SECTION 7

ARRAYS + FOR LOOPS

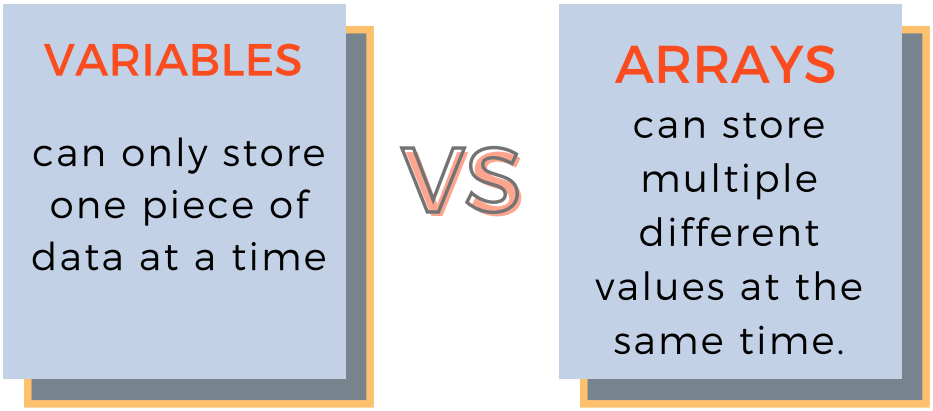
SECTION CHEAT SHEET

WORKING WITH INDEXED ARRAYS:

Variables can only store one piece of data at a time.

Arrays, however, can store multiple different values at the same time.

This makes arrays great tools for storing a lot of data in one place for batch processing.



Syntax for creating a literal indexed array:

```
array=(element1 element2 element3 ... elementN)
```

Example:

```
numbers=(1 2 3 4)
```

Note: Bash uses spaces to separate values within an array, not commas

Each entry of an array is called an element and each element has its own index. Indexes start from 0 and count up.

EXPANDING AN ARRAY

Expansions help us to draw data from an array without modifying the underlying data. Some of these will be familiar to you from our discussion of "parameter expansion tricks" from section 2

Assume we have the array: array=(1 2 3 4 5)

Expansion	Description	Result
${array}	Gives the first element of an array	1
${array[@]}	Expands to all the elements of an array	1 2 3 4 5
${!array[@]}	Expands to all the index numbers of all the elements of an array	0 1 2 3 4
${array[@]:offset}** **E.g.** **${array[@]:2}	Starts at the index specified by offset rather than at index 0, and then continue until the end of array[@]. So, in this example, we would start at index 2, which is the number 3.	3 4 5
${array[@]: -2}	We can also provide negative offsets. In this example, we will start two elements from the end, which is the number 4. **Note:** You must have a space after the ":" and before the "-".	4 5
${array[@]:offset:length}** **E.g.** **${array[@]:2:2}	Skips the first offset elements, and continues until the whole length of the array is length So, in this example, we would skip the first 2 elements, and continue until we had a total of 2 elements.	3 4

MODIFYING AN ARRAY

Once again, assume we have the array: array=(1 2 3 4 5)

Operation	Description	Result
array+=(6)	Appends 6 to the end of array	array becomes (1 2 3 4 5 6)
array+=(a b c)	Appends (a b c) to the end of array	array becomes (1 2 3 4 5 a b c)
unset array[2]	Removes the specified element from the array. In this example the element at index 2 will be removed Note: Index numbers do not update automatically, so this will create a gap in the indexes!	array becomes (1 2 4 5 6)
array[0]=100	Changes the value of a specific element of the array. In this example the element with index 0 will become 100	array becomes (100 2 3 4 5 6)

THE READARRAY COMMAND:

The readarray command converts what it reads on its standard input stream into an array.

CREATING AN ARRAY FROM A FILE:

```
readarray -t arrayname < file
```

Example:

```
readarray -t days < days.txt
```

CREATING AN ARRAY FROM THE OUTPUT OF A COMMAND:

This is achieved using a technique known as “process substitution”. Process substitution simply allows us to treat the output of a command (or commands) as a file.

Syntax for process substitution

```
<(command) # You can run one command...
<(command1 | command2 | ... | commandN) #... Or an entire pipeline
```

We can then simple read the output of the command into the readarray command as if it was a file:

```
readarray -t arrayname < <(command)
```

Example:

```
readarray -t files < <(ls ~/Documents)
```

A NOTE ABOUT NEWLINE CHARACTERS

By default, when the readarray command reads a file (or process substitution), it will save each entire line as a new array element, including the invisible newline character at the end of the line.

Storing new lines characters can cause issues when it comes to formatting, so it is best to remove them.

Therefore, it is suggested that you always use the readarray command's -t option unless you have a strong reason otherwise. This will prevent any newline characters from being stored in your array values, and help prevent issues down the road.

ITERATING OVER ARRAYS WITH FOR LOOPS:

A **for** loop iterates over a list of words or elements and performs a set of commands ***for*** each element within that list, hence its name.

for loops are an amazingly powerful tool when coupled with arrays, as they allow us to create "assembly lines" that we can use for batch processing.

for loop syntax (without array):

```
for <variable> in value1 value2 value3; do
   commands...
done
```

Example Script:

```
#!/bin/bash

for element in first second third; do
        echo "This is $element"
done
```

forloop syntax (with array):

```
for element in "${array[@]}"; do
   commands...
done
```

Example Script:

```
#!/bin/bash

readarray -t files < file_list.txt

for file in "${files[@]}"; do
    if [ -f "$file" ]; then
       echo "$file already exists"
    else
       touch "$file"
       echo "$file was created"
    fi
done
```