taking no arguments)
- Ensure each property you wish to be included in the generated XML is public
    - See below for using accessor methods to protect such properties if needed

## There's more...

Some details you don't want to miss:

### Accessor methods to protect public member properties

Just having public properties is generally poor practice, since any part of an application with a reference to an object can make any kinds of changes to those properties. However, if you wish to use this serialisation recipe then you have to have public properties. A good solution is to provide public accessor methods (that behave as if public properties), but allowing you to add validation code behind get/set methods for each hidden corresponding private property. Here we provide such an improved implementation of the PlayerScore class PlayerScore2. Data is stored in private variables _*name* and _*score*; they are accessed via public variables *name* and *score*, which have get/set statements that handle changes to the private variables, and appropriate validation logic could be implemented there (e.g. only changing score if the new score is zero or greater, to prevent negative scores):

```
// file: PlayerScore2.cs
public class PlayerScore2
{
   private string _name;
   private int _score;

   public string name {
      get{ return _name; }
      set{ _name = value; }
   }

   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;
   }
}
```

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

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

### Getting ready

### How to do it...

Create a new C# script *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);
   }
}
```

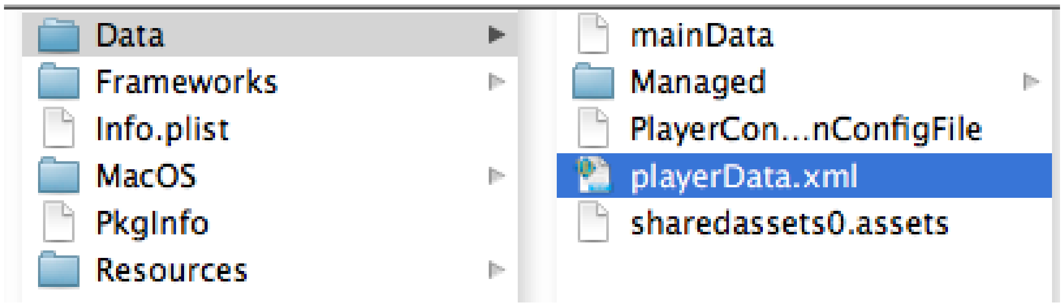
2 – Add the following C# script to the *Main Camera*:

```
// 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 = Path.Combine(
Application.dataPath, folderName);
      filePath = Path.Combine( filePath, fileName);
```

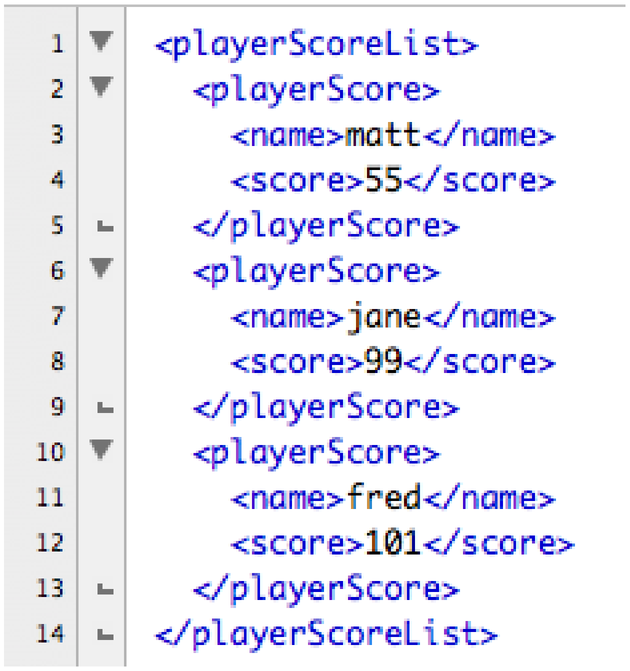
```
        PlayerXMLWriter myPlayerXMLWriter = new
PlayerXMLWriter(filePath);
        myPlayerXMLWriter.AddXMLElement("matt", "55");
        myPlayerXMLWriter.AddXMLElement("jane", "99");
        myPlayerXMLWriter.AddXMLElement("fred", "101");
        myPlayerXMLWriter.SaveXMLFile();

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

3 – Build and run your (Windows or Mac or Linux) standalone executable. You'll need to save the current scene, and add this to the scenes in the build.

4 – 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.



0423_08_03.png



```
<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>
```

0423_08_04.png

## How it works...

Method Start() creates *myPlayerXMLWriter* a new object of class *PlayerXMLWriter*; passing the desired new XML text file *filePath* as an argument. Three elements are added to the *PlayerXMLWriter* object, storing the names and scores of three players. The *SaveXMLFile()* method is called, and a debug *print()* message displayed.

The *PlayerXMLWriter* class works as follows. When a new object is created, the provided string file path is stored in a private variable, also a check is made to see whether any file exists already. If an existing file is found, any 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 to be created for. Method *AddXMLElement()* appends a new data node for the provided player name and score. Method *SaveXMLFile()* saves the XML data structure as a text file for the stored file path string.