PowerShell for Beginners

Basics and Complex Exercises

Table of Contents

Creating PowerShell Scripts

Creating ps1 files, execution policy

Creating PowerShell Scripts

- PowerShell scripte are plain text files (like batch/bash scripts)
- Default file extension is .ps1
- PowerShell Integrated Scripting Environment (ISE)
  - Editor with syntax highlighting (upper part)
  - PowerShell session (lower part)
  - Only available for Windows
  - Run as administrator to be able to configure execution policy of scripts
- Visual Studio Code is an alternative
  - Also offers a quite powerful debugger for PowerShell

☞ Demo

Executing PowerShell Scripts

- Execution in PowerShell by typing the file name preceded with „.\“ .
  - If working directory is not where the scripts resides, use relative or absolute path to the script file.
  - Example: `PS> .\skript.ps1`
- Execution in Windows Explorer by clicking *Execute with PowerShell* in context menu
- Caution:
  - Execution of batch files is possible per default
  - Execution of PS scripts has to be explicitly allow through an execution policy
- Query or set execution policy with
  - `Get-ExecutionPolicy` and `Set-ExecutionPolicy`

Execution Policy

- PowerShell implements 7 different execution policies
- The scope of the policy can be defined by `-Scope .`

- Restricted:
  - no execution of any scripts
  - default for Windows clients (workstations)

- RemoteSigned:
  - scripts downloaded from the internet require a trusted signature
  - default für Windows Server

Execution Policy

- Unrestricted:
  - all scripts can be executed
  - default for non-Windows computers
- Additionally available: *AllSigned*, *Bypass*, *Default*, *Undefined*
- Setting an execution policy requires administrator privileges

☞ Decide for yourself about your desired execution policy.

Exercise PS81
Scripting with PowerShell

- Creating script files
- Link files to a suitable editor
- Create simple scripts
- Configure execution policy

Subsessions in cmd.exe

Access to legacy cmd environment inside PowerShell

Subsessions in cmd

- Use the command `cmd` to start a *cmd* inside a PS session
- In the subsession, only cmd commands are available
- Direct execution of batch scripts is possible
- Leaving the subsession with `exit`
- Useful for processing legacy batch programs etc.

Exercise PS82
Subsession in PowerShell

- Open a *cmd* subsession
- Use DOS commands to solve simple text processing tasks
- Do some batch programming inside the subsession

Variables and Control Structures

Variable notation, data types, branching, loops

Control Flow: Sequence, Structure

- Script file is processed line wise from top to bottom
- Sequencie of commands in one line denoted by semicolon ;

- Blocks of statements enclosed in curly brackets { }
- Syntax like in C# (or Java)

- Functions can be defined with keyword *Function*

Variables

- Variables are defined with a $ preceding the name, e.g. $a=0, $str=“anr“
- Explicit typing is *possible*, but not required
- Explicit type annotation with `[type]::$var` or `[type]$var`

```
PS C:\Users\anr> [string]$str="anr"
PS C:\Users\anr> [string]$a=10
```

- `$str` explicitly is a string
- `$a` is converted (casted) to a string as well

```
PS C:\Users\anr> $a*10
10101010101010101010
```

- `$a` now also behaves like a string ☺

Cmdlets for Variable Management

- Cmdlets for managing variables are found in the *Variable* family.

```
PS C:\Users\anr\Downloads\App2> gcm -Noun Variable

CommandType     Name
-----------     ----
Cmdlet          Clear-Variable
Cmdlet          Get-Variable
Cmdlet          New-Variable
Cmdlet          Remove-Variable
Cmdlet          Set-Variable
```

- Define constants with *-Option Constant*
  - Changes of a constant reuqire a restart of PowerShell (i.e. a new session)

- Upon removal of variables omit the dollar sign ‚$‘

Lists, Arrays, Dictionaries

- Lists are defined in round brackets
- Access through square brackets and index (0-based)
- Concatenation von Listen mit + (operator is overloaded)

- Arrays and dictionaries can be defined with @ symbol.
- Create an empty dictionary with `$dict = @{}` for instance.
- Dictionary can be used (in almost every sense) like for example in Python.

Dictionary Example

```
PS C:\Users\anr\Downloads\App2> $arr=@{}
PS C:\Users\anr\Downloads\App2> $arr['anr']="Angres"
PS C:\Users\anr\Downloads\App2> $arr['anr']
Angres
PS C:\Users\anr\Downloads\App2> $arr['anr'] += ",Julius"
PS C:\Users\anr\Downloads\App2> $arr['anr']
Angres,Julius
```

Data Types: Numbers

- Built-in number data types
  - Int, UInt, long, byte, short, …
- Arithmetic standard operations (including *modulo* %) directly available in infix notation
- Special number functions in library [math]
- E.g. power 2^{10}

```
PS C:\Users\anr> [math]::Pow(2,10)
1024
```

- Floating point numbers available in single (Float) and double (Double) precision.

Data Types: Strings

- Denoted in double quotes:
  - Variable expression are dynamically replaced

```
PS C:\Users\anr> $str="anr"
PS C:\Users\anr> Write-Host "Hallo $str"
Hallo anr
```

- Denoted in single quotes:
  - No replacement (interpretation as *literal string*)

```
PS C:\Users\anr> $str="anr"
PS C:\Users\anr> Write-Host 'Hallo $str'
Hallo $str
```

Data Types: Strings

- Usually denoted in double quotes
- However, quotes can often be omitted → value is interpreted as string
- Some Cmdlets offer a LiteralPath parameter

- To evaluate an expression after replacement inside a string, enclose the expression in $(<EXPR>).

Data Types: Strings

- Example:
- Evaluation inside a string

```
PS C:\Users\anr> $a = 4
PS C:\Users\anr> $b = 2
PS C:\Users\anr> "$a/$b"
4/2
PS C:\Users\anr> "$($a/$b)"
2
```

- Only by using the surrounding $() we get the value of the expression inside the string.

Data Types: Boolean Values

- Truth values are predefined in `System.Boolean`
- In some cases they can even be used for calculations (like in C/C++)
- `$True` is the logical truth
- `$False` is the logical contradiction

Control Flow: Branching

- PowerShell implements classic If- and Switch-Statements
- `If` does not necessarily need an `Else`

```
PS C:\Users\anr> If ($n % 2 -eq 0) { Write-Host "even" } Else { Write-Host "odd" }
```

- Mind the „dangling Else“ problem

Control Flow: Iterations

- PowerShell implements lots of common loops:
  - loop with fixed number of iterations and index variable (For-Schleife)
  - loop with fixed number of iterations and object reference (ForEach-Schleife)
  - loop with variable number of iterations (While-Schleife, Do-While-Schleife)
- All loops are semantically equivalent
- Running Example:
  - Print the natural number from 1 thru 5

For-Loop

- PowerShell offers a C-style For-loop

```
PS C:\Users\anr> For ($i = 1; $i -le 5; $i++) { Write-Host "$i" }
1
2
3
4
5
```

- Variable reference is `$i`
- Mind the comparison operator (`-le` instead of `<=` like in C/C#/Java etc.)
- The quotes for the output are optional.

ForEach-Loop

- PowerShell offers two possibilities for iteration over lists:
- A ForEach-loop (like in most common languages)

```
PS C:\Users\anr> ForEach ($i in 1..5) { Write-Host $i }
1
2
3
4
5
```

- Variable reference is `$i`
- No comparison operator needed

ForEach-Loop

- PowerShell offers two possibilities for iteration over lists:
- The Cmdlet `ForEach-Object` (since PowerShell is object oriented)

```
PS C:\Users\anr> 1..5 | ForEach-Object { Write-Host $_ }
1
2
3
4
5
```

- Variable reference is `$_` (anonymous, similar to *lambda* in F# or `this` in Java)
- ForEach-Object is fed by the pipeline
- More general pattern: `$LO..$HI | ForEach-Object { … }`

While-Loop

- Classic While-loop with keyword *While*
- There are also *Do-While* and *Do-Until*…

```
PS C:\Users\anr> $i=1
PS C:\Users\anr> While ($i -le 5) { Write-Host $i; $i++ }
1
2
3
4
5
```

- Variable reference is $i
- Variable must be initialised outside the loop

Exercise PS83
Programmierung mit PowerShell

- Create and manage variables of various data types
- Branching with If-statements
- Applying different types of loops
- Recognize assets and drawbacks for each type of loop

Ranges and Pipeline

Range notation, PowerShell pipeline and progress bar

Ranges

- Ranges can be defined by two dots between start and end value.
- Prerequisite: Index set with ordering relation
  - an ordering relation is reflexive, transitive and antisymmetric
- Examples:
  - (Integer, $\leq$) natural ordering
  - (Char, $\leq_{\mathrm{Lex}}$) lexicographic ordering
  - (Boolean, $\leq_{\mathrm{B}}$) contradiction is smaller than truth

Ranges

- Examples:
  - 1..10 → List of the natural number 1 thru 10 (both included)
  - ‘a‘..‘z‘ → List of lower case lettes from the English alphabet
  - $False..$True → List of numbers 0,1 mapped to the Boolean values
- Use of the pipe operator ‚|‘ possible just like at the prompt
- Ranges are often used to kick off a pipeline

Exercise PS84
Programmierung mit PowerShell

- Working with ranges
- Using the pipeline in scripts

Progress Bar

- Possible enhancement of the user interface

- Implemented in Cmdlet `Write-Progress`
  - Informs user about current status of an activity
  - Useful for long running, CPU intense commands
  - Wichtige Parameter: *-Activity, -Status, -PercentComplete*
  - *Activity* can descibe what happens
  - *Status* provides feedback to the user
  - *PercentComplete* can show percentages related to the activity

Progress Bar
Example

- Simulated search as 20 times iteration through a loop with a progress bar showing the current percentage.
- We set a variable to 20 (such that the value is not *hard coded*)

```
PS C:\Users\anr> $n = 20
PS C:\Users\anr> For ($i = 1; $i -le 20; $i++) { Write-Progress -Activity "Suche..
" -Status "$($i/$n*100)% abgeschlossen"; Start-Sleep -Milliseconds 100 }
```

- ☞ Demo

Progress Bar
Hints

- An artificial sleeping interval of at least 100 ms is necessary for the progress bar to become visible
- Percent completed can be calculated as status with the expression

  `"$($i/$n*100)%"` Here *$i* is the index variable of the loop and *$n* the total number of iterations.

- Caution:
  - Mind the (double) quotes (parameter *Status* is a string)
  - Use dollar sign and braces $() surrounding the expression, because it is inside a string

Funktions in PowerShell Scripts

User Defined Functions

Defining Functions

- Defining custom PowerShell funktions in a script:
  - Keyword *Function*
  - Enclose body of the function in curly brackets (like C/C++, C#, Java, …)
  - Parameter defined in the body of the function with *param*(…)
  - Call functions by name and use parameters like with Cmdlets

Defining Functions
Naming and Calling

- In principle function names can be chosen (almost) arbitrarily
  - usual restrictions concerning keywords, etc.

- As a convention Verb-Noun-syntax is to be used
  - at least for exported functions visible outside the source code file
  - not necessarily for small helper functions
  - this is no formal constraint, but *strongly recommended*
  - Microsoft provides us even with a list of approved verbs
    - Approved Verbs for PowerShell Commands - PowerShell | Microsoft Learn

Defining Functions
Naming and Calling

- Functions can be called like Cmdlets with their name
  - Parameters space separated behind the name
  - Parameters can be used as named parameters just like with Cmdlets

- To have a function in the scope of a PS session, the script containing the function must be dot-sourced:

```
PS C:\Users\anr\Downloads> . .\MyScript.ps1
```

- Now all functions defined in *MyScript.ps1* are available in the session
  - This also works with built-in help and tab completion!

Defining Functions
Using Parameters

- Function with a single parameter

```
Function MySingleFun { param($i) Write-Output $i }
```

- Call:

```
PS C:\Users\anr> MySingleFun 10
```

  - like a Cmdlet or in functional languages (F#/Haskell)

- …this also works:

```
PS C:\Users\anr> MySingleFun(10)
```

  - like in C#/Java

Defining Functions
Using Parameters

- Function with two parameters

```
PS C:\Users\anr> Function MyTwinFun { param($i,$j) Write-Output "$i $j" }
```

- Call:

```
PS C:\Users\anr> MyTwinFun 10 20
```

- …this also works:

```
PS C:\Users\anr> MyTwinFun(10,20)
```

- Also working correctly, if one parameter is a list

Defining Functions
Custom Named and Switch Parameters

- All defined parameters can automatically be used like named parameters or switch parameters.

- Example:
- Definition of a function, that lists all (local) users and groups.
- Parameters:
  - *$User* the user to be listed (provided as string)
  - *$NoGroups* a switch Parameter that prohibits listing groups, if active

Defining Functions
Custom Named and Switch Parameters

- Demo:
  - Short source code analysis of *Groupmembership.ps1*
  - Several examplary calls using the parameters

Defining Functions
Example

- A function that prints the first 3 processes (in alphabetical order).

```
PS C:\Users\anr> Function Get-FirstThreeProcess { Get-Process | Select-Obje
ct -First 3 }
PS C:\Users\anr> Get-FirstThreeProcess
```

Defining Functions
Example

- Extension resp. generalisation:
- A function that prints the first n processes (in alphabetical order).

```
PS C:\Users\anr> Function Get-FirstNProcess { Get-Process | Select-Object -
First $n }
```

- Problem:
  - Variable *$n* must be declared and initialised before the function call. werden.

```
PS C:\Users\anr> $n=3
PS C:\Users\anr> Get-FirstNProcess
```

Defining Functions
Parameters

- Extension resp. generalisation:
- The function gets a parameter *$Number*
- The keyword *param* can be written in either lower or upper case

```
PS C:\Users\anr> Function Get-FirstNProcess { param($Number) Get-Process |
Select-Object -First $Number }
PS C:\Users\anr> Get-FirstNProcess 3
```

- Problem:
  - Variable *$n* must be used with the correct type.
  - Function calls might otherwise lead to unpredictable behavior

☞ What is the result of *GetFirstNProcess* mit 3.01, 3.5, “3“, 14.5, “drei“ or $True ?

Defining Functions
Parameters

- Extension resp. generalisation:
- The function gets a parameter *$Number* of type *int*

```
PS C:\Users\anr> Function Get-FirstNProcess { param([int]$Number) Get-Proce
ss | Select-Object -First $Number }
PS C:\Users\anr> Get-FirstNProcess 3
```

- Problem:
  - What happens, if the parameter is missing in the function call ?
  - Decision: error action, exception, change parameter handling ?

Defining Functions
Parameters

- Extension resp. generalisation:
- The function get s parameter *$Number* of type *int*, that must be provided upon function call

```
Function Get-FirstNProcess {
  param(
    [Parameter(Mandatory=$True)]
    [int]$Number
  )
  Get-Process | Select-Object -First $Number
}
```

- Problem:
  - A forced use of the parameter is not always desirable.

Defining Functions
Mandatory Parameters

- Proper function call

```
PS C:\Users\anr\Downloads> Get-FirstNProcess 3

 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
     16     3,63      12,48       0,00    2924   0 AppHelperCap
     27    20,20      14,13       0,36    9464   2 ApplicationFrameHost
     39    26,50      13,60       0,00   12056   0 AWACMClient
```

Defining Functions
Mandatory Parameters

- Call without parameter („forgotten parameter“)

```
PS C:\Users\anr\Downloads> Get-FirstNProcess

cmdlet Get-FirstNProcess at command pipeline position 1
Supply values for the following parameters:
Number:
```

- Not aborted, but user interaction
- Name of the parameter in question is displayed
- Value can be provided on the fly
- Function evaluation proceeds normally

Defining Functions
Parameter with Default Value

- Extension resp. generalisation:
- The function gets a parameter *$Number* of type *int*, that has a default value of 3

```
Function Get-FirstNProcess {
  param([int]$Number=3)
  Get-Process | Select-Object -First $Number
}
```

Defining Functions
Parameter with Default Value

- Call without parameter („forgotten parameter“)

```
PS C:\Users\anr\Downloads> Get-FirstNProcess

 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
     16     3,63      12,43       0,00    2924   0 AppHelperCap
     27    20,20      14,13       0,34    9464   2 ApplicationFrameHost
     39    26,50      13,58       0,00   12056   0 AWACMClient
```

- Parameter *$Number* is automatically defaulted to value 3
- No further action warranted ☺

Defining Functions
Documentation

- (Important, large) functions should be documented
- Through comments in the source code
- And through a special markup language (like e.g. Javadoc, Haddock)
- Important tags are
- .SYNOPSIS, .DESCRIPTION, .PARAMETER
- This documentation is displayed if *Get-Help* is called with the custom function as argument!

Defining Functions
Documentation

- Example: custom function *Set-NtfsPermissions*

```
<#
  .SYNOPSIS
  configure NTFS permissions for a folder
  .DESCRIPTION
  sets desired NTFS permissions for an existing object or an object that is to be created
  .PARAMETER Folder
  the folder the permissions of which are to be created resp. updated
  .PARAMETER BreakInheritence
  breaking up existing inherited permissions is disabled by default
  .PARAMETER FA
  list of comma separated principals to grant them FullAccess
  .PARAMETER MA
  list of comma separated principals to grant them ModifyAcess
  .PARAMETER RA
  list of comma separated principals to grant them ReadAccess
#>
```

Defining Functions
Documentation

- Example: custom function *Set-NtfsPermissions*

```
PS C:\Users\anr> Get-Help Set-NtfsPermissions

NAME
    Set-NtfsPermissions

SYNOPSIS
    configure NTFS permissions for a folder


SYNTAX
    Set-NtfsPermissions [[-Folder] <String>] [-BreakInheritence] [[-FA]
    <Array>] [[-MA] <Array>] [[-RA] <Array>] [<CommonParameters>]


DESCRIPTION
    sets desired NTFS permissions for an existing object or an object
    that is to be created
```

Defining Functions
Hints

- Defining functions is generally easy ☺
- Parameter in the *param* Tag within the function body
- Call function by name, parameter without brackets (recommended)
  - not like in C#/Java
  - just like in F#/Haskell
- Values in brackets are interpreted as lists
  - usually makes no difference

Defining Functions
Hints

- Parameter can be refined by…
  - type annotation (before the name in square brackets)

```
[int]$Number
```

  - making them mandatory through a mandatory-clause

```
[Parameter(Mandatory=$True)]
```

  - providing a default value for them

```
[int]$Number=3
```

Defining Functions
Outlook

- There are even more details concerning functiones:
- Possibility to execute code explicitly at the beginning (when entering the function) or at the end (before leaving the function) of function evaluation.
- …and other advanced concepts

Exercise PS85
Programming with PowerShell

- Work with a progress bar
- Implement classical mathematical functions
- Use parameters and other structures within function bodies

Working with COM- and .NET-Objects

Creating objects, automating MS Office, building system tray icons

The Component Object Model (COM)

- Developed by Microsoft
- Introduced in 1992 with Windows 3.1
- Allows for interprocess communication
- Allows e.g. accessing MS Office applications

The Component Object Model (COM)

- COM-objects are (still) commonly used under Windows
- They offer various functionalities through its interface
- Can be directly created and used in PowerShell
  - Cmdlet `New-Object` with parameter `-ComObject`
- Goal: Automating MS Office (Word, Excel, Access)

PowerShell and COM-Objects
Example Excel Document

- Task:
  - Populating an Excel document with PowerShell output
- Approach:
  - Create a COM object
  - Add Excel-specific object structure (according to COM)
    - Excel file → workbook → worksheet
  - Write data
  - Save and close Excel file

PowerShell and COM-Objects
Example Excel Document

- Realization:

```
PS C:\Users\anr> $dokument=New-Object -ComObject Excel.Application
PS C:\Users\anr> $mappe=$dokument.Workbooks.Add()
PS C:\Users\anr> $tabelle=$mappe.WorkSheets.Item(1)
PS C:\Users\anr> $tabelle.Cells.Item(1,1)="Windows PowerShell"
PS C:\Users\anr> $tabelle.Cells.Item(1,2)="Fortbildung"
PS C:\Users\anr> $tabelle.Cells.Item(2,1)="=2+2"
```

PowerShell and .NET-Objects

- In PowerShell .NET-objects can be created anytime anywhere
  - remember Cmdlets themselves are .NET-classes
- Needed moduls must be explicitly loaded in advance, e.g. the Windows Presentation Forms for creation of GUI elements
- Graphical objects may be *MessageBox*, etc. (actually every object, that belongs to Windows Presentation Forms)

PowerShell and .NET-Objects
Example Message Box

- Task:
  - Creation of a message box displaying a greeting.
- Approach:
  - Loading Windows Presentation Forms
  - Static access to *MessageBox* object
  - Display greeting by calling the appropriate method

PowerShell and .NET-Objects
Example Message Box

- Realization:

```
PS C:\Windows\System32> [System.Reflection.Assembly]::LoadWithPartialName("PresentationFramework")


GAC    Version        Location
---    -------        --------
False  v4.0.30319     C:\Program Files\PowerShell\7\PresentationFramework.dll

PS C:\Windows\System32> [System.Windows.MessageBox]::Show("Hello World!")
OK
```

- Result:

PowerShell and .NET-Objects
Example RSS Feed

- Task:
  - Build an RSS Feed reader
- Approach:
  - Create object of type *Net.Webclient*
  - Download website using the feed‘s URL
  - Interpret the result (server answer) as XML document
  - Accessing the desired properties

PowerShell and .NET-Objects
Example RSS Feed

- Realization:

```
PS C:\Users\anr> $webclient=New-Object Net.WebClient
PS C:\Users\anr> $xmlpage=[xml]$webclient.DownloadString("https://devblogs.microsoft.com/po
wershell/feed/")
PS C:\Users\anr> $xmlpage.rss.channel.Item | Format-List -Property title, link
```

- Result:

```
title : PowerShell/OpenSSH Team Investments for 2023
link  : https://devblogs.microsoft.com/powershell/powershell-openssh-team-invest

title : PowerShell Extension for Visual Studio Code January 2023 Update
link  : https://devblogs.microsoft.com/powershell/powershell-extension-for-visua

title : PowerShellGet 3.0 Preview 18
link  : https://devblogs.microsoft.com/powershell/powershellget-3-0-preview-18/
```

Exercise PS86
Programming with PowerShell

- Create COM-objects
- Controlling an Excel file with PowerShell
- Create a second RSS feed reader
- Use message boxes and system tray icons

Complex Programs, HOF

Programming larger applications

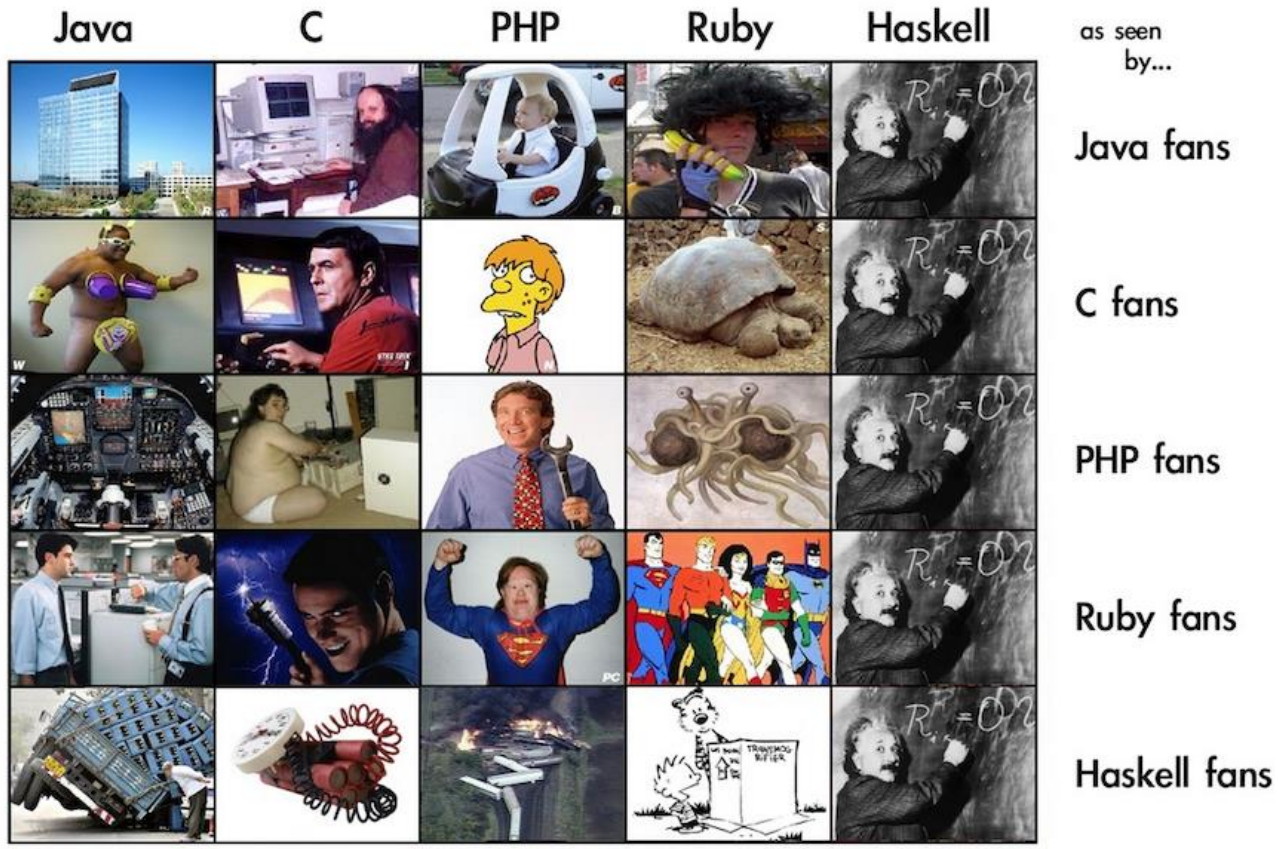
Complex Programs

- In general arbitrary programs can be created with PowerShell
- Examples (see exercise):
  - Caesar encryption
  - 15-puzzle

Higher Order Functions

- Higher Order Functions, HOF for short, are functions that have a function as parameter or return a function.
- HOF are a typical element of functional programming.
- Nowadays functional aspects influence modern programming languages (Java, C#, Python)
- HOF realize a much higher level of abstraction

Higher Order Functions
Code becomes shorter, clearer and cooler



Higher Order Functions

- Three important, fundamental HOF:
  - Applying a function to each element in a list (or list-like data type)
  - Filter elements in a list according to a user defined predicate
  - Folding a list with a binary function such that a single value is the result

Higher Order Functions
Scriptblocks

- In PowerShell expressions and functions can be stored in variables.
- The associated type is *scriptblock*.
- The function stored in a variable of type *scriptblock* can be used as input for a HOF.

- Syntax:
- `[scriptblock]$var = { <Expression> }`

Higher Order Functions
Scriptblocks

- Example:
- A function that adds the constant 5 to the input value

- Common function definition:

```
PS C:\Users\anr> Function addiere5 { param($n) $n + 5 }
```

- Definition as *scriptblock*:

```
PS C:\Users\anr> [scriptblock]$addiere5 = { $_ + 5 }
```

Higher Order Functions
Mapping

- Mapping means applying a function to all elements in a list.
- Result is (usually) a list of the return type of the function.
- Equivalent to the Haskell function *map* with the signature

  map :: (a → b) → [a] → [b]

- In PowerShell, mapping is implemented by the Cmdlet `ForEach-Object`
  - A brief alias for `ForEach-Object` is ‚%‘ (percent sign)

Higher Order Functions
Mapping Example

- A function that adds the constant 5 to the input value
- Common PowerShell solution with *Function addiere5*:

```
PS C:\Users\anr> 1..10 | ForEach-Object { addiere5 $_ }
```

- Solution with script block *$addiere5*:

```
PS C:\Users\anr> 1..10 | % $addiere5
```

- Advantage:
  - Script block needs less code
  - Logic is encapsulated (increases reusability)

Higher Order Functions
Mapping Hints

- Generic approach:
- Define function as script block *$block*
- Define input *$input*

- Execute in a pipeline with `ForEach-Object`:
- `PS> $input | % $block`

- Chaining function can be realized as a mapping-pipeline

Higher Order Functions
Filter

- Filtering means applying a predicate on all elements in a list.
- A predicate is a function that returns a truth value (Boolean value).
- Result is (usually) a list of the same type as the input.
- Equivalent to Haskell function *filter* with the signature

  filter :: (a → Bool) → [a] → [a]

- In PowerShell, a filter is implemented by the Cmdlet `Where-Object`
  - A brief alias for `Where-Object` is ‚?‘ (question mark)

Higher Order Functions
Filter Example

- Example:
- A function that checks, whether an input value is an even number.

- Common function definition:

```
PS C:\Users\anr> Function gerade { param($n) If ($n % 2 -eq 0) { $True } Else { $Fa
lse } }
```

- Definition as script block:

```
PS C:\Users\anr> [scriptblock]$gerade = { $_ % 2 -eq 0 }
```

Higher Order Functions
Filter Example

- A function that checks, whether an input value is an even number.
- Common PowerShell solution with *Function gerade*:

```
PS C:\Users\anr> 1..10 | Where-Object { gerade $_ }
```

- Solution with script block *$gerade*:

```
PS C:\Users\anr> 1..10 | ? $gerade
```

- ☞ What output is produced by `PS C:\Users\anr> 1..10 | % $gerade`?

Higher Order Functions
Filter Hints

- Generic approach:
- Define predicate as script block *$pred*
- Define input *$input*

- Execution in a pipeline with `Where-Object`:
- `PS> $input | ? $pred`

- There exists also a keyword *filter* → that‘s a different story…

Higher Order Functions
Fold

- Fold means that a list is folded to a single value using a binary function.
- Result is (usually) a value of some type (not necessarily the type of the input list, but rather the return type of the binary function).
- Equivalent to Haskell function *foldr* with the signature

  foldr :: (a → b → b) → b → [a] → [b]

- In PowerShell, the is currently no Cmdlet that implements a fold.
  - The module *functional* from the PowerShell Gallery offers all three classic HOF
  - The PowerShell Gallery contains third-party modules unvalidated by Microsoft

Exercise PS87
Programming with PowerShell

- Create larger, more structured programs
- Use the speech synthesizer to create audio output
- Implement Caesar encryption or 15-puzzle