CSc 346
Object Oriented Programming

Ken Gamradt

Spring 2024

Chapter 9 – Part 2

Encoding and decoding text

- Text characters can be represented in different ways
  - E.g., the alphabet can be encoded using Morse code into a series of dots and dashes for transmission over a telegraph line
- In a similar way, text inside a computer is stored as bits (ones and zeros) representing a code point within a code space
  - Most code points represent a single character, but they can also have other meanings like formatting
  - E.g., **ASCII** has a code space with 128 code points
  - .NET uses a standard called **Unicode** to encode text internally
  - Unicode has more than one million code points
- Sometimes, you will need to move text outside .NET for use by systems that do not use Unicode or use a variation of Unicode
  - It is important to learn how to convert between encodings

Encoding and decoding text

Encoding	Description
ASCII	This encodes a limited range of characters using the lower seven bits of a byte.
UTF-8	This represents each Unicode code point as a sequence of one to four bytes.
UTF-7	This is designed to be more efficient over 7-bit channels than UTF-8, but it has security and robustness issues, so UTF-8 is recommended over UTF-7.
UTF-16	This represents each Unicode code point as a sequence of one or two 16-bit integers.
UTF-32	This represents each Unicode code point as a 32-bit integer and is, therefore, a fixed-length encoding unlike the other Unicode encodings, which are all variable-length encodings.
ANSI/ISO encodings	This provides support for a variety of code pages that are used to support a specific language or group of languages.

Encoding and decoding text

- **Good Practice**:
  - In most cases today, **UTF-8** is a good default, which is why it is literally the default encoding
    - **Encoding.Default**
  - You should avoid using Encoding.UTF-7 because it is unsecure
  - Due to this, the C# compiler will warn you when you try to use UTF-7
  - You might need to generate text using that encoding for compatibility with another system, so it needs to remain an option in .NET

Encoding strings as byte arrays

```
using System.Text; // To use Encoding
WriteLine("Encodings");
WriteLine("[1] ASCII");
WriteLine("[2] UTF-7");
WriteLine("[3] UTF-8");
WriteLine("[4] UTF-16 (Unicode)");
WriteLine("[5] UTF-32");
WriteLine("[6] Latin1");
WriteLine("[any other key] Default encoding");
WriteLine();
Write("Press a number to choose an encoding.");
ConsoleKey number = ReadKey(intercept: true).Key;
WriteLine();
WriteLine();
Encoding encoder = number switch {
    ConsoleKey.D1 or ConsoleKey.NumPad1 => Encoding.ASCII,
    ConsoleKey.D2 or ConsoleKey.NumPad2 => Encoding.UTF7,
    ConsoleKey.D3 or ConsoleKey.NumPad3 => Encoding.UTF8,
    ConsoleKey.D4 or ConsoleKey.NumPad4 => Encoding.Unicode,
    ConsoleKey.D5 or ConsoleKey.NumPad5 => Encoding.UTF32,
    ConsoleKey.D6 or ConsoleKey.NumPad6 => Encoding.Latin1,
    _                                   => Encoding.Default
};
```

Encoding strings as byte arrays

```
// Define a string to encode
string message = "Café £4.39";
WriteLine($"Text to encode: {message}  Characters: {message.Length}.");
// Encode the string into a byte array
byte[] encoded = encoder.GetBytes(message);
// Check how many bytes the encoding needed
WriteLine("{0} used {1:N0} bytes.", encoder.GetType().Name, encoded.Length);
WriteLine();
// Enumerate each byte
WriteLine("BYTE | HEX | CHAR");
foreach (byte b in encoded) {
  WriteLine($"{b,4} | {b,3:X} | {(char)b,4}");
}
// Decode the byte array back into a string and display it
string decoded = encoder.GetString(encoded);
WriteLine($"Decoded: {decoded}");
```

Encoding strings as byte arrays

```
Text to encode: Café £4.39  Characters: 10
ASCIIEncodingSealed used 10 bytes.

BYTE | HEX | CHAR
  67 |  43 |    C
  97 |  61 |    a
 102 |  66 |    f
  63 |  3F |    ?
  32 |  20 |
  63 |  3F |    ?
  52 |  34 |    4
  46 |  2E |    .
  51 |  33 |    3
  57 |  39 |    9
Decoded: Caf? ?4.39
```

```
Text to encode: Café £4.39  Characters: 10
UTF8EncodingSealed used 12 bytes.

BYTE | HEX | CHAR
  67 |  43 |    C
  97 |  61 |    a
 102 |  66 |    f
 195 |  C3 |    Ã
 169 |  A9 |    ©
  32 |  20 |
 194 |  C2 |    Â
 163 |  A3 |    £
  52 |  34 |    4
  46 |  2E |    .
  51 |  33 |    3
  57 |  39 |    9
Decoded: Café £4.39
```

Encoding and decoding text in files

- When using stream helper classes, such as **StreamReader** and **StreamWriter**, you can specify the encoding you want to use
  - As you write to the helper, the text will automatically be encoded
  - As you read from the helper, the bytes will be automatically decoded
- To specify an encoding, pass the encoding as a second parameter to the helper type's constructor

```
StreamReader reader = new(stream, Encoding.UTF8);
StreamWriter writer = new(stream, Encoding.UTF8);
```

Serializing object graphs

- An **object graph** is a structure of multiple objects that are related to each other, either through a direct reference or indirectly through a chain of references
- **Serialization** is the process of converting a live object into a sequence of bytes using a specified format
- **Deserialization** is the reverse process
- You would do this to save the current state of a live object so that you can recreate it in the future
  - E.g., saving the current state of a game so that you can continue at the same place tomorrow
- Serialized objects are usually stored in a file or database
- There are dozens of formats you can specify, the two most common ones are
  - **eXtensible Markup Language** (**XML**)
  - **JavaScript Object Notation** (**JSON**)

Serializing object graphs

- **Good Practice**:
  - JSON is more compact and is best for web and mobile applications
  - XML is more verbose but is better supported in more legacy systems
  - Use JSON to minimize the size of serialized object graphs
  - JSON is also a good choice when sending object graphs to web applications and mobile applications because JSON is the native serialization format for JavaScript, and mobile apps often make calls over limited bandwidth, so the number of bytes is important

Serializing as XML

- Let's start by looking at XML, probably the world's most used serialization format (for now)
- To show a typical example, we will define a custom class to store information about a person and then create an object graph using a list of Person instances with nesting

```
3 references
public class Person {
    0 references
    public Person() {}  // without - System.InvalidOperationException
    0 references
    public Person(decimal initialSalary) { Salary = initialSalary; }
    0 references
    public string? FirstName { get; set; }
    0 references
    public string? LastName { get; set; }
    0 references
    public DateTime DateOfBirth { get; set; }
    0 references
    public HashSet<Person>? Children { get; set; }
    1 reference
    protected decimal Salary { get; set; }
}
```

Serializing as XML

```
using System.Xml.Serialization; // To use XmlSerializer
List<Person> people = new() {
    new(initialSalary: 30_000M) {
        FirstName = "Alice", LastName = "Smith", DateOfBirth = new(year: 1974, month: 3, day: 14)
    },
    new(initialSalary: 40_000M) {
        FirstName = "Bob", LastName = "Jones", DateOfBirth = new(year: 1969, month: 11, day: 23)
    },
    new(initialSalary: 20_000M) {
        FirstName = "Charlie", LastName = "Cox", DateOfBirth = new(year: 1984, month: 5, day: 4),
        Children = new() {
            new(initialSalary: 0M) {
                FirstName = "Sally", LastName = "Cox", DateOfBirth = new(year: 2012, month: 7, day: 12)
            }
        }
    }
};
SectionTitle("Serializing as XML");
```

Serializing as XML

```
// Create serializer to format a "List of Person" as XML
XmlSerializer xs = new(type: people.GetType());
// Create a file to write to
string path = Combine(CurrentDirectory, "people.xml");
using (FileStream stream = File.Create(path)) {
    // Serialize the object graph to the stream
    xs.Serialize(stream, people);
} // Closes the stream
OutputFileInfo(path);
```

Serializing as XML

```
**** File Info ****
File: people.xml
Path: C:\cs12dotnet8\Chapter09\WorkingWithSerialization\bin\Debug\net8.0
Size: 793 bytes.
/------------------
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Person>
    <FirstName>Alice</FirstName>
    <LastName>Smith</LastName>
    <DateOfBirth>1974-03-14T00:00:00</DateOfBirth>
  </Person>
  <Person>
    <FirstName>Bob</FirstName>
    <LastName>Jones</LastName>
    <DateOfBirth>1969-11-23T00:00:00</DateOfBirth>
  </Person>
  <Person>
    <FirstName>Charlie</FirstName>
    <LastName>Cox</LastName>
    <DateOfBirth>1984-05-04T00:00:00</DateOfBirth>
    <Children>
      <Person>
        <FirstName>Sally</FirstName>
        <LastName>Cox</LastName>
        <DateOfBirth>2012-07-12T00:00:00</DateOfBirth>
      </Person>
    </Children>
  </Person>
</ArrayOfPerson>
------------------/
```

Generating compact XML

```
[XmlAttribute("fname")]
0 references
public string? FirstName { get; set; }
[XmlAttribute("lname")]
0 references
public string? LastName { get; set; }
[XmlAttribute("dob")]
0 references
public DateTime DateOfBirth { get; set; }
```

Generating compact XML

```
**** File Info ****
File: people.xml
Path: C:\cs12dotnet8\Chapter09\WorkingWithSerialization\bin\Debug\net8.0
Size: 488 bytes.
/------------------
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Person fname="Alice" lname="Smith" dob="1974-03-14T00:00:00" />
  <Person fname="Bob" lname="Jones" dob="1969-11-23T00:00:00" />
  <Person fname="Charlie" lname="Cox" dob="1984-05-04T00:00:00">
    <Children>
      <Person fname="Sally" lname="Cox" dob="2012-07-12T00:00:00" />
    </Children>
  </Person>
</ArrayOfPerson>
------------------/
```

Deserializing XML files

```
SectionTitle("Deserializing XML files");
using (FileStream xmlLoad = File.Open(path, FileMode.Open)) {
    // Deserialize and cast the object graph into a "List of Person"
    List<Person>? loadedPeople = xs.Deserialize(xmlLoad) as List<Person>;
    if (loadedPeople is not null) {
        foreach (Person p in loadedPeople) {
            WriteLine("{0} has {1} children.", p.LastName, p.Children?.Count ?? 0);
        }
    }
}
```

```
Smith has 0 children.
Jones has 0 children.
Cox has 1 children.
```

High-performance JSON processing

- .NET Core 3.0 introduced a new namespace for working with JSON, System.Text.Json, which is optimized for performance by leveraging APIs like Span<T>
- Also, older libraries like Json.NET are implemented by reading UTF-16
- It would be more performant to read and write JSON documents using UTF-8 because most network protocols, including HTTP, use UTF-8 and you can avoid transcoding UTF-8 to and from Json.NET's Unicode string values
- With the new API, Microsoft achieved between 1.3x and 5x improvement, depending on the scenario
- The original author of Json.NET, James Newton-King, joined Microsoft and has been working with them to develop their new JSON types

High-performance JSON processing

```
using FastJson = System.Text.Json.JsonSerializer;
SectionTitle("Deserializing JSON files");
await using (FileStream jsonLoad = File.Open(jsonPath, FileMode.Open)) {
    // Deserialize object graph into a "List of Person"
    List<Person>? loadedPeople =
        await FastJson.DeserializeAsync(
            utf8Json: jsonLoad,
            returnType: typeof(List<Person>)) as List<Person>;
    if (loadedPeople is not null) {
        foreach (Person p in loadedPeople) {
            WriteLine("{0} has {1} children.", p.LastName, p.Children?.Count ?? 0);
        }
    }
}
```

Controlling JSON processing

- There are many options for taking control of how JSON is processed:
  - Including and excluding fields
  - Setting a casing policy
  - Selecting a case-sensitivity policy
  - Choosing between compact and prettified whitespace

Controlling JSON processing

```
using System.Text.Json.Serialization; // To use [JsonInclude]
1 reference
public class Book {
    // Constructor to set non-nullable property
    0 references
    public Book(string title) {
        Title = title;
    }
    // Properties
    1 reference
    public string Title { get; set; }
    0 references
    public string? Author { get; set; }
    // Fields
    [JsonInclude] // Include this field
    0 references
    public DateTime PublishDate;
    [JsonInclude] // Include this field
    0 references
    public DateTimeOffset Created;
    0 references
    public ushort Pages;
}
```

Controlling JSON processing

```
using System.Text.Json; // To use JsonSerializer.
Book csharpBook = new(title: "C# 12 and .NET 8 - Modern Cross-Platform Development Fundamentals") {
  Author = "Mark J Price",
  PublishDate = new(year: 2023, month: 11, day: 14),
  Pages = 823,
  Created = DateTimeOffset.UtcNow,
};
JsonSerializerOptions options = new() {
    IncludeFields = true, // Includes all fields
    PropertyNameCaseInsensitive = true,
    WriteIndented = true,
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
};
string path = Combine(CurrentDirectory, "book.json");
using (Stream fileStream = File.Create(path)) {
    JsonSerializer.Serialize(utf8Json: fileStream, value: csharpBook, options);
}
WriteLine("**** File Info ****");
WriteLine($"File: {GetFileName(path)}");
WriteLine($"Path: {GetDirectoryName(path)}");
WriteLine($"Size: {new FileInfo(path).Length:N0} bytes.");
WriteLine("/------------------");
WriteLine(File.ReadAllText(path));
WriteLine("------------------/");
```

Controlling JSON processing

```
**** File Info ****
File: book.json
Path: C:\cs12dotnet8\Chapter09\ControllingJson\bin\Debug\net8.0
Size: 221 bytes.
/------------------
{
  "title": "C# 12 and .NET 8 - Modern Cross-Platform Development
Fundamentals",
  "author": "Mark J Price",
  "publishDate": "2023-11-14T00:00:00",
  "created": "2023-07-13T14:29:07.119631+00:00",
  "pages": 823
}
------------------/
```

Controlling JSON processing

- Note the following:
  - The JSON file is 221 bytes
  - The member names use camelCasing
    - E.g., publishDate
    - This is best for sub-sequent processing in a browser with JavaScript
  - All fields are included due to the options set, including pages
  - JSON is prettified for easier human legibility
  - DateTime and DateTimeOffset values are stored as a single standard string format

Controlling JSON processing

- Now, comment out the casing policy, write indented, and include fields of the JsonSerializerOptions

```
**** File Info ****
File: book.json
Path: C:\cs12dotnet8\Chapter09\ControllingJson\bin\Debug\net8.0
Size: 184 bytes.
/-----------------
{"Title":"C# 12 and .NET 8 - Modern Cross-Platform Development
Fundamentals","Author":"Mark J Price","PublishDate":"2023-11-
14T00:00:00","Created":"2023-07-13T14:30:29.2205861+00:00"}
-----------------/
```

- Note the following:
  - The JSON file has about a 20% reduction
  - The member names use normal casing
    - E.g., PublishDate
  - The Pages field is missing
  - The other fields are included due to the [JsonInclude] attribute on the PublishDate and Created fields