&

The call operator (&) allows you to execute a command, script or function.

Many times you can execute a command by just typing its name, but this will only run if the command is in the environment path. Also if the command (or the path) contains a space then this will fail. Surrounding a command with quotes will make PowerShell treat it as a string, so in addition to quotes, use the & call operator to force PowerShell to treat the string as a command to be executed.

```
Syntax
      & "[path] command" [arguments]

Key:
    command   An executable filename (.exe), script or function.

   arguments  The call operator will only handle a single command.
              Any arguments may follow the called command.

              If you are calling a non-PowerShell command/utility then the
              and any arguments should be surrounded in quotes if needed
              due to spaces/long filenames:

              & "C:\batch\someutil.exe" test 123 "long path to\some file.tx
```

Precedence of commands:

```
     Alias > Function > Filter > Cmdlet > Application > ExternalScript > Sc
       Highest priority .................................... Lowest priorit
```

If you need to run a specific type of command which may not be the highest priority use Get-Command. For example if you have an external command called Ping and a function also called ping, normally the function will be run as it has higher priority, `Get-Command -commandType Application Ping` will return the external application instead.

Script blocks

Several commands, statements or expressions (a script block) can be stored in a variable: `$myVar = { Scriptblock }`
Then execute the script using `&`
`PS C:\> & $myVar`
or even without the variable:
`PS C:\> & {Scriptblock}`

This usage (calling a script block) is similar to using Invoke-Expression to run a set of commands but has a key difference in that the `&` call operator will create an additional scope, while Invoke-Expression will not.

Start-Process

If you need to capture a return value from the new process, store the output the process generates (stdout and stderr) and control the style or visibility of the Window, then use Start-Process which makes all those options available.

EchoArgs

EchoArgs is a simple utility that spits out the arguments it receives. This is very useful for testing, just replace the program name in your script with EchoArgs.exe to see which parameters are being passed.
EchoArgs is part of the PowerShell Community Extensions, but you can download a copy of EchoArgs.exe right here.

Dot-Sourcing

Invoking a command (either directly or with the call operator) will create a child scope that will be thrown away when the command exits. If the command/script changes a global variable those changes will be lost when the scope ends.
To avoid this and preserve any changes made to global variables you can 'dot' the script which will execute the script in the current scope.

```
PS C:\> . C:\scripts\myscript.ps1
PS C:\> . ./script64.ps1
```

Dot sourcing runs a function or script within the current scope.
unlike the call operator (&) which will run a function or script, within a separate scope.

```
PS C:\> $x=1
PS C:\> &{$x=2};$x
1
PS C:\>.{$x=2};$x
2
```

Examples

Run the script mycommand.exe:

```
PS C:\> & "C:\Program files\mycommand.exe"
```

or using a variable:

```
PS C:\> $program = "C:\Program files\mycommand.exe"
PS C:\> & $program
```

Run a command + options, note that we just pass the parameter as a separate string on the same line:

```
PS C:\> $program = "Get-ChildItem"
PS C:\> & $program "*.txt"
> Directory listing
```

Call one PowerShell script from another script saved in the same directory:

```
#Requires -Version 3.0
& "$PSScriptRoot\set-consolesize.ps1" -height 25 -width 90
```

Run a specific non-PowerShell command via Get-Command:

```
PS C:\> $myPing = Get-Command -commandType Application Ping.exe
PS C:\> & $myPing
```

Run a scriptblock (original value of variable is preserved):

```
PS C:\> $i = 2
PS C:\> $scriptblock = { $i=5; echo $i }
PS C:\> & $scriptblock
```

```
5
PS C:\> $i
2
```

With Invoke-expression the original value of the variable would be changed, because it runs within the same scope:

```
PS C:\> invoke-expression ' $i=5; echo $i '
5
PS C:\> $i
5
```

#You went away, And I wonder where you will stay, My little runaway, Run, run, run, run, runaway# ~ Dell Shannon

Related PowerShell Cmdlets:

. (source) - Run a command script in the current shell (persist variables and functions).
Run a PowerShell script - More examples of running scripts, .Bat, .vbs, dot-sourcing, elevation.
Invoke-Command - Run commands on local and remote computers.
Invoke-Expression - Run a PowerShell expression.
Start-Process - Start one or more processes, optionally as a specific user.
--% - Stop parsing input as PowerShell commands.
PowerShell Operators - SubExpressions Syntax.
bash equivalent: - in bash `&` starts a background process, in PS use Start-Job for that.
Keith Hill's blog - Command parsing mode vs Expression parsing mode.