Windows Shells with an emphasis on PowerShell (Vista)

Voutouri Panagiotis
Lada Nikolas

History of Windows Shells

- COMMAND.COM(1981) for MS-DOS, Windows95/98/SE/Me
- cmd.exe(1993) for Windows 2000, XP, Server 2003
- Powershell(2006)
  - PowerShell is the new command line/scripting environment from Microsoft.

Command.com and cmd.exe limitations

- Cannot automate all administrative functions found in GUI applications
- Non POSIX.2 conformant
- Lack of adequate documentation
- Scripts where built as wrappers of GUI applications (and not the other way around like UNIX-based systems)
- Windows Script Host was introduced to address some of these shortcomings but suffered severe security vulnerabilities and was not well intergraded into the shell

Central Concepts

- PowerShell is a composite of the complex tasks of a series of components. The components are special programs called *cmdlets* (pronounced *command lets*), which are .NET classes designed to use the features of the environment.
- The key difference between the usual UNIX approach and the PowerShell one is that rather than creating a "pipeline" based on *textual* input and output, PowerShell passes data between the various cmdlets as *objects* (structured data).
- If accessed individually from the command line, a cmdlet's output will automatically be converted into text, but if its output is to be used by another cmdlet, it will be converted into whatever form of object is most appropriate for that cmdlet's input. This has the advantage of eliminating the need for the many text-processing utilities which are common in UNIX pipelines, such as grep and awk, as well as allowing things to be combined interactively, or in a scripting environment, which would otherwise require a more complex programming language

Advantages/Disadvantages

- **Advantages:**
  - Supports hash tables, switch statements , regular expressions, arrays, looping (for/foreach/while), conditional statements (if/switch), variable scoping (global/script/local), pipeline, functions.
  - Easy to discover its features (get-command, get –help, man)
  - Object Orientation (The output of a command is an object)
  - Using Familiar Command Names (cat, pwd, cls, rm, sort…)
  - Processing text, files, registry values, XML
  - Creates graphical User interface with Forms

- **Disadvantages:**
  - Too slow compared with unix shells (tested with fork)
  - Nothing new compared with unix shells (almost)

Install

- Download installation file from http://www.microsoft.com/windowsserver2003/technologies/management/powershell/download.mspx
- Supported Operating Systems:
  - Windows XP Service Pack 2 ,Vista
- Requires:
  - .NET Framework Version 2.0
- Enable execute for scripts
  - Set-ExecutionPolicy remotesigned
- Program HELLO, WORLD!
  - "hello, world!".ToUpper()

Powershell Basics

- Commands

  command –parameter1 –parameter2 argument1 argument2

- Powershell is POSIX compliant and backwards compatible with cmd.exe

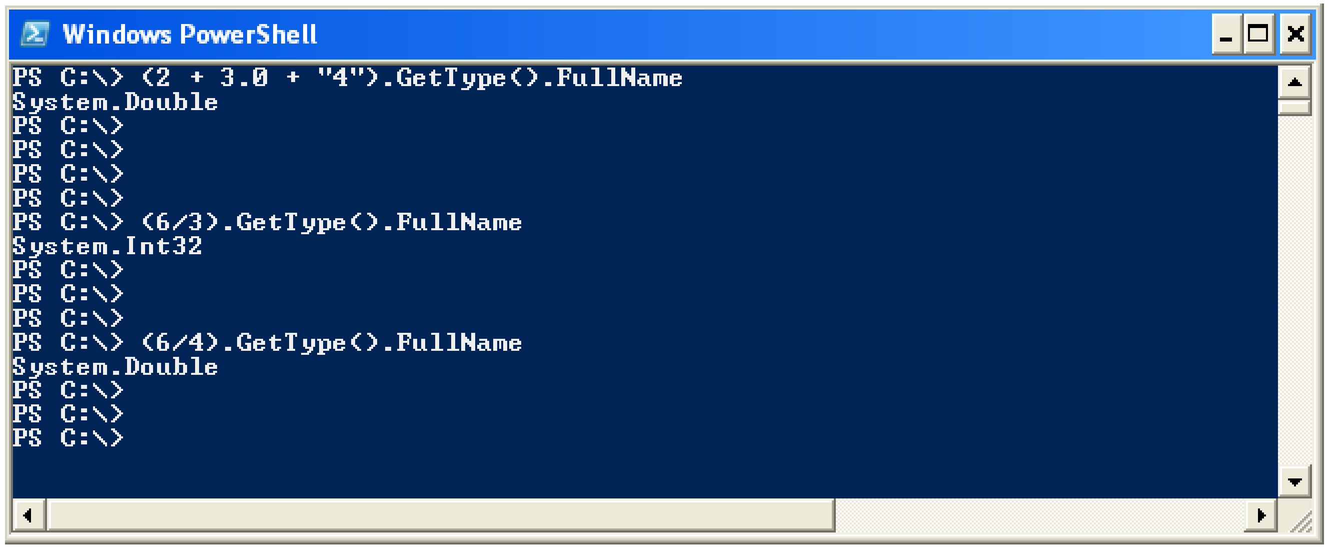
  ```
  > ls / | sort
  > dir c:\ | sort
  > get-process
  > ps
  ```

- Powershell parameter binding

  ```
  > ls –Recurse ;  ls –rec ; ls –r ; ls –R ; “all do the same thing”
  ```

Powershell Basics

- Commands in Powershell return .NET objects



```
PS C:\> (2 + 3.0 + "4").GetType().FullName
System.Double
PS C:\>
PS C:\>
PS C:\>
PS C:\>
PS C:\> (6/3).GetType().FullName
System.Int32
PS C:\>
PS C:\>
PS C:\>
PS C:\> (6/4).GetType().FullName
System.Double
PS C:\>
PS C:\>
PS C:\>
```

Variables

- Declaration is not required

```
> $a = 123123
```

- Type is automatically assumed

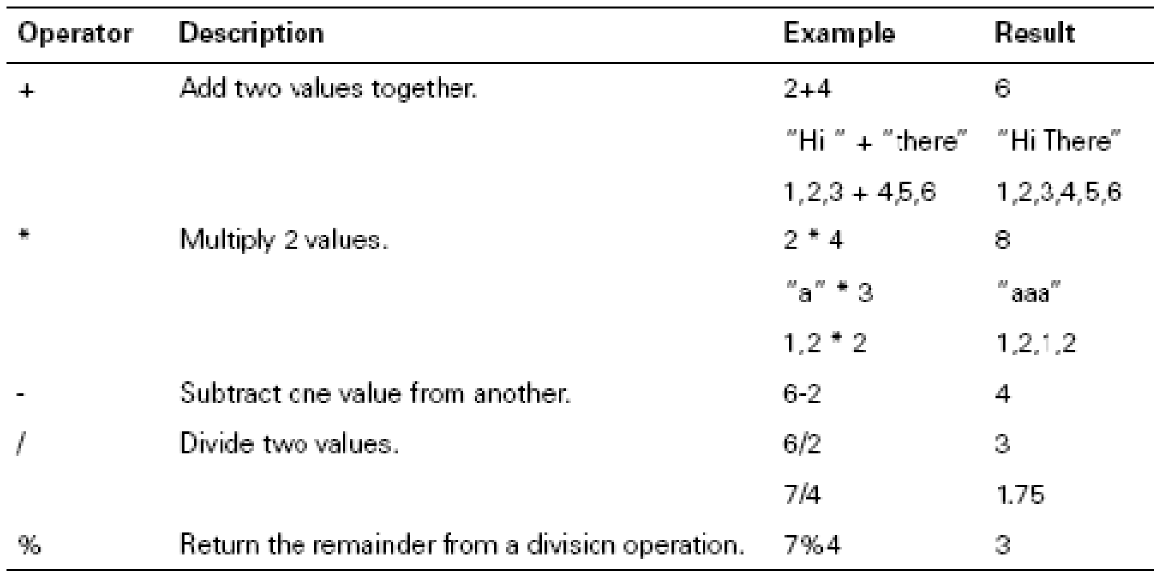
```
> $a = 12
> $a.GetType() ---int32
> $a = “aaa”
> $a.GetType() ---string
```

- Types can be freely mixed as long as there is no loss in precision

```
> 2 + 3.0 + "4"
```

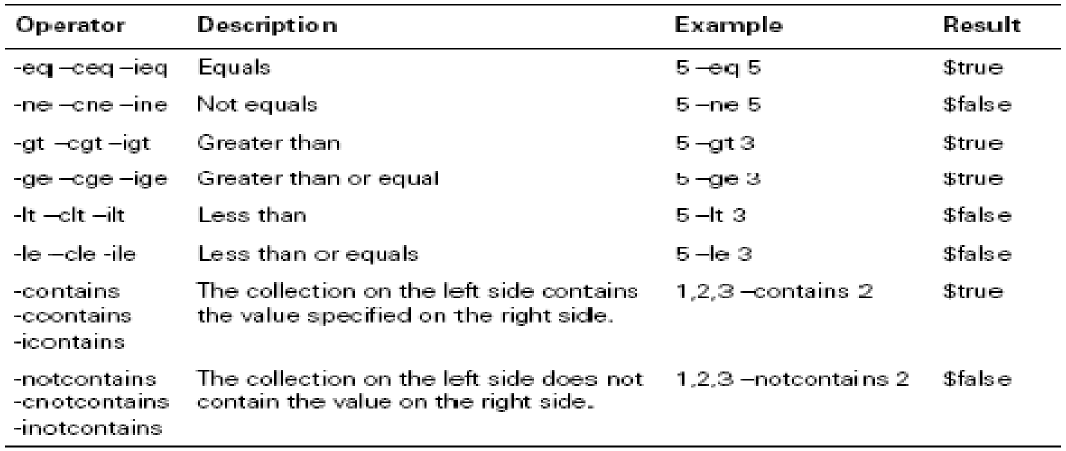
Operators

- Powershell supports all common operators found in programming and extends their functionality (operators are polymorphic)

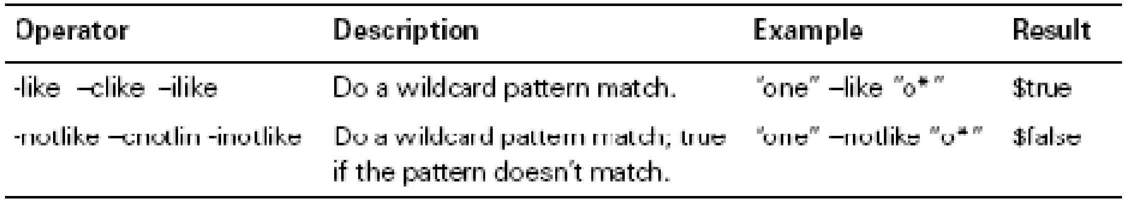


Operator	Description	Example	Result
+	Add two values together.	2+4	6
		"Hi " + "there"	"Hi There"
		1,2,3 + 4,5,6	1,2,3,4,5,6
*	Multiply 2 values.	2 * 4	8
		"a" * 3	"aaa"
		1,2 * 2	1,2,1,2
-	Subtract one value from another.	6-2	4
/	Divide two values.	6/2	3
		7/4	1.75
%	Return the remainder from a division operation.	7%4	3

Operators cont.

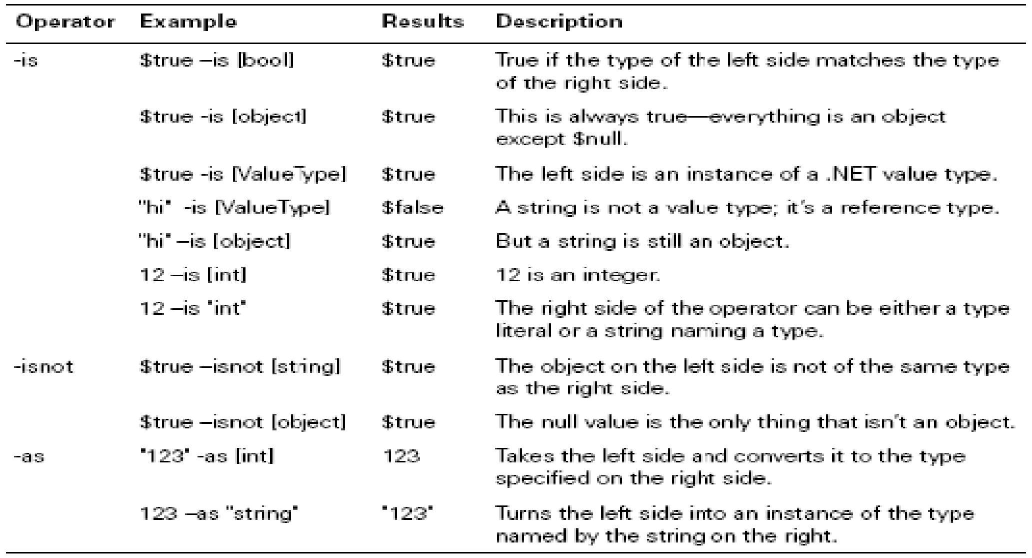


Operator	Description	Example	Result
-eq –ceq –ieq	Equals	5 –eq 5	$true
-ne –cne –ine	Not equals	5 –ne 5	$false
-gt –cgt –igt	Greater than	5 –gt 3	$true
-ge –cge –ige	Greater than or equal	5 –ge 3	$true
-lt –clt –ilt	Less than	5 –lt 3	$false
-le –cle -ile	Less than or equals	5 –le 3	$false
-contains -ccontains -icontains	The collection on the left side contains the value specified on the right side.	1,2,3 –contains 2	$true
-notcontains -cnotcontains -inotcontains	The collection on the left side does not contain the value on the right side.	1,2,3 –notcontains 2	$false



Operator	Description	Example	Result
-like –clike –ilike	Do a wildcard pattern match.	"one" –like "o*"	$true
-notlike –cnotlin -inotlike	Do a wildcard pattern match; true if the pattern doesn't match.	"one" –notlike "o*"	$false

Operators for working with types



Operator	Example	Results	Description
-is	$true –is [bool]	$true	True if the type of the left side matches the type of the right side.
	$true -is [object]	$true	This is always true—everything is an object except $null.
	$true -is [ValueType]	$true	The left side is an instance of a .NET value type.
	"hi" -is [ValueType]	$false	A string is not a value type; it's a reference type.
	"hi" –is [object]	$true	But a string is still an object.
	12 –is [int]	$true	12 is an integer.
	12 –is "int"	$true	The right side of the operator can be either a type literal or a string naming a type.
-isnot	$true –isnot [string]	$true	The object on the left side is not of the same type as the right side.
	$true –isnot [object]	$true	The null value is the only thing that isn't an object.
-as	"123" -as [int]	123	Takes the left side and converts it to the type specified on the right side.
	123 –as "string"	"123"	Turns the left side into an instance of the type named by the string on the right.

```
> foreach ($t in [float],[int],[string]) {"0123.45" -as $t}
```

Regular Expressions

```
Hello there.
My car is red. Your car is blue.
His car is orange and hers is gray.
Bob's car is blue too.
Goodbye.
```

old.txt

```
>${c:old.txt} -replace 'is (red|blue)','was $1' > new.txt
```

```
Hello there.
My car was red. Your car was blue.
His car is orange and hers is gray.
Bob's car was blue too.
Goodbye.
```

new.txt

Arrays

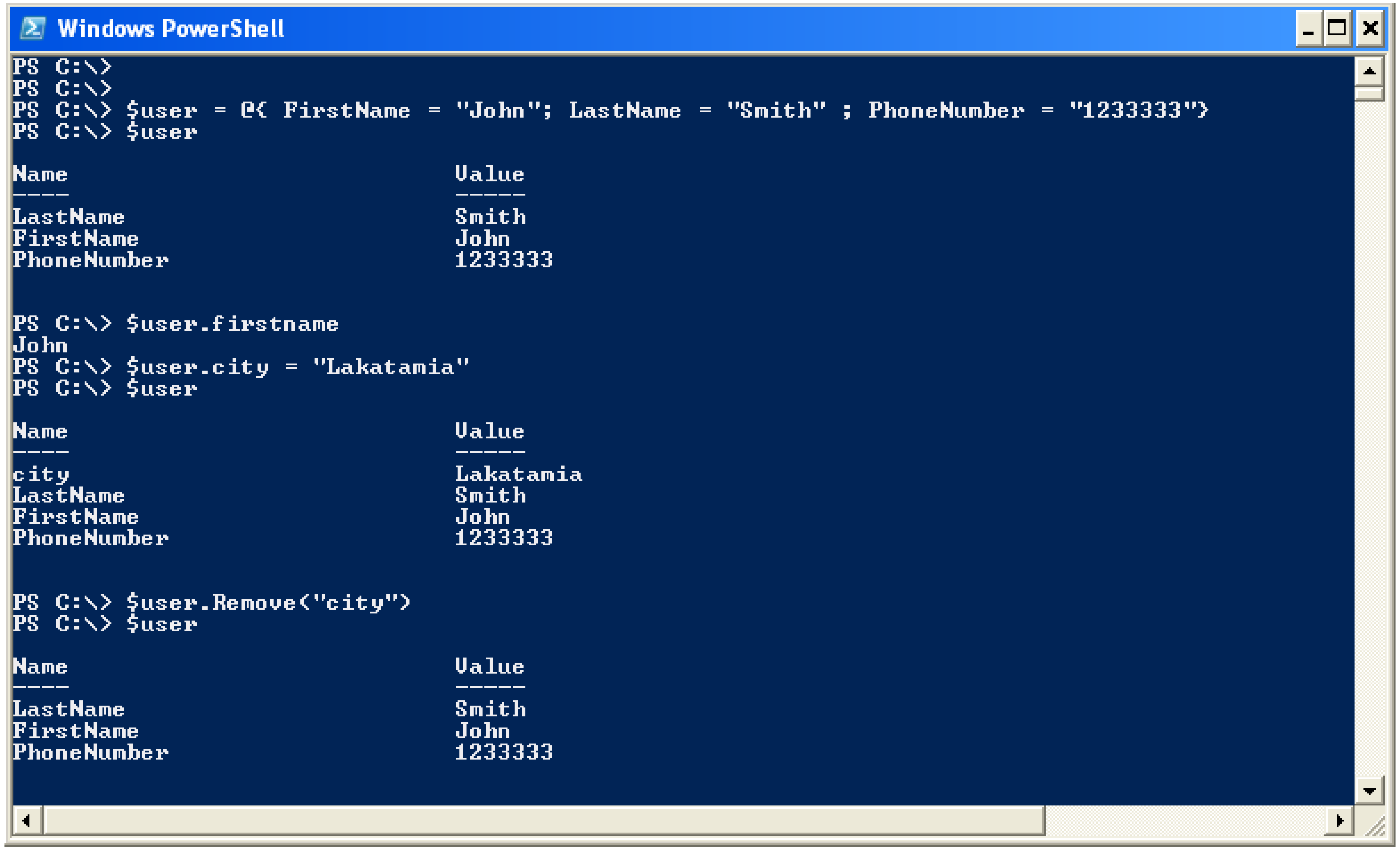
- Creating and Working with Arrays

```
> $a = 1,2,3,3,1,5
> $b = 3,3,3,3
> $a = $a+$b
> $a = $a * 3
```

- Arrays of Objects

```
>  $a = 1,2,"a string",4,5
> $a.GetType()
> $a[1].GetType()
> $a[1].GetType()
```

Hashtables (as records)



Windows PowerShell

```
PS C:\>
PS C:\>
PS C:\> $user = @{ FirstName = "John"; LastName = "Smith" ; PhoneNumber = "1233333"}
PS C:\> $user

Name                           Value
----                           -----
LastName                       Smith
FirstName                      John
PhoneNumber                    1233333


PS C:\> $user.firstname
John
PS C:\> $user.city = "Lakatamia"
PS C:\> $user

Name                           Value
----                           -----
city                           Lakatamia
LastName                       Smith
FirstName                      John
PhoneNumber                    1233333


PS C:\> $user.Remove("city")
PS C:\> $user

Name                           Value
----                           -----
LastName                       Smith
FirstName                      John
PhoneNumber                    1233333
```

Flow control
If / Elseif / Else

- *Syntax*

if (condition) {statement} elseif (condition) {statement} else {statement}

- *Example*

```
if ($x –gt 100)
{
"It's greater than one hundred"
} elseif ($x –gt 50)
{
"It's greater than 50"
} else
{
"It's not very big."
}
```

Loops

THE WHILE LOOP

- Syntax

```
While(condition=true)
   {statements }
```

- Example

```
$val = 0
while($val -ne 3){
$val++
"The number is $val"}
```

- Output

```
The number is 1
The number is 2
The number is 3
```

THE DO-WHILE LOOP

- Syntax

```
  Do { statements }
While (condition=true)
```

- Example

```
$val = 0
do {
$val++
"The number is $val"}
  while($val -ne 3)
```

- Output

```
The number is 1
The number is 2
The number is 3
```

Loops (continue)

THE FOR LOOP

- Syntax
```
For (Initialize;
Condition=true;
increment ){
statements }
```
- Example
```
for ($i=0; $i -lt 4 ; $i++)
 {
 ”The number is $i”
  }
```
- Output
```
The number is 1
The number is 2
The number is 3
```

THE FOR-EACH LOOP

- Syntax
```
 foreach ( variable in
loop_over ){
statements }
```
- Example
```
$c=0
foreach ($f in dir *.txt)
{$c += 1
 “The name of the $c txt
 file is $f”}
"We have total $c text
files in current dir“
```
- Output
```
 The name of the 1 txt file is
  new.txt
 The name of the 2 txt
 file is old.txt
 We have total 2 text
 files in current dir
```

FLOW CONTROL USING CMDLETS

- Foreach-Object
  - Similar to the foreach command
  - The automatic variable `$_` is used as the loop variable.
  - Usually comes with pipeline
  - Example:
    - dir *.txt | foreach-object {$_.length}
    - Finds the length of all the text files in the current dir

FLOW CONTROL USING CMDLETS (continue)

- Where-Object
  - select objects from a list
  - Comes with pipeline and the current pipeline element is passing to the _$ elemen
  - Example:
    - get-service | where { $_.Status -eq "Running" }
    - Finds all the running services

Functions

- Function Hello World
  - Create the function
    - function hello { "Hello world" }
  - Call the function
    - Hello
  - The function returns:
    - Hello world
- Returned values
  - Function returns the results of every statement executed in an array. Then you can get that array by running it and assign the results to a variable.
    - PS (4) > $result = numbers

Functions (continue)

- Function parameters
  - There is a default argument array called $args and contains all of the arguments to the function
  - If you want to specify formal parameters you can use the param statement
    - param($name="world“)
  - Personalized Hello Function
    - function hello { "Hello $args" }
    - Call the function
      - Hello George Tom
    - The function returns:
      - Hello George Tom

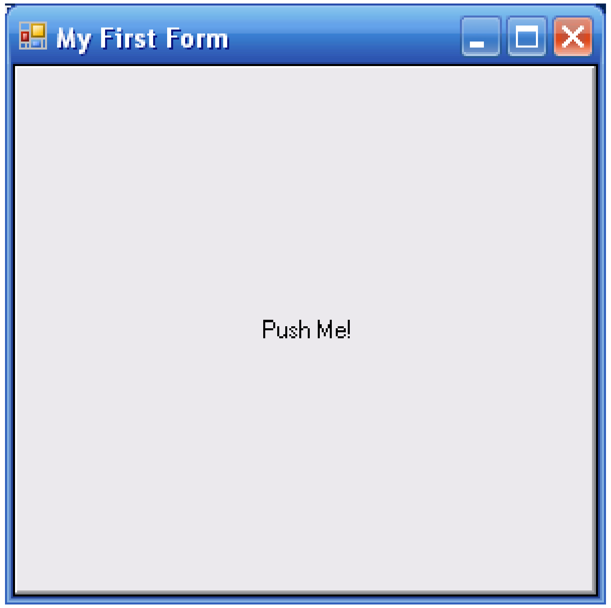
Scripts

- Create a script
  - Name a file with extension .ps1
  - Write the code the same way as you did in powershell
  - Write in the powershell ./filename to run it
- Scripts arguments
  - Just like functions (the default array $args)
- The param statement
  - specify formal parameters

Forms

- Powershell can load .NET libraries (assemblies) such as forms.
- Example

```
[void][reflection.assembly]::LoadWithPartialName
   ("System.Windows.Forms“)
   $form = New-Object Windows.Forms.Form
     $form.Text = "My First Form"
     $button = New-Object Windows.Forms.Button
     $button.text="Push Me!"
     $button.Dock="fill"
     $button.add_click({$form.close()})
     $form.controls.add($button)
     $form.Add_Shown({$form.Activate()})
     $form.ShowDialog()
```



Conclutions

- Powershell is easy to learn (syntax is similar to UNIX shells)
- Powershell is a powerfull command line tool
- Powershell provides windows with much needed scripting and administrative functionality
- Powershell still needs improvements (execution time)

Bibliography

- http://en.wikipedia.org/wiki/Windows_PowerShell
- http://www.microsoft.com/windowsserver2003/technologies/management/powershell/default.mspx
- Windows powershell in action (ebook)