Drukuj Zamknij

Zastosowanie InteropServices do utworzenia biblioteki w C# umożliwiającej wczytywanie znaków z konsoli bez wyprowadzania echa na ekran

2003-12-14 22:56:28
Krzysztof Woźniak - Wydział Informatyki, Politechnika Szczecińska

Wydawać by się mogło, że programy okienkowe całkowicie wyprą aplikacje konsolowe. Tymczasem Microsoft dopiero w Windows Server 2003 wprowadził możliwość wykonania wszystkich zadań administracyjnych z poziomu konsoli. We wcześniejszych wersjach systemu niektóre funkcje wymagały użycia narzędzi okienkowych. Interfejs tekstowy posiada wciąż niesłabnącą popularność wśród administratorów systemów zwłaszcza Unixowych. Jest też podstawą do tworzenia rozmaitych skryptów.

Załóżmy na przykład, że tworzymy prostego, konsolowego klienta FTP. Wśród parametrów programu możemy podać adres serwera FTP, czy nazwę użytkownika. Absolutnie nie na miejscu byłoby natomiast takie samo potraktowanie hasła. To ostatnie powinno pojawić się w formie co najwyżej znaków '*'.

Aby zaimplementować wspomnianą funkcjonalność konieczne jest uzyskanie bezpośredniego dostępu do metod pozwalających na kontrolę standardowego wejścia. Metody takiej jak już wiemy nie znajdziemy wśród klas .NET Framework Skoro jednak istniały i istnieją konsolowe aplikacje Windows pozwalające na poufne podawanie danych, to jasne jest, że niezbędne możliwości zawarte są wśród funkcji Win32API.

Mała dygresja – jak szukać niezbędnych funkcji:

W tym konkretnym przypadku potrzebowaliśmy funkcji pozwalających zmieniać stan (tryb) pracy wejścia konsolowego. Stąd też zaczynamy poszukiwać metody zawierającej słowo Console i Mode. Bingo – istnieje: ***SetConsoleMode*** *i* ***GetConsoleMode.*** *Obie funkcje wymagają wśród parametrów uchwytu do buforu wejściowego bądź wyjściowego. Podążając tym tropem (słowa Get, Handle) trafiamy na ostatnią niezbędną metodę:* ***GetStdHandle.*** *Jeśli w poszukiwaniach wykorzystaliśmy MSDN, to w opisie każdej z tych funkcji znajdziemy również jej lokalizację – w naszym przypadku zawsze jest to* ***kernel32.dll.***

W celu umożliwienia programistom .NET korzystanie z zewnętrznych API, pisanych w niezarządzanym kodzie wprowadzono atrybut **DllImportAttribute**. Jego znajomość jest nam niezbędna aby kontrolować standardowe wejście z konsoli.

Tymczasem rozpocznijmy budowę naszej biblioteki. Na początku warto zaimportować stosowną przestrzeń nazw:

```
using System.Runtime.InteropServices;
```

Jak widać wynika to z szacunku do klawiatury i własnych nadgarstków. Teraz możemy przystąpić do tworzenia naszej klasy. W artykule zaimplementujemy projekt o następującej strukturze:

```
namespace KRIS
{
        public class PasswordConsole
    {
        //Import niezbędnych funkcji Win32API
            //Deklaracje statycznych pól klasy
        //Prywatna metoda TurnOffEcho
            //Prywatna metoda TurnOnEcho
            //Publiczna metoda Read
        //Publiczna metoda ReadLine
        }
}
```

Na samym początku ciała klasy **PasswordConsole** należy dodać prototypy wykorzystywanych funkcji Win32API:

```
    [DllImport("kernel32")]
        private static extern int GetStdHandle(int whichHandle);
    [DllImport("kernel32")]
private static extern bool GetConsoleMode(int handle,          out uint mode);
    [DllImport("kernel32")]
private static extern bool SetConsoleMode(int handle,            uint mode);
```

W tym miejscu warto zatrzymać się przy dwóch rzeczach – wykorzystaniu atrybutu **DllImportAttribute** tworzeniu prototypów funkcji zewnętrznych API.

Atrubut **DllImportAttribute** wskazuje, że oznaczona nim metoda została zaimplementowana w zewnętrznej, niezarządzanej bibliotece. Jedynym i zarazem koniecznym parametrem pozycyjnym tego atrybutu jest nazwa biblioteki dll zawierającej daną metodę. Podawanie rozszerzenia dll pliku jest opcjonalne:

```
[DllImport("kernel32")]
```

jest równoznaczne:

```
[DllImport("kernel32.dll")]
```

Wszystkie parametry nazwane tego atrybutu są opcjonalne i w naszym przypadku zbędne. Ogólnie natomiast należy zwracać szczególną uwagę na dobór parametru **CharSet** dla zewnętrznych funkcji operujących na ciągach znakowych. Pozwala on ustalić sposób kodowania wartości typu **string**. Natomiast w przypadku wywoływania metod należących do niezarządzanych klas konieczne jest ustawienie parametru **CallingConvention** na **ThisCall**. Dzięki temu pierwszym parametrem stanie się **this** i trafi on do rejestru, nie na stos. Wszystkie użyte tu funkcje zewnętrzne zwracają wartość pozwalającą stwierdzić, czy wykonały się poprawnie. Jeśli natomiast korzystamy z metod Win32API nie dostarczających takich informacji to możemy ustawić parametr **SetLastError=true**. Pozwoli to na odczytanie informacji o błędnym wykonaniu poprzez inną metodę Win32API – **GetLastError**.

Przejdźmy teraz to prototypów funkcji. Jak widać zaczynamy od standardowych modyfikatorów dostępu – public, protected itd. Są one opcjonalne – dokładnie jak w przypadku zwyczajnych, lokalnych metod klasy. Następnie muszą pojawić się słowa kluczowe **static extern**. Innymi słowy funkcje takie są statyczne, a słowo extern oznacza po prostu, że są zdefiniowane w zewnętrznej, niezarządzanej bibliotece. Dalsza część wygląda już zupełnie zwyczajnie. Jednakże kryje się tu pewien haczyk. Wszystkie typy parametrów, oraz typ zwracanej wartości wymagają przekształcenia na typy zgodne z .NET Framework. Dodatkowo parametry, poprzez które jest zwracana jakaś wartość muszą być opatrzone słowem kluczowym **out**. Samo mapowanie typów niezarządzanych na zarządzane jest natomiast sprawą dość swobodną. Należy jednak uważać!!! Kompilator nie zaprotestuje, jeśli użyjemy typu „mniejszego”, bądź „większego” niż oryginalny. Na przykład typem zwracanym przez **GetStdHandle** jest zgodnie z Win32API 4-bajtowy **DWORD**. Program zadziała bez przeszkód, jeśli w prototypie użyjemy w tym miejscu 2-bajtowego **Int16.** Problem się jednak pojawi, jeśli funkcja zwróci wartość przekraczającą **Int16.MaxValue**. Analogiczna sytuacja ma miejsce w przypadku parametrów wejściowych, jeśli przekażemy wartość większą niż dopuszcza typ oryginalnej funkcji.

Aby zapewnić podstawową funkcjonalność związaną z odczytem danych, musimy dodać do naszej klasy **PasswordConsole** dwie, mające swoje funkcjonalne odpowiedniki w klasie Console, metody: **Read** i **ReadLine**:

```
public static int Read()
{
    //Wyłączenie echa
    TurnOffEcho();
    //Odczyt kodu znaku
    //Przywrócenie echa
    TurnOnEcho();
    //Zwrócenia kodu wczytanego znaku
}

public static string ReadLine()
{
    //Wyłączenie echa
    TurnOffEcho();
    //Odczyt znaków aż do wciśnięcia Entera
    //Przywrócenie echa
    TurnOnEcho();
    //Zwrócenie wczytanego ciągu
}
```

Skupmy się teraz na zapewnieniu naszej klasie możliwości włączania i wyłączania echa.

Zacznijmy od funkcji **TurnOffEcho**. Pierwszą rzeczą, której potrzebujemy jest uchwyt do standardowego wejścia konsolowego. Uchwyt taki zwraca **GetStdHandle** po wybraniu i przekazaniu w postaci argumentu wybranego urządzenia (a ściślej – odpowiadającej mu wartości liczbowej):

Nazwa urządzenia w Win32API	**Wartość**	**Opis urządzenia**
STD_INPUT_HANDLE	-10	Standardowe wejście – klawiatura

STD_OUTPUT_HANDLE	-11	Standardowe wyjście – ekran
STD_ERROR_HANDLE	-12	Standardowe wyjście dla błędów

Uchwyt ten niezbędny będzie również w metodzie TurnOnEcho, dlatego należy stworzyć przechowujące go pole w naszej klasie. Tak samo zadeklarujemy stałą oznaczającą urządzenie standardowego wejścia:

```
/// <summary>
/// Numer urządzenia oznaczający standardowe wejście
/// </summary>
private const int STD_INPUT_HANDLE = -10;
/// <summary>
/// Uchwyt do standardowego wejścia z konsoli
/// </summary>
private static int hConsole;
```

W celu pobrania uchwytu musimy użyć poniższej konstrukcji:

```
//Pobranie uchwytu do standardowego wejścia
if((hConsole = GetStdHandle(STD_INPUT_HANDLE)) == -1)
    throw new ApplicationException("GetStdHandle error");
```

W przypadku błędu funkcja zwraca **–1**, na co reagujemy rzucając stosowny wyjątek.

Następnie mając uchwyt do standardowego wejścia możemy dowolnie zmienić tryb jego pracy. Na początek odczytujemy bieżący tryb – **GetConsoleMode** – przyda się później m. in. do jego przywrócenia. Nie chcemy też całkowicie go zmieniać, a jedynie wyłączyć kilka znaczników – flag. Do jego przechowywania użyjemy poniższego pola klasy:

```
/// <summary>
/// Zestaw flag oznaczający początkowy tryb pracy konsoli
/// </summary>
private static uint oldMode;
```

A tak odczytujemy tryb:

```
//Pobranie trybu pracy konsoli
if(!GetConsoleMode(hConsole,out oldMode))
    throw new ApplicationException("GetConsoleMode error");
```

Po wykonaniu powyższego kodu zmienna oldMode zawierać będzie zbiór flag oznaczający tryb pracy (w przypadku błędu funkcja zwróci **false**, a my rzucimy wyjątek). Możliwe flagi trybu to:

Nazwa trybu w Win32API	**Wart.**	**Opis**
ENABLE_PROCESSED_INPUT	1	Ctrl+C jest obsługiwane przez system i nie trafia do bufora
ENABLE_LINE_INPUT	2	Powrót z funkcji odczytujących następuje dopiero po wciśnięciu klawisza Enter
ENABLE_ECHO_INPUT	4	Wczytane znaki trafiają automatycznie na ekran
ENABLE_WINDOW_INPUT	8	Do bufora trafiają również informacje o zmianach rozmiaru okna konsoli
ENABLE_MOUSE_INPUT	16	Zdarzenia myszy trafiają do bufora

Jako że chcemy odczytywać z konsoli po jednym znaku i nie wyprowadzać echa na ekran musimy wyłączyć flagi **ENABLE_LINE_INPUT** i **ENABLE_ECHO_INPUT**. Tworzymy więc następujące stałe:

```
/// <summary>
/// Flaga oznaczająca powrót po wciśnięciu Entera
/// </summary>
private const uint ENABLE_LINE_INPUT = 2;
/// <summary>
/// Flaga oznaczająca włączone echo
/// </summary>
private const uint ENABLE_ECHO_INPUT = 4;
```

I wyłączamy niezbędne flagi:

```
//Wyłączenie niezbędnych flag
uint newMode = oldMode & ~(ENABLE_LINE_INPUT | ENABLE_ECHO_INPUT);
```

Teraz możemy przystąpić już do zmiany trybu pracy wejścia - **SetConsoleMode**:

```
//Ustawienie nowego trybu pracy konsoli
if(!SetConsoleMode(hConsole,newMode))
    throw new ApplicationException("SetConsoleMode error");
```

Jeśli funkcja zwróci **true** to znaczy, że zmiana trybu udała się.

Pora zająć się operacją odwracającą zmiany. Tym razem nasze zadanie jest banalne. Wewnątrz funkcji **TurnOnEcho** wystarczy jedno wywołanie funkcji **SetConsoleMode**:

```
//Przywrócenia starego trybu pracy konsoli
if(!SetConsoleMode(hConsole,oldMode))
    throw new ApplicationException("SetConsoleMode error");
```

Oczywiście wszelkie zmiany trybu pracy konsoli zawsze dotyczą jedynie konsoli należącej do wykonywanego programu. Ewentualnie używane inne konsole zachowywać się będą bez zmian.

Jedyne, co pozostało do wykonania to wczytywanie wpisywanych danych. W przypadku naszej funkcji **Read** wystarczy posłużyć się standardową metodą klasy **Console**:

```
//Odczyt kodu znaku
int i = Console.Read();
```

Odczytaną wartość należy zwrócić po przywróceniu echa – po wywołaniu **TurnOnEcho**:

```
//Zwrócenie kodu wczytanego znaku
return i;
```

Funkcja ReadLine wymaga nieco więcej operacji. Ponieważ wyłączyliśmy flagę ENABLE_LINE_INPUT, standardowe wejście zwraca każdy wprowadzony znak pojedynczo. Tym samym **Console.ReadLine** nie działa prawidłowo i musimy korzystać z **Console.Read**:

```
int i;//Kod aktualnie odczytanego znaku
StringBuilder secret = new StringBuilder();//Odczytany ciąg
//Odczyt znaków aż do wciśnięcia Entera
while(true)
{
    //Odczyt następnego znaku
    i = Console.Read();
    if(i == 13)//Jeśli Enter to koniec odczytywania
        break;
    //Dodanie znaku do ciągu
    secret.Append((char) i);
}
//Symulacja wciśniętego Entera
Console.WriteLine();
```

Przy dodawaniu po jednym znaku do ciągu znacznie efektywniejszy jest obiekt klasy **StringBuilder**, od stałego obiektu **string**. Jeśli użytkownik wciśnie **Enter**, czyli znak o kodzie 13, kończymy. Po wywołaniu **TurnOffEcho** należy zwrócić utworzony ciąg:

```
//Zwrócenie wczytanego ciągu
return secret.ToString();
```

Teraz wystarczy już tylko wykonać polecenie:

```
csc /t:library PasswordConsole.cs
```

aby otrzymać gotową do użycia bibliotekę **PasswordConsole.dll**.

Użycie tak zbudowanej biblioteki wygląda identycznie (czytaj równie łatwo) jak każdej innej, standardowej biblioteki. Wcześniej wspominałem o przykładzie z klientem FTP. Załóżmy więc, że mamy klasę **FTPClient**. Klasa ta reprezentuje prostego klienta FTP i dostarcza jedną, statyczną metodę – **CreateMirror**. Ta z kolei służy do wykonania na dysku lokalnym zwierciadlanej kopii danego katalogu z serwera FTP:

Po pierwsze importujemy przestrzeń nazw klasy **PasswordConsole** (Jeśli korzystamy z MS VS .NET to musimy dodać do projektu referencję do naszej biblioteki – **PasswordConsole.dll**):

```
using KRIS;//Klasa PasswordConsole
```

A tak wygląda właściwy kod:

```
/// <summary>
/// Tworzy obraz katalogu serwera FTP przy pomocy obiektu FTPClient
/// </summary>
class Tester
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        //
        // TODO: Add code to start application here
        //
        //Program wymaga podania 3 parametrów
        if(args.Length != 3)
        {
            Console.WriteLine("Użycie:");
Console.WriteLine("MirrorFTP.exe NazwaSerwera Użytkownik Katalog");
            return;
        }
        //Pobieranie hasła
        Console.Write("Wprowadź hasło: ");
        string pass = PasswordConsole.ReadLine();
        //Próba utworzenia zwierciadlanego odbicia katalogu z
        //serwera ftp
        if(FTPClient.CreateMirror(args[0],args[1],pass,args[2]))
            Console.WriteLine("Operacja udana");
        else
            Console.WriteLine("Operacja nie powiodła się");
    }
}
```

Jak widać aplikacja wymaga podania trzech parametrów. Sam odczyt podawanego hasła (wytłuszczony druk) jest sprawą zupełnie banalną. Użycie metod klasy **PasswordConsole** nie odbiega zupełnie od tego, do czego przyzwyczaiła nas klasa **Console**.

Jeśli nie używamy Visual Studio to aplikację MirrorFTP kompilujemy w poniższy sposób:

```
csc /r:PasswordConsole.dll MirrorFTP.cs
```

Nie należy moim zdaniem oczekiwać, że w przyszłych wersjach biblioteki klas .NET Framework zaimplementowana zostanie cała funkcjonalność Win32API. W związku z tym poznanie opisywanego w tym artykule rozwiązania, może okazać się bardzo pomocne.

Do artykułu załączony jest pełen kod źródłowy zarówno biblioteki PasswordConsole jak i aplikacji MirrorFTP.

Drukuj Zamknij