Extension Methods in Windows PowerShell

https://web.archive.org/web/20201109023419/http://bartdesmet.net/blogs/bart/archive/2007/09/06/extension-methods-in-windows-powershell.aspx

You've probably already heard about this new feature in .NET 3.5: extension methods. If not, check out a few posts on this topic:

- C# 3.0 Feature Focus - Part 4 - Extension Methods
- Visual Basic 9.0 Feature Focus - Extension Methods

As a quick recap, extension methods allow you to "extend" an existing type with additional methods, without touching the type's implementation itself. In fact, it's nothing more than syntactical sugar on top of static methods. The code below shows such a set of extension methods written in C# 3.0:

```
using System;

namespace MyExtensions
{
   public static class Extensions
   {
      private static Random rand = new Random();

      public static string Reverse(this string s)
      {
         char[] c = s.ToCharArray();
         Array.Reverse(c);
         return new string(c);
      }

      public static string Permute(this string s, int n)
      {
         char[] c = s.ToCharArray();

         for (int i = 0; i < n; i++)
         {
            int a = rand.Next(s.Length);
            int b = rand.Next(s.Length);
            char t = c[a];
            c[a] = c[b];
            c[b] = t;
         }
```

```
            return new string(c);
        }

        public static int Power(this int i, int n)
        {
            int res = 1;
            for (int j = 0; j < n; j++)
                res *= i;
            return res;
        }
    }
}
```

These methods allow you to write the following, if the MyExtensions namespace is imported:

```
string name = "Bart";
string reverseName = name.Reverse(); //traB
string fuzzyName = name.Permute(1); //can produce different results where two letters are switched, e.g. Brat
int n = 2;
int kb = n.Power(10); //1024
```

So far so good. But what does all of this magic have to do with PowerShell? The answer: nothing (so far). As you know, Windows PowerShell 1.0 was developed in the .NET 2.0 timeframe, so it would be very strange if PowerShell understood extension methods. And indeed, it doesn't ... which made me think how we could make this available using some plumbing. Here's what I came up with.

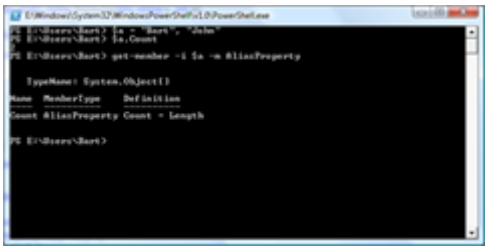
Windows PowerShell has a feature called the Extended Type System. This allows types to be extended with additional properties, methods, etc in order to provide additional IT admin convenience. For example, take a look at the types.ps1xml file in the %windir%\system32\WindowsPowerShell\v1.0 folder on your system. In there you'll find things like:

```
<Type>
  <Name>System.Array</Name>
  <Members>
    <AliasProperty>
      <Name>Count</Name>
      <ReferencedMemberName>Length</ReferencedMemberName>
    </AliasProperty>
  </Members>
</Type>
```

This defines an "alias property" which adds a property called Count to each instance of System.Array, pointing to the Length property available on System.Array. So, you can write this:

```
> $a = "Bart", "John"
> $a.Count
2
```

In fact, if you use get-member on $a, you'll see the AliasProperty listed out there (click to enlarge):



In a similar way, one can make different types of extensions: Alias Properties, Code Properties, Note Properties, Script Properties, Code Methods, Script Methods. Take a closer look at types.ps1xml for additional samples. Back to our mission now. It seems ETS is an appropriate vehicle to make extension methods available using Script Methods. Basically, we'll provide a script for each extension method that takes the set of original parameters and rewrites these to become parameters of the static method. For example, if you write:

```
> $name = "Bart"
> $name.Reverse()
```

the last call should become:

```
> [MyExtensions.Extensions]::Reverse($name)
```

Similarly, a Power call on an int should be translated from:

```
> $n = 2
> $n.Power(10)
```

into:

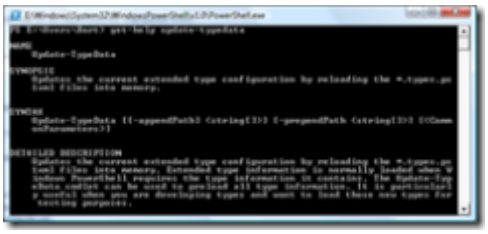
```
> [MyExtensions.Extensions]::Power($n, 10)
```

Taking possible method overloads into account, we should end up with something like this:

```
<?xml version="1.0" encoding="utf-16"?>
<Types>
  <Type>
    <Name>System.String</Name>
    <Members>
      <ScriptMethod>
        <Name>Reverse</Name>
        <Script>
          switch ($args.Count) {
            0 { [MyExtensions.Extensions]::Reverse($this) }
            default { throw "No overload for Reverse takes the specified number of parameters." }
```

```
        }
      </Script>
    </ScriptMethod>
    <ScriptMethod>
      <Name>Permute</Name>
      <Script>
        switch ($args.Count) {
          1 { [MyExtensions.Extensions]::Permute($this, $args[0]) }
          default { throw "No overload for Permute takes the specified number of parameters." }
        }
      </Script>
    </ScriptMethod>
  </Members>
 </Type>
 <Type>
  <Name>System.Int32</Name>
  <Members>
    <ScriptMethod>
      <Name>Power</Name>
      <Script>
        switch ($args.Count) {
          1 { [MyExtensions.Extensions]::Power($this, $args[0]) }
          default { throw "No overload for Power takes the specified number of parameters." }
        }
      </Script>
    </ScriptMethod>
  </Members>
 </Type>
</Types>
```

Once we have such a file, it can be "imported" in Windows PowerShell using the Update-TypeData cmdlet:



All we have to do is call this cmdlet as follows:

```
> Update-TypeData -prependPath MyExtensions.ps1xml
```

But it gets even better: this chunk of XML is something that's an ideal candidate for dynamic ~~code~~ XML generation. Guess what, let's use LINQ for this task and wrap the "extension method export" functionality in a custom cmdlet:

```
using System;
using System.IO;
```

```
using System.Linq;
using System.Management.Automation;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Text;
using System.Xml;
using System.Xml.Linq;

namespace BdsSoft.PowerShell.ExtensionMethods
{
   [Cmdlet("Import", "ExtensionMethods")]
   public class ImportExtensionMethods : PSCmdlet
   {
      [Parameter(Mandatory=true)]
      public Assembly Assembly { get; set; }

      [Alias("ns"), Parameter(Mandatory=false)]
      public string Namespace { get; set; }

      protected override void ProcessRecord()
      {
         if (Namespace == null)
            Namespace = "";

         var res =
            new XDocument(
               new XElement("Types",
                  from t in Assembly.GetTypes()
                  where t.Namespace != null && t.Namespace.StartsWith(Namespace)
                  from m in t.GetMethods(BindingFlags.Public | BindingFlags.Static)
                  where m.GetCustomAttributes(typeof(ExtensionAttribute), false).Length == 1
                  group m by m.GetParameters()[0].ParameterType into g
                  select
                     new XElement("Type",
                        new XElement("Name", g.Key.FullName),
                        new XElement("Members",
                           from m in g
                           group m by m.Name into h
                           select
                              new XElement("ScriptMethod",
                                 new XElement("Name", h.Key),
                                  new XElement("Script", GetScriptFor(h))
                           )
                        )
                     )
               )
            );

         StringBuilder sb = new StringBuilder();
```

```
            using (TextWriter tw = new StringWriter(sb))
            {
               using (XmlTextWriter xtw = new XmlTextWriter(tw))
               {
                  xtw.Indentation = INDENT;
                  xtw.Formatting = Formatting.Indented;
                  res.WriteTo(xtw);
               }
            }

            base.WriteObject(sb.ToString());
         }

         static int INDENT = 2;

         static string GetScriptFor(IGrouping<string, MethodInfo> m)
         {
            StringBuilder sb = new StringBuilder();
            sb.AppendFormat("\r\n{0}switch ($args.Count) {{\r\n", new String(' ', INDENT * 5));

            foreach (var e in m)
            {
               int n = e.GetParameters().Length - 1;
               sb.AppendFormat("{0}{1} {{ {2} }}\r\n", new String(' ', INDENT * 6), n,
GetScriptFor(e.DeclaringType.FullName, e.Name, n));
            }

            sb.AppendFormat("{0}default {{ throw \"No overload for {1} takes the specified
number of parameters.\" }}\r\n", new String(' ', INDENT * 6), m.Key);
            sb.AppendFormat("{0}}}\r\n{1}", new String(' ', INDENT * 5), new String(' ',
INDENT * 4));

            return sb.ToString();
         }

         static string GetScriptFor(string type, string method, int n)
         {
            StringBuilder sb = new StringBuilder();
            sb.Append("$this");

            for (int i = 0; i < n; i++)
               sb.AppendFormat(", $args[{0}]", i);

            string args = sb.ToString();

            return String.Format("[{0}]::{1}({2})", type, method, args);
         }
      }
   }
```

Quite a bit of code, but lots of plumbing to get a smooth output (notice the code can be improved in many places). The GetScriptFor methods are pretty simple to understand and generate the script for a given (group of) method (overloads) associated with a method name (the second GetScriptFor method is a helper to get the method calls themselves, using the $this and $args variables). For what the core functionality is concerned, take a look at the ProcessRecord method that contains a LINQ query that looks for all extension methods in the given assembly and namespace:

```
        var res =
            new XDocument(
                new XElement("Types",
                    from t in Assembly.GetTypes()
                    where t.Namespace IsNot Nothing AndAlso
t.Namespace.StartsWith(Namespace)
                    from m in t.GetMethods(BindingFlags.Public | BindingFlags.Static)
                    where m.GetCustomAttributes(typeof(ExtensionAttribute), false).Length == 1
                    group m by m.GetParameters()[0].ParameterType into g
                    where !g.Key.IsGenericType
                    select
                        new XElement("Type",
                            new XElement("Name", g.Key.FullName),
                            new XElement("Members",
                                from m in g
                                group m by m.Name into h
                                select
                                    new XElement("ScriptMethod",
                                        new XElement("Name", h.Key),
                                        new XElement("Script", GetScriptFor(h))
                                )
                            )
                        )
                )
            );
```

In here, we're using the new System.Xml.Linq API to contruct an XML fragment on the fly (LINQ to XML style). A method is considered to be an extension method if it's public, static and marked with a System.Runtime.CompilerServices.ExtensionAttribute custom attribute. Furthermore, it's first parameter (the "this" parameter in C# 3.0) shouldn't be a generic type, since PowerShell doesn't have first-level support for generics at the moment (this restriction rules out the use of System.Core's extension methods unfortunately, at the moment). Using the grouping constructs, methods are grouped per type and per method name (to account for overloads). VB folks have an easier job when it comes down to generating XML fragments. The query above looks as follows in VB 9.0 (more info on VB 9.0 XML integration):

```
Dim res = New XDocument( _
    <Types>
        <%= From t In _assembly.GetTypes _
            Where t.Namespace IsNot Nothing AndAlso t.Namespace.StartsWith(_namespace) _
            From m In t.GetMethods(BindingFlags.Public + BindingFlags.Static) _
            Where m.GetCustomAttributes(GetType(ExtensionAttribute), False).Length = 1 _
            Group m By Key = m.GetParameters()(0).ParameterType Into g = Group _
            Where Not Key.IsGenericType _
            Select <Type>
                       <Name><%= Key.FullName %></Name>
                       <Members>
                           <%= From m In g _
                               Group m By Key2 = m.Name Into h = Group _
                               Select <ScriptMethod>
                                          <Name><%= Key2 %></Name>
                                          <Script><%= GetScriptFor(h) %></Script>
                                      </ScriptMethod> _
                           %>
                       </Members>
                   </Type> _
        %>
    </Types>)
```

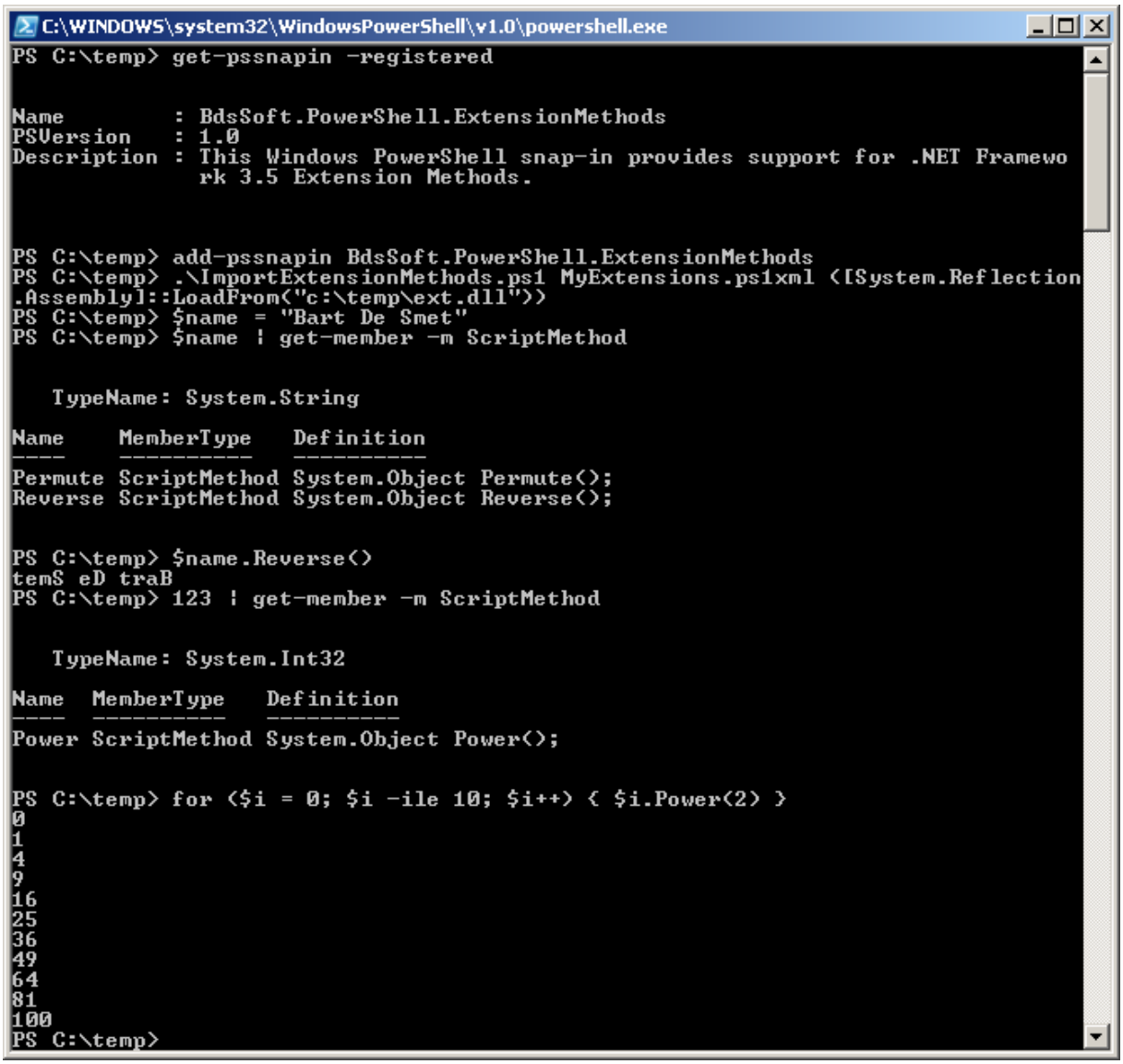
When writing a cmdlet, we need a snap-in as its distribution vehicle; below is a simple one:

```
using System.ComponentModel;
using System.Management.Automation;

namespace BdsSoft.PowerShell.ExtensionMethods
{
   [RunInstaller(true)]
   public class ExtensionMethodsSnapIn : PSSnapIn
   {
      public override string Description
      {
         get { return "This Windows PowerShell snap-in provides support for .NET
Framework 3.5 Extension Methods."; }
      }

      public override string Name
      {
         get { return "BdsSoft.PowerShell.ExtensionMethods"; }
      }

      public override string Vendor
      {
         get { return "BdsSoft"; }
      }
   }
}
```

Strong-name the assembly, build it and run installutil -i against the generated dll file. Finally, create the following PowerShell script, ImportExtensionMethods.ps1:

```
if ($args.Count -lt 2)
{
  throw "Usage: .\ImportExtensionMethods.ps1 output assembly [namespace]"
}
else
{
  $emOutput = (join-path (split-path $profile) $args[0])

  Import-ExtensionMethods -assembly $args[1] -namespace $args[2] | Out-File $emOutput
  Update-TypeData -PrependPath $emOutput
}
```

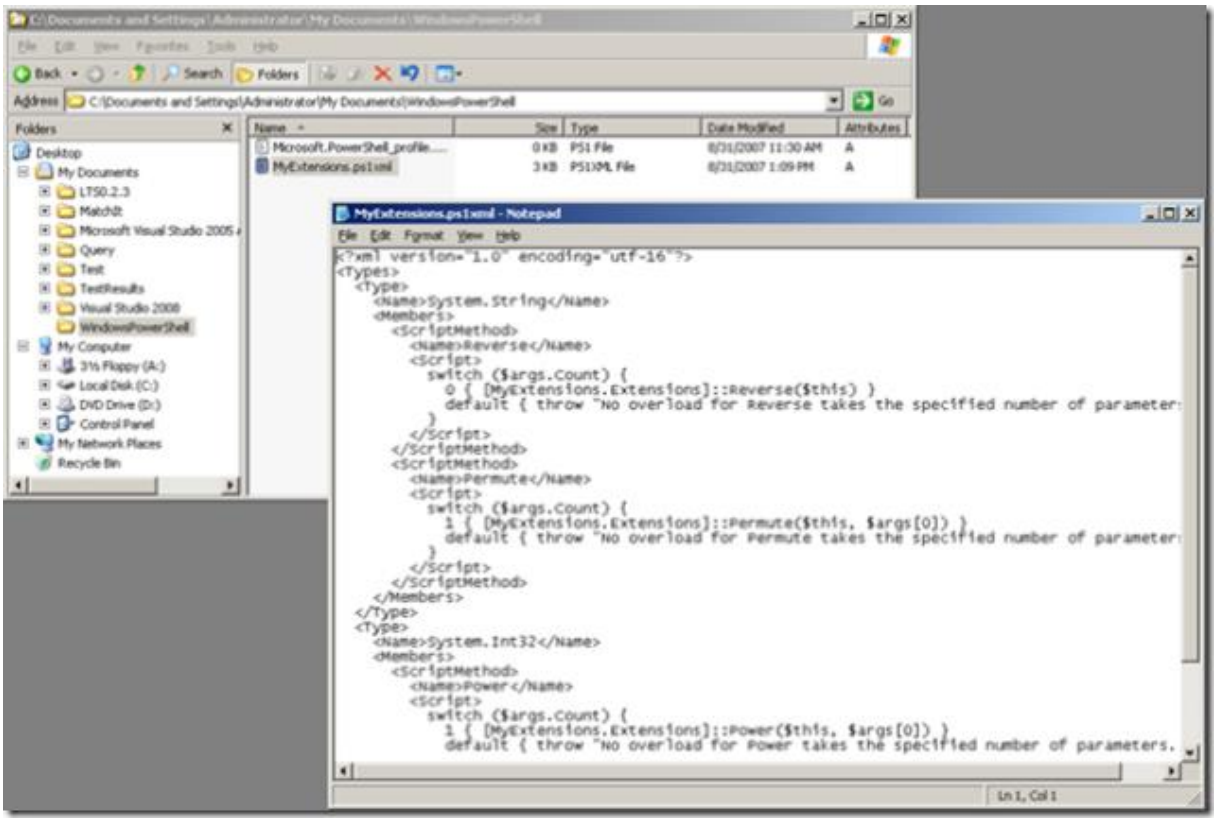
This simplifies calling the Import-ExtensionMethods cmdlet and to update the ETS type data, so that the extensions become available. In the screenshot below, you can see the whole thing in use:



All you need to do is:

1. Make sure the snap-in is loaded, using add-pssnapin (you can move this call to your PowerShell profile if you want to load it automatically).
2. Call ImportExtensionMethods.ps1, passing in a name for the ETS ps1xml file that should be generated followed by the assembly that contains the extension methods (use Assembly.LoadFrom or Assembly.Load to get the assembly either by file name or by assembly name, i.e. for GAC'ed assemblies).

Of course, once you have the ps1xml files, you could simply adapt your PowerShell profile file in order to load the assemblies and call Update-TypeData to load the ps1xml file. In the next screenshot you can see the generated ps1xml file (click to enlarge):



Have fun!