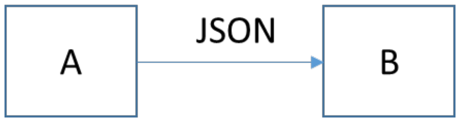
Module 3 – Working with JSON

Overview

In modules 1 and 2 we looked at how to create and consume XML. JSON (JavaScript Object Notation) is another human and machine readable and writable way of storing information or passing it between system components.



Like XML, JSON (JavaScript Object Notation) is language-neutral. In general, because JSON does not use element names, it is less verbose than XML and requires less bytes to encode the same information. As a result, for data transport operations, the use of JSON is preferred over XML. For data storage operations (e.g., a configuration file that an application uses at startup) the slight difference in size is less important. What a developer uses depends on preferences.

JSON and XML do have differences and one should not take it that one is a full substitute of another. For example, the names of elements in XML and the support for comments makes it self-describing. Ponder the following XML and JSON (Figure 1).

```
<student>
   <!-- Legal name of student -->
   <firstName>Tim</firstName>
   <lastName>Smith</lastName>
</student>

{
   “firstName” : “Tim”,
   “lastName” : “Smith”
}
```

Figure 1: Similar data stored in XML (top) and JSON (bottom).

Although both store the first and last names of a person, it is not evident in the JSON if the person is a student, teacher, customer, or something else!. XML also supports comments: In the XML, a comment is used to emphasize that this is the student’s legal name. JSON does not have support for comments and it is nowhere emphasized if Tim is a nickname or a legal name. The verbosity of XML (which has advantages) comes at the expense of size – it is clear that less characters are need in JSON to encode identical information.

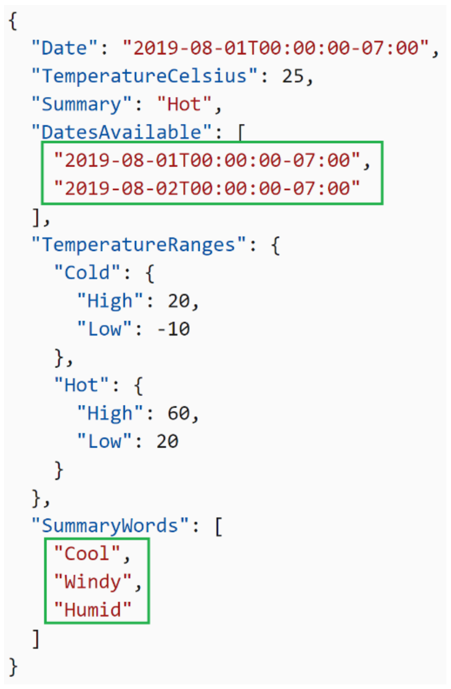
Furthermore, XML stores data in a tree structure, while JSON stores data in map (key : value pairs).

Another important difference is that XML grammar can be validated against a schema that can also ne written in XML and embedded in the document. JSON doesn't have native support for that. You need extensions to provide this capability. To learn more how the grammar of JSON can be validated, search the web for "JSON Schema".

Putting the differences aside, what we are more interested in is their similarities: that is both are used to store or transport data that is human and machine readable and writable. Both are widely used and you need to know both.

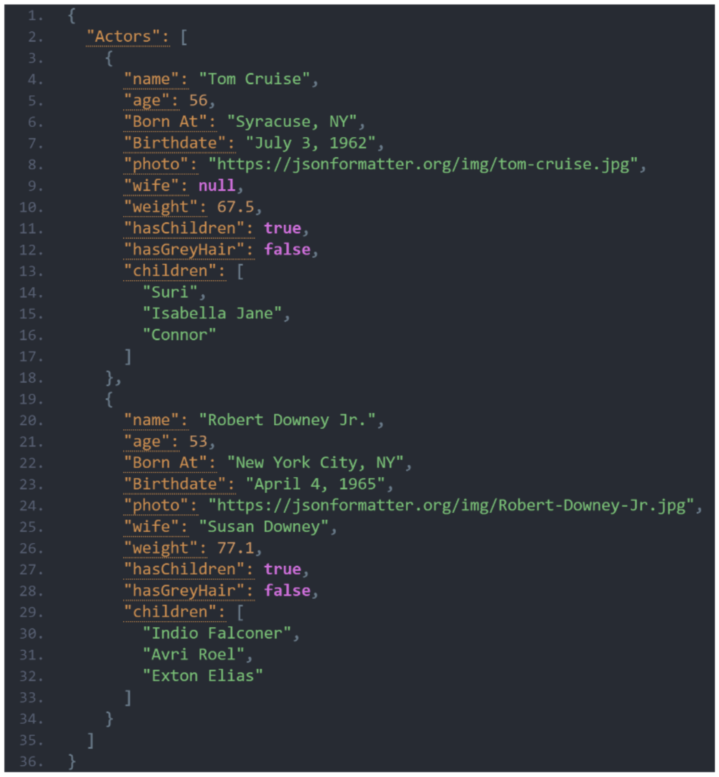
Structure of JSON

Read this for the general structure of JSON. Notice the slight differences in key-value pairs and arrays where [and] are used to specify a list of values. Array items are separated by commas and each list item can be a single value (Figure 2) or a complex JSON (Figure 3).



```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "TemperatureRanges": {
    "Cold": {
      "High": 20,
      "Low": -10
    },
    "Hot": {
      "High": 60,
      "Low": 20
    }
  },
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
```

Figure 2: JSON arrays using the [and] brackets (example obtained from Microsoft documentation).

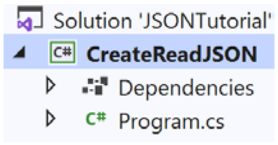


```
{
  "Actors": [
    {
      "name": "Tom Cruise",
      "age": 56,
      "Born At": "Syracuse, NY",
      "Birthdate": "July 3, 1962",
      "photo": "https://jsonformatter.org/img/tom-cruise.jpg",
      "wife": null,
      "weight": 67.5,
      "hasChildren": true,
      "hasGreyHair": false,
      "children": [
        "Suri",
        "Isabella Jane",
        "Connor"
      ]
    },
    {
      "name": "Robert Downey Jr.",
      "age": 53,
      "Born At": "New York City, NY",
      "Birthdate": "April 4, 1965",
      "photo": "https://jsonformatter.org/img/Robert-Downey-Jr.jpg",
      "wife": "Susan Downey",
      "weight": 77.1,
      "hasChildren": true,
      "hasGreyHair": false,
      "children": [
        "Indio Falconer",
        "Avri Roel",
        "Exton Elias"
      ]
    }
  ]
}
```

Figure 3: JSON arrays using the [and] brackets. In this case each list item is an actor defined by a complex JSON structure (example obtained from here).

Manipulating JSON In Code

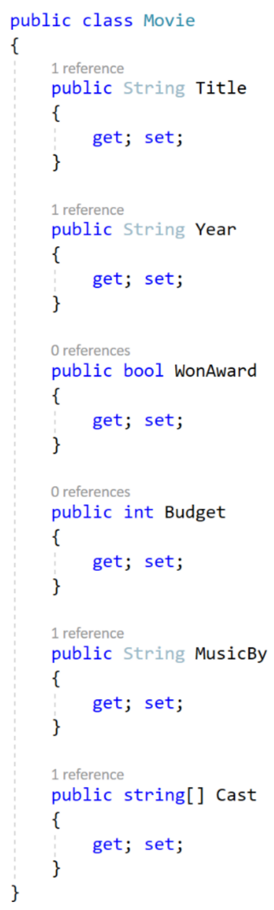
1. Create a Visual Studio blank solution and name it JSONTutorial.
2. Add to the above solution and project of type Console App (.NET Core) and name it CreateReadJSON.



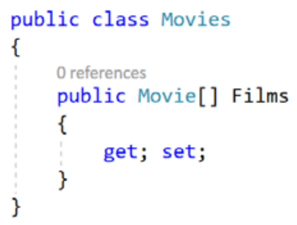
3. Remove unneeded code and add two additional namespaces that have the JSON classes you will use:

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;
```

4. Add two classes to the project: **Movie** and **Movies** (you add a class by right-clicking the project and selecting menu **Add → class…** (Figure 4).



```
public class Movie
{
    1 reference
    public String Title
    {
        get; set;
    }

    1 reference
    public String Year
    {
        get; set;
    }

    0 references
    public bool WonAward
    {
        get; set;
    }

    0 references
    public int Budget
    {
        get; set;
    }

    1 reference
    public String MusicBy
    {
        get; set;
    }

    1 reference
    public string[] Cast
    {
        get; set;
    }
}
```



```
public class Movies
{
    0 references
    public Movie[] Films
    {
        get; set;
    }
}
```

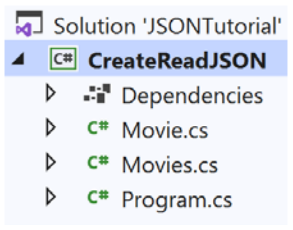
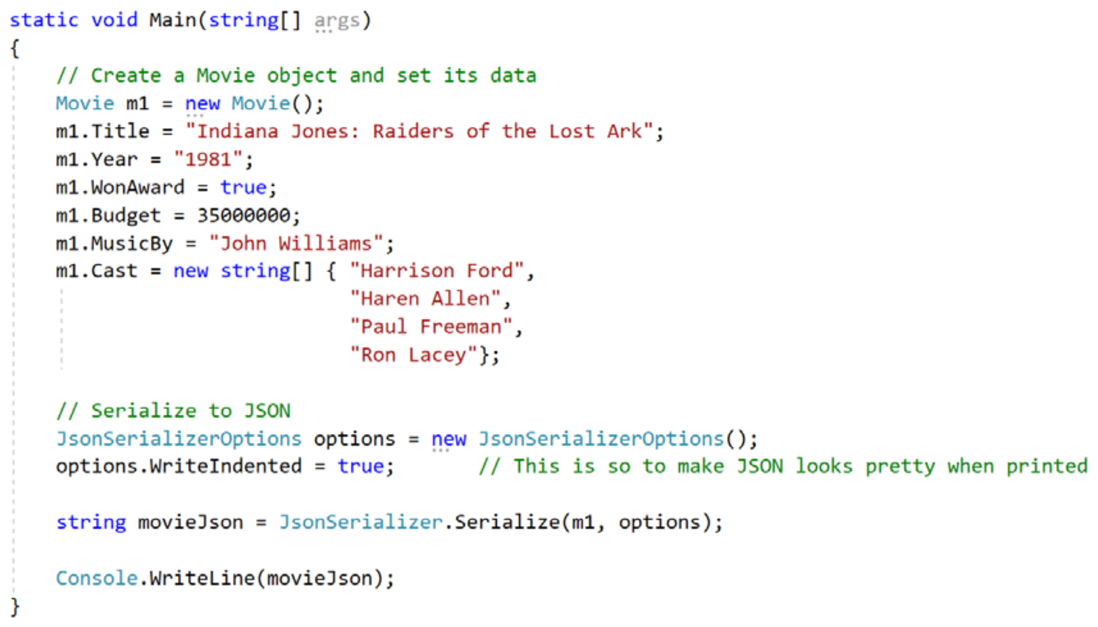


Figure 4: The Movie and Movies classes. The Movies class has one field (Films) which is an array of Movie objects.

5. Write the following code and run the program:

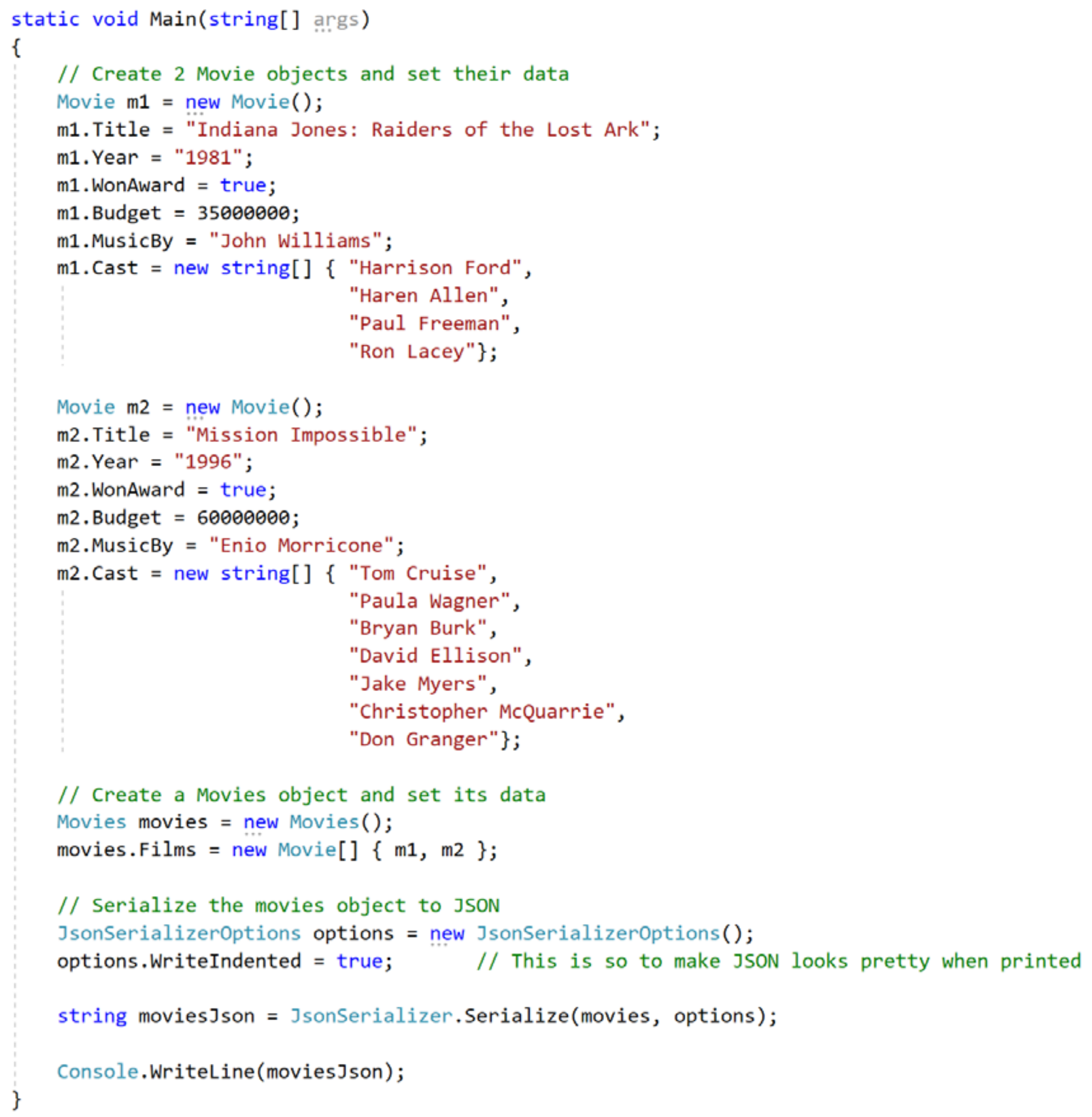


```
static void Main(string[] args)
{
    // Create a Movie object and set its data
    Movie m1 = new Movie();
    m1.Title = "Indiana Jones: Raiders of the Lost Ark";
    m1.Year = "1981";
    m1.WonAward = true;
    m1.Budget = 35000000;
    m1.MusicBy = "John Williams";
    m1.Cast = new string[] { "Harrison Ford",
                             "Haren Allen",
                             "Paul Freeman",
                             "Ron Lacey"};

    // Serialize to JSON
    JsonSerializerOptions options = new JsonSerializerOptions();
    options.WriteIndented = true;       // This is so to make JSON looks pretty when printed

    string movieJson = JsonSerializer.Serialize(m1, options);

    Console.WriteLine(movieJson);
}
```

Look at the printed output and notice the following:

a. The Cast (defined as an array in Movie) appears as an array in JSON.
b. WonAward is a Boolean.
c. Budget is a number.

6. Now let's change the code to create an array of complex JSON structures. Update the code to make it look like the below:



```
static void Main(string[] args)
{
    // Create 2 Movie objects and set their data
    Movie m1 = new Movie();
    m1.Title = "Indiana Jones: Raiders of the Lost Ark";
    m1.Year = "1981";
    m1.WonAward = true;
    m1.Budget = 35000000;
    m1.MusicBy = "John Williams";
    m1.Cast = new string[] { "Harrison Ford",
                             "Haren Allen",
                             "Paul Freeman",
                             "Ron Lacey"};

    Movie m2 = new Movie();
    m2.Title = "Mission Impossible";
    m2.Year = "1996";
    m2.WonAward = true;
    m2.Budget = 60000000;
    m2.MusicBy = "Enio Morricone";
    m2.Cast = new string[] { "Tom Cruise",
                             "Paula Wagner",
                             "Bryan Burk",
                             "David Ellison",
                             "Jake Myers",
                             "Christopher McQuarrie",
                             "Don Granger"};

    // Create a Movies object and set its data
    Movies movies = new Movies();
    movies.Films = new Movie[] { m1, m2 };

    // Serialize the movies object to JSON
    JsonSerializerOptions options = new JsonSerializerOptions();
    options.WriteIndented = true;       // This is so to make JSON looks pretty when printed

    string moviesJson = JsonSerializer.Serialize(movies, options);

    Console.WriteLine(moviesJson);
}
```

Run the program and observe the output. You should notice that you now have an array of movies where each movie is a complex structure. Also notice that in this case we had to serialize an object of type Movies (not Movie).

7. The above two examples showed how to create JSON by serializing objects. To read an existing JSON you do the opposite: de-serialize a string into an object.

Creating JSON: Object $\xrightarrow{\text{Serialize}}$ JSON as a string

Reading JSON: JSON as a string $\xrightarrow{\text{De-Serialize}}$ Object

Add the enclosed **json1.txt** file to the project and set its **Copy to Output Directory** property to

Copy if newer.

8. Comment out the code of step 6. Then add this code:

```
// Get the content of file json1.txt
string jsonString = System.IO.File.ReadAllText("json1.txt");

// De-serialize into an object (here we are doing the opposite of the above)
// That is, we have the JSON (jsonString) and we are converting it to a Movie object
Movie movie = JsonSerializer.Deserialize<Movie>(jsonString);

// Now we can access the data in movie
Console.WriteLine("Title: {0}, WonAwards: {1}\n", movie.Title, movie.WonAward);
```

Run it and verify that you can print the content of the JSON (now de-serialized into a movie object).

Exercise To Do On Your Own

1. Create a project of type Console App (.NET Core) and name it JSONExample.
2. Create a class of your choice (kind of similar to Movie – but choose something else).
3. Add code to serialize your object of choice to a JSON string.
4. Save the JSON string in a file (say myJson.txt) and add it to the project.
5. Add more code to de-serialize it back into an object. That is, read the JSON string into an instance of the class you created.

What to Submit

Nothing to submit for this module