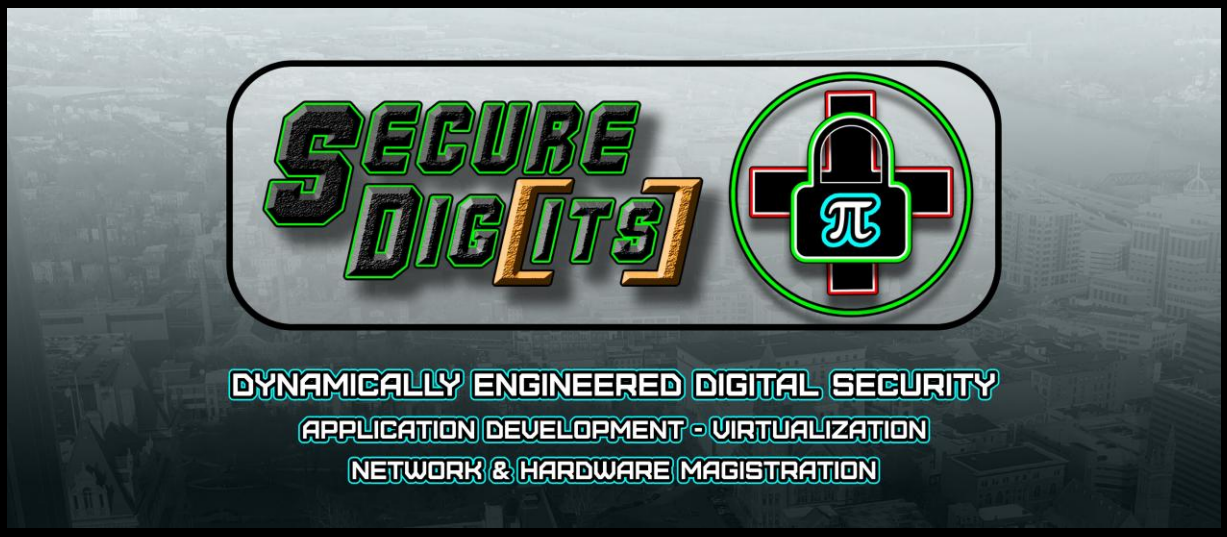
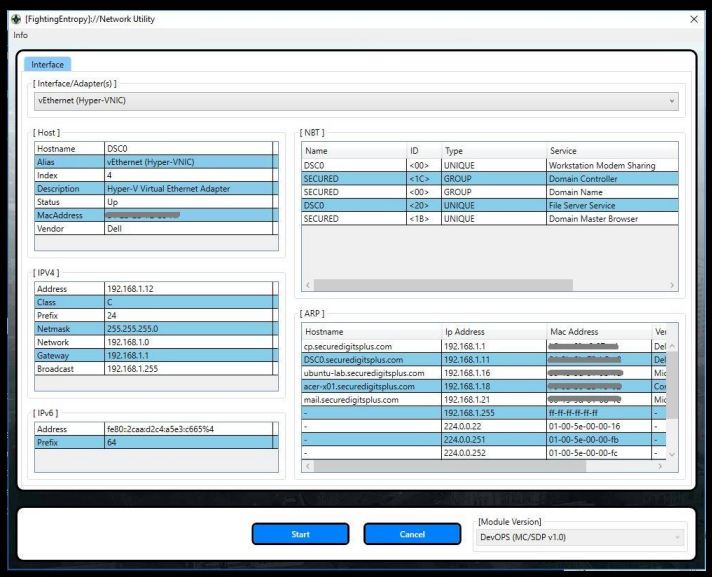


A Deep Dive: [PowerShell, XAML/WPF, Classes, Functions, and Networking]

By: Michael C. Cook Sr. [Developer + System Engineer]

[Introduction : Graphical User Interface]



Today, the world is full of a wide variety of programming choices.

Not everyone that tries to learn how to program, is successful in their endeavor.

Now, because it is tough to do right, AND, because it can be highly stressful to think about abstract solutions to real world problems...?

Having an example to work with doesn't hurt.

When it comes to learning how programming in general, works...?

Well, having an obsession with problem solving is a pretty good first step.

However, that alone, won't be *enough* to build tools with high build quality, capability, or desire to reuse.
Second step, is *experience*, from *working on things that have a sense of practicality to it*. That means, real world use case scenarios.

If an idea is (*practical/beneficial*)...? Then, it is probably worth investing (*time/resources*) into creating it, or at the very least, understanding it. That is what research and development is all about... *not* skipping firmware updates, *not* pretending like using the same password for 5 years isn't a bad idea, staying up to date on updates, security packages... That is a lot of work.

Suffice to say, **1) drive** and **2) inspiration** go a long way, if you really are taking a journey deeper into application development.

Now, there are a number of other caveats that I wish somebody would've told me when I first started...

Like, "*Hey. You're gonna spend thousands of hours doing this once you get going pal, so…? Buckle up.*" That's not to say that I haven't learned a lot. Because. I have. Anyway… in this lesson, I'm going to talk about a range of things that are on the **[Intermediate-Advanced]** side… So, if you're relatively *new to programming* or **PowerShell**…?

Then, this may seem a bit daunting to follow. However, I am certain that anyone can learn **something** from it.

The subject matter (at hand/pictured above), is a "*Graphical User Interface*" concept, which is highly useful for assembling a program that does something specific, or general. The picture above, serves as a face to this network utility that I had to create, in order to orchestrate **DC Promo**. I won't talk about the **DC Promo** tool, yet.

What I *will* talk about in this portion, is the function "**Get-FENetwork**", and… the many embedded classes it contains.
Before I do that however, I will discuss the perilous journey I had to undertake… in order to learn how to build this.

[Foreword : History with Graphic Design]

I once had an opportunity to learn how to program many years ago when I was a kid, but gravitated toward design work. Whether it was *designing websites* or *building levels* for **Quake III Arena..?** Well, I had a future in graphic design in mind.

This is a Q3A map I made, named "*Insane Products*", which featured a number of themed map elements like brightly colored slime pits covered by transparent glass, smooth game-play, and aesthetics that pop out and practically smack you across the face with a real wow factor. (Doesn't actually smack you across the face…)

I finished this map when I was attending **ITT Technical Institute** for Drafting and Design, in **2006**. I was given a preliminary course on programming in **Visual Basic.Net** at that school; but- wasn't very interested in it.

For my capstone project, I made a **(scene, animation, model, video)** in **3DSMax** where I made the following 3D models: **[command center], [supply depot], [barracks], [minerals], [marine], [SCV]…** The animation involved an SCV building a barracks and supply depot, the other SCV's mining, the barracks lifting off and moving, landing, all while the marine patrols around the base.

Probably sounds like **Starcraft II**, doesn't it? Well, guess what..? This capstone project that I (*made in 2006/just described*) was about *3 years before* they made **Starcraft II**. So, **Blizzard** hadn't yet made a 3D version of **Starcraft.** When I say all of this out loud, I begin to realize how old I am getting… Anyway look, I wish I still had that video… but I have no idea where it could be.

Back in **2006**, **YouTube** wasn't a commercial success yet, so, uploading random videos for safe keeping wasn't widely available.

What I do have, are these pictures from the map file that **[..::LvL]** still has.
If you've never heard of **[..::LvL]**...?
Then, I'll let you in on a little secret...

It was a website where **Quake III Arena** maps were made by people in the community.

Maps that were *reviewed* by the community...?
Played by the community.

Tig (ran/still runs) things there, I think.
I reviewed a number of maps on the site.

Anyway, my association with **PlanetQuake & [..::LvL]** began back around the year 2000.

We're talkin... **World Trade Center** was still standing... **Disturbed** was busy dropping plates and releasing '*The Sickness*', **iPods** *hadn't been invented yet*, **iPhones** came *long after* **iPods**, **Napster** was really in full swing, **Sean Parker,** (*kicking ass/taking names*)...
...and *many* people falling into the void...
...in one ***particular*** **Quake III Arena map...**
https://quake.fandom.com/wiki/Q3DM17:_The_Longest_Yard

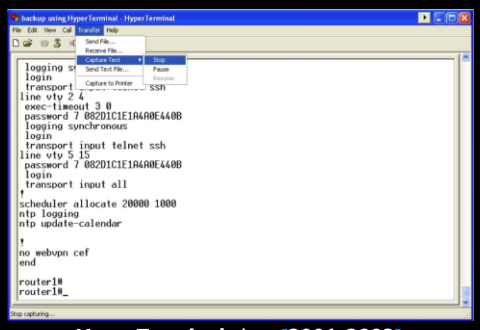
Because it was a **q3demo** map, that's why.

Back to more recent times. I held a number of positions that involved some of my technical experience, but weren't design related... I hadn't been sure that graphic design was the right fit. I went back to school around (**2008-2009**) for (**MCSA/MCSE**) training.

A number of years went by, being a [computer/network] (technician/administrator/engineer), in addition to having solved many other technical problems in some creative way... something prompted me into going back in time to review things I used to do a long time ago... such as Unix, Putty, Networking, Active Directory, that type of stuff.

I also began to revisit many of my print-based-graphics days where I made print documents with notepad and HTML tables, and then loaded up the web browser and then bam. Print that. I found myself having a higher amount of interest in **PowerShell** year after year, because... let's face it.

This **PowerShell** thing looked like it'd make my life a lot easier... I saw that I could theoretically do some serious (*damage/development work*) with it... Decided that it'd be the thing that'd get me to start programming.
Like... actual bonafide programming.



HyperTerminal circa **[2001-2003]**

Anyway, **PowerShell** has always reminded me a lot of **HyperTerminal** from back in the day, I used **HyperTerminal** to connect to **Cisco** *routers & switches* in high school, setting up VLANS, subnets, and such. The content of the picture to the left looks accurate.

My vocational school had this equipment, we had to use serial cables to connect to routers and switches and do ***many*** labs. I did fairly well in that class, and it resulted in me being offered a scholarship, but I declined.

As it turns out... **PowerShell** can be used in place of **CMD**, **Putty**, **HyperTerminal**, **SSH**, **Python**... Which means, its 1) *potential*, 2) *power*, 3) *ease of use*, 4) *compatibility*, 5) *scalability*, and 6) *usefulness*... not to forget, 7) *security*...? It's just *way* too much to ignore.

__All__ of *those things* prompted me to say to myself… "*Hey. I could totally brush up on everything with **PowerShell**. Couldn't I?*"
Nobody was standing around within earshot to respond "*Yeah man… you totally could. Might be fun.*"

I remember many years ago… an instructor I once had, told me how powerful this "**Monad/PowerShell**" would be…
He said, "*It's gonna change everything, pal.*"
I said, "*Yeah…? Everything…?*"
He's like, "*Yeah man. Just you wait and see…*"

I only wish I had listened to him more carefully, because he was right… **PowerShell** did change everything.

At some point, I made this realization when I was intrigued enough with **PowerShell** in (**2018**), AND, the number of things that could be done with it…? I began to catch up on, AND review, educational material I had once studied. Old (**MCSE/MCSA**) material, **Virtualization**, **Network Administration/Security**. Then, I started to connect the **Graphic Design** stuff with all of that… and bam.

Spending the last 2 years researching and developing uncommon methods of using **PowerShell**… in ways that no sane man has done yet. Willing cool ideas into existence, so to speak. Cause. *All* of my technical abilities with a computer, or a network…?

Simply put… they all seemed to be much more manageable via this whole **PowerShell** phenomenon.
PowerShell actually made it seem a lot more fun and cool… the question was, how do I learn about something *so advanced*..?

[Tangent : PowerShell & Dark Mage Wizardry]

In order to ***really*** embark upon this journey…? Challenging myself to learn this sophisticated scripting language… Well, I thought to venture into the darkest chapters of the online **PowerShell** sacred dungeon/library archives…
It's called… **Microsoft TechNet** and it's @
https://gallery.technet.microsoft.com/scriptcenter

If you didn't know… this place I just mentioned…?
Happens to be where some of the lightest and darkest mages of all time, have left many pages worth of spells… for both **good** AND **evil** purposes.

You might think I'm being ridiculous or over the top, but… look at this warrior to the right. She's got some scars on her face from the war she's been fighting against these dark mages. Ask anybody at this address…

Microsoft Corporation
1 Microsoft Way,
Redmond WA, 98052

…if any dark mage wizardry occurs within their wake.
They will say, "*No, yeah… **dark mage wizardry** does occur within our wake. It's why we came up with this awesome lookin' avatar.*"

Dark mage wizardry *__also__* goes by the name of:
cybercriminal activity, identity theft, and douchebaggery.

Any educational resource eventually becomes a morality battleground, so to speak.
With one new cool idea that could change the world for the better, there is another that says "*No way, Jose…*"

One side…? A plethora of positive Pam's.
The other…? A number of negative Nancy's.

I'm not saying that things at battleground **TechNet** are made with malice in mind, but…
…there is definitely some **dark mage wizardry** going on within **Microsoft's** wake…

Now, in order to *combat* this **dark mage wizardry**…?

You need a proper master to train you.

When I see the picture of the character above, I'm not thinking "bad guy".
I'm thinkin', "a war torn veteran mage that's seen it all, time and time again…"

One who doesn't like the **dark mage wizardry** at hand…?
But, still willing to fight.
Hand on her hip.
Waiting for somebody to throw down the gauntlet.

Anyway, look. Legend has it, if you have the heart and mind of a worthy warrior…?
Then you'll be able to pull the **Excalibur** from the stone.
That's where **Merlin** comes into play… I'm not saying that's **Merlin** up above by the way…

But, **Merlin**, is the wizard supreme. Led a pack of heroes throughout the **Crusades**. Managed to make it back alive, to splendor in his heightened magic capabilities as guardian of planet Earth… Now, keep in mind, **Merlin** could be a code name.

I like to look at this **PowerShell**, as if I **could** be **King Arthur** himself…
…finally removing the **Excalibur** from the enchanted stone of **Camelot**…

Merlin, the wizard supreme… the war-torn, veteran mage that's seen it all, watching from a distance…
…keeping a careful, AND watchful eye over this process…
…of a boy named **Arthur** who grew up as one among the commonwealth…
…becoming the man who would pull the sword from the enchanted stone…
…because it was his fate, to be the best king that *ever* lived.

Now, that probably sounds *wicked* intense. Awe inspiring, even.
Way more intense than you were expecting, probably…
However, these are the terms that need to be laid down, when you're comparing **PowerShell** to the **Excalibur**.
Earning accolades and such…
…fighting a war against dark mage wizardry.

While there probably are an infinite number of ways that you can use **PowerShell**…?
You don't want to go recklessly adding yourself to the list of people that **Merlin**'s gotta keep a watchful eye over…
…for safety reasons…

Cause. You might've pulled the **Excalibur** from the enchanted stone of **Camelot**, and not even know it…

Now look. Whether you're comparing **PowerShell** to the **Excalibur**, or not…? (*Microsoft/Merlin*) knew how much power (*PowerShell/Excalibur*) was capable of, since its inception.
Seriously.
Why do you think they made it…?
It's because… they needed an *extremely powerful* AND *versatile* tool that didn't need to be *compiled*, in order to *run*.

With **PowerShell**, well… you might get just-in-time compilation, but that's it…
…you just, *highlight the text*…? **Press F8**…? Bam. There it is. Pretty simple stuff.

Sometimes… you don't even have to **press F8**. You can dot-source a file…
…and that counts as like a programmed (select text -> **Press F8**) … bam.
There's your output.

Regardless, because of how much it is capable of…?
Well, there is a lot of responsibility involved in the handling of this **PowerShell**.

So, if you're going to learn how to use it…?
Then, make sure you use it responsibly…
That's all I'm gonna say about that.

[Subject Matter : Network Utility ("code-behind")]

So, here is a listing of all the "*code-behind*" components of the **GUI**.
This doesn't include the "*code-front*" **XAML/Window** object... but these need to be discussed first, in order to understand the XAML riggings. I will go over that in another chapter, because **XAML** is a beast in and of itself.

```
#/------------------------------------------------------------------------------------------------\#

[ 0] Function Get-FENetwork    {}
[ 1] Class _VendorList         {}
[ 2] Class _NbtReferenceObject {}
[ 3] Class _NbtReference       {}
[ 4] Class _NbtHostObject      {}
[ 5] Class _NbtTable           {}
[ 6] Class _NbtStat            {}
[ 7] Class _V4PingObject       {}
[ 8] Class _V4PingSweep        {}
[ 9] Class _V4Network          {}
[10] Class _V6Network          {}
[11] Class _NetInterface       {}
[12] Class _ArpHostObject      {}
[13] Class _ArpTable           {}
[14] Class _ArpStat            {}
[15] Class _NetStatAddress     {}
[16] Class _NetStatObject      {}
[17] Class _NetStat            {}
[18] Class _Controller         {}
[19] Class _DGList             {}
[20] Class _FENetwork          {}

#\________________________________________________________________________________________________/#
```

Granted, I probably don't need *this many classes* to do what I'm doing, but- I like to keep groups of variables together.
A strange side effect of this, is that the process remains quite optimized.

The main (**function/namespace)** is **Get-FENetwork**. I previously had these classes *outside* of this **function**... However, that is a long list of (*objects/files*) to maintain and keep track of, especially when all of them correspond to networking functions. Embedding them together within the same class is a great idea in this case, as it can allow me to package them together in a similar manner to a folder full of files. That's the idea behind a module.

Anyway, when **classes** are embedded within a **function**, it provides a **scope** for those classes to **exist within**...
...so, if the **namespaces** that a **class** or **type** needs aren't loaded **FIRST**-? Then you'll run into *problems*.

[Glossary : Common Terms]

What is a function...?

A **function** is essentially an executable, whether it is a *code-block*, *program*, *or a command*.
Variables are *technically* **functions** as well.
I'm not saying that **variables** definitely *are* **functions**, but... I don't see how they can't be called that.

(*Functions/Variables*) are ALSO used in mathematics to determine property values, as well as in theorems or algorithms.
In Computer Science, you'll get a lot of exposure to that.

What is a class, anyway?

A ***class*** is a *special function*.

One that has an extra added bonus, of being able to carry out additional functions within itself.
Functions can also do this too, but to be REREFERENCABLE...? From *outside scopes*..?

It is better to use classes for this, because that's when you can ***really*** play with fire. (*Don't actually play with fire, it's a metaphor...*)

Referencability is not exactly a word in the dictionary, although I believe it should be, because an objects ability to be referenced, OR, re-referenced, should be called as such. If I cast a **function** to a **variable**, then the **variable** will **only** be the **output** of that **function**. But, if I cast a class to a **variable**, I can call upon further (*actions/methods*) from the **variable**. That means, rereferencability

In simpler terms, the **function** can do something once. The **class** can do something a bunch of times, and keep those things *clean* and *accessible*. Think about how most people like their food, home, or significant other to be clean and accessible. Why wouldn't they want their (*input/output*) to be *clean* and *accessible* too?

Anyway, functions can return class types that allow for further method execution, but in the end those are *still* CLASSES.
Classes are essentially *fancy* **functions**.

I can imagine, some people might be thinking. "*Yeah, well... just how fancy **are** these classes, Michael...?*"

[Skit : Fancy]

Fancy, like a guy in a limo, having ***his*** limo driver drive right up next to you in ***your*** limo...
...and in the most nonchalant manner, rolling his window down...?
...respectfully asking you if you just so happen to have any "**Grey Poupon**" on hand... that's how fancy.

You would think that this guy would give up after a while, but no...
Guy asks you every day, multiple times a day...
"*Excuse me sir,*" clears his throat mid-sentence, "*Do you have any... **Grey Poupon**..?*"

You respond sharply "*No bro. I don't.*"
You take a moment to let out the frustration of having to deal with this guy repeatedly...
"*Listen pal, you keep driving up to me... asking me the **same question**, **over** and **over** again.*"

He responds, "*Yeah..? So..?*"
You say, "*It's annoying. What gives...?*"
He says, "*Look, don't get mad at me bro. You look like the type of dude who's fancy, sittin' there in a sick limo... wearin' a top hat. That's why I stop and ask you if you have any **Grey Poupon**, as a token of gratitude and respect.*"

You mull over what this guy just said... "*YEAH, but 5 times a day? Really? C'mon bro. It's ridiculous!*"

Guy stares at you for a moment.
Doesn't say anything but makes a hand gesture to his driver that says "Alright, he's got no ***Grey Poupon***... let's go."
But then, after a few seconds, he must've told the driver to stop and back up to your window, cause that's what happens.

Guy's slightly upset. "*Listen man, you're pretty **obnoxious**..!*"
You're confused, "*Me..!? How am I **obnoxious**..!? You're the one that's being obnoxious, pal..!*"
You hold up your hand with all fingers and the thumb too... "*...that's the fifth time you've shown up and asked me today..!*"

The guy responds in an unexpected way... "*Nah man. I'm not the one who's being obnoxious. You are. I literally ***just*** complimented you...? And you had the nerve to tell me that I'm ridiculous. Says a lot about ***you*** man... Probably wouldn't invite you to ***my*** birthday party...*"

Anyway, now you feel bad about hurting this guy's feelings... you didn't realize he would get so emotional about being told off.
He literally just threw down the gauntlet with the birthday party thing... now it's serious.

Guy continues, "*You know... what if I never came back to ask you if you have any **Grey Poupon**? I'll bet that you'd feel pretty sad about me not showing up anymore. There'll come a day where you remember me, asking you if you just so happen to have some of my favorite mustard... You'll think to yourself 'where did that guy go?' And in that moment, you'll miss me bro. So, don't be mean.*"

You see the sense in what was said to you... but, you're still conflicted. "*Alright, what am I supposed to do, apologize to you now?*

*Cause you wanna keep stopping ***your*** limo next to ***mine*** like some type of 80's commercial? Asking me if I have any **Grey Poupon**…? Annoying me…? Several times a day…? Same answer every time…?"*

Guy responds… "*It's because I'm fancy, isn't it? I knew it…*"
You respond "*Well, no. Has nothing to do with you being fancy. It is as if you simply forgot that you asked me already, and you have to ask me ***again***, just to make sure. You're not putting 2 + 2 together, that I've never actually had any **Grey Poupon**… have I?*"

He says, "*I do that because that's what being fancy is all about. I remember each and every time you say no. I could buy my own **Grey Poupon**… but, I still show up and ask **you**, because I respect another guy sittin' in a limo… wearin' a top hat… and a monocle. And I think that's cool, even if you do look like the monopoly guy a little bit. So, maybe you'll eventually think 'You know what, **Grey Poupon** dude's probably gonna stop and ask me for some later today… Suppose I just went ahead, picked some up, and then totally surprised this guy for a change…?'*"

You respond, "*Well now, why would I do that…?*"
He says, "*I don't know, maybe I have something you might need.*"
You say, "*Look chum. I appreciate everything, but… you're making this far more dramatic than it needs to be. If it'll make you happy, I'll pick up some **Grey Poupon**, and I'll carry it around with me everywhere I go. That way, next time you ask, I'll have it. Just for you. *"

He says, "*Nah man… It's too late now. I don't feel like asking you anymore… Take care.*"

[End Skit : Fancy]

Now look. These classes might not be ***that*** (*fancy/emotional*), where they show up 5 times a day, respectfully asking you if you happen to have any ***Grey Poupon***…? But hey. **Style** and **finesse** go *a long way in improving efficiency*, and *fancy classes* help make that happen… whether the classes come with a story or not, they're still **pretty fancy**.

In **C++** I believe *classes* are called *structs*, but I'm not positive.
I'm still trying to finagle a way to use *structs* in **PowerShell**.

Anyway, classes have *properties* that *functions* typically **do not** have.
Classes are wrapped in square brackets like `[This].`
Data types are ***also*** wrapped in `[Square Brackets],` but the use of the (double colon/::) and "new" indicates a new instance of a class. If you attempt to use an incorrect `[Data type/Class],` you will get an error that states that the `[Variable/Object]` you're working with could not be (converted/exchanged).

In some cases you can use a specific class as a data type, but during the property declarations, it's best to use generic system types like `[String], [UInt32], [Char], [Hashtable],` or `[Object]. [Object]` is a big one, because that works for everything.

The properties I'm referring to are *also* methods and constructors.
Same thing as **ScriptProperty** or **CodeProperty**, I believe.
They can be embedded within functions and vice versa.

However the ***scope*** from which that *embedded* (***class/function***) can be used, *remains within that container* (***class/ function***).

If that is confusing, don't worry. It is. The main trick is, learning how to deal with said confusion in an orchestrated way, AND, forcing things to work correctly. In order to do this, you'll be tempted to experiment and figure out where you can do certain things, and where you can't.

What is Regex? Is that like… advanced?

Regex is short for **Regular Expressions**. And, yeah… **Regex** is pretty advanced.
Regular expressions are a matter of using (*mathematics/patterns*) to *filter* through *data*, *communications*, *algorithms*, *books*, you name it… it is essentially a **GREP** command from back in the day of (Unified Information and Computing Service/UNICS) .

It is able to access the registers of the memory directly apparently, which is why it is as performant as it is.

(*Side Note: You can tune raw **PowerShell** code to work as fast as compiled **C#** code using **Regex**, if you know how to use it.*)

So, if you **match** up a **letter** in a sample string of text, then you get a **match count**.
You can match other tokens like words, symbols, numbers, or even some arrangement of all those things combined.
Anyway, if you **match** up a **pattern**, then you can filter out the rest of the data, whereby parsing what you want.

I use a fair amount of **Regex** in this lesson. So, whether you are just starting out with it, or you're a master warrior extraordinaire at **Regex**...? You should still be able to follow along.

```
#/------------------------------------------------------------------------------------------------------------\#
[ 1] Class _VendorList         {}
#\____________________________________________________________________________________________________________/#
```

The name of the first class is **[_VendorList].** It is good practice to use names for **variables**, **functions**, **classes**, **methods**, or **constructors**, in ways that describe themselves and are revealing about what that object is, or does.

You wouldn't call your buddy "**Barbie Doll**", or "**Nancy Boy**" if his name is **Jim**... right?
Alright then.

You can call your buddy **Jim** those names if you ***really*** want to, but... make things easier on yourself.
If you try to talk to OTHER people about your buddy **Jim**, but you refer to him as "**Nancy Boy**", then people are going to be confused. They might say "Which **Nancy Boy** are you talkin' about?"
Then you'll have to start your story over and say "my buddy **Jim**"...

Same thing goes for functions and classes. They need to know what to look for, and they need the correct name.
So, name your (**functions/classes/variables/methods**) so that they are relevant to the output.

I haven't been an application developer over the last 20 years... but, I have used a number of them and have had to do (**light programming/config file editing**) over the years, and I have seen many programmers name a function something that is incredibly cryptic and hard for someone that isn't the software developer to repair.

Sometimes that process is referred to as **obfuscation**.
Other times, it is just *questionable code that someone slapped together*.
The difference between the two can be incredibly difficult to discern sometimes.

Anyway... the class **[_Vendorlist]** seeks to pull a list of (hex = vendor) codes from the same codebase as **nmap**.
At one point I had edited the data so that the vendor list was 400kb smaller than the original file, but that was a pain in the neck to edit or make changes/updates to.

[_VendorList] is going to have several properties, but the only property that matters externally, is the **GetVendor()** method.

So, if I were to hand a $MacAddress like "00-50-56-C0-00-01" to the method...

```
#/------------------------------------------------------------------------------------------------------------\#

$MacAddress = "00-50-56-C0-00-01"
$This.GetVendor($MacAddress)

VMWare

#\____________________________________________________________________________________________________________/#
```

...then the output will be **VMware**.

Here's an examination of the class **[_VendorList]**

```
#/------------------------------------------------------------------------------------------------------------\#

Class _VendorList # Obtains hardware vendor list to convert MacAddress to correct vendor name
```

```
{
    Hidden [Object]    $File
    [String[]]          $Hex
    [String[]]        $Names
    [String[]]         $Tags
    [Hashtable]          $ID
    [Hashtable]       $VenID

    _VendorList([String]$Path)
    {
        Switch ([Int32]($Path -Match "(http|https)"))
        {
            0
            {
                If ( ! ( Test-Path -Path $Path ) )
                {
                    Throw "Invalid Path"
                }

                $This.File = (Get-Content -Path $Path) -join "`n"
            }
            1
            {
                [Net.ServicePointManager]::SecurityProtocol = 3072
                $This.File = Invoke-RestMethod -URI $Path

                If ( ! $This.File )
                {
                    Throw "Invalid URL"
                }
            }
        }

        $This.Hex            = $This.File -Replace "(\t){1}.*","" -Split "`n"
        $This.Names          = $This.File -Replace "([A-F0-9]){6}\t","" -Split "`n"
        $This.Tags           = $This.Names | Sort-Object

        $This.ID             = @{ }
        ForEach ( $I in 0..( $This.Tags.Count - 1 ) )
        {
            If ( ! $This.ID[$This.Tags[$I]] )
            {
                $This.ID.Add($This.Tags[$I],$I)
            }
        }

        $This.VenID          = @{ }
        ForEach ( $I in 0..( $This.Hex.Count - 1 ) )
        {
            $This.VenID.Add($This.Hex[$I],$This.Names[$I])
        }
    }
}

#\_____________________________________________________________________________/#
```

Now, this class should immediately cause someone to ask questions about what it is doing, especially when they see the first block of variables.

```
#/------------------------------------------------------------------------------------------------------------\#

Hidden [Object]    $File
[String[]]          $Hex
[String[]]        $Names
[String[]]         $Tags
[Hashtable]          $ID
[Hashtable]       $VenID

#\____________________________________________________________________________________________________________/#
```

These here, are *properties*.
The first *property* is **$File**
Above, the (*property/variable*) is accessed via **$File**

Once the **_VendorList([String]$Path){..}** method is entered...?
Then, the variable **$File** will be accessible via **$This.File**.
Pullin' the ol' switcheroo on ya.

That one reason alone, is why classes are far more flexible than naked functions.
It's as if you are setting up strings that go through wormholes and back.
At any rate, let's not get lost in charades that involve manipulation of the time-space continuum...
Eyes on the prize.

When I first review a block of code, I look at the (*variables/properties*) so that I can keep those values or placeholders in mind as I review the sample.

In this case, **$File** is an **[Object]** placeholder for the string...
So, we're likely to load a (**UNC/Universal Naming Convention**) path or a file system path.

```
#/------------------------------------------------------------------------------------------------------------\#

Switch ([Int32]($Path -Match "(http|https)"))
{
    0
    {
        If ( ! ( Test-Path -Path $Path ) )
        {
            Throw "Invalid Path"
        }
        $This.File = (Get-Content -Path $Path) -join "`n"
    }
    1
    {
        [Net.ServicePointManager]::SecurityProtocol = 3072
        $This.File = Invoke-RestMethod -URI $Path

        If ( ! $This.File )
        {
            Throw "Invalid URL"
        }
    }
}

#\____________________________________________________________________________________________________________/#
```

This is a switch that I will likely rewrite.

It is using Regex to test a pattern on the input string, and while it does work... it leaves the possibility of problems happening.

Regardless, the input string is being checked for either "http" or "https".
With Regex, I probably could've used "(http[s*])"... but the pipe symbol in the middle of "(http|https)" allows for either/or.

If the string does not match, then the file system will test the path of the object.
If it is not there, it will throw the error.
If it is, then the property, **$This.File** is then provided the content of that file.

If the string does match, then the shell will set the download security protocol to **TLS 1.2**, and then commence with an **Invoke-RestMethod** command... whereby pulling content from a URL.

```
#/------------------------------------------------------------------------------------------------------\#

$This.Hex            = $This.File -Replace "(\t){1}.*","" -Split "`n"
$This.Names          = $This.File -Replace "([A-F0-9]){6}\t","" -Split "`n"
$This.Tags           = $This.Names | Sort-Object
$This.ID             = @{ }

#\______________________________________________________________________________________________________/#
```

This block here is making adjustments to the content using Regex.
It is best to manipulate content when it is all one contiguous string, for performance reasons.

When the content is split into lines, then the Regex function has to work harder and more often, resulting in more time being needed to accomplish the same goal.

$This.Hex is pulling all of the Hex codes from the sample input.
$This.Names is pulling all of the vendor names from the sample input.
$This.Tags is taking all of the names and then sorting them alphabetically.
$This.ID is setting up a **[Hashtable]** so that the components of each **[String]**/line can be *readded*.

```
#/------------------------------------------------------------------------------------------------------\#

ForEach ( $I in 0..( $This.Tags.Count - 1 ) )
{
    If ( ! $This.ID[$This.Tags[$I]] )
    {
        $This.ID.Add($This.Tags[$I],$I)
    }
}

#\______________________________________________________________________________________________________/#
```

Hashtables are great for storing data in a way that allows for immediate reaccessiblity.
I prefer to use arrays to collect objects, however, sometimes the hashtable (collection/recollection) process is faster.

[Analogy : Hashtable vs. Array]

Suppose you're in charge of delivering a bunch of packages to each and every floor on a particular building...
These packages cannot be left there without someone being there to receive the package.

Now, if you can take all of the packages with you, then you could drop them all off when going from top to bottom... that's what an array does. But, numerical index may not be the token you want to (iterate/scale) with...
So, you might want to iterate with the name of something. Maybe for some reason you want to start with the T's first or something...

Enter, the legendary hashtable.
With a hashtable, it's very different than an array.
An array, you enter a number, it retrieves that slot. An example of an array is (*1,2,3,4,5,6,7/1..7*)

With a hashtable, you enter a key, it retrieves that slot.
Keys can also be numbers, but typically keys are word-based.
Keys can also have a space or hyphen, but they need to be wrapped in quotes if you use them.

Anyway, with a set of keys, you're not going from top to bottom based on numerical index.
You're probably doing so in *alphabetical order*.
Maybe it's not even in alphabetical order.
Maybe you gotta wait for Frank, Louis, Bob, Joe, Walter, Ed, Bernie, or Steve... and in that order too.

Well, you just say the name of the person that needs their package, and then they're actually at the lobby at that exact moment that you say their name.
That's the idea behind a **[Hashtable].**

With an array, it has to destroy and recreate the entire array in order to add an item to it.
So if you get caught up in a loop, the loop has to iterate all the way through for each name on the list, which takes a lot of time.

A **[Hashtable]** doesn't work that way... once the value is stored, you just call that property name, and bam. There it is.
Which is why learning when and where to use each **[Tool/Data Type]** is vital.

[End Analogy]

$This.ID is holding a numerical value for each unique entry in the list of vendor names, for rereferencing.

```
#/------------------------------------------------------------------------------------------------------------\#

$This.VenID          = @{ }
ForEach ( $I in 0..( $This.Hex.Count - 1 ) )
{
    $This.VenID.Add($This.Hex[$I],$This.Names[$I])
}

#\____________________________________________________________________________________________________________/#
```

Now, this part here says to me that some of the logic isn't being used, but that's ok.
Sometimes I will leave logic that accomplishes some other goal, within the code that produces the correct result.

In this case, the sorting process for **$This.ID** isn't being used. It isn't impacting the performance by much to leave it in there. Now, all of that code thus far, allows the vendor name to be discoverable from the **Mac Address**.

In the case of wanting to find devices on a given network that aren't supposed to be there...? It helps to know if there's a hidden "**Dell**" in your house or business...

[Host]

Hostname	DSC0
Alias	vEthernet (Hyper-VNIC)
Index	4
Description	Hyper-V Virtual Ethernet Adapter
Status	Up
MacAddress	[redacted]
Vendor	Dell

```
#/------------------------------------------------------------------------------------------------------------\#
[ 2] Class _NbtReferenceObject {}
#\____________________________________________________________________________________________________________/#
```

Now I'll talk about **NBT stat**, which was necessary for building a new version of **DC Promo**...
Not going to get into the **DC Promo** tool quite yet... but, **NBT stat** parsings required a few classes.

```
#/------------------------------------------------------------------------------------------------------------\#
```

```
Class _NbtReferenceObject # Object for the Nbt reference table
{
    [String]      $ID
    [String]    $Type
    [String] $Service

    _NbtReferenceObject([String]$In)
    {
        $This.ID, $This.Type, $This.Service = $In -Split "/"
        $This.ID = "<$($This.ID)>"
    }
}

#\_____________________________________________________________________________________________/#
```

This one is rather quick and concise.
Every "**registered**" object that shows up with the "**nbtstat –N**" utility is parsed and objectified by matching values here.

```
#/-----------------------------------------------------------------------------------------------\#
[ 3] Class _NbtReference       {}
#\_____________________________________________________________________________________________/#
#/-----------------------------------------------------------------------------------------------\#

Class _NbtReference # NBT Reference table
{
    [String[]] $String = (("00/{0}/Workstation {4};01/{0}/Messenger {6};01/{1}/Master Browser;03/{0}/Me"+
    "ssenger {6};06/{0}/RAS Server {6};1F/{0}/NetDDE {6};20/{0}/File Server {6};21/{0}/RAS Client {6};2"+
    "2/{0}/{2} Interchange(MSMail Connector);23/{0}/{2} Exchange Store;24/{0}/{2} Directory;30/{0}/{4} "+
    "Server;31/{0}/{4} Client;43/{0}/{3} Control;44/{0}/SMS Administrators Remote Control Tool {6};45/{"+
    "0}/{3} Chat;46/{0}/{3} Transfer;4C/{0}/DEC TCPIP SVC on Windows NT;42/{0}/mccaffee anti-virus;52/{"+
    "0}/DEC TCPIP SVC on Windows NT;87/{0}/{2} MTA;6A/{0}/{2} IMC;BE/{0}/{5} Agent;BF/{0}/{5} Applicati"+
    "on;03/{0}/Messenger {6};00/{1}/{7} Name;1B/{0}/{7} Master Browser;1C/{1}/{7} Controller;1D/{0}/Mas"+
    "ter Browser;1E/{1}/Browser {6} Elections;2B/{0}/Lotus Notes Server;2F/{1}/Lotus Notes ;33/{1}/Lotu"+
    "s Notes ;20/{1}/DCA IrmaLan Gateway Server;01/{1}/MS NetBIOS Browse Service") -f "UNIQUE","GROUP",
    "Microsoft Exchange","SMS Clients Remote","Modem Sharing","Network Monitor","Service","Domain"
    ).Split(";")

    [Object[]] $Output

    _NbtReference()
    {
        $This.Output = @( )
        $This.String | % {

            $This.Output += [_NbtReferenceObject]::New($_)
        }
    }
}

#\_____________________________________________________________________________________________/#
```

This is essentially an object that charts out what the items found on an NBT map have for a service.
Each item found in **$This.String**, gets parsed into an object and **$This.Output** collects each iteration.
Here's how to get that output...

```
#/-----------------------------------------------------------------------------------------------\#
PS:\> [_NBTReference]::New().Output
```

```
ID   Type   Service
--   ----   -------
<00> UNIQUE Workstation Modem Sharing
<01> UNIQUE Messenger Service
<01> GROUP  Master Browser
<03> UNIQUE Messenger Service
<06> UNIQUE RAS Server Service
<1F> UNIQUE NetDDE Service
<20> UNIQUE File Server Service
<21> UNIQUE RAS Client Service
<22> UNIQUE Microsoft Exchange Interchange(MSMail Connector)
<23> UNIQUE Microsoft Exchange Exchange Store
<24> UNIQUE Microsoft Exchange Directory
<30> UNIQUE Modem Sharing Server
<31> UNIQUE Modem Sharing Client
<43> UNIQUE SMS Clients Remote Control
<44> UNIQUE SMS Administrators Remote Control Tool Service
<45> UNIQUE SMS Clients Remote Chat
<46> UNIQUE SMS Clients Remote Transfer
<4C> UNIQUE DEC TCPIP SVC on Windows NT
<42> UNIQUE mccaffee anti-virus
<52> UNIQUE DEC TCPIP SVC on Windows NT
<87> UNIQUE Microsoft Exchange MTA
<6A> UNIQUE Microsoft Exchange IMC
<BE> UNIQUE Network Monitor Agent
<BF> UNIQUE Network Monitor Application
<03> UNIQUE Messenger Service
<00> GROUP  Domain Name
<1B> UNIQUE Domain Master Browser
<1C> GROUP  Domain Controller
<1D> UNIQUE Master Browser
<1E> GROUP  Browser Service Elections
<2B> UNIQUE Lotus Notes Server
<2F> GROUP  Lotus Notes
<33> GROUP  Lotus Notes
<20> GROUP  DCA IrmaLan Gateway Server
<01> GROUP  MS NetBIOS Browse Service

#\_____________________________________________________________________________/#
```

Now compare the properties that you see here, to the properties in the class below.

```
#/-----------------------------------------------------------------------------\#
[ 4] Class _NbtHostObject      {}
#\_____________________________________________________________________________/#
#/-----------------------------------------------------------------------------\#

Class _NbtHostObject # Nbt Host object to be used for NBT service name resolution
{
    Hidden [String[]]  $Line
    [String]           $Name
    [String]             $ID
    [String]           $Type
    [String]        $Service

    _NbtHostObject([String]$Line)
    {
        $This.Line    = $Line.Split(" ") | ? Length -gt 0
```

```
        $This.Name    = $This.Line[0]
        $This.ID      = $This.Line[1]
        $This.Type    = $This.Line[2]
    }
}

#\_____________________________________________________________________________________________/#
```

Here, the object looks very similar to **`[_NbtReferenceObject],`** but it has an additional property.
That is because the first (**2**) NBT classes are meant to be used as a reference. This means, the raw data has yet to be processed.

The first (**2**) classes are merely guides that keep the (**input/output**) clean, and accessible.
Each input string **`$Line`** is fed into this function, where **`$This.Line`** collects it while also splitting it.

This line alone does all of the work, the other (**3**) lines just assign a slice of that array.

```
#/---------------------------------------------------------------------------------------------\#
[ 5] Class _NbtTable           {}
#\_____________________________________________________________________________________________/#
#/---------------------------------------------------------------------------------------------\#

Class _NbtTable # To chart a section/adapter in nbtstat
{
    [String]      $Name
    [String] $IpAddress
    [Object]     $Hosts

    _NbtTable([String]$Name)
    {
        $This.Name = $Name
        $This.Hosts = @( )
    }

    NodeIp([String]$Node)
    {
        $This.IpAddress = [Regex]::Matches($Node,"(\d+\.){3}(\d+)").Value
    }

    AddHost([String]$Line)
    {
        $This.Hosts += [_NbtHostObject]::New($Line)
    }
}

#\_____________________________________________________________________________________________/#
```

This is meant to collect pieces of the NBT table. It is capturing raw output.
This class is making use of additional methods, to which they are called from an *external scope*.

So, these methods **`NodeIp()`**, and **`AddHost(),`** are both dependent on input from an *external scope*.

The method **`NodeIp()`** is collecting an IP Address from the brackets using a **`[Regex]::Matches($Content,$Pattern).Value`** trick.

```
#/---------------------------------------------------------------------------------------------\#
[ 6] Class _NbtStat            {}
#\_____________________________________________________________________________________________/#
#/---------------------------------------------------------------------------------------------\#
```

```
Class _NbtStat # Charts the entire table/output of nbtstat
{
    Hidden [Object] $Alias
    Hidden [Object] $Table
    Hidden [Object] $Section
    [Object] $Output

    _NbtStat([Object[]]$Interface)
    {
        $This.Alias   = $Interface.Alias | % { "{0}:" -f $_ }
        $This.Table   = nbtstat -N
        $This.Section = @{ }
        $X            = -1

        ForEach ( $Line in $This.Table )
        {
            If ( $Line -in $This.Alias )
            {
                $X ++
                $This.Section.Add($X,[_NbtTable]::New($Line))
            }

            ElseIf ( $Line -match "Node IpAddress" )
            {
                $This.Section[$X].NodeIp($Line)
            }

            ElseIf ( $Line -match "Registered" )
            {
                $This.Section[$X].AddHost($Line)
            }
        }

        $This.Output = $This.Section | % GetEnumerator | Sort-Object Name | % Value
    }
}

#\_____________________________________________________________________________________________/#
```

Now, here's where the last (4) classes that I talked about start to converge.
Because there are typically multiple adapters in any given device, it is best to handle them all as a group of (interface(s)/adapter(s)), rather than each individual (interface/adapter)...

So, whether it is NO adapters, ONE adapter, or MULTIPLE adapters, then this will handle all of those cases.
I began to use a theme in collecting raw input from these legacy console commands like arp and nbtstat.

```
#/-------------------------------------------------------------------------------------------------\#

Hidden [Object]   $Alias
Hidden [Object]   $Table
Hidden [Object] $Section
[Object]         $Output

#\_____________________________________________________________________________________________/#
```

The first 3 properties here are **hidden**, all we really want is the **output**.

But, that **output** needs to be **generated** based on these **hidden values**.

In **PowerShell**, **hidden items** can still be accessed if that property hasn't been filtered out.

(Side note: Do not name a property AND a (*constructor/method*) the same thing. It'll void/nullify whatever you assign to it, and the conflict will show up in the console as "That property doesn't exist." That's why I typically add on an underscore in front of the name)

Hidden items just simply aren't drawn to the screen.
Also, hiding (*input/output*) is an art form in and of itself... but, I digress.

$Alias is going to collect the name of the interface.
$Table is going to collect the output of the console application, in this case "**nbtstat -N**"
$Section is going to parse out each section of the table, so that it can be sorted and added to the correct (**interface/adapter**).
$Output is the result of these things being mixed together like a cocktail of correct information.

Let's talk about this line specifically...
_NbtStat([Object[]]$Interface)

This is where **[_NbtStat],** and all of the prior **[_Nbt*]** based classes are initialized.
(Side Note: * is used as a wildcard character in Regex, 0 or more of that string)

[Object[]]$Interface is an array of interface objects.

```
#/------------------------------------------------------------------------------------------------------\#

$This.Alias   = $Interface.Alias | % { "{0}:" -f $_ }
$This.Table   = nbtstat -N
$This.Section = @{ }
$X            = -1

ForEach ( $Line in $This.Table )
{
    If ( $Line -in $This.Alias )
    {
        $X ++
        $This.Section.Add($X,[_NbtTable]::New($Line))
    }

    ElseIf ( $Line -match "Node IpAddress" )
    {
        $This.Section[$X].NodeIp($Line)
    }

    ElseIf ( $Line -match "Registered" )
    {
        $This.Section[$X].AddHost($Line)
    }
}

$This.Output = $This.Section | % GetEnumerator | Sort-Object Name | % Value

#\______________________________________________________________________________________________________/#
```

Here, the Interface object has a property named "**Alias**".
For each of those objects, a colon is added to the string to match the **nbtstat** content, and pair it with its correct counterpart.

If a match is made, then a new instantiation of class **[_NbtTable]** with that line as input is created.
Each line afterward is processed and checked for (**1**) of those (**3**) conditions.

If a condition is matched, then the line is handed off to the corresponding method in **[_NbtTable]**

[NBT]

Name	ID	Type	Service
DSC0	<00>	UNIQUE	Workstation Modem Sharing
SECURED	<1C>	GROUP	Domain Controller
SECURED	<00>	GROUP	Domain Name
DSC0	<20>	UNIQUE	File Server Service
SECURED	<1B>	UNIQUE	Domain Master Browser

If the line matches an adapter alias, then guess what...?
The current table is complete, and a new one is opened up.

At the tail-end of each section, the (**name/value**) pairs are enumerated, sorted by *name*, and then the *value* is selected...

The end result of this process is saved to the output property, AND the items in the **DataGrid** control box to the left.

In the (**XAML/WPF**) section, I will discuss how to tie the (**classes/variables**) to **XAML** controls via **WPF** binding.
XAML is very much like **HTML** and **CSS** combined.

```
#/------------------------------------------------------------------------------------------------------------\#
[ 7] Class _V4PingObject       {}
#\____________________________________________________________________________________________________________/#
#/------------------------------------------------------------------------------------------------------------\#

Class _V4PingObject # Used as a container for a ping reply and hostname resolution
{
    Hidden [Object]   $Reply
    [UInt32]          $Index
    [String]         $Status
    [String]      $IPAddress
    [String]       $Hostname

    _V4PingObject([UInt32]$Index,[String]$Address,[Object]$Reply)
    {
        $This.Reply          = $Reply.Result
        $This.Index          = $Index
        $This.Status         = @("-","+")[[Int32]($Reply.Result.Status -match "Success")]
        $This.IPAddress      = $Address
        $This.Hostname       = Switch ($This.Status)
        {
            "+"
            {
                Resolve-DNSName $This.IPAddress | % NameHost
            }

            Default
            {
                "-"
            }
        }
    }
}

#\____________________________________________________________________________________________________________/#
```

In **[_V4PingObject],** this object helps to sort through an asynchronous pinging of all hosts on a subnet.
By itself, there's not much to say.
If the reply result status matches success, then the command "**Resolve-DNSName**" is issued.

Resolve-DNSName is a lengthy command when a DNS zone isn't properly configured, and even if it is correctly configured, some

hosts won't respond to this command. So far, it is what takes the most time with the utility so far.

As far as **DNS** is concerned, knowing whether (**A/PTR**) records exist, and how they relate to obtaining **hostnames**...? That is all very relevant to the task at hand. I will not get into DNS management in this lesson... though *that is definitely important in networking*.

```
#/------------------------------------------------------------------------------------------------------\#
[ 8] Class _V4PingSweep        {}
#\______________________________________________________________________________________________________/#
#/------------------------------------------------------------------------------------------------------\#

Class _V4PingSweep # Used to asynchronously ping an entire network and all of its potential hosts.
{
    [String]         $HostRange
    [String[]]       $IPAddress
    Hidden [Hashtable] $Process
    [Object] $Buffer         = @( 97..119 + 97..105 | % { "0x{0:X}" -f $_ } )
    [Object] $Options
    [Object] $Output
    [Object] $Result

    _V4PingSweep([String]$HostRange)
    {
        $This.HostRange = $HostRange
        $Item           = $HostRange -Split "/"

        $Table          = @{ }
        $This.Process   = @{ }

        ForEach ( $X in 0..3 )
        {
            $Table.Add( $X, (Invoke-Expression $Item[$X]) )
        }

        $X = 0

        ForEach ( $0 in $Table[0] )
        {
            ForEach ( $1 in $Table[1] )
            {
                ForEach ( $2 in $Table[2] )
                {
                    ForEach ( $3 in $Table[3] )
                    {
                        $This.Process.Add($X++,"$0.$1.$2.$3")
                    }
                }
            }
        }

        $This.IPAddress = $This.Process | % GetEnumerator | Sort-Object Name | % Value
        $This._Refresh()
    }

    _Refresh()
    {
        $This.Process        = @{ }

        ForEach ( $X in 0..( $This.IPAddress.Count - 1 ) )
```

```
        {
            $IP             = $This.IPAddress[$X]

            $This.Options   = [System.Net.NetworkInformation.PingOptions]::new()
            $This.Process.Add($X,
            [System.Net.NetworkInformation.Ping]::new().SendPingAsync($IP,100,$This.Buffer,$This.Options))
        }

        $This.Output        = @( )

        ForEach ( $X in 0..( $This.IPAddress.Count - 1 ) )
        {
            $IP             = $This.IPAddress[$X]
            $This.Output   += [_V4PingObject]::New($X,$IP,$This.Process[$X])
        }
    }
}

#\_____________________________________________________________________________/#
```

This one here, is a bit of a doozy. In this specific case, the *shaolin master* that I studied from in order to write this, is **Boe Prox**.

While he may not be an actual *shaolin master*..? That's beside the point... He practically is one, based on his plethora of knowledge in things **PowerShell** related. **Mr. Prox** wrote an article featuring the differences between **asynchronous pinging** and then **pinging in other ways**. I don't have the specific link.

But, I jumped into *tweaking* what his function did until it suited the conditions that could scan an IPv4 network of any size... Because I thought perhaps threading might be a great way to ping ***that*** many hosts...
Boe Prox also developed **Posh-RSJobs**, which can be found here... https://github.com/proxb/PoshRSJob

I've had to take a step back from the idea of spinning up threads willy-nilly... because the possibilities begin to ramp up drastically when mulling over what you could do... especially if you have access to a Threadripper... Oh boy.

[Tangent : Threadripper]

I built this machine for a client who trades currency, in August of 2017 when the first-gen TR4 platform launched. The Zen, Ryzen, and Threadripper platforms were all first created about 10 miles north of where I have lived most of my life- that's **Malta, NY**.

Anyway, wasn't until I was virtualizing about a dozen machines with this single machine pictured here to the left, that I saw how cool programming could be.

Originally, I was using **Oracle VirtualBox**, however there were limitations with the hardware AND being able to use native UEFI. **VMWare Workstation Player/Pro** has this capability, but what I didn't know then, was that **VMWare** and **Hyper-V** both use the same hypervisor. **Hyper-V** is what I use now.

[End Tangent : Threadripper]

Anyway, depending on its subnet mask, gateway, etc... well, this **[_V4PingSweep]** object will take a single string named "**HostRange**"... and then expand it into all (**available/possible**) hosts on that given network.

Now... you could have your standard issue **250**-ish hosts on a network, and the function will work fast.

But, you could *also* have a network with **16,777,214** hosts on it, if not more via **VLSM**...

I probably don't have to say it, but... if you have a network of that size...? You're probably not going to want to wait for the original function I wrote, to ping each individual host on that network... in order to locate an **Active Directory** domain controller that you could use to promote a **New-ADDomain**, **New-ADDomainController**, or **New-ADForest**...

But, even if you do...? Then the way I wrote it, *should* take the same amount of time...
In other words, it should only take a few seconds if **DNS** is properly configured on your network.

In order to *really* dive deeply into this class... however, I have to discuss this much longer one below.

```
#/------------------------------------------------------------------------------------------------------------\#
[ 9] Class _V4Network          {}
#\____________________________________________________________________________________________________________/#
#/------------------------------------------------------------------------------------------------------------\#

Class _V4Network # Used to contain an interface configured to use IPv4
{
    [String]            $IPAddress
    [String]                $Class
    [Int32]                $Prefix
    [String]               $Netmask
    Hidden [Object]          $Route
    [String]               $Network
    [String]               $Gateway
    [String[]]              $Subnet
    [String]            $Broadcast
    [String]            $HostRange

    [String] GetNetmask([Int32]$CIDR)
    {
        $Switch         = 0

        Return @( ForEach ( $I in 0..3 )
        {
            If ( $CIDR -in @{ 0 = 1..7; 1 = 8..15; 2 = 16..23; 3 = 24..30 }[$I] )
            {
                $Switch = 1
                @(0,128,192,224,240,248,252,254,255)[$CIDR % 8]
            }

            Else
            {
                @(255,0)[$Switch]
            }
        }) -join "."
    }

#    IPCheck() // Left this in here, it's not being used.
#    {
#        $Item = [IPAddress]$This.IPAddress | % GetAddressBytes

#        If ( $Item[0] -in @(0,127;224..255) )
#        {
#            Throw "Invalid Address Detected"
#        }

#        If ( ( $Item[0..1] -join '.' ) -eq "169.254" )
```

```
#        {
#            Throw "Automatic Private IP Address Detected"
#        }
#    }

    GetHostRange()
    {
        $Item           = [IPAddress]$This.IPAddress | % GetAddressBytes
        $Mask           = [IPAddress]$This.Netmask   | % GetAddressBytes
        $This.HostRange = @( ForEach ( $I in 0..3 )
        {
            $Step = 256 - $Mask[$I]

            Switch ( $Step )
            {
                1
                {
                    $Item[$I]
                }

                256
                {
                    "0..255"
                }

                Default
                {
                    $Slot = 256 / $Step

                    ForEach ( $X in 0..( ( 256 / $Slot ) - 1 ) )
                    {
                        $IRange = ( $X * $Slot ) | % { $_..( $_ + $Slot - 1 ) }

                        If ( $Item[$I] -in $IRange )
                        {
                            "{0}..{1}" -f $IRange[0,-1]
                        }
                    }
                }
            }
        }) -join '/'
    }

    _V4Network([Object]$Address)
    {
        If ( ! $Address )
        {
            Throw "Address Empty"
        }

        $This.IPAddress = $Address.IPAddress
       # $This.IPCheck() // Not used
        $This.Class     = @('N/A';@('A')*126;'Local';@('B')*64;@('C')*32;@('MC')*16;@('R')*15;'BC')[[Int32
]$This.IPAddress.Split(".")[0]]
        $This.Prefix    = $Address.PrefixLength
        $This.Netmask   = $This.GetNetMask($This.Prefix)
        $This.Route     = Get-NetRoute -AddressFamily IPV4 | ? InterfaceIndex -eq $Address.InterfaceIndex
        $This.Network   = $This.Route | ? { ($_.DestinationPrefix -Split "/")[1] -
match $This.Prefix } | % { ($_.DestinationPrefix -Split "/")[0] }
```

```
        $This.Gateway    = $This.Route | ? NextHop -ne 0.0.0.0 | % NextHop
        $This.Subnet     = $This.Route | ? DestinationPrefix -notin  255.255.255.255/32, 224.0.0.0/4,
0.0.0.0/0 | % DestinationPrefix | Sort-Object
        $This.Broadcast = ( $This.Subnet | % { ( $_ -Split "/" )[0] } )[-1]
        $This.GetHostRange()
    }

    [Object[]] ScanV4()
    {
        Return @( [_V4PingSweep]::New($This.HostRange).Output | ? Status -eq + )
    }
}

#\_____________________________________________________________________________________________/#
```

What you're seeing here, is a chaotic consortium of carefully AND considerately choreographed characters... class for short.
This class **[_V4Network]** ... is massive. But, it does a lot.

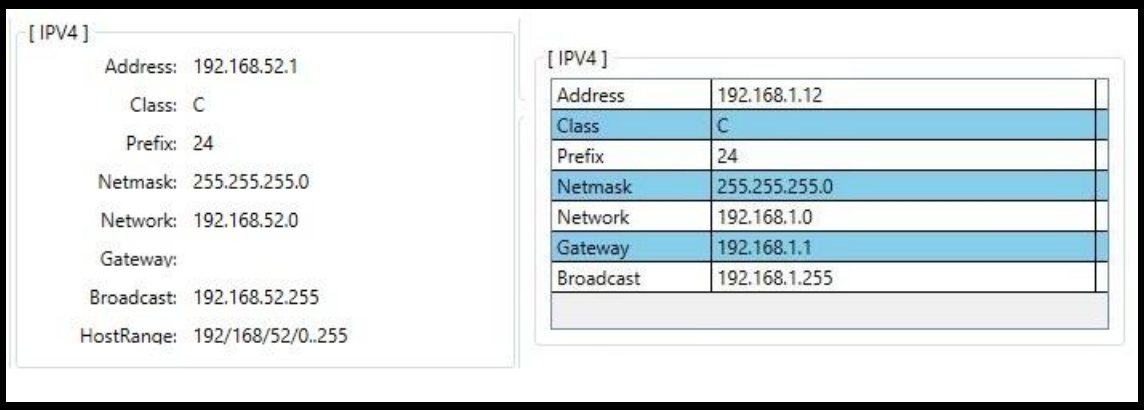
Any interface that has an IPV4 address is going to be collected, and then sent in here.
Sometimes an interface will have multiple addresses, and we want each address to have its network mapped out correctly.
The address object is obtained and if it is (**empty/null**), then the object will be thrown.

Like **Thor**, the **god of lighting**'s hammer... Or, **Captain America**'s shield. Sometimes *they* get ***thrown***.
If the interface is not null, then it (*won't get thrown/will show the output below*).

```
#/-------------------------------------------------------------------------------------------------\#

IPAddress : 192.168.52.1
Class     : C
Prefix    : 24
Netmask   : 255.255.255.0
Network   : 192.168.52.0
Gateway   :
Subnet    : {192.168.52.0/24, 192.168.52.1/32, 192.168.52.255/32}
Broadcast : 192.168.52.255
HostRange : 192/168/52/0..255

#\_____________________________________________________________________________________________/#
```



The left side is the older, **Grid**, **RowDefinitions**, **ColumnDefinitions**, **Labels**, **Styling**, **Margin**, **Label** -based approach.
The right side is a single **DataGrid** box, it has a fair amount of code as well, but... still a lot less.
The amount of code that the **DataGrid** approach replaces, is staggering actually.
I will go over comparisons between the two in the (**XAML/WPF**) section.

Being able to utilize **DataGrid** at its *maximum capacity*… requires a bit of **code-behind**. AND, **code-front**.
That's why it is rather cool to know how these **DataGrid** controls can in fact, be powered by **PowerShell**, *classes*, and ***WPF DataBinding***.

There are ways of using **Windows Presentation Foundation** without even writing a chunk of **XAML**..? That'll be a future project.
When it comes to making GUI's that look brilliant, do a lot, and function rather efficiently...?
(**XAML/WPF**) is a stellar choice. As a result… well, I thought that it was *so cool and interesting*…

That I was determined to figure out how it works.

Damien Van Robaeys from http://www.systanddeploy.com/, made a video on **YouTube** that covers this exact process that I'm sharing. Maybe not the networking components, but the **XAML** based **DataGrid** stuff with variables and such?
That he did do. **Damien** had a number of inspirational projects that I stumbled into while researching ways to automate the *Microsoft Deployment Toolkit* like a *boss*.

Anyway, I saw a way to sew several ideas together, and, this lesson here is something I conceptualized then. My research into **CSS Grid** at or about that time, allowed me to see a way to apply my **HTML/CSS** skills to create **XAML** content. **CSS Grid** doesn't hold a candle to **XAML** based **Grid**'s, and the reason I say that is mainly because I had a serious issue being able to nest further tables dynamically and for scalability with **CSS Grid**. Switching between **Flexbox** and **Grid**, is a lot like switching between **StackPanels** and **Grid**…

With **XAML based Grids,** nesting additional grids, and correctly aligning things… that's far more controllable via **XAML**.
That's why I like **XAML** as much as I do. I'm sure **JavaScript** can do plenty to match **XAML**, but ask yourself how many **cross-site script attacks** take place, *wiping out companies left and right* by either **stealing their data, intellectual property, or money** through **advanced cyberattacks**…? Not unlike the recent issue with **SolarWinds. JavaScript** has a serious problem with security.

Perhaps **JavaScript** could be (*controlled/interoperated*) via **PowerShell** or **C#**… and they could just call it `[JSInterop]`…
Blazor is something that sorta does that... its design provides protection against (*XSS/Cross-Site Script*) attacks, where passwords can be easily siphoned, allowing access to people's *digital (bank/idea) vaults*… or against **DNS rebind attacks** where someone **keylogs passwords** via a **fake website vishing attack**.

Yeah, when I managed a chain of computer shop networks, some customers were buying brand new devices from **Walmart** or **Amazon**, and what they received, was a device that came with a virus on it. So, they go to log into **Facebook**, they log in, but it wasn't actually **Facebook**. I'm not being obtuse. **Amazon** doesn't have a rule for "what if the thing I bought from your company steals all of my money…?", and they probably should.

Normally, people don't find out until long after someone has stolen their money or credit, that they were a victim of identity theft. When the computer or device is running slowly, it doesn't automatically indicate that there was a hacker involved, but sometimes that's how it happens… computer starts running extremely slowly and THAT'S when the customer brings it to be serviced.

Anyway, it was a main objective in the beginning of my research, to get well acquainted with **Blazor**, since it is essentially (**ASP.Net/C#)** but… on steroids. I do intend to get back into full development with **Blazor** at some point. Hasn't been a priority recently. At any rate, when comparing the left side to the right side in the image above... which one looks better to you?
I'm not likely to speak for others, but *my* opinion tells me that the *right side* looks *slightly* better.

That is a **WPF DataGrid**, controlled via the classes that I talk about above. In **PowerShell**.
No **C#**…
No **HTML**…
No **CSS**…
No **JavaScript**…
No **image editing**…
No cheating, either.

This is a **Graphical User Interface** that's ready to roll, with your standard issue installation of **Windows 10**.

Anyway, getting a **DataGrid** to display a table of (*name*, *value*) pairs, you can get snagged pretty quickly without enumerating the values. **Enumeration** is where each *name* and *value* is retrieved. Enumerate the (*name*, *value*) pairs and it'll show up in a **DataGrid**

pretty easily.

That's what **[_DGList]** is for...

```
#/------------------------------------------------------------------------------------------------------------\#
[19] Class _DGList            {}
#\____________________________________________________________________________________________________________/#
#/------------------------------------------------------------------------------------------------------------\#

Class _DGList # Enumerates (Name/Value) pairs to populate a DataGrid
{
    [String] $Name
    [Object] $Value

    _DGList([String]$Name,[Object]$Value)
    {
        $This.Name  = $Name
        $This.Value = $Value
    }
}

#\____________________________________________________________________________________________________________/#
```

I skipped ahead slightly, but this class was needed to "**splat**" the class into an object that the (**DataGrid/Table**) in the graphic above can display. You *can* use **column headers** when there are **multiple objects**, but- when there is a **single object**, then having a class like this to pipe the (**input/output**) into the **DataGrid**.**ItemsSource** is helpful- it'll keep the (*input/output*) clean and accessible.

```
#/------------------------------------------------------------------------------------------------------------\#
[10] Class _V6Network         {}
#\____________________________________________________________________________________________________________/#
#/------------------------------------------------------------------------------------------------------------\#

Class _V6Network # Container for an IPV6 network
{
    [String] $IPAddress
    [Int32]  $Prefix
    [String] $Link

    _V6Network([Object]$Address)
    {
        $This.IPAddress = $Address.IPAddress
        $This.Prefix    = $Address.PrefixLength
    }
}

#\____________________________________________________________________________________________________________/#
```

This class will be getting additional attention, but it's worth noting that **IPV6** does not need as much micromanagement as **IPV4**.

Right now I'm about to cover the whole **[_NetInterface]** business, where the above classes get combined into a single control. You might be thinking... "*Well, this sounds like wizardry...*" But, it's not. It's just some **PowerShell**...

```
#/------------------------------------------------------------------------------------------------------------\#
[11] Class _NetInterface      {}
#\____________________________________________________________________________________________________________/#
#/------------------------------------------------------------------------------------------------------------\#

Class _NetInterface
```

```
{
    Hidden [Object] $Interface
    [String] $Hostname
    [String] $Alias
    [Int32]  $Index
    [String] $Description
    [String] $Status
    [String] $MacAddress
    [String] $Vendor
    [Object] $IPv4
    [Object] $IPv6
    [Object] $Nbt
    [Object] $Arp

    _NetInterface([Object]$Interface)
    {
        $This.Interface   = $Interface
        $This.HostName    = $Interface.ComputerName
        $This.Alias       = $Interface.InterfaceAlias
        $This.Index       = $Interface.InterfaceIndex
        $This.Description = $Interface.InterfaceDescription
        $This.Status      = $Interface.NetAdapter.Status
        $This.MacAddress  = $Interface.NetAdapter.LinkLayerAddress

        $This.IPV4        = @( )

        ForEach ( $Address in $Interface.IPV4Address )
        {
            $This.IPV4   += [_V4Network]::New($Address)
        }

        $This.IPV4        = $This.IPV4 | Select-Object -Unique

        $This.IPV6        = @( )

        ForEach ( $Address in $Interface.IPV6Address)
        {
            $This.IPV6   += [_V6Network]::New($Address)
        }

        ForEach ( $Address in $Interface.IPV6LinkLocalAddress )
        {
            $This.IPV6   += [_V6Network]::New($Address)
        }

        ForEach ( $Address in $Interface.IPV6TemporaryAddress )
        {
            $This.IPV6   += [_V6Network]::New($Address)
        }

        $This.IPV6        = $This.IPV6 | Select-Object -Unique
    }

    GetVendor([Object]$Vendor)
    {
        $This.Vendor = $Vendor.VenID[ ( $This.MacAddress -Replace "(-|:)","" | % Substring 0 6 ) ]
    }

    Load([Object]$Nbt,[Object]$Arp)
```

```
    {
        $This.Nbt = $Nbt
        $This.Arp = $Arp
    }
}

#\_____________________________________________________________________________________________/#
```

This is an instantiation of a class that allows us to pool together the variables and properties in the above classes.
The method **GetVendor($Vendor)** is fed the vendor list in order to set the vendor tag.

The process in which it does this allows for the list to be prepared once, and then each adapter found in any object that calls the **[_VendorList]**... to immediately return the correct vendor.

The method **Load($Nbt,$Arp)** is essentially putting the correct (**adapter/interface**) **ARP** and **NBT** pools into the interface, already parsed and objectified.

```
#/---------------------------------------------------------------------------------------------\#
[12] Class _ArpHostObject      {}
#\_____________________________________________________________________________________________/#
#/---------------------------------------------------------------------------------------------\#

Class _ArpHostObject # Used to identify ARP network hosts
{
    Hidden [String]    $Line
    [String]       $Hostname
    [String]      $IpAddress
    [String]     $MacAddress
    [String]         $Vendor
    [String]           $Type

    _ArpHostObject([String]$Line)
    {
        $This.Line       = $Line
        $This.IpAddress  = $This.Line.Substring(2,22).Replace(" ","")
        $This.MacAddress = $This.Line.Substring(24,17)
        $This.Type       = $This.Line.Substring(46)
    }

    GetVendor([Object]$Vendor)
    {
        $This.Vendor     = $Vendor.VenID[ ( $This.MacAddress -Replace "(-|:)","" | % Substring 0 6 ) ]
    }
}

#\_____________________________________________________________________________________________/#
```

This simply parses the **arp** table content it is fed into a host object.
You can see that it is the same method **GetVendor()** as the prior class, but it is within a different object.

```
#/---------------------------------------------------------------------------------------------\#
[13] Class _ArpTable           {}
#\_____________________________________________________________________________________________/#
#/---------------------------------------------------------------------------------------------\#

Class _ArpTable
{
    [String]      $Name
    [String] $IpAddress
```

```
    [Object]     $Hosts

    _ArpTable([String]$Line)
    {
        $This.Name      = $Line.Split(" ")[-1]
        $This.IPAddress = $Line.Replace("Interface: ","").Split(" ")[0]
        $This.Hosts     = @( )
    }
}

#\_____________________________________________________________________________________________________/#
```

This will do a similar thing to the **arp** table that was done to the **nbtstat** output, where it parses the table into objects and returns the hosts on the network.

The below class creates an instance of the above class for each adapter that is found.

```
#/-----------------------------------------------------------------------------------------------------\#
[14] Class _ArpStat           {}
#\_____________________________________________________________________________________________________/#
#/-----------------------------------------------------------------------------------------------------\#

Class _ArpStat
{
    [Object] $Alias
    [Object] $Table
    [Object] $Section
    [Object] $Output

    _ArpStat([Object[]]$Interface)
    {
        $This.Alias = ForEach ( $I in $Interface )
        {
            "Interface: {0} --- 0x{1:x}" -f $I.IPV4.IPAddress, $I.Index
        }

        $This.Table   = arp -a
        $This.Section = @{ }
        $X            = -1

        ForEach ( $Line in $This.Table )
        {
            If ( $Line -in $This.Alias )
            {
                $X ++
                $This.Section.Add( $X,[_ArpTable]::New($Line))
            }

            ElseIf ( $Line -match "(static|dynamic)" )
            {
                $This.Section[$X].Hosts += [_ArpHostObject]::New($Line)
            }
        }

        $This.Output = $This.Section | % GetEnumerator | Sort-Object Name | % Value
    }
}
```

```
#\____________________________________________________________________________________________/#
```

This class is also parsing the broadcast and multicast objects into the table.

I left those in there intentionally, because when it comes to hidden devices on a network...?
Well, it's worth taking a closer look at anomalies in those scopes, if any exist.

```
#/--------------------------------------------------------------------------------------------\#
[15] Class _NetStatAddress     {}
#\____________________________________________________________________________________________/#
#/--------------------------------------------------------------------------------------------\#

Class _NetStatAddress
{
    Hidden [String]  $Item
    [String]    $IPAddress
    [String]        $Port

    _NetStatAddress([String]$Item)
    {
        $This.Item      = $Item

        If ( $Item -match "(\[.+\])" )
        {
            $This.IPAddress = [Regex]::Matches($Item,"(\[.+\])").Value
            $This.Port      = $Item.Replace($This.IPAddress,"")
            $This.IPAddress = $Item.TrimStart("[").Split("%")[0]
        }

        Else
        {
            $This.IPAddress = $This.Item.Split(":")[0]
            $This.Port      = $This.Item.Split(":")[1]
        }
    }
}

#\____________________________________________________________________________________________/#
```

I believe that I am not using the above class at all, but it's worth checking out what its purpose was. In this case, I was looking to **create an object** that **could be usable for both**: the local address + port, as well as the remote address + port, when looking at the **netstat** table. **Netstat** isn't _particularly_ needed for connecting to an **Active Directory** instance... but, it is handy to have in a networking tool.

```
#/--------------------------------------------------------------------------------------------\#
[16] Class _NetStatObject      {}
#\____________________________________________________________________________________________/#
#/--------------------------------------------------------------------------------------------\#

Class _NetStatObject
{
    Hidden [String]   $Line
    Hidden [Object]   $Item
    [String]      $Protocol
    [String]  $LocalAddress
    [String]     $LocalPort
    [String] $RemoteAddress
    [String]    $RemotePort
    [String]         $State
```

```
    [String]     $Direction

    _NetStatObject([String]$Line)
    {
        $This.Line          = $Line
        $This.Item          = $This.Line -Split " " | ? Length -gt 0
        $This.Protocol      = $This.Item[0]
        $This.LocalAddress  = $This.GetAddress($This.Item[1])
        $This.LocalPort     = $This.Item[1].Replace($This.LocalAddress + ":","")
        $This.RemoteAddress = $This.GetAddress($This.Item[2])
        $This.RemotePort    = $This.Item[2].Replace($This.RemoteAddress + ":","")
        $This.State         = $This.Item[3]
        $This.Direction     = $This.Item[4]
    }

    [String] GetAddress([String]$Item)
    {
        Return @( If ( $Item -match "(\[.+\])" )
        {
            [Regex]::Matches($Item,"(\[.+\])").Value
        }

        Else
        {
            $Item.Split(":")[0]
        })
    }
}

#\_____________________________________________________________________________________________________/#
```

This class probably looks a fair amount like a firewall entry..!
That is mainly because that's what it is...
...it's the firewall and the interface to it.

However, saying that the class *itself*, is a firewall entry... is nonsense.
Because it isn't a firewall entry...
I know I just said that's what it is... but- it's just a *class* that *parses objects* in **netstat**.
That's why it LOOKS like a firewall entry... because they're pieces of (*session data).*

[Tangent : (OSI/Open Systems Interconnect) Model]

7 / Application – The program talking to itself, an API/Application-Presentation Interface
6 / Presentation – When the operating system takes session data and presents it to the user
5 / Session – Whatever ports are open and sending either TCP or UDP
4 / Transport – For sending/receiving packets over the network (IPX/SPX/TCP/UDP).
3 / Network – Routing, NAT/Network Address Translation, Internet, OSPF/RIP
2 / Data Link – Hardware interface with the copper, maybe this is flow control.
1 / Physical – Copper/Optical

Get your 7 layer (*OSI model/crunchwrap supreme*) in order here...

[End Tangent : OSI Model]

```
#/-----------------------------------------------------------------------------------------------------\#
[17] Class _NetStat           {}
#\_____________________________________________________________________________________________________/#
#/-----------------------------------------------------------------------------------------------------\#
```

```
Class _NetStat
{
    [Object] $Alias
    [Object] $Table
    [Object] $Section
    [Object] $Output

    _NetStat()
    {
        $This.Alias   = "Active Connections"
        $This.Table   = netstat -ant

        $This.Section = @{}
        $X            = -1

        ForEach ( $Line in $This.Table )
        {
            If ( $Line -match "(TCP|UDP)" )
            {
                $X ++
                $This.Section.Add($X,[_NetStatObject]::New($Line))
            }
        }

        $This.Output  = $This.Section | % GetEnumerator | Sort-Object Name | % Value
    }
}

#\_____________________________________________________________________________________/#
```

Anyway, knowing where the firewall action is happening, is where the **netstat** money could be made...

Now, turning all of these classes (*input/output*), into tangible objects on the screen...?
Well, that's where you could then begin to melt people's minds with numerous possibilities...

```
#/-------------------------------------------------------------------------------------\#
[18] Class _Controller         {}
#\_____________________________________________________________________________________/#
#/-------------------------------------------------------------------------------------\#

Class _Controller
{
    Hidden [Object]      $VendorList
    Hidden [Object]    $NbtReference
    Hidden [Object]             $Nbt
    Hidden [Object]             $Arp
    [Object]              $Interface
    [Object]                $Network
    [Object]                $NetStat

    _Controller()
    {
        Write-Host "Collecting Network Adapter Information"

        $This.VendorList       = [_VendorList]::New(
        "https://raw.githubusercontent.com/mcc85sx/FightingEntropy/master/scratch/VendorList.txt")
        $This.NBTReference     = [_NBTReference]::New().Output
        $This.Interface        = @( )
```

```
        ForEach ( $Interface in Get-NetIPConfiguration )
        {
            $Adapter           = [_NetInterface]::New($Interface)
            Write-Host ( "[+] {0}" -f $Adapter.Alias )
            $Adapter.GetVendor($This.VendorList)
            $This.Interface   += $Adapter
        }

        $This.NBT              = [_NbtStat]::New($This.Interface).Output
        $This.ARP              = [_ArpStat]::New($This.Interface).Output

        ForEach ( $Interface in $This.NBT )
        {
            ForEach ( $xHost in $Interface.Hosts )
            {
                $xHost.Service = $This.NBTReference | ? {

                    $_.ID -match $xHost.ID -and $_.Type -eq $xHost.Type

                } | % Service
            }
        }

        ForEach ( $I in 0..( $This.Interface.Count - 1 ) )
        {
            $IPAddress         = $This.Interface[$I].IPV4.IPAddress

            $xNbt              = $This.Nbt | ? IpAddress -match $IpAddress | % Hosts
            $xArp              = $This.Arp | ? IpAddress -match $IpAddress | % Hosts

            ForEach ( $Item in $xArp )
            {
                If ( $Item.Type -match "static" )
                {
                    $Item.Hostname = "-"
                    $Item.Vendor   = "-"
                }

                If ( $Item.Type -match "dynamic" )
                {
                    $Item.GetVendor($This.VendorList)

                    If ( !$Item.Vendor )
                    {
                        $Item.Vendor = "<unknown>"
                    }
                }
            }

            $This.Interface[$I].Load($xNbt,$xArp)
        }

        $This.Network = $This.Interface | ? { $_.IPV4.Gateway }

        $This.RefreshIPv4Scan()
        $This.RefreshNetStat()
    }

    RefreshNetStat()
```

```
{
    $This.NetStat   = [_NetStat]::New().Output
}

RefreshIPv4Scan()
{
    If (!$This.Network)
    {
        Throw "No available network found"
    }

    Else
    {
        Write-Host "Scanning available IPv4 network(s)..."
        ForEach ( $Item in $This.Network.IPv4.ScanV4() )
        {
            $This.Network.Arp | ? IpAddress -match $Item.IpAddress | % {

                $_.HostName = $Item.Hostname
            }
        }
    }
}

Report()
{
    ForEach ( $Interface in $This.Interface )
    {
        $Interface | % {

            Write-Theme @(
            "Interface [$($_.Alias)]",
            " ",
            "---- Host Information --------------------------";
            @{
                Hostname    = $_.Hostname
                Alias       = $_.Alias
                Index       = $_.Index
                Description = $_.Description
                Status      = $_.Status
                MacAddress  = $_.MacAddress
                Vendor      = $_.Vendor
            };
            " ",
            "---- IPv4 Information --------------------------";
            ForEach ( $IPV4 in $_.IPV4 )
            {
                @{
                    IPAddress   = $IPV4.IPAddress
                    Class       = $IPV4.Class
                    Prefix      = $IPV4.Prefix
                    Netmask     = $IPV4.Netmask
                    Network     = $IPV4.Network
                    Gateway     = $IPV4.Gateway
                    Subnet      = $IPV4.Subnet
                    Broadcast   = $IPV4.Broadcast
                    HostRange   = $IPV4.HostRange
                }
            };
```

```
                " ",
                "---- IPv6 Information ---------------------------";
                ForEach ( $IPV6 in $_.IPV6 )
                {
                    @{
                        IPAddress = $IPV6.IPAddress
                        Prefix    = $IPV6.Prefix
                    }
                })

                Start-Sleep -Seconds 2
            }
        }
    }
}

#\______________________________________________________________________________________________________/#
```

This thing up above, is the class that controls all of the interfaces that are collected/found, as well as all of the **nbtstat**, **arp**, and **netstat** tables, so that all of the information can be seen, and dynamically update itself depending on the selected **ComboBox** item.

```
[ Section    ]

        Interface [VMware Network Adapter VMnet1]

        ---- Host Information ------------


        Alias                      : VMware Network Adapter VMnet1
        Description                : VMware Virtual Ethernet Adapter for VMnet1
        Hostname                   : BLUESTREET
        Index                      : 10
        MacAddress                 : 00-50-56-C0-00-01
        Status                     : Up
        Vendor                     : VMware

        ---- IPv4 Information -----------


        Broadcast                  : 192.168.52.255
        Class                      : C
        Gateway                    :
        Hostrange                  : 192/168/52/0..255
        IPAddress                  : 192.168.52.1
        Netmask                    : 255.255.255.0
        Network                    : 192.168.52.0
        Prefix                     : 24
        Subnet                     : 192.168.52.0/24 192.168.52.1/32 192.168.52.255/32

        ---- IPv6 Information -----------


        IPAddress                  : fe80::f0da:f07f:22a5:73f9%10
        Prefix                     : 64

[  Press Enter to Continue   ]
```

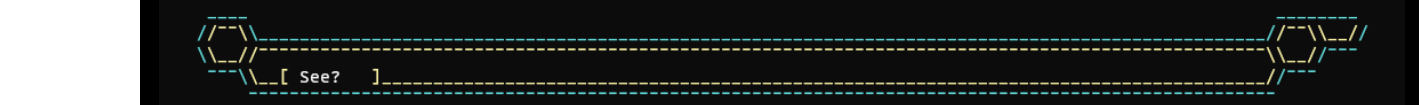
This is a function that is not part of this utility, but when the values that are stored in the GUI are fed into this function, then it will produce a themed message that contains those parameters. It is doing so in alphabetical order, so classes are losing their intended shaping, but, the function still displays the correct information.

There is a method that refreshes the **netstat** panel (**netstat** panel is not in this GUI sample). There is another method that refreshes the **IPV4** hosts found panel.

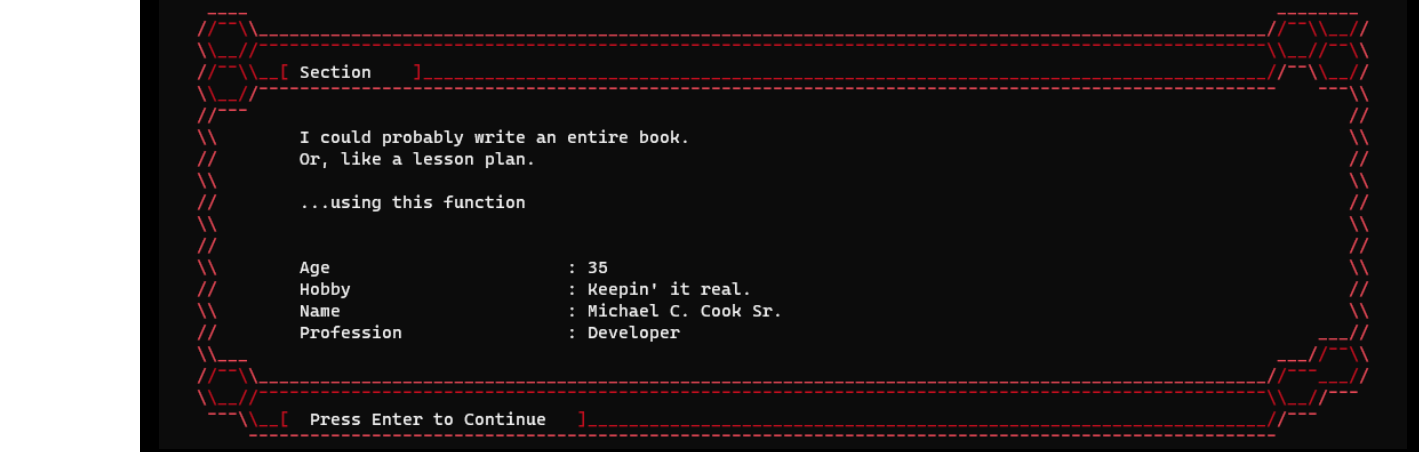
There is a method that will print the information found for each interface that is on the host machine featuring the function "**Write-Theme**" which is not featured in this lesson.

But, this is what "**Write-Theme**" also does below... Changing colors.


```
PS C:\Users\mcook85> write-theme "See?" 11,14,15,0
```




```
PS C:\Users\mcook85> Write-Theme @("I could probably write an entire book.","Or, like a lesson plan."," ",
>> "...using this function"," ";@{ Age = 35; Name = "Michael C. Cook Sr."; Profession = "Developer";
>> Hobby = "Keepin' it real."}) 12,4,15,0
```



How cool is that...?

```
#/-------------------------------------------------------------------------------------------------------------\#
[20] Class _FENetwork          {}
#\_____________________________________________________________________________________________________________/#
#/-------------------------------------------------------------------------------------------------------------\#

Class _FENetwork
{
    [Object] $Window
    [Object] $IO
    [Object] $Control

    _FENetwork()
    {
        $This.Window                    = Get-XamlWindow -Type FENetwork
        $This.IO                        = $This.Window.IO
        $This.Control                   = [_Controller]::New()
    }

    [Object] HostInfo([Object]$Interface)
    {
        Return @(

            ("Hostname"     , $Interface.Hostname        ),
```

```
            ("Alias"       , $Interface.Alias           ),
            ("Index"       , $Interface.Index           ),
            ("Description" , $Interface.Description     ),
            ("Status"      , $Interface.Status          ),
            ("MacAddress"  , $Interface.MacAddress      ),
            ("Vendor"      , $Interface.Vendor          ) | % { [_DgList]::New($_[0],$_[1]) }
        )
    }

    [Object] IPV4Info([Object]$Interface)
    {
        Return @(

            ("Address"     , $Interface.IPV4.IPAddress  ),
            ("Class"       , $Interface.IPV4.Class      ),
            ("Prefix"      , $Interface.IPV4.Prefix     ),
            ("Netmask"     , $Interface.IPV4.Netmask    ),
            ("Network"     , $Interface.IPV4.Network    ),
            ("Gateway"     , $Interface.IPV4.Gateway    ),
            ("Broadcast"   , $Interface.IPV4.Broadcast  ) | % { [_DgList]::New($_[0],$_[1]) }
        )
    }

    [Object] IPV6Info([Object]$Interface)
    {
        Return @(

            ("Address"     , $Interface.IPV6.IPAddress  ),
            ("Prefix"      , $Interface.IPV6.Prefix     ) | % { [_DgList]::New($_[0],$_[1]) }
        )
    }

    Stage([Object]$Interface)
    {
        $This.IO._HostInfo.ItemsSource      = $This.HostInfo($Interface)
        $This.IO._IPV4Info.ItemsSource      = $This.IPV4Info($Interface)
        $This.IO._IPV6Info.ItemsSource      = $This.IPV6Info($Interface)
        $This.IO._Nbt.ItemsSource           = $Interface.Nbt
        $This.IO._Arp.ItemsSource           = $Interface.Arp
    }
}

#\_____________________________________________________________________________________________/#
```

This class here, combines the **XAML** and prepares some of the "code-behind", as well as providing all of the above classes, thrown together... mainly to handle the **GUI**, it's not needed by the **DC Promo** tool.

The result of this class, is an object that has all of its subcomponents able to be REREFERENCED and reset, according to the **GUI** and its settings or actions.

When all of the above classes are wrapped into a function, then only this one single file will be needed to use and gain use out of. Still, the final code that starts turning the LOOK of the page into actionable, able to be interacted with, code...?

```
#/-------------------------------------------------------------------------------------------\#

If ( $GUI )
{
    $UI = [_FENetwork]::New()
```

```
    $UI.IO._Interfaces.ItemsSource              = $UI.Control.Interface.Alias
    $UI.IO._Interfaces.SelectedIndex            = 0

    $UI.Stage($UI.Control.Interface[($UI.IO._Interfaces.SelectedIndex)])

    $UI.IO._Interfaces.Add_SelectionChanged(
    {
        $UI.Stage($UI.Control.Interface[($UI.IO._Interfaces.SelectedIndex)])
    })

    $UI.IO.Cancel.Add_Click(
    {
        $UI.IO.DialogResult = $False
    })

    $UI.Window.Invoke()
}

Else
{
    [_Controller]::New()
}

#\_____________________________________________________________________________________________/#
```

Normally, the function '**Get-FENetwork -GUI**' would be available, but... because I'm discussing it's content...? Then, that's why it's not wrapped in the function.

The variable **$UI** gets the class **[_FENetwork].**
Just for the sake of drawing parallels here, using:

```
#/---------------------------------------------------------------------------------------------\#

$UI = [_FENetwork]::New()
$UI = New-Object _FENetwork

#\_____________________________________________________________________________________________/#
```

These commands both do the same exact thing. **$UI.IO._Interfaces** is a **ComboBox** control, and setting its **.ItemsSource** property to the **$UI.Control.Interface.Alias** allows each interface name selectable by said **ComboBox**, as well as all of the child item content, such as, the **DataGrid's** and their content..

$UI.IO._Interfaces.Add_SelectionChanged({$codeblock}), is an (*action/event handler*) that is technically referencing itself, allowing easy **DataBinding**. You used to need to be adept at writing **C#**, in order to lay down the proverbial (*programming/graphic design*) law... but now...? You can do it in **PowerShell**.

Amazing, right? There's nothing wrong with **C#** by the way. Don't get your knickers in a jam, if I talk wicked highly about **PowerShell**... and I don't talk about the thing that **PowerShell** owes its entire existence/life to... Cause. I'm not trying to come across as some kind of twisted sicko that'd leave **PowerShell**'s dad out in the cold... C'mon now.

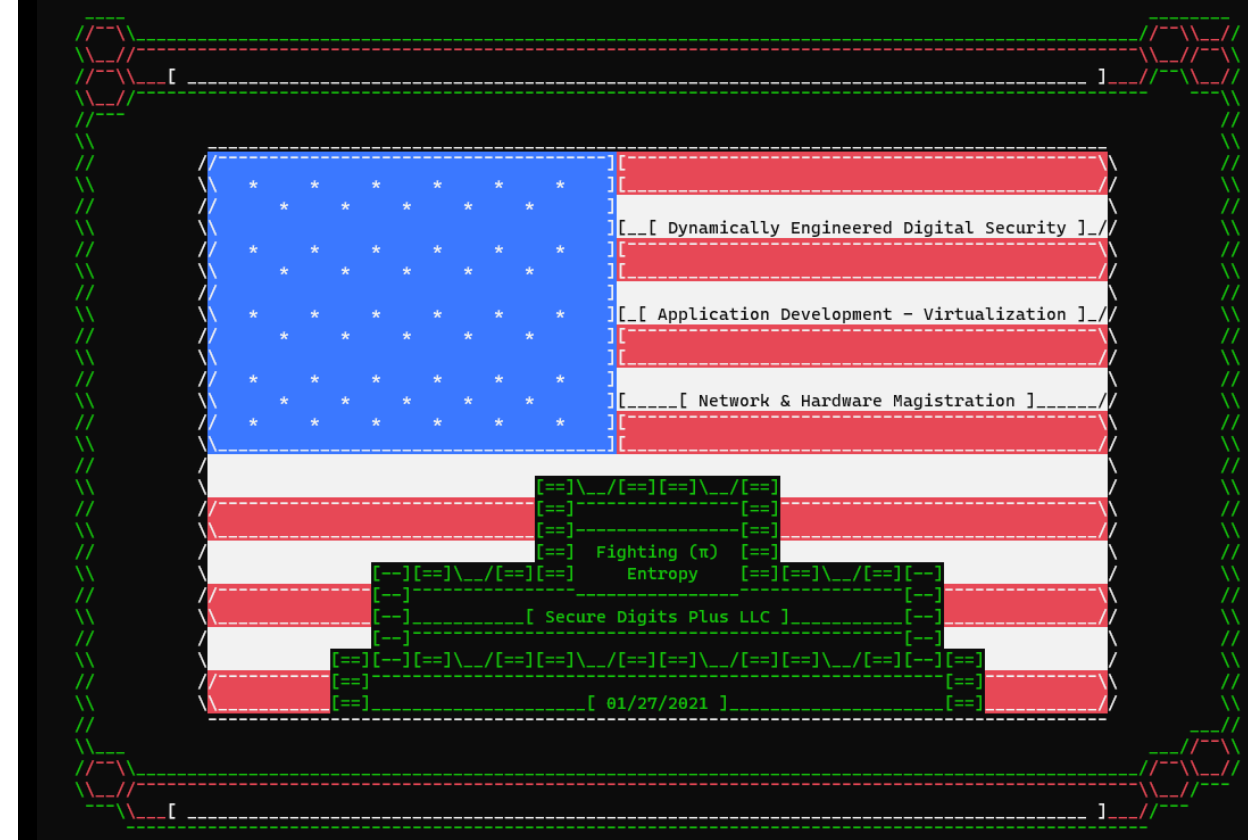
Anyway, this function isn't complete. Menu options, tagging, collecting user input isn't needed nor applied... but it certainly could be. The menu links don't go to web pages or message boxes like they do in the other utilities I have completed.

There's a great chance that the **GUI** you see in this lesson... Well, at any point in time, it could easily be scrapped and rebuilt from scratch. Since the **XAML** objects cover another shade of complexity, I'll reserve the remainder of this tutorial for where the **GUI** gets its controls all rigged up.

If you want to scope out the entire script, the URL is here...

https://github.com/mcc85sx/FightingEntropy/blob/master/2021.1.1/Functions/Get-FENetwork.ps1
The function Write-Theme is in that same folder…
https://github.com/mcc85sx/FightingEntropy/blob/master/2021.1.1/Functions/Write-Theme.ps1

```
PS C:\Users\mcook85> Write-Theme -Flag
```




```
PS C:\Users\mcook85> Write-Theme -Banner
```


```
PS C:\Users\mcook85> Write-Theme "Peace out, (boy/girl) scout..." 11,4,15,0
```