Programming in C#

E05 Implement data access

Yu Guan | Microsoft MVP
2018

AGENDA

- Multithreading
- Perform I/O operations
- Consume data
- LINQ
- Serialize and deserialize

Multithreading

- Understand threads
- Use the Task Parallel Library
- Use the Parallel class
- Use the new async and await keywords
- Use concurrent collections
- Synchronize resources
- Cancel long-running tasks

Understanding threads

- Using the Thread class
- The Thread class can be found in the System.Threading namespace. This class enables you to create new treads, manage their priority, and get their status.

```
public static class Program
{
    public static void ThreadMethod()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(“ThreadProc: {0}”, i);
            Thread.Sleep(1000);
        }
    }
   public static void Main()
    {
        Thread t = new Thread(new ThreadStart(ThreadMethod));
        t.IsBackground = true;
        t.Start();
    }
}
```

Understanding threads

- Thread pools
- A thread pool is created to reuse those threads, similar to the way a database connection pooling works. Instead of letting a thread die, you send it back to the pool where it can be reused whenever a request comes in.

```
public static void Main()
{
    ThreadPool.QueueUserWorkItem((s) =>
    {
        Console.WriteLine("Working on a thread from threadpool");
    });
    Console.ReadLine();
}
```

Using Tasks

- The Task can tell you if the work is completed and if the operation returns a result, the Task gives you the result.

```
public static void Main()
{
       Task t = Task.Run(() =>
       {
           for (int x = 0; x < 100; x++)
           {
               Console.Write(‘*’);
            }
       });
}
```

Using the Parallel class

- Parallelism involves taking a certain task and splitting it into a set of related tasks that can be executed concurrently. This also means that you shouldn’t go through your code to replace all your loops with parallel loops. You should use the Parallel class only when your code doesn’t have to be executed sequentially.

```
Parallel.For(0, 10, i =>
{
    Thread.Sleep(1000);
});

var numbers = Enumerable.Range(0, 10);
Parallel.ForEach(numbers, i =>
{
    Thread.Sleep(1000);
});
```

Using async and await

- You use the async keyword to mark a method for asynchronous operations. This way, you signal to the compiler that something asynchronous is going to happen. The compiler responds to this by transforming your code into a state machine.

```
public static void Main()
{
    string result = DownloadContent().Result;
    Console.WriteLine(result);
}

public static async Task<string> DownloadContent()
{
    using(HttpClient client = new HttpClient())
    {
        string result = await client.GetStringAsync(“http://www.microsoft.com”);
        return result;
    }
}
```

Using concurrent collections

- When working in a multithreaded environment, you need to make sure that you are not manipulating shared data at the same time without synchronizing access.
- The .NET Framework offers some collection classes that are created specifically for use in concurrent environments, which is what you have when you're using multithreading. These collections are thread-safe, which means that they internally use synchronization to make sure that they can be accessed by multiple threads at the same time.
  - BlockingCollection<T>
  - ConcurrentBag<T>
  - ConcurrentDictionary<TKey,T>
  - ConcurrentQueue<T>
  - ConcurrentStack<T>

Synchronizing resources

- lock
- volatile

  ```
  int i = 5;
  Console.WriteLine(i);
  ```

  The compiler may optimize this to just print 5, like this:

  ```
  Console.WriteLine(5);
  ```

  However, if there is another thread which can change i, this is the wrong behaviour. If another thread changes i to be 6, the optimized version will still print 5.

  The volatile keyword prevents such optimization and caching, and thus is useful when a variable can be changed by another thread.

- Interlocked

Canceling tasks

- When working with multithreaded code such as the TPL, the Parallel class, or PLINQ, you often have long-running tasks. The .NET Framework offers a special class that can help you in canceling these tasks: CancellationToken.
- You pass a CancellationToken to a Task, which then periodically monitors the token to see whether cancellation is requested.

Perform I/O operations

- Work with files
- Work with streams
- Read and write from the network
- Implement asynchronous I/O operations

Work with files

- Files are an important aspect of almost every application. That's why the .NET Framework offers an infrastructure that can help you with all your file-related work. All the necessary types can be found in the System.IO namespace.
- DriveInfo
  - `DriveInfo[] drivesInfo = DriveInfo.GetDrives();`
- Directory
  - `var directory = Directory.CreateDirectory(@”C:\Temp”);`
- File
  - `File.Delete(path);`
- Path
  - `string fullPath = Path.Combine(folder, fileName);`

Working with streams

- When working with files in the .NET Framework, it's important to know about the Stream class, which is a base class that is used in the .NET Framework for I/O operations. It's an abstraction of a sequence of bytes. For example, a file is, in essence, a sequence of bytes stored on your hard drive or a DVD, but a network socket also works with sequences of bytes just as an inter-process communication pipe. The Stream class provides a generic interface for all these types of input/output.
- A stream has three fundamental operations:
  - Reading
  - Writing
  - Seeking

Working with streams

```
string path = @"c:\temp\test.dat";
using (FileStream fileStream = File.Create(path))
{
    string myValue = "MyValue";
    byte[] data = Encoding.UTF8.GetBytes(myValue);
    fileStream.Write(data, 0, data.Length);
}
```

Working with streams

```
string folder = @"c:\temp";
string uncompressedFilePath = Path.Combine(folder,"uncompressed.dat");
string compressedFilePath = Path.Combine(folder,"compressed.gz");
byte[] dataToCompress = Enumerable.Repeat((byte)'a', 1024 * 1024).ToArray();
using (FileStream uncompressedFileStream = File.Create(uncompressedFilePath))
{
    uncompressedFileStream.Write(dataToCompress, 0, dataToCompress.Length);
}
using (FileStream compressedFileStream = File.Create(compressedFilePath))
{
    using (GZipStream compressionStream = new GZipStream(
    compressedFileStream, CompressionMode.Compress))
    {
        compressionStream.Write(dataToCompress, 0, dataToCompress.Length);
    }
}
FileInfo uncompressedFile = new FileInfo(uncompressedFilePath);
FileInfo compressedFile = new FileInfo(compressedFilePath);
Console.WriteLine(uncompressedFile.Length); // Displays 1048576
Console.WriteLine(compressedFile.Length); // Displays 1052
```

Communicating over the network

- WebRequest and WebResponse

```
WebRequest request = WebRequest.Create(“http://www.microsoft.com”);
WebResponse response = request.GetResponse();
StreamReader responseStream = new StreamReader(response.GetResponseStream());
string responseText = responseStream.ReadToEnd();
Console.WriteLine(responseText); // Displays the HTML of the website
response.Close();
```

Implementing asynchronous I/O operations

```
public async Task CreateAndWriteAsyncToFile()
{
    using (FileStream stream = new FileStream(“test.dat”, FileMode.Create,
        FileAccess.Write, FileShare.None, 4096, true))
    {
        byte[] data = new byte[100000];
        new Random().NextBytes(data);
        await stream.WriteAsync(data, 0, data.Length);
    }
}
```

Implementing asynchronous I/O operations

```
public async Task ReadAsyncHttpRequest()
{
    HttpClient client = new HttpClient();
    string result = await client.GetStringAsync(“http://www.microsoft.com”);
}


public async Task ExecuteMultipleRequests()
{
    HttpClient client = new HttpClient();
    string microsoft= await client.GetStringAsync(“http://www.microsoft.com”);
    string msdn = await client.GetStringAsync(“http://msdn.microsoft.com”);
    string blogs = await client.GetStringAsync(“http://blogs.msdn.com/”);
}
```

Implementing asynchronous I/O operations

```
public async Task ExecuteMultipleRequestsInParallel()
{
    HttpClient client = new HttpClient();
    Task microsoft = client.GetStringAsync(“http://www.microsoft.com”);
    Task msdn =  client.GetStringAsync(“http://msdn.microsoft.com”);
    Task blogs = client.GetStringAsync(“http://blogs.msdn.com/”);
    await Task.WhenAll(microsoft, msdn, blogs);
}
```

Consume data

- Consume XML
- Consume JSON
- Access web services

Consuming XML

- XmlReader
- XmlWriter
- XmlDocument
- XPath

Consuming XML

```
string xml = @”<?xml version=””1.0”” encoding=””utf-8”” ?>
                <people>
                  <person firstname=””john”” lastname=””doe””>
                    <contactdetails>
                      <emailaddress>john@unknown.com</emailaddress>
                    </contactdetails>
                  </person>
                  <person firstname=””jane”” lastname=””doe””>
                    <contactdetails>
                      <emailaddress>jane@unknown.com</emailaddress>
                      <phonenumber>001122334455</phonenumber>
                    </contactdetails>
                  </person>
                </people>”;
```

XmlReader

```
using (StringReader stringReader = new StringReader(xml))
{
    using (XmlReader xmlReader = XmlReader.Create(stringReader,
        new XmlReaderSettings() { IgnoreWhitespace = true }))
    {
        xmlReader.MoveToContent();
        xmlReader.ReadStartElement(“People”);
        string firstName = xmlReader.GetAttribute(“firstName”);
        string lastName = xmlReader.GetAttribute(“lastName”);
        Console.WriteLine(“Person: {0} {1}”, firstName, lastName);
        xmlReader.ReadStartElement(“Person”);
        Console.WriteLine(“ContactDetails”);
        xmlReader.ReadStartElement(“ContactDetails”);
        string emailAddress = xmlReader.ReadString();
        Console.WriteLine(“Email address: {0}”, emailAddress);
    }
}
```

XmlWriter

```
StringWriter stream = new StringWriter();
using (XmlWriter writer = XmlWriter.Create(
    stream,
    new XmlWriterSettings() { Indent = true }))
{
    writer.WriteStartDocument();
    writer.WriteStartElement(“People”);
    writer.WriteStartElement(“Person”);
    writer.WriteAttributeString(“firstName”, “John”);
    writer.WriteAttributeString(“lastName”, “Doe”);
    writer.WriteStartElement(“ContactDetails”);
    writer.WriteElementString(“EmailAddress”, “john@unknown.com”);
    writer.WriteEndElement();
    writer.WriteEndElement();
    writer.Flush();
}
Console.WriteLine(stream.ToString());
```

XmlDocument

```
XmlDocument doc = new XmlDocument();
doc.LoadXml(xml);
XmlNodeList nodes = doc.GetElementsByTagName(“Person”);
// Output the names of the people in the document
foreach (XmlNode node in nodes)
{
    string firstName = node.Attributes[“firstName”].Value;
    string lastName = node.Attributes[“lastName”].Value;
    Console.WriteLine(“Name: {0} {1}”, firstName, lastName);
}
// Start creating a new node
XmlNode newNode = doc.CreateNode(XmlNodeType.Element, “Person”, “”);
XmlAttribute firstNameAttribute = doc.CreateAttribute(“firstName”);
firstNameAttribute.Value = “Foo”;
XmlAttribute lastNameAttribute = doc.CreateAttribute(“lastName”);
lastNameAttribute.Value = “Bar”;
newNode.Attributes.Append(firstNameAttribute);
newNode.Attributes.Append(lastNameAttribute);
doc.DocumentElement.AppendChild(newNode);
Console.WriteLine(“Modified xml...”);
doc.Save(Console.Out);
```

XPath

```
XmlDocument doc = new XmlDocument();
doc.LoadXml(xml);
XPathNavigator nav = doc.CreateNavigator();
string query = “//People/Person[@firstName=’Jane’]”;
XPathNodeIterator iterator = nav.Select(query);
Console.WriteLine(iterator.Count); // Displays 1
while(iterator.MoveNext())
{
    string firstName = iterator.Current.GetAttribute(“firstName”,””);
    string lastName = iterator.Current.GetAttribute(“lastName”,””);
    Console.WriteLine(“Name: {0} {1}”, firstName, lastName);
}
```

Consuming JSON

```
{
  “People”: {
    “Person”: [
      {
        “firstName”: “John”,
        “lastName”: “Doe”,
        “ContactDetails”: { “EmailAddress”: “john@unknown.com” }
      },
      {
        “firstName”: “Jane”,
        “lastName”: “Doe”,
        “ContactDetails”: {
          “EmailAddress”: “jane@unknown.com”,
          “PhoneNumber”: “001122334455”
        }
      }
    ]
  }
}
```

LINQ (Language Integrated Query)

- Use the standard LINQ operators
- Use LINQ in the most optimal way
- Use LINQ to XML

Standard LINQ query operators

- The standard query operators are: All, Any, Average, Cast, Count, Distinct, GroupBy, Join, Max, Min, OrderBy, OrderByDescending, Select, SelectMany, Skip, SkipWhile, Sum, Take, TakeWhile, ThenBy, ThenByDescending, and Where.

Using LINQ to XML

```
XDocument doc = XDocument.Parse(xml);
IEnumerable<string> personNames = from p in doc.Descendants(“Person”)
                                  select (string)p.Attribute(“firstName”)
                                  + “ “ + (string)p.Attribute(“lastName”);
foreach (string s in personNames)
{
    Console.WriteLine(s);
}
```

Serialize and deserialize

- Using binary serialization
- Using XmlSerializer
- Using JSON serializer

Using binary serialization

- [Serializable]
- BinaryFormatter

Using binary serialization

```
BinaryFormatter binaryFormatter = new BinaryFormatter();

using (MemoryStream memoryStream = new MemoryStream())
{
    binaryFormatter.Serialize(memoryStream, source);
    memoryStream.Position = 0;
    return memoryStream.ToArray();
}
```

```
BinaryFormatter binaryFormatter = new BinaryFormatter();

using (MemoryStream memoryStream = new MemoryStream(source))
{
    memoryStream.Position = 0;
    return binaryFormatter.Deserialize(memoryStream);
}
```

Using XmlSerializer

- XmlIgnore
- XmlAttribute
- XmlElement
- XmlArray
- XmlArrayItem
- XmlSerializer

Using XmlSerializer

```
XmlSerializer xmlSerializer =new XmlSerializer(source.GetType());

using (MemoryStream memoryStream = new MemoryStream())
using (StreamReader streamReader = new StreamReader(memoryStream))
using (XmlWriter xmlWriter = XmlWriter.Create(memoryStream))
{
    xmlSerializer.Serialize(xmlWriter, source);
    xmlWriter.Flush();
    memoryStream.Position = 0;

    return streamReader.ReadToEnd();
}
```

Using JSON serializer

- DataContractJsonSerializer
- [DataContract]
- [DataMember]

- Json.Net

Using JSON serializer

```
DataContractJsonSerializer dataContractJsonSerializer = new DataContractJsonSerializer(source.GetType());

using (MemoryStream memoryStream = new MemoryStream())
{
    dataContractJsonSerializer.WriteObject(memoryStream, source);
    memoryStream.Position = 0;
    string result = (encoding ?? Encoding.UTF8).GetString(memoryStream.ToArray());
    return result;
}
```

Recap

- Exam Ref 70-483
  - CHAPTER 4 Implement data access
    - Objective 4.1: Perform I/O operations
    - Objective 4.2: Consume data
    - Objective 4.3: Query and manipulate data and objects by using LINQ
    - Objective 4.4. Serialize and deserialize data
    - Objective 4.5. Store data in and retrieve data from collections

THANK YOU!

Q&A