NConsole Menu System

Documentation Guide

Last Updated: 7-13-2022
By: Mark Alicz – mark@ivolt.io - https://github.com/InspiredVoltage-IVolt

Table of Contents

Advantages

- Simple, short syntax
  - Simple commands are created in a single line
  - Complex commands can be created or modified in steps or in their own, short class
- Powerful and extensible
  - Inner commands effortlessly parse parameters to commands
  - Nested menus allow fully or partially independent submenus
  - Own commands as well as subclassing existing commands are fully supported
- Low maintenance
  - Adaptive, automatically configures itself for optimal usage. For instance, command abbreviations are created and updated automatically when new commands (with potentially similar name) are added or removed.
  - Very tolerant about its input, allowing partial matches and being (by default) case insensitive.
- Integrated tutorial and examples
  - Contains a library of example commands, including a conditional operator, a macro recorder, and a procedural call structure
  - Comprehensive, documented code in case you want to dig into the details
- Result: An easy-to-use menu structure
  - Effortless command detection, including automatic abbreviations and partial matching
  - Intelligent corrections for mistyped commands
  - Integrated useful help which also works in command tree branches

Basics

Creating and using basic commands

A single command is comprised of a keyword (selector), an optional help text describing it, and, most importantly, its behavior. The behavior can be defined as a simple lambda, and it is not unusual for a whole command to be defined in a single line.

```
// Create menu
var menu = new CMenu ();
// Add simple Hello World command
menu.Add ("hello", s => Console.WriteLine ("Hello world!"));
// Run menu. The menu will run until quit by the user.
menu.Run ();
```

While running, NConsoleMenu will continuously prompt the user for input, then feeds it to the respective command. Let's see how to use the "hello" command defined above:

```
$ hello
Hello world!
```

If the command happens to be more complex, you can just put it in a separate method.

```
menu.Add ("len", s => PrintLen (s));
static void PrintLen (string s)
{
        Console.WriteLine ("String \"" + s + "\" has length " + s.Length);
}

$ len 54321
String "54321" has length 5
```

It is also possible to return an exit code to signal that processing should be stopped.
By default, the command "quit" exists for this purpose. Let's add an alternative way to stop processing input.

```
menu.Add ("exit", s => menu.Quit ());
```

To create a command with help text, simply add it during definition.

```
menu.Add ("time",
        s => Console.WriteLine (DateTime.UtcNow),
        "Help for \"time\": Writes the current time");

$ time
2015.10.01 17:54:38
$ help time
Help for "time": Writes the current time
```

You can also access individual commands to edit them later, though this is rarely required.

```
menu["time"].HelpText += " (UTC).";

$ help time
Help for "time": Writes the current time (UTC).
```

Command abbreviations and integrated help

A menu keeps an index of all available commands and lists them upon user request via typing "help". Moreover, it also automatically assigns abbreviations to all commands (if useful) and keeps them up to date when you later add new commands with similar keywords.

```
$ help
Available commands:
```

```
e   | exit
      hello
      help
l   | len
q   | quit
t   | time
Type "help <command>" for individual command help.
```

The builtin command "help" also displays usage information of individual commands:

```
$ help q
quit
Quits menu processing.
$ help help
help [command]
Displays a help text for the specified command, or
Displays a list of all available commands.
```

Commands can by entered abbreviated if it is clear which one was intended. If it is not clear, then the possible options will be displayed.

```
$ hel
Command <hel> not unique. Candidates: hello, help
$ hell
Hello world!
```

Commands are case *in*sensitive by default. This can be changed using the `StringComparison` property.
When in case sensitive mode, NConsoleMenu will helpfully point out similar commands with different caseing.

```
menu.StringComparison = StringComparison.InvariantCulture;
menu.Add ("Hello", s => Console.WriteLine ("Hi!"));
```

```
$ help
Available commands:
e   | exit
H   | Hello
      hello
      help
l   | len
q   | quit
r   | repeat
t   | time
Type "help <command>" for individual command help.
$ H
Hi!
$ h
Command <h> not unique. Candidates: hello, help
$ hE
Unknown command: hE
Did you mean "hello", "Hello" or "help"?
```

Conditional Commands

Commands which cannot currently be used, but should still be available in the menu tree at other times, can disable themselves. Disabled commands cannot be used and are not listed by `help`.

In this example, a global flag (`bool myEnabledProperty`) is used to determine if a command is enabled. It is initially cleared, the `enable` command sets it.

```
m.Add ("enable", s => myEnabledProperty = true);
```

Create a new inline command, then set its enabledness function so it returns the above flag.

```
var mi = m.Add ("inline", s => Console.WriteLine ("Disabled inline command was enabled!"));
mi.SetEnablednessCondition (() => myEnabledProperty);
```

```
$ inline
Unknown command: inline
$ enable
$ inline
Disabled inline command was enabled!
```

It is also possible to override the enabledness by subclassing.

```
private class DisabledItem : CMenuItem
{
        public DisabledItem ()
                : base ("subclassed")
```

```
                {
                        HelpText = "This command, which is defined in its own class, is disabled by default.";
                }

                public override bool IsEnabled ()
                {
                        return myEnabledProperty;
                }

                public override void Execute (string arg)
                {
                        Console.WriteLine ("Disabled subclassed command was enabled!");
                }
        }

        $ subclassed
        Unknown command: subclassed
        $ enable
        $ subclassed
        Disabled subclassed command was enabled!
```

Disabled commands are not displayed by `help`.

```
        $ help
        Available commands:
        e   | enable
        h   | help
        q   | quit
        $ enable
        $ help
        Available commands:
        e   | enable
        h   | help
        i   | inline
        q   | quit
        s   | subclassed
```

Command abbreviations do not change when disabled items become enabled, i.e. it is made sure they are already long enough to be unique. This avoids confusion about abbreviations suddenly changing.

```
        m.Add ("incollision", s => Console.WriteLine ("The abbreviation of 'incollision' is longer to account for the hidden 'inline' command."));

        $ help
        Available commands:
        e   | enable
        h   | help
        inc | incollision
        q   | quit
        $ enable
        $ help
        Available commands:
        e   | enable
        h   | help
        inc | incollision
        inl | inline
        q   | quit
        s   | subclassed
```

Example Project Documentation

The source code contains an example project. It offers commands, which illustrate several (more or less advanced) use cases. It may be useful to reference them in your own projects.

echo

Simply prints text to the console. This is probably most useful in batch processing.

Help text

```
        echo [text]
        Prints the specified text to stdout.
```

Example

```
$ echo 123
123
```

if

Simple conditional execution. By default only supports the `not` operator and the constants `true` and `false`, but can be extended with arbitrary additional conditions.

Condition combination is not currently supported, though it can in part be emulated via chaining ("if <c1> if <c2> ...")
It is allowed to specify multiple concurrent `not`, each of which invert the condition again.

Help text

```
if [not] <condition> <command>
Executes <command> if <condition> is met.
If the modifier <not> is given, the condition result is reversed.
```

Example

```
$ if true echo 1
1
$ if not true echo 1
$ if not false echo 1
1
```

pause

pause Stops further operation until the enter key is pressed.

record, replay

`record` and `replay` allow persisting several commands to disk for later reading them as input. This can be used for basic batch processing.

Replaying can be stopped via if `endreplay` is encountered as a direct command in the file. It does not work as an indirect statement (e.g. `if true endreplay`). For an example of how that functionality can be achieved, see `return`.

record

```
record name
Records all subsequent commands to the specified file name.
Recording can be stopped by the command endrecord
Stored records can be played via the "replay" command.
```

replay

```
replay [name]
Replays all commands stored in the specified file name, or
Displays a list of all records.

Replaying puts all stored commands in the same order on the stack as they were originally entered.
Replaying stops when the line "endreplay" is encountered.
```

Example

```
$ record r1
Recording started. Enter "endrecord" to finish.
record> echo 1
record> echo 2
record> endrecord
$ replay r1
1
2
```

proc, call, return, goto

These implement a basic procedural calling system. Early exiting, jumping within the local procedure and reentrant calls are supported.

Example

```
$ proc p1
    Recording started. Enter "endproc" to finish.
    proc> echo In proc p1
    proc> return
    proc> echo This line will never be displayed.
    proc> endproc
    $ call p1
    In proc p1

    $ proc p2
    Recording started. Enter "endproc" to finish.
    proc> echo 1 - entered p2
    proc> goto g
    proc> echo 2 - this line will not be displayed.
    proc> :g
    proc> echo 3 - p2 completed
    proc> endproc
    $ call p2
    1 - entered p2
    3 - p2 completed
```

Input

Immediate mode

By default, the user has to type in commands (and possibly their arguments). Sometimes though, only selection of options is required, similar to a classic 'menu'.

For this purpose, set the `ImmediateMode` flag in your menu. When run, it will display all available options by default (no need to type `help`). Each option will be preceded by a selection number, and simply entering that number will activate the corresponding menu item - no need to even press the Enter key.

```
    var m = new CMenu ();
    m.ImmediateMenuMode = true;
    m.Add ("foo", s => Console.WriteLine ("foo"));
    m.Add ("bar", s => Console.WriteLine ("bar"));
    m.Run ();

     1 quit
     2 help
     3 foo
     4 bar
    [presses 3 key]
    foo
     1 quit
     2 help
     3 foo
     4 bar
```

Modifying the input queue

It is also possible to modify the input queue. The `IO` class provides flexible means to add input either directly or via an `IEnumerable<string>`. The latter allows you to stay in control over the input even after you added it, for instance by changing its content or canceling it.

Check out how the "repeat" command adds its argument to the input queue two times.

```
    // Add a command which repeats another command
    menu.Add ("repeat",
            s => {
                    IO.ImmediateInput (s);
                    IO.ImmediateInput (s);
            },
            "Repeats a command two times.");

    $ repeat hello
    Hello world!
    Hello world!
    $ r l 123
    String "123" has length 3
    String "123" has length 3
```

Passive mode

By default, input is handled in prompting mode, i.e. a menu will actively prompt the user for required input and read their console input.

This behavior may be undesirable if you want closer control, for instance:

* In a GUI environment, creating input in the GUI instead of the console as usual
* In a batch or shell environment, feeding stored input instead of prompting the user for it

To suppress active prompting, enable promptless mode by clearing the `PromptUserForInput` flag. The menu will then wait for programmatic input, e.g. via `CQ.AddInput`.

```
IO is currently in active mode - you will be prompted for input.
The 'promptless' command will turn on promptless mode, which disables interactive input.
The 'active' command will turn active mode back on.
Please enter 'promptless' (or 'p').

$ p
Promptless mode selected. Input will be ignored.
A timer will be set which will input 'active' in 5 seconds.
5...
4...
3...
2...
1...
0...
Sending input 'active' to the command queue.
Prompting mode selected again.
$
```

Side note: Switching from prompting to promptless mode during a prompting input query (i.e. while the user is being prompted for input) will cause the prompt to still wait for input after switching. This is due to limitations in the underlying system.

Nesting

Inner commands

If a command needs further choices, you may want to select those in a similar manner as in the menu. To do that, simply add sub-items to the main item. If no other behavior is specified, the main item will continue selection within those embedded items.

```
var mi = menu.Add ("convert", "convert upper|lower [text]\nConverts the text to upper or lower case");
mi.Add ("upper", s => Console.WriteLine (s.ToUpperInvariant ()), "Converts to upper case");
mi.Add ("lower", s => Console.WriteLine (s.ToLowerInvariant ()), "Converts to lower case");

$ convert upper aBcD
ABCD
$ convert lower aBcD
abcd
```

The integrated help is able to "peek" into commands.

```
$ help convert
convert upper|lower [text]
Converts the text to upper or lower case
$ help c u
Converts to upper case
```

If only a single inner item exists, users do not have to type it. Automatic fall-through forwards to the inner item directly.

```
var mi = menu.Add ("fall");
mi.Add ("through", s => Console.WriteLine ("Fell through to the innermost item."));

$ fall through
Fell through to the innermost item.
$ fall t
Fell through to the innermost item.
$ fall
Fell through to the innermost item.
$ fall
Fell through to the innermost item.
$ fall xy
Unknown command: xy
```

Nested menus and default commands

Usually, all commands are processed by the same CMenu, which itself consists of several CMenuItems each responsible for a different command type. However, sometimes a command opens up a new submenu, often with commands different from its parent menu.

To achieve this functionality, just append a new child CMenu to its parent. When the user enters the submenu, all typing will be routed through it instead of the parent menu. The user can at any time return to the parent menu by quitting the child menu, for instace by providing a "quit" command.

Embedding a CMenu instead of a CMenuItem is advantageous if you do not want to keep track of state manually, possible including a menu stack (which parent menu to return to when a submenu is quit). You may opt for a custom prompt character to distinguish the submenu from its parent menu.

Especially in this context, it may be useful to capture all input which lacks a corresponding command in a "default" command. The default command has the unique selector null.

Let's see an example for a command which calculactes the sum of integers entered by the user. The sum is calculated and output once the user enters "=", which will be implemented as a subcommand. Capturing the integers is done with a default command. To clarify to the user that this is a different menu, we also replace the prompt character with a "+".

```
public class MI_Add : CMenu
{
        public MI_Add ()
                : base ("add")
        {
                HelpText = ""
                        + "add\n"
                        + "Adds numbers until \"=\" is entered.";
                PromptCharacter = "+";

                Add ("=", s => Quit (), "Prints the sum and quits the add submenu");
                Add (null, s => Add (s));
        }

        private int _Sum = 0;

        private void Add (string s)
        {
                int i;
                if (int.TryParse (s, out i)) {
                        _Sum += i;
                }
                else {
                        Console.WriteLine (s + " is not a valid number.");
                }
        }

        public override void Execute (string arg)
        {
                Console.WriteLine ("You're now in submenu <Add>.");
                Console.WriteLine ("Enter numbers. To print their sum and exit the submenu, enter \"=\".");
                _Sum = 0;
                Run ();
                Console.WriteLine ("Sum = " + _Sum);
        }
}

$ add
You're now in submenu <Add>.
Enter numbers. To print their sum and exit the submenu, enter "=".
+ 2
+ 3
+ =
Sum = 5
```

Sharing code between nested items

If your inner menu items should share code, you need to overwrite the menu's Execute method, then call ExecuteChild to resume processing in child nodes.

This allows you to alter the command received by the children, or to omit their processing altogether (e.g. in case a common verification failed).

```
var m = menu.Add ("shared");
m.SetAction (s => {
        Console.Write ("You picked: ");
        m.ExecuteChild (s);
```

```
});
m.Add ("1", s => Console.WriteLine ("Option 1"));
m.Add ("2", s => Console.WriteLine ("Option 2"));

$ shared 1
You picked: Option 1
```

Initialization syntax for menu trees

It may be useful to create complex menu trees using collection initializers.

```
var m = new CMenu () {
        new CMenuItem ("1") {
                new CMenuItem ("1", s => Console.WriteLine ("1-1")),
                new CMenuItem ("2", s => Console.WriteLine ("1-2")),
        },
        new CMenuItem ("2") {
                new CMenuItem ("1", s => Console.WriteLine ("2-1")),
                new CMenuItem ("2", s => Console.WriteLine ("2-2")),
        },
};

$2 1
2-1
```

You can also combine object and collection initializers

```
m = new CMenu () {
        PromptCharacter = "combined>",
        MenuItem = {
                new CMenuItem ("1", s => Console.WriteLine ("1")),
                new CMenuItem ("2", s => Console.WriteLine ("2")),
        }
};
```

Output

Any output a menu item writes is, by default, printed to the console. Output hooks allow intercepting this output. This can be useful to change of format it before printing, or to redirect or clone it to other locations (e.g. a logfile).

There are five kinds of output events that can be intercepted. They all refer to the equivalently named method or property of the `Console` class.

* Write
* WriteLine
* SetForegroundColor
* SetBackgroundColor
* ResetColor

It is recommended to let all custom menu items opt in to this pattern. For this purpose, simply call `OnWrite` instead of `Console.Write`, etc.

If no hook is present in a particular menu item, the call is redirected to its parent item. If no hook is found while moving up the chain, the regular method of the `Console` class is invoked. If a hook is found in a child item, the parent's hooks are not invoked.

Hooks can be added or removed at any time and for any menu item individually.

Example

This class demonstrates using output hooks. It intercepts `Console.Write` commands and prints all lower-case letters with a red background. It would just as well be possible to copy the printed text to a log file.

```
private class OutputHook : CMenuItem
{
        public OutputHook () : base ("outputhook")
        {
                HelpText = "Demonstrates using output hooks.";
                Add ("add", s => Parent.Write += HookedWrite);
                Add ("remove", s => Parent.Write -= HookedWrite);
        }

        private static void HookedWrite (string s)
        {
                var fc = Console.ForegroundColor;
```

```
                var bc = Console.BackgroundColor;

                foreach (var c in s) {
                        if (!char.IsLower (c)) {
                                Console.Write (c);
                        }
                        else {
                                Console.ForegroundColor = ConsoleColor.Black;
                                Console.BackgroundColor = ConsoleColor.Red;
                                Console.Write (c);
                                Console.ForegroundColor = fc;
                                Console.BackgroundColor = bc;
                        }
                }
        }
}
```

APPENDIX

The appendix contains code that was copied from the example project include with the download. Please check that directory for a full working example.

Example Code – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using NConsoleMenu;
using NConsoleMenu.Sample.Examples;

namespace NConsoleMenu.Sample
{
	/// <summary>
	/// Test program for CMenu.
	/// </summary>
	class Program
	{
		static void Main (string[] args)
		{
			Console.WriteLine ("Simple CMenu demonstration");

			var mainmenu = new CMenu ();
			mainmenu.PromptCharacter = "main>";
			mainmenu.Add ("tutorial", s => new Tutorial ().Run ());
			mainmenu.Add ("tree-init", s => TreeInitialization ());
			mainmenu.Add ("disabled", s => DisabledCommands ());
			mainmenu.Add ("promptless", s => PromptlessMode ());
			mainmenu.Add ("immediate", s => ImmediateMode ());
			mainmenu.Add (new ExamplesMenu ());
			mainmenu.Add (new OutputHook ());

			mainmenu.CQ.ImmediateInput ("help");
			mainmenu.Run ();
		}

		static void TreeInitialization ()
		{
			/*
			 * It may be useful to create complex menu trees using collection initializers
			 */
			var m = new CMenu () {
				new CMenuItem ("1") {
					new CMenuItem ("1", s => Console.WriteLine ("1-1")),
					new CMenuItem ("2", s => Console.WriteLine ("1-2")),
				},
				new CMenuItem ("2") {
					new CMenuItem ("1", s => Console.WriteLine ("2-1")),
					new CMenuItem ("2", s => Console.WriteLine ("2-2")),
				},
			};
			m.PromptCharacter = "tree>";
			m.Run ();

			/*
			 * You can also combine object and collection initializers
			 */
			m = new CMenu () {
				PromptCharacter = "combined>",
				MenuItem = {
					new CMenuItem ("1", s => Console.WriteLine ("1")),
					new CMenuItem ("2", s => Console.WriteLine ("2")),
				}
			};
			m.Run ();
		}

		static bool myEnabledProperty = false;

		/*
		 * Commands which cannot currently be used, but should still be available in the
		 * menu tree at other times, can disable themselves. Disabled commands cannot be
		 * used and are not listed by `help`.
		 */
		static void DisabledCommands ()
		{
			var m = new CMenu ();

			/*
			 * In this example, a global flag (`bool myEnabledProperty`) is used to
			 * determine the visibility of disabled commands. It is initially cleared,
```

```
                                * the `enable` command sets it.
                                */
                               myEnabledProperty = false;
                               m.Add ("enable", s => myEnabledProperty = true);

                               /*
                                * Create a new inline command, then set its enabledness function so it
                                * returns the above flag.
                                */
                               var mi = m.Add ("inline", s => Console.WriteLine ("Disabled inline command was enabled!"));
                               mi.SetEnablednessCondition (() => myEnabledProperty);

                               /*
                                * It is also possible to override the visibility by subclassing.
                                */
                               m.Add (new DisabledItem ());
                               m.Run ();

                               /*
                                * Disabled commands are not displayed by `help`.
                                */

                               /*
                                * Command abbreviations do not change when hidden items become visible,
                                * i.e. it is made sure they are already long enough. This avoids confusion
                                * about abbreviations suddenly changing.
                                */
                               m.Add ("incollision", s => Console.WriteLine ("The abbreviation of 'incollision' is longer to account
for the hidden 'inline' command."));
                    }

                    private class DisabledItem : CMenuItem
                    {
                               public DisabledItem ()
                                          : base ("subclassed")
                               {
                                          HelpText = "This command, which is defined in its own class, is disabled by default.";
                               }

                               public override bool IsEnabled ()
                               {
                                          return myEnabledProperty;
                               }

                               public override void Execute (string arg)
                               {
                                          Console.WriteLine ("Disabled subclassed command was enabled!");
                               }
                    }

                    static void PromptlessMode ()
                    {
                               var m = new CMenu ();
                               m.Add ("promptless", s => {
                                          m.CQ.PromptUserForInput = false;
                                          Console.WriteLine ("Promptless mode selected. Input will be ignored.");
                                          Console.WriteLine ("A timer will be set which will input 'active' in 5 seconds.");
                                          new Thread (() => {
                                                     for (int i = 5; i >= 0; i--) {
                                                                Console.WriteLine (i + "...");
                                                                Thread.Sleep (1000);
                                                     }
                                                     Console.WriteLine ("Sending input 'active' to the command queue.");
                                                     m.CQ.ImmediateInput ("active");
                                          }).Start ();
                               });
                               m.Add ("active", s => {
                                          m.CQ.PromptUserForInput = true;
                                          Console.WriteLine ("Prompting mode selected again.");
                               });

                               Console.WriteLine ("IO is currently in active mode - you will be prompted for input.");
                               Console.WriteLine ("The 'promptless' command will turn on promptless mode, which disables interactive
input.");
                               Console.WriteLine ("The 'active' command will turn active mode back on.");
                               Console.WriteLine ("Please enter 'promptless' (or 'p').");

                               m.Run ();
                    }

                    static void ImmediateMode ()
```

```
		{
			var m = new CMenu ();
			m.ImmediateMenuMode = true;
			m.Add ("foo", s => Console.WriteLine ("foo"));
			m.Add ("bar", s => Console.WriteLine ("bar"));
			m.Run ();
		}

		/// <summary>
		/// <para>
		/// This class demonstrates using output hooks. It intercepts Console.Write
		/// commands and prints all lower-case letters with a red background. It would just
		/// as well be possible to copy the printed text to a log file.
		///
		/// Any output a menu item writes is, by default, printed to the console. Output
		/// hooks allow intercepting this output. This can be useful to change or redirect
		/// it before printing, or to redirect or clone it to other locations (e.g. a
		/// logfile).
		///
		/// There are five kinds of output events that can be intercepted. They all refer
		/// to the equivalently named method or property of the Console class.
		/// <list type="bullet">
		/// <item>Write</item>
		/// <item>WriteLine</item>
		/// <item>SetForegroundColor</item>
		/// <item>SetBackgroundColor</item>
		/// <item>ResetColor</item>
		/// </list>
		/// </para>
		/// <para>
		/// It is recommended to let all custom menu items opt in to this pattern. For this
		/// purpose, simply call OnWrite instead of Console.Write, etc.
		///
		/// If no hook is present in a particular menu item, the call is redirected to its
		/// parent item. If no hook is found while moving up the chain, the regular method
		/// of the Console class is invoked. If a hook is found in a child item, the
		/// parent's hooks are not invoked.
		///
		/// Hooks can be added or removed at any time and for any menu item individually.
		/// </para>
		/// </summary>
		private class OutputHook : CMenuItem
		{
			public OutputHook () : base ("outputhook")
			{
				HelpText = "Demonstrates using output hooks.";

				Add ("add", s => Parent.Write += HookedWrite);
				Add ("remove", s => Parent.Write -= HookedWrite);
			}

			private static void HookedWrite (string s)
			{
				var fc = Console.ForegroundColor;
				var bc = Console.BackgroundColor;

				foreach (var c in s) {
					if (!char.IsLower (c)) {
						Console.Write (c);
					}
					else {
						Console.ForegroundColor = ConsoleColor.Black;
						Console.BackgroundColor = ConsoleColor.Red;
						Console.Write (c);
						Console.ForegroundColor = fc;
						Console.BackgroundColor = bc;
					}
				}
			}
		}
	}
}
```

Example Code - Tutorial.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using NConsoleMenu;
using NConsoleMenu.Sample.Examples;

namespace NConsoleMenu.Sample
{
	class Tutorial
	{
		private CMenu menu;

		public void Run ()
		{
			Basics ();
			CaseSensitivity ();
			InputModification ();
			InnerCommands ();
			InnerCommandFallThrough ();
			NestedCommands ();
			SharingInInners ();
			menu.CQ.ImmediateInput ("help");
		}

		private void Basics ()
		{
			// Create menu
			menu = new CMenu ();

			// Let's specify a custom prompt so we later clearly know we're in the tutorial menu.
			menu.PromptCharacter = "tutorial>";

			// Add simple Hello World command
			menu.Add ("hello", s => Console.WriteLine ("Hello world!"));

			/*
			 * If the command happens to be more complex, you can just put it in a separate method.
			 */
			menu.Add ("len", s => PrintLen (s));

			/*
			 * It is also possible to return an exit code to signal that processing should be stopped.
			 * By default, the command "quit" exists for this purpose. Let's add an alternative way to stop
processing input.
			 */
			menu.Add ("exit", s => menu.Quit ());

			/*
			 * To create a command with help text, simply add it during definition.
			 */
			menu.Add ("time",
				s => Console.WriteLine (DateTime.UtcNow),
				"Help for \"time\": Writes the current time");

			/*
			 * You can also access individual commands to edit them later, though this is rarely required.
			 */
			menu["time"].HelpText += " (UTC).";

			// Run menu. The menu will run until quit by the user.
			Console.WriteLine ("Enter \"help\" for help.");
			Console.WriteLine ("Enter \"quit\" to quit (in this case, the next step of this demo will be
started).");
			menu.Run ();

			Console.WriteLine ("(First menu example completed, starting the next one...)");
		}

		private void PrintLen (string s)
		{
			Console.WriteLine ("String \"" + s + "\" has length " + s.Length);
		}

		private void CaseSensitivity ()
		{
			/*
```

```
                         * Commands are case *in*sensitive by default. This can be changed using the `StringComparison`
property.
                         */
                        menu.StringComparison = StringComparison.InvariantCulture;
                        menu.Add ("Hello", s => Console.WriteLine ("Hi!"));

                        Console.WriteLine ("The menu is now case sensitive.");
                        menu.Run ();
                }

                private void InputModification ()
                {
                        /*
                         * It is also possible to modify the input queue.
                         * Check out how the "repeat" command adds its argument to the input queue two times.
                         */
                        menu.Add ("repeat",
                                s => {
                                        menu.CQ.ImmediateInput (s);
                                        menu.CQ.ImmediateInput (s);
                                },
                                "Repeats a command two times.");

                        Console.WriteLine ("New command available: repeat");
                        menu.Run ();
                }

                private void InnerCommands ()
                {
                        var mi = menu.Add ("convert", "convert upper|lower [text]\nConverts the text to upper or lower case");
                        mi.Add ("upper", s => Console.WriteLine (s.ToUpperInvariant ()), "Converts to upper case");
                        mi.Add ("lower", s => Console.WriteLine (s.ToLowerInvariant ()), "Converts to lower case");

                        Console.WriteLine ("New command <convert> available. It features the inner commands \"upper\" and
\"lower\".");
                        menu.Run ();
                }

                private void InnerCommandFallThrough ()
                {
                        var mi = menu.Add ("fall");
                        mi.Add ("through", s => Console.WriteLine ("Fell through to the innermost item."));

                        Console.WriteLine ("The new inner command 'fall' contains only a single inner item 'through'.");
                        Console.WriteLine ("Any of the following will directly invoke 'through':");
                        Console.WriteLine ("'fall through', 'fall t', 'fall ', 'fall'");
                        menu.Run ();
                }

                private void NestedCommands ()
                {
                        menu.Add (new MI_Add ());

                        Console.WriteLine ("New command <add> available.");
                        menu.CQ.ImmediateInput ("help add");
                        menu.Run ();
                }

                private void SharingInInners ()
                {
                        /*
                         * If your inner menu items should share code, you need to overwrite the menu's Execute
                         * method, then call ExecuteChild to resume processing in child nodes.
                         *
                         * This allows you to alter the command received by the children, or to omit their
                         * processing altogether (e.g. in case a common verification failed).
                         */
                        var m = menu.Add ("shared");
                        m.SetAction (s => {
                                Console.Write ("You picked: ");
                                m.ExecuteChild (s);
                        });
                        m.Add ("1", s => Console.WriteLine ("Option 1"));
                        m.Add ("2", s => Console.WriteLine ("Option 2"));

                        Console.WriteLine ("New command <shared> available.");
                        menu.Run ();
                }
        }
}
```

Additional Examples

More examples can be found in the actual folder for this Example Menu. Look in the src\ExampleMenu directory of the downloaded source code, which can be found at the URL on page 1.