

Institutt for datateknologi og informatikk (IDI), NTNU

Lærestoffet er utviklet for faget DCST1001 – Grunnleggende ferdigheter

Innhold

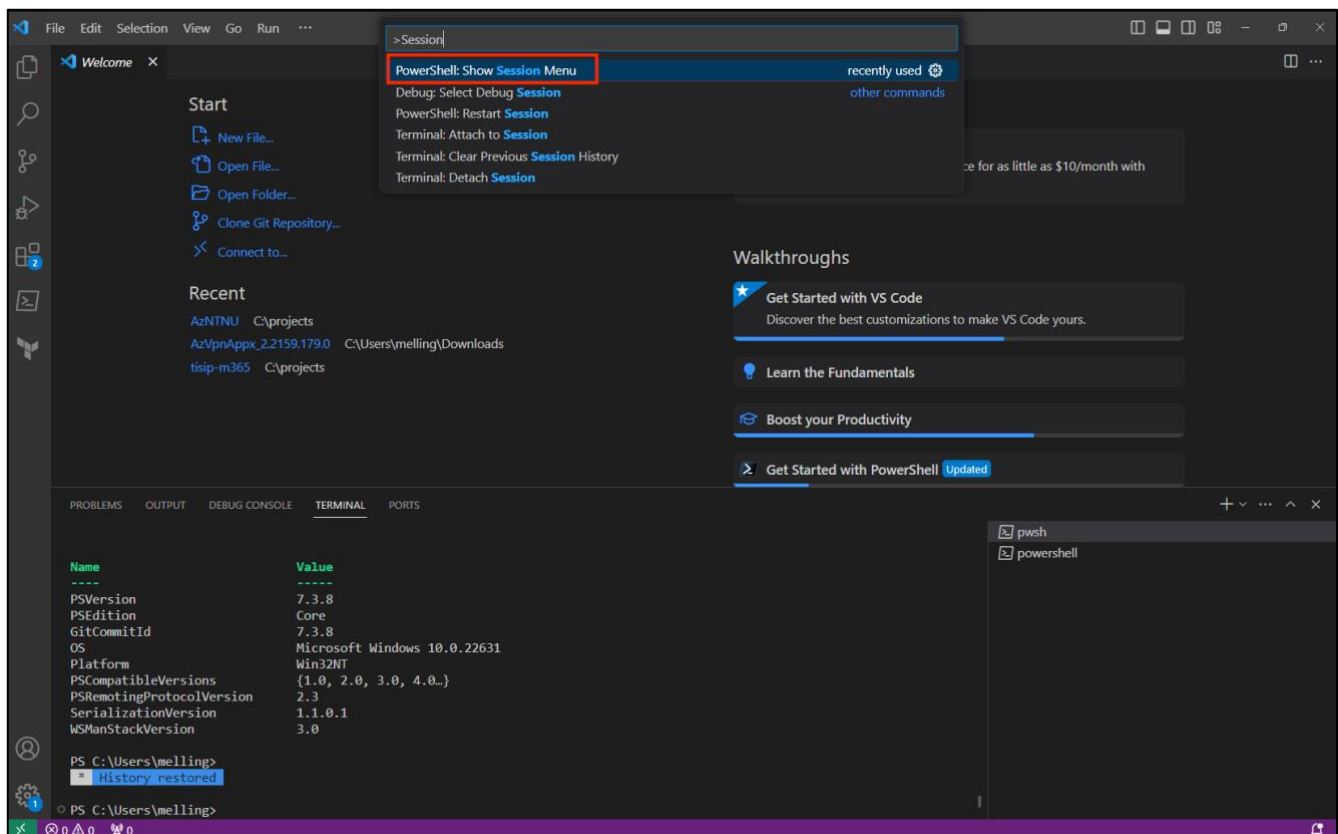
1	Introduksjon.....	2
2	Kom i gang med PowerShell.....	5
2.1	Variables	5
2.2	Data types.....	8
2.3	Boolean	8
2.4	Integers og Floating Points	8
2.5	Strings	9
2.5.1	Kombinere string og variabler.....	10
2.5.2	Dobbelt og enkelt anførselstegn	10
2.6	Objects	11
2.7	Get-Member cmdlet	12
2.8	Hvordan bruke <i>Methods</i>	12
2.9	Data Structures	13
2.10	Arrays.....	13
2.10.1	Definere et Array	14
2.10.2	Lese fra et Array	14
2.10.3	Editere element i et array.....	15
2.10.4	Legge til element i et array	15
2.11	ArrayLists.....	16
2.12	Hashtables	17
2.12.1	Legg til / editere innhold i hashtable	19
2.13	Custom objects	19
3	Referanse	21

1 Introduksjon

PowerShell er et objektorientert scriptspråk (object-oriented programming, OOP), som tidligere kun var for Microsoft Windows, som benyttes for å automatisere oppgaver. Objektorientert vil det si at språket fokuserer på objekter i stedet for handling. En identifiserer de objektene en vil behandle og deretter utfører endringene, eventuelt henter ut informasjon. Kommandoene i PowerShell heter cmdlets, uttales commandlets, og er oppbygd på verb-substantiv form. PowerShell 7 er i likhet med Windows PowerShell et .NET-program som er bygget på toppen av og utnytter .NET rammeverket (framework). Til forskjell fra tidligere versjoner av PowerShell er PowerShell 7 en komplett og åpen kildekode basert på open source .NET Core Framework ved hjelp av .NET Core 3.1. Dette har vært en enorm endring, som naturligvis også har skapt noen utfordringer.

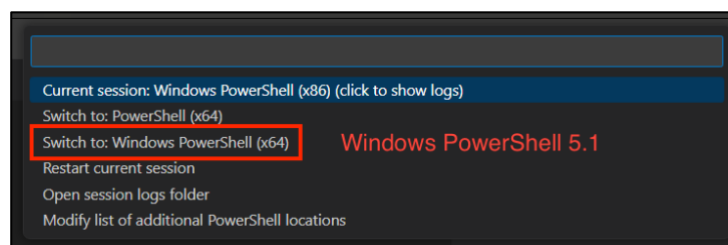
I august 2016 ble PowerShell Core lansert, som var en den første kryss plattform versjonen av PowerShell. Dette resulterte i at PowerShell også ble populært på Linux. Dessverre så det ikke ut til at det fikk den nødvendige trekraften det hadde håpet på for å erstatte PowerShell 5.1. PowerShell 7 er arvtakeren for PowerShell Core 6.x-produktene, samt Windows PowerShell 5.1. PowerShell 7 lansert versjon 7.2 preview 2 i desember 2020 (*PowerShell 7.2 Preview 2 Release*, 2020) For å få tilgang til alle kommandoene i PowerShell må en installere noe som kalles for moduler. PowerShell kan bare laste inn moduler som inneholder cmdlets og andre binære artefakter gitt at utvikleren av modulen har aktivert dette. PowerShell-teamet konverterer de fleste kommandoene, som fra før var å finne i kjernen av PowerShell, til moduler. Dette var med på å sørge for at blant annet Active Directory-modulen fungerer med PowerShell 7.

Merk at det fortsatt er noen Microsoftprodukter og andre eksterne som enda ikke har endret på modulene sine slik at de kan brukes i PowerShell 7. For å komme seg rundt dette problemet har PowerShell gjort det mulig med en kompatibilitetsløsning som gjør det mulig for de fleste kommandoer som tidligere er utviklet for Windows PowerShell å fungere i PowerShell 7. For den som kjører Windows er det heller ikke noe problem å kjøre PowerShell 5.1 sammen med PowerShell 7, hvor en kan ha flere terminaler kjørende i for eksempel Visual Studio Code. En kan trykke CTRL + SHIFT + P, og velg deretter PowerShell: Show Session Menu.



Figur 1 - VS Code, velg PowerShell versjon

Deretter må en velge Windows PowerShell (x86 eller x64). En kan deretter sjekke PowerShell versjonen i terminal med `$PSVersionTable`.



Figur 2 - Velg Windows PowerShell



Figur 3 - Sjekk PowerShellversjon i terminal

Kompatibilitetsløsningen lar en bruke PowerShell 7 til å administrere et bredt spekter av funksjoner i Windows Server, som vi skal se på i dette faget. Det er, som tidligere nevnt, et lite sett med funksjoner, moduler og kommandoer som en ikke kan bruke med PowerShell 7, men vi skal se om det tilgjengelige work-arounds for de aktuelle tilfellene (Lee, 2020).

Over de siste årene har PowerShell blitt ganske utbredt som et programmeringsspråk, som gjenspeiles i tilgjengelige moduler samt kontinuerlig utvikling. Dette kommer blant annet av Microsoft sin beslutning om å gå fra å bare være tilgjengelig på Windowsplattformen til å være et produkt som er tilgjengelig kryss av plattformer. PowerShell har hjulpet ansatte innen IT med overvåking, kontroll, administrasjon og automatisering av Windows-operativsystemet og applikasjoner som kjører i Windows Server-miljøet. Programvaren er blitt ganske utbredt, dette vises godt med tanke på alle modulene som nå finnes til PowerShell.

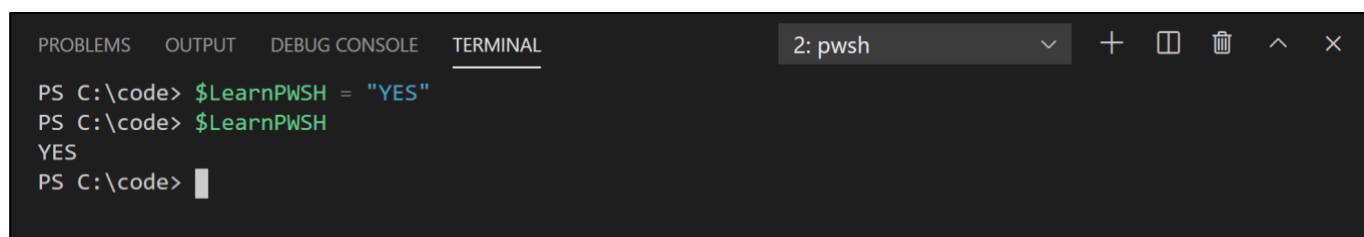
2 Kom i gang med PowerShell

I dette kapitlet skal vi se på fire grunnleggende konsepter i PowerShell: *variables*, *data types*, *objects* og *data structures*. Disse konseptene er grunnleggende for omtrent alle vanlige programmeringsspråk, men det er, som vi nevnte innledningsvis, noe som gjør PowerShell karakteristisk, og det er at alt i PowerShell er objekter. Det gir kanskje ikke så mye til eller fra nå helt i starten, men husk det når en går gjennom resten av dette kapitlet.

2.1 Variables

En variabel er et sted hvor en kan lagre verdier. Du kan tenke på en variabel som en digital boks for oppbevaring. Når en ønsker å benytte en verdi eller informasjon flere ganger i løpet av et script eller en kommando, kan en legge denne informasjonen i en variabel. I stedet for å skrive det samme tallet eller informasjonen om og om igjen i koden, kan en legge inn en variabel som henter opp denne verdien når en trenger det i koden eller scriptet. Det som er viktig å merke seg, som også navnet gjenspeiler, er at en variabel sitt innhold kan endre seg. For å fortsette med samme visualisering som ovenfor, en kan både legge til informasjon i den digitale boksen, eller bytt ut det som er i boksen med noe annet, eller. Som en vil se senere vil bruken av variabler la deg bygge kode som kan håndtere en generell situasjon, i motsetning til å være hardkodet, eller skreddersydd til et bestemt scenario (Bertram, 2020).

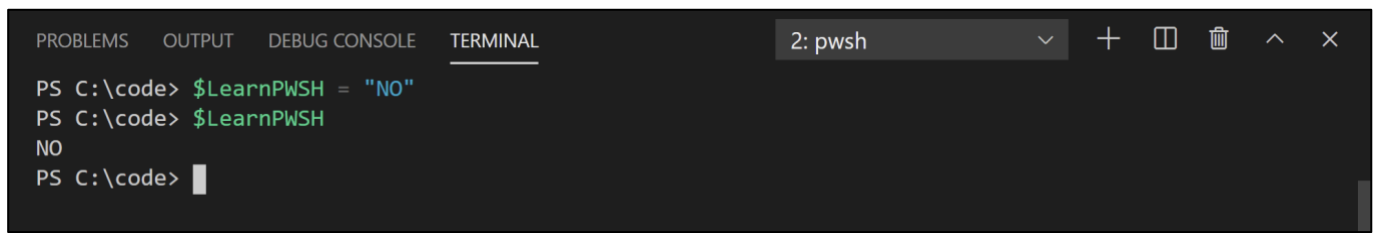
Alle variabler i PowerShell starter med et dollartegn (\$), som indikerer for PowerShell at du setter en variabel og ikke en cmdlet, funksjon, skriptfil eller kjørbart fil. For eksempel hvis du vil vise verdien av en variabel ved navn LearnPWSH, kan en skrive inn variabelen i terminalen. Først gis variabelen en verdi, og deretter listes verdien ut ved å kalle på denne variabelen.

A screenshot of a PowerShell terminal window within the Visual Studio Code editor. The terminal title bar shows '2: pwsh'. The command prompt is 'PS C:\code>'. The user enters '\$LearnPWSH = "YES"', and the prompt changes to 'PS C:\code> \$LearnPWSH'. The output 'YES' is displayed. The prompt returns to 'PS C:\code>'.

Figur 4 - Variabel i VC Code terminal

Det finnes også variabler med allerede eksisterende innhold i PowerShell. Disse omtales som *automatic variables*. Et eksempel er \$host som inneholder et objekt som representerer gjeldende host applikasjon for PowerShell. Du kan bruke denne variabelen til å representere den gjeldende hosten i kommandoer eller for å vise eller endre egenskapene til hosten. En annen variabel er \$null, som inneholder en tom verdi. En kan bruke denne variabelen til å representere et fravær eller udefinert verdi i kommandoer og skript. PowerShell behandler \$null som et objekt med en verdi, slik at du kan bruke \$null til å representere en tom verdi i en serie verdier. En kan kanskje lure på hvorfor vi ønsker å definere noe som \$null. Men det er overraskende nyttig, spesielt om en gir en variabel en verdi som svar på noe annet, som resultatet av en bestemt funksjon eller lignende. Hvis en sjekker variabelen og ser at verdien fortsatt er \$null, vet en at noe har gått galt i funksjonen og kan handle deretter. En kan også bruke *Get-Variable* for å returnere alle tilgjengelige variabler. Denne kommandoen vil liste opp alle variablene som er i minnet i skrivende stund, og som vi nevnte ovenfor, her vil en finne variabler som en selv ikke har definert (SteveL-MSFT, u.å.).

Du kan endre verdien til en variabel ved å skrive inn \$ og variabelnavnet og deretter likhetstegn og den nye verdien.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: pwsh
PS C:\code> $LearnPWSH = "NO"
PS C:\code> $LearnPWSH
NO
PS C:\code> 
```

Figur 5 - Endre innhold i variabel

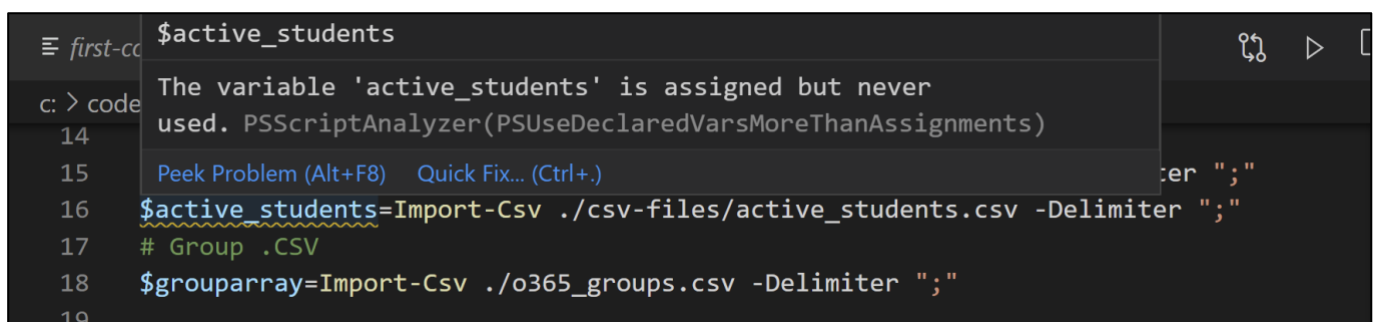
Hvis variabelen er tom, vil det ikke listes ut noen verdi. Noen vil også se følgende melding fra PowerShell: *The variable '\$variabelnavn' cannot be retrieved because it has not been set.*



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: pwsh
PS C:\code> $LearnPowerShell
PS C:\code> 
```

Figur 6 - Variable uten innhold

Hvis en ikke ser feilmeldingen *The variable '\$variabelnavn' cannot be retrieved because it has not been set*, kan det være fordi Set-StrictMode ikke er slått på. Ved å slå på streng modus ber en PowerShell om å vise feil når en ikke overholder god kodingspraksis. For eksempel tvinger streng modus PowerShell til å returnere en feil når du refererer til en objekttegenskap som ikke eksisterer eller en udefinert variabel. Det regnes som god praksis å slå på denne modusen når en skriver script siden det tvinger deg til å skrive renere og mer forutsigbar kode. Når en kjører interaktiv kode fra PowerShell-konsollen, brukes denne innstillingen vanligvis ikke. For mer informasjon om streng modus kan en kjøre kommandoen *Get Help Set-StrictMode Examples*. VS Code har også mye nyttig informasjon som kan vises til vedkommende som sitter og skriver script. Det kan for eksempel være at en definert en variabel, men aldri brukt den i scriptet. Her markert med gul linje under variabelen. Ved å holde musa over variabelen vil en se informasjonene som vises under.



```
first-cd $active_students
c: > code The variable 'active_students' is assigned but never used. PSScriptAnalyzer(PSUseDeclaredVarsMoreThanAssignments)
14
15 Peek Problem (Alt+F8) Quick Fix... (Ctrl+.)
16 $active_students=Import-Csv ./csv-files/active_students.csv -Delimiter ";"
17 # Group .CSV
18 $grouparray=Import-Csv ./o365_groups.csv -Delimiter ";"
19
```

Figur 7 - VS Code Variabel som ikke er i bruk

En annen ofte brukt variabel er `$LASTEXITCODE`. En kan i PowerShell kjøre eksterne kjørbare applikasjoner som for eksempel `ping.exe`. De fleste av oss vet at `ping` sender ut en pakke for å se om en får svar fra en annen maskin. Dette kan være en nettside eller andre typer tjenester. Når applikasjonene er ferdig med å kjøre, avsluttes de med en *exit code* eller *return code* som indikerer en melding. Vanligvis ser en 0, som indikerer suksess, og alt annet betyr enten en feil eller en annen avvik. For `ping.exe` indikerer en 0 at det klarte å pinge en node, mens en 1 indikerer at den ikke klarte pinge. Når `ping.exe` kjører får en, som forventet en *output*, men ikke en *exit code*. Det er fordi koden er skjult inne i `$LASTEXITCODE`. Verdien av `$LASTEXITCODE` er alltid koden til den siste applikasjonen som ble kjørt.

```
PS C:\code> ping www.ntnu.no

Ping request could not find host www.ntnu.no. Please check the name and try again.
PS C:\code> $LASTEXITCODE

1
PS C:\code> ping www.ntnu.no

Pinging lvs160vip02.it.ntnu.no [129.241.160.102] with 32 bytes of data:
Reply from 129.241.160.102: bytes=32 time=4ms TTL=56
Reply from 129.241.160.102: bytes=32 time=5ms TTL=56
Reply from 129.241.160.102: bytes=32 time=3ms TTL=56
Reply from 129.241.160.102: bytes=32 time=4ms TTL=56

Ping statistics for 129.241.160.102:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 3ms, Maximum = 5ms, Average = 4ms
PS C:\code> $LASTEXITCODE

0
PS C:\code>
```

Figur 8 - PowerShell og ping.exe med \$LASTEXITCODE

I tillegg har PowerShell en type *automatic variables* som kalles *preference variables*. Disse variablene styrer standard oppførsel for forskjellige output. Dette kan for eksempel være *Error*, *Warning*, *Verbose*, *Debug*, and *Information*. Du finner en liste over alle *preference variables* ved å kjøre `Get-Variable` og filtrere etter alle variablene som slutter på *Preference*, som vist her: `PS> Get-Variable -Name * Preference`.

```
PS C:\code> Get-Variable -Name *Preference

Name                           Value
----                           -
ConfirmPreference              High
DebugPreference                 SilentlyContinue
ErrorActionPreference           Continue
InformationPreference           SilentlyContinue
ProgressPreference              Continue
VerbosePreference               SilentlyContinue
WarningPreference               Continue
WhatIfPreference                False

PS C:\code>
```

Figur 9 - Preference variables

Disse variablene kan brukes til å konfigurere de forskjellige typene *output* fra PowerShell. For eksempel, hvis en har gjort en feil vil skjermen blir overfylt av rød tekst med forskjellige feilmeldinger. Hvis en av en eller annen grunn ønsker å ikke vise eventuelle feilmeldinger når en kjører scriptene sine, kan en omdefinere `$ErrorActionPreference`-variabelen til `SilentlyContinue` eller `Ignore`, som begge vil fortelle PowerShell for ikke å sende ut noen feiltekst: `PS> $ErrorActionPreference = 'SilentlyContinue'`. Som de fleste skjønner er det å ignorere feilmeldinger generelt sett en dårlig praksis. En vil jo helst vite om noe er galt, og eventuelt hva, så en kan handle deretter. Det ville vært veldig

dumt om en kunne satt varselampe for lite olje på bilen til å aldri lyse, siden det er så plagsomt med mange lamper i dashbordet (Bertram, 2020).

2.2 Data types

Når en legger inn et innhold i en variable kan en definere typen data som denne variabelen inneholder. Vi skal ikke gå igjennom alt sammen i detalj i dette kapitlet, men det en trenger å vite er at PowerShell har flere datatyper, eksempelvis: bools, strings, and integers, og en kan endre datatypen til en variabel uten at en får feil. PowerShell kan selv finne ut datatyper basert på verdiene en gir den. Hva som faktisk skjer under panseret, er litt for avansert i forhold til hva vi tenker å se på i dette kapitlet. Det som er viktig er at en forstår de grunnleggende typene og hvordan de samhandler.

```
PS C:\code> $innholdstest = 1
Integer
PS C:\code> $innholdstest
1
PS C:\code> $innholdstest = "en"
String
PS C:\code> $innholdstest
en
PS C:\code> $innholdstest = $true
Boolean
PS C:\code> $innholdstest
True
PS C:\code> 
```

Figur 10 - Integer, String og Boolean verdi i variabel

2.3 Boolean

Omtrent alle programmeringsspråk bruker *boolean value*, som har en *true* eller *false* verdi (1 eller 0). Boolean brukes for det oftest til å representere binære forhold, som for eksempel en bryter som er av eller på. I PowerShell kalles det bools, og de to *boolean value* er representert av de automatiske variablene \$true og \$false. Disse automatiske variabler er hardkodet til PowerShell og kan ikke endres. I figuren ovenfor vises hvordan en setter variabelen innholdstest til \$true.

2.4 Integers og Floating Points

I PowerShell er det to hovedmåter en kan representere tall på: *Integer* eller *Floating Points*. Integer, også kalt heltall, inneholder bare hele tall og vil avrunde alle desimalinnganger til nærmeste heltall. Heltalls datatyper kommer i både *signed* og *unsigned*. *Signed* datatype kan lagre både positive og negative tall, mens *unsigned* datatype lagrer verdier uten tegn. Som standard lagrer PowerShell heltall ved å bruke den 32-bits *signed* Int32-typen. Det er bitantallet som bestemmer hvor stort, eller hvor lite, et tall variabelen kan inneholde. I dette tilfellet vil det si alt i området fra 2,177,483,648 til 2.147.483.647. For tall utenfor det området kan du bruke den 64-bits signerte Int64-typen, som strekker seg fra -9,223,372,036,854,775,808 til 9,233,372,036,854,775,807.

La oss gå gjennom trinnene i figuren nedenfor. Først opprettes en variabel ved navn \$tall og den får verdien 1. Deretter sjekkes innholdet ved å skrive \$tall. Som vi ser, PowerShell tolker 1 som en Int32. Deretter endres \$tall-variabelen til desimaltallet 1.5. Når vi sjekker innholdet, ser vi at PowerShell er flink til å kjenne igjen innholdet som blir lagt i variablene. Dette stemmer overens med det vi nevnte tidligere i denne leksjonen. PowerShell har endret typen fra *Int32* til *Double*. Dette er fordi PowerShell vil endre en variabel type avhengig av verdien. En kan tvinge

PowerShell til å behandle en variabel som en bestemt type ved å bruke for eksempel [Int32]-syntaks foran variablene. En ser nå at når variablene blir tvunget til å behandle 1,5 som et heltall runder det opp til 2.

```
PS C:\code> $tall = 1
PS C:\code> $tall
1
PS C:\code> $tall.GetType().name
Int32
PS C:\code> $tall = 1.5
PS C:\code> $tall
1.5
PS C:\code> $tall.GetType().name
Double
PS C:\code> [Int32]$tall
2
PS C:\code> 
```

Figur 11 - Variablen \$tall med innhold

Double tilhører klassen av variabler som kalles *floating point* variabler. Selv om de kan brukes til å representere hele tall, blir variabler med *floating point* oftest brukt til å representere desimaler. Den annen hovedtype variabel med *floating point* er *Float*. Det blir ikke noen dypdykk i hva *Float* og *Doble*, men det en trenger å vite er at selv om *Float* og *Double* er i stand til å representere desimal tall, kan disse typene være upresise. Et eksempel på det er vist i figuren under, hvor PowerShell bruker *Double*-typen som standard. Legg merke til hva som skjer når en plusser \$tall med seg selv og bruker [Float]-syntaksen. Igjen, årsakene er utenfor omfanget av denne boken, men vær oppmerksom på at feil som dette kan skje når du bruker *Float* og *Double*.

```
PS C:\code> $tall = 0.123565332
PS C:\code> $tall.GetType().name
Double
PS C:\code> $tall + $tall
0.247130664
PS C:\code> [float]$tall + [float]$tall
0.247130662202835
PS C:\code> 
```

Figur 12 - Legge sammen float-variabler

2.5 Strings

Denne typen variabler ble introdusert allerede når vi la inn verdien «YES» og «NO» i variablen \$LearnPWSH helt i starten av leksjonen. Ved å benytte anførselstegn blir det indikert for PowerShell at verdi er en serie bokstaver eller en *string*. Hvis en prøver å tildele variablene en verdi uten anførselstegn vil PowerShell returnerer en feil:

```

PS C:\code> $LearnPSH = MAYBE

MAYBE : The term 'MAYBE' is not recognized as the name of a cmdlet, function, script file, or
operable program. 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:14
+ $LearnPSH = MAYBE
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (MAYBE:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

PS C:\code>

```

Figur 13 - String uten " "

Uten anførselstegn tolker PowerShell *MAYBE* som en kommando og prøver å utføre det. Kommandoen *MAYBE* eksisterer ikke og PowerShell returnerer derfor en feilmelding som forteller akkurat det. Husk derfor å definere en *string* ved bruk av anførselstegn rundt verdien.

2.5.1 Kombinere string og variabler

Strenger er ikke begrenset til ord, de kan også være setninger. For eksempel kan en opprette variabelen *\$setning* med denne setningen: "I dag skal jeg lære meg PowerShell". En kan også slå sammen variabler med innhold og bruke de i samme setningen. For eksempel ønsker vi at *\$code* skal kunne inneholde andre typer script eller programmeringsspråk. Merk at en må definere variabelen *\$setning* på nytt etter at *\$code* er endret. Hvis ikke vil *\$setning* inneholde den gamle informasjonen.

```

PS C:\code> $code = "PowerShell"

PS C:\code> $setning = "I dag skal jeg lære meg $code"

PS C:\code> $code

PowerShell
PS C:\code> $setning

I dag skal jeg lære meg PowerShell
PS C:\code> $code = "Python"

PS C:\code> $setning = "I dag skal jeg lære meg $code"

PS C:\code> $setning

I dag skal jeg lære meg Python
PS C:\code>

```

Figur 14 - Variabel i variabel

2.5.2 Dobbelt og enkelt anførselstegn

Når en tildeler en variabel en enkel streng, kan en bruke enkle eller doble anførselstegn. Som en ser i figuren nedenfor spiller det ingen rolle hvilken type anførselstegn som brukes for å definere en enkel streng. Når en skriver inn *\$LearnPSH* alene i konsollen og trykker ENTER, interpolerer eller utvider PowerShell denne variabelen. Dette er fancy ord som betyr at PowerShell leser verdien i variabelen, eller åpner boksen for å se hva som er inni den. Når en bruker doble anførselstegn for å kalle en variabel skjer det samme, variabel interpolerer. Legg derimot merke til hva som skjer når en bruker enkelt anførselstegn. I terminalen sendes variabelen selv og ikke verdien. Enkelt anførselstegn forteller PowerShell at en mener nøyaktig hva en skriver, enten det er et ord som NO eller det som ser

ut som en variabel som heter `$LearnPWSH`. For PowerShell spiller det ingen rolle, siden det ikke ser forbi verdien i singelt anførselstegn. Dvs. når en bruker en variabel i enkle anførselstegn vil ikke PowerShell utvide variabelen og se dens verdi. Dette er grunnen til at en må bruke doble anførselstegn når en setter inn variabler i strengene. Enkelt forklart ville eksemplet ovenfor, hvor vi skrev «I dag skal jeg lære meg PowerShell», blitt vist som *I dag skal jeg lære meg \$code* hvis en hadde brukt enkle anførselstegn.

```
PS C:\code> $LearnPWSH = "YES"

PS C:\code> $LearnPWSH

YES
PS C:\code> $LearnPWSH = 'NO'

PS C:\code> $LearnPWSH

NO
PS C:\code> "$LearnPWSH"

NO
PS C:\code> '$LearnPWSH'

$LearnPWSH
PS C:\code> █
```

Figur 15 - Enkle og doble anførselstegn

2.6 Objects

Som vi allerede har nevnt, i PowerShell er alt et objekt. I tekniske termer er et objekt en individuell forekomst av en bestemt mal, kalt en *class*. En *class* spesifiserer hvilken type innhold et objekt vil inneholde. *Object class* bestemmer hvilke metoder, eller handlinger som kan utføres på akkurat dette objektet. Metodene er med andre ord alle tingene et objekt kan gjøre. For eksempel kan et listeobjekt ha en *sort()* -metode som vil sortere listen når den blir tatt i bruk. På samme måte bestemmer *Object class* dets *properties*(egenskaper), som vil si objektets variabler. En kan tenke på *properties* som all dataene om objektet. Når det gjelder listeobjektet, kan det hende en har en *length property* som lagrer antall elementer i listen. Noen ganger vil en *class* gi standardverdier for objektets *properties*, men for det oftest er dette verdier en vil gi til objektene en jobber med. Alt virker veldig abstrakt uten noe form for referanse å knytte det til. Et eksempel som kanskje hjelper på forståelsen, er: En bil. Bilen starter med en plan og en designfase. Denne planen definerer hvordan bilen skal se ut, hva slags motor den skal ha, hvilken type chassis den skal ha og så videre. Planen viser også hva bilen vil kunne gjøre når den er fullført. Nærmere bestemt å forflytte seg fremover, rygge, åpne og lukk dører med mer. En kan tenke på denne planen som bilen *class*. Hver bil er bygget med utgangspunkt i denne klassen, og alle bilens egenskaper og metoder blir lagt til den. En bil kan være sort, mens den samme modellbilen kan også være hvit. En annen bil kan ha en annen girkasse. Disse attributtene er egenskapene til et bestemt bilobjekt. Likevel vil hver av bilene kunne kjøre fremover, rygge, og har samme metode for å åpne og lukke dører. Disse handlingene er bilens *methods* (metoder) (Bertram, 2020).

I eksempel under er det et enkelt objekt som undersøkes. Det gjør det også litt enklere å vise de forskjellige fasettene til et PowerShell-objekt, `$LearnPWSH`. Som tidligere erfart, når en skriver inn `$LearnPWSH`, får en variabelens verdi. Siden det er objekter har den vanligvis mer informasjon enn bare verdien til variabelen. De har også *properties* (egenskaper). For å se egenskapene til et objekt, bruker en *Select-Object*-kommandoen og *property*-parameteren. Ved å benytte seg av `*` vil den liste ut alt sammen den finner. Som en kan se, har `$LearnPWSH` bare en enkelt egenskap, kalt *Length*. En kan referere direkte til egenskapen *Lengde* ved å bruke punktnotasjon, som vil si at en bruker navnet på objektet etterfulgt av et punktum og navnet på *property* en vil ha tilgang til. Å referere til objekter på denne måten vil bli en naturlig del av Powershellscriptingen når en får litt mer erfaring og tid på baken.

```

PS C:\code> $LearnPWSH = 'YES'

PS C:\code> $LearnPWSH

YES
PS C:\code> Select-Object -InputObject $LearnPWSH -Property *

Length
-----
      3

PS C:\code> $LearnPWSH.Length

3
PS C:\code>

```

Figur 16 - Finne et objekts egenskaper

2.7 Get-Member cmdlet

Ved å bruke `Select-Object` kunne en se at `$LearnPWSH`-strengen hadde en enkelt egenskap, men husk at objekter også noen ganger har, som vi nente ovenfor, metoder. For å se alle metodene og egenskapene som finnes for dette objektet, kan en bruke `Get-Member`. Denne cmdleten vil være til god nytte i mange sammenhenger fremover. Det er en enkel måte å raskt liste opp alle egenskapene og metodene til et bestemt objekt, samlet referert til som *Object Members* (objektets medlemmer).

```

PS C:\code> Get-Member -InputObject $LearnPWSH

    TypeName: System.String

Name      MemberType      Definition
----      -
Clone     Method          System.Object Clone(), System.Object ICloneable.Clone()
CompareTo Method          int CompareTo(System.Object value), int CompareTo(strin...
Contains  Method          bool Contains(string value), bool Contains(string value...
CopyTo    Method          void CopyTo(int sourceIndex, char[] destination, int de...
EndsWith  Method          bool EndsWith(string value), bool EndsWith(string value...
EnumerateRunes Method        System.Text.StringRuneEnumerator EnumerateRunes()
Equals    Method          bool Equals(System.Object obj), bool Equals(string valu...
GetEnumerator Method        System.CharEnumerator GetEnumerator(), System.Collectio...
GetHashCode Method        int GetHashCode(), int GetHashCode(System.StringCompari...
GetPinnableReference Method        System.Char&, System.Private.CoreLib, Version=5.0.0.0, ...

```

Figur 17 - Get-Member cmdlet

2.8 Hvordan bruke *Methods*

En kan referere til metoder ved bruk av punktnotasjon. I motsetning til en *propertie*, vil en *method* alltid ende i et sett av åpne og lukke parenteser. Den kan også ta en eller flere parametere. Hvis en for eksempel at fjerne et tegn i `$LearnPWSH` variabelen. En kan fjerne tegn fra en streng ved hjelp av `remove()` metoden.

```
PS C:\code> Get-Member -InputObject $LearnPWSH -Name Remove

    TypeName: System.String

Name      MemberType Definition
-----
Remove Method      string Remove(int startIndex, int count), string Remove(int startIndex)

PS C:\code> █
```

Figur 18 - Remove method

Som en kan se, er det to definisjoner, og det betyr at en kan bruke metoden på to måter: enten med `startIndex` og `count`-parameteren, eller med bare `startIndex`. Så for å fjerne det andre tegnet i `$LearnPWSH`, angir en stedet for tegnet en vil starte fjerning, som vi kaller indeksen. Indekser starter fra 0, så den første bokstaven har et startsted på 0, den andre en indeks på 1, og så videre. I tillegg til en indeks kan en oppgi antall tegn en vil fjerne ved å bruke komma for å skille parameterargumentene. Siden variabelen bare består av tre tegn kan vi her vise hvordan en fjerner bokstaven på posisjon 1 og antallet tegn er 1.

```
PS C:\code> $LearnPWSH
YES
PS C:\code> $LearnPWSH.Remove(1,1)
YS
PS C:\code> $LearnPWSH
YES
PS C:\code> █
```

Figur 19 - Bruk av `.Remove()`

Merk at metoden `Remove()` ikke endrer verdien til en variabel permanent. Hvis en ønsker å beholde denne endringen, må en tilordne utdataene fra `Remove()` -metoden til en variabel.

```
PS C:\code> $NewLearnPWSH = $LearnPWSH.Remove(1,1)
PS C:\code> $NewLearnPWSH
YS
PS C:\code> █
```

Figur 20 – Legger ny verdi i en variabel

Hvis en trenger å vite om en metode returnerer et objekt, sånn som `Remove()` gjør, eller om det endrer et eksisterende objekt, kan en sjekke metodens beskrivelsen. En kan se i figur 16 at `Remove()` har definisjon `string` foran `Remove`. Dette betyr at funksjonen returnerer en ny streng. Funksjoner med ordet `void` foran endrer vanligvis eksisterende objekter.

2.9 Data Structures

Data structures er en måte å organisere data på i PowerShell. Dette er representert av objekter lagret i variabler. De finnes i tre hovedtyper: *Array*, *ArrayLists* og *hashtables*.

2.10 Arrays

Tidligere beskrev vi en variabel som en boks med innhold. Men hvis en enkel variabel, for eksempel en *Float*-type er en enkel boks, så er et *Array* er en hel haug med bokser limt sammen. Det er en liste over elementer representert av en enkelt variabel. Ofte trenger en flere relaterte variabler, for eksempel et standard sett med svar på om en har

lyst til å lære seg PowerShell i dag. Dvs. at en heller bør lagre alle mulige svar til spørsmålet i en enkel datastruktur fremfor å lagre hvert svar som en egen *string* variabel, og deretter refererer til hver av disse individuelle variablene.

2.10.1 Definere et Array

Først definerer en variabelene kalt \$muligesvar og tildele den et *Array* som har tre mulig svar som strenger. For å gjøre dette bruker en krøllalfa-tegnet @ etterfulgt av de tre strengene separert med komma innenfor parentes.

```
PS C:\code> $muligesvar = @('YES','NO','MAYBE')
PS C:\code> $muligesvar
YES
NO
MAYBE
PS C:\code> █
```

Figur 21 - Definere et Array

@-tegnet etterfulgt av en åpneparentes og null eller flere elementer separert med komma gir beskjed til PowerShell om at en vil lage et *array*. Legg merke til at når en ønsker å se innholdet til \$muligesvar, viser PowerShell hvert av *array* elementene på en ny linje.

2.10.2 Lese fra et Array

For å få tilgang til et av element i arrayet, bruker en navnet på arrayet etterfulgt av firkantede parenteser, [], som inneholder indeksen til elementet en vil ha tilgang til. Som med string, begynner nummereringen i arrayet på 0. Dvs. det første elementet er indeks 0, det andre indeks 1 og så videre. I PowerShell bruker en -1 som indeks hvis en vil returnere det siste elementet i arrayet.

```
PS C:\code> $setning = "Vil du lære PowerShell i dag? "
PS C:\code> $setning
Vil du lære PowerShell i dag?
PS C:\code> $muligesvar = @('YES','NO','MAYBE')
PS C:\code> $setning + $muligesvar[0]
Vil du lære PowerShell i dag? YES
PS C:\code> █
```

Figur 22 - Lese fra Array

Hvis en forsøker å returnere et indeksnummer som ikke finnes i arrayet vil PowerShell gi deg en feilmelding. For å få tilgang til flere elementer i et array samtidig, kan en bruke rekkeviddeoperatøren, dvs. to punktum .. mellom to tall. Rekkeviddeoperatøren vil få PowerShell til å returnere de to tallene og hvert tall mellom dem: \$muligesvar[0..2]. En kan også hente opp de forskjellige elementene ved å skrive deres indeks og komma mellom dem: \$muligesvar[1,2]

```

PS C:\code> $muligesvar[0..2]

YES
NO
MAYBE
PS C:\code> $muligesvar[1..2]

NO
MAYBE
PS C:\code> $muligesvar[0,1,2]

YES
NO
MAYBE
PS C:\code> $muligesvar[0,2]

YES
MAYBE
PS C:\code> 

```

Figur 23 - Range operator

2.10.3 Editere element i et array

For å endre et element i et array, trenger en ikke å definere hele arrayet på nytt. En kan i stedet referere til et element med indeksen og bruk likhetstegnet til å tilordne en ny verdi. `$muligesvar[0] = 'Oh YES'` Det er alltid lurt å dobbeltsjekker at indeksnummeret er riktig ved å vise elementet til terminalen før en endrer et element. I dette tilfellet ville YES blitt erstattet med Oh YES.

```

PS C:\code> $muligesvar[0] = 'Oh YES'

PS C:\code> $muligesvar[0,1,2]

Oh YES
NO
MAYBE
PS C:\code> 

```

Figur 24 - Editere et element i et array

2.10.4 Legge til element i et array

En kan også legge til elementer i et array med plusstegnet. Dette vil legge til en verdi som før neste indeksnummer i arrayet. Merk at en kan gjøre dette på to måter: `$muligesvar = $muligesvar + 'Later'`, `$muligesvar += 'Tomorrow'`

Legg merke til at i det første eksemplet skriver en inn `$muligesvar` på begge sider av likhetstegnet. Dette er fordi en spør PowerShell om å interpolere `$muligesvar`-variabelen og deretter legge til et nytt element. `+` metoden fungerer, men det er raskere og mer lesbart å bruke plusstegnet sammen med likhetstegnet.

```

PS C:\code> $muligesvar = $muligesvar + 'Later'

PS C:\code> $muligesvar += 'Tomorrow'

PS C:\code> $muligesvar

Oh YES
NO
MAYBE
Later
Tomorrow
PS C:\code>

```

Figur 25 - Legg til element i et array

Som vi sa, += ber PowerShell om å legge dette elementet til det eksisterende arrayet. Med denne metoden unngår du å måtte skrive ut arraynavner to ganger, og det er mye mer vanlig enn å bruke full syntaks. En kan også legge til array i eksisterende array.

```

PS C:\code> $muligesvar += @('NOPE, NEXT WEEK', 'WHY HAVENT WE STARTED YET???)

PS C:\code> $muligesvar

Oh YES
NO
MAYBE
Later
Tomorrow
NOPE, NEXT WEEK
WHY HAVENT WE STARTED YET???
PS C:\code>

```

Figur 26 - Legg til array i array

Å legge til flere elementer samtidig kan spare en for mye tid, spesielt hvis en lager et array med et stor antall elementer. Merk at PowerShell behandler ethvert komma-separert verdisett som en matrise, og det gjør du ikke trenger eksplisitt å bruke @ eller parentes. Dessverre er det ingen tilsvarende enkel måte som += for å fjerne element fra et array. Fjerne elementer fra et array er mer komplisert enn en kanskje tror.

2.11 ArrayLists

Når en legger til et element i et array, lage en faktisk et nytt array fra det gamle (interpolerte) arrayet med det nye elementet. Det samme skjer når du fjerner et element fra et array, PowerShell ødelegger det gamle array og lager et nytt et. Dette er fordi array i PowerShell har en fast størrelse. Når en endrer dem, kan en ikke endre størrelse, så en må opprette et nytt array. For små array, sånn som de eksemplene vist her, vil en ikke legg merke til at dette skjer. Men når du begynner å jobbe med store array, med titalls eller hundretusener av elementer, vil en se et stort ytelsesbehov. Hvis en vet at en må fjerne eller legge til mange elementer i et array, bør en heller bruke en ArrayList. ArrayLists oppfører seg nesten identisk med det typiske PowerShell-arrayet, men med en elementær forskjell, de har ikke en fast størrelse. De kan justeres dynamisk for å bli lagt til eller fjernet elementer, noe som gir mye høyere ytelse når en arbeider med store datamengder. Å definere en ArrayList er akkurat som å definere et array, bortsett fra at en trenger å definere den som en ArrayList. Som med et array, når du lister ut en ArrayList, vises hvert element på en egen linje.


```
PS C:\code> $muligesvar = [System.Collections.ArrayList]@('YES','NO','MAYBE','LATER','TOMORROW',
',','NOPE, NEXT WEEK','WHY HAVENT WE STARTED YET??')

PS C:\code> $muligesvar

YES
NO
MAYBE
LATER
TOMORROW
NOPE, NEXT WEEK
WHY HAVENT WE STARTED YET??
PS C:\code> █
```

Figur 27 – ArrayList

For å legge til eller fjerne et element fra en ArrayList uten å ødelegge det, kan en bruke metoden `.Add()` og `.Remove()`.

```
PS C:\code> $muligesvar.Add('LET'S GO')

7
PS C:\code> █
```

Figur 28 - Legg til element i ArrayList

Legg merke til utgangen: tallet 7, som er indeksen til det nye elementet du la til. Vanligvis bruker en ikke dette nummeret til noe spesielt, dvs. en kan sende outputen fra `.Add()` til \$null-variabelen om en ønsker. Ved å gjøre det vil ikke outputen ikke blir sendt ut.

```
PS C:\code> $null = $muligesvar.Add('ALWAYS')

PS C:\code> █
```

Figur 29 - Send output til \$null -variabelen

En kan fjerne elementer på samme måte ved bruk av `.Remove()` -metoden. For eksempel hvis en vil fjerne verdi ALWAYS fra \$muligesvar, skriver en bare inn verdien i parentes av metoden. Legg merke til at en ikke trenger å vite indeksnummeret for å fjerne et element. En kan referere til elementet av dens faktiske verdi, som i dette tilfellet er ALWAYS. Hvis arrayet har flere elementer med samme verdi, vil PowerShell fjern elementet nærmest starten på ArrayList. Det er vanskelig å se ytelsesforskjellen med små eksempler som disse, men ArrayLists yter mye bedre på store datasett enn bare et array. Som i de fleste situasjoner må en analysere sin spesifikke situasjon for å avgjøre om det er mer fornuftig å bruke et array eller en ArrayList. Tommelfingerregelen er at jo større samling med elementer en jobber med, jo bedre er det å bruke ArrayList. Hvis en er arbeider med små matriser på færre enn 100 elementer eller så, vil en merke liten forskjell mellom et array og en ArrayList (Bertram, 2020).

2.12 Hashtables

Arrays og ArrayLists er gode når en trenger dataene tilknyttet bare en posisjon i en liste. Noen ganger derimot vil en ha en måte å sette dataen i sammenheng. For eksempel kan en ha en liste over brukernavn en vil matche med virkelige navn. I så fall kan en bruke en hashtable, som er en datastruktur som inneholder en liste over nøkkelverdi-par. I stedet for å bruke en numerisk indeks, oppretter PowerShell en nøkkel, og den returnerer verdien som er knyttet til den nøkkelen. Et eksempel vil være å bruke brukernavnet og deretter få det vil returnere brukerens reelle navn.

```
PS C:\code> $users = @{"olanord"="Ola Nordmann"
>> "karihans"="Kari Hansen"
>> "perjens"="Per Jensen"
>> }
```

```
PS C:\code> $users
```

Name	Value
----	-----
karihans	Kari Hansen
perjens	Per Jensen
olanord	Ola Nordmann

I

```
PS C:\code> █
```

Figur 30 – Hashtable

PowerShell lar en ikke definere en hashtable med duplikate nøkler. Hver nøkkel må være unik og peke på en enkeltverdi, som kan være en array eller til og med en annen hashtable. For å få tilgang til en bestemt verdi i en hashtable, bruker en bare nøkkelen til den. Det er to måter en kan gjøre dette på. For eksempel ønsker en å finne ut det virkelige navnet på brukerens *perjens*. En kan da bruke en av de to tilnærminger:

```
PS C:\code> $users['perjens']
```

```
Per Jensen
```

```
PS C:\code> $users.perjens
```

I

```
Per Jensen
```

```
PS C:\code> █
```

Figur 31 - Hent informasjon fra hashtable

De to alternativene har subtile forskjeller, men foreløpig kan en velge metode en selv foretrekker. Den andre kommandoen bruker en *property*: `$users.perjens`. PowerShell vil legge til hver nøkkel i objektets egenskaper. Hvis en ønsker å se alle nøklene og verdiene en hashtable har, kan gjøre det ved å skrive `$users.Keys` og `$users.Values`. Hvis en ønsker å se alle *properties* (egenskapene) til en hashtable kan en kjøre følgende kommando: *Select-Object \$users -Property ** `$users` er i dette tilfellet objektet jeg ønsker å hente ut all informasjonen fra.

```
PS C:\code> $users.Keys
```

```
karihans
```

```
perjens
```

```
olanord
```

```
PS C:\code> $users.Values
```

```
Kari Hansen
```

```
Per Jensen
```

```
Ola Nordmann
```

```
PS C:\code> █
```

Figur 32 - Hente ut Keys og Values

2.12.1 Legg til / editere innhold i hashtable

For å legge til et element i en hashtable kan en bruke metoden `.Add()` eller opprette en ny indeks ved å bruke kvadrat parenteser og likhetstegn. I likhet med `ArrayLists` har hashtables en `.Remove()` -metode. Når en har fjernet en bruker, kan en sjekke hashtabellen for å verifisere at brukeren faktisk er borte. Husk at en kan bruke `.Keys` -egenskapen til å finne frem til hvilket som helst nøkkelnavn.

```
PS C:\code> $users.Add('nilsmons', 'Nils Monsen')

PS C:\code> $users['markjohn'] = 'Markus Johnsen'

PS C:\code> $users
```

Name	Value
----	-----
karihans	Kari Hansen
markjohn	Markus Johnsen
perjens	Per Jensen
olanord	Ola Nordmann
nilsmons	Nils Monsen

```
PS C:\code> 
```

Figur 33 - Legg til innhold i hashtable

Hvis en ønsker å endre en av verdiene i en hashtable er det alltid lurt å sjekke nøkkelverdiparet du ønsker editere faktisk eksisterer. For å sjekke om en nøkkel allerede finnes i en hashtable, kan en bruke metoden `ContainsKey()`. Hvis hashtabellen inneholder nøkkelen vil den returnere `True`, hvis ikke vil den returnere `False`. Når en har bekreftet at nøkkelen er i hashtabellen kan en endre verdien ved å bruke likhetstegn.

```
PS C:\code> $users.ContainsKey('karihans')

True
PS C:\code> $users.ContainsKey('karijens')

False
PS C:\code> $users['karihans'] = 'Karina Hansen'

PS C:\code> $users.karihans

Karina Hansen
PS C:\code> 
```

Figur 34 - Endre innhold i hashtable

2.13 Custom objects

Så langt i denne leksjonen har vi laget og brukt innebygde objekter i PowerShell. Mesteparten av tiden vil en benytte seg av disse typene og spare seg for arbeidet med å lage sine egne. Noen ganger derimot trenger en å lage et tilpasset objekt med egenskaper og metoder en selv definerer. Kommandoen en benytter er `New-Object` og benyttes på følgende måte: `$CustomObjectWithASuitableName = New-Object -TypeName PSCustomObject`. For å definere en hashtable med `key` `propertynames` og `values` på følgende måte: `$CustomObjectWithASuitableName = [PSCustomObject]@{OSBuild = 'x'; OSVersion = 'y'}`. Legg merke til semikolon (;) for å skille key- og

valuedefinisjonene. Når en har et tilpasset objekt, kan en bruke det som alle andre objekter. Vi kan benytte *Get-Member cmdlet* for å sjekke at det er et *PSCustomObject*-type.

```
PS C:\code> $CustomObjectWithASuitableName = New-Object -TypeName PSCustomObject
PS C:\code> $CustomObjectWithASuitableName = [PSCustomObject]@{OSBuild = 'x'; OSVersion = 'y'}
PS C:\code> Get-Member -InputObject $CustomObjectWithASuitableName

TypeName: System.Management.Automation.PSCustomObject

Name      MemberType Definition
----      -
Equals     Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType    Method      type GetType()
ToString   Method      string ToString()
OSBuild    NoteProperty string OSBuild=x
OSVersion  NoteProperty string OSVersion=y

PS C:\code> $
```

Figur 35 - Custom Object

Som en kan se har objektet allerede noen metoder, for eksempel en som returnerer objektets type sammen med egenskapene som ble definerte ved opprettelse av objektet. Som tidligere vist, kan en her også få tilgang til disse egenskapene ved å bruke punktnotasjon: *PS> \$CustomObjectWithASuitableName.OSBuild* og *.OSVersion*.

```
PS C:\code> $CustomObjectWithASuitableName.OSBuild
x
PS C:\code> $CustomObjectWithASuitableName.OSVersion
y
PS C:\code> 
```

Figur 36 - Punktnotasjon

3 Referanse

Bertram, A. R. (2020). *PowerShell for sysadmins*. No Starch Press.

Lee, T. (2020). *PowerShell 7 for IT pros: A guide to using PowerShell 7 to manage Windows systems*.

PowerShell 7.2 Preview 2 release. (2020, desember 16). PowerShell.
<https://devblogs.microsoft.com/powershell/powershell-7-2-preview-2-release/>

SteveL-MSFT. (u.å.). *about_Automatic_Variables—PowerShell*. Hentet 20. desember 2020, fra
https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables