Ookii.CommandLine 2.2

User Guide

Copyright © Sven Groot (Ookii.org) 2013  
For license information see license.txt

# Introduction

Ookii.CommandLine is a library that helps you to parse command line arguments for your applications. It allows you to easily define a set of accepted arguments, and then parse the command line supplied to your application for those arguments. In addition, it allows you to generate usage help from the arguments that you defined which you can display to the user.

Ookii.CommandLine can be used with any kind of .Net application, whether Console, Windows Forms, or WPF. Although a limited subset of functionality—particularly related around generating usage help text—is geared primarily towards console applications that are invoked from the command prompt, the main command line parsing functionality is usable in any application that needs to process command line arguments.

To define a set of command line arguments, you create a class that will hold their values. The constructor parameters and properties of that class determine the set of arguments that are accepted. For each argument you can customize things like the argument name and whether or not an argument is required, and you can specify descriptions used to customize the usage help.

Two samples are provided with the library, one for basic command line parsing and one for shell commands. Both have a C# and Visual Basic version so you can see how to use the Ookii.CommandLine library in your own code. Besides the source code, they are also included as binaries so you can experience how the argument parsing works by trying them out.

# System requirements

Ookii.CommandLine 2.2 is a class library for use in your own applications. It can be used with applications targeting at least one of the following environments:

* [Microsoft .Net Framework](http://msdn.microsoft.com/netframework) 2.0 or later
* [Mono](http://www.mono-project.com/) 2.6 or later, using the .Net 2.0 or .Net 4.0 profile

To view and edit the source code of Ookii.CommandLine or the included sample applications, you must use [Microsoft Visual Studio 2010](http://msdn.microsoft.com/vstudio). Because Ookii.CommandLine targets .Net 2.0, you can use the library with Visual Studio 2005 or later or any other development environment that targets .Net 2.0.

# What’s new in Ookii.CommandLine 2.2

Ookii.CommandLine 2.2 offers the following improvements over version 2.1:

* Added support for alternative names (aliases) for command line arguments.
* An argument’s aliases and default value can be included in the argument description when generating usage.

Ookii.CommandLine 2.1 offers the following improvements over version 2.0:

* Added support for dictionary arguments; these are special multi-value arguments whose values take the form key=value.
* Multi-value arguments can be specified using a read-only property of any collection type (in addition to the previous array support).
* Multi-value properties can optionally use a separator character to allow multiple values to be specified without specifying the argument multiple times.
* Added support for specifying a custom type converter for individual arguments.
* When specifying the default value for an argument defined by a property you can now use any type that can be converted to the argument’s type using its type converter. This makes it possible to define default values for arguments with a type for which there are no literals.
* A CommandLineArgumentException is thrown when the argument type’s constructor or a property setter throws an exception (instead of a TargetInvocationException).
* The CommandLineParser no longer sets the property value for an unspecified argument with a default value of null.
* Shell commands can take their name from the type name.
* Shell commands can use custom argument parsing.
* Various minor bug fixes.

Ookii.CommandLine 2.0 offers the following improvements compared to Ookii.CommandLine 1.0:

* Improved argument parsing:
  + All arguments can be specified by name.
  + Support for using whitespace to separate an argument name from its value.
  + Support for multiple argument name prefixes.
  + Support for using a custom StringComparer for argument name matching (to allow case sensitive or insensitive matching).
  + Support for use a custom CultureInfo for argument value conversion.
  + Non-positional arguments can be required arguments.
* Properties can be used to define positional arguments.
* More customizable generation of usage help text.
* The new shell commands functionality lets you easily create shell utilities with multiple operations that each uses its own command line arguments.
* The LineWrappingTextWriter class provides support for writing word-wrapped text to any output stream, with greater flexibility than the SplitLines method provided in Ookii.CommandLine 1.0.
* Targets .Net 2.0 for wider applicability.

## Upgrading from Ookii.CommandLine 1.0

Ookii.CommandLine 2.0 and newer version have substantial changes from version 1.0 and are not designed to be backwards compatible. There are changes in argument parsing behavior and API names and usage.

Upgrading an existing project that is using Ookii.CommandLine 1.0 to Ookii.CommandLine 2.0 or newer may require substantial code changes and may change how command lines are parsed.

# Command line arguments

Command line arguments are passed to your application when it is started, and are typically accessed through the parameter of the static void Main(string[] args) method (Shared Sub Main(ByVal args() As String) in Visual Basic). This provides with the arguments as an array of strings, which is not terribly useful. What Ookii.CommandLine allows you to do is to convert that array of strings into a strongly typed set of named values, which are stored in the properties of the class that was used to define the arguments.

The method used to extract values from the array of string arguments is determined by the command line argument parsing rules. Ookii.CommandLine uses parsing rules that are very similar to how Microsoft PowerShell parses arguments for cmdlets, so if you have used PowerShell these rules will be familiar with you.

Command line arguments follow the name of your application on the command prompt, and typically take the following form:

-ArgumentName ArgumentValue

The argument name is preceded by the *argument name prefix*. This prefix is configurable, but Ookii.CommandLine defaults to accepting a forward slash (/) and a dash (-) on Windows, and only a dash (-) on other platforms (other platforms are supported through Mono).

The argument value follows the name, separated either by a space or a colon (:). Not all arguments require values; those that do not are called *switch arguments* and have a value determined by their presence or absence on the command line.

An argument can have one or more aliases: alternative names that can also be used to specify the command name. For example, a parameter named “Verbose” might use the alias “v” as a shorter to type alternative.

## Positional arguments

An argument can be *positional*, which means that its value can be specified either by name as indicated above, or by position. In this case the name of the argument is not required, and the argument’s value can be supplied by specifying it in the correct position in relation to other positional arguments.

If an argument value is encountered without being preceded by a name, it is matched to the positional argument at the current position. For example, take the following command line arguments:

Value1 –ArgumentName Value2 Value3

In this case, Value1 is not preceded by a name; therefore it is matched to the first positional argument. Value2 follows a name, so it is matched to the argument with the name “ArgumentName”. Finally, Value3 is matched to the second positional argument.

A positional argument can still be supplied by explicitly supplying its name. If a positional argument is supplied by name, it cannot also be specified by position; in the previous example, if the argument named “ArgumentName” was the second positional argument, then Value3 becomes the value for the third positional argument, because the value for “ArgumentName” was already specified by name.

## Required arguments

A command line argument that is required must be present on all invocations of the application. If a required argument is not present, the CommandLineParser class will throw an exception during parsing.

Any argument can be made required. Usually it is recommended for any required argument to also be a positional argument, but this is not necessary.

For positional arguments, required arguments must always come before optional arguments; it is an error to define a required positional argument after an optional positional argument.

## Arguments with multiple values

Some arguments can take multiple values; these are *multi-value arguments*, also known as *array arguments*. These arguments can be supplied multiple times, and each value is added to the set of values. For example, consider the following command line arguments:

-ArgumentName Value1 –ArgumentName Value2 –ArgumentName Value3

In this case, the value of the argument named “ArgumentName” will be a list holding all three values.

The type of a multi-value argument must be an array type, for example string[] or int[] in C#, or any type implementing ICollection<T> when the argument is defined by a read-only property.

It’s possible to specify a separator for multi-value arguments using the MultiValueSeparatorAttribute attribute. This makes it possible to specify multiple values for the argument while the argument itself is specified only once. For example, if the separator is set to a comma, you can specify the values as follows:

-ArgumentName Value1,Value2,Value3

In this case, the value of the argument named “ArgumentName” will be a list with the three values “Value1”, “Value2” and “Value3”.

**Note:** if you specify a separator for a multi-value argument, it is *not* possible to have an argument value containing the separator. There is no way to escape the separator. Therefore, make sure you pick a separator that will never be used in the argument values, and be extra careful with culture-sensitive argument types (for example, if you use a comma as the separator for a multi-value argument of floating point numbers, cultures that use a comma as the decimal separator will not be able to specify values properly).

If an argument is not a multi-value argument, it is an error to supply it more than once, unless duplicate arguments are allowed in which case only the last value is used.

If a multi-value argument is positional, it must be the last positional argument. All remaining positional argument values will be considered values for the multi-value argument.

If a multi-value argument is required, it means it must have at least one value.

## Dictionary arguments

Dictionary arguments are multi-value arguments that specify a set of key/value pairs. Each value for a dictionary argument takes the form key=value, like in the following example:

-ArgumentName Key1=Value1 –ArgumentName Key2=Value2

In this case, the value of the argument named “ArgumentName” will be a dictionary with two keys, Key1 and Key2, with the associated values Value1 and Value2 respectively.

A dictionary argument must have a type of Dictionary<TKey, TValue> where TKey and TValue are the types of the key and value. When using a read-only property to define the argument, you can use any type that implements IDictionary<TKey, TValue>.

If you specify the same key more than once an exception will be thrown unless the AllowDuplicateDictionaryKeysAttribute attribute is specified on the constructor parameter or property that defines the dictionary argument.

## Switch arguments

A switch argument is an argument with a Boolean type (bool in C#). Its value is determined by its presence or absence on the command line; the value will be true if the argument is supplied, and false if not. The following arguments set the switch argument named “switch” to true:

-switch

A switch argument’s value can be specified explicitly, as in the following example:

-switch:true

You must use a colon to specify an explicit value for a switch argument; you cannot use white space to separate the name and the value.

If you use a nullable Boolean type (bool? In C#) as the type of the argument, it will be null if omitted, true if supplied, and false only if explicitly set to false using -switch:false.

If the type of the argument is an array of Boolean values, it will act as a multi-value argument and a switch. A value of true (or the explicit value if one is given) gets added to the array for every time that the argument is supplied.

## Argument value conversion

Ookii.CommandLine allows you to define arguments with any .Net type, including types such as System.String, System.Int32, System.DateTime, and many more. Any type can be used; the only requirement is that it is possible to convert a string value to that type.

The .Net Framework provides a very flexible method for converting one type to another through the System.ComponentModel.TypeConverter class. You can use any type that has a type converter than can convert from a string for a command line argument. Most built-in types in .Net Framework have such a type converter. You can also use your own types by creating a type converter for that type.

It is possible to override the default conversion by specifying a custom type converter using the System.ComponentModel.TypeConverterAttribute. When this attribute is applied to a constructor parameter or property that defines an argument, the specified type converter will be used for conversion instead. Note that for multi-value, dictionary and nullable value-type arguments the converter must be for the element type (e.g. if the argument is a multi-value argument of type int[], the type converter must be able to convert to int). For a dictionary argument the element type is KeyValuePair<TKey, TValue>, and the type converter is responsible for parsing the key and value from the argument value.

For many types, the conversion can be culture dependent. For example, converting numbers or dates depends on the culture which defines the accepted formats and how they’re interpreted; some cultures might use a period as the decimal separators, while others use a comma.

The culture used for argument value conversions is specified the CommandLineParser.Culture property, which defaults to the current culture. If you wish your argument parsing to be independent of the user’s culture, set this property to System.Globalization.CultureInfo.InvariantCulture.

# Defining command line arguments

In order to use Ookii.CommandLine, you must create a class that defines the arguments accepted by your application. This type will specify the names, types and attributes (required, positional) of each argument.

There are two ways to define arguments: using the properties of the class, or using constructor parameters for the class.

## Using properties

The preferred way to define arguments is by using properties. A property defines an argument only when it has the CommandLineArgumentAttribute attribute applied to it. The property must have a getter and setter, except for multi-value and dictionary arguments which can be defined by read-only properties under certain conditions.

The type of the argument is the type of the property, and the name of the argument matches the property name by default, but this can be overridden using the CommandLineArgumentAttribute constructor.

An argument defined by a property is by default optional and not positional. Its default value can be set using the CommandLineArgumentAttribute.DefaultValue property.

To create a required argument, set the CommandLineArgumentAttribute.IsRequired property to true.

To create a positional argument, set the CommandLineArgumentAttribute.Position property to a non-negative number. This property determines the relative ordering of the positional arguments only, not their actual position, so it’s fine if you skip numbers. Positional arguments defined by properties come after arguments defined by constructor parameters, so for example if there are three constructor parameters, the property with the lowest position value will be the fourth positional argument. Remember that you cannot have required positional arguments after optional ones, and that a multi-value positional argument must be the last positional argument. If your properties violate these rules, the CommandLineParser class’s constructor will throw an exception.

To define a multi-value argument, you can use either a read-write property of an array type (e.g. int[]) or a read-only property of any type implementing ICollection<T> (e.g. List<int>).

To define a dictionary argument, you can use either a read-write property of type Dictionary<TKey, TValue> (e.g. Dictionary<string, int>) or a read-only property of any type implementing IDictionary<TKey, TValue>.

If the type of an argument is a Boolean, a nullable Boolean, or an array of Booleans, this defines a switch argument unless the argument is positional.

Consider the following properties:

$lang: C#; CommandLineArgument$  
[CommandLineArgument(DefaultValue = "default")]  
public string SomeArgument { get; set; }  
  
[CommandLineArgument("OtherName", Position = 0)]  
public int OtherArgument { get; set; }  
  
[CommandLineArgument]  
public bool Switch { get; set; }  
  
public string NotAnArgument { get; set; }

The first property defines an optional non-positional argument of type string with the name “SomeArgument” and the default value “default”.

The second property defines an optional positional argument of type int with the name “OtherName”. The argument will be the first positional argument (after the arguments defined by constructor parameters, if there are any).

The third property defines a switch argument, because its type is a Boolean. Switch arguments can be supplied without a value; they will be true if present and false is not.

The fourth property does not define an argument, because it doesn’t have the CommandLineArgumentAttribute applied.

## Using constructor parameters

An alternative way to define positional parameters is using a constructor. The parameters of the public constructor for the class that defines the arguments will be used to define arguments. These arguments will be positional arguments, and required if the parameter is a required parameter.

Every constructor parameter creates a positional argument with its position matching the position of the constructor parameter. The type of the constructor parameter is the type of the argument, and by default the name of the constructor parameter is used of as the argument name, but this can be overridden using the ArgumentNameAttribute attribute.

For example, consider a class with the following constructor:

$lang: C#; ArgumentName$  
public MyArguments(string arg1,  
 int arg2,  
 [ArgumentName("CustomName")] float arg3 = 0f)  
{  
}

This constructor defines the following arguments: a required positional argument of type string with the name “arg1”, a required positional argument of type int with the name “arg3”, and an optional positional argument of type float with the name “CustomName” and a default value of 0.

If you are not using the C# 4.0 compiler, you can apply the System.Runtime.InteropServices.OptionalAttribute attribute and the System.Runtime.InteropServices.DefaultParameterValueAttribute to the parameter to define an optional argument. In Visual Basic, you can use the usual syntax for optional parameters (for an example, see the source code of the Visual Basic sample application included with the library).

If the type of a constructor parameter is an array, this defines a multi-value positional argument.

If your type has more than one constructor, you must mark one of them using the CommandLineConstructorAttribute attribute. You don’t need to use this attribute if you have only one constructor.

If you don’t wish to define arguments using the constructor, simply use a constructor without any parameters (or don’t define an explicit constructor).

## Defining aliases

An alias is an alternative name that can be used to specify a command line argument. Aliases can be added to a command line argument by applying the AliasAttribute to the property or constructor parameter that defines the argument.

For example, the following code defines a switch argument that can be specified using either the name “Verbose” or the alias “v”:

$lang: C#; CommandLineArgument Alias$  
[CommandLineArgument, Alias("v")]  
public bool Verbose { get; set; }

To specify more than one alias for an argument, simply apply the AliasAttribute multiple times.

# Parsing command line arguments

When you have defined the command line arguments, you can now parse the command line to determine the argument values.

In order to do this, you first must create an instance of the CommandLineParser class, passing the type of the class that defines the parameters to the constructor. Then call the CommandLineParser.Parse method, passing the arguments your application received.

If argument parsing is successful, the CommandLineParser.Parse method will create a new instance of the class defining the arguments, passing the values parsed from the command line to the constructor parameters (if any). It will then set the value of each property to the value of the corresponding argument. This is not done in any particular order, so do not write code that makes assumptions about this. Finally, it will return this instance.

If argument parsing fails, the CommandLineParser.Parse method will throw a CommandLineArgumentException exception. Argument parsing can fail because not all required arguments were specified, an argument name was supplied without a value for an argument that’s not a switch, an unknown argument name was supplied, a non-multi-value argument was specified more than once, too many positional argument values were supplied, or argument value conversion failed for one of the arguments. Check the CommandLineParser.Category property to determine the cause of the exception.

A typical usage sample for the CommandLineParser class is as follows:

$lang: C#; CommandLineParser MyArguments CommandLineArgumentException Console$  
static void Main(string[] args)  
{  
 CommandLineParser parser = new CommandLineParser(typeof(MyArguments));  
 try  
 {  
 MyArguments arguments = (MyArguments)parser.Parse(args);  
 RunApplication(arguments);  
 }  
 catch( CommandLineArgumentException ex )  
 {  
 Console.WriteLine(ex.Message);  
 parser.WriteUsageToConsole();  
 }  
}

This sample tries to parse the arguments, and if it fails prints the error message and usage help to the console.

The CommandLineParser class provides several ways to customize the parsing behavior; check the class library documentation for more details.

# Generating usage help

When you have an application that accepts command line arguments, it’s often useful to be able to provide the user with information about how to invoke the application, including an overview of all the arguments.

Creating this kind of usage help text is tedious, and you must make sure it is kept up to date whenever you change the arguments to your application. For this reason, Ookii.CommandLine can generate this usage help text automatically. The generated usage help can be customized by adding descriptions to the arguments.

Usage help can be generated using the CommandLineParser.WriteUsage method. The output can be customized using the WriteUsageOptions class. The CommandLineParser.WriteUsageToConsole method provides a convenient way to write the usage help to the standard output stream, properly word-wrapping the text at the console width.

The following example shows the usage help generated for the sample application included with the Ookii.CommandLine library:

Sample command line application. The application parses the command  
line and prints the results, but otherwise does nothing, and none  
of the arguments are actually used for anything.  
  
Usage: CommandLineSampleCS.exe [-source] <String>   
 [-destination] <String> [[-index] <Number>] [[-id] <String>] [-?]  
 [-Count <Number>] [-Date <DateTime>] [-v] [-val <String>...]  
  
 -source <String>  
 The source data.  
  
 -destination <String>  
 The destination data.  
  
 -index <Number>  
 The operation's index. This argument is optional, and the  
 default value is 1.  
  
 -id <String>  
 Sets the operation ID. The default value is "default".  
  
 -? [<Boolean>]  
 Displays this help message.  
  
 -Count <Number>  
 Provides the count for something to the application. This  
 argument is required.  
  
 -Date <DateTime>  
 Provides a date to the application; the format to use  
 depends on your regional settings.  
  
 -v [<Boolean>]  
 Print verbose information; this is an example of a switch  
 argument.  
 -val <String>  
 This is an example of an array argument, which can be  
 repeated multiple times to set more than one value.

The usage help consists of three components: the application description, the argument syntax, and the argument descriptions.

## Application description

The first part of the usage help is a description of your application. This is a short description that explains what your application does and how it can be used. It can be any text you like, though it’s recommended to keep it short.

The description is specified by specifying the System.ComponentModel.DescriptionAttribute to the class that defines the command line arguments, as in the following example:

$lang: C#; Description MyArguments$  
[Description("This is the application description that is included in the usage help.")]  
class MyArguments  
{  
}

If this attribute is not specified, no description is included in the usage help. The description can also be omitted by setting the WriteUsageOptions.IncludeApplicationDescription property to false.

## Argument syntax

The argument syntax indicates how your application can be invoked from the command line. The argument syntax typically starts with the name of your application, and is followed by all the arguments, indicating their name and type. There is an indication of which arguments are required or optional, and whether they allow multiple values. For positional arguments, the order is indicated as well.

The syntax for a single argument has the following format:

-ArgumentName <ValueDescription>

For optional arguments, the name and value description are enclosed by square brackets. For a positional argument, the name is enclosed by square brackets to indicate the name itself is optional. For an array argument, the value description is followed by three periods.

The value description of an argument is short description (typically one word) that describes what kind of value the argument expects. It default to the type of the argument (for array arguments, the element type is used; for nullable types, the underlying type is used).

The value description can be specified explicitly. For example, you may want to set the value description of a numeric argument to “Number” rather than “Int32”. For arguments defined using constructor parameters, use the ValueDescriptionAttribute attribute. For arguments defined by a property, use the CommandLineArgumentAttribute.ValueDescription property.

The value description is omitted for switch arguments.

The exact format of the argument syntax can be customized using the WriteUsageOptions class. You can specify the usage prefix, and various format strings that control how optional arguments, value descriptions, and multi-value arguments are displayed.

## Argument descriptions

The final part of the usage help is a description for all the arguments. A list is written to the output of all arguments, followed by their description.

The description of an argument can be specified using the System.ComponentModel.DescriptionAttribute attribute. Apply this attribute to the constructor parameter or property defining the argument.

The exact format of the argument descriptions can be customized using the WriteUsageOptions class. You can specify a format string that controls how the argument name and description are laid out, and the amount of indentation to use for additional lines of the description.

By default, the default value and aliases of an argument are not included in the argument syntax or description. If you wish to advertise these to the user, you can add them in the argument’s description itself, or set the WriteUsageOptions.IncludeDefaultValueInDescription and WriteUsageOptions.IncludeAliasInDescription properties to true.

# Shell commands

Ookii.CommandLine provides the ability to create an application that has more than one function by using shell commands.

Shell commands can be used to create shell utilities that perform more than one operation, where each operation has its own set of command line arguments. For example, a utility might be used to modify or query different configuration parameters of a system. Depending on whether it's a query or a modification, and which configuration parameter is used, the arguments to such a utility might differ. Rather than provide different executables for each operation, it is often more convenient to combine related operations in a single utility.

Think for example of the Windows net command, which provides many different operations through the same command. Shell commands allow you to easily create such an application.

For a program using shell commands, typically the first command line argument will be the name of the operation and identifies which shell command to use, while the remaining arguments are arguments to the command. The ShellCommand class aids you in creating utilities that follow this pattern.

## Defining shell commands

A shell command is created by deriving a type from the ShellCommand class, specifying the ShellCommandAttribute on that type to specify the name of the command, and implementing the ShellCommand.Run method for that type.

The class inheriting the ShellCommand class defines the arguments for the command. This class will be used with the CommandLineParser class to parse the arguments for the command.

## Using shell commands

To use a shell command, you must first determine the shell command the user wishes to invoke, typically by inspecting the first element of the array of arguments passed to the Main method of your application.

You can then get the Type instance of the shell command’s class by calling the ShellCommand.GetShellCommand method. This method searches the specified assembly for a type that inherits from the ShellCommand class, and has the ShellCommandAttribute attribute applied with the ShellCommandAttribute.Name property set to the specified name. You can also get a list of all shell commands in an assembly by using the ShellCommand.GetShellCommands method.

This Type instance can be passed to the constructor of the CommandLineParser class, after which you can parse arguments for the command as usual (make sure to pass an index so that the command name is not treated as an argument), and finally invoke its ShellCommand.Run method.

The ShellCommand class provides static utility methods that perform these tasks for you. The ShellCommand.CreateShellCommand method finds and creates a shell command, and writes error and usage information to the output if it failed. If no command name was specified, or the specified command name could not be found, it writes a list of all shell commands in the assembly and their descriptions to the output. If the command was found but parsing its arguments failed, it writes usage information for that command to the output.

The ShellCommand.RunShellCommand method works the same as the ShellCommand.CreateShellCommand method, but also invokes the ShellCommand.Run method if the command was successfully created.

It is recommended to return the value of the ShellCommand.ExitCode property to the operating system (by returning it from the Main method or by using the Environment.ExitCode property) after running the shell command.

The source code of a full sample application that defines two commands is included with the Ookii.CommandLine library.

# Line wrapping text writer

The Ookii.CommandLine library includes a utility class, the LineWrappingTextWriter class, which it uses for properly word-wrapping usage help when writing it to the console, but which can also be used in your own code.

The LineWrappingTextWriter class allows you to write text to another TextWriter class, wrapping that text at the specified line length, and optionally indenting subsequent lines.