Overview

Creating console application is an easy task, but it is not very easy on implementing a console application that must handle various forms of command line syntax. You have no choice but parsing the syntax and getting the option values from the command line arguments. Furthermore, you have to write your help message for each option that is acceptable by the console application.

If your console application becomes more complex and you are tired of dealing with various forms of the command line syntax, maybe Adaptive Console Framework (ACF) can help you. With ACF, you can:

- Separate your console application implementation from the definition of the command line syntax (Known as the Option Contracts)
- Simply define your command line options within the syntax (Known as the Option)
- Generate help screen automatically. Once you have defined the descriptions to your contracts and the options, help screen will be automatically generated
- If you are not satisfied with the generated information, you can make your customization by using ACF

Let's have a quick start on how to use ACF with our console applications[1].

Quick Start

The easiest way of introducing the new matter is to give a quick start tutorial. Here we assume that we are going to create a console application called **catool** which has the following command line syntax:

- catool.exe
- catool.exe /help | /?
- catool.exe </m | /method:<add | sub | mul | div>> <number1> <number2> [/nologo]

This **catool** application performs add, subtract, multiply or division on two given numbers. If no command line argument is provided, the application will print the help text on the screen and exit. If /help or /? is provided, also the help text will be printed.

For the third syntax listed above, /m or /method argument specifies which calculation method should be taken, both number1 and number2 are integer numbers that are used for calculation. These three arguments are required. If /nologo switch is turned on, no application logo and description information will be printed to the console, it is optional and by default the application

[1] ACF was built with .NET Framework 3.5, and our demonstration will also be performed with this version of .NET Framework.

will print logo and description information.

Followings are the valid commands to use the **catool** application.

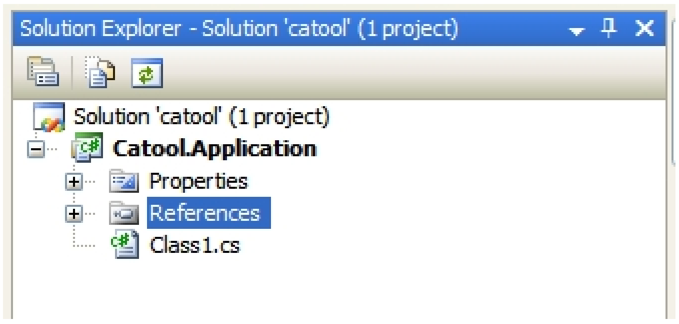
- catool.exe
- catool.exe /help
- catool.exe /?
- catool.exe /m:add 10 20
- catool.exe /method:sub 20 10
- catool.exe /m:mul 20 10 /nologo

Now let’s start to use ACF to develop such console application.

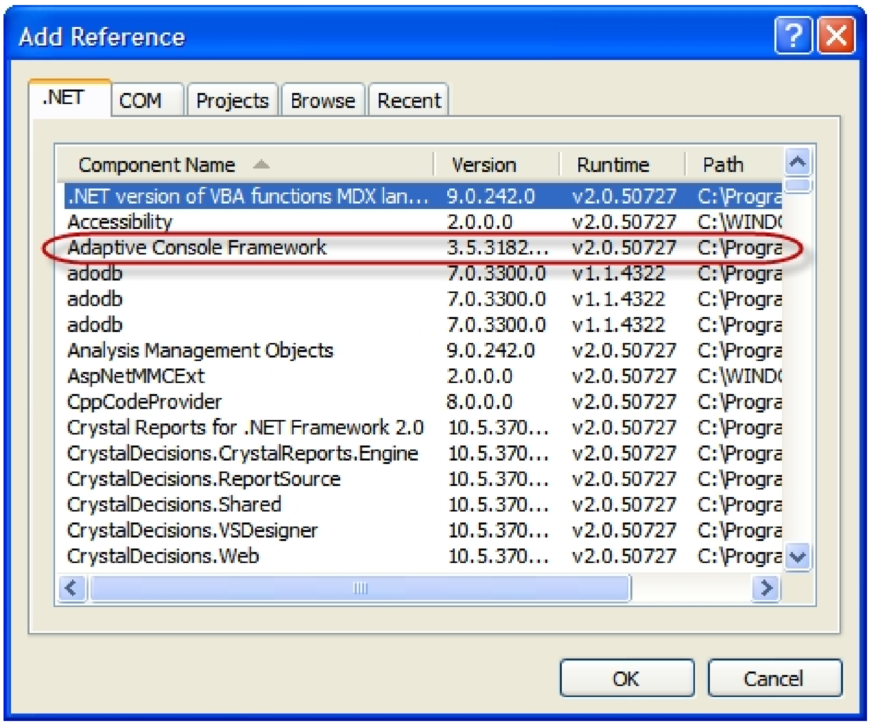
Create the Console Application Instance

The first step is to create a new console application instance. It is implemented by inheriting the **AdaptiveConsole.ConsoleApplicationBase** class. Follow the steps below to achieve the goal.

1. Start Visual Studio 2008, create a new blank solution named **catool**. On the **catool** solution, right click and add a new class library called **Catool.Application**.


Solution Explorer - Solution 'catool' (1 project)
Solution 'catool' (1 project)
Catool.Application
Properties
References
Class1.cs

2. On the **References** node, right click and select **Add Reference** item. An **Add Reference** dialog box will appear. Select **Adaptive Console Framework** in the dialog box and click **OK** to add the assembly.



Component Name	Version	Runtime	Path
.NET version of VBA functions MDX lan...	9.0.242.0	v2.0.50727	C:\Progra
Accessibility	2.0.0.0	v2.0.50727	C:\WIND(
Adaptive Console Framework	3.5.3182...	v2.0.50727	C:\Progra
adodb	7.0.3300.0	v1.1.4322	C:\Progra
adodb	7.0.3300.0	v1.1.4322	C:\Progra
adodb	7.0.3300.0	v1.1.4322	C:\Progra
Analysis Management Objects	9.0.242.0	v2.0.50727	C:\Progra
AspNetMMCExt	2.0.0.0	v2.0.50727	C:\WIND(
CppCodeProvider	8.0.0.0	v2.0.50727	C:\Progra
Crystal Reports for .NET Framework 2.0	10.5.370...	v2.0.50727	C:\Progra
CrystalDecisions.CrystalReports.Engine	10.5.370...	v2.0.50727	C:\Progra
CrystalDecisions.ReportSource	10.5.370...	v2.0.50727	C:\Progra
CrystalDecisions.Shared	10.5.370...	v2.0.50727	C:\Progra
CrystalDecisions.VSDesigner	10.5.370...	v2.0.50727	C:\Progra
CrystalDecisions.Web	10.5.370...	v2.0.50727	C:\Progra

3. Create a new class named **Catool** within the class library; make the class inherited from the AdaptiveConsole.ConsoleApplicationBase class, and implements the abstract memebers.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using AdaptiveConsole;

namespace Catool.Application
{
    public class Catool : ConsoleApplicationBase
    {
        public Catool(string[] args) : base(args) { }

        protected override string Description
        {
            get
            {
                return "The simple calculator for integer numbers.";
            }
        }

        protected override string Logo
        {
            get
            {
                StringBuilder sb = new StringBuilder();
                sb.Append("Catool v1.0\n");
                sb.Append("Copyright (C) 2007-2008, SunnyChen.ORG, all rights reserved.");
                return sb.ToString();
            }
        }
    }
}
```

Create Option Contracts

ACF defined four types of option contracts: None, Exact, Patternized and Free.

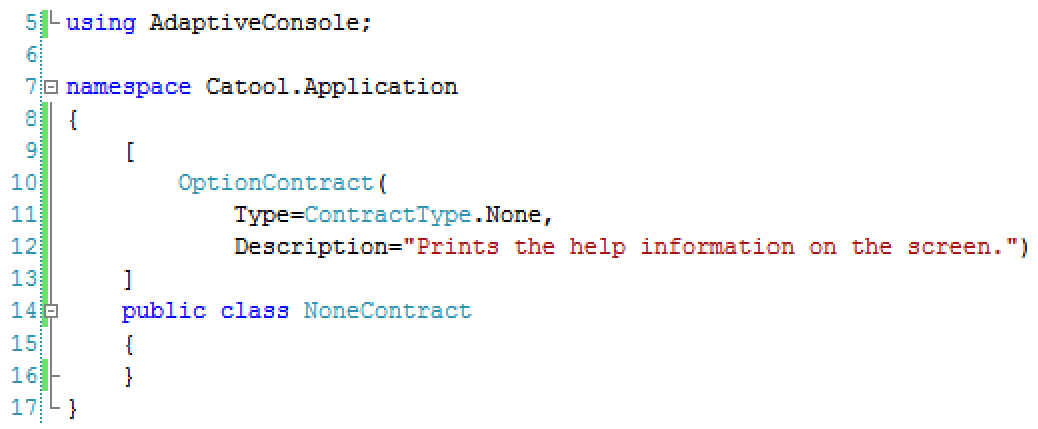
Type	Description
None	The console application requires no command line argument. When no argument is provided, the contract will be executed.
Exact	The console application requires exactly matched argument. For example, "**catool.exe /help"** matches such contract.
Patternized	The console application requires a complex command line argument. You can define the options that are not mandatory within the contract, you can define the options that carry a list of values and you can even define the switches in the patternized contracts. For example, "**catool.exe /m:add 10 20**" matches such contract.
Free	Any command line argument is acceptable. No argument will be considered as option when using this type of contract.

Note: A console application can only have one contract with the type of **None** and **Free**. And if the **Free** contract is used, you cannot define any **Patternized** contract within the application.

Because we can start **catool** application without any command line argument, which makes the application prints the help information on the screen, so we must define a **None** contract for the application. We want the application to handle the /help or /? arguments, so an **Exact** contract is also required. Furthermore, a **Patternized** contract must be defined either.

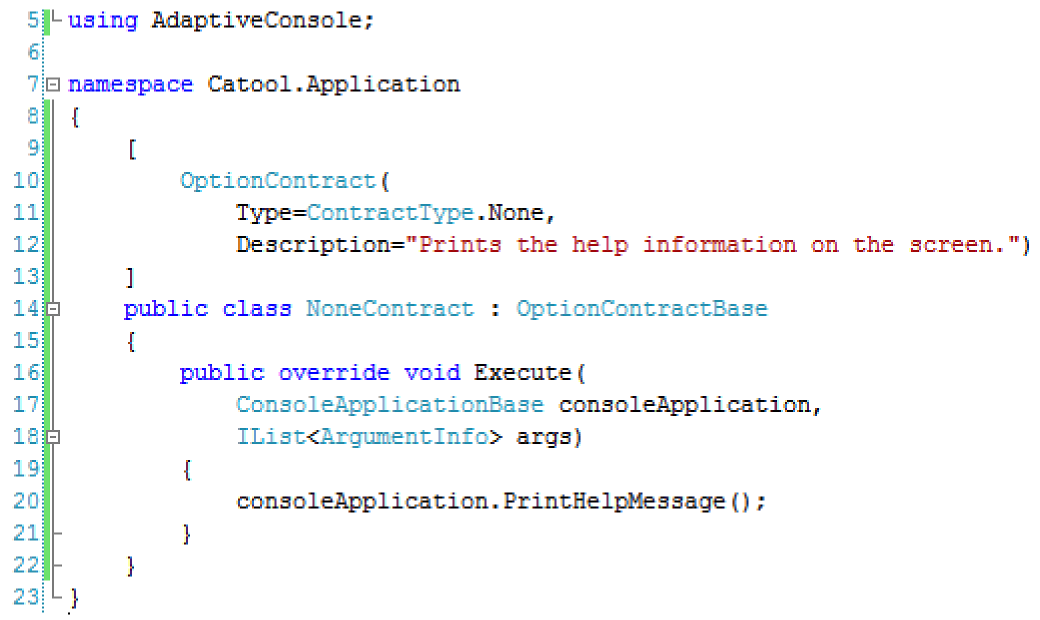
Create the None Contract

1. Add a new class named **NoneContract** to the class library, applies the **OptionContract** attribute on the class as follows. The description field, although it is not mandatory, will be specified here for generating the help documentation.



```
using AdaptiveConsole;

namespace Catool.Application
{
    [
        OptionContract(
            Type=ContractType.None,
            Description="Prints the help information on the screen.")
    ]
    public class NoneContract
    {
    }
}
```

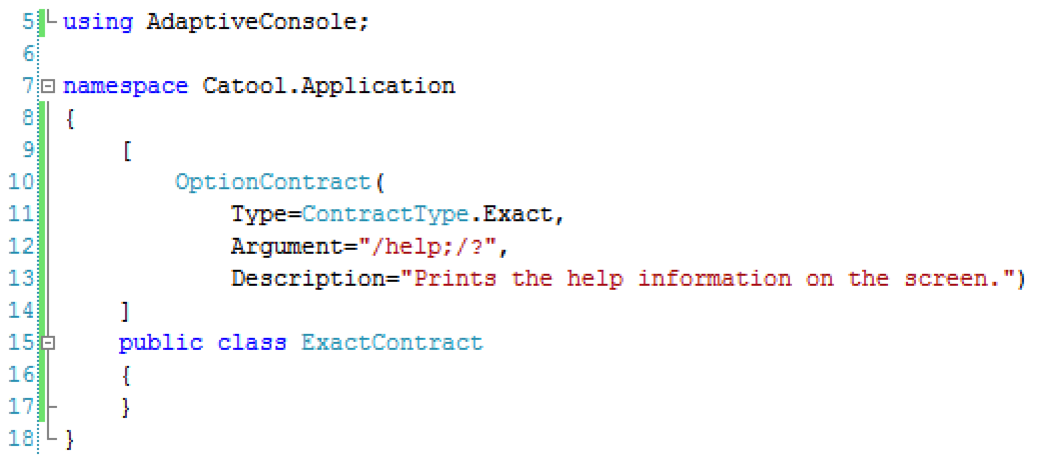
2. Make the **NoneContract** class inherited from the **OptionContractBase** class and implements the abstract members. Here we want to implement the **Execute** method so that the contract will be executed when no command line argument is provided. In the **Execute** method of the contract, we simply call the **PrintHelpMessage** method on the console application instance.



```
using AdaptiveConsole;

namespace Catool.Application
{
    [
        OptionContract(
            Type=ContractType.None,
            Description="Prints the help information on the screen.")
    ]
    public class NoneContract : OptionContractBase
    {
        public override void Execute(
            ConsoleApplicationBase consoleApplication,
            IList<ArgumentInfo> args)
        {
            consoleApplication.PrintHelpMessage();
        }
    }
}
```

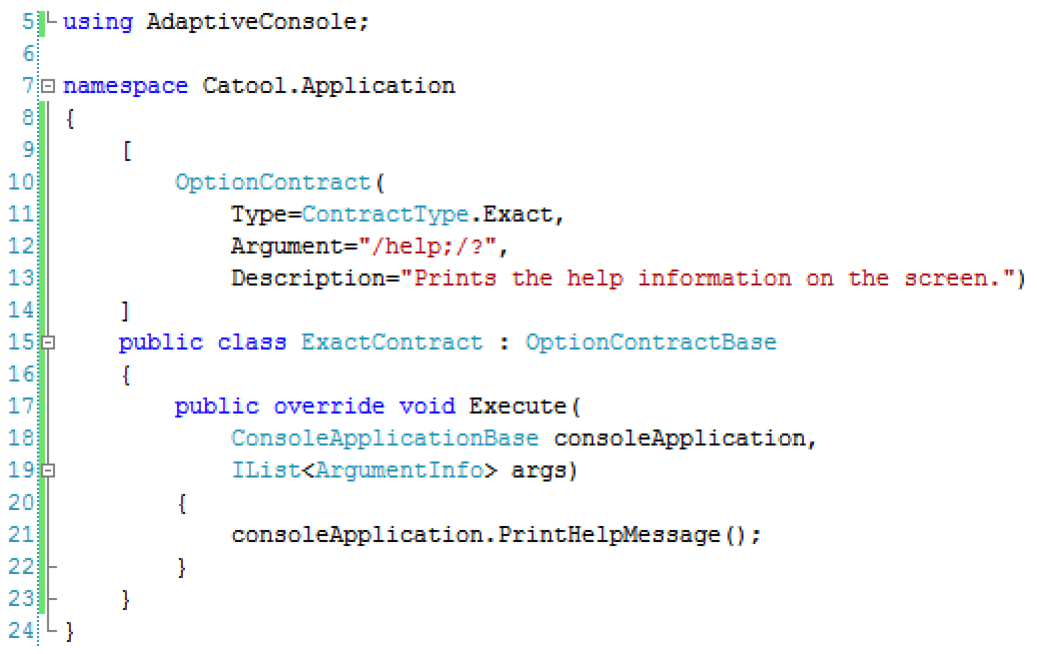
Create the Exact Contract

1. Add a new class named **ExactContract** to the class library, applies the **OptionContract** attribute on the class as follows. Note that the **Argument** field must be specified on the **Exact** contracts. The description field, although it is not mandatory, will be specified here for generating the help documentation.



```
using AdaptiveConsole;

namespace Catool.Application
{
    [
        OptionContract(
            Type=ContractType.Exact,
            Argument="/help;/?",
            Description="Prints the help information on the screen.")
    ]
    public class ExactContract
    {
    }
}
```

2. Make the **ExactContract** class inherited from the **OptionContractBase** class and implements the abstract members. Here we also want to implement the **Execute** method so that the contract will be executed when no command line argument is provided. In the **Execute** method of the contract, we simply call the **PrintHelpMessage** method on the console application instance.

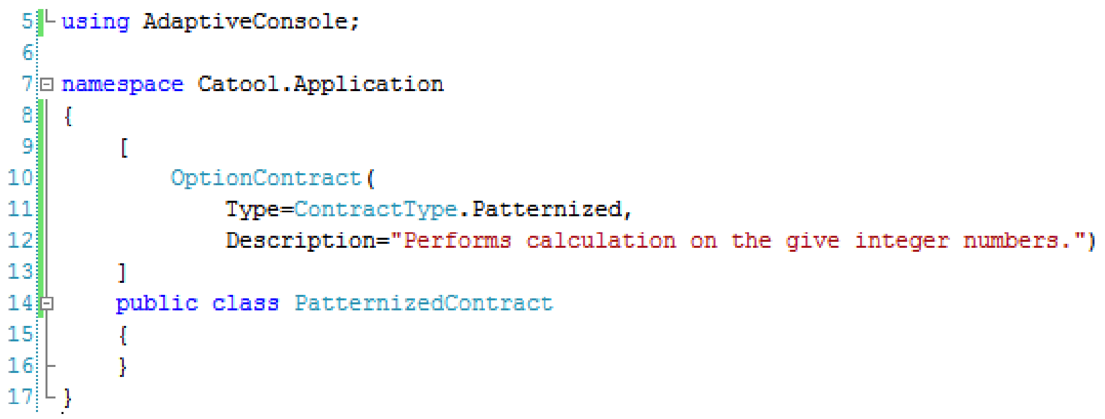


```
using AdaptiveConsole;

namespace Catool.Application
{
    [
        OptionContract(
            Type=ContractType.Exact,
            Argument="/help;/?",
            Description="Prints the help information on the screen.")
    ]
    public class ExactContract : OptionContractBase
    {
        public override void Execute(
            ConsoleApplicationBase consoleApplication,
            IList<ArgumentInfo> args)
        {
            consoleApplication.PrintHelpMessage();
        }
    }
}
```

Create the Patternized Contract

Now we are going to create a patternized contract, which is the core executing unit in our application. The contract accepts the command line arguments as options or parameters and ACF will populate the properties defined in the contract automatically once the contract is being matched.

1. Add a new class named **PatternizedContract** to the class library, applies the **OptionContract** attribute on the class as follows.



```
using AdaptiveConsole;

namespace Catool.Application
{
    [
        OptionContract(
            Type=ContractType.Patternized,
            Description="Performs calculation on the give integer numbers.")
    ]
    public class PatternizedContract
    {
    }
}
```

2. Add contract options to the **PatternizedContract** class.

   Each contract option refers to a public property within the class. And the option may have three different types: **SingleValue**, **ValueList** and **Switch**.

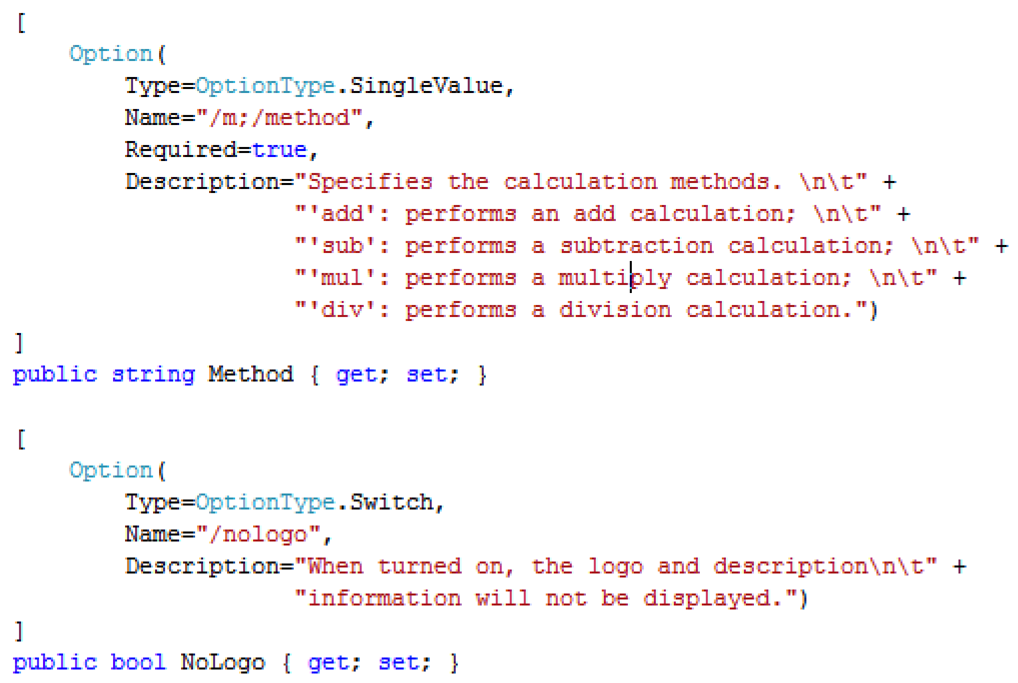
Type	Description	Property Type	Example
SingleValue	The current option has a single value parameter.	System.String	/output:a.xml
ValueList	The current option has a list of parameters.	System.Array	/input:a.xsd,b.xsd
Switch	The current option is a switch. This means it is not mandatory and if it is not specified in the command line arguments, the value of the property will be set to False.	System.Boolean	/verbose

   No matter which type the option is, the **Name** field of the option must be specified. And we also define the **Description** field on each option here for the documentation.

   <u>Something about command line arguments.</u>

   In ACF, command line arguments are grouped into two different types: **Option** and **Parameter**. When ACF tries to match each contract within the application, it will try to populate all the public properties that are marked by **OptionAttribute**, and mark the matched argument as **Option**. For those that don't match any defined option will be considered as **Parameter**.

   For example, in the command "**catool.exe /m:add 10 20 /nologo**", the argument "/m:add" and "/nologo" will be defined within the contract and it will be considered as options. Others don't match anything (like the integral values 10 and 20) will be the type of parameter.



```
[
    Option(
        Type=OptionType.SingleValue,
        Name="/m;/method",
        Required=true,
        Description="Specifies the calculation methods. \n\t" +
                    "'add': performs an add calculation; \n\t" +
                    "'sub': performs a subtraction calculation; \n\t" +
                    "'mul': performs a multiply calculation; \n\t" +
                    "'div': performs a division calculation.")
]
public string Method { get; set; }

[
    Option(
        Type=OptionType.Switch,
        Name="/nologo",
        Description="When turned on, the logo and description\n\t" +
                    "information will not be displayed.")
]
public bool NoLogo { get; set; }
```

3. Make the **PatternizedContract** class inherited from the **OptionContractBase** class and implements the abstract members. Now we get the full implementation of the class which would be something like the following.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using AdaptiveConsole;

namespace Catool.Application
{
    [
       OptionContract(
           Type=ContractType.Patterized,
           Description="Performs calculation on the give integer numbers.")
    ]
    public class PatternizedContract : OptionContractBase
    {
        [
            Option(
                Type=OptionType.SingleValue,
                Name="/m;/method",
                Required=true,
                Description="Specifies the calculation methods. \n\t" +
                           "'add': performs an add calculation; \n\t" +
                           "'sub': performs a subtraction calculation; \n\t" +
                           "'mul': performs a multiply calculation; \n\t" +
                           "'div': performs a division calculation.")
       ]
       public string Method { get; set; }

        [
            Option(
                Type=OptionType.Switch,
                Name="/nologo",
                Description="When turned on, the logo and description\n\t" +
                           "information will not be displayed.")
        ]
       public bool NoLogo { get; set; }

       public override void Execute(
            ConsoleApplicationBase consoleApplication,
            IList<ArgumentInfo> args)
       {
            // Checks if the command line argument carries just 2 parameter
            // arguments.
            var parameterArguments = from arg in args
                                     where arg.Type == ArgumentType.Parameter
                                     select arg;
            if (parameterArguments.Count() != 2)
           {
                consoleApplication.PrintHelpMessage();
                return;
           }
           // If NoLogo is not specified, print the logo to console.
           if (!this.NoLogo)
                consoleApplication.PrintLogo();

           // Gets the two numbers from command line argument.
           int num1, num2;
           try
           {
                num1 = Convert.ToInt32(parameterArguments.ElementAt(0).Argument);
                num2 = Convert.ToInt32(parameterArguments.ElementAt(1).Argument);
           }
           catch (FormatException)
           {
                Console.WriteLine("{0} requires two integral numbers as parameters.",
                    consoleApplication.ApplicationName);
                return;
```

```
            }
            catch (OverflowException)
            {
                Console.WriteLine("Parameter overflow.");
                return;
            }

            // Calculates the result
            try
            {
                int result = 0;
                switch (this.Method.ToUpper())
                {
                    case "ADD":
                        result = num1 + num2;
                        break;
                    case "SUB":
                        result = num1 - num2;
                        break;
                    case "MUL":
                        result = num1 * num2;
                        break;
                    case "DIV":
                        result = num1 / num2;
                        break;
                }
                Console.WriteLine(result);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                return;
            }
        }
    }
}
```

Now we have finished implementing the option contracts and the console application instance. Next step we should create a console application host and configures the components we have created.

The host of the ACF console application is of course a console application. You just need to create a console application project within the solution, write little code and do some configuration to the host.

Create and Configure the Console Application Host

Create the Host

1. Add a new console application project named **catool** to the solution. Also add the reference to the ACF as mentioned in the previous steps.

2. On **catool** console application, add the reference to **Catool.Application** class library, this ensures that the generated assembly will be copied to the **catool** output directory once the solution is being compiled.

3. Using the **AdaptiveConsole** namespace in your console application and write the code in the **Main** method as following.

```
using AdaptiveConsole;

namespace catool
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                ConsoleApplicationManager.RunApplication(args);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

Now we have successfully created the console application host by using ACF. How easy it is to create a console application with complex command line arguments.

Configure the Console Application Host

1. Add an **app.config** file to your **catool** console application project.

2. Modify the configuration file as follows.

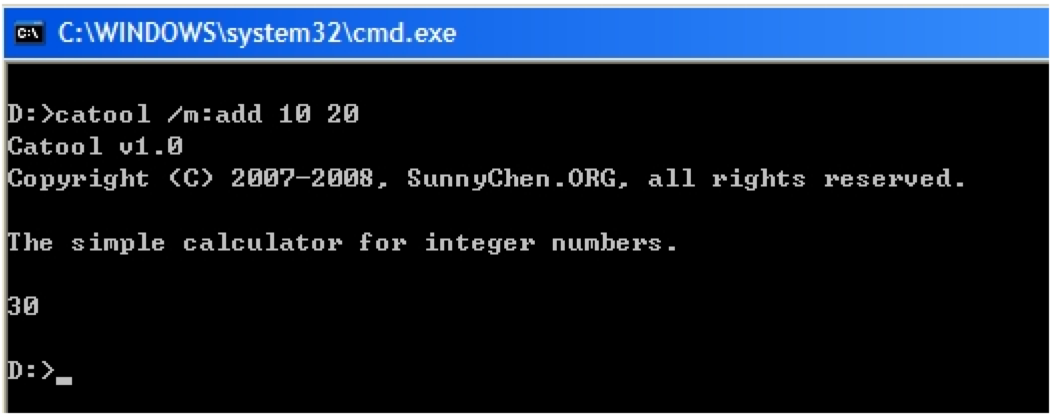
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="AdaptiveConsole"
             type="AdaptiveConsole.Config.AdaptiveConsoleConfigHandler,
             AdaptiveConsole, Version=3.5.3182.35766, Culture=neutral, PublicKeyToken=f7c45863541f79da"/>
  </configSections>
  <AdaptiveConsole provider="Catool.Application.Catool, Catool.Application"
                   contractRepository="Catool.Application"/>
</configuration>
```

In the configuration file, firstly we created a configuration section named **AdaptiveConsole**, this name is specific to ACF and cannot be changed. For the ACF configuration, it has two attributes. The **provider** attribute indicates the console application provider, which is the assembly qualified name of the console application instance we have created. The **contractRepository** refers to the assembly name in which the contracts are contained.

Run the Console Application

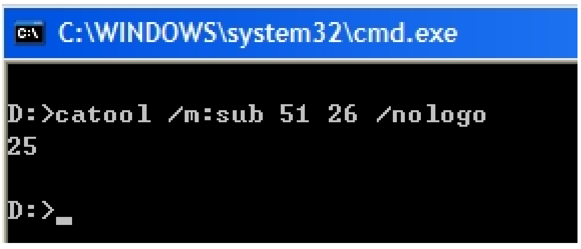
After compiling the solution, we can run the console application by typing its name with arguments in the command window.

- Add 10 with 20 and print the result



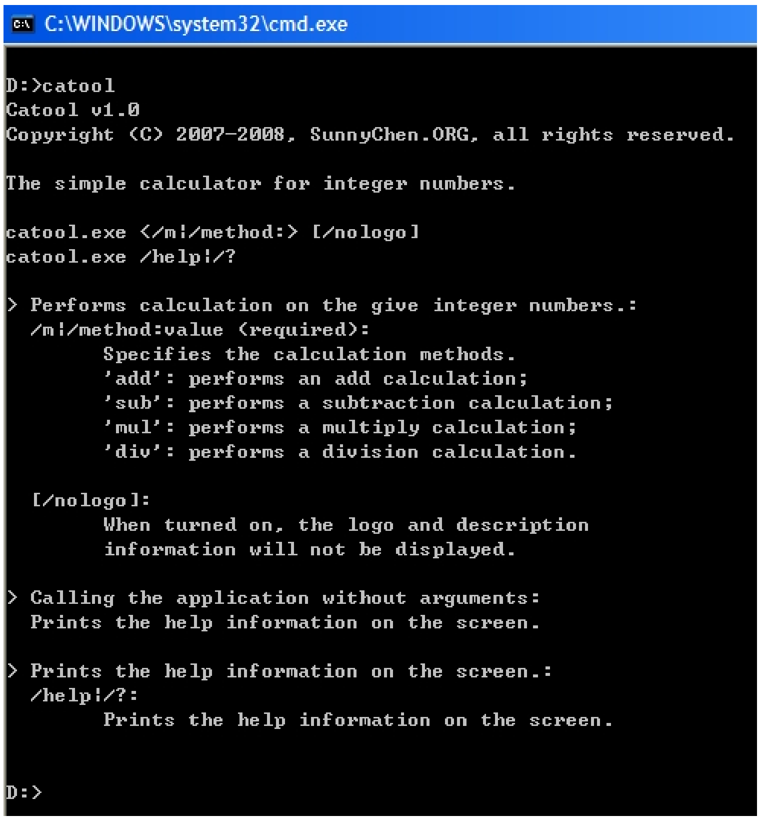
```
D:>catool /m:add 10 20
Catool v1.0
Copyright (C) 2007-2008, SunnyChen.ORG, all rights reserved.

The simple calculator for integer numbers.

30

D:>_
```

- Subtract 26 from 51 and print the result without any logo and information



```
D:>catool /m:sub 51 26 /nologo
25

D:>_
```

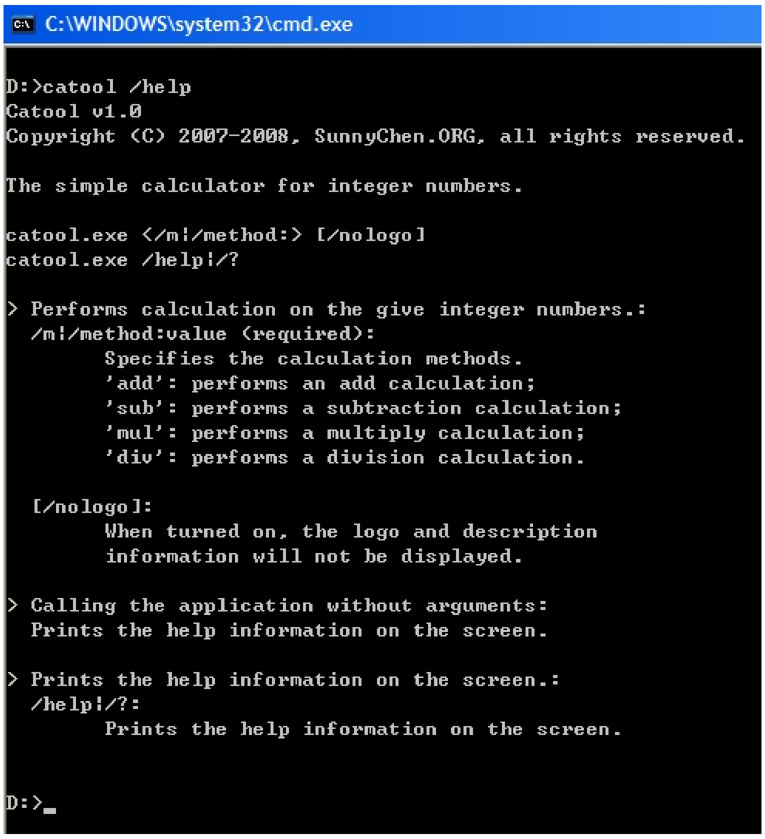
- Running the application without any argument



C:\WINDOWS\system32\cmd.exe

```
D:>catool
Catool v1.0
Copyright (C) 2007-2008, SunnyChen.ORG, all rights reserved.

The simple calculator for integer numbers.

catool.exe </m|/method:> [/nologo]
catool.exe /help|/?

> Performs calculation on the give integer numbers.:
  /m|/method:value (required):
        Specifies the calculation methods.
        'add': performs an add calculation;
        'sub': performs a subtraction calculation;
        'mul': performs a multiply calculation;
        'div': performs a division calculation.

  [/nologo]:
        When turned on, the logo and description
        information will not be displayed.

> Calling the application without arguments:
  Prints the help information on the screen.

> Prints the help information on the screen.:
  /help|/?:
        Prints the help information on the screen.


D:>
```

- Running the application with /help argument



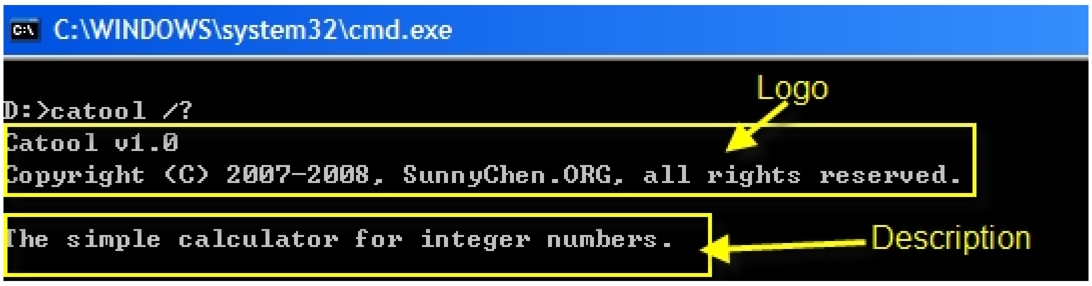
C:\WINDOWS\system32\cmd.exe

```
D:>catool /help
Catool v1.0
Copyright (C) 2007-2008, SunnyChen.ORG, all rights reserved.

The simple calculator for integer numbers.

catool.exe </m|/method:> [/nologo]
catool.exe /help|/?

> Performs calculation on the give integer numbers.:
  /m|/method:value (required):
        Specifies the calculation methods.
        'add': performs an add calculation;
        'sub': performs a subtraction calculation;
        'mul': performs a multiply calculation;
        'div': performs a division calculation.

  [/nologo]:
        When turned on, the logo and description
        information will not be displayed.

> Calling the application without arguments:
  Prints the help information on the screen.

> Prints the help information on the screen.:
  /help|/?:
        Prints the help information on the screen.


D:>_
```

Something about the help screen

1. Logo and Description

   Both Logo and Description are the abstract properties in **ConsoleApplicationBase** class. When creating a concrete class for console application instance, you must implement the Logo and Description properties. This is done by simply returning a human readable string so that when help screen is displayed, information about the console application will be shown properly.

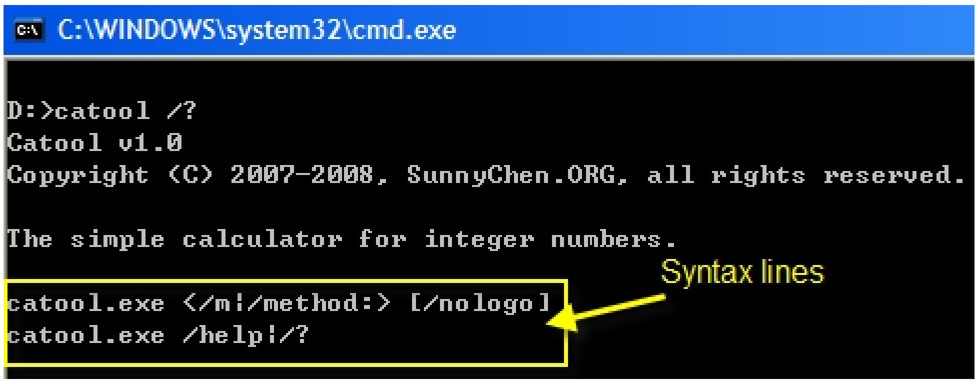


```
C:\WINDOWS\system32\cmd.exe

D:>catool /?
Catool v1.0
Copyright (C) 2007-2008, SunnyChen.ORG, all rights reserved.

The simple calculator for integer numbers.
```

   Note: If no description is specified, the description line will be ignored.

2. Contract syntax lines

   Syntax lines are for guiding the user to use the application with valid arguments. Each contract represents a single syntax line, which will be generated automatically when application runs. When showing the help screen, syntax lines for all the contracts will be displayed just after the Logo and Description information.

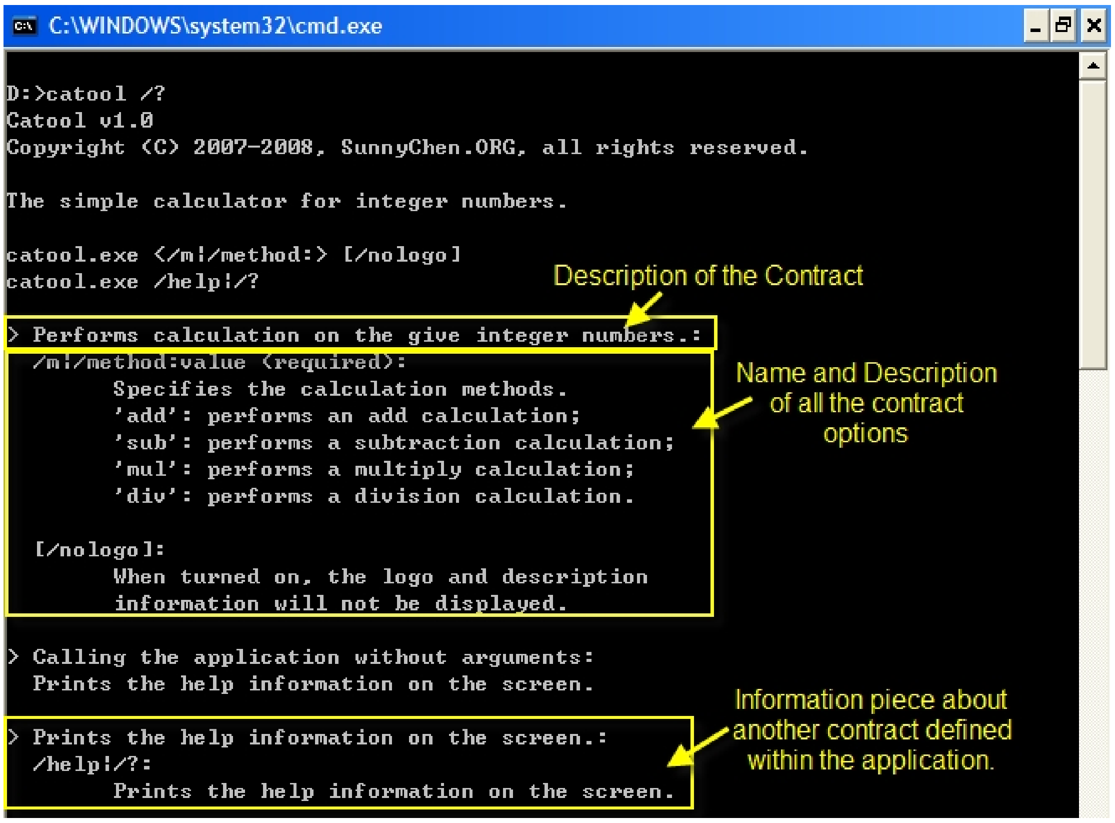


```
C:\WINDOWS\system32\cmd.exe

D:>catool /?
Catool v1.0
Copyright (C) 2007-2008, SunnyChen.ORG, all rights reserved.

The simple calculator for integer numbers.

catool.exe </m|/method:> [/nologo]
catool.exe /help|/?
```

   You can see that for mandatory options (**Required** field is specified on the **Option** attribute), the name of the option will be embraced by angular brackets, for optional options (**Required** field is not specified on the **Option** attribute), the name of the option will be embraced by square brackets. These are also done by ACF and you don't need to do anything at all.

3. Detailed information about Contracts

   Each contract may have detailed information be displayed on the help screen. The first line is defined by the Description field of the **OptionContract** attribute. ACF will then enumerate all the options within the contract and display its name and description.


C:\WINDOWS\system32\cmd.exe
Description of the Contract
Name and Description of all the contract options
Information piece about another contract defined within the application.

```
D:>catool /?
Catool v1.0
Copyright (C) 2007-2008, SunnyChen.ORG, all rights reserved.

The simple calculator for integer numbers.

catool.exe </m|/method:> [/nologo]
catool.exe /help|/?

> Performs calculation on the give integer numbers.:
  /m|/method:value (required):
        Specifies the calculation methods.
        'add': performs an add calculation;
        'sub': performs a subtraction calculation;
        'mul': performs a multiply calculation;
        'div': performs a division calculation.

  [/nologo]:
        When turned on, the logo and description
        information will not be displayed.

> Calling the application without arguments:
  Prints the help information on the screen.

> Prints the help information on the screen.:
  /help|/?:
        Prints the help information on the screen.
```

You can also define your own help screen by overriding the **Syntax** and **HelpText** attributes that are defined within the **OptionContractBase** class when implementing your contracts.