

# LINUX

## WINDOWS HISTORIKK

Microsoft ble grunnlagt av to PC entusiaster Bill Gates og Paul Allen i 1975. I 1980 kom de ut med et ikke-grafisk OS kalt MS-Dos (Microsoft Disk Operating System). De skjønner fort at et OS må være mer brukervennlig, Grafisk Windows 1.0 gis ut i 1985. Helt ny Windows kjerne introduseres med Windows NT i 1993. PowerShell ble lansert i 2006, kommando-linje og script-språk. Windows 10 er siste versjon.

## WINDOWS BRUKERE, GRUPPER OG LOCAL SECURITY POLICY

Tanken denne uka er at vi skal jobbe med Windows og Hårek anbefaler at vi laster ned en VirtualBox, også installerer en Virtual Windows 10.

- Brukere kan defineres i Control Panel → System and Security → Administrative Tools → Computer Management → Local Users and Groups → Users.
- Alternativt: trykk Windows-tasten og skriv Computer Management.
- Eller Windows-tasten og x som gir en meny hvor du kan velge Computer Management.
- Det er som i Linux grovt sett to typer kontoer, Administrator og Standard.
- Administrator-typen (root på Linux) har alle rettigheter mens standard har begrensede.
- Som i Linux, vanlig brukere kan bruke systemet men har begrenset mulighet til å endre systemet.

På samme måte som grupper i Linux har vi grupper på Windows hvor administrator kan lage nye grupper. For at en ny bruker skal ha administrator-rettigheter må han være medlem av gruppen administrators.

- Grupper kan også defineres i Control Panel → System and Security → Administrative Tools → Computer Management → Local Users and Groups → Groups.
- På samme måte som i Linux finnes det noen predefinerte grupper som Administrators og users.

I Local Security Policy kan man definere en rekke sikkerhetsregler som gjelder for brukere og en Windows-maskin.

Under Control Panel → Administrative Tools → Local Security Policy kan man i stor detalj definere hvilke sikkerhetsregler som skal gjelde for brukerne på maskinen, for eksempel

- sette krav til lengden på passord
- bestemme hvor lang tid det skal gå før et passord må skiftes
- kreve at man ikke bruker tidligere passord om igjen
- velge at en konto stenges automatisk etter et gitt antall mislykkede forsøk på å logge seg inn

Du kan åpne Local Security Policy-verktøyet ved å følge disse trinnene:

1. Trykk på Windows-tasten + R for å åpne "Kjør"-dialogboksen.
2. Skriv "secpol.msc" i tekstfeltet og trykk "Enter".
3. Dette vil åpne Local Security Policy-vinduet.

## WINDOWS FILSYSTEMER OG RETTIGHETER

Filrettigheter er definert i filsystemet. Så hvis man ikke har filrettigheter satt opp i filsystemet så kan ikke OS gjøre så mye med det. For eksempel det gamle Windows filsystemet FAT (File Allocation Table), i det systemet er det veldig begrenset hvilke rettigheter man kan ta. Hvis du har et FAT filsystem så er det vanskelig å definere hvilke brukere som har tilgang til hvilke filer. Men siden 1996 har NTFS vært default filsystem på Windows siden Windows XP og Windows NT i 1996.

Når man først har NTFS som støtter blant annet Access Control List (ACL) så har man muligheten til å styre i stor detalj hvilke rettigheter brukere og grupper skal ha til enhver fil og mappe. Dette kan gjøres ved å høyreklikke på en fil eller en katalog, velge properties og deretter velge security fanen. Dette valget har man ikke med et FAT-filsystem. Ext3 eller ext4 filsystemer som brukes av Linux, støtter ACL, men må slås på, og default brukes ikke Linux dette. Default har ext et merbegrenset rettighetsystem: hver fil eller mappe kan gis rettigheter for eier, en gruppe og alle andre brukere på systemet.

## WINDOWS KOMMANDOLINJE, CMD.EXE, BATCH OG VBSCRIPT

Da skal vi begynne å se på Windows kommandolinjen. Tidligere var cmd.exe den eneste kommandolinjen som man hadde i Windows. Helt opprinnelig så var det DOS, og da måtte vi gå inn i DOS og endre på ting. Cmd.exe har mange av de samme kommandoene som DOS hadde, men det er egentlig et helt nytt shell. Og det er fortsatt default shell i Windows, hvis du

starter opp et shell så er det cmd.exe du får. Det er mulig å omkonfigurere det i Windows 7, du kan sette opp PowerShell som default shell. Cmd.exe har kommandoer slik som dir, copy, cd, del, undelete osv som likner veldig på Linux kommandoer. Dir er det samme som ls. I PowerShell er det aliaser som gjør at både DOS kommandoer eller cmd.exe kommandoer som dette her og Linux kommandoer kan kjøres.

I cmd.exe kan man lage såkalte batch script (script.bat) som utføres linje for linje akkurat som bash-script. Ganske begrenset hva man kan gjøre i batch script akkurat som i bash script på Linux. I 1996 kom VBScript (virtual basic script) som er et slags type versjon av batch men er mye kraftigere og mer fleksibelt.

Et lite batch script som rebooter PC'en etter 5 sekunder:

```
Shutdown -r -t 05
```

## WINDOWS SCRIPT HOST

WSH (Windows Script Host) er et rammeverk som støtter flere scriptspråk, inkludert VBScript, Python og masse annet. På Windows er VBScript nå den mest populære, så det finnes en rekke script der ute som er skrevet i VBScript. Og det har en default script-tolker (som bash på Linux-9, Wscript.exe og .vbs-filer assosieres med denne. Så hvis jeg for eksempel klikker på en test.vbs-fil så er det Wscript som tolker og kjører denne. I tillegg så har man Cscript som er en script-tolker som kjører script fra kommandolinjen. Men i utgangspunktet så er VBScript visuelt så det viser output, og knapper og dialoger i popup-vinduer.

- C:\Windows\system32\cscript.exe tilsvarer /bin/bash i Linux.

cscript.exe tilsvarer /bin/bash i Linux.

Eksempel på et kort VBScript Hello.vbs:

```
WScript.Echo "Hello World!"
```

## WINDOWS POWERSHELL, INTRODUKSJON-SLIDES

Windows PowerShell kom ut i 2006 og er både kommandolinjeverktøy og scriptsspråk for Windows. Som vi skal se er det kraftig inspirert av Linux bash-shell med mål om å få et minst like kraftig kommandolinje-verktøy. Det har vært installert som default fra og med Windows 2008 Server og Windows 7. På samme måte som Linux er PowerShell bygd opp av mange små programmet eller Cmdlets som gir en fleksibel måte å løse oppgaver på. Siste versjon er 7.0, men script har fortsatt filendelse .ps1, og det også selv om PowerShell versjon 2 er ute.

Et veldig viktig prinsipp i PowerShell er at objekter, og i utgangspunktet så er Windows objektorientert, det er skrevet i C++. Alt i Windows er i utgangspunktet objekter.

Konfigurasjonen er ikke basert på tekstfiler som i Linux, men på binære filer og databaser slik som Registry som er en hierarkisk database brukt av Windows. Og dermed kan man ikke, som vi er vant med i Linux, trekke ut alt informasjon fra tekstfiler, man må også kommunisere med disse binære databasene. Og da er det en veldig effektiv måte å gjøre dette på, og det er ved å bruke den samme objektorienteringen. Og derfor så er PowerShell også objektorientert. Vi har sett med Linux at der sendes strømminger av tekst med pipes og omdirigering. Og det er et veldig kraftig verktøy.

PowerShell derimot sender hele objekter mellom sine cmdlets med pipes. Dette er noe som gjør at PowerShell blir enda kraftigere enn Linux-shell. For når man har objekter kan man på en helt annen måte spørre objektet hvilke egenskaper den har også bruke de senere. Da slipper man veldig mange av de feilene som i Linux med syntaksfeil og finne ut hvilke informasjon man skal gjøre, parse tekst og plukke ut tekst osv.

## FIRE TYPER KOMMANDOER

1. cmdlets: er den viktigste. Det tilsvarer et helt bash-shell built-ins. Kommandoer som pwd, som er en del av shellet. I PowerShell heter det Get-Location og echo er Write-Output. De fleste kommandoer er cmdlets.
2. applications: og det er eksisterende Windowsprogrammer som ping og ipconfig, og det er programmer som ligger på disk, på Windows-disken akkurat som /bin/mv i Linux. I Linux ligger de fleste kommandoene i bin og og /usr/bin.
3. scripts: tekstfiler med endelse .ps1, det er powerscripts og det tilsvarer bash-script.
4. functions: så har vi funksjoner som også likner det i Linux.

Windows PowerShell har verdens korteste «Hello World!» program. Man trenger ikke skrive echo så man kan bare ha en fil med følgende innhold:

```
"Hello World!" ↵
```

Lagringen og kjøringen er slik:

```
Lagres som en fil med navn hello.ps1 og er verdens korteste Hello  
World program; trenger ikke echo. Kjøres med:
```

```
PS> .\hello.ps1  
Hello World!
```

For å få kjøre et script i det hele tatt må du sette Set-ExecutionPolicy, for i utgangspunktet er det satt som i utgangspunktet om at det ikke er lov å kjøre script i det hele tatt fra PowerShell.

Da må man gjøre det på bildet under:

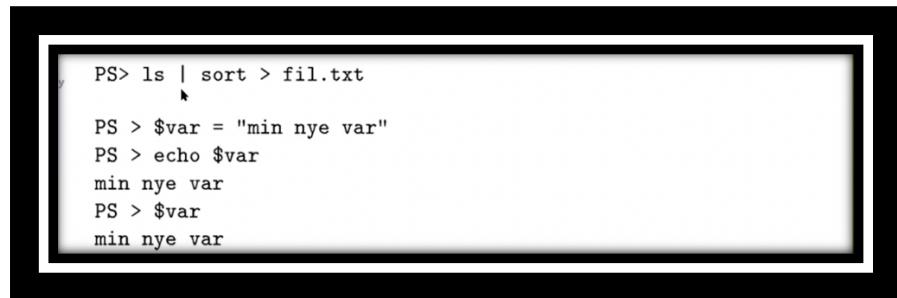
```
PS> set-executionPolicy remoteSigned
```

Da kan du kjøre shell på din pc og trusted filer. Dette kan sikre at hackere ikke laster ned dårlige script og kjører det lokalt på din maskin, hvor denne policien sørger for at det ikke kan skje så lenge det ikke kommer fra en annen trusted publisher. For å få til dette må det gjøres med en som har administrator-rettigheter.

Det er mange likheter med bash og det er også gjort eksplisitt ved at det er lagd mange aliaser for en rekke cmd.exe. på høyre side er det cmd.exe og vi ser at det står et verb først og hva man skal gjøre.

```
set-allias cat      get-content
set-allias cd       set-location
set-allias cp       copy-item
set-allias history  get-history
set-allias kill     stop-process
set-allias ls       get-childitem ↵
set-allias mv       move-item
set-allias ps       get-process
set-alalias pwd    get-location
set-alalias rm     remove-item
set-alalias rmdir   remove-item
set-alalias echo   write-output
```

Et par andre likheter gjelder pipes og omdirigering:



```
PS> ls | sort > fil.txt
PS > $var = "min nye var"
PS > echo $var
min nye var
PS > $var
min nye var
```

Det første ser ut som en standard Linux kommando, men den kan også kjøres på en PowerShell også, og den gjør omtrent det samme som et Linux shell. Det vi skal se på er at 'ls' i utgangspunktet ikke bare er tekst, det er egentlig et objekt, men likevel kan man skrive det slik som kommandoen over og man vil få ut tekstversjonen av objektet og legge det inn i en fil.txt. slik som i php skal det være et dollartegn foran en variabel når en definerer den, ellers er det akkurat sånn man definerer variablene i Linux. Siden PowerShell ikke er så ekstremt sensitiv på syntaks spiller det ingen rolle om man har med mellomrom før og etter '='-tegnet. Man trenger heller ikke å skrive echo, hvis man bare skriver \$variabel så får man innholdet skrevet ut.

## **DEMO AV WINDOWS POWERSHELL I WINDOWS 10 I VIRTUAL BOX, SET-EXECUTIONPOLICY**

Hårek starter opp Windows 10 fra imaget som ligger I oppgavene med virtualbox. Hvis man taster Windows-tasten + x, så får man opp et følgende kommando promt.



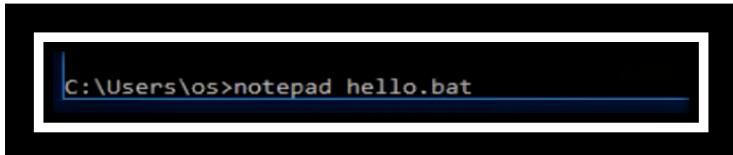
Det er mulig å sette det opp til å gi PowerShell, men default så er det cmd.exe.



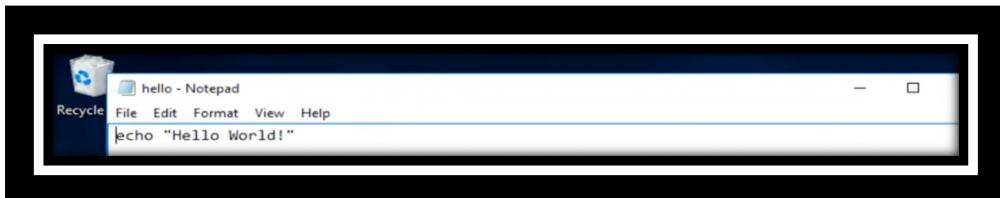
```
C:\ Select Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\os>
```

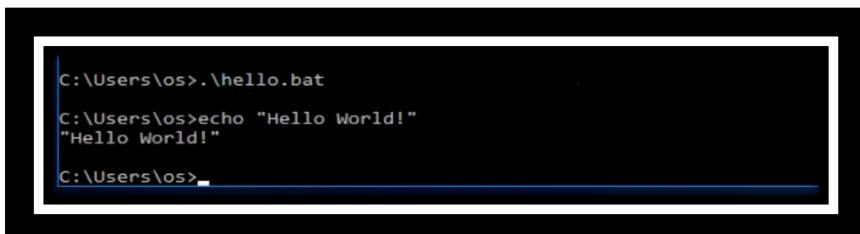
Hvis man skriver 'ls' får man opp at det ikke er en kommando og det derfor da må skrives det tilsvarende for det fra Linux til Windows som er 'dir'.



Lager en hello.bat, og da kom dette opp:



Under vises hvordan den kan kjøres:



Hvis man taster 'PowerShell' så får man opp et PowerShell. Det ser vi ved at det står PS foran



Det er viktig å vite at når man skal lagre denne filen på pc-en så skal man ikke lagre den som punktum .txt, og dette er fordi man heller skal gå og trykke på lagre som type og velge alle filer.

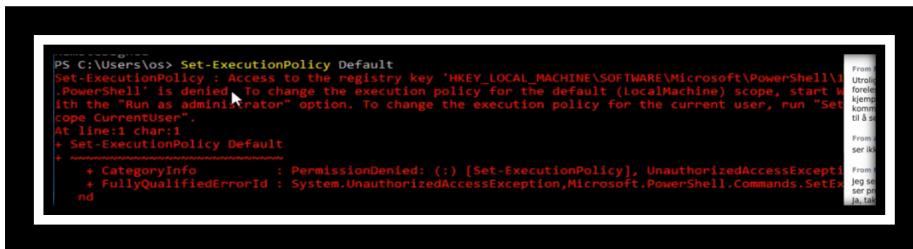


Vi fikk lov til å kjøre filen det betyr at ‘Get-ExecutionPolicy’ den henter ut policy. Under kan vi se at Hårek allerede har satt den til ‘RemoteSigned’:



```
PS C:\Users\os> Select-Object -Property ExecutionPolicy
PS C:\Users\os> notePad
PS C:\Users\os> .\hello.ps1
Hello!
PS C:\Users\os> Get-ExecutionPolicy
RemoteSigned
```

Default er den ellers satt til restricted. Under kan vi se på bildet at håret ikke får tillatelse til å endre det tilbake til ‘restricted’. Dette er fordi Hårek har en vanlig bruker, for å få lov til dette her så må han starte PowerShell som en administrator.

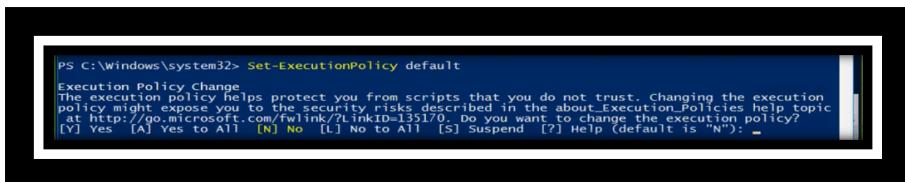


```
PS C:\Users\os> Set-ExecutionPolicy Default
Set-ExecutionPolicy : Access to the registry key 'HKEY_LOCAL_MACHINE\Software\Microsoft\PowerShell\1\PowerShell' is denied. To change the execution policy for the default (LocalMachine) scope, start Windows PowerShell with the "Run as administrator" option. To change the execution policy for the current user, run "Set-ExecutionPolicy -Scope CurrentUser".
At line:1 char:1
+ Set-ExecutionPolicy Default
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : PermissionDenied: () [Set-ExecutionPolicy], UnauthorizedAccessException
    + FullyQualifiedErrorId : System.UnauthorizedAccessException,Microsoft.PowerShell.Commands.SetExecutionPolicyCommand
```

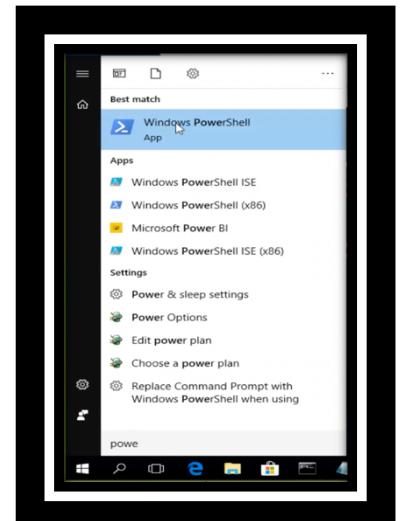
Da kan man ved å taste Windows + r og PowerShell, da får man opp et vanlig PowerShell vindu.

Se på bildet til Høyre og høyreklikk på Windows PowerShell da må du velge kjør som administrator for å åpne et PowerShell som administrator.

Da vil det være mulig å sette execution policy til default.



```
PS C:\Windows\system32> Set-ExecutionPolicy default
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution
policy might expose you to the security risks described in the about_Execution_Policies help topic
at http://go.microsoft.com/fwlink/?LinkId=135170. Do you want to change the execution policy?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): -
```



Når Hårek går tilbake til det vanlige shellet så kan han se at han ikke får lov til å kjøre skriptet, lenger og dette er fordi han må ha satt execution policy tilbake til default. Hvis han blir endre dette og kjører skriptet må han tilbake til administrator shellet og endre execution-policy til set-ExecutionPolicy RemoteSigned.

Bruke tab på samme måte som i Linux. Så kan man velge å fortsette å tabbe gjennom mange cmdlets som begynner på Get også.



```
PS C:\Users\os> Get-Command | more
```

Da ville man kunne få en lang liste med ikke bare cmdlets, men også alias-er, og alle funksjoner. Også bruker Get-Command til å finne ut hva ulike kommandoer gjør:



```
PS C:\Users\os> Get-Command ls
 CommandType      Name          Version   Source
-----      ----          -----   -----
 Alias        ls -> Get-ChildItem
```

Og så kan jeg igjen søker hvor kommandoen fra da skriver jeg Get-Command og navnet på kommandoen, og så kan man se at det kommer informasjon om versjonen til Windows og kilden:



```
PS C:\Users\os> Get-Command Get-ChildItem
 CommandType      Name          Version   Source
-----      ----          -----   -----
 Cmdlet        Get-ChildItem    3.1.0.0  Microsoft.PowerShell
```

Hvis man vil vite mer om en kommando så kan man skrive Get-Help etterfulgt av kommandoen. Siden som kommer opp, er det som likner Linux sin manual side. Et annet valg er å kunne skrive help og kommandonavnet som også gir samme manual side med informasjon og hjelpe.

## POWERSHELL-DEMO: LIKHETER MED LINUX BASH

Nå skal vi se på litt flere likheter med bash, vi kan her prøve å gjøre en liten sesjon.

Bruker mkdir for å lage en mappe dir.



```
PS C:\Users\os> mkdir dir
Directory: C:\Users\os

Mode                LastWriteTime         Length Name
----                LastWriteTime         Length Name
d---       17.04.2020 11:39            0    dir
```

Går inn i mappa og sender inn tekst til en fil.txt



```
PS C:\Users\os> cd dir
PS C:\Users\os\dir> echo "tekst" > fil.txt
PS C:\Users\os\dir>
```

Også med ls kan jeg nå se at en fil.txt eksisterer:

```
PS C:\Users\os\dir> ls
Directory: C:\Users\os\dir

Mode LastWriteTime Length Name
---- ----- ----- ----
-a--- 17.04.2020    16 fil.txt
```

Bruk cat for å se innholdet.

```
PS C:\Users\os\dir> cat .\fil.txt
tekst
```

Det er ikke alt som funket f eks å lage en fil med touch:

```
PS C:\Users\os\dir> touch file
touch : The term 'touch' is not recognized as the name of a cmdlet, function, script file, or opera
        he spelling of the name, or if a path was included, verify that the path is correct and try again.
At line:1 char:1
+ touch file
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (touch:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

Kommandoen for å lage en ny fil er New-Item etterfulgt av filnavnet.

```
PS C:\Users\os\dir> New-Item file.txt

Directory: C:\Users\os\dir

Mode LastWriteTime Length Name
---- ----- ----- ----
-a--- 17.04.2020    0 file.txt
```

Under ser vi nå 2 filer.

```
PS C:\Users\os\dir> ls
Directory: C:\Users\os\dir

Mode LastWriteTime Length Name
---- ----- ----- ----
-a--- 17.04.2020    16 fil.txt
-a--- 17.04.2020    0 file.txt
```

NYTTIG: Tast Ctrl + r, for å søke rekursivt! Du skriver det som du vil skal slå ut i feltet og skriver Ctrl + r for å søke reskursivt bak. ‘back-ile-search’.

Under skrives ls pipe sort, med pipe tegnet vil resultatet da sendes videre over i filen sort.txt.

Hvis vi da skriver og ser innholdet fra filen, så ser det ut som bildet under:



```
PS C:\Users\os\dir> cat .\sort.txt

Directory: C:\Users\os\dir

Mode                LastWriteTime     Length Name
----                -----          ----- 
-a---      17.04.2020    11:40           16 fil.txt
-a---      17.04.2020    11:41            0 file.txt
-a---      17.04.2020    11:42            0 sort.txt
```

man kan overføre et helt program fra Linux shell og kjøre det i et PowerShell.

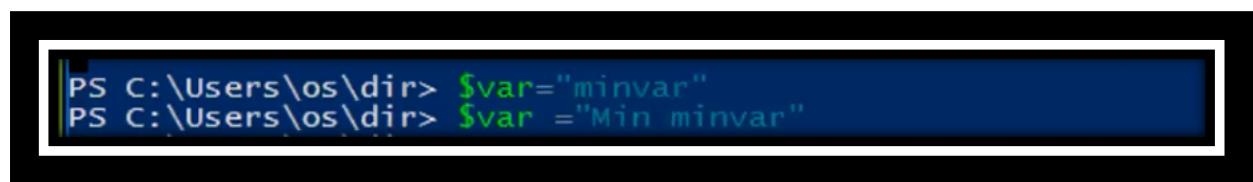
## POWERSHELL-DEMO: VARIABLER OG ENVIRONMENT-VARIABLER

Variabler her defineres nesten akkurat likt som i et Linux shell. I et linux shell kunne det ha sett slik ut:



```
PS C:\Users\os\dir> var=minvar
var=minvar : The term 'var=minvar' is not recognized as the name of a cmdlet, function, script file, or variable. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.
At line:1 char:1
+ var=minvar
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (var=minvar:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

Men det kan vi se ikke fungerer her. Dette er fordi PowerShell vil tro dette ser ut som en kommando på grunn av syntaksen. I PowerShell vil vi heller skrive \$foran variabelen navnet, i tillegg til at man også må ha med anførselstegn. Slik lager man variabler, og det spiller ingen rolle om man har mellomrom eller ikke. Som nevnt er det viktigst at man har \$ foran variabelnavnene og anførselstegn på det som skal tilhøre variabelen.



```
PS C:\Users\os\dir> $var="minvar"
PS C:\Users\os\dir> $var = "Min minvar"
```

Man kan også skrive ut verdien til variabelen ved å ha med kommandoen echo \$variabelnavn. Hvis man også bare skriver \$variabelnavnet får man også ut innholdet.



```
PS C:\Users\os\dir> echo $var
Min minvar
```

Hvis man vil finne alle variabler som er blitt definert kan man skrive Get-Variable. Da får vi opp en rekke variabler som allerede er definert. Hvis vi for eksempel skriver \$home, så får vi opp home:



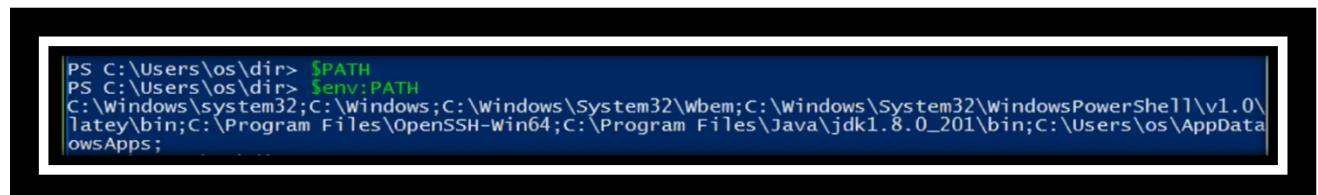
```
PS C:\Users\os\dir> $HOME
C:\Users\os
PS C:\Users\os\dir>
```

Også har vi også miljøvariablene på engelsk environment variables, og disse kan vi få ut ved å skrive ls env:



```
PS C:\Users\os>
PS C:\Users\os\dir> ls env:<pre>
```

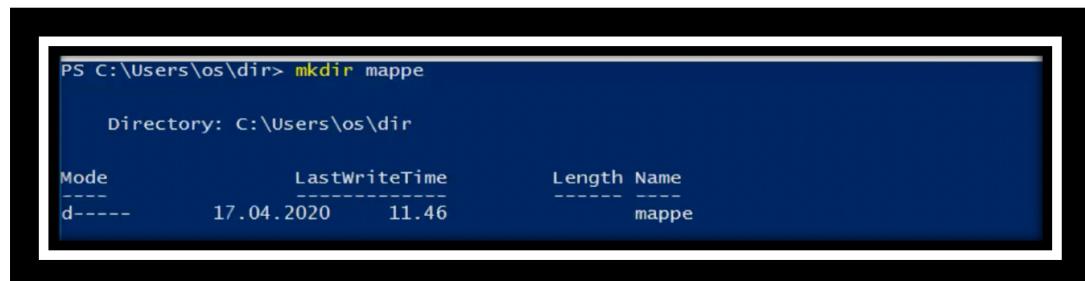
Et eksempel på en miljøvariabel som allerede finnes er \$path. Vi kunne ha tenkt oss å skrive \$path hadde gitt oss output. Men når det kommer til miljøvariabler så må man ha dollarstege env kolon slik:



```
PS C:\Users\os\dir> $PATH
PS C:\Users\os\dir> $env:PATH
C:\Windows\system32;C:\Windows;C:\Windows\System32\wbem;C:\Windows\System32\windowsPowerShell\v1.0\T
lately\bin;C:\Program Files\OpenSSH-Win64;C:\Program Files\Java\jdk1.8.0_201\bin;C:\Users\os\AppData\Ro
wsApps;
```

## POWERSHELL-DEMO: APOSTROFER OG \$(LS \$DIR)

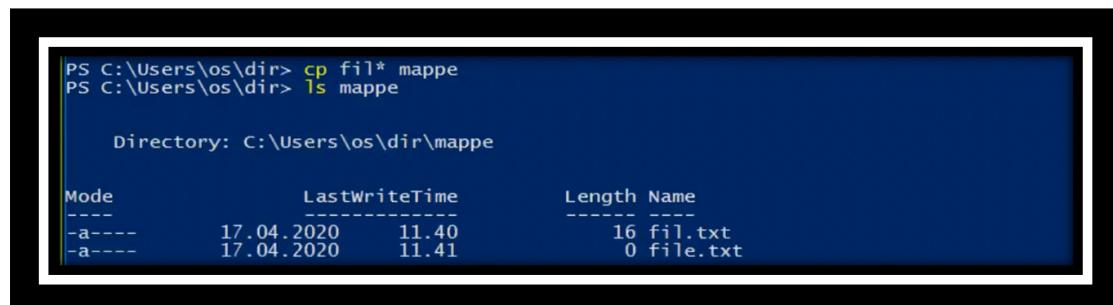
Apostrofer virker så å si som linux. La oss si vi lager en mappe som heter mappe her.



```
PS C:\Users\os\dir> mkdir mappe
Directory: C:\Users\os\dir

Mode          LastWriteTime        Length Name
----          -----          ----- 
d---          17.04.2020       11.46  mappe
```

Og så kopieres fil\* til mappe. Og så lister Hårek opp innholdet i mappen:



```
PS C:\Users\os\dir> cp fil* mappe
PS C:\Users\os\dir> ls mappe

Directory: C:\Users\os\dir\mappe

Mode          LastWriteTime        Length Name
----          -----          ----- 
-a---         17.04.2020       11.40  fil.txt
-a---         17.04.2020       11.41  file.txt
```

Så lager Hårek en variabel som er lik mappe/ har innholdet: ordet mappe.



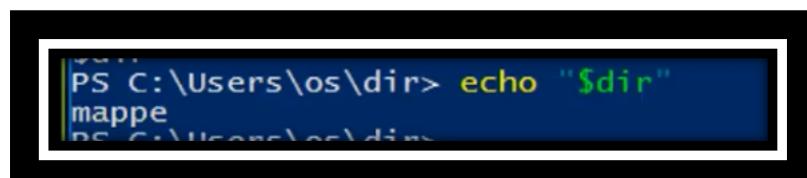
```
PS C:\Users\os\dir> $dir = "mappe"
```

Vi ser under at hvis vi skriver echo med enkelt anførselstegn og variabel inni, så fungerer det akkurat som i Linux, og da får vi bare skrevet ut det vi har skrevet ved siden av variabelen.



```
PS C:\Users\os\dir> echo '$dir'  
$dir
```

Men hvis vi bruker doble apostrofer etter echo, så får vi skrevet ut variabelens innhold.



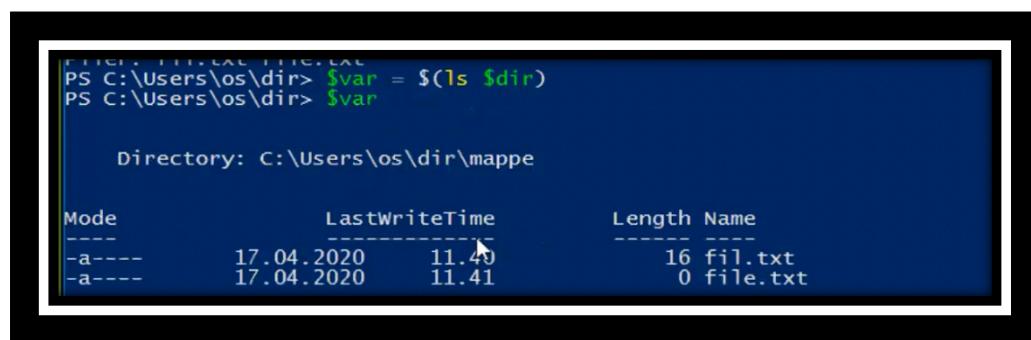
```
PS C:\Users\os\dir> echo "$dir"  
mappe  
PS C:\Users\os\dir>
```

Bildet under viser et eksempel på syntaksen som skal til for å kunne bruke en variabel inn i en streng.



```
PS C:\Users\os\dir> echo "Filer: $(ls $dir)"  
Filer: fil.txt file.txt
```

På samme måte viser bildet under at man kan lage en variabel ut av en kommando.



```
PS C:\Users\os\dir> $var = $(ls $dir)  
PS C:\Users\os\dir> $var  
  
Directory: C:\Users\os\dir\mappe  
  
Mode LastWriteTime Length Name  
---- -- - - - -  
-a--- 17.04.2020 11.40 16 fil.txt  
-a--- 17.04.2020 11.41 0 file.txt
```

I PowerShell er all output fra kommandoer eller cmdlets, er egentlig objekter. Og det er dette vi skal se på nå.

## POWERSHELL-DEMO: FIL-OBJEKTER (VIKTIG!)

Det er også greit å bemerke seg at i bildet under er det 2 ulike kommandoer skrevet, og begge gjør akkurat det samme. De betyr det samme og vil derfor gi samme output.

```
PS C:\Users\os\dir> $fil = $(ls fil.txt)
PS C:\Users\os\dir> $fil = ls fil.txt
```

men det som er et stort problem med PowerShell er at den variabelen fil som Hårek får ut nå, Ja, det ser jo bare ut som en tekststreng. Men i virkeligheten er dette et helt objekt. Og hvordan kan man sjekke at det er et objekt? Hvordan vet vi dette? Jo vi kan ta den variabelen og sende inn en pipe til Get-Member. Get-Member er en cmdlet som viser hvilke metoder og hvilke properties et objekt har.

Her sendes alle \$fil til Get-Member og da dukker det opp en lang liste med metoder og objekter som dette objektet \$fil har. Det er et stort og omfattende objekt. I motsetning til Linux fordi da hadde vi bare fått med den fil.txt, men her tar denne variabelen med seg masse metoder og properties.

```
PS C:\Users\os\dir> $fil | Get-Member
```

Og dette kan være veldig nyttig fordi vi blar lenger ned så kan vi se at \$fil har en property name. Akkurat som i java kan vi bruke denne propertien, igjen med fil, og så få ut output til det den propertien tar med seg.

```
LastWriteTimeUtc      Property   datetime LastWriteTimeUtc {get;set;}
Length                Property   long Length {get;}
Name                  Property   string Name {get;}
BaseName              ScriptProperty System.Object BaseName {get;if ($this.Extension.Length -gt 0) {$_} else {$_}}
VersionInfo            ScriptProperty System.Object VersionInfo {get=[System.Diagnostics.FileVersionInfo]::new($_.Name)}
```

Under nå skal jeg skrive variabel fil sin name:

```
PS C:\Users\os\dir> $fil.Name
fil.txt
```

Og lengdt, gir lengden til variabelen:

```
PS C:\Users\os\dir> $fil.Length
16
```

Alt i PowerShell er i utgangspunktet objekter. Hvis jeg nå bruker variabelen 'LastWriteTime' får vi ut når denne variabelen ble endret på. Og det er ikke bare en dato, fordi det er et objekt også.

Dette vil gi oss properties og metoder for 'LastWriteTime'.

```
PS C:\Users\os\dir> $fil.LastWriteTime | Get-Member | more
```

Under kan vi se at det er typen: 'System.DateTime':

```
Type: System.DateTime
Name          MemberType   Definition
----          --          -----
Add           Method      datetime Add(timespan value)
AddDays       Method      datetime AddDays(double value)
AddHours      Method      datetime AddHours(double value)
AddMilliseconds Method      datetime AddMilliseconds(double value)
AddMinutes    Method      datetime AddMinutes(double value)
```

Et DateTime objekt har også en rekke begreper som kan slenges på. Dette kan se slik ut.

```
PS C:\Users\os\dir> $fil.LastWriteTime.Year
2020
```

## POWERSHELL-DEMO: PROSESS-OBJEKTER (VIKTIG!)

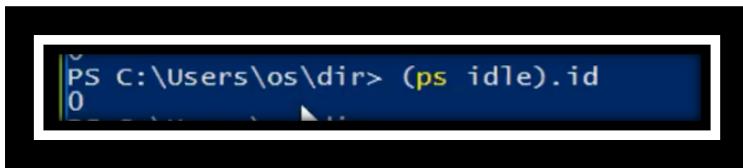
En kommando som PS gir heller ikke bare output, men det er også objekter. Vi har for eksempel en prosess som heter idle, noe som Linux også har. Vi kan på samme måte ta og sende den videre til Get-Member og pipe til more:

```
PS C:\Users\os\dir> ps idle | gm | more
```

Denne kommandoen henter alle funksjoner og properties til idle. Et eksempel på en slik property er id og at vi kan sjekke id en slik. Under definerer jeg kommandoen PS adle til en variabel og tar den sin id.

```
PS C:\Users\os\dir> $ps = ps idle
PS C:\Users\os\dir> $ps.id
0
```

Det er også mulig å gjøre dette inni en kommando slik som bildet under viser.



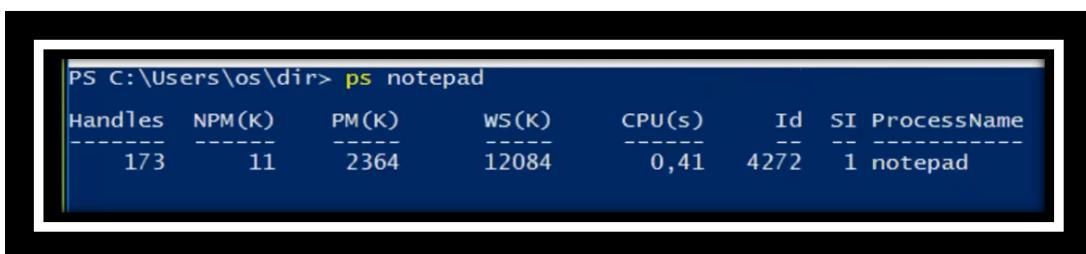
```
PS C:\Users\os\dir> (ps idle).id
```

The screenshot shows a PowerShell window with the command `(ps idle).id` entered. The output is a single digit '0'.

Under skal vi lage en notepad prosess. For det skriver man bare notepad i PowerShell. Da vil en fane som bilde under viser dukke opp:



Under skriver Hårek PS notepad. Og så blar vi gjennom propertiesene til PS og bruker StartTime.

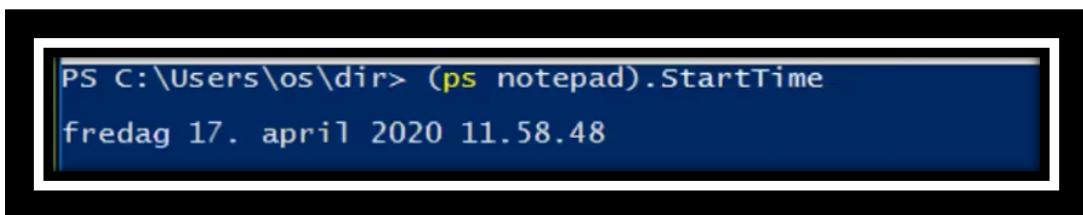


Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
173	11	2364	12084	0,41	4272	1	notepad

The screenshot shows a PowerShell window with the command `ps notepad`. The output is a table showing process details:

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
173	11	2364	12084	0,41	4272	1	notepad

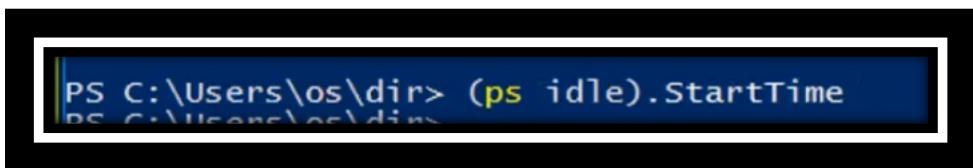
Bildet under viser starttiden til prosessen når vi åpnet opp notepad.



```
PS C:\Users\os\dir> (ps notepad).StartTime
```

The screenshot shows a PowerShell window with the command `(ps notepad).StartTime`. The output is the date and time when the process was started: "fredag 17. april 2020 11.58.48".

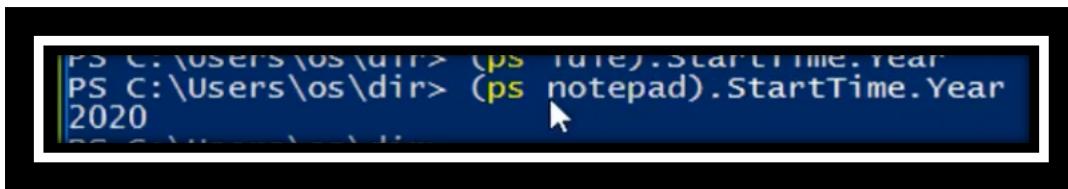
Idol er en veldig spesiell prosess.



```
PS C:\Users\os\dir> (ps idle).StartTime
```

The screenshot shows a PowerShell window with the command `(ps idle).StartTime`. The output is the date and time when the process was started: "PS C:\Users\os\dir> (ps idle).StartTime".

Under trekkes ut notepad prosessen sin start time, mer spesifikt år.



```
PS C:\Users\os\dir> (ps notepad).StartTime.Year
```

The screenshot shows a PowerShell window with the command `(ps notepad).StartTime.Year`. The output is the year the process was started: "2020".

## POWERSHELL-DEMO: ARRAY AV OBJEKTER

Noe som kan være veldig forvirrende i starten er informasjonen slik som ls. Vi ser jo at denne kommandoen gir flere filer.

```
PS C:\Users\os\dir> ls

    Directory: C:\Users\os\dir

Mode LastWriteTime      Length Name
---- -----          ----- 
d--- 17.04.2020       11.46 mappe
-a--- 17.04.2020       11.40 fil.txt
-a--- 17.04.2020       11.41 file.txt
-a--- 17.04.2020      1300 sort.txt
```

Hvis vi skriver en ls fil.txt, får vi ut et objekt som er ganske enkelt et filobjekt:

```
PS C:\Users\os\dir> ls fil.txt

    Directory: C:\Users\os\dir

Mode LastWriteTime      Length Name
---- -----          ----- 
-a--- 17.04.2020       11.40 fil.txt
```

vi kan finne ut hva objektet heter ved å skrive dette:

```
PS C:\Users\os\dir> ls fil.txt | gm | more
```

Og her ser vi, at det er av type System.IO.FileInfo

```
Windows PowerShell

TypeName: System.IO.FileInfo

Name           MemberType      Definition
----           -----          -----
LinkType       CodeProperty   System.String LinkType{get=GetLinkType; }
Mode          CodeProperty   System.String Mode{get=Mode; }
Target        CodeProperty   System.Collections.Generic.IEnumerable[...]
```

Når vi skriver, ellers så ser vi at vi får en del output, og det ser jo også ut som en array. Altså en liste. Hvis vi tar ls sin get type property, så ser vi at ls egentlig er et array.

```
PS C:\Users\os\dir> (ls).GetType()

IsPublic IsSerial Name          BaseType
-----  -----  Object[]       System.Array
```

Det man må videre huske er at ls er et array og man kan ta ut objekter akkurat slik som man gjør i vanlig kodespråk som java:

Derfor kan jeg hente objekt nummer en ved å skrive dette under:

```
PS C:\Users\os\dir> (ls)[0]

Directory: C:\Users\os\dir

Mode          LastWriteTime    Length Name
----          -----          --     --
d--- 17.04.2020      11.46      mappe
```

På denne måten ser vi at vi har et array av filer. Det samme er det med PS. Hvis vi skriver PS parenteser 4 får vi den fjerde prosessen:

```
PS C:\Users\os\dir> (ps)[4]

Handles  NPM(K)    PM(K)    WS(K)    CPU(s)   Id  SI ProcessName
-----  -----    -----    -----    -----   --  --  -----
267        13       1296      3624      436    1   1  csrss
```

## POWERSHELL-DEMO: TELLE PROSESSER

I linux har vi sett at hvis vi skal telle antall prosesser så bruker vi ps aux | wc -l. Bare som et eksempel fra PowerShell går det an å installere ssh:

```
PS C:\Users\os\dir> ssh haugerud@rex.cs.hioa.no
haugerud@rex.cs.hioa.no's password:
rex:~$
```

Etter å ha logget inn så har vi et Linux skjell inni PowerShell. Så hvis man installerer ssh kan man på denne måten logge inn på Windows maskiner. Det vi skulle se her er at for å sjekke antall kjørende prosesser i Linux, skriver vi kommandoen under.

```
rex:~$ ps aux | wc -l
255
rex:~$ logg ut
Connection to rex.cs.hioa.no closed.
```

I powershell er dette lettere? Vi vet at ps, gir et array, og da kan vi bare søkte etter ps sin lengde:

```
PS C:\Users\os\dir> (ps).Length
53
```

## POWERSHELL-DEMO: FOREACH OG FOREACH-OBJECT

### ONLINERE

Vi har sett at ls og ps og andre cmdlets er veldig kraftige ved at de ikke bare sender ut tekst som resultat, men hele objekter.

Under vil lokka gå gjennom alle filene i ls og legge sammen lengden av alle filene:

```
PS C:\Users\os> foreach ($ls in ls *.txt){$sum += $ls.Length}
```

Hvis vi tar ls-kommandoen og ser hvor mange filer vi skal få ut som ender med .txt, så ser vi at det er 4 stykker.

```
PS C:\Users\os> foreach ($ls in ls *.txt){$sum += $ls.Length}
PS C:\Users\os> ls *.txt

Directory: C:\Users\os

Mode                LastWriteTime     Length Name
----                -----          -----    ----- 
-a----   16.04.2020      21.51      0 fil.txt
-a----   27.03.2019      08.44    1216 hist.txt
-a----   19.03.2019      14.11    2676 hist2.txt
-a----   19.03.2019      13.57   4128 hist3.txt
```

Og så skriver vi \$ variabelnavnet, altså \$sum for å se ut hva det blir, og vi ser at det er 8020 som er summen av de 3 filene. Enheten her er byte. Svaret blir da 8020 byte. Vi ser med en gang hvor lett det er å plukke ut det her som lengden.

Under tar vi i en ls alle sammen (\*) som slutter med .txt eller har .txt inni seg, og så piper vi det videre til et for each objekt. Da vil nå vært objekt behandles inni i denne konstruksjonen. Og da gjør vi litt av det samme som vi gjorde opp, så forEach, altså hvert av de objektene som sendes ut av ls \*.txt, det blir pipet, og tatt imot også ønsker jeg (Hårek) å øke lengden. Etter hvert som lokka gjennomgår, vil dollar undertegn være detaljer som øker. Det vil være hvert objekt. For hver gang inni løkka vil \$\_ være ny fil.

```
PS C:\Users\os> ls *.txt | ForEach-Object {$sum += $_.Length}
PS C:\Users\os> $sum
16040
```

Når vi skriver ut summen, så ser vi at det ser litt rart ut. Vi får det dobbelte. Det skyldes, at vi ikke har initiert variabelen sum.

Så da initierer vi variabelen sum i starten lik null. Med ; så skiller vi kommandoer.

```
PS C:\Users\os> $sum = 0; ls *.txt | ForEach-Object {$sum += $_.Length}
PS C:\Users\os> $sum
8020
```

Vi kan også på den kommandolinjen legge til ; sum til slutt.

```
PS C:\Users\os> $sum = 0; ls *.txt | ForEach-Object {$sum += $_.Length}; $sum
8020
```

Denne løkka regner ut summen av alle .txt-filer sin lengde og legger det sammen.

## **POWERSHELL-DEMO: SELECT-STRING SOM ERSTATNING FOR LINUX-GREP**

Bildet under viser hva vi kan være kjent med fra shellet i Linux men å skrive ls og greppe det til tekst vil ikke fungere i her.

```
PS C:\Users\os> ls | grep txt
grep : The term 'grep' is not recognized as the name of a cmdlet, function, script file, or operabl
spelling of the name, or if a path was included, verify that the path is correct and try again.
At line:1 char:7
+ ls | grep txt
+
+ CategoryInfo          : ObjectNotFound: (grep:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

Dette er fordi i powershell så finnes det ikke noe som heter grep. Det finnes en litt tilsvarende kommando og den heter select-string.

```
PS C:\Users\os> ls | select-string hist
hist.txt:8: 6 history | Out-File -Encoding ascii > hist.txt
hist.txt:11: 9 cat .\hist.txt
hist.txt~:9: 6 history | Out-File -Encoding ascii > hist.txt
hist.txt~:12: 9 cat .\hist.txt
hist2.txt:9: 6 history | Out-File -Encoding ascii > hist.txt
```

Vi ser at dette ikke gir akkurat det vi ønsket. Det gir ikke det samme som grep. Select string går inn i filene. Og så grepper den i en på teksten inni der. Da kan vi endre output ls sånn at det blir tekst. Da tvinger vi objektet til å ta tekstform. Og først kan man pipe det til

kommandoen out-string som bildet viser under. Så vi tar ls og piper det til out-string med opsjonen -stream. Og dette er en ganske tungvint måte på å få gjøre akkurat det som grep kunne ha gjort de linux skjellet.

```
PS C:\Users\os> ls | Out-String -Stream | select-string txt
-a---- 16.04.2020 21.51 0 fil.txt
-a---- 27.03.2019 08.44 1216 hist.txt
-a---- 19.03.2019 13.46 1337 hist.txt~
-a---- 19.03.2019 14.11 2676 hist2.txt
-a---- 19.03.2019 13.57 4128 hist3.txt
```

Oftest hadde man gjort noe som bildet under viser. Da sier vi bare at vi skal liste alle som har noe med.txt å gjøre. Det er akkurat det som er greia med PowerShell, man har ferdige objekter, klare objekter, og så må man kunne hente informasjon man trenger fra de objektene.

```
PS C:\Users\os> ls *.txt
Directory: C:\Users\os

Mode                LastWriteTime       Length Name
----                -----          ---- 
-a----        16.04.2020    21.51      0 fil.txt
-a----        27.03.2019    08.44    1216 hist.txt
-a----        19.03.2019    14.11    2676 hist2.txt
-a----        19.03.2019    13.57    4128 hist3.txt
```

## JEFFEREY SNOVER OM SCRIPTING I POWERSHELL

Jeffrey snover er en PowerShell arkitekt. Det er han som ledet teamet som utviklet PowerShell. Han sier at måten han utvikler på er litt sånn vi gjør i Linux shellet. At først så tester man ut det viktigste i skjellet direkte, og får det til å virke. Og så putter han det inn i en fil, og han bruker faktisk notepad også legger han på parametere og tester og kjører og så videre. Vi skal se litt mer på Windows PowerShell ISE til å skrive PowerShell script, som generelt er veldig effektivt.

## DEMO: FUNKSJONER I POWERSHELL

Vi har sett tidligere i Linux at vi kan skrive funksjoner. Hårek logger inn på sin desktop i Linux for å vise oss noen eksempler på det igjen.



```
PS C:\Users\os> ssh haugerud@rex.cs.hioa.no
haugerud@rex.cs.hioa.no's password:
```

Under er det et eksempel på, hvor han lager en funksjon som heter hvor, der han skriver ut ordet 'her', og så bruker han funksjonen navnet på å kjøre funksjonen. Og så får vi se at det skrives ut riktig begrep.



```
rex:~$ function hvor() { echo "her"; }
rex:~$ hvor
her
```

Og så gjør han en funksjon litt kulere ved å legge til en kommando PWD og skriver ut er i print working directory.



```
her
rex:~$ function hvor() { echo "Er i $(pwd)"; }
rex:~$ hvor
Er i /home/haugerud
rex:~$
```

I bildet under er vi tilbake i Powershell, og det kan vi se ved at det står PS helt til venstre. Det vi har lagt merke til er at vi kan kopiere akkurat den samme funksjonen fra Linux over i PowerShell, og det vil fungere helt fint. Dette er enda et bevis på at PowerShell er veldig inspirert av Linux.



```
PS C:\Users\os> function hvor() {echo "Er i $(pwd)"; }
PS C:\Users\os> hvor
Er i C:\Users\os
PS C:\Users\os>
```

Vi kan også lage en litt mer avansert funksjon som tar inn input. Funksjonen under heter 'adel' og tar inn en variabel av A og B. Inne i funksjonen skal A og B adderes. Den første metoden viser at parenteser bare skriver ut tallene tilbake. Hvis man egentlig skal sende inn tall og vil at det skal komme ut som en sum, må man bare skrive funksjonsnavnet og tallene ved siden av.



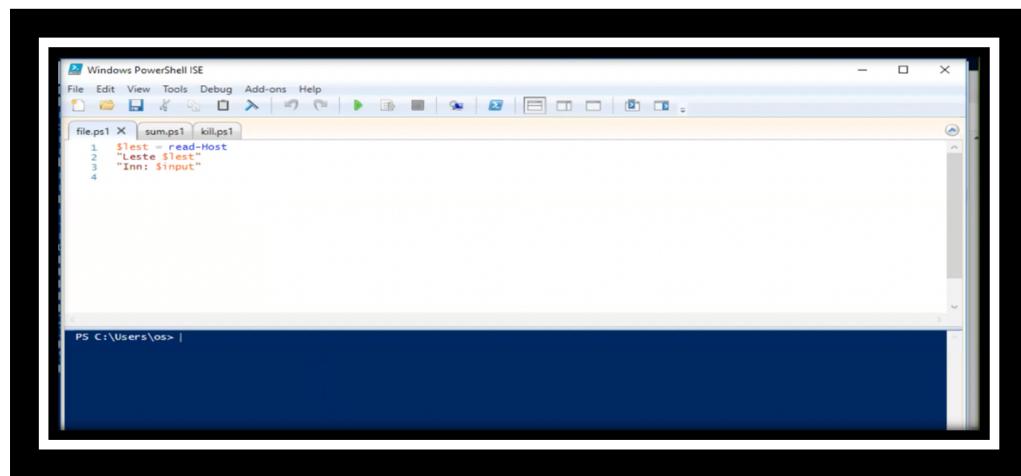
```
PS C:\Users\os> function add($a,$b) { $a + $b}
PS C:\Users\os> add 2 3
5
PS C:\Users\os>
```

## DEMO: SCRIPTING I POWERSHELL ISE, SUMMASJON AV FIL-STØRRELSER

Vi kan teste PowerShell GUI utviklingsverktøyet som er default. Under er det et bilde av hvordan det ser ut som når man søker etter applikasjonen.



Det er ganske enkelt for at man skriver script oppe og så tester og kjører nede. Dette er ofte noe vi har gjort i linux hvor vi åpner opp flere vinduer for å skrive og kjøre i en annen fane.

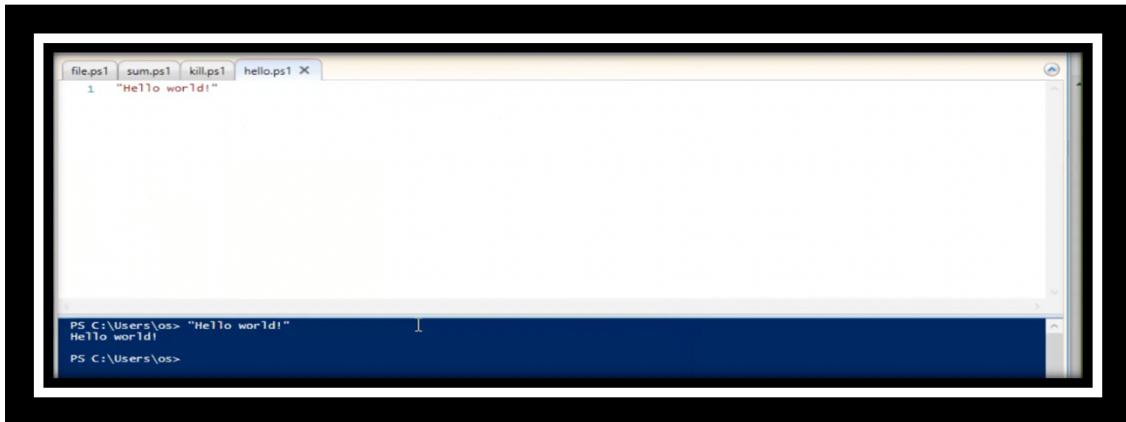


```
Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help

file.ps1 X sum.ps1 kill.ps1
1 $test = read-Host
2 "Leste $test"
3 "Inn: $input"
4

PS C:\Users\os>
```

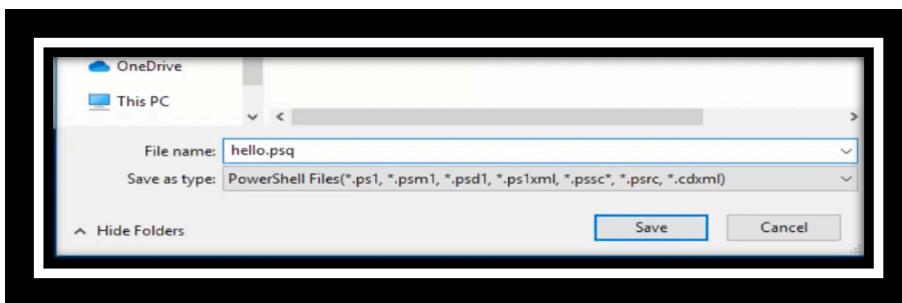
vi kan starte med å skrive en enkel hello world applikasjon. Man må ikke ha med begrepet echo i Windows scripting. Man skriver hello world med doble apostrofer i skriptet. Og under, så skal man bare taste F5 for å se at det kjører.



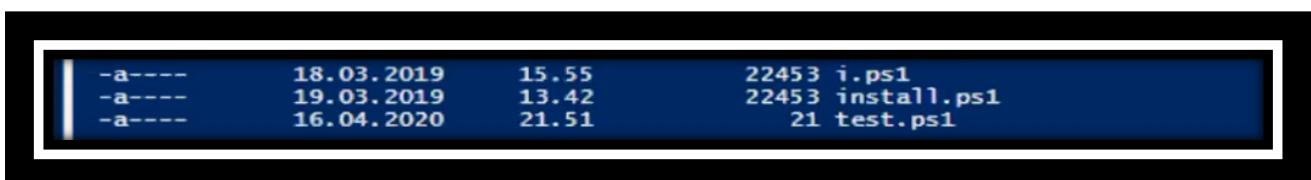
```
file.ps1 sum.ps1 kill.ps1 hello.ps1
1 "Hello world!"

PS C:\Users\os> "Hello world!
Hello world!
PS C:\Users\os>
```

Under lagrer Hårek den som hello.ps1. Hvis vi under skriver ls i powershellet. Så ser vi at vi har fått en ny fil som heter testpunkt PC en.



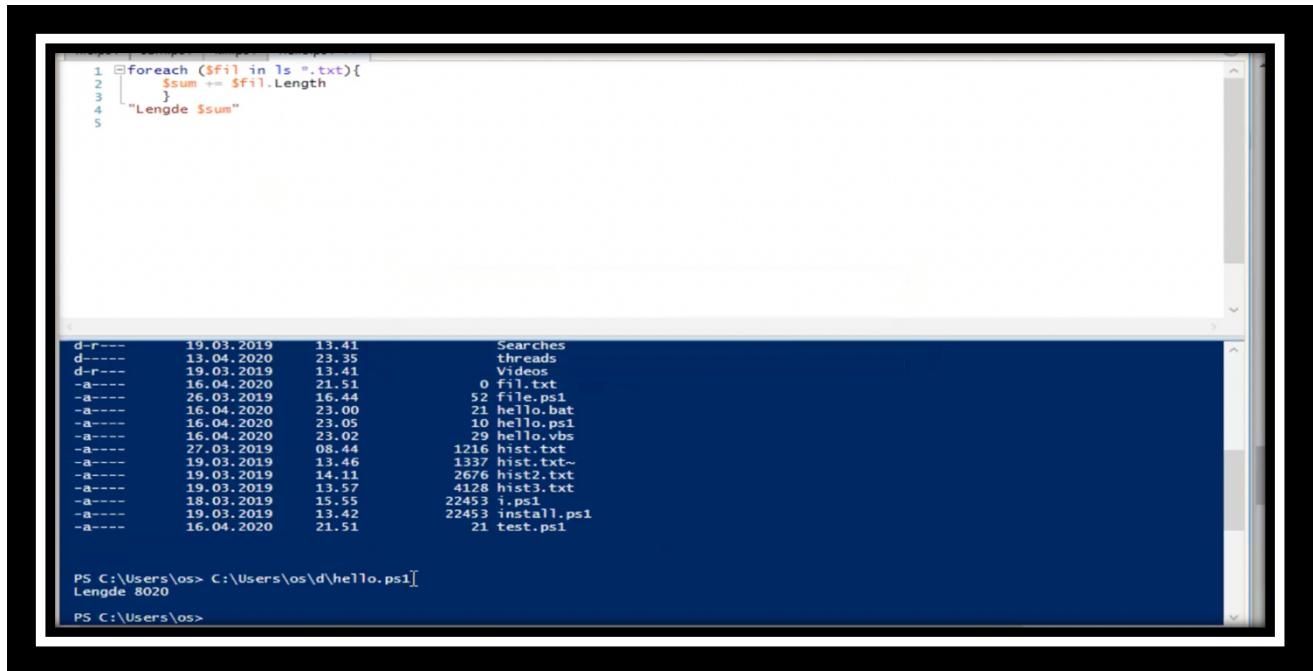
Filen:



-a---	18.03.2019	15.55	22453 i.ps1
-a---	19.03.2019	13.42	22453 install.ps1
-a---	16.04.2020	21.51	21 test.ps1

Vi kan prøve å lage et skript som legger sammen summen av størrelsen på filer.

Under skriver vi et skript som går inn for hver fil i ls, og så lager vi en variabel sum, hvor man skal plusse lengden på hver fil sin lengde, og det skal bli til variabelen sum, og så skriver vi ut lengden på summen. Man kan også i nedre fanen gå opp og ned med kontroll i og kontroll d og kjøre skriptet der i kommandolinjen.



```

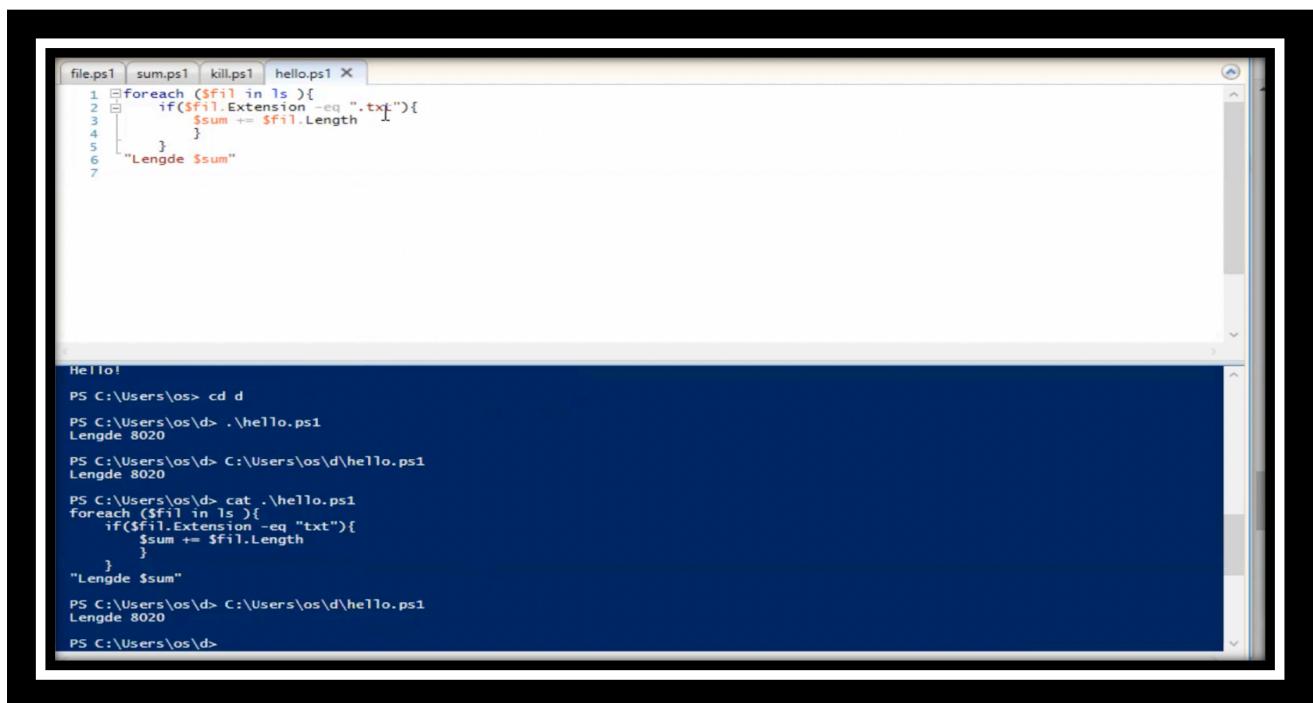
1 foreach ($fil in ls *.txt){
2     $sum += $fil.Length
3 }
4 "Lengde $sum"
5

d-r--- 19.03.2019 13.41      Searches
d-r--- 13.04.2020 23.35      threads
d-r--- 19.03.2019 13.41      Videos
-a---- 16.04.2020 21.51      0 fil.txt
-a---- 26.03.2019 16.44      52 file.ps1
-a---- 16.04.2020 23.00      21 hello.bat
-a---- 16.04.2020 23.05      10 hello.ps1
-a---- 16.04.2020 23.02      20 hello.vbs
-a---- 27.03.2019 0.44       1216 hist1.txt
-a---- 19.03.2019 13.46      1337 hist1.txt
-a---- 19.03.2019 14.11      2676 hist2.txt
-a---- 19.03.2019 13.57      4128 hist3.txt
-a---- 18.03.2019 15.55      22453 i.ps1
-a---- 19.03.2019 13.42      22453 install.ps1
-a---- 16.04.2020 21.51      21 test.ps1

PS C:\Users\os> C:\Users\os\d\hello.ps1
Lengde 8020
PS C:\Users\os>

```

Under er skriptet utvidet med en if-test hvor vi bruker en property som heter extension. Då sier vi hvis fil sin extension er lik, altså -eq, .txt, så skal man gjøre det som er inne i testen. Man må ikke ha med '.' i '.txt'.



```

file.ps1 sum.ps1 kill.ps1 hello.ps1 ✘
1 foreach ($fil in ls ){
2     if($fil.Extension -eq ".txt"){
3         $sum += $fil.Length
4     }
5 }
6 "Lengde $sum"
7

Hello!
PS C:\Users\os> cd d
PS C:\Users\os\d> .\hello.ps1
Lengde 8020
PS C:\Users\os\d> C:\Users\os\d\hello.ps1
Lengde 8020
PS C:\Users\os\d> cat ..\hello.ps1
foreach ($fil in ls ){
    if($fil.Extension -eq "txt"){
        $sum += $fil.Length
    }
}
"Lengde $sum"
PS C:\Users\os\d> C:\Users\os\d\hello.ps1
Lengde 8020
PS C:\Users\os\d>

```

## DEMO: SCRIPTING, DREP PROSESSER, KILL.PS1, ARGUMENTER

I dette skriptet skal det forsøkes å drepe prosesser med prosessenavn.

Hårek starter en notepad ved å skrive notepad ved å skrive notepad.exe i terminalen:

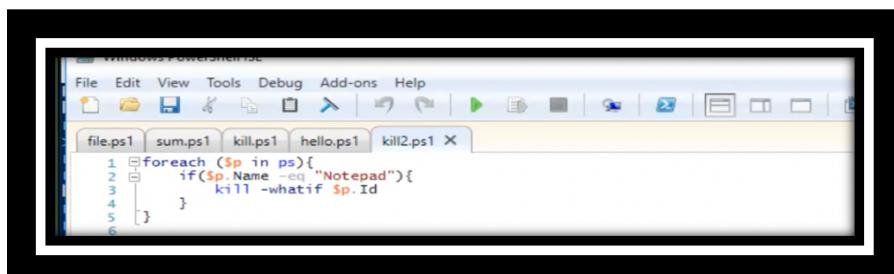


```
Lengde 00:00
PS C:\Users\os\d> notepad.exe
```

da dukket denne siden opp



Husk at han kjører skriptet med F5. Under har Hårek laget et skript hvor vi går gjennom alle linjer i ps som lister prosesser. Og så er det en if-løkke som sier at hvis prosessen sitt navn, er lik notepad, så går man inn i den løkka og dreper den prosessen med den prosess id-en. Hvis man ikke husker hvilke properties som ps kan ha, kan man gå tilbake i powershellet og tabbe seg gjennom. Eller så kan man skrive ps pipe gm eller ps pipe get-member.

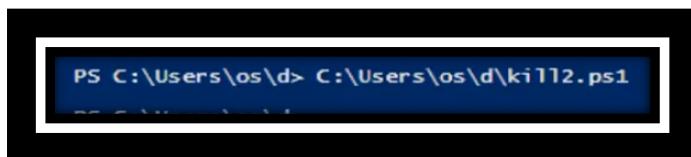


Under kan vi se at det gikk an å kjøre dette skriptet.



```
PS C:\Users\os\d> C:\Users\os\d\kill2.ps1
What if: Performing the operation "Stop-Process" on target "notepad (2960)".
```

For denne kjøringen så ble what if operasjon fjernet. Dette gir oss en mulighet til å sjekke om skriptet hadde fungert eller ikke.



```
PS C:\Users\os\d> C:\Users\os\d\kill2.ps1
```

vi skal prøve å utvide skriftet til å ta imot argumenter. Argumenter tas ut med i et array som heter args. I stedet for notepad i de doblet apostofene skriver vi da '\$args[0]'. det blir da det første argumentet som man sender med kommandoen.

I skriptet under lager vi en variabel som tilhører args sin 0 posisjon eller første argument. Og i if-statement, så bruker vi den variabelen. Det er veldig viktig å tilordne en vanlig variabel når det kommer til array, med andre ord lister.

```

file.ps1 sum.ps1 kill.ps1 hello.ps1 kill2.ps1 X
1 $s = $args[0]
2
3 foreach ($p in $ps){
4     if($p.Name -eq "$s"){
5         kill -whatif $p.Id
6     }
7 }

```

Resultatet blir da:

```

PS C:\Users\os\d> .\kill2.ps1 notepad
What if: Performing the operation "Stop-Process" on target "notepad (4464)".

```

Vi kunne også tenke oss at vi skal løpe gjennom en rekke argumenter. Og da kan man legge inn en ny foreach. Sånn går vi gjennom alle objektene i lista.

```

file.ps1 sum.ps1 kill.ps1 hello.ps1 kill2.ps1 X
1 $s = $args[0]
2
3 foreach ($p in $ps){
4     foreach($navn in $args){
5         if($p.Name -eq "$navn"){
6             kill -whatif $p.Id
7         }
8     }
9 }

```

Koden under har en ytre foreach løkke som går gjennom alle prosesser i ps og den indre foreach løkka går gjennom alle a i lista, og hvis a er lik den vi skriver inn som et argument i terminalen, så skal man drepe den prosessen.

```

1 $s = $args[0]
2
3 foreach ($p in $ps){
4     foreach($a in $args){
5         if($p.Name -eq "$a"){
6             kill -whatif $p.Id
7         }
8     }
9 }

```

I dette tilfellet dreper vi prosessen idle:

```
PS C:\Users\os\d> .\kill2.ps1 idle notepad
What if: Performing the operation "Stop-Process" on target "Idle (0)".
What if: Performing the operation "Stop-Process" on target "notepad (4464)".
```

Hvis vi prøver å drepe prosessen idle uten whatif opsjonen, så går det ikke. Dette er fordi det er en veldig grunnleggende prosess

```
PS C:\Users\os\d> .\kill2.ps1 idle notepad
kill : Cannot stop process "Idle (0)" because of the following error: Access is denied
At C:\Users\os\d\kill2.ps1:6 char:9
+         kill $p.Id
+ CategoryInfo          : CloseError: (System.Diagnostics.Process (Idle):Process) [Stop-Process], ProcessCommandException
+ FullyQualifiedErrorId : CouldNotStopProcess,Microsoft.PowerShell.Commands.StopProcessCommand
```

Og dette som det er et veldig kraftig verktøy å kunne dette. Vi kan løpe gjennom argumenter og prosesser inn i løkke. Og det kule er at output fra cmdlets er objekter. Det er ikke bare tekst, men det er objekter som vi kan plukke ut egenskaper, altså properties og bruke de.

## DEMO: POWERSHELL ONLINERE

Generelt kan man ta prosesser fra PS og pipe det til foreach. Foreach er et alias for: foreach-object. Inni {} vil man løpe gjennom hvert objekt man får tilsendt. Inne i klammeparentesene kan vi for eksempel ta å skrive et if-statement og da har man en slags default variabel (også hentet fra Linux shell). \$\_ er det objektet som testes.

```
PS C:\Users\os> notepad
PS C:\Users\os> ps | ForEach-Object {if($_.Name -eq "Notepad"){kill -whatif $_.Id} }
What if: Performing the operation "Stop-Process" on target "notepad (1504)".
```

Men inne i if-statementet, så sier vi hvis prosessen sitt navn er lik notepad, så skal vi drepe prosessen, og da har vi med what if opsjonen som da forteller oss hva som skjer hvis vi hadde runnet dette. What if er et verktøy som hjelper å se hva som kan være out av noe vi gjør. Dette er en veldig kraftig oneliner metode. Where object plukker ut objekter av en spesiell type. Også kan man parse det videre til foreach-en.

```
PS C:\Users\os> ps | where-Object {} | ForEach-Object {kill -whatif $_.Id}
```

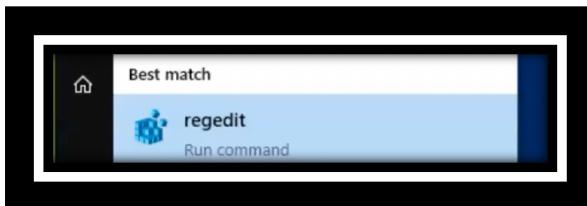
Det vi da har gjort under er å ta alle prosesser fra ps, sendt dem over til where-object, så de plukkes ut med alle som har navn, lik notepad og så skal alt det sendes videre til foreach objektet hvor? Alle prosessene der skal drepes. På denne måten blir det enda mer pipelining.

```
PS C:\Users\os> ps | Where-Object {$_.Name -eq "Notepad"} | ForEach-Object {kill -whatif $_.Id} }
```

## DEMO: WINDOWS REGISTRY, ENDRINGER MED SETITEMPROPERTY

Registry er en type database som inneholder det meste av konfigurasjoner i Windows operativsystemet. Så hvis man skal lage en større del endringer som skal gjøres for godt, så er det i registry dette det gjøres. Stort sett er alt på Windows lagret binært og ikke som tekstfiler som i Linux. Derfor er det litt andre måter å drive og gjøre endringer på. Vi skal nå se på et eksempel på hvordan man kan gjøre noen endringer i registry.

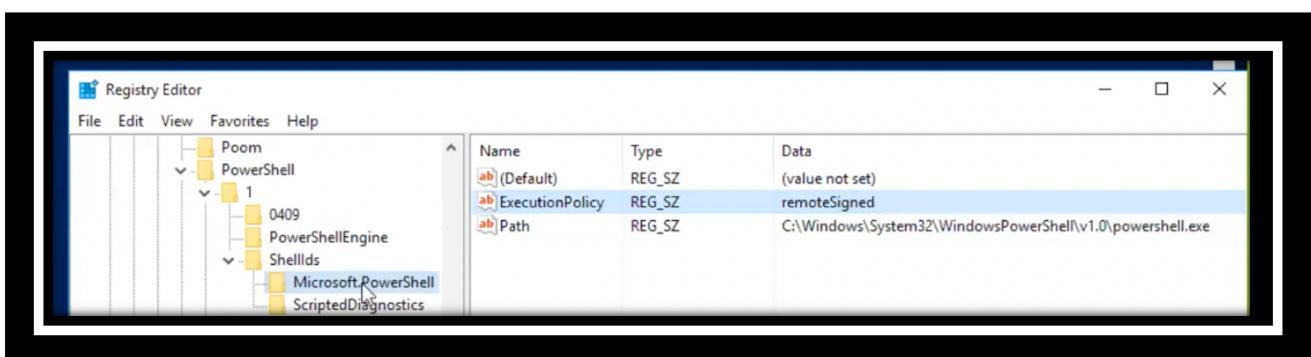
Vi skal prøve å endre registration-policy som er i registry. Det første vi gjør er å starte dette programmet som er en registry editor:



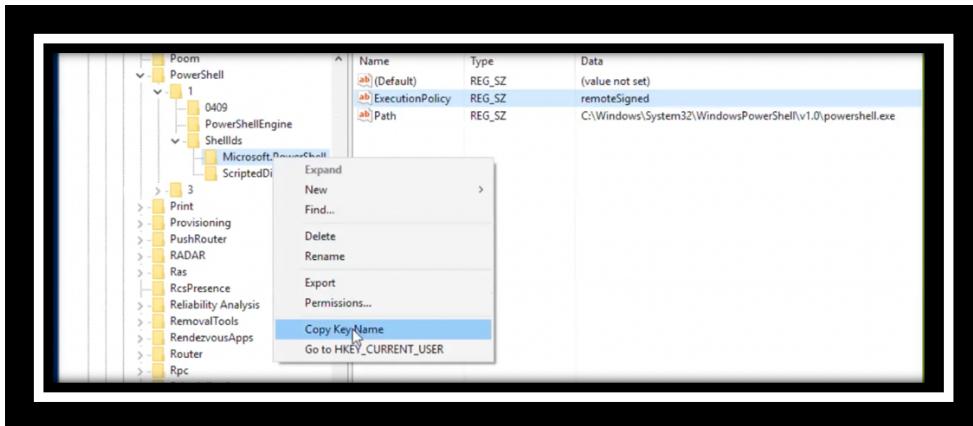
Stien under forteller om hvor vi er på ferd med å gjøre endringer.



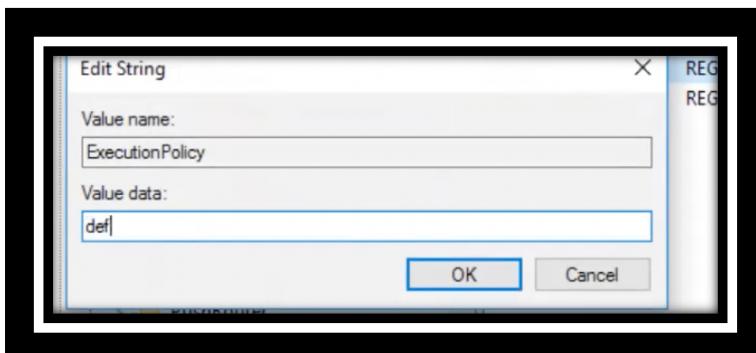
men vi kan se under her at execution-policy er definert



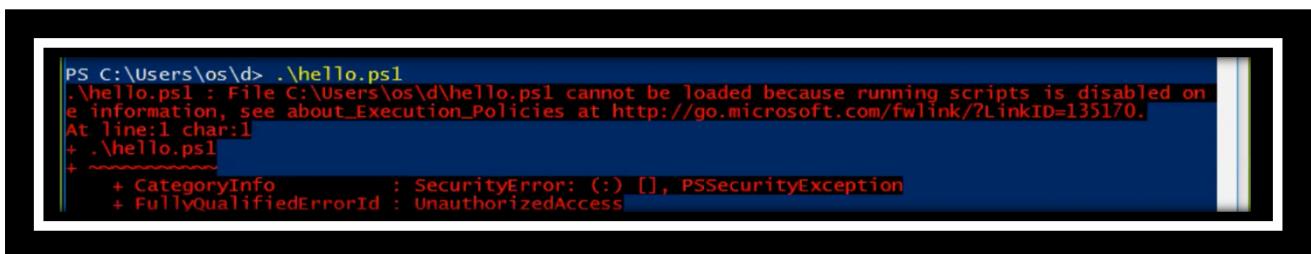
Den veien eller stien kan vi kopiere ved å høyreklikke på mappen vi fant stien execution-policy inni, også velge kopier nøkkelnavn.



Hvis vi vil endre et execution-policyen, kan vi høyreklikke 2 ganger og bare definere under. Under nå skal vi skrive i feltet default, og det betyr at vi ikke kommer til å få lov til å kjøre skript.



Hvis vi nå i powershellet prøver å kjøre et skript, ser vi at vi får en feilmelding.



Det vi nå kan gjøre er å gå tilbake til registry og endre verdien tilbake, men nå skal vi prøve å endre det tilbake fra powershellet. Under defineres hele stien vi kopierte til en variabel key. Og så skal vi prøve å endre execution policy med den nøkkelen her.



For å gjøre det, trenger vi en powershell metode. Den metoden heter set item property. Under, så prøver vi først å sette Ite-Property til stien vi lagde med key. Hvor av det vi skal endre: execution policy etterfulgt av hva vi vil endre den til.

```
PS C:\Users\os\d> Set-ItemProperty $key ExecutionPolicy RemoteSigned
Set-ItemProperty : Cannot find path 'C:\Users\os\d\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell
t.PowerShell' because it does not exist.
At line:1 char:1
+ Set-ItemProperty $key ExecutionPolicy RemoteSigned
+ ~~~~~~
+ CategoryInfo          : ObjectNotFound: (C:\Users\os\d\H...soft.PowerShell:String) [Set-ItemP
undException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.SetItemPropertyCommand
```

Det over er en vanlig feilmelding, og vi velger å sende det til null med 2>. Men vi ser at feilmeldingen til oss, sier object not found.

```
PS C:\Users\os\d> Set-ItemProperty $key ExecutionPolicy RemoteSigned 2> $null
```

Det er fordi set itemProperty venter ikke den formen vi sender inn i key som er slik:

```
|rs\os\d> $key = "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\shellIds\Microsoft.PowerShell"
```

Den propertien venter heller denne formen her, og da har vi endret før den første slashen så står det masse HKEY-greier, og det bytter vi ut med 'HKLM:'

```
|rs\os\d> $key = "HKLM:\SOFTWARE\Microsoft\PowerShell\1\shellIds\Microsoft.PowerShell"
```

Etter å ha gjort den endringen, så ser vi at det fremdeles dukker opp en feilmelding. Det kommer av at powershell her kjøres som en vanlig bruker. Og her finnes det ikke noe sudo rettigheter, så vi må være en admin bruker for å utføre kommandoen. Det vi akkurat gjorde nå må vi heller gjøre i et administrator vindu.

```
PS C:\Users\os\d> Set-ItemProperty $key ExecutionPolicy RemoteSigned
Set-ItemProperty : Requested registry access is not allowed.
At line:1 char:1
+ Set-ItemProperty $key ExecutionPolicy RemoteSigned
+ ~~~~~~
+ CategoryInfo          : PermissionDenied: (HKEY_LOCAL_MACH...soft.PowerShell:String) [Set-ItemP
undException
+ FullyQualifiedErrorId : System.Security.SecurityException,Microsoft.PowerShell.Commands.SetIT
```

Da åpner vi PowerShell, høyreklikker og velger kjør som administrator. Da må vi huske å ta med oss definisjonen hvor vi lager key variabelen. Og så kan vi kjøre kommandoen. Husk at vi må ta med key variabelen, hvor vi endret på definisjonen eller stien.



I bildet under, så er det satt med opsjonene som vi skriver inn. Og da ser vi sti navn og verdi. Slik er rekkefølgen når vi endrer fra powershellet.

```
PS C:\Windows\system32> Set-ItemProperty -path $key -name ExecutionPolicy -value RemoteSigned
```

## SORT-OBJECT OG SELECT-OBJECT, MEASURE-COMMAND (TAR TIDEN)

Sort object kan sortere med bokstaver og tall, men select objekt kan være nyttig til når vi vil trekke ut ting fra lister. Under sender vi alt fra LS til sort objekt, som skal sortere med lengde økende. Ja vi ser at de default, så sorterer den økende.

```
PS C:\Users\os\d> ls | Sort-Object Length
```

Vi kan også legge på opsjoner hvis vi skriver minus d og så tabber, så ser vi at det kommer for eksempel descending. Det betyr rett og slett at hvis du sorterer i synkende rekkefølge.

```
PS C:\Users\os\d> ls | Sort-Object -des Length

Directory: C:\Users\os\d

Mode LastWriteTime      Length Name
---- -- -              --- -
-a--- 23.04.2020 23.04          142 sum.ps1
-a--- 24.04.2020 11.04          128 kill12.ps1
-a--- 24.04.2020 10.52          125 hello.ps1
-a--- 23.04.2020 23.11           81 kill.ps1
-a--- 23.04.2020 23.59            0 file
```

Vi ser at vi har en spesifikk utskriftsmåte. Hvis vi ønsker å endre det kan vi skrive med enda en pipe, og sende det med det som heter Format-Table. Under velger vi kun å ha med navn og lengde.

```
PS C:\Users\os\d> ls | Sort-Object -des Length | Format-Table Name,Length

Name      Length
---- -----
sum.ps1    142
kill12.ps1 128
hello.ps1   125
kill.ps1     81
file        0
```

Format-table kan skrives med en forkortelse som er 'ft', slik vist under:

```
PS C:\Users\os\d> ls | Sort-Object -des Length | ft Length, name
Length Name
-----
142 sum.ps1
128 kill2.ps1
125 hello.ps1
81 kill.ps1
0 file
```

Hårek beveger seg opp med cd.. helt til han kommer til root, også skriver han samme kommando, men etter LS skriver han -r opsjonen. Dette skriver ut rekursivt.

```
PS C:\Users> ls -r | Sort-Object -des Length | ft Length, name
```

I kommandoen under sender vi alle unødvendige feilmeldinger til null. Og så sorterer vi alle objektene med med mindre og mindre lengde. Og så med select objekt velger vi de første 5. Der har vi med -første opsjonen. Til slutt skriver vi i format at vi bare vil ha med lengde og navn.

```
PS C:\Users> ls -r 2> $null | Sort-Object -des Length | Select-Object -First 5 | ft Length, name
```

Vi kan også gjøre noe lignende som bildet under og søke i alle mappene ved c:\ Og dette vil ta litt lenger tid.

```
PS C:\Users> ls -r c:\ 2> $null | Sort-Object -des Length | Select-Object -First 5 | ft Length, name
Length Name
-----
442499072 DataStore.edb
207422015 1ca2bc84dfb08d96_blobs.bin
204779835 chrome.7z
128303600 chrome.dll
121542864 MRT.exe
```

Vi har også et veldig nyttig verktøy som heter measure command. Når vi bruker denne kommandoen, så skal vi skrive den først, og alt det resterende skal være i krøllparenteser. Vi ser at vi får ut en lang liste, og dette er et tids objekt. Så det vi ser her er en utskrift men det vi vet fra powershell er at dette egentlig er et objekt

```
PS C:\Users> Measure-Command {ls -r c:\ 2> $null | Sort-Object -des Length | Select-Object -First 5}
Days          : 0
Hours         : 0
Minutes       : 0
Seconds       : 22
Milliseconds  : 896
Ticks         : 228962595
TotalDays     : 0.000265003003472222
TotalHours    : 0.00636007208333333
TotalMinutes  : 0.381604325
TotalSeconds  : 22.8962595
TotalMilliseconds : 22896.2595
```

## DEMO: DATETIME-OBJEKTER

Cmdlet-en get-date gir datoen nå. Dette ser vi under. Og dette er et date time object.

```
PS C:\Users> get-date  
fredag 24. april 2020 11.53.48
```

Som vi vet til nå, kan vi se dette ved å sende alt fra get date med pipe over til get member. I tillegg så ser vi at dette har en del metoder som add eller add-days. Dette gjør at man enkelt kan legge til en dag eller legge til en time.

```
PS C:\Users> get-date | Get-Member
```

Man kan enkelt trekke ut time eller år slik vist under.

```
PS C:\Users> (get-date).hour  
11  
PS C:\Users> (get-date).year  
2020
```

Vi skal nå se på hvordan vi kan definere en dato og senere dra det ut som et objekt. I tillegg skal vi se at vi kan manipulere på dette objektet.

```
PS C:\Users> get-date -year 2017 -month 5 -day 17 -hour 16 -minute 01 -second 05  
onsdag 17. mai 2017 16.01.05
```

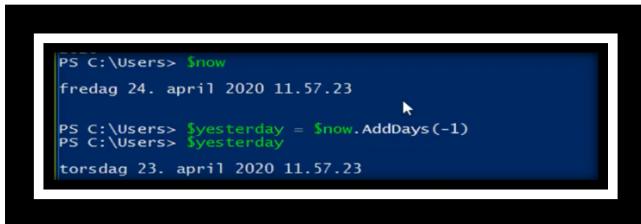
En alternativ måte å definere dette på er vi st under uten alle opsjonene som vi først brukte over. Men siden dette er objekter, skal vi prøve å manipulere disse litt for å se hva som skjer.

```
PS C:\Users> get-date "17 may 2017 16:01"  
onsdag 17. mai 2017 16.01.00
```

Vi lager en variabel nå som er lik get date.

```
PS C:\Users> $now = get-date  
PS C:\Users> $now.year  
2020
```

Vi ser under at nå har vi fredag den 24. April 2020. Vi bruker metoden add. Vi sier at yesterday variabelen skal være lik nå som vår get-date, og så legger vi til i parentesen - 1, som trekker bort en dag.



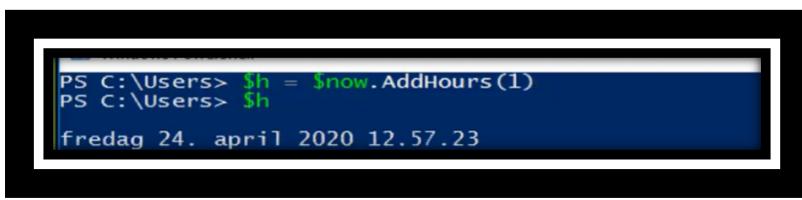
```
PS C:\Users> $now
fredag 24. april 2020 11.57.23
PS C:\Users> $yesterday = $now.AddDays(-1)
PS C:\Users> $yesterday
torsdag 23. april 2020 11.57.23
```

Igjén kan vi manipulere for å lage en variabel imorgen, altså tomorrow og sette den lik now variabelen sin addDays 1.



```
PS C:\Users> $tm = $now.AddDays(1)
PS C:\Users> $tm
lørdag 25. april 2020 11.57.23
```

Vi ser at akkurat slik som vi legger inn dager eller trekker bort dager, kan vi også legge til timer.



```
PS C:\Users> $h = $now.AddHours(1)
PS C:\Users> $h
fredag 24. april 2020 12.57.23
```

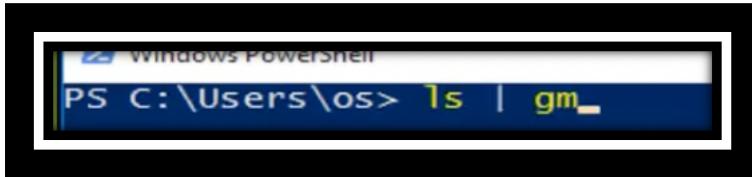
## DEMO: FINNE FILER SOM BLE ENDRET I ET GITT TIDSROM

Vi skal nå i power shellet lete etter alle filer som ble sist endret den 19. mars. Vi definerer under en variabel morgen. Vi ser at morgen variabelen er definert på morgen klokken 08.00 og kveld. Variablen er definert på kvelden klokken 22.00. Og så skal vi forsøke å finne filer som er endret innenfor tidsrommet her:



```
PS C:\Users\os> $morgen = get-date "19 march 2019 08:00"
PS C:\Users\os> $kveld = get-date "19 march 2019 22:00"
```

Da sender vi ls til get-method for å finne ut hvilke properties vi kan bruke med filer.



```
Windows PowerShell
PS C:\Users\os> ls | gm
```

Vi ser under, at vi har en profil som heter last write time.

LastAccessTime	Property	datetime LastAccessTime {get;set;}
LastWriteTime	Property	datetime LastWriteTime {get;set;}
LastWriteTimeUtc	Property	datetime LastWriteTimeUtc {get;set;}
Length	Property	long Length {get;}

det er det samme som dukker opp her opp når vi tar LS:

Mode	LastWriteTime	Length	Name
d-----	19.03.2019 14.02		.ssh
d-r---	19.03.2019 13.41		Contacts
d-----	24.04.2020 10.53		d

Kommandoen under tar for seg LS altså lister alt og sender det videre til where objekt inni parentesen. Der sier vi at det skal være skrevet større enn morgen variabelen, og mindre enn kveld variabelen.

Mode	LastWriteTime	Length	Name
d-----	19.03.2019 14.02		.ssh
d-r---	19.03.2019 13.41		Contacts
d-r---	19.03.2019 14.00		Documents
d-r---	19.03.2019 13.41		Favorites
d-r---	19.03.2019 13.41		Links

Med -r altså rekursiv søking, og c:\users så letes det rekursivt under users. I kommandoen under, så inkluderer vi at alle unødvendige uønskede feilmeldinger skal sendes videre til null. Og vi legger også til format time med kun navn og sist skrevet tid.

Mode	LastWriteTime	Length	Name
d-----	19.03.2019 14.02		.ssh
d-r---	19.03.2019 13.41		Contacts
d-r---	19.03.2019 14.00		Documents
d-r---	19.03.2019 13.41		Favorites
d-r---	19.03.2019 13.41		Links

## DEMO: POWERSHELL ARRAYS

Under ii bildet ser vi hvordan vi enklest mulig kan lage arrays i PowerShell. Og det er ved å bare ta en variabel lik et par verdier hvor verdiene skiller med komma. Vi kan også skrive ut de enkelte verdiene ved array navnet og firkantparenteser.

```
PS C:\Users\os> $a = 1, 2, 3, 4
PS C:\Users\os> $a[0]
1
PS C:\Users\os> $a[1]
2
PS C:\Users\os> $a[1] = 1
PS C:\Users\os> $a[1]
1
```

Slik vi annet vant med fra før, kan ikke vi bare lage lister slik som bildet underviser.

```
PS C:\Users\os> $tall[2] = 2
Cannot index into a null array.
At line:1 char:1
+ $tall[2] = 2
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [], RuntimeException
+ FullyQualifiedErrorId : NullArray
```

Heller kan vi definere eller deklarer arras ved å bare skrive \$Navnet-på-lista = @():

```
PS C:\Users\os> $tall = @()
```

Likevel av forskjellige verdier kan man fremdeles ikke bare deklarer verdier som bildet underviser. Måten man gjør dette på er ved å legge til verdier inni lista.

```
PS C:\Users\os> $tall += 0
PS C:\Users\os> $tall[0] = 0
Index was outside the bounds of the array.
At line:1 char:1
+ $tall[0] = 0
+ ~~~~~
+ CategoryInfo          : OperationStopped: (:) [], IndexOutOfRangeException
+ FullyQualifiedErrorId : System.IndexOutOfRangeException
```

Den første linja i bildet under viser hvordan vi kan legge til et tall i listen. Det er enkelt +=. Og så kan vi endre de verdiene vi har lagt til, slik at de blir noe annet, slik som for eksempel da jeg legger inn posisjon 0 til lik null. Det som er litt kjedelig er at vi må initiere hver verdi med +=.

```
PS C:\Users\os> $tall += 0
PS C:\Users\os> $tall[0]
0
PS C:\Users\os> $tall[0] = null
null : The term 'null' is not recognized as the name of a cmdlet, function, script file, or operable
spelling of the name, or if a path was included, verify that the path is correct and try again.
At line:1 char:12
+ $tall[0] = null
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (null:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
PS C:\Users\os> $tall[0] = "null"
PS C:\Users\os> $tall[0]
null
```

Hvis jeg nå hadde prøvd å sette inn \$tall[1] = 1, hadde ikke det gått. I et script kan vi legge til like mange ganger som vi har array elementer den setningen her:

```
PS C:\Users\os> $tall += ""  
PS C:\Users\os> _
```

Fordi hvis vi gjør dette om og om igjen, og typisk skript vil dette gjøres i en løkke, så kan man bruke arrayet, som man ønsker.

