



Working With JSON In C#



Uladzislau Baryshchyk
17 Feb 2021 CPOL

Easy way to work with JSON file in C#/.NET

In this article, I show you the basic operations with JSON. This is one of the key topics in learning C#.

Introduction

Today, JSON is one of the main formats for representing complex structures and data exchange. Therefore, all major programming languages have built-in support for working with it. C# is no exception. In this article, I would like to tell you about an easy way to work with JSON file in C#/.NET. I would like to show you simple examples with this format.

Background

There are two main ways to work .NET objects to JSON:

- Using the **DataContractJsonSerializer** class
- Using the **JavaScriptSerializer** class

Let's look at the **JavaScriptSerializer** class. This class allows you to serialize an object to JSON and, conversely, deserialize JSON code to a C# object.

To save an object to JSON, the **JsonSerializer** class defines a **static** method called **Serialize()**, which has a number of overloads. Some of them are:

```
string Serialize <T> (T obj, JsonSerializerOptions options)
```

The typed version serializes the **obj** object of type **T** and returns the JSON code as a **string**.

```
string Serialize (Object obj, Type type, JsonSerializerOptions options)
```

Serializes **obj** of **type type** and returns JSON code as **string**. The last optional parameter, **options**, allows you to specify additional serialization options.

```
Task SerializeAsync (Object obj, Type type, JsonSerializerOptions options)
```

Serializes **obj** of **type type** and returns JSON code as **string**. The last optional parameter, **options**, allows you to specify additional serialization options.

```
Task SerializeAsync <T> (T obj, JsonSerializerOptions options)
```

The typed version serializes the **obj object** of type **T** and returns the JSON code as a **string**.

```
object Deserialize (string json, Type type, JsonSerializerOptions options)
```

Deserializes the JSON **string** into an **object** of **type type** and returns the deserialized object. The last optional parameter, **options**, allows you to specify additional deserialization options.

```
T Deserialize <T> (string json, JsonSerializerOptions options)
```

Deserializes a **json string** into an **object** of type **T** and returns it.

```
ValueTask <object> DeserializeAsync
(Stream utf8Json, Type type, JsonSerializerOptions options, CancellationToken token)
```

Deserializes UTF-8 text that represents a JSON object into an **object** of **type type**. The last two parameters are optional: **options** allows you to set additional deserialization options, and **token** sets a **CancellationToken** to cancel the task. Returns a deserialized **object** wrapped in a **ValueTask**.

```
ValueTask <T> DeserializeAsync <T>
(Stream utf8Json, JsonSerializerOptions options, CancellationToken token)
```

Deserializes the UTF-8 text that represents a JSON **object** into an **object** of **type T**. Returns a deserialized **object** wrapped in a **ValueTask**.

Getting the New JSON Library

- If you're targeting .NET Core. Install the latest version of the NET Core. This gives you the new JSON library and the ASP.NET Core integration.
- If you're targeting .NET Standard or .NET Framework. Install the **System.Text.Json** NuGet package (make sure to include previews and install version 4.6.0-preview6.19303.8 or higher). In order to get the integration with ASP.NET Core, you must target .NET Core 3.0.

Points of Interest

Let's consider the use of the class using a simple example. Serialize and deserialize the simplest **object**:

```
class Cellphone
{
    public string Name { get; set; }
    public float Price { get; set; }
}
class Program
{
    static void Main(string[] args)
    {
        Cellphone cellphone1 = new Cellphone { Name = "Iphone 12", Price = 750.00f };
        string json = JsonSerializer.Serialize<Cellphone>(cellphone1);
        Console.WriteLine(json);
        Cellphone restoredcellphone = JsonSerializer.Deserialize<Cellphone>(json);
        Console.WriteLine(restoredcellphone.Name + " " +
            Convert.ToString( restoredcellphone.Price));
        Console.ReadLine();
    }
}
```

In this case, we have the **object** with type **Cellphone** and then this **object** was serialized by method **JsonSerializer.Serialize()**. Then we made deserialize from this JSON in the **object** with type **Cellphone** using method **JsonSerializer.Deserialize()**. In Image 1, we have the result:

```

C:\Users\vlaslau\source\repos\Json\bin\Debug\Json.exe
{"Name": "Iphone 12", "Price": 750}
Iphone 12 750

```

Image 1 – The result by JsonSerializer.Serialize() and JsonSerializer.Deserialize() methods.

Moreover, for serializing/ deserializing, we can use a structure.

Only **public** properties of an **object** (with the **public** modifier) are subject to serialization.

The **object** being deserialized must have a parameterless constructor. For example, in the example above, this is the default constructor, but you can also explicitly define a similar constructor in a class.

Writing and Reading JSON File

We can create a JSON file because **SerializeAsync/DeserializeAsync** can accept **stream** for saving and writing data. Let's consider an instance.

```

class Cellphone
{
    public string Name { get; set; }
    public float Price { get; set; }
}
class Program
{
    static async Task Main(string[] args)
    {
        using (FileStream fs = new FileStream("cellphone.json", FileMode.OpenOrCreate))
        {
            Cellphone cellphone1 = new Cellphone { Name = "Iphone 12", Price = 750.00f };
            await JsonSerializer.SerializeAsync<Cellphone>(fs, cellphone1);
            Console.WriteLine("We are done.File has benn saved");
        }
        using (FileStream fs = new FileStream("cellphone.json", FileMode.OpenOrCreate))
        {
            Cellphone restoredcellphone1 =
                await JsonSerializer.DeserializeAsync<Cellphone>(fs);
            Console.WriteLine($"Name: {restoredcellphone1.Name}
                Price: {restoredcellphone1.Price}");
        }
        Console.ReadLine();
    }
}

```

Here, we used **using** because **FileStream** is an uncontrolled resource and **Data** has been written and read.

Serialization Settings by JsonSerializerOptions

By default, the **JsonSerializer** serializes **objects** to minified code. With an add-on package like **JsonSerializerOptions**, you can customize the serialization / deserialization engine using the **JsonSerializerOptions** properties. Some of its properties are as given below:

- 🔑 **IgnoreReadOnlyProperties**: Similarly sets whether read-only properties are serialized
- 🔑 **IgnoreNullValues**: Sets whether to serialize / deserialize in json objects and their properties to **null**
- 🔑 **WriteIndented**: Sets whether spaces are added to json (relatively speaking, for beauty). If set correctly, extra spaces
- 🔑 **AllowTrailingCommas**: An element whether to add a comma after the last one to json. If equal is **true**, a comma is added

Customizing Serialization using Attributes

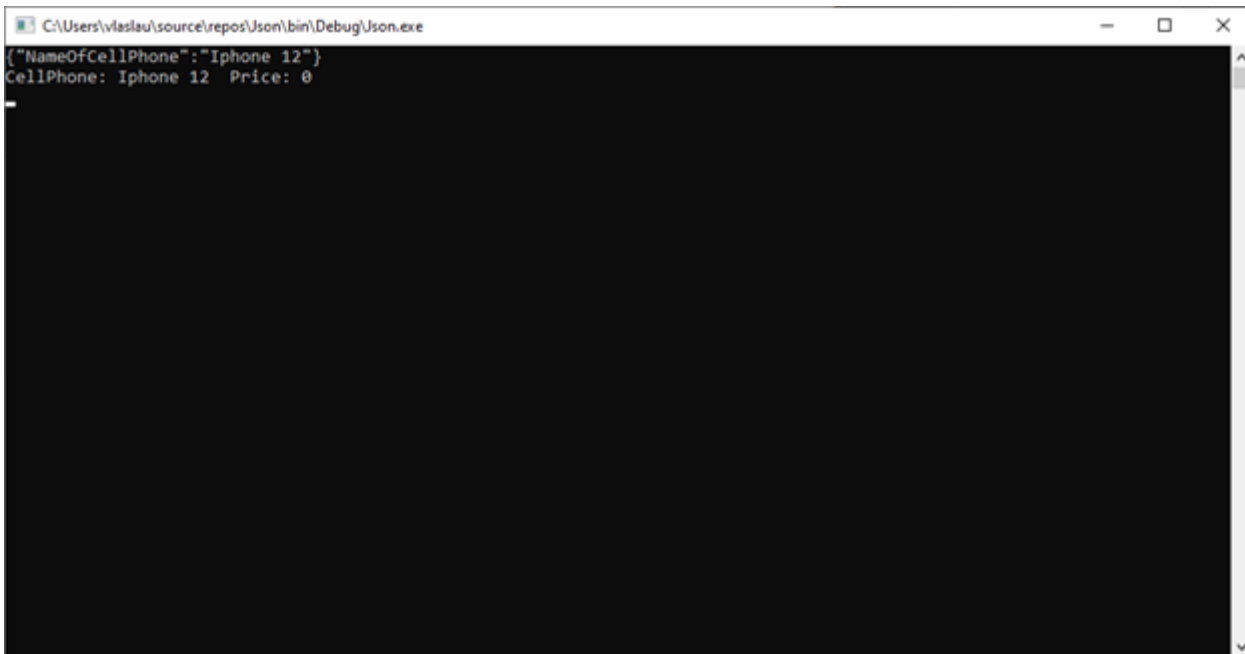
By default, all **public** properties are serialized. Also, in the output json object, all property names match the property names of the C # object. However, using the **JsonIgnore** and **JsonPropertyName** attributes.

The **JsonIgnore** attribute allows you to exclude a specific property from serialization. **JsonPropertyName** allows you to override the original property name. Let's consider an example:

```
class Cellphone
{
    [JsonPropertyName("NameOfCellPhone")]
    public string Name { get; set; }
    [JsonIgnore]
    public float Price { get; set; }
}
class Program
{
    static void Main(string[] args)
    {
        Cellphone cellphone1 = new Cellphone() { Name = "Iphone 12", Price = 750.00f };
        string json = JsonSerializer.Serialize<Cellphone>(cellphone1);
        Console.WriteLine(json);
        Cellphone restoredPerson = JsonSerializer.Deserialize<Cellphone>(json);
        Console.WriteLine($"CellPhone: {restoredPerson.Name}
                          Price: {restoredPerson.Price}");

        Console.ReadLine();
    }
}
```

The result is on image 2.



```
C:\Users\vlaslau\source\repos\json\bin\Debug\json.exe
{"NameOfCellPhone": "Iphone 12"}
CellPhone: Iphone 12 Price: 0
```

Image 2 - The result of `JsonIgnore`

In this case, the **Price** property was ignored and the " **NameOfCellPhone** " alias will be used for the **CellPhone** property. Note that since the **Priceproperty** has not been serialized, it uses its default value when deserialized.

In conclusion, JSON is one of the main formats for representing complex structures and data exchange. In addition, all major programming languages have built-in support for working with it and C # is no exception.

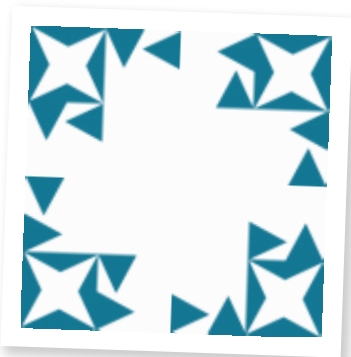
History

- 17th February, 2021: Initial version

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author




Uladzislau Baryshchyk

United States 

No Biography provided

Comments and Discussions

 **0 messages** have been posted for this article Visit <https://www.codeproject.com/Tips/5295019/Working-With-JSON-In-Csharp> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#)
[Advertise](#)
[Privacy](#)
[Cookies](#)
[Terms of Use](#)

Article Copyright 2021 by Uladzislau
Baryshchyk
Everything else Copyright © [CodeProject](#),
1999-2021

Web01 2.8.20210217.1