1
Bonus Chapter 11: Working with Plain Text, XML, and JSON Text Files

In this chapter, we will cover the following topics:

- 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
- Saving and Loading XML text data automatically through serialization
- Creating XML text files, and saving XML directly to text files with XMLDocument.Save()
- Creating JSON strings from individual objects and lists of objects
- Creating individual objects and lists of objects from JSON strings

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. Text-based formats also permit **serialization**—the encoding of live object data suitable for transmission, storing, and later retrieval.

Unity treats all of the following (and also C# scripts) as **Text Assets**:

- `.txt`: Plain text file

- `.html`, `.htm`: HTML page markup (HyperText Markup Language)
- `.xml`: XML data (eXtensible Markup Language)
- `.bytes`: Binary data (accessed through bytes property)
- `.json`: JSON (JavaScript Object Notation)
- `.csv`: CSV (Comma Separate Variable)
- `.yaml`: YAML Ain't Markup Language
- `.fnt`: Bitmap font data (with associated image texture file)

To learn more about Unity Text Assets in the manual pages, click on the following link:
`https://docs.unity3d.com/Manual/class-TextAsset.html`.

Many web-based systems use XML for asynchronous communications without requiring user interaction, leading to the term **AJAX**: **Asynchronous JavaScript XML**. Some modern web-based systems now use JSON for text-based communication. For this reason, this chapter puts special focus on these two text file formats.

The Big picture

Apart from plain text, there are two common text interchange file formats: XML and JSON. Each is discussed and illustrated through recipe examples in this chapter.

XML – the eXtensible markup language

XML is a meta-language, that is, a set of rules that allows markup languages to be created to encode specific kinds of data. Some examples of data-description language formats using the XML syntax include the following:

- `.txt`: Plain text file
- `.html`, `.htm`: HTML page markup (HyperText Markup Language)
- `.xml`: XML data (eXtensible Markup Language)
- `SVG`: Scalable Vector Graphics—an open standard method of describing graphics supported by the Worldwide Web consortium
- `SOAP`: Simple Object Access Protocol for the exchange of messages between computer programs and web services
- `X3D`: Xml 3D—an ISO standard for representing 3D objects—it is the successor to **VRML** (**Virtual Reality Modeling Language**)

JSON – the JavaScript object notation

JSON is sometimes referred to as the *fat-free alternative to XML*—offering similar data interchange strengths, but being smaller, and simpler, both by not offering extensibility and using just three characters for formatting:

- `property : value`: the colon character separates a property name form its value
- `{ }`: braces are for an object
- `[ ]`: square brackets are for an array of values/objects

You can read more about JSON versus XML at `https://www.json.org/xml.html`

In `Chapter 10`, *Working with External Resource Files*, several methods for loading external resource files were demonstrated, which work for image, audio, and text resources. In this chapter, several additional methods for loading text files in particular are presented.

In `Chapter 13`, *Shader Graphs and Video Players*, some recipes illustrate the use of JSON for a database-driven web leaderboard, and Unity game communication with that leaderboard.

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, since the text file data is serialized (mixed into) the general build resources, and so cannot be changed after the build has been created.

Getting ready

For this recipe, you'll need a text (`.txt`) file. In the `11_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. Create a new 2D project.
2. Create a **UI TextGameObject**, center it on screen with the **Rect Transform**, and set its horizontal and vertical overflow to overflow.
3. Import the text file you wish to use into your project (for example, `cities.txt`)
4. Create a C# `ReadPublicTextAsset` script class and attach an instance as a component to your **UI Text GameObject**:

```
using UnityEngine;
 using UnityEngine.UI;

 public class ReadPublicTextAsset : MonoBehaviour {
  public TextAsset dataTextFile;

  private void Start() {
      string textFromFile = dataTextFile.text;
      Text textOnScreen = GetComponent<Text>();
      textOnScreen.text = textFromFile;
  }
 }
```

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

How it works...

When the scene starts, the text content of the text file is read into variable `textFromFile`. A reference is found to the **UI Text** component, and the text property of that UI component is set to be the content of `textFromFile`. The user can then see the content of the text file displayed in the middle of the screen.

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. This recipe illustrates how to read a text file, while the next recipe illustrates how to write text data to files.

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

Getting ready

For this recipe, you'll need a text file; two have been provided in the `11_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-class `FileReadWriteManager`:

```
using System;
using System.IO;

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

```
            StreamWriter writer;
            writer = textFile.CreateText();

            writer.Write(stringData);
            writer.Close();
        }

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

            try {
                StreamReader textReader = File.OpenText(
pathAndName );

                dataAsString = textReader.ReadToEnd();
                textReader.Close();
            }
            catch (Exception e) {
                return "error:" + e.Message;
            }

            return dataAsString;
        }
 }
```

2. Create a C# `ReadWithStream` script class and attach an instance as a component to your **UI Text GameObject**:

```
using UnityEngine;
 using UnityEngine.UI;
 using System.IO;

 public class ReadWithStream : MonoBehaviour {
     private string fileName = "cities.txt";

     private string textFileContents = "(file not found yet)";
     private FileReadWriteManager fileReadWriteManager = new
FileReadWriteManager();

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

         textFileContents = fileReadWriteManager.ReadTextFile(
filePath );

         Text textOnScreen = GetComponent<Text>();
```

```
                textOnScreen.text = textFileContents;
            }
        }
```

3. Save the current scene and then add this to the list of scenes in the build.
4. Build and run your (Windows, Mac, or Linux) standalone executable.
5. 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 macOS "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.

6. When you run your built executable, you should see the text file content loaded and displayed in the middle of the application window.

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 the filePath string. This method reads the content 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 `11_02` folder, we have provided a text file containing the completed C# script class created in the previous recipe:

```
FileReadWriteManager.cs
```

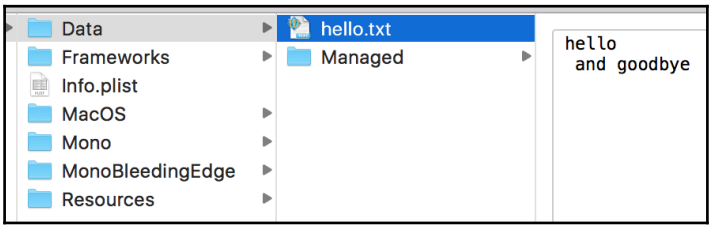
How to do it...

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

1. Create a new 2D project.
2. Import the C# `FileReadWriteManager.cs` script class into your project.
3. Add the following C# `SaveTextFile` script class to the **Main Camera**:

```
using UnityEngine;
 using System.IO;

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

     void Start () {
         string textData = "hello \n and goodbye";
         fileManager = new FileReadWriteManager();
         filePath = Path.Combine(Application.dataPath,
folderName);
         filePath = Path.Combine(filePath, fileName);
         fileManager.WriteTextFile( filePath, textData );
     }
 }
```

4. Save the current scene and then add this to the list of scenes in the build.
5. Build and run your (Windows, Mac, or Linux) standalone executable.
6. After running the built executable, 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). 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...

The following are 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 that have had their content changed.

Before you build your executable, you can specify a different file and folder name (for example, `Resources` instead of `Data`). Ensure the **Main Camera GameObject** is selected in the **Hierarchy**, and then change the values in those public variables in the **Inspector** component **Save text File (Script)**.

Loading and parsing external XML

It is useful to be able to parse (process the content of) text files and strings containing data in the XML format. C# offers a range of classes and methods to make such processing straightforward, which we'll explore in this recipe.

Getting ready

You'll find player name and score data in XML format in the `playerScoreData.xml` file in the `11_04` folder. The content of this file is 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. Create a C# `PlayerScoreDate` script class containing the following:

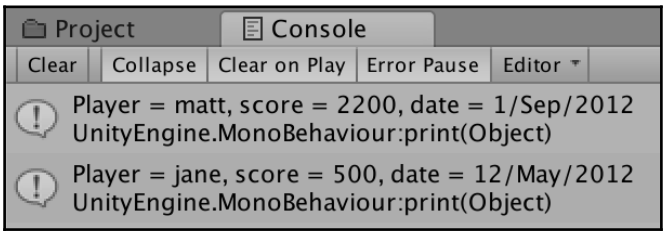
```
public class PlayerScoreDate
 {
     private string playerName;
     private int score;
     private string date;

     public void SetPlayerName(string playerName)
     { this.playerName = playerName; }

     public void SetScore(int score)
     { this.score = score; }

     public void SetDate(string date)
     { this.date = date; }

     override public string ToString()
     {
         return "Player = " + this.playerName + ",
         score = " + this.score + ", date = " + this.date;
     }
 }
```

2. Create a C# `ParseXML` script class and attach an instance as a component to the **Main Camera**:

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

 public class ParseXML : MonoBehaviour {
     public TextAsset scoreDataTextFile;
     private PlayerScoreDate[] playerScores = new
PlayerScoreDate[999];

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

         for (int i = 0; i < numberObjects; i++)
             print(playerScores[i]);
     }
```

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

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

            int i = 0;
            foreach(XmlNode node in myNodeList){
                playerScores[i] = NodeToPlayerScoreObject(node);
                i++;
            }

            return i;
        }

        private PlayerScoreDate NodeToPlayerScoreObject(XmlNode
node) {
            XmlNode playerNode = node.FirstChild;
            string playerName = playerNode.InnerXml;

            XmlNode scoreNode = playerNode.NextSibling;
            string scoreString = scoreNode.InnerXml;
            int score = Int32.Parse(scoreString);

            XmlNode dateNode = scoreNode.NextSibling;
            string date = NodeToDateString(dateNode);

            PlayerScoreDate playerObject = new PlayerScoreDate();
            playerObject.SetPlayerName(playerName);
            playerObject.SetScore(score);
            playerObject.SetDate(date);

            return playerObject;

        }

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

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

3. Run the scene, and the output of the `print()` statements should be visible in the **Console** window:



How it works...

The `PlayerScoreDate` script class simply contains the three pieces of data for player-dated scores:

- The player's name (string)
- The player's score (integer)
- The date the score was recorded (string—to keep this recipe short...)

Note the need to use the `System`, `System.Xml` and `System.IO` packages for the C# `ParseXML` script class.

The `text` property of the `TextAsset` variable `scoreDataTextFile` provides the content of the XML file as a string, which is passed to the `ParseScoreXML(...)` method.

This `ParseScoreXML(...)` method creates a new `XmlDocument` variable with the content 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 loops for each scoreRecord, passing the current node to method `NodeToPlayerScoreObject(...)`, and storing the returned object into the next slot in the `playerScores` array.

The `NodeToPlayerScoreObject(...)` method relies on the ordering of the XML elements to retrieve the player's name, score, and data strings. The score string is parsed into an integer, and the date node is converted to a date string using method `NodeToDateString(...)`. A new `PlayerScoreDate` object is created, and the name, score and date stored in it, and then that object is returned.

The `NodeToDateString(...)` method creates a date string as a slash-separated string by parsing the node containing the three date components.

There's more...

The following are 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 content, using the `XMLWriter` class.

How to do it...

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

1. Create a C# `CreateXMLString` script class to add an instance as a component to the **Main Camera**:

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

 public class CreateXMLString : MonoBehaviour {

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

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

```
            // start doc and root element
            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 in the **Console** panel when the scene is run, and should look as follows (some newline characters have been added to make the output more human-readable...):

```
<?xml version="1.0" encoding="utf-16"?>
   <playerScoreList>
   <player>
   <name>matt</name>
   <score>200</score>
   </player>
   <player>
   <name>jane</name>
   <score>150</score>
   </player>
   </playerScoreList>
```

How it works...

The `Start()` method calls `BuildXMLString()` and stores the returned string in the output variable. This output text is then printed to the Console debug panel.

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()` and `WriteEndElement()`. String content for an element is added using `WriteElementString()`.

There's more...

Here are some details that you won't want to miss.

Adding new lines 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 new line's 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 `List<T>` containing `PlayerScore` objects, and then calls the (static) method `ListToXML()`, passing in the list of objects.

The following should be a single block of code:

```
using UnityEngine;
using System.Collections.Generic;

public class CreateXMLFromArray : MonoBehaviour {
   private List<PlayerScore> playerScoreList;

   private void Start () {
       playerScoreList = new List<PlayerScore>();
```

```
        playerScoreList.Add (new PlayerScore("matt", 200) );
        playerScoreList.Add (new PlayerScore("jane", 150) );

        string output = PlayerScore.ListToXML( playerScoreList );
        print(output);
    }
}
```

All the hard work is now the responsibility of the `PlayerScore` class. This class has two private variables for the player's name and score and a constructor that accepts values for these properties. The public static method `ListToXML()` takes a `List` 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:

```
using System.Collections.Generic;
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;
   }

   static public string ListToXML(List<PlayerScore> playerList) {
       StringWriter str = new StringWriter();
       XmlTextWriter xml = new XmlTextWriter(str);
       xml.WriteStartDocument();
       xml.WriteStartElement("playerScoreList");
       foreach (PlayerScore playerScoreObject in playerList) {
          playerScoreObject.ObjectToElement( xml );
       }

       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();
        }
    }
```

Saving and loading XML text data automatically through serialization

Another way to work with XML data structures from game objects and properties is by serializing the content of an object automatically. This technique automatically generates XML for all the public properties of an object. This recipe uses the `XmlSerializer` class that can be found in the standard `System.Xml` C# package.

This recipe has been adapted from this 2013 (still works!) Unify Community Wiki article: `http://wiki.unity3d.com/index.php?title=Saving_and_Loading_Data:_XmlSerializer`

Getting ready

In the `11_06` folder, you'll find two XML data files, allowing you to test the reader with different XML text file data files.

How to do it...

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

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

```
using System.Xml.Serialization;

 [System.Serializable]
 public class PlayerScore
 {
     [XmlElement("Name")]
     public string name;

     [XmlElement("Score")]
     public int score;
```

```
        [XmlElement("Version")]
        public string version;
    }
```

2. Create a C# `PlayerScoreCollection` scrip class:

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

 [XmlRoot("PlayerScoreCollection")]
 public class PlayerScoreCollection
 {
     [XmlArray("PlayerScores"), XmlArrayItem("PlayerScore")]
     public PlayerScore[] playerScores;

     public void Save(string path) {
         var serializer = new
XmlSerializer(typeof(PlayerScoreCollection));
         using (var stream = new FileStream(path,
FileMode.Create)) {
             serializer.Serialize(stream, this);
         }
     }
 }
```

3. Create a C# `XmlWriter` script class and attach an instance as a component to the **Main Camera**:

```
using UnityEngine;
 using System.IO;

 public class XmlWriter : 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);

         PlayerScoreCollection psc =
CreatePlayScoreCollection();
         psc.Save(filePath);
         print("XML file should now have been created at: " +
filePath);
     }

     private PlayerScoreCollection CreatePlayScoreCollection()
```

```
{
            PlayerScoreCollection playerScoreCollection = new
PlayerScoreCollection();

            // make 2 slot array
            playerScoreCollection.playerScores = new
PlayerScore[2];

            playerScoreCollection.playerScores[0] = new
PlayerScore();
            playerScoreCollection.playerScores[0].name = "matt";
            playerScoreCollection.playerScores[0].score = 22;
            playerScoreCollection.playerScores[0].version =
"v0.5";

            playerScoreCollection.playerScores[1] = new
PlayerScore();
            playerScoreCollection.playerScores[1].name =
"joelle";
            playerScoreCollection.playerScores[1].score = 5;
            playerScoreCollection.playerScores[1].version =
"v0.9";

            return playerScoreCollection;
        }
    }
```

4. You can quickly test the scene in the Unity Editor if you create a `Data` folder 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 the `Data` folder.
5. Save the current scene and then add this to the list of scenes in the build.
6. Build and run your (Windows, Mac, or Linux) standalone executable.
7. 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.
8. The content of the `playerData.xml` file should be the XML player list data, that is:

```
<?xml version="1.0" encoding="us-ascii"?>
 <PlayerScoreCollection
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <PlayerScores>
     <PlayerScore>
```

```
      <Name>matt</Name>
      <Score>22</Score>
      <Version>v0.5</Version>
    </PlayerScore>
    <PlayerScore>
      <Name>joelle</Name>
      <Score>5</Score>
      <Version>v0.9</Version>
    </PlayerScore>
  </PlayerScores>
</PlayerScoreCollection>
```

How it works...

The `Start()` method of class `XmlWriter` defines the file path (`Data/playerData.xml`) and creates a new `PlayerScoreCollection` object **psc**, by invoking the method `CreatePlayScoreCollection()`. The Save(...) method of the `PlayerScoreCollection` object is then invoked, passing the file path.

The `CreatePlayScoreCollection()` method of class `XmlWriter` creates a new `PlayerScoreCollection`, and inserts into this an array of two `PlayerScore` objects with name/score/version values as follows:

```
matt, 22, v0.5
joelle, 5, v.09
```

The `Save(...)` method of class `PlayerScoreCollection` creates a new `XmlSerializer` for the class type, and uses a `FileStream` to tell C# to serialize the content of the `PlayerScoreCollection` object (that is, its array of `PlayerScore` objects) as text to that file.

There's more...

Here are some details you don't want to miss.

Defining the XML node names

We can use compiler statements to define the XML element name that will be used to encode each object property. For example, we have defined element **Name** (with a capital letter **N**) for the name property of each `PlayerScore`:

```
    [XmlElement("Name")]
    public string name;
```

If this hadn't been declared, then the **XML Serializer** would have defaulted to the lowercase property name `name` for these data elements.

Loading data objects from XML text

We can write static (class) methods for class `PlayerScoreCollection` that use the `XMLSerializer` to load XML data and create data objects from that loaded data.

Here is a method to load from a **filepath**:

```
    public static PlayerScoreCollection Load(string path) {
        var serializer = new
XmlSerializer(typeof(PlayerScoreCollection));
        using (var stream = new FileStream(path, FileMode.Open)) {
            return serializer.Deserialize(stream) as
PlayerScoreCollection;
        }
    }
```

Another method can be written to load from a text string:

```
    public static PlayerScoreCollection LoadFromString(string text) {
        var serializer = new
XmlSerializer(typeof(PlayerScoreCollection));
        return serializer.Deserialize(new StringReader(text)) as
PlayerScoreCollection;
    }
```

For example, you could create a public `TextAsset` variable, and then create a `PlayerScoreCollection` object by invoking this static method, then loop through and print out the loaded objects with code such as the following:

```
    public TextAsset dataAsXmlString;

    private void Start()
    {
        PlayerScoreCollection objectCollection =
        PlayerScoreCollection.LoadFromString(dataAsXmlString.text);

        foreach(PlayerScore playerScore in
objectCollection.playerScores){
            print("name = " + playerScore.name + ", score = " +
playerScore.score + ",
```

```
                version = " + playerScore.version);
        }
    }
```

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# `PlayerXMLWriter` script class:

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

 public class PlayerXMLWriter {
     private string filePath;
     private XmlDocument xmlDoc;
     private XmlElement elRoot;

     public PlayerXMLWriter(string filePath) {
         this.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 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);
        }

        public void SaveXMLFile() {
           xmlDoc.Save(filePath);
        }
    }
```

2. Create a C# `CreateXMLTextFile` script class and attach an instance as a component to the **Main Camera**:

```
    using UnityEngine;
     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 playerXMLWriter = new
PlayerXMLWriter(filePath);
            playerXMLWriter.AddXMLElement("matt", "55");
            playerXMLWriter.AddXMLElement("jane", "99");
            playerXMLWriter.AddXMLElement("fred", "101");
            playerXMLWriter.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 `Data` folder 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 the `Data` folder.
4. Save the current scene and then add this to the list of scenes in the build.
5. Build and run your (Windows, Mac, or Linux) standalone executable.
6. 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:
7. The content of the `playerData.xml` file should be the XML player list data:

How it works...

The `Start()` method creates `playerXMLWriter`, 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 constructor method of 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 method `AddXMLElement()` appends a new data node for the provided player name

and score. The method `SaveXMLFile()` saves the XML data structure as a text file for the stored file path string.

Creating JSON strings from individual objects and lists of objects

The `JsonUtility` class allows us to easily create JSON strings from individual objects, and also **Lists** of objects.

How to do it...

To create JSON strings from individual and **Lists** of objects perform the following steps:

1. Create a new C# script class named `PlayerScore`:

```
using UnityEngine;

 [System.Serializable]
 public class PlayerScore {
     public string name;
     public int score;

     public string ToJson() {
         bool prettyPrintJson = true;
         return JsonUtility.ToJson(this, prettyPrintJson);
     }
 }
```

2. Create a new C# script class named `PlayerScoreList`:

```
using UnityEngine;
 using System.Collections.Generic;

 [System.Serializable]
 public class PlayerScoreList {
     public List<PlayerScore> list = new
List<PlayerScore>();

     public string ToJson() {
         bool prettyPrint = true;
         return JsonUtility.ToJson(this, prettyPrint);
```

```
        }
    }
```

3. Create a C# `ToJson` script class and attach an instance as a component to the **Main Camera**:

```
using UnityEngine;

 public class ToJson : MonoBehaviour {
     private PlayerScore playerScore1 = new PlayerScore();
     private PlayerScore playerScore2 = new PlayerScore();
     private PlayerScoreList playerScoreList = new
PlayerScoreList();

     private void Awake() {
         playerScore1.name = "matt";
         playerScore1.score = 800;

         playerScore2.name = "joelle";
         playerScore2.score = 901;

         playerScoreList.list.Add(playerScore1);
         playerScoreList.list.Add(playerScore2);
     }

     void Start() {
         ObjectToJson();
         CollectionToJson();
     }

     public void ObjectToJson() {
         string objectAsString = playerScore1.ToJson();
         print("1: Object to JSON \n" + objectAsString);
     }

     public void CollectionToJson() {
         string objectListAsString =
playerScoreList.ToJson();
         print("2: List of objects to JSON \n" +
objectListAsString);
     }
 }
```

4. Run the scene. Two messages should be output to the **Console** panel:

```
1: Object to JSON
 {
     "name": "matt",
```

```
        "score": 800
    }
```

These should be followed by this:

```
2: List of objects to JSON
{
    "list": [
        {
            "name": "matt",
            "score": 800
        },
        {
            "name": "joelle",
            "score": 901
        }
    ]
}
```

How it works...

The `PlayerScore` scrip class declares two public properties: `name` and `score`. It also defines a public method `ToJson()` that returns a string containing the values of the properties encoded in JSON via the `JsonUtility.ToJson(...)` method.

The `PlayerScoreList` script class declares a single public property `list`, which is a C# `List<>` of `PlayerScore` obejcts. So, we can store zero, one, or any number of `PlayerScore` objects in our list. Also declared is a single public method `ToJson()` that returns a string containing the values of the content of the `list` property encoded in JSON via the `JsonUtility.ToJson(...)` method.

It also defines a public `ToJson()` method that returns a string containing the values of the properties encoded in JSON via the `JsonUtility.ToJson(...)` method.

The `Awake()` method creates two `PlayerScore` objects, with some demo data in objects `playerScore1` and `playerScore2`. It also creates an instance-object of class `PlayerScoreList`, and adds references to the two objects to this list.

The `Start()` method first invokes the `ObjectToJson()` method, then it invokes the `CollectionToJson()`.

Method `ObjectToJson()` invokes the `ToJson()` method of object `playerScore1`, which returns a string, which is then printed to the **Console** panel.

Method `CollectionToJson()` invokes the `ToJson()` method of object list `playerScoreList` which returns a string, which is then printed to the Console panel.

As we can see, both `PlayerScore` and `PlayerScoreList` classes define a `ToJson()` method, which makes use of the `JsonUtiltyToJson()` method. In both cases, their `ToJson()` method returns a string that is the JSON representation of the object's data.

The `PlayerScore` class's `ToJson()` method outputs a JSON object string in this form:

```
{ "name": "matt", "score": 800 }.
```

The `ToJson()` method of the `PlayerScoreList` class outputs a JSON array string in this form:

```
{
     "list": [
         { "name": "matt", "score": 800 },
         { "name": "joelle", "score": 901 }
     ]
 }
```

As we can see, the `JsonUtility` class's `ToString()` method is capable of serializing individual objects, and C# **Lists** of objects into a storable string.

Creating individual objects and Lists of objects from JSON strings

The `JsonUtility` class allows us to easily parse (process) JSON strings and extract individual objects, and also **Lists** of objects.

How to do it...

To create individual objects and **Lists** of objects from JSON strings, perform the following steps:

1. Create a new C# script class named `PlayerScore`:

```
using UnityEngine;
```

```
[System.Serializable]
public class PlayerScore {
    public string name;
    public int score;

    public static PlayerScore FromJSON(string jsonString)
{
        return
JsonUtility.FromJson<PlayerScore>(jsonString);
    }
}
```

2. Create a new C# script class named `PlayerScoreList`:

```
using UnityEngine;
 using System.Collections.Generic;

 [System.Serializable]
 public class PlayerScoreList {
    public List<PlayerScore> list = new
List<PlayerScore>();

    public static PlayerScoreList FromJSON(string
jsonString) {
        return
JsonUtility.FromJson<PlayerScoreList>(jsonString);
    }
}
```

3. Create a C# `ToJson` script class and attach an instance as a component to the **Main Camera**:

```
using UnityEngine;

 public class FromJson : MonoBehaviour {
    private void Start() {
        JsonToObject();
        JsonToList();
    }

    public void JsonToObject() {
        string playerScoreAsString = "{
\"name\":\"matt\", \"score\":201}";
        PlayerScore playerScore =
PlayerScore.FromJSON(playerScoreAsString);

        print(playerScore.name + ", " +
playerScore.score);
```

```
        }

        public void JsonToList() {
            string playerScorelistAsString = "";

            playerScorelistAsString += "{";
            playerScorelistAsString += "\"list\": [";
            playerScorelistAsString += "        {";
            playerScorelistAsString += "            \"name\":
\"matt\",";
            playerScorelistAsString += "
\"score\": 800";
            playerScorelistAsString += "        },";
            playerScorelistAsString += "        {";
            playerScorelistAsString += "            \"name\":
\"joelle\",";
            playerScorelistAsString += "
\"score\": 901";
            playerScorelistAsString += "        }";
            playerScorelistAsString += "    ]";
            playerScorelistAsString += "}";

            PlayerScoreList playerScoreList =
PlayerScoreList.FromJSON(playerScorelistAsString);


            foreach (var playerScore in playerScoreList.list)
{
                print("from list :: " + playerScore.name + ",
" + playerScore.score);
            }
        }
    }
```

4. Run the scene. Three messages should be output to the **Console** panel:

```
matt, 201
```

They should be followed by this:

```
from list :: matt, 800
from list :: joelle, 901
```

How it works...

The `Start()` method first invokes method `JsonToObject()`, and then it invokes

method `JsonToList()`.

The method `JsonToObject()` declares a string that defines one `PlayerScore` object: `{name:matt, score:201}`. The string is created using escaped double-quote characters for property names and text data (quotes aren't needed for the numeric data). The JSON string contains the following:

```
{
   "name":"matt",
   "score":201
}
```

Using escaped double-quote characters for property names and text data (quotes aren't needed for the numeric data). The **static** method `FromJSON(...)` of the class `PlayerScore` is then invoked with this JSON data string as its argument. The method returns a `PlayerScore` object, whose value is then printed to the **Console** panel.

The **static** method `FromJSON(...)` of class `PlayerScore` invokes the `FromJson()` method of the `JsonUtility` class, and provides the class type `PlayerScore`.

The method `JsonToList()` declares a string that defines a list of two `PlayerScore` objects: `{name:matt, score:800}` and `{name:joelle, score:901}`. The JSON string contains the following:

```
{
   "list": [
         {
               "name": "matt",
               "score": 800
         },
         {
               "name": "joelle",
               "score": 901
         }
   ]
}
```

The **static** method `FromJSON(...)` of the class `PlayerScoreList` is then invoked with this JSON data string as its argument. The method returns a `PlayerScoreList` object. A `foreach` loop extracts each `PlayerScore` object from the **list** property of the `PlayerScoreList` object. Each object's `name` and `score` values are printed out to the **Console** panel, prefixed with the text from the list.

The **static** method `FromJSON(...)` of class `PlayerScoreList` invokes the

`FromJson()` method of the `JsonUtility` class, and provides the class type `PlayerScoreList`.

As we can see, the `JsonUtility` class's `FromJson()` method is capable of deserializing data into individual objects, and C# **Lists** of objects inside repository objects, such as `PlayerScoreList`.