Shell Scripting II: Advanced Scripts

Session 7

Shell Scripting II: Advanced Scripts

Session 7

Objectives:

- Handle command-line arguments using $0..$9, $#, $@, and shift
- Use functions in shell scripts
- Implement control structures (case, if/elif/else, loops) in real examples
- Improve reusability and modularity with functions
- Use exit codes and simple debugging tools
- Write scripts that behave like command-line tools

Shell Positional Parameters

There are a set of variables which are set for you already, and most of these cannot have values assigned to them.

- $0: Name of script
- $1..$9: Arguments
- $#: Number of arguments
- $@: All arguments

Shell Positional Parameters

myvar.sh

```
#!/bin/sh
echo "I was called with $# parameters"
echo "My name is $0"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

Shell Positional Parameters

myvar.sh

```
#!/bin/sh
echo "I was called with $# parameters"
echo "My name is $0"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

./myvar.sh

```
I was called with 0 parameters
My name is ./myvar.sh
My first parameter is
My second parameter is
All parameters are
```

Shell Positional Parameters

myvar.sh

```
#!/bin/sh
echo "I was called with $# parameters"
echo "My name is $0"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

./myvar.sh

```
I was called with 0 parameters
My name is ./myvar.sh
My first parameter is
My second parameter is
All parameters are
```

./var3.sh hello world earth

Shell Positional Parameters

myvar.sh

```
#!/bin/sh
echo "I was called with $# parameters"
echo "My name is $0"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

./myvar.sh

```
I was called with 0 parameters
My name is ./myvar.sh
My first parameter is
My second parameter is
All parameters are
```

./var3.sh hello world earth

```
I was called with 3 parameters
My name is ./var3.sh
My first parameter is hello
My second parameter is world
All parameters are hello world earth
```

While

The **while** loop in shell is especially useful when you don’t know in advance how many times you'll need to repeat something — for example, reading a file line by line or waiting for a process to finish

- while evaluates a test condition
- while [condition]: the loop continues while the condition is true
- do … done: everything between do and done is the loop body
- Common use cases: reading lines, looping over arguments, timers

While

```
#!/bin/bash
count=1
while [ $count -le 5 ]; do
  echo "Count is $count"
  count=$((count + 1))
done
```

Output:

While

```
#!/bin/bash
count=1
while [ $count -le 5 ]; do
  echo "Count is $count"
  count=$((count + 1))
done
```

Output:

```
Count is 1
Count is 2
Count is 3
Count is 4
Count is 5
```

Shift command

The **shift** command in shell scripting discards the first positional parameter ($1) and shifts all others one position to the left.

- shift — by default shifts by 1
- shift N — shifts by N positions
- Affects $1, $2, ..., $n
- $# decreases with each shift (the number of arguments)

Shift command

The **shift** command in shell scripting discards the first positional parameter ($1) and shifts all others one position to the left.

- shift — by default shifts by 1
- shift N — shifts by N positions
- Affects $1, $2, ..., $n
- $# decreases with each shift (the number of arguments)

When to use shift

- When you want to process all arguments one-by-one
- Inside loops to consume parameters cleanly
- Especially useful when argument count is variable

Shift command

```
#!/bin/bash
while [ "$#" -gt 0 ]; do
  echo "Argument: $1"
  shift
done
```

./shift_example.sh hello world 42 earth

Shift command

```
#!/bin/bash
while [ "$#" -gt 0 ]; do
  echo "Argument: $1"
  shift
done
```

./shift_example.sh hello world 42 earth

- Prints each argument
- Removes it from the list ($1 becomes $2, etc.)
- Stops when no arguments remain

```
Argument: hello
Argument: world
Argument: 42
Argument: earth
```

Functions

A **function** in a shell script is a reusable block of code that can be called by name, helping organize and avoid repetition.

Structure:

```
function_name() {
  # commands
}
```

Or

```
function function_name {
  # commands
}
```

Useful:

- When you need to repeat the same logic multiple times
- To make scripts easier to read and maintain
- To logically separate different parts of a script

Functions

Example:

```
hello() {
  echo "Hello!"
}
hello
```

Output: Hello!

With argument:

```
hello_person() {
  echo "Hello, $1!"
}
hello_person "Thomas“
```

Output: Hello, Thomas

Using exit, return, and $?

Use **exit, return, and $?** to handle success or failure in scripts and functions, and to debug by checking what happened last.

- **exit <code>** — Terminates the script with a status code (usually 0 = success, non-zero = error)
- **return <code>** — return is like exit, but it's used inside functions — it tells us whether the function succeeded or failed.
- **$?** — Stores the exit status of the last command run

Using exit, return, and $?

Exit ./checkfile.sh

```
#!/bin/sh
if [ ! -f myfile.txt ]; then
  echo "File not found!"
  exit 1
fi
```

Output:

File not found

$?

After the output run
echo $?

Output: 1

return

```
#!/bin/bash

check_number() {
  if [ "$1" -gt 10 ]; then
    return 0    # success
  else
    return 1    # failure
  fi
}
check_number 15
echo "Return code was: $?“
```

Output: Return code was 0

```
check_number 5
echo "Return code was: $?“
```

Output: Return code was 1

Debugging

Use **exit, return, and $?** to handle success or failure in scripts and functions, and to debug by checking what happened last.

Useful debugging tools:

- **set -x** — Print each command before it runs (trace mode)
- **set +x** — Turn off trace mode
- **echo** — Print variable values manually
- **$?** — Check the result of the previous command
- **bash -n script.sh** — Check for syntax errors without running
- **trap** — Handle unexpected errors or cleanup on script exit

Debugging

Use **exit, return, and $?** to handle success or failure in scripts and functions, and to debug by checking what happened last.

Useful debugging tools:

- **set -x** — Print each command before it runs (trace mode)
- **set +x** — Turn off trace mode
- **echo** — Print variable values manually
- **$?** — Check the result of the previous command
- **bash -n script.sh** — Check for syntax errors without running
- **trap** — Handle unexpected errors or cleanup on script exit. It lets us clean up resources, log actions, or gracefully handle interruptions when a script exits or gets interrupted.

Debugging

Example

```
set -x
NAME="Thomas"
echo "Hello, $NAME"
set +x
```

Debugging

Example

```
set -x
NAME="Thomas"
echo "Hello, $NAME"
set +x
```

Outputs: Bash will print every command it executes, along with its expanded arguments

```
+ NAME=Thomas
+ echo 'Hello, Thomas'
Hello, Thomas
+ set +x
```

Labs

Lab 1: Read lines from a file (10 min)

- Script readline.sh:
- Create a file with random text, called it “myfile.txt”
- Read line by line

Labs

Lab 1: Read lines from a file (10 min)

```
#!/bin/bash

# Step 1: Create a file with random text
cat > myfile.txt << EOF
apple banana cherry
dog elephant frog
grape house igloo
jackal kite lemon
monkey night owl
EOF

# Step 2: Read the file line by line
echo "Reading lines from myfile.txt:"
while read line; do
  echo "Line: $line"
done < myfile.txt
```

Labs

Lab 2: Print Positional Parameters

Script: args.sh

Instructions:

- Print the following:
  - $# (number of arguments)
  - $0 (script name)
  - $1, $2 (first and second arguments)
  - $@ (all arguments)
- Run
  - ./args.sh A B C
  - ./args.sh "hello" "world"

Labs

Lab 2: Print Positional Parameters

```
#!/bin/bash
echo "Script name: $0“
echo "Number of arguments: $#“
echo "First argument: $1“
echo "Second argument: $2“
echo "All arguments: $@"
```

Labs

Lab 3: Creating Users with a Function (20 min)

- Create script adduser.sh
- Define a function add_a_user
- Use positional parameters to:
  - Assign $1 to USER
  - Assign $2 to PASSWORD
  - Use shift to remove the first two arguments
  - Assign the rest ($@) to COMMENTS
- Simulate creating users with echo commands
- Call the function 2 times with different arguments

Labs

Lab 3: Creating Users with a Function (20 min)

```
#!/bin/sh
add_a_user() {
  USER=$1
  PASSWORD=$2
  shift; shift
  COMMENTS=$@
  echo "Adding user $USER ..."
  echo useradd -c "$COMMENTS" $USER
  echo passwd $USER $PASSWORD
  echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}

# Main script starts here
echo "Start of script..."
add_a_user thomas tompwd Thomas deJaeger the Teacher
add_a_user student studentpwd Student Good students
echo "End of script..."
```

Labs

Lab 5: Greeting Function with Validation

- Create script greet.sh
  - Define a function greet_user that takes a name
  - If name is missing, show an error
  - Otherwise, print: "Hello, NAME. Today is DATE.“

Ex:
./greet.sh Thomas
Result: Hello, Thomas. Today is Fri May 2 12:00:00 CEST 2025

Hint: You need to check if the argument exist or not, you can use:

- -z STRING is a test operator used inside […] to check whether a string is empty.

Labs

Lab 5: Greeting Function with Validation

```
#!/bin/bash
greet_user() {
  if [ -z "$1" ]; then
    echo "Error: No name provided"
    return 1
  fi
  echo "Hello, $1. Today is $(date)."
}

greet_user "$1"
```