C#/.NET Console Argument Parser and Validation with ConsoleCommon

Yisrael Lax
11 Nov 2020 CPOL

.NET Library for automatically validating and casting console input parameters

The simplest and most configurable .net argument parser on the market!

Download library - 17.3 KB

Download source - 55.3 KB

Introduction

`ConsoleCommon` is a .NET library that provides a set of helper tools intended for use with console applications. These tools focus on automating argument typing and validation, and creating help text.

Available on Nuget: Search for ConsoleCommon

Background

To better understand what these tools do, consider the following scenario:

A programmer creates a console application that searches a database for a specific customer. The console application requires the following arguments: first name, last name and date of birth. In order to call the console application, a user would type something like the following at the command line:

```
CustomerFinder.exe Yisrael Lax 11-28-1987
```

In the code, the application would first have to parse the user's input arguments. In a robust design, the application would then create a `customer` object that has a first name, last name, and date of birth property. It would then set each of these properties to the values passed in from the user. Several issues arise immediately. The first is that the application would have to require the user to pass in each argument in a pre-set order (first name then last name then date of birth, in our example). Next, to be robust, the application would do some type validation prior to setting fields on the new customer object in order to avoid a type mismatch error. If the user, for example, passed in an invalid date, the application should know that before attempting to set the date field on the `customer` object and throw an appropriate error with a descriptive message to alert the user of what the issue it encountered was. Next, the application would conduct other validation checks. For example, let's say that the application considers it invalid to pass in a date that is set for the future. Doing so then should cause an error.

The `ConsoleCommon` toolset solves all of these issues through automatic typing and error validation. In addition, `ConsoleCommon` also provides some automatic help text generation. A good console application should always come packaged with good help text which should typically be displayed to the user upon typing something like the following:

```
CustomerFinder.exe /?
```

This help text is often annoying to create, properly format, and maintain. ConsoleCommon makes this easy, as you will see below.

Basic Implementation Example

ConsoleCommon works by implementing the abstract ParamsObject class. This class will contain strongly typed properties that represent the input arguments for the application. These properties are created on the fly and, in order to indicate to the ConsoleCommon library that these are arguments, they must be decorated with the Switch attribute. For the *CustomerFinder.exe* application, we will implement a CustomerParamsObject. First, reference the *ConsoleCommon .dll* and create a using statement for ConsoleCommon. Then, implement the class and create some basic switch properties:

```
using System;
using ConsoleCommon;
namespace ConsoleCommonExplanation
{
    public class CustomerParamsObject : ParamsObject
    {
        public CustomerParamsObject(string[] args)
            : base(args)
        {

        }
        [Switch("F")]
        public string firstName { get; set; }
        [Switch("L")]
        public string lastName { get; set; }
        [Switch("DOB")]
        public DateTime DOB { get; set; }
    }
}
```

The string values in the Switch attributes ("F", "L", and "DOB") specify how the application requires the user to pass in arguments. Rather than specifying that the user must pass in arguments in a specific order, ConsoleCommon has users pass in arguments using switches, in any order, such as in the following example:

```
CustomerFinder.exe /F:Yisrael /DOB:11-28-1987 /L:Lax
```

Each "switch" is an argument combined with a switch name. Users specify switches using the format /[SwitchName]:[Argument]. In the above example, "/F:Yisrael" is a single switch. ConsoleCommon will then search the input arguments for switch values that align with switch properties defined on CustomerParamsObject, it will then do type checking, execute some automatic validation, and, if everything passes, set the switch properties on the CustomerParamsObject to the input arguments. To consume the CustomerParamsObject, our client will look like this:

```
using System;
using ConsoleCommon;
namespace ConsoleCommonExplanation
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                 //This step will automatically cast the string args to a strongly typed
object:
                CustomerParamsObject _customer = new CustomerParamsObject(args);

                //This step will do type checking and validation
                _customer.CheckParams();

                string _fname = _customer.firstName;
                string _lname = _customer.lastName;
                DateTime _dob = _customer.DOB;
            }
```

```
            catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

Switch properties can also be specified using their property names as such:

```
CustomerFinder.exe /FirstName:Yisrael /DOB:11-28-1987 /LastName:Lax
```

Beginning with v4.0, specifying a switch name in the SwitchAttribute it is no longer required. If one is not used, callers are required to use the property name to specify the switch.

Basic Validation

All validation is executed when `CheckParams()` is called. Always wrap this method call in a `try... catch` because validation errors result in a descriptive exception being throw. When the `ParamsObject` is instantiated, errors resulting from type mismatches (example, the user passed in an invalid date for the "`DOB`" switch) and invalid switches are queued, These errors are then thrown when `CheckParams()` is called.

Optional Validation

Aside from type checking and switch validation, additional "optional" validation is available on the `ParamsObject`. Unlike basic validation where the validation check happens upon instantiation but is only executed when `CheckParams()` is called, optional validation is both checked and executed when `CheckParams()` is called.

Required

A `switch` property can be marked as "required" or "optional" by passing in an additional parameter into the property's `SwitchAttribute` constructor. By default, switches are not required. In the following example, "`firstName`", and "`lastName`" are required and "`DOB`" is optional:

```
using System;
using ConsoleCommon;

namespace ConsoleCommonExplanation
{
    public class CustomerParamsObject : ParamsObject
    {
        public CustomerParamsObject(string[] args)
            : base(args)
        {

        }
        [Switch("F", true)]
        public string firstName { get; set; }
        [Switch("L", true)]
        public string lastName { get; set; }
        [Switch("DOB", false)]
        public DateTime DOB { get; set; }
    }
}
```

Users now can choose whether or not to pass in a date of birth, but must pass in, at the minimum, a first name and last name. In order to cause `ConsoleCommon` to check for required fields and other types of validation checks that are covered below, use the following command:

```
_customer.CheckParams();
```

Always wrap the `CheckParams()` call in a `try… catch` because any validation error causes a descriptive exception to be thrown.

Restricted Value Set

To restrict the set of values a single argument can be set to, set the "`switchValues`" constructor argument in the `SwitchAttribute` to a list of values. For example, say you only want to allow users to search for people with the last name "`Smith`", "`Johnson`", or "`Nixon`". Modify the `CustomerParamsObject` to look like this:

```
using System;
using ConsoleCommon;

namespace ConsoleCommonExplanation
{
    public class CustomerParamsObject : ParamsObject
    {
        public CustomerParamsObject(string[] args)
            : base(args)
        {

        }
        [Switch("F", true)]
        public string firstName { get; set; }
        [Switch("L", true , -1,  "Smith", "Johnson", "Nixon")]
        public string lastName { get; set; }
        [Switch("DOB", false)]
        public DateTime DOB { get; set; }
    }
}
```

Note the modified `SwitchAttribute` on the `lastName` property. If the customer passes in a last name other than one in the restricted list defined above, `ConsoleCommon` will throw an error as soon as `CheckParams()` is called. The '`-1`' value is used to set the default ordinal value to its default value so `ConsoleCommon` ignores it. We will explain default ordinals later.

Another way to establish a restricted list is to use an `enum` type for your switch property. `ConsoleCommon` will automatically restrict the value set of the `enum` property to the names of the `enum`'s values. Notice that, in the below example, we've created a new `enum LastNameEnum` and changed the type of the `lastName` property from `string` to `LastNameEnum`:

```
using System;
using ConsoleCommon;

namespace ConsoleCommonExplanation
{
    public enum LastNameEnum
    {
        Smith,
        Johnson,
        Nixon
    }
    public class CustomerParamsObject : ParamsObject
    {
        public CustomerParamsObject(string[] args)
            : base(args)
        {

        }
        [Switch("F", true)]
        public string firstName { get; set; }
        [Switch("L", true)]
        public LastNameEnum lastName { get; set; }
        [Switch("DOB", false)]
        public DateTime DOB { get; set; }
```

```
    }
}
```

Custom Validation

If additional validation is needed aside from the built-in validation, you can implement the `GetParamExceptionDictionary()` method on the `ParamsObject` implementation. This allows you to add additional validation checks that are called when `CheckParams()` is called. This method requires a `Dictionary` to be returned. Each entry in this `Dictionary` contains a `Func<bool>` which contains a validation check, paired with a `string` containing an exception message to be returned in the case that the validation check fails. `Func`s are used for their delayed processing feature. When `CheckParams()` is called, `ConsoleCommon` iterates through the `Dictionary` returned from the `GetParamExceptionDictionary()` method, processes each `Func<bool>`, and, if any are `false`, throws an error with the `Func`'s paired `string` message as the exception message. In the below example, we've added an exception check that ensures a user does not pass in a future date for the date of birth field:

```
public override Dictionary<Func<bool>, string> GetParamExceptionDictionary()
{
    Dictionary<Func<bool>, string> _exceptionChecks = new Dictionary<Func<bool>, string>();

    Func<bool> _isDateInFuture = new Func<bool>( () => DateTime.Now <= this.DOB );

    _exceptionChecks.Add(_isDateInFuture,
                         "Please choose a date of birth that is not in the future!");
    return _exceptionChecks;
}
```

If the user does do so, when `CheckParams()` is called, an exception with the message "Please choose a date of birth that is not in the future!" will be returned to the caller.

Automatic Help Text Generation

The user triggers printing out of help text by passing "/?", "/help" or "help" as the first argument into the application:

The most basic way to implement help text generation is by overriding the `GetHelp()` method on the `ParamsObject` implementation. In our case, that's the `CustomerParamsObject`:

```
    public class CustomerParamsObject : ParamsObject
    {
        public CustomerParamsObject(string[] args)
            : base(args)
        {

        }

        #region Switch Properties
        [Switch("F", true)]
        public string firstName { get; set; }
        [Switch("L", true)]
        public LastNameEnum lastName { get; set; }
        [Switch("DOB", false)]
        public DateTime DOB { get; set; }
        #endregion

        public override Dictionary<Func<bool>, string> GetParamExceptionDictionary()
```

```
        {
            Dictionary<Func<bool>, string> _exceptionChecks = new Dictionary<Func<bool>,
string>();

            Func<bool> _isDateInFuture = new Func<bool>( () => DateTime.Now <= this.DOB );

            _exceptionChecks.Add(_isDateInFuture,
                                 "Please choose a date of birth that is not in the future!");
            return _exceptionChecks;
        }

        public override string GetHelp()
        {
            return
             "\n-----------This is help-----------\n\nBe smart!\n\n----------------------------
------";
        }
    }
}
```

Then, the client would call the `GetHelp()` method like this:

```
//CustomerParamsObject _customer = new CustomerParamsObject(args)
String _helptext = _customer.GetHelp();
Console.Writeline(_helptext);
```

However, this approach doesn't take advantage of `ConsoleCommon`'s more automatic features surrounding help text generation.

Help Text Properties

`ConsoleCommon` provides some automatic formatting features to assist in creating help text. To take advantage of these, do not override the `GetHelp()` method. Rather, create `string` properties on the `ParamsObject` implementation and decorate them with the `HelpTextAttribute`. These properties return a single help text component. Note the new `Description` and `ExampleText` help text properties, below:

```
using System;
using ConsoleCommon;
using System.Collections.Generic;

namespace CustomerFinder
{
    public enum LastNameEnum
    {
        Smith,
        Johnson,
        Nixon
    }
    public class CustomerParamsObject : ParamsObject
    {
        public CustomerParamsObject(string[] args)
            : base(args)
        {

        }

        #region Switch Properties
        [Switch("F", true)]
        public string firstName { get; set; }
        [Switch("L", true)]
        public LastNameEnum lastName { get; set; }
        [Switch("DOB", false)]
        public DateTime DOB { get; set; }
        #endregion

        public override Dictionary<Func<bool>, string> GetParamExceptionDictionary()
```

```
        {
            Dictionary<Func<bool>, string> _exceptionChecks = new Dictionary<Func<bool>,
string>();

            Func<bool> _isDateInFuture = new Func<bool>( () => DateTime.Now <= this.DOB );

            _exceptionChecks.Add(_isDateInFuture,
                                 "Please choose a date of birth that is not in the future!");
            return _exceptionChecks;
        }

        [HelpText(0)]
        public string Description
        {
            get { return "Finds a customer in the database."; }
        }
        [HelpText(1, "Example")]
        public string ExampleText
        {
            get { return "This is an example: CustomerFinder.exe Yisrael Lax 11-28-1987"; }
        }
    }
}
```

The `HelpTextAttribute`'s constructor must specify an ordinal value, which determines where that help text component appears when the user requests help. By default, the help text component is preceded with the name of the property and a colon. However, the help text component's name can be overridden by specifying a name in the `HelpTextAttribute`'s constructor. The `Description` property above uses the property name and the `ExampleText` property overrides this default behavior and, instead, uses the name "`Example`". The result can be seen below:

Included Help Text Properties

There are several included help text properties that you can use if you choose to. These are the `Usage` property and the `SwitchHelp` property. To use either of these, override them, decorate them with a `HelpTextAttribute` just like you would with any other help text property, but return the base's implementation of the property instead of a custom implementation:

```
[HelpText(2)]
public override string Usage
{
    get { return base.Usage; }
}
```

Usage Property

This help text property will print out an example of how to call the application:

```
USAGE:                CustomerFinder.exe /F:"firstName" /L:"lastName" /DOB:"DOB"
```

SwitchHelp Property

This help text property prints out help text for each switch property/argument and requires additional work before it will print out anything meaningful. To implement this, decorate each of the switch properties with a `SwitchHelpTextAttribute` and pass in to the attribute's constructor some basic help text for that switch property. In the following example, note the new attributes decorating the switch properties and the new `Description` and `Usage` properties:

```
using System;
using ConsoleCommon;
using System.Collections.Generic;

namespace CustomerFinder
{
    public enum LastNameEnum
    {
        Smith,
        Johnson,
        Nixon
    }
    public class CustomerParamsObject : ParamsObject
    {
        public CustomerParamsObject(string[] args)
            : base(args)
        {

        }

        #region Switch Properties
        [Switch("F", true)]
        [SwitchHelpText("First name of customer.")]
        public string firstName { get; set; }
        [Switch("L", true)]
        [SwitchHelpText("Last name of customer.")]
        public LastNameEnum lastName { get; set; }
        [Switch("DOB", false)]
        [SwitchHelpText("The date of birth of customer")]
        public DateTime DOB { get; set; }
        #endregion

        public override Dictionary<Func<bool>, string> GetParamExceptionDictionary()
        {
            Dictionary<Func<bool>, string> _exceptionChecks = new Dictionary<Func<bool>,
string>();

            Func<bool> _isDateInFuture = new Func<bool>( () => DateTime.Now <= this.DOB );

            _exceptionChecks.Add(_isDateInFuture,
                                 "Please choose a date of birth that is not in the future!");
            return _exceptionChecks;
        }

        [HelpText(0)]
        public string Description
        {
            get { return "Finds a customer in the database."; }
        }
        [HelpText(1, "Example")]
        public string ExampleText
        {
            get { return "This is an example: CustomerFinder.exe Yisrael Lax 11-28-1987"; }
        }
        [HelpText(2)]
        public override string Usage
        {
            get { return base.Usage; }
        }
        [HelpText(3,"Parameters")]
        public override string SwitchHelp
        {
            get { return base.SwitchHelp; }
        }
    }
}
```

And, the output is:

As you can tell, the `SwitchHelp` help text property provides quite a bit of information that you did not specify explicitly including whether or not the argument is required, if there is a default ordinal, and a list of allowed values for that property.

Starting with v4.0, you can now add help text directly to a SwitchAttribute instead of adding an additional SwitchHelpTextAttribute to a property. For example:

```
public class CustomerParamsObject : ParamsObject
{
     public CustomerParamsObject(string[] args) : base(args) { }

     //...
     #region Switch Properties
     [Switch("F", true, helptext: "First name of customer.")]
     public string firstName { get; set; }

     //...

     [HelpText(3,"Parameters")]
     public override string SwitchHelp => base.SwitchHelp;
}
```

Getting Help If Needed

In most cases, you only want to print out help to the console if the user requested it. To do this, use the `GetHelpIfNeeded()` method, which returns a `string` containing the help text if the user passed in a help switch ("`/?`", "`/help`", or "`help`") as the first argument, or a `string.Empty` if the user did not. A simple implementation looks like this:

```
using System;

namespace CustomerFinder
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                //This step will do type validation
                //and automatically cast the string args to a strongly typed object:
                CustomerParamsObject _customer = new CustomerParamsObject(args);
                //This step does additional validation
                _customer.CheckParams();
                //Get help if user requested it
                string _helptext = _customer.GetHelpIfNeeded();
                //Print help to console if requested
                if(!string.IsNullOrEmpty(_helptext))
                {
                    Console.WriteLine(_helptext);
                    Environment.Exit(0);
                }
                string _fname = _customer.firstName;
                string _lname = _customer.lastName.ToString();
                string _dob = _customer.DOB.ToString();
            }
            catch(Exception ex)
```

```
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

Default Ordinal

Default ordinals are parameters that can be specified for `switch` properties that allow your console application to be called with ordered arguments rather than switches. This feature allows for backwards compatibility in cases where `ConsoleCommon` is being implemented for a console application that is already built and has external batch scripts and applications currently calling it with non-switched ordered arguments such as:

```
CustomerFinder.exe Yisrael Lax 11-28-1987
```

To allow for this implementation, add an ordinal value in each switch property's `SwitchAttribute`:

```
#region Switch Properties
[Switch("F", true,1)]
[SwitchHelpText("First name of customer")]
public string firstName { get; set; }
[Switch("L", true,2)]
[SwitchHelpText("Last name of customer")]
public LastNameEnum lastName { get; set; }
[SwitchHelpText("The date of birth of customer")]
[Switch("DOB", false,3)]
public DateTime DOB { get; set; }
#endregion
```

You'll notice that the help text has changed as well. It now lists `switch` properties in order of their default ordinals as well has some additional messaging surrounding default ordinals.

Default ordinals can get complicated because calling the application with a mix of arguments using default ordinals and arguments using switches is allowed. Making things even more complicated, it is perfectly allowable to have some `switch` properties that have a default ordinal, and some properties that do not. There is some complex logic that determines specific requirements when doing any mixing and matching, which we won't go into detail here. However, much of it is intuitive and, playing around with mixing and matching can help you determine this logic on your own.

Using Type Type Switch Properties

This is not a typo. `Switch` properties can be made to be any type. However, types that are specifically supported include all primitive types, `DateTimes`, `enums`, `Types`, `System.Security.SecureStrings`, `KeyValuePairs`, and any type that implements `IConvertible`. All other types will use the type's default `ToString()` method to attempt to match an argument to the property's value. This may result in erroneous data or unexpected exceptions being thrown.

`Type` types have a specific implementation. The idea of using a `Type` type is so that a user can specify any type that is in any of the assemblies associated with the application by name or by a descriptor specified in a certain type of attribute decorating the type. For example, say we have two classes:

```
class PizzaShopCustomer{ }
class BodegaCustomer { }
```

Now, on our `ParamsObject` implementation, we add a new property "`CustomerType`":

```
#region Switch Properties
[Switch("F", true,1)]
[SwitchHelpText("First name of customer")]
public string firstName { get; set; }
[Switch("L", true,2)]
```

```
[SwitchHelpText("Last name of customer")]
public LastNameEnum lastName { get; set; }
[SwitchHelpText("The date of birth of customer")]
[Switch("DOB", false,3)]
public DateTime DOB { get; set; }
[Switch("T",false)]
public Type CustomerType { get; set; }
#endregion
```

The user now has the ability to specify the type of `Customer` class with the `/T` switch:

(Note that we used a mix of switches and default ordinals in the above example. We also added "`Lax`" to the `LastNameEnum`). There is one annoying aspect of this implementation, though: in our application example, the user is required to input the name of a class, which isn't necessarily user friendly (the user, for example, might not be able to figure out why they have to type the word "`customer`" at the end of the customer type). To work around this, the class can be decorated with a `ConsoleCommon.TypeParamAttribute` that specifies a friendly name:

```
[TypeParam("pizza shop")]
public class PizzaShopCustomer { }
[TypeParam("bodega")]
public class BodegaCustomer { }
```

Now, the user can use the friendly names specified above when calling the application:

Using a `Type` type switch property is most useful when used in conjunction with restricted values feature. When doing it that way, this property then becomes quite useful when used together with a dynamic factory class.

Using Arrays

Arrays can be used as switch properties as long their underlying types are supported. To pass in a switched array value set, enclose the whole value set in double quotes and separate each value with a comma:

```
public class CustomerParamsObject : ParamsObject
{
    //...
    [Switch("NN")]
    public string[] Nicknames { get; set; }
}
```

And, from the console, we'd pass in something like this:

```
CustomerFinder.exe Yisrael Lax /NN:"Codemaster,Devguru"
```

Using Enums

`Enum`s that are decorated with the `FlagsAttribute` can have their values specified in a comma delimited list. This will result in the values being bitwise ORed together so make sure that no two `enum` values can overlap. This is done by using powers of two when specified `enum` values. See the following example:

```
[Flags]
public enum CustomerInterests
{
    Pizza = 1,
    Crabcakes = 2,
    Parachuting = 4,
    Biking = 8
}

public class CustomerParamsObject : ParamsObject
{
    //...
    [Switch("Int")]
    public CustomerInterests Interests { get; set; }
}
```

```
And, from the console, we can pass in something like this:

CustomerFinder.exe Yisrael Lax /Int:"Crabcakes,Biking"
```

Using KeyValuePairs

`KeyValuePair`s can be used by specifying a key-value pair separated by a colon (e.g., "`Key:Value`"). Using arrays of `KeyValuePair`s can be useful for cases where there is a need to fill arbitrary variables, such as in the below example:

```
public class CustomerParamsObject : ParamsObject
{
    //...
    [Switch("Pets")]
    public KeyValuePair<string,int>[] PetCount { get; set; }
}
```

And, from the console, we can pass in something like this:

```
CustomerFinder.exe Yisrael Lax /Pets:"dog:5,cat:3,bird:1"
```

Additional Options

`ConsoleCommon` is fairly customizable. Custom type parsing, custom help handling, and custom switch parsing can all be implemented. However, several features exist for implementors who are simply looking to customize certain options using the default parsing and handling.

Customizing switch start characters, switch end characters, and the legal switch name regex can be done like this:

```
public class CustomerParamsObject : ParamsObject
{
    //...
    public override SwitchOptions Options
    {
        get
        {
            return new SwitchOptions(switchStartChars: new List<char> { '-','/'},
            switchEndChars: new List<char> { ':', '-'},
                                        switchNameRegex: "[_A-Za-z]+[_A-Za-z0-9]*");
        }
    }
}
```

In the above example, all of the following can be used to call the program:

```
CustomerFinder.exe /F:Yisrael /L:Lax
```

```
CustomerFinder.exe -F:Yisrael -L:Lax
```

```
CustomerFinder.Exe -F-Yisrael -L-Lax
```

Customizing which commands request help can be done like this:

```
public class CustomerParamsObject : ParamsObject
{
    //...
    public override List<string> HelpCommands
    {
        get
        {
            return new List<string> { "/?", "help", "/help"};
        }
    }
}
```

Custom Parsing

`ConsoleCommon` comes packaged with a set of default type parsers that can handle the following all primitive types, DateTime, Type, Enum, Array, SecureString, Nullable, Bool (w/ additional values allowed to represent 'true' and 'false'). Starting with version 3.0, implementers can override parsing of any one of the packaged type parsers or add parsing for a non-included type without having to override all of the default parsing. Below is an example of how to override the handling of DateTime types:

```
public class DayFirstDashOnlyDateTimeParser : TypeParserBase<DateTime>
{
     public override object Parse(string toParse, Type typeToParse, ITypeParserContainer
parserContainer)
     {
          CultureInfo _culture = CultureInfo.CreateSpecificCulture("");
          return DateTime.ParseExact(toParse, "dd-MM-yyyy", _culture);
     }
}
public class CustomerParamsObject : ParamsObject
{
    //...
    [Switch("Bday")]
    public DateTime Birthday { get; set; }

    protected override ITypeParserContainer TypeParser => new
TypeParserContainer(overwriteDupes: true, parserContainer: new DefaultTypeContainer(),
typeParsers: new DateTimeParser());
}
```

In the above example, the `TypeParser` property is ovverriden with a new `TypeParserContainer`. An overload for `TypeParserContainer`'s constructor is used that specifies that all the type parsers in `DefaultTypeContainer` should be added to the new `TypeParserContainer`. The `overwriteDupes` parameter specifies that any of the new `ITypeParser`'s that handle types already handled by one of the `ITypeParser`'s in the base container should overwrite the similar `ITypeParser` in the base container.

When `ParamsObject` needs to parse a switch property, it sends a request to `TypeParserContainer` to return a parser that handles that particular property's type. `TypeParserContainer` will return a parser that handles the most derived type that matches the type of the property. For example, if the `TypeParserContainer` had both a `DateTimeParser` and an `ObjectParser` and `ParamsObject` wanted to parse a DateTime property, the `DateTimeParser` would be returned. If only an `ObjectParser` existed, it would be returned.

Non-Arg Parsing

Beginning with v4.0, `ConsoleCommon` has the ability to parse a string containing the switch statements instead of an array. To use this, the constructor overload that intakes a string:

```
class Program
{
    static void Main(string[] args)
    {
       try {
            string _argText = "/F:Yisrael /L:Lax /Bday:11/28/1987";
            CustomerParamsObject _customer = new CustomerParamsObject(_argText);
         } catch Exception ex { Console.WriteLine(ex.Message); }
    }
}
```

ConsoleCommon first splits the string into arguments, then proceeds as normal. Be careful, though. ConsoleCommon does not split strings into arguments in the same way that the Windows Shell or Command Prompt does. Double quotes and most spaces are considered part of the command text and do not affect how arguments are broken up. All characters following a switch key and up to the next switch key or the end of the command text is considered part of the switch's value, except running and trailing spaces. Using this overload is helpful for scenarios where switch values are expected to have allot of spaces. It saves users from having to surround switch arguments (key and value) in quotation marks.

ConsoleCommon also has an overload that will read the command text from the environment. Be aware, though, that this overload expects the environment's current directory to be the same directory as the one from which the application was called. Also, obviously, this overload is only good for when using ConsoleCommon to parse arguments that were passed into the program at start.

Full Example Code

```
using System;
using ConsoleCommon;
using System.Collections.Generic;

namespace CustomerFinder
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                <code>args</code> = new string[] { "Yisrael", "Lax",
                "/Int:Pizza,Parachuting", "/Pets:dog:5,cat:3,bird:1" };//"/T:pizza shop",
                                // "/DOB:11-28-1987", "/Case", "/Regions:Northeast,Central" };
                //This step will do type validation
                //and automatically cast the string args to a strongly typed object:
                CustomerParamsObject _customer = new CustomerParamsObject(args);
                //This step does additional validation
                _customer.CheckParams();
                //Get help if user requested it
                string _helptext = _customer.GetHelpIfNeeded();
                //Print help to console if requested
                if (!string.IsNullOrEmpty(_helptext))
                {
                    Console.WriteLine(_helptext);
                    Environment.Exit(0);
                }
                string _fname = _customer.firstName;
                string _lname = _customer.lastName.ToString();
                string _dob = _customer.DOB.ToString("MM-dd-yyyy");
                string _ctype = _customer.CustomerType == null ? "None" :
```

```
_customer.CustomerType.Name;
                string _caseSensitive = _customer.CaseSensitiveSearch ? "Yes" : "No";
                string _regions = _customer.CustRegions == null ? "None" :
                string.Concat(_customer.CustRegions.Select(r => "," +
r.ToString())).Substring(1);
                string _interests = _customer.Interests.ToString();
                string _petCount = _customer.PetCount == null || _customer.PetCount.Length == 0
?
                "None" : string.Concat(_customer.PetCount.Select
                            (pc => ", " + pc.Key + ": " + pc.Value)).Substring(2);

                Console.WriteLine();
                Console.WriteLine("First Name: {0}", _fname);
                Console.WriteLine("Last Name: {0}", _lname);
                Console.WriteLine("DOB: {0}", _dob);
                Console.WriteLine("Customer Type: {0}", _ctype);
                Console.WriteLine("Case sensitive: {0}", _caseSensitive);
                Console.WriteLine("Regions: {0}", _regions);
                Console.WriteLine("Interests: {0}", _interests);
                Console.WriteLine("Pet count: {0}", _petCount);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }

    [TypeParam("pizza shop")]
    public class PizzaShopCustomer { }
    [TypeParam("bodega")]
    public class BodegaCustomer { }

    public enum LastNameEnum
    {
        Smith,
        Johnson,
        Nixon,
        Lax
    }
    public class CustomerParamsObject : ParamsObject
    {
        public CustomerParamsObject(string[] args)
            : base(args)
        {

        }

        #region Switch Properties
        [Switch("F", true,1)]
        [SwitchHelpText("First name of customer")]
        public string firstName { get; set; }
        [Switch("L", true,2)]
        [SwitchHelpText("Last name of customer")]
        public LastNameEnum lastName { get; set; }
        [SwitchHelpText("The date of birth of customer")]
        [Switch("DOB", false,3)]
        public DateTime DOB { get; set; }
        [Switch("T",false,4, "bodega", "pizza shop")]
        public Type CustomerType { get; set; }
        #endregion

        public override Dictionary<Func<bool>, string> GetParamExceptionDictionary()
        {
            Dictionary<Func<bool>, string> _exceptionChecks = new Dictionary<Func<bool>,
string>();

            Func<bool> _isDateInFuture = new Func<bool>( () => DateTime.Now <= this.DOB );
```

```
            _exceptionChecks.Add(_isDateInFuture,
                                 "Please choose a date of birth that is not in the future!");
            return _exceptionChecks;
        }

        [HelpText(0)]
        public string Description
        {
            get { return "Finds a customer in the database."; }
        }
        [HelpText(1, "Example")]
        public string ExampleText
        {
            get { return "This is an example: CustomerFinder.exe Yisrael Lax 11-28-1987"; }
        }
        [HelpText(2)]
        public override string Usage
        {
            get { return base.Usage; }
        }
        [HelpText(3,"Parameters")]
        public override string SwitchHelp
        {
            get { return base.SwitchHelp; }
        }
    }
}
```

Conclusion

I created this library by piecing together bits of code I had used in various places, then generalizing it so it could be used widely and, of course, adding some additional features. As we coders well know, useful tools often evolve from a very organic process similar to this and, sometimes, they happen almost accidentally. That being said, this library *was* made on the fly and by piecing bits and pieces together so there is definitely room for improvement. Feel free to extend, modify, and improve this so you can also contribute to improving our overall toolset. Happy coding!

History

- Version 1, published 3/31/2017
- Version 5, published 4/12/2018
- Version 6, published 10/30/2018. Fixed bug: args being lowercased. Added: array element values can have spaces in them; arrays can now only be delimited w/ commas.
- Version 7, published 11/2018. Added: ability to instantiate ParamsObject w/o specifying args; args dynamically derived. Added: ability to instantiate ParamsObject w/ specify string w/ all args instead of str array.
- Version 8, published 12/6/2018. Fixed bug: GetHelpIfNeeded() w/ cstor ParamsObject(CommandText).

License

About the Author

Yisrael Lax
United States
No Biography provided

Comments and Discussions

25 messages have been posted for this article Visit **https://www.codeproject.com/Articles/1179863/Csharp-NET-Console-Argument-Parser-and-Validation** to post and view comments on this article, or click **here** to get a print view with messages.