2018

A Historical Architectural Tour of PowerShell (Part 1)

Bruce Payette
Microsoft Corp.
@BrucePayette

PowerShell Conference EU 2018

Agenda

- This is round 1 of a 2 round set of talks.

```
compress-archive 16years -DestinationPath 90minutes
```

- In this session, we're going to trace PowerShell's evolution from the beginning up to shipping V1 in 2006
- We'll look at some of the technical, cultural and business forces that drove this evolution
- The goal of these talks is both to present and to gather interesting (and hopefully useful) historical aspects of PowerShell's evolution.
  - The plan is to put all of this information into the GitHub repo so it will be available to everyone.
  - So if you have suggestions, anecdotes, questions or complaints, please let me know.

PowerShell Conference EU 2018

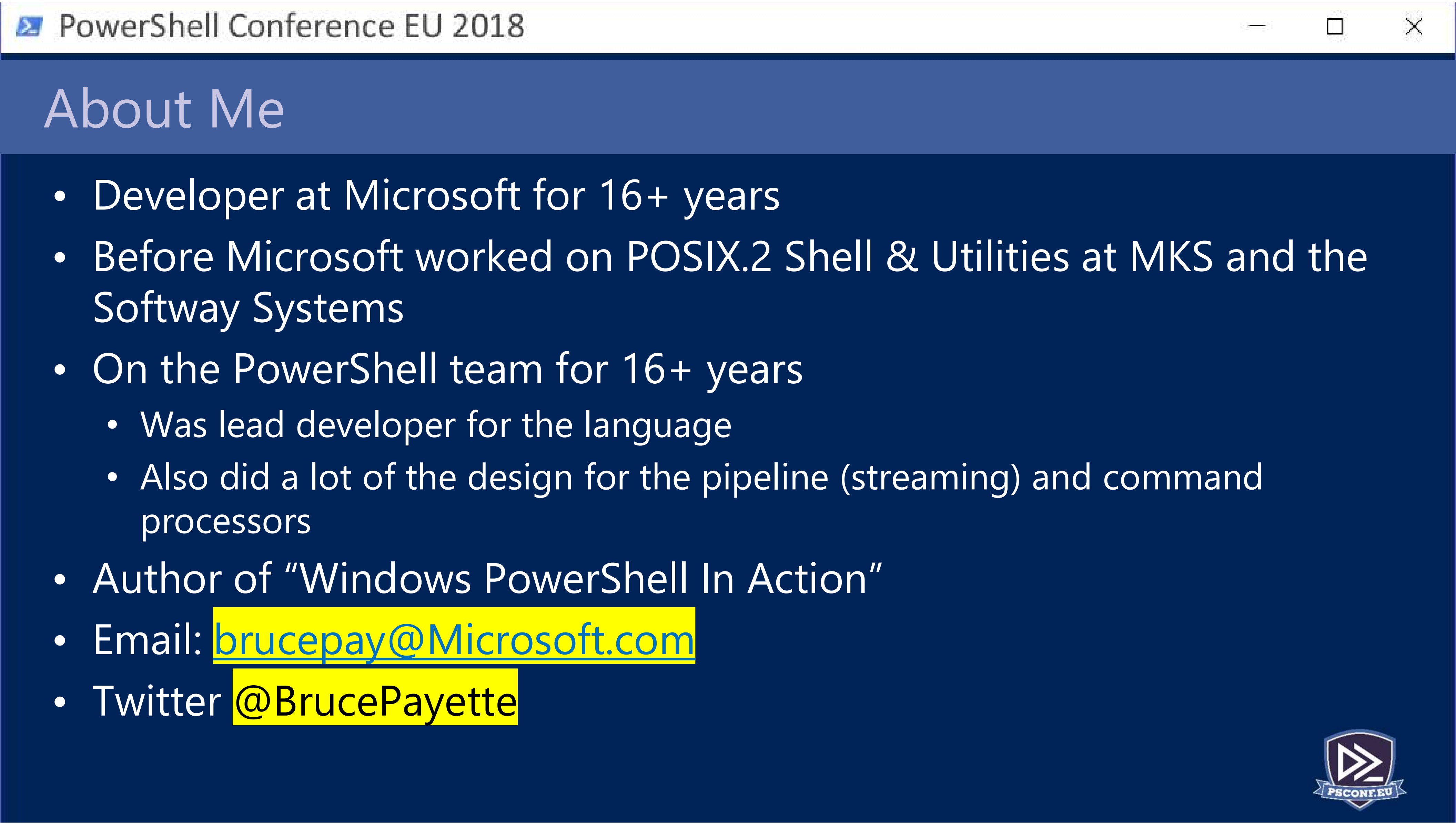
Architecture (and Beer) Makes #JoeyHappy

PSCONF.EU

This is important stuff!

- We thought about (almost) everything in PowerShell
  - *Very little is arbitrary; most everything is intentional*
  - Doesn’t mean we got it all right
  - Doesn’t mean we thought *of* everything…

Those who forget the past are doomed to relive it.



PowerShell Conference EU 2018

About Me

- Developer at Microsoft for 16+ years
- Before Microsoft worked on POSIX.2 Shell & Utilities at MKS and the Softway Systems
- On the PowerShell team for 16+ years
  - Was lead developer for the language
  - Also did a lot of the design for the pipeline (streaming) and command processors
- Author of “Windows PowerShell In Action”
- Email: brucepay@Microsoft.com
- Twitter @BrucePayette

In the beginning…

- (Power)Shell development started in 2001
  - A project was funded out of the India Development Center (IDC) to investigate improving the shell experience on Windows.
  - Code name was *Kermit* (for the hermit crab who lost his shell)
  - Staffed with people from the Interix/Services for Unix (SFU) team with Unix shell and utilities experience
- Why was this funded?
  - Technology driver: Survey Says: Automation on Windows was hard
    - 10x effort, 10x time to achieve the same goals as on Unix
  - Business driver: Unix esp. Linux and the LAMP stack were hot (competition)
  - Culture(?): 1 server wasn’t enough and automation was key to managing large (and getting larger) groups of servers.
- But where’s Jeffrey in all this activity ???

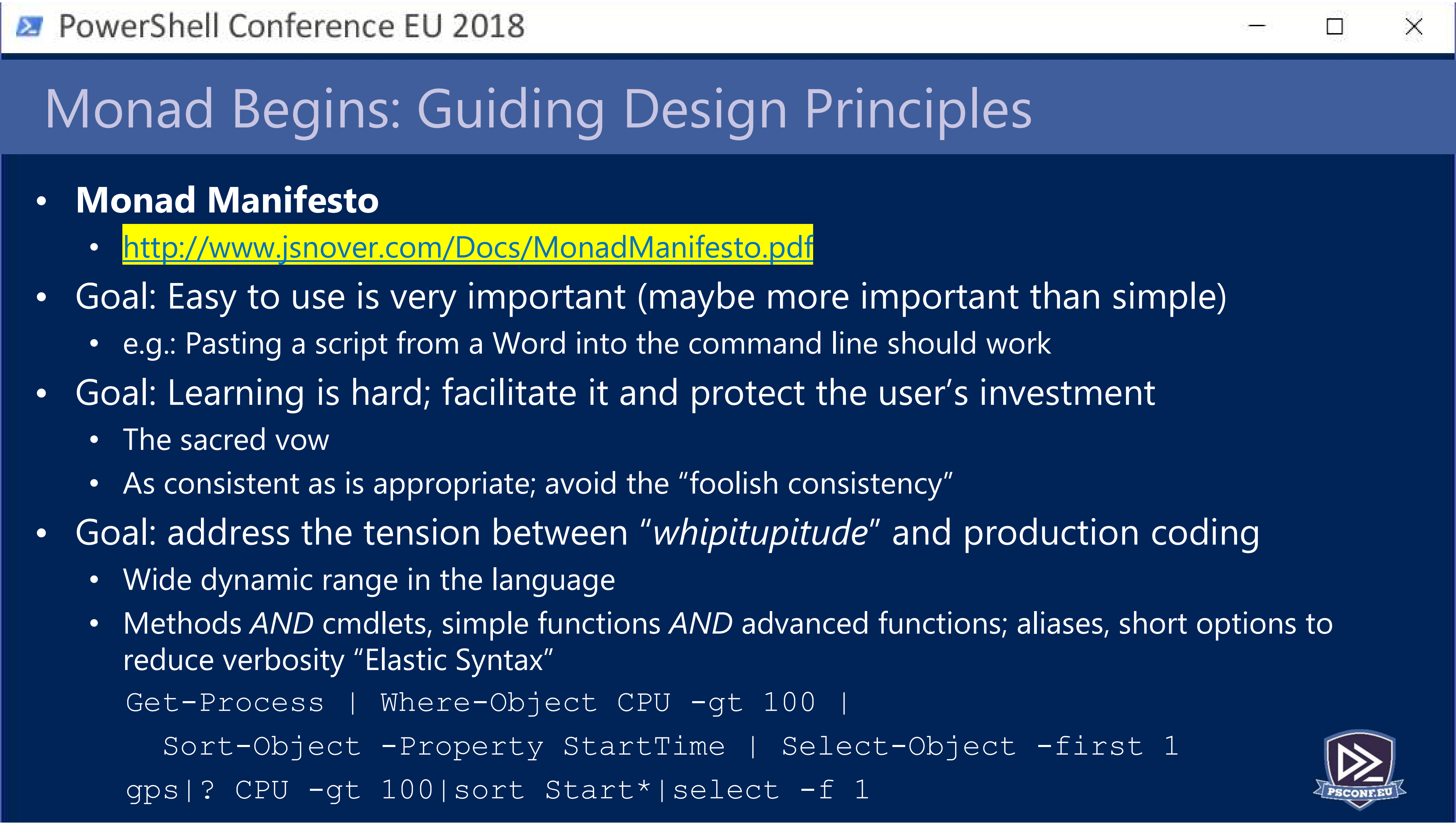
Meanwhile, in Building 42…

- Jeffrey was madly building a prototype of a command line experience based on a radically new set of concepts
  - A command language based on the composition of Functional Units (or FU for short) and an extended mutable type system
- The prototype used a different model of composition compared to what we eventually shipped in PowerShell.
  - An FU was an “extension” of the command before it which looked something like:

    ```
    process/get /name:c* /sort: CPU /select: 5
    ```

  - Instead of the final PowerShell syntax (note: order matters)

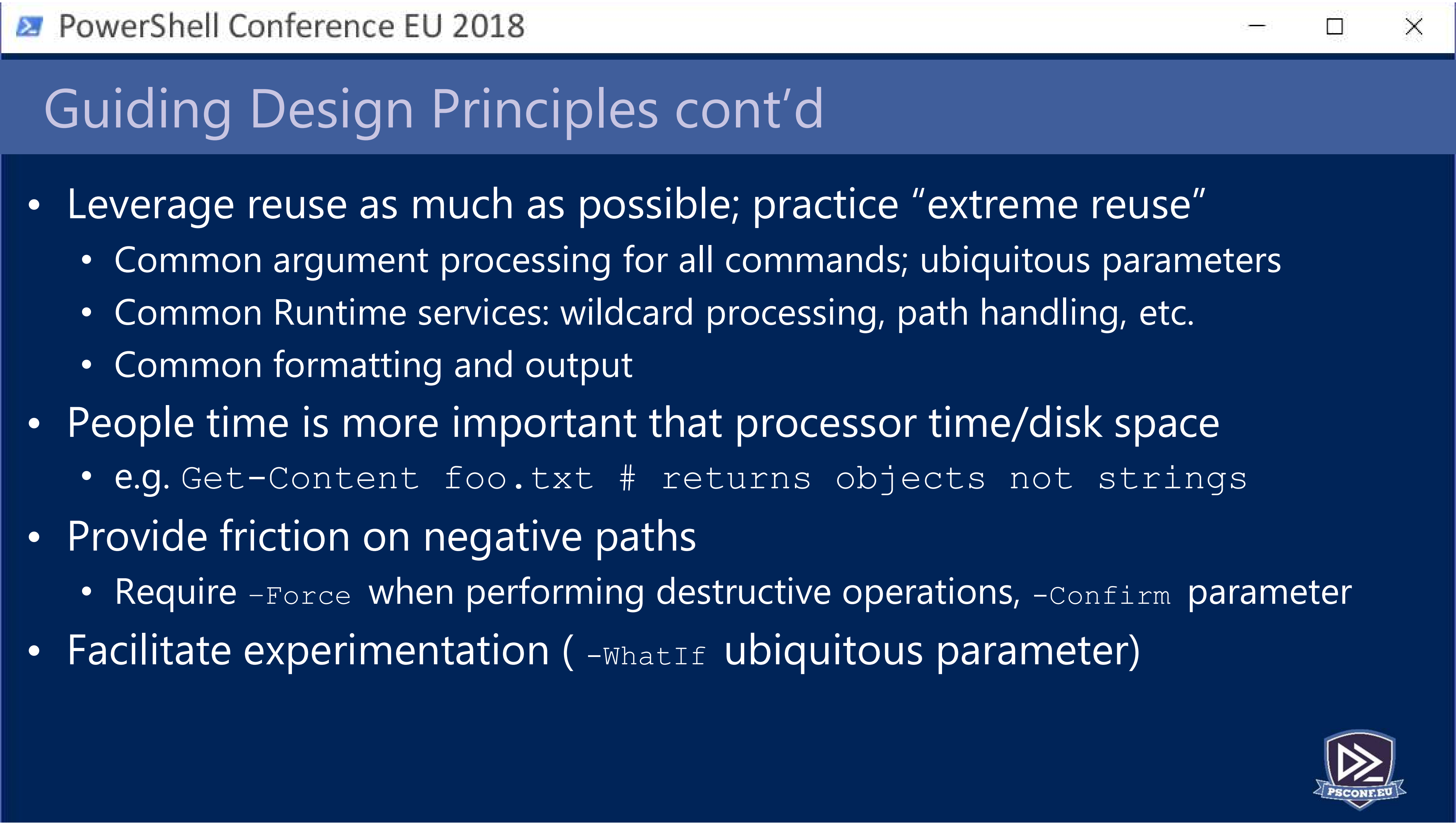
    ```
    Get-Process -Name c* | sort CPU | select -first 5
    ```



Monad Begins: Guiding Design Principles

- **Monad Manifesto**
  - http://www.jsnover.com/Docs/MonadManifesto.pdf
- Goal: Easy to use is very important (maybe more important than simple)
  - e.g.: Pasting a script from a Word into the command line should work
- Goal: Learning is hard; facilitate it and protect the user's investment
  - The sacred vow
  - As consistent as is appropriate; avoid the "foolish consistency"
- Goal: address the tension between "*whipitupitude*" and production coding
  - Wide dynamic range in the language
  - Methods *AND* cmdlets, simple functions *AND* advanced functions; aliases, short options to reduce verbosity "Elastic Syntax"

```
Get-Process | Where-Object CPU -gt 100 |
  Sort-Object -Property StartTime | Select-Object -first 1
gps|? CPU -gt 100|sort Start*|select -f 1
```



PowerShell Conference EU 2018

Guiding Design Principles cont'd

- Leverage reuse as much as possible; practice "extreme reuse"
  - Common argument processing for all commands; ubiquitous parameters
  - Common Runtime services: wildcard processing, path handling, etc.
  - Common formatting and output
- People time is more important that processor time/disk space
  - e.g. `Get-Content foo.txt # returns objects not strings`
- Provide friction on negative paths
  - Require `-Force` when performing destructive operations, `-Confirm` parameter
- Facilitate experimentation (`-WhatIf` ubiquitous parameter)

Project begins: we acquire a team…

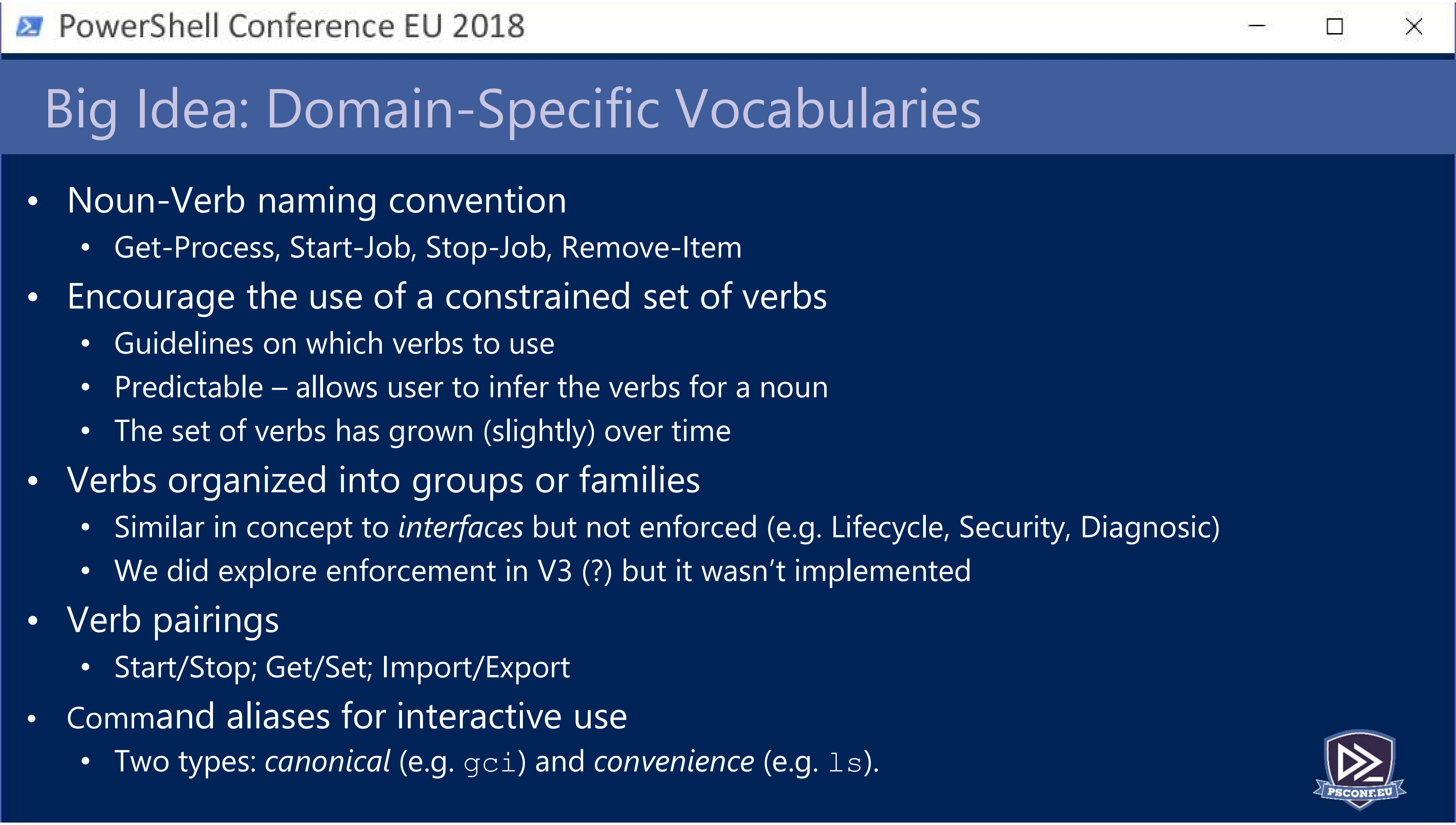
- Staffed out of IDC in Hyderabad, India
  - Architecture owners in Redmond
  - Development and Test to be done in India
- But radical experimentation does not flourish with this level of geographic distribution
  - It was taking too long to round-trip ideas
  - First development then test were pulled back to Redmond
- We acquire another team
  - Built from the Windows Management Pack team so lots of Windows management experience.
  - Now we can begin in earnest…
- We change buildings. And again. And again. (6x in 2 years)

Oh, and be ready to ship in 2 weeks…

- The *Longhorn time frame*
  - Microsoft was dipping a toe into the idea of continuous delivery
  - It was an exciting time…
- This meant that the initial release plan for PowerShell wouldn’t have been much more than Jeffrey’s prototype slightly cleaned up
  - No performance work
  - No coverage
  - No ability to write scripts or functions
  - There would have only been an interactive experience
- This was not ideal.
- Fortunately it didn’t happen ☺

We Had Big Ideas

- “Big Ideas” represent some of the major design elements of PowerShell
  - Concrete design decisions rather than abstract principles



PowerShell Conference EU 2018

Big Idea: Domain-Specific Vocabularies

- Noun-Verb naming convention
  - Get-Process, Start-Job, Stop-Job, Remove-Item
- Encourage the use of a constrained set of verbs
  - Guidelines on which verbs to use
  - Predictable – allows user to infer the verbs for a noun
  - The set of verbs has grown (slightly) over time
- Verbs organized into groups or families
  - Similar in concept to *interfaces* but not enforced (e.g. Lifecycle, Security, Diagnosic)
  - We did explore enforcement in V3 (?) but it wasn't implemented
- Verb pairings
  - Start/Stop; Get/Set; Import/Export
- Command aliases for interactive use
  - Two types: *canonical* (e.g. `gci`) and *convenience* (e.g. `ls`).

Big Idea: Universal command-line parsing

- Unlike most shells, cmdlets are not responsible for parsing their own parameters
- Common parameter parsing code shared by all types of commands
  - Command-specific parameter *binding*

```
…\src\System.Management.Automation\engine\*parameterbinder*.cs
```

- This gives us the broad consistency across commands
  - Except for native commands ☹ which do their own argument parsing
  - Native commands on Unix are pretty regular, commands on Windows – not so much
  - Native command parameter binder takes the parsed parameters and tries to put them back into something like what the user typed. Sometimes it works, sometimes it doesn’t

PowerShell Conference EU 2018

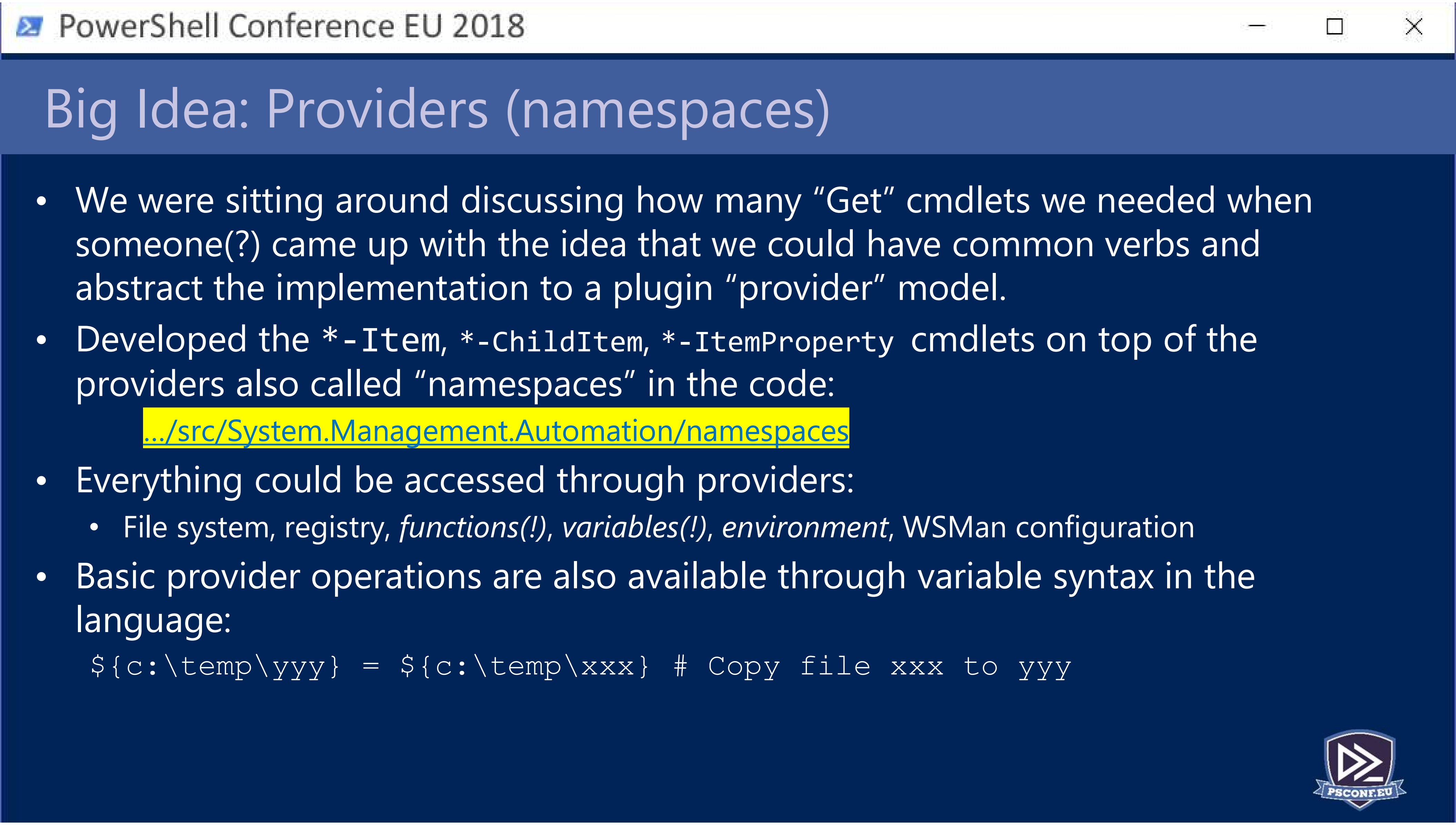
Big Idea: Declarative Parameter Constraints

- Rather than writing imperative code to do checking, use declarative attributes to deal with constraints:
  - `[ValidateIsNotNullOrEmpty]`, `[ValidateRange()]`, `[ValidateSet()]`
- These attributes both simplified the code and resulted in a consistent experience (i.e. common error messages)
- Sometimes the consistent experience could be suboptimal
  - e.g. with `[ValidatePattern(“[0-9]+”)]` the error message would say “failed to match pattern [0-9]+” instead of “number expected”
- And sometimes it just didn’t work, per this comment:

…\src\Microsoft.PowerShell.Commands.Management\commands\management\Process.cs

```
// 2004/12/17-XXXX ProcessNameGlobAttribute was deeply wrong.
// For example, if you pass in a single Process, it will match
// all processes with the same name.
// I have removed the globbing code.
```

PSCONF.EU



PowerShell Conference EU 2018

Big Idea: Providers (namespaces)

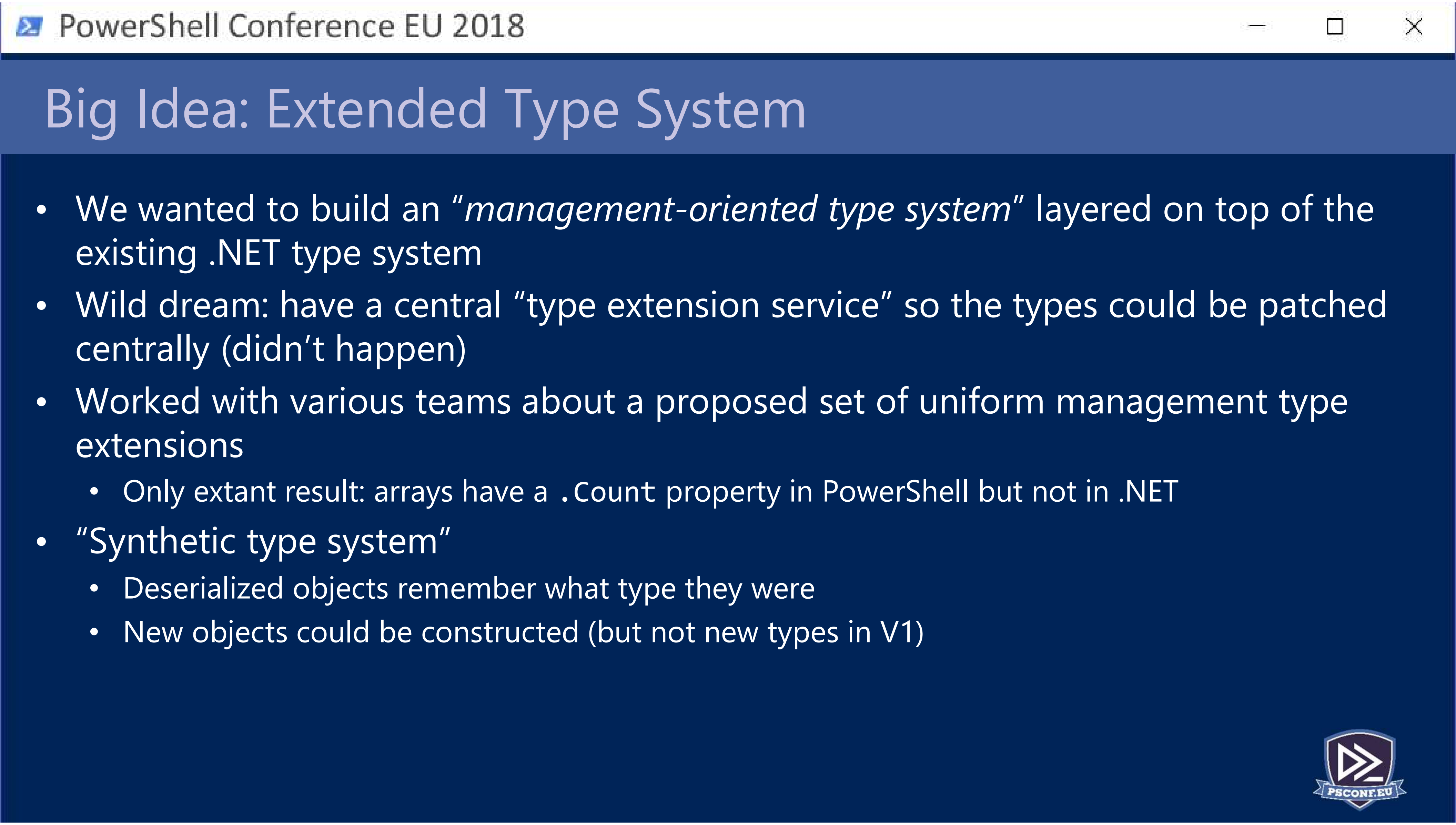
- We were sitting around discussing how many “Get” cmdlets we needed when someone(?) came up with the idea that we could have common verbs and abstract the implementation to a plugin “provider” model.
- Developed the `*-Item`, `*-ChildItem`, `*-ItemProperty` cmdlets on top of the providers also called “namespaces” in the code:

  .../src/System.Management.Automation/namespaces
- Everything could be accessed through providers:
  - File system, registry, *functions(!)*, *variables(!)*, *environment*, WSMan configuration
- Basic provider operations are also available through variable syntax in the language:

```
${c:\temp\yyy} = ${c:\temp\xxx} # Copy file xxx to yyy
```

Big Idea: Streaming

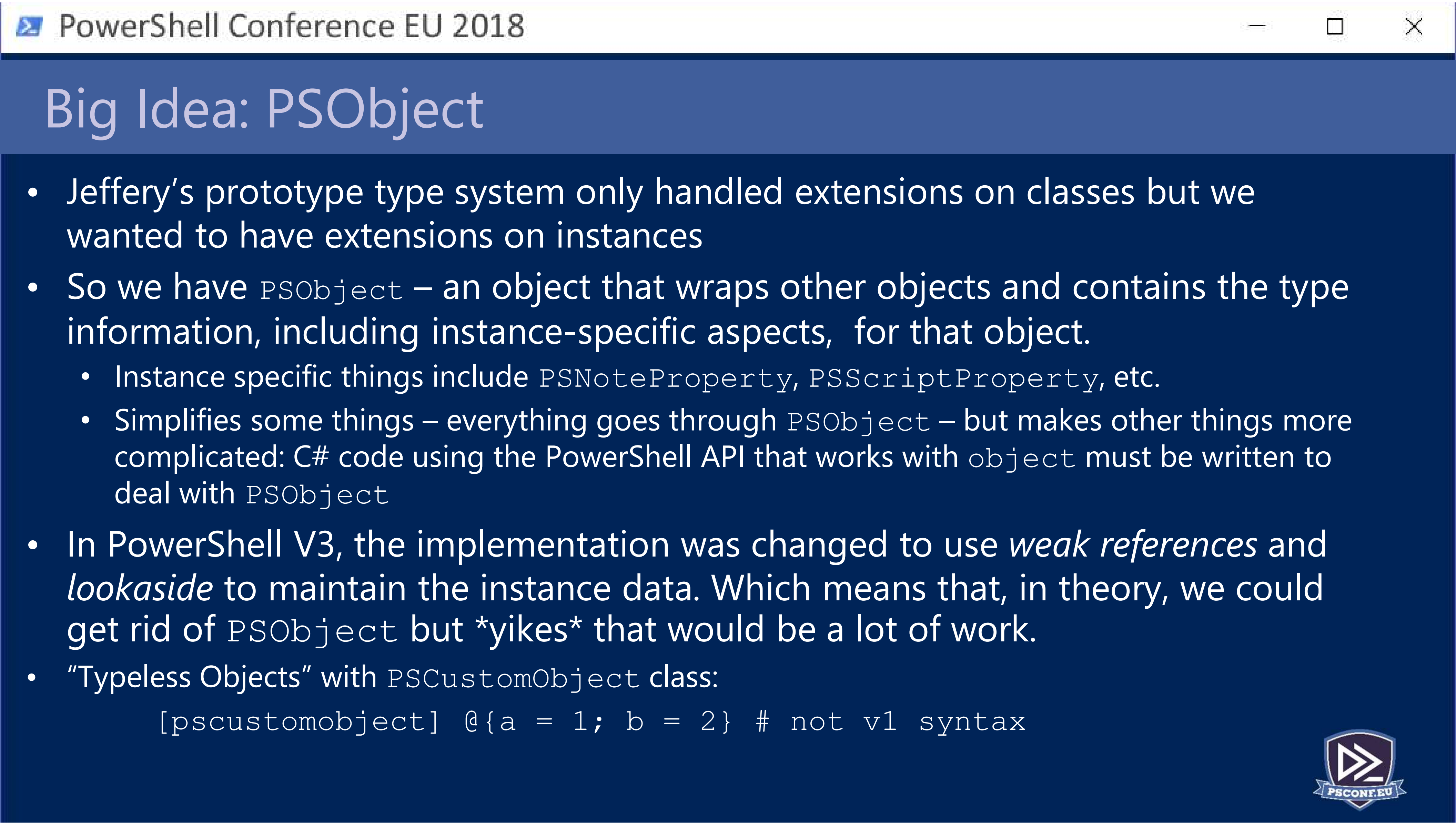
- The Unix shell pipeline was definitely a feature we wanted in PowerShell
  - But Unix used multiple processes to do this; commands were external executables
  - Commands in PowerShell all run *in-process* with the engine so the traditional approach couldn’t work
- Next we looked at using 1 thread per command
  - More light weight than processes but still heavy
- Solution was to sequence objects through the pipeline 1 at a time
- Problem (we still have today) is that when streaming, there is ambiguity as to whether or no the single object is scalar or an array.
  - Mitigate with @(command)
- Visible in the API as public class `SteppablePipeline`
  - Wraps the internal Pipeline class
  - Introduced in V2



PowerShell Conference EU 2018

Big Idea: Extended Type System

- We wanted to build an “*management-oriented type system*” layered on top of the existing .NET type system
- Wild dream: have a central “type extension service” so the types could be patched centrally (didn’t happen)
- Worked with various teams about a proposed set of uniform management type extensions
  - Only extant result: arrays have a `.Count` property in PowerShell but not in .NET
- “Synthetic type system”
  - Deserialized objects remember what type they were
  - New objects could be constructed (but not new types in V1)



PowerShell Conference EU 2018

Big Idea: PSObject

- Jeffery’s prototype type system only handled extensions on classes but we wanted to have extensions on instances
- So we have `PSObject` – an object that wraps other objects and contains the type information, including instance-specific aspects, for that object.
  - Instance specific things include `PSNoteProperty`, `PSScriptProperty`, etc.
  - Simplifies some things – everything goes through `PSObject` – but makes other things more complicated: C# code using the PowerShell API that works with `object` must be written to deal with `PSObject`
- In PowerShell V3, the implementation was changed to use *weak references* and *lookaside* to maintain the instance data. Which means that, in theory, we could get rid of `PSObject` but *yikes* that would be a lot of work.
- “Typeless Objects” with `PSCustomObject` class:

```
[pscustomobject] @{a = 1; b = 2} # not v1 syntax
```

PSCONF.EU

Error Handling

- Universal streams – Error, Warning, Verbose, Debug (, Logging)
- Live error logging in `$Error`
- Terminating vs Non-terminating errors
  - Non-terminating errors for shell-like semantics
  - NOT different types of errors; different dispositions (different *Error Actions*)
- Checking command execution status
  - Errors go to a stream – hard to test; check `$?` to see if a command had an error
  - `$LASTEXITCODE` for status of native commands
- Exceptions !!!!
  - trap statement – taken from Perl 6; like VB OnError statement
    - $_ is the error record, not the exception inside a trap
  - Lexically scoped; can't set a durable trap
  - `throw` statement

Break and Continue

- Break out of loops; continue in loops
- Can break to a labelled loop; *break label is an expression (!)*

```
$l = label ; :label while (1) { while (1) { break $l }
```

- Uncaught breaks propagate until they are caught
  - If they aren't caught, they stop the current execution
  - Yeah it's weird; limited utility
  - Somewhat like C's non-local goto `setjmp/longjmp`
- `break` and `continue` in traps
  - Continue in a `trap` resumes execution after the statement in the current context that caused the trap.

Big Idea: Security, Security, Security

- Secure by default
- Threat modelling
    - Thread, ***Asset***, Mitigation
    - We had some problems figuring out what an asset was
- Struggled with Code Access Security (CAS)
    - Was designed to allow trusted code to call untrusted code while restricting capabilities - too complicated to use
- For PowerShell, the trust boundary is at the runspace
    - Untrusted code must cross the runspace boundary (see part 2 of this talk.)
- ExecutionPolicy – *it's a safety belt, not a door look*
    - Reduces risk by reducing possibility of accidents

The Implementation

Implementation…

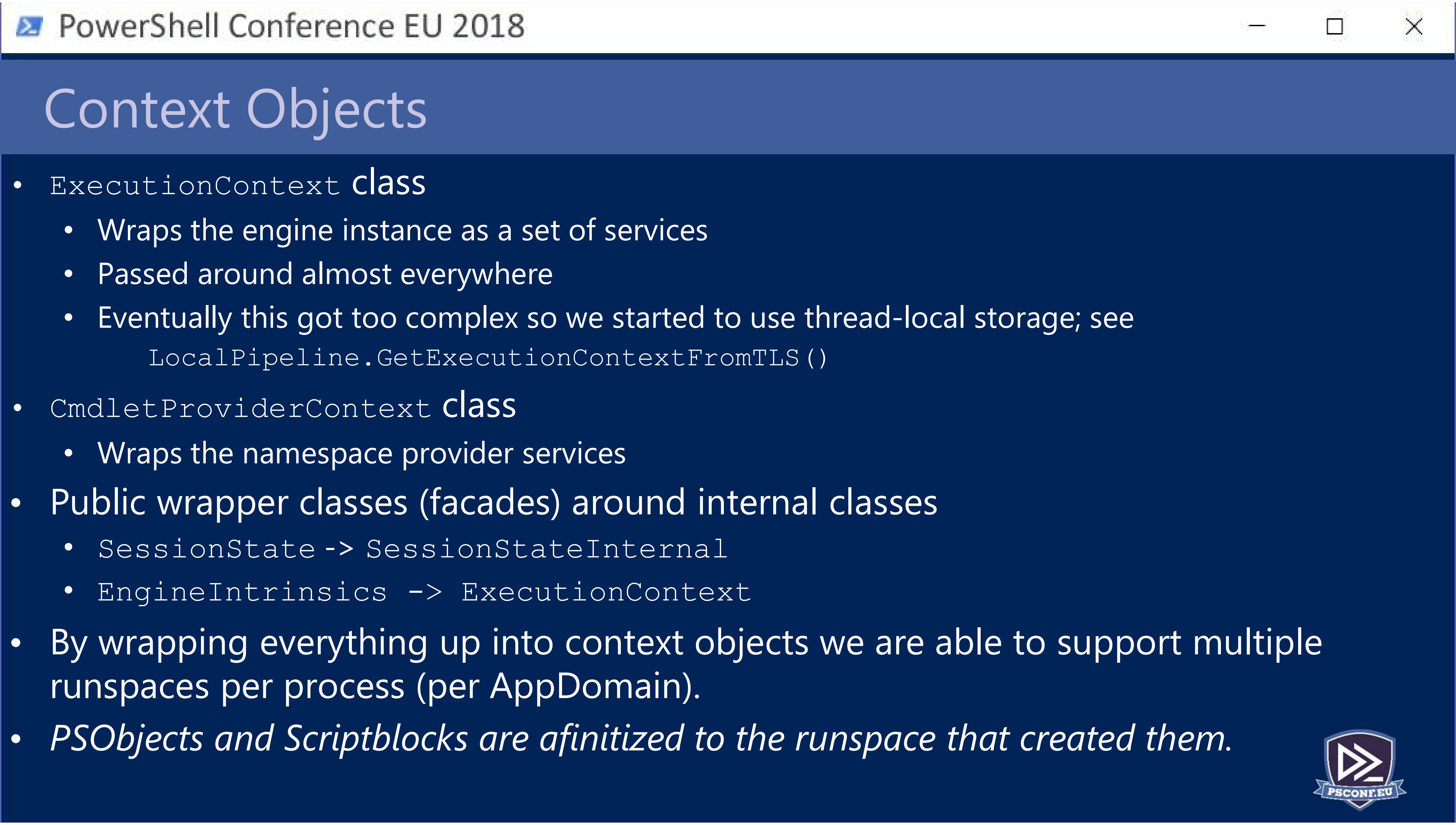
- Initial engine design was highly componentized; most of there components live in `…/src/System.Management.Automation/engine/`
- Some of the pieces:
  - Runspace configuration (supplanted by InitialSessionState in V2)
  - Language Tokenizer, Parser & Executor
  - Execution Context (includes session state)
  - Command Discovery (subsystem to find commands)
  - Pipeline processor
    - Command processors
  - Error handling subsystem
- With this design you had to build your own shell – high barrier to entry
- To simplify the API, everything was bundled up into ***Runspaces***.
  - A Runspace is a space where you run things. Why is that so hard?

Followed .NET Guidelines

- Followed .NET guidelines *as they existed at the time*
- Sealed all classes that weren’t explicitly extensible
  - Intended to mitigate versioning
  - Kind of a pain; derivation is useful
- Made everything internal unless it was a explicit API
  - Reduce support costs; `LanguagePrimitives` should have been public
- APIs that return collections should never return null
  - return empty collection instead.
- Favor generic collections over polymorphic collections
  - `List<int>` instead of `ArrayList`
- We also broke a bunch over rules (see pragmas in code)
  - e.g. “Properties should not return array”

Lots of (Informal) Use of Design Patterns

- We used a lot of OO design patterns in the code
  - *Prefer composition over inheritance*
- Adapter pattern
  - Type Adapters (Extension point)
- Façade pattern
  - *Command Processors* – hides the implementation differences in various command types
    - Defined (but not public) extension point; other command types could be added
- Strategy
  - Command Parameter Binders
- Factory
  - `RunspaceFactory`



PowerShell Conference EU 2018

Context Objects

- `ExecutionContext` class
  - Wraps the engine instance as a set of services
  - Passed around almost everywhere
  - Eventually this got too complex so we started to use thread-local storage; see
    `LocalPipeline.GetExecutionContextFromTLS()`
- `CmdletProviderContext` class
  - Wraps the namespace provider services
- Public wrapper classes (facades) around internal classes
  - `SessionState -> SessionStateInternal`
  - `EngineIntrinsics -> ExecutionContext`
- By wrapping everything up into context objects we are able to support multiple runspaces per process (per AppDomain).
- *PSObjects and Scriptblocks are afinitized to the runspace that created them.*

PowerShell Conference EU 2018

The PowerShell Language

The Language

- In the original architecture, the language was conceived as being separate from the PowerShell engine.
  - In theory, you would be able to substitute difference languages
- In practice, over time the language and the engine became deeply intermixed
  - E.g. the engine uses formatting and output which can have script properties which requires the language
- Starting with version 2, you can *use* PowerShell from other languages using the PowerShell API.

PowerShell Conference EU 2018 — □ ×

Language roots…

- Started with the IEEE POSIX.2 shell grammar (essentially the Korn Shell (ksh))
  - From the shell we got `$variables`, `$( subexpression ), $?,` pipelines (|), dot-sourcing, *direct execution* of external programs
- Parameter syntax inspired, in part, by DCL (DEC Command Language)

```
Get-Something -parameter: $argument
```

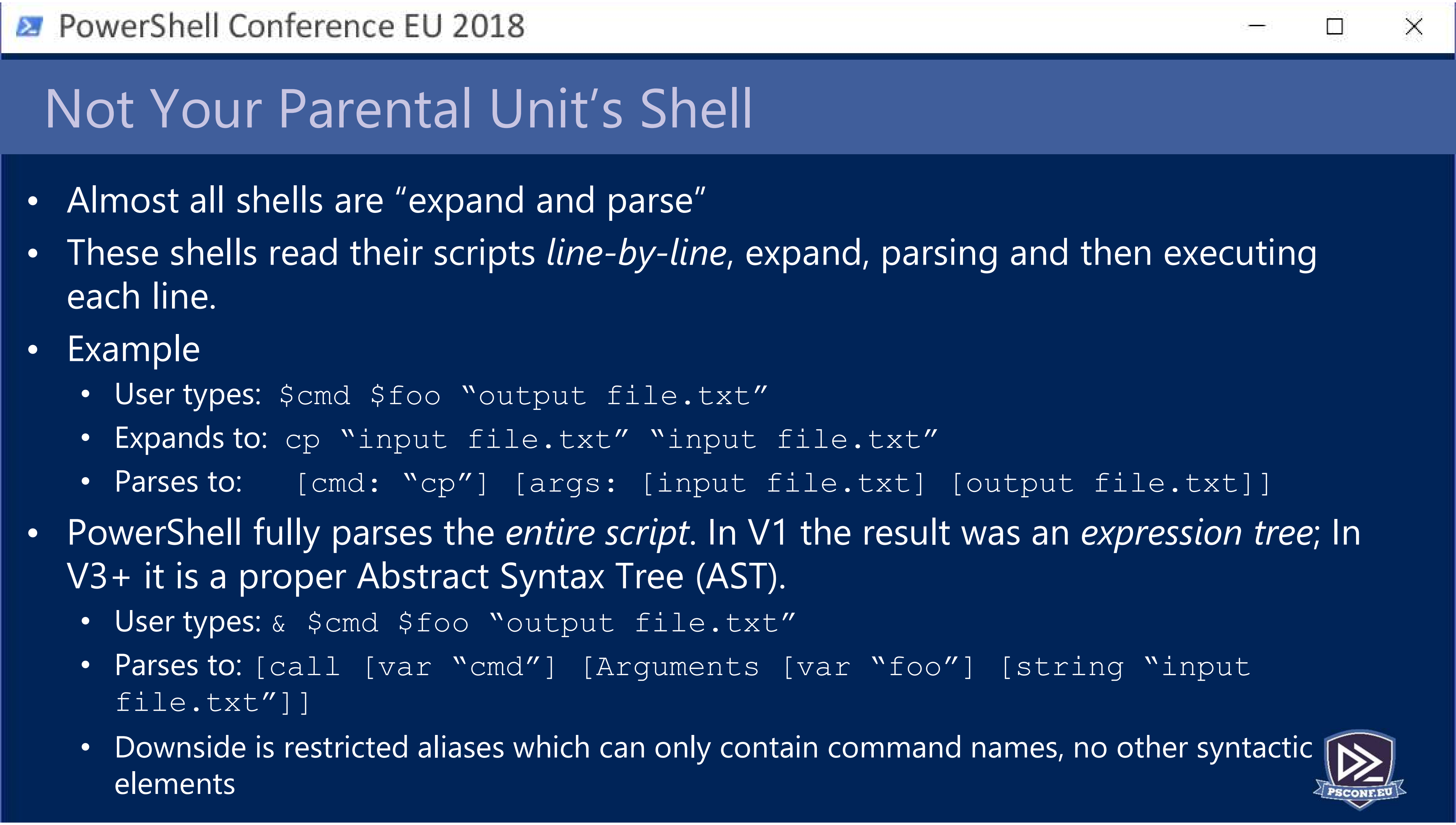
- For concepts that weren't in the POSIX Shell, we started by adapting elements of Perl:
  - Perl was pretty much the dominant scripting language on the Internet.
  - Perl had arrays, hashtables, regular expression all of which we wanted
  - Super significant – Perl had CPAN (Comprehensive Perl Archive Network)
  - Syntax elements we retained from PERL
    - $_, the use of @ in places (@(), @(array subexpression)), & call operator, regular expressions

The language evolves…

- But in the end, Perl was kind of … ummm … icky

  “Line noise should not compile”

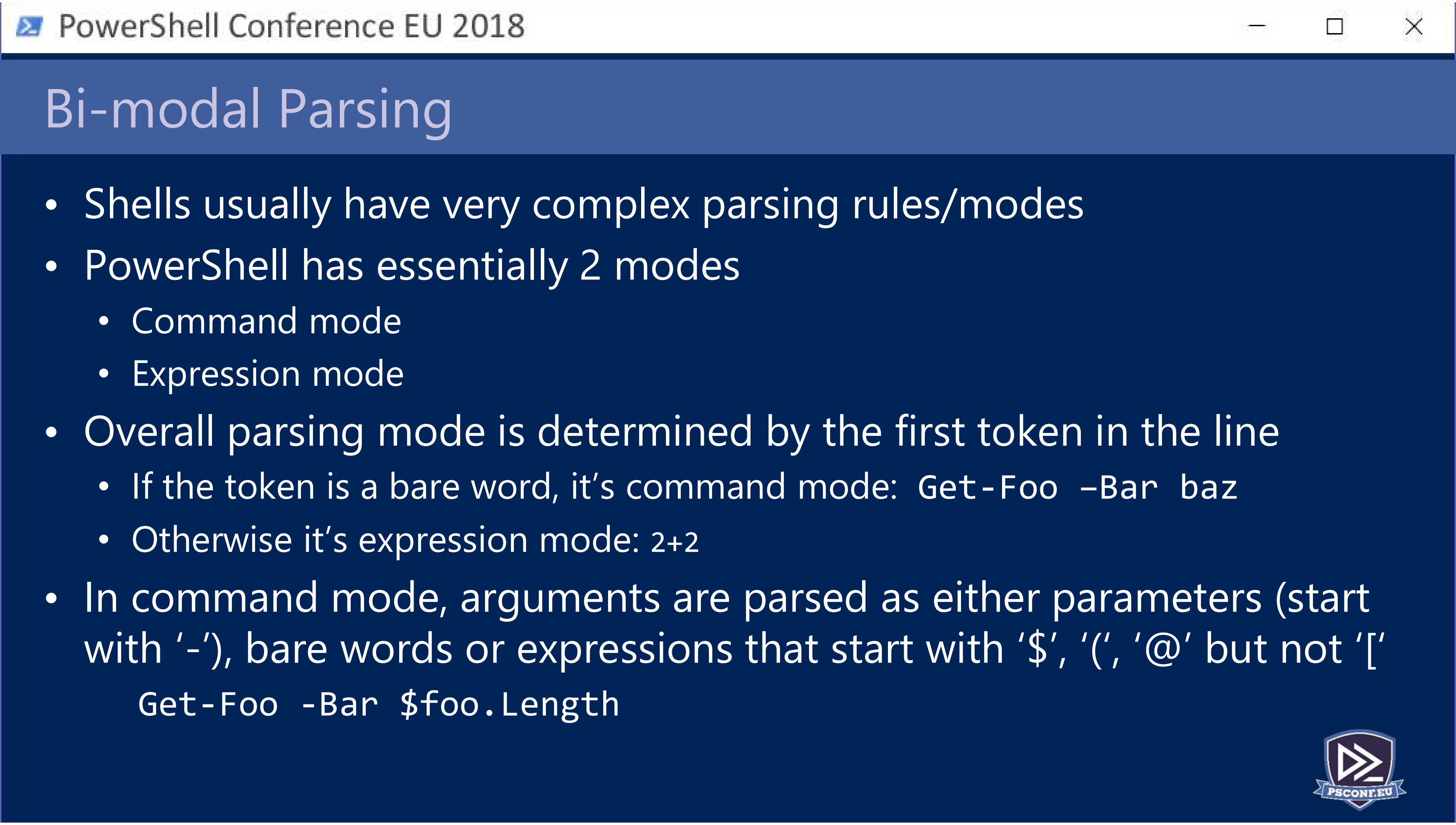
- So we switched our syntax model to align with C#
  - which essentially means aligning, to a greater or lesser extent, with C, C++, Java, AWK, Go, CShell, Objective-C, PHP, PERL etc.
- The value proposition became, in part: if you learned PowerShell, you could move to C# pretty easily and if you knew C#, you could pick up PowerShell pretty easily.
  - Protects the student’s investment
  - Not sure how well that worked sometimes ☺



PowerShell Conference EU 2018

Not Your Parental Unit’s Shell

- Almost all shells are “expand and parse”
- These shells read their scripts *line-by-line*, expand, parsing and then executing each line.
- Example
  - User types: `$cmd $foo “output file.txt”`
  - Expands to: `cp “input file.txt” “input file.txt”`
  - Parses to: `[cmd: “cp”] [args: [input file.txt] [output file.txt]]`
- PowerShell fully parses the *entire script*. In V1 the result was an *expression tree*; In V3+ it is a proper Abstract Syntax Tree (AST).
  - User types: `& $cmd $foo “output file.txt”`
  - Parses to: `[call [var “cmd”] [Arguments [var “foo”] [string “input file.txt”]]`
  - Downside is restricted aliases which can only contain command names, no other syntactic elements

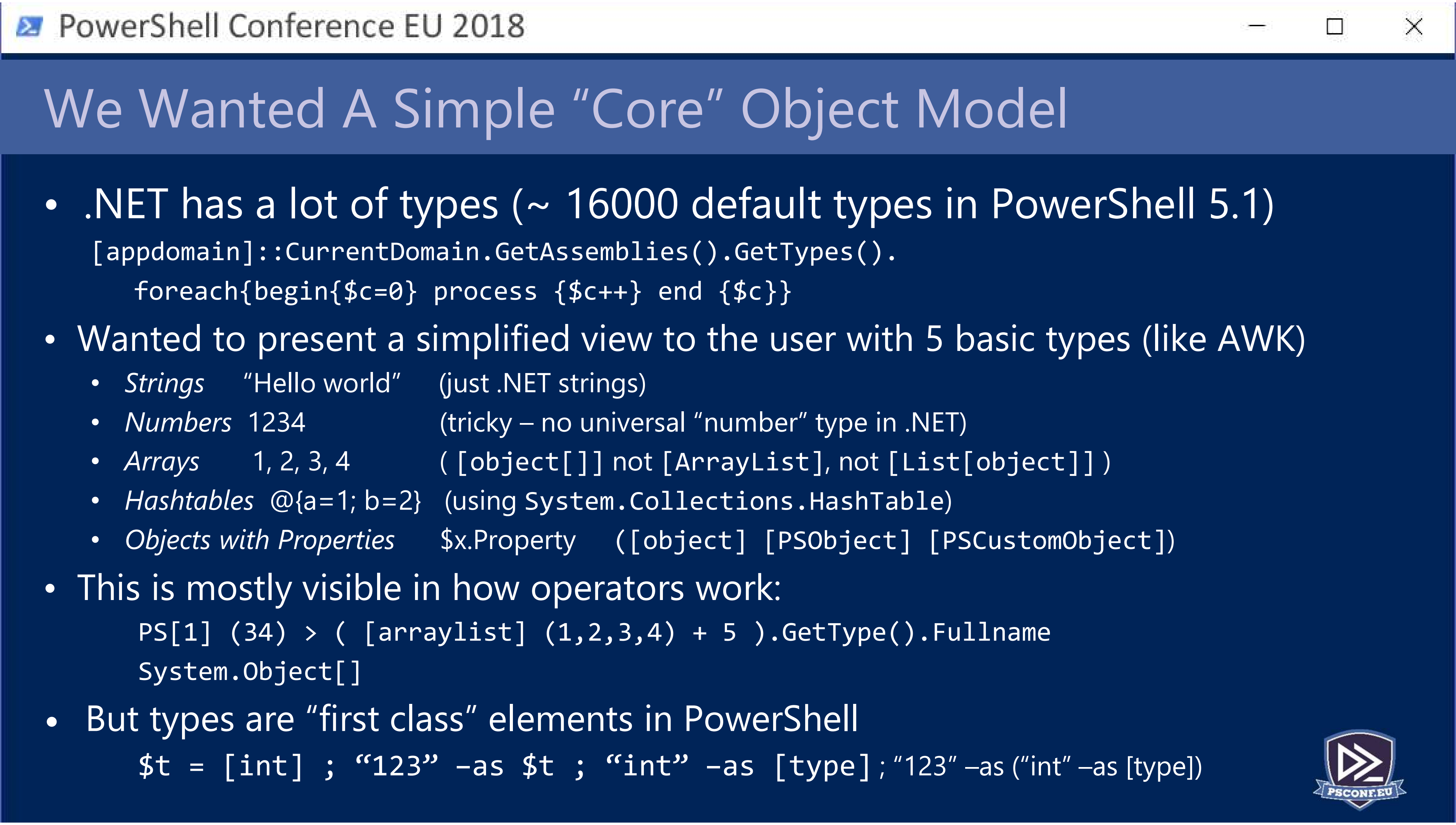


PowerShell Conference EU 2018

Bi-modal Parsing

- Shells usually have very complex parsing rules/modes
- PowerShell has essentially 2 modes
  - Command mode
  - Expression mode
- Overall parsing mode is determined by the first token in the line
  - If the token is a bare word, it's command mode: `Get-Foo –Bar baz`
  - Otherwise it's expression mode: `2+2`
- In command mode, arguments are parsed as either parameters (start with '-'), bare words or expressions that start with '$', '(', '@' but not '['

```
Get-Foo -Bar $foo.Length
```



PowerShell Conference EU 2018

We Wanted A Simple “Core” Object Model

- .NET has a lot of types (~ 16000 default types in PowerShell 5.1)

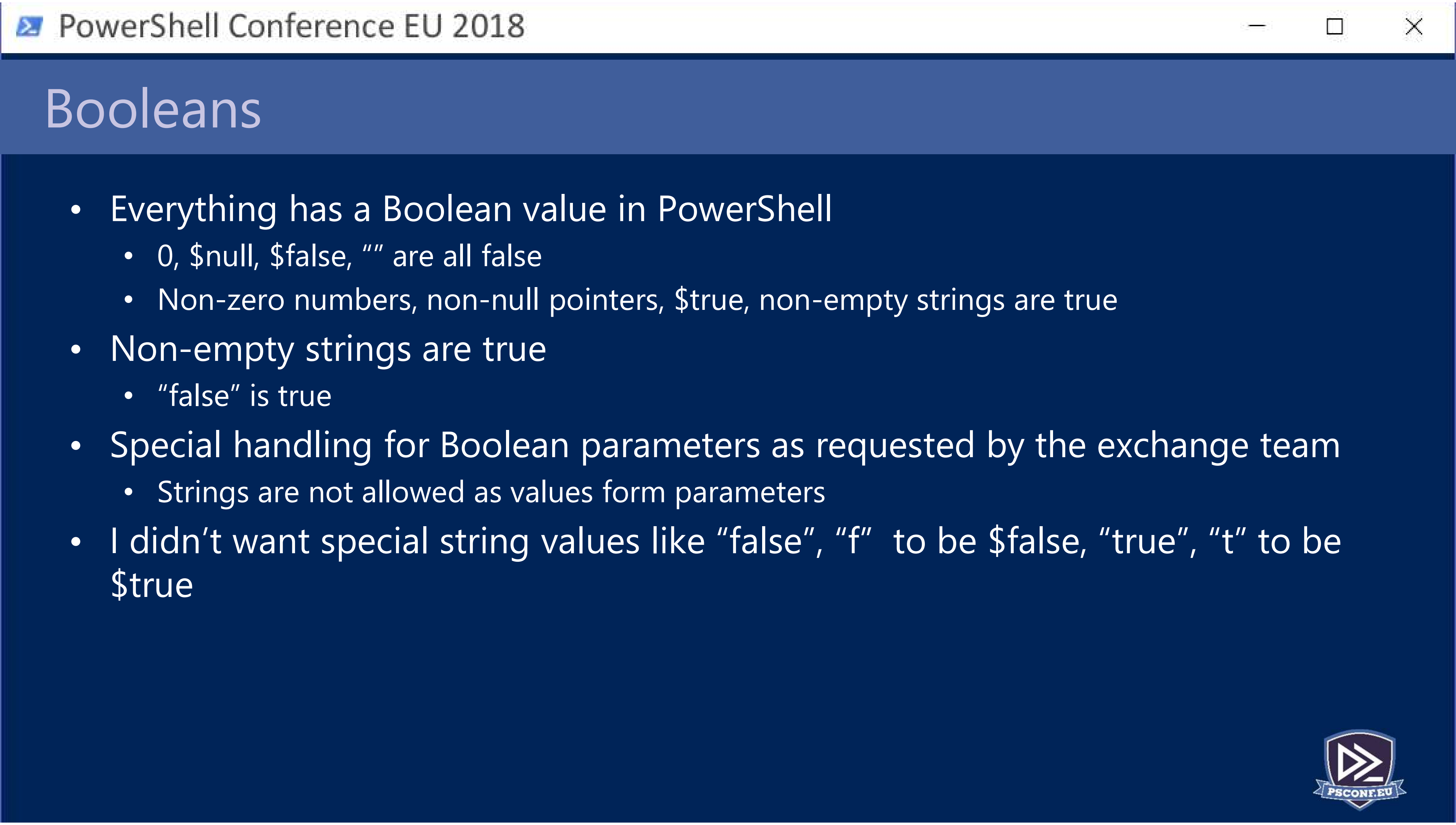
```
[appdomain]::CurrentDomain.GetAssemblies().GetTypes().
   foreach{begin{$c=0} process {$c++} end {$c}}
```

- Wanted to present a simplified view to the user with 5 basic types (like AWK)
  - *Strings* “Hello world” (just .NET strings)
  - *Numbers* 1234 (tricky – no universal “number” type in .NET)
  - *Arrays* 1, 2, 3, 4 (`[object[]]` not `[ArrayList]`, not `[List[object]]`)
  - *Hashtables* @{a=1; b=2} (using `System.Collections.HashTable`)
  - *Objects with Properties* $x.Property (`[object] [PSObject] [PSCustomObject]`)
- This is mostly visible in how operators work:

```
PS[1] (34) > ( [arraylist] (1,2,3,4) + 5 ).GetType().Fullname
System.Object[]
```

- But types are “first class” elements in PowerShell

```
$t = [int] ; “123” -as $t ; “int” -as [type] ; “123” –as (“int” –as [type])
```

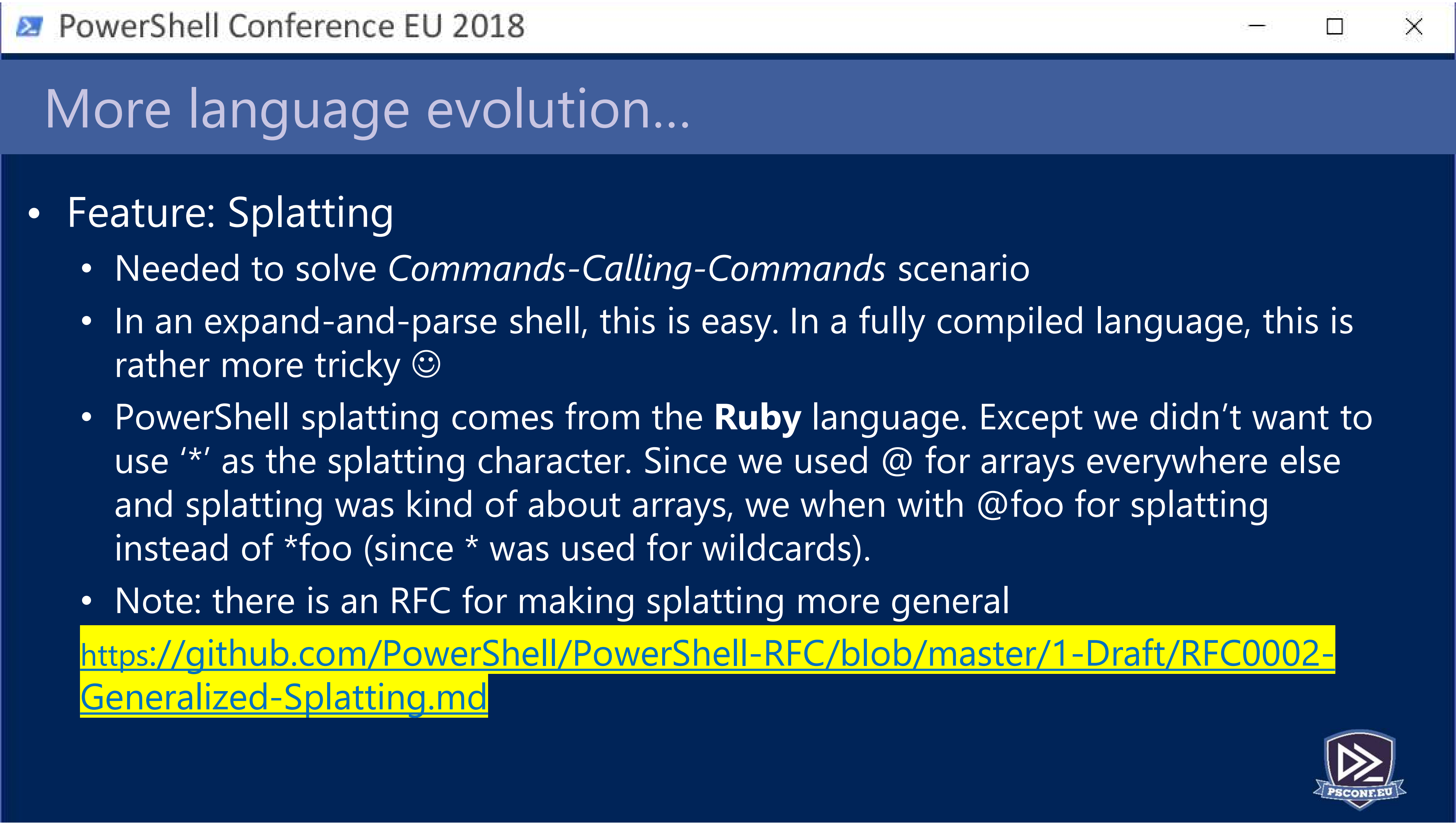


PowerShell Conference EU 2018

Booleans

- Everything has a Boolean value in PowerShell
  - 0, $null, $false, “” are all false
  - Non-zero numbers, non-null pointers, $true, non-empty strings are true
- Non-empty strings are true
  - “false” is true
- Special handling for Boolean parameters as requested by the exchange team
  - Strings are not allowed as values form parameters
- I didn’t want special string values like “false”, “f” to be $false, “true”, “t” to be $true

PSCONF.EU



More language evolution…

- Feature: Splatting
  - Needed to solve *Commands-Calling-Commands* scenario
  - In an expand-and-parse shell, this is easy. In a fully compiled language, this is rather more tricky ☺
  - PowerShell splatting comes from the **Ruby** language. Except we didn't want to use '*' as the splatting character. Since we used @ for arrays everywhere else and splatting was kind of about arrays, we when with @foo for splatting instead of *foo (since * was used for wildcards).
  - Note: there is an RFC for making splatting more general

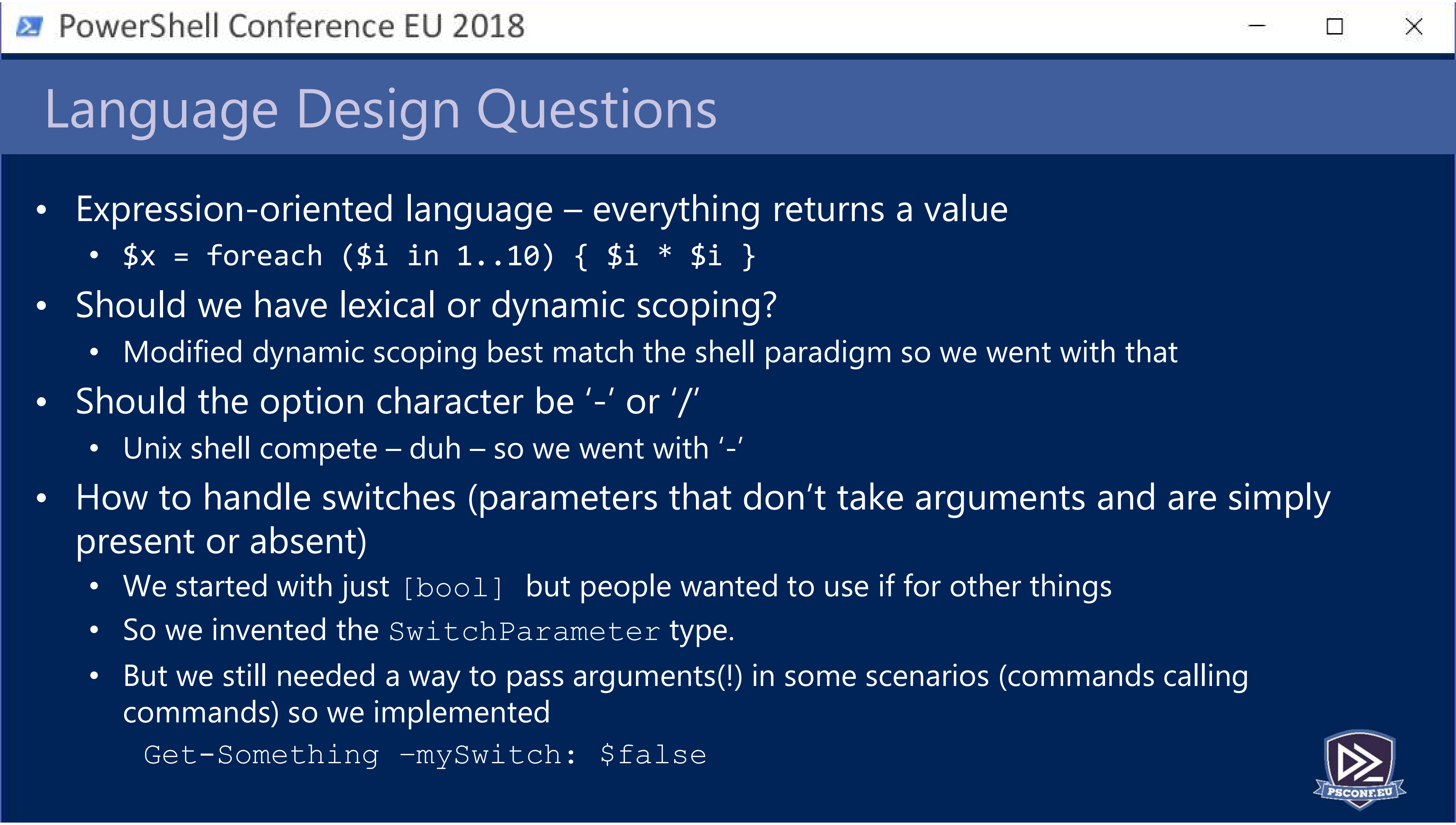
https://github.com/PowerShell/PowerShell-RFC/blob/master/1-Draft/RFC0002-Generalized-Splatting.md

Scriptblocks and the call (&) operator …

- We needed the call operator to do indirect invocations, again because we're not expand and parse
  - `& “a command with spaces” so there`
- MMC team requested a way to have isolated (anonymous) blocks of code they could call later
  - This seemed like a job for *lambda functions*
  - We wanted them to be friendly to non-coders so they became *Scriptblocks*.
  - Example with `foreach` and `where` cmdlets

    ```
    dir | where { $_.Length -gt 100 }
    ```
  - Users are not necessarily aware they are using lambdas
- Scriptblocks became the core of script execution scenarios, including things like .NET delegates which allowed WinForms and XAML scripting.
- Example: function definition through assignment:

  ```
  $function:bob = {param ($name) Write-Host “Hi $name, I’m Bob!”}
  ```



PowerShell Conference EU 2018

Language Design Questions

- Expression-oriented language – everything returns a value
  - `$x = foreach ($i in 1..10) { $i * $i }`
- Should we have lexical or dynamic scoping?
  - Modified dynamic scoping best match the shell paradigm so we went with that
- Should the option character be '-' or '/'
  - Unix shell compete – duh – so we went with '-'
- How to handle switches (parameters that don't take arguments and are simply present or absent)
  - We started with just `[bool]` but people wanted to use if for other things
  - So we invented the `SwitchParameter` type.
  - But we still needed a way to pass arguments(!) in some scenarios (commands calling commands) so we implemented

```
Get-Something -mySwitch: $false
```

Language Questions cont'd

- Shells just deal with *strings* so everything is a *string*. PowerShell Lives in a world of objects but it had to *behave* like every other shell.
- Solution was to have *super-aggressive* type conversions
  - This means that the PowerShell runtime makes a lot of implicit guesses at the types you meant (but conversions do not chain implicitly)
- It does this by *searching* for a converter with the best type match
  - Type distance algorithm – in ambiguous cases, it computes a "distance" between the various candidate types and their target. Shortest distance wins.
  - Parameter binder doesn't do this. Binding is done in two passes:
    - Exact type match is required on first pass
    - *First* possible valid conversion is taken on the second pass.

Shipping Mon^h^h^h PowerShell

To Ship is to Choose.

… was like being a cockroach in a room full of cowboy boots.

-Jeffrey Snover

Shipping Is Fun!

- PowerShell started as an “independent” project inside Microsoft
  - Had it’s own build system, etc.
  - Would ship on its own (?)
- PowerShell joined Windows
  - Switched to the Windows build system
  - Would ship with Windows (Longhorn)
- LONGHORN RESET – WORLD EXPLODES
  - We were a .NET component so we were “kicked out” of Windows
- Crap. Now what do we do…

Shipping is Fun 2!

- Ok – drifting aimlessly – and along comes the Exchange team
  - They were investigating an automation solution for managing Exchange servers; especially GUI over command line.
  - PowerShell pretty much matched everything they were looking for. Yay – we have a new partner and a new shipping vehicle.
- Some impact on the product though
  - First, the shipping delay gave us a lot more time to refine what we had.
  - The focus was on Exchange features first. This meant that some things did not get as much attention as we would have liked
    - Native command support was weak in V1 (and still is to some extent).

Shipping Is Fun 3!

- Finally we get approval to ship as a *Windows* feature, not as Exchange feature
  - But first we have to pass some gates…
- Versioning requirement – you must never break anyone *ever ever ever*
  - Loose versioning == DLL hell. Rigid versioning is brittle and breaks; *worse is better*
  - Versioning is not a problem – problems can be solved. Versioning can't be solved, only mitigated. (`ICommandRuntime class`)
- No plugins requirement
  - At the time, plugins in IE were causing a lot of grief.
  - But PowerShell is all about plugins. So we had to do some work to meet this requirement.
    - First attempt – *minishells* – essentially requiring every module vendor to compile their own PowerShell executable: `powershell { invoke-code in new shell }`
    - Round 2 was Snap-Ins – patterned after MMC which was already shipping with Windows; included "console files" to preconfigure your environment

…`/src/System.Management.Automation/singleshell/config`

Shipping is Fun 4!

- Language reviews with the CLR & DLR teams
  - One of the big issues discussed was push vs pull
    - LINQ was being released; used a pull model
  - PowerShell used push (similar to the reactive framework (Rx))
  - Ultimately, this was resolved without much difficulty ☺
- Then someone noticed we could build software with PowerShell. *Rats!*
  - The msbuild development effort was a significant work effort in DevDiv
  - Through mechanisms unknown, somebody saw a slide talking about how PowerShell could be used as part of a software construction toolchain (just like sh/ksh/bash is used on UNIX).
    - Led to an unfortunate misunderstanding
    - Initially quite tense but it got cleared up reasonably quickly.

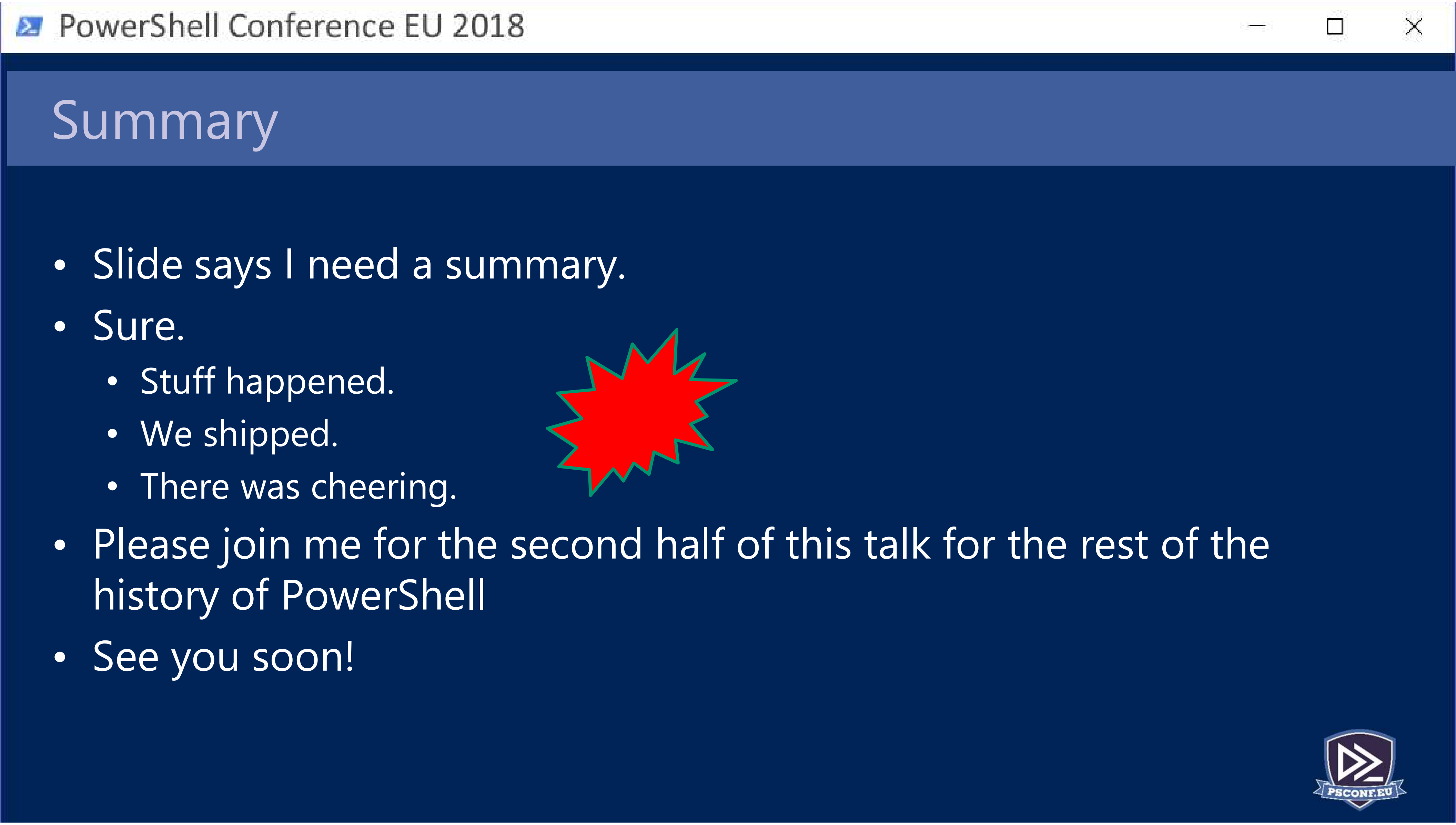
Shipping is Fun 5

- All clear to ship except for one little detail.
- Had to change the product name from “Monad” to “Windows PowerShell”.
  - Did a survery to justify name change
    - Which name sounds more powerful: Monad or PowerShell
    - Which name sounds more like a shell: Monad or PowerShell
  - Lots of code changes, setup changes, registry changes
- Most everything got cleared up but there are still traces in the code
  - e.g. PSObject is defined in the file `…/System.Management.Automation/engine/MshObject.cs`

PowerShell Conference EU 2018

And We Shipped!

- PowerShell Version 1 shipped November 2006!



PowerShell Conference EU 2018

Summary

- Slide says I need a summary.
- Sure.
  - Stuff happened.
  - We shipped.
  - There was cheering.
- Please join me for the second half of this talk for the rest of the history of PowerShell
- See you soon!

Next Steps

- Now: 15 min break

- Grab a coffee
- Stay here to enjoy next presentation
- Change track and switch to another room

- Ask me questions or meet me in a breakout session room afterwards

Questions?

```
# use this template for code samples
# and follow these instructions to show perfectly
# color-coded PowerShell code:

# paste code to PowerShell ISE, select it, and copy it
# to the clipboard
Paste-Code -ToISE | Select-Code | Copy-ToClipboard

# to insert it with full color-coding into a slide,
# paste the code to your PPT slide. It will be black,
# and there is a toolbutton labelled (Ctrl) in PPT.
Click-ToolButton -Choose SymbolWithClipboardAndBrush
# Click the toolbutton, and choose the button that shows
# a clipboard with a brush. This will add color-coding
# back to the pasted code
```

PLEASE NOTE: You do not need to use PowerShell ISE to code, or to demo. Use whatever editor you like best.

These steps use PowerShell ISE to color-code your code correctly and consistently, and insert the color-coded code into the slide. Please help make all code look consistent. Many thanks!

about_Author

- Feel free to add information about yourself
- You may advertise for you or your company
- Keep slide at the end of your presentation
- Do not use presentation time for your bio
  - Instead, your bio is published in conf materials