GlobalLogic

A Hitachi Group Company

EDUCATION

Smart Start: Linux/Networking Bash Scripting

Sergii Kudriavtsev

Agenda

1. Shell scripting - general information
2. Input arguments, return value
3. Conditional statements (if, then else, case)
4. Loops (while, until, for)
5. Functions
6. Types of variables
7. Variables
8. Evaluations
9. String operations in bash
10. Bash command substitution
11. Basic Bash commands (echo, read, etc.)

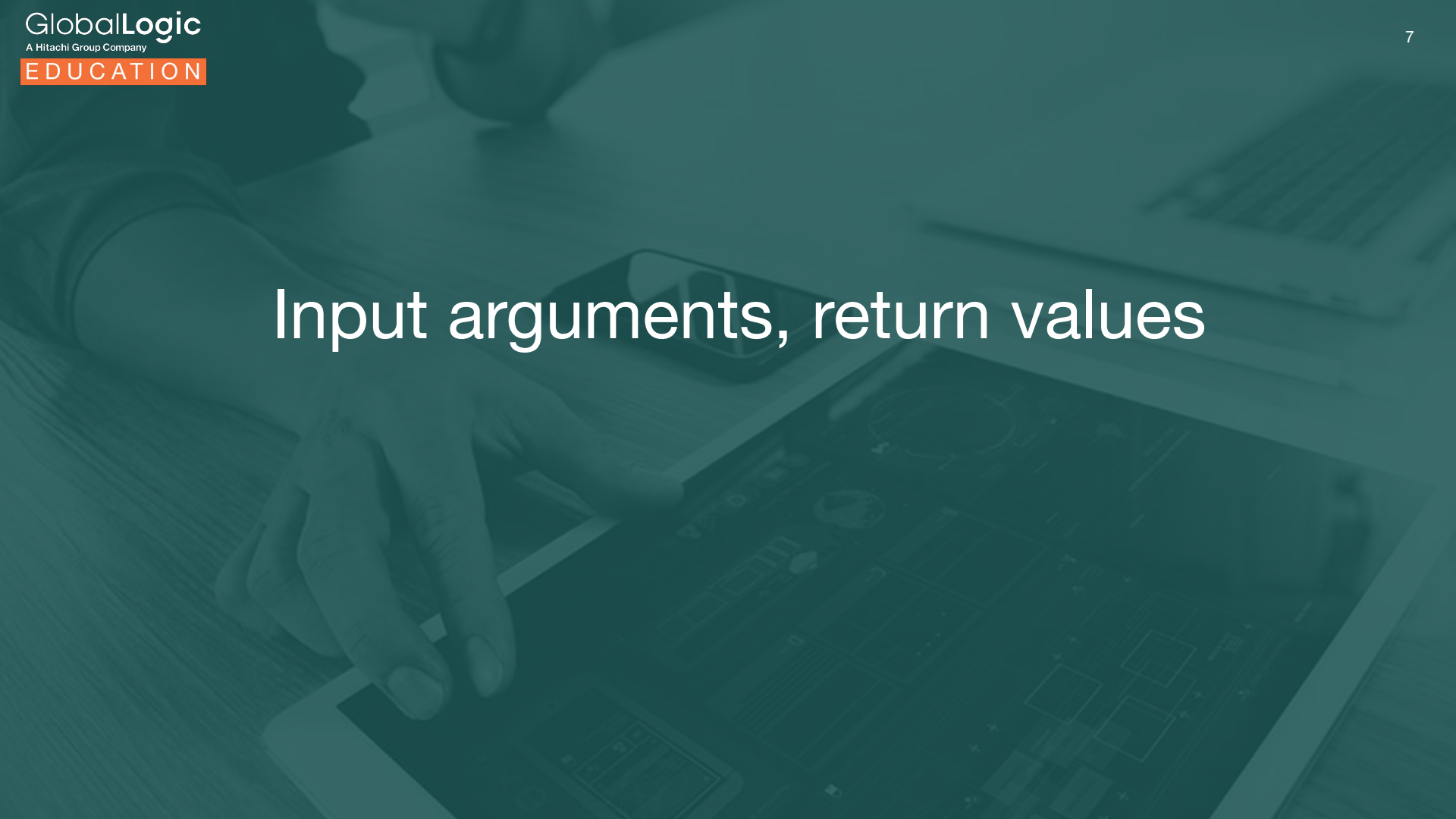
Bash scripting - general information

- Shell scripting- general information
  - Types of Shell
    - sh (Bourne Shell)
      - ksh (Korn Shell)
        - **bash (Bourne Again Shell)**
          - zsh (Z-Shell)
    - csh (C-Shell)
      - tcsh
    - dash

- Shell scripting- general information
  - Running scripts
    - executable bit
      - `$ chmod +x script.sh`
      - `$ ./script.sh`
    - mount point options
      - noexec
        - `$ mount | grep noexec`
    - current `SHELL` setting
      - `$ echo $SHELL`
    - direct Shell invocation
      - `$ bash ./script.bash`
    - hard-coded Shell interpreter. The first line of the script:
      - `#!/bin/bash`
    - running != sourcing
      - `$ ./script.sh`
      - `$ . ./script.sh`

- Shell scripting- general information
  - Debugging scripts
    - inside script
      - `set -x`
      - `set +x`
      - `set -v`
      -
      - `#!/bin/bash -x`
      - `#!/bin/bash -v`
  - Comments
    - #

Input arguments, return values



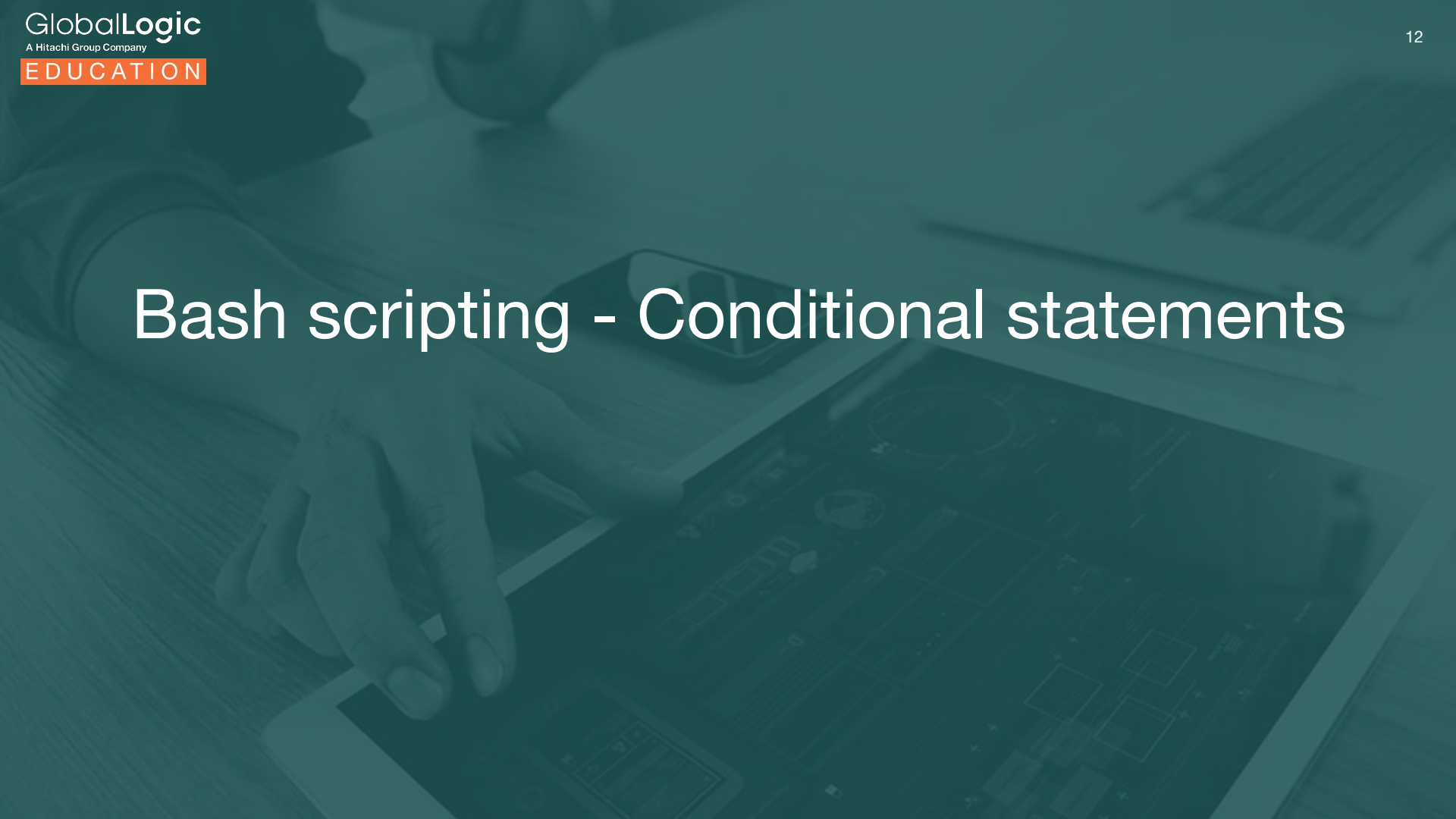
- Bash scripting - input arguments, return value
  - Field separator
    - $IFS (space, tab, new line) - determines how Bash recognizes word boundaries while splitting a sequence of character strings.
      - `$ echo "$IFS" | cat -et`
      - `$ export IFS=:`
      - `$ ls -ld $PATH`
  - Input variables
    - Exported
      - `$ export VAR1=value1`
      - `$ declare -x VAR2=value2`
      - `$ /home/user/script.sh`
    - Specified in command line
      - `$ VAR1=value1 VAR2=value2 /home/user/script.sh arg1 arg2`

- Bash scripting - input arguments, return value
  - Input arguments
    - numbering: 1, 2, …
    - all arguments: *
    - number of all arguments: #
    - parsing by position:
      - ```
        $ /home/user/script.sh arg1 arg2 arg3
        # script.sh:
        echo "argument1: $1"
        echo "argument2: $2"
        echo "argument3: $3"
        ```
        ```

- parsing by name:
  - ```
    $ /home/user/script.sh -b -a argA -c argC argD --longopt -l
    # script.sh
    declare -i I=0
    while getopts a:bc: OPT
    do
            case "$OPT" in
                    a)
                            echo "flag: -a, argument: $OPTARG"
                            I=$(( $I + 1 ))
                            if [[ -n "$OPTARG" ]]
                            then
                                    I=$(( $I + 1 ))
                            fi
                            ;;
                    b)
                            echo "flag: -b, no arguments"
                            I=$(( $I + 1 ))
                            ;;
                    c)
                            echo "flag: -c, argument: $OPTARG"
                            I=$(( $I + 2 ))
                            ;;
                    '?')
                            echo "ERROR";;
            esac
    done
    shift $I
    ```
    ```

- Bash scripting - input arguments, return value
  - Exit code
    - 0 - success, not 0 - failure:
    - `$ exit 0`
    - `$ echo $?`
    - signaled process returns value: 128 + signal number

Bash scripting - Conditional statements



- Bash scripting - conditions and sequences
  - test command (POSIX compliance)
    - $ man test
    - $ test <arguments>
      - $ [3 -eq 3] && echo “Numbers are equal”

        Numbers are equal
      - $ test 3 -eq 3 && echo “Numbers are equal”

        Numbers are equal

        The only difference between **[** and **test** is that we must use the closing **]** for surrounding the comparison.
    - $ echo $?

- Bash scripting - conditions and sequences

- Condition possibilities
  - **File-based conditions:**
    - `[ -a existingfile ]` #True if file 'existingfile' exists
    - `[ -b blockspecialfile ]` #file 'blockspecialfile' exists and is block special.
    - `[ -c characterspecialfile ]` #file 'characterspecialfile' exists and is character special.
    - `[ -d directory ]` #file 'directory' exists and is a directory.
    - `[ -f regular file ]`#file 'directory' exists and is a directory.
    - `[ -h symboliclink ]` #file 'symboliclink' exists and is a symbolic link.
    - `[ -S socket ]` #file 'socket' exists and is a socket.
    - `[ -w writeablefile ]` #file 'writeablefile' exists and is writeable to the script.
    - `[ -x executablefile ]` # file 'executablefile' exists and is executable.

- Bash scripting - conditions and sequences

- Condition possibilities
  - **String-based conditions:**
    - **[STRING1 == STRING2]** #True if STRING1 is equal to STRING2.
    - **[STRING1 != STRING2]** # True if STRING1 is not equal to STRING2.
    - **[STRING1 > STRING2]** # True if STRING1 sorts after STRING2 in the current locale (lexographically).
    - **[STRING1 < STRING2]** # STRING1 sorts before STRING2 in the current locale (lexographically).
    - **[-n NONEMPTYSTRING]** # True if NONEMPTYSTRING has a length of more than zero.
    - **[-z EMPTYSTRING]** # True if EMPTYSTRING is an empty string.
    - **Double-bracket syntax only:[[STRING1 =~ REGEXPATTERN]]** # STRING1 matches REGEXPATTERN.

- Bash scripting - conditions and sequences

- Condition possibilities
  - **Arithmetic (number-based) conditions:**
    - `[ NUM1 -eq NUM2 ] # True if NUM1 is Equal to NUM2..`
    - `[ NUM1 -ne NUM2 ] # True if NUM1 is Not Equal to NUM2.`
    - `[ NUM1 -gt NUM2 ] # True if NUM1 is Greater Than NUM2.`
    - `[ NUM1 -ge NUM2 ] # True if NUM1 is Greater than or Equal to NUM2.`
    - `[ NUM1 -lt NUM2 ] # True if NUM1 is Less Than NUM2.`
    - `[ NUM1 -le NUM2 ] # True if NUM1 is Less than or Equal to NUM2.`

- Bash scripting - conditions and sequences

- Condition possibilities

  - **Miscellaneous conditions:**
    - **[-o shelloption]** # True if shell option 'shelloption' is enabled.
      - $ set -o

- Bash scripting - conditions and sequences

- Condition possibilities
  - **Double-parenthesis syntax conditions:**
  - **These conditions only accept integer numbers. Strings will be converted to integer numbers, if possible.**

```
(( NUM1 == NUM2 )) # True if NUM1 is equal to NUM2.
(( NUM1 != NUM2 )) # True if NUM1 is not equal to NUM2.
(( NUM1 > NUM2 )) # True if NUM1 is greater than NUM2.
(( NUM1 >= NUM2 )) # True if NUM1 is greater than or equal to NUM2.
(( NUM1 < NUM2 )) # True if NUM1 is less than NUM2.
(( NUM1 <= NUM2 )) # True if NUM1 is less than or equal to NUM2.
```

- Bash scripting - (single brackets ([ ]) and double brackets ([[ ]])) (#aren't POSIX compliant)
- Differences between [and [[:
  - [3 -eq 3] or [[3 -eq 3]] - Same result:
    - **$ [3 -eq 3] && echo "Numbers are equal"**
      Numbers are equal
    - **$ test 3 -eq 3 && echo "Numbers are equal"**
      Numbers are equal
  - [is a shell builtin and [[is a shell keyword # $type [[/ $type [.
  - The double brackets, [[ ]], were introduced in the Korn Shell as an enhancement that makes it easier to use in tests in shell scripts. [[is just a convenient alternative to single brackets.
  - **Other differences:** Comparison Operators, Boolean Operators, Grouping Expressions, Pattern Matching , Regular Expressions, Word Splitting

- Bash scripting - conditions and sequences
  - condition evaluation: [condition] and [[condition]]
    - /bin/sh: `[ condition ]`
      - AND:
        - `[ condition1 -a condition2 ]`
          - `[ "$VAR1" = A -a "$VAR2" = B ]`
      - OR:
        - `[ condition1 -o condition2 ]`
          - `[ "$VAR1" = A -o "$VAR2" = B ]`
      - NOT:
        - `[ ! condition ]`
          - `[ ! "$VAR" = A ]`
      - grouping:
        - `[ \( a = a -a b = b \) -a \( c = X -o d = e \) ]`

- Bash scripting - conditions and sequences
  - /bin/ksh and later: `[[ condition ]]`
    - AND:
      - `[[ condition1 && condition2 ]]`
        - `[[ "$VAR1" = A && "$VAR2" = B ]]`
    - OR:
      - `[[ condition1 || condition2 ]]`
        - `[[ "$VAR1" = A || "$VAR2" = B ]]`
    - NOT:
      - `[[ ! condition ]]`
        - `[[ ! "$VAR" = A ]]`
    - grouping:
      - `[[ ( a = a && b = b ) && ( c = X || d = e ) ]]`

- Bash scripting - conditions and sequences
  - if - then - else - fi
    - ```
      if [[ condition ]]
      then
            echo "true"
            echo ""
      fi
      ```
    - ```
      if [[ condition ]]
      then
            echo "true"
            echo ""
      else
            echo "false"
      fi
      ```
    - ```
      if [ condition ]; then echo "true"; echo ""; else echo "false"; fi
      ```
      ```

- Bash scripting - conditions and sequences
  - Conditional sequence: &&, ||
    - `[ condition ] && echo "true" || echo "false"`
    - `[[ condition ]] && echo "true" || echo "false"`
    - `rm file.txt && echo "file removed" || echo "file not removed"`
  - Unconditional sequence: ;, \n (end of line)
    - `command1; command2; command3`
      - ```
        command1
        command2
        command3
        ```
        ```

- Bash scripting - conditions and sequences
  - Grouping: {}, ()
    - {} - commands run in current process
      - `{ command1; command2; command3; }`
      - `VAR=value; { unset VAR; }; echo $VAR`
        nothing is printed
      - `{ echo "Cleaning /tmp dir"; rm -r /tmp/*; } > $HOME/log.txt 2>&1`
      - `{ echo "Cleaning /tmp dir"; rm -r /tmp/* && exit 0 || exit 1; } > $HOME/log.txt 2>&1`
    - () - commands run in child process
      - `( command1; command2; command3 )`
      - `VAR=value; ( unset VAR ); echo $VAR`
        `value`
      - `( echo "Cleaning /tmp dir"; rm -rf /tmp/* ) > $HOME/log.txt 2>&1`
      - `( echo "Cleaning /tmp dir"; rm -rf /tmp/* && exit 0 || exit 1 ) > $HOME/log.txt 2>&1`

- Bash scripting - conditions and sequences
  - Grouping: {}, ()
    - $ echo a1; echo a2 | grep -o a
      a1
      a
    - $ { echo a1; echo a2 ;} | grep -o a
      a
      a
    - $ (echo a1; echo a2)| grep -o a
      a
      a
    - $ a=1; { a=2 ; echo $a ; } ; echo $a
      2 2
    - $ a=1; (a=2 ; echo $a ;) ; echo $a
      2 1

NOTE: $$ stays the same in the subshell because bash does not need to be reinitialized. **$BASHPID** changes, though.

- Bash scripting - (single brackets ([ ]) and double brackets ([[ ]])) (#aren't POSIX compliant)
  - [is a shell builtin and [[is a shell keyword # $type [[/ $type [. The double brackets, [[ ]], were introduced in the Korn Shell as an enhancement that makes it easier to use in tests in shell scripts.
    - Comparison Operators
      - $ [[1 < 2]] && echo "1 is less than 2"
        1 is less than 2
      - $ [1 < 2] && echo "1 is less than 2"
        bash: 2: No such file or directory
      - $ [1 \< 2] && echo "1 is less than 2"
        1 is less than 2

- Bash scripting - (single brackets ([ ]) and double brackets ([[ ]])) (#aren't POSIX compliant)
    - Boolean Operators (AND = && vs -a, OR = || vs -o)
        - $ [[3 -eq 3 && 4 -eq 4]] && echo "Numbers are equal"
          Numbers are equal
        - $ [3 -eq 3 -a 4 -eq 4] && echo "Numbers are equal"
          Numbers are equal

- Bash scripting - (single brackets ([ ]) and double brackets ([[ ]])) (#aren't POSIX compliant)
  - Grouping Expressions (parentheses usage)
    - $ [[3 -eq 3 && (2 -eq 2 && 1 -eq 1)]] && echo "Parentheses can be used"
      Parentheses can be used
    - $ [3 -eq 3 -a (2 -eq 2 -a 1 -eq 1)] && echo "Parentheses can be used"
      bash: syntax error near unexpected token '('
    - $ [3 -eq 3 -a \(2 -eq 2 -a 1 -eq 1 \)] && echo "Parentheses can be used"
      Parentheses can be used
    -

- Bash scripting - (single brackets ([ ]) and double brackets ([[ ]])) (#aren't POSIX compliant)
  - Pattern Matching (wildcard * (asterisk) within the double brackets)
    - $ name="Alice"
    - $ [[$name = *c*]] && echo "Name includes c"

      Name includes c
    - $ echo $?

      0

    - $ name="Alice"
    - $ [$name = *c*] && echo "Name includes c"
    - $ echo $?

      1

- Bash scripting - (single brackets ([ ]) and double brackets ([[ ]])) (#aren't POSIX compliant)
  - Regular Expressions (=~)
    - $ name="Alice"
    - $ [[$name =~ ^Ali]] && echo "Regular expressions can be used"
      Regular expressions can be used
    - $ name="Alice"
    - $ [$name =~ ^Ali] && echo "Regular expressions can be used"
      bash: [: =~: binary operator expected

- Bash scripting - (single brackets ([ ]) and double brackets ([[ ]])) (#aren't POSIX compliant)
  - Word Splitting
    - $ filename="nonexistent file"
    - $ [[! -e $filename]] && echo "File doesn't exist"

      File doesn't exist

    - $ filename="nonexistent file"
    - $ [! -e $filename] && echo "File doesn't exist"

      bash: [: nonexistent: binary operator expected

    - This is related to the IFS variable. If IFS isn't set, the shell splits the string when it encounters a space, tab, or newline. Variable must be put in to the double quotes if we want to prevent the word splitting within single brackets:

    - $ filename="nonexistent file"
    - $ [! -e "$filename"] && echo "File doesn't exist"

      File doesn't exist

- Bash scripting - conditions and sequences
  - case - in - esac
    - ```
      case value in
            value1) command1;;
            value2) command2;;
            ...
      esac
      ```
    - case uses shell pattern matching
      - https://www.gnu.org/software/bash/manual/bashref.html#Pattern-Matching
      ```

- Bash scripting - conditions and sequences
  - case examples:
    - ```
      # example:
      PLACE=4508
      case "$PLACE" in
           1) echo "gold";;
           2) echo "silver";;
           3) echo "bronze";;
           [1-9]*) echo "place number: $PLACE";;
      esac
      ```
    - ```
      # example
      case "$COUNTRY" in
           "United Kingdom"|Ukraine) echo "Europe";;
           USA|Canada) echo "America";;
           Ira?) echo "Asia (Iran or Iraq)";;
           *) echo "unknown country";;
      esac
      ```

Bash scripting - loops

- Bash scripting - loops
  - while - do - done, until - do - done
    - ```
      while [ condition ]
      do
            # commands
      done
      ```
      - ```
        while [ condition ]; do command1; command2; done
        ```
      - ```
        # example:
        while [ "$PASSWORD" != "telcordia_4ever" ]
        do
              print -n "Input password: "
              read -s PASSWORD
        done
        ```
        ```

- Bash scripting - loops
  - while - do - done, until - do - done
    - ```
      # example
      while read VARIABLE
      do
            echo "$VARIABLE"
      done < input_file.txt | grep XXX
      ```
    - ```
      # example
      typeset -i C=10
      while [[ $C -gt 0 ]]
      do
              echo "C = $C"
              (( C -= 1 ))
      done
      ```

- Bash scripting - loops
  - for - do - done
    - ```
      for VARIABLE in value1 value2 …
      do
            # commands
      done
      ```
      - ```
        # example:
        for SEA in Black Caribbean Mediterranean Baltic
        do
              echo "$SEA Sea"
        done
        ```
      - ```
        # example:
        for FILE in ~/*.txt
        do
              echo "File found: $FILE"
        done
        ```
        ```

- Bash scripting - loops
  - Interrupting loops
    - break
      - ```
        while true
        do
              print -n "Input password: "
              read -s PASSWORD
              [ "$PASSWORD" = "telcordia_4ever" ] && break
              echo "Password is incorrect"
        done
        echo ""
        ```
    - continue
      - ```
        while true
        do
              echo "Input username and password:"
              read USERNAME
              [ -z "$USERNAME" ] && continue
              read -s PASSWORD
              [ "$PASSWORD" = "telcordia_4ever" ] && break
        done
        ```

Bash scripting - functions

- Bash scripting - functions
  - Syntax
    - ```
      functionName(){
            # some code
      }
      ```
    - ```
      function functionName {
            # some code
      }
      ```
  - Input arguments
    - ```
      functionName(){
            echo "Argument #1: $1"
            echo "Argument #2: $2"
            echo "All arguments: $*"
            echo "Number of arguments: $#"
            # some code
      }

      functionName arg1 arg2
      ```
      ```

- Bash scripting - functions
  - Return value
    - 0 - success, not 0 - failure
    - ```
      functionName(){
            # some code
            rm file.txt
            RET_CODE=$?
            echo "Exiting..."
            return $RET_CODE
      }
      ```
  - Calling
    - `functionName arg1 arg2`
    - `FUNC_OUTPUT=`functionName arg1 arg2``
    - `echo $?`
      ```

- Bash scripting - functions
  - Local and global variables

```
f(){
   A=local_a
   local B #bash only
   typeset B #bash and ksh
   B=local_b
}

function g {
   C=local_c
   local D #bash only
   typeset D #bash and ksh
   D=local_d
}
```

```
f
g
echo "A = $A"
echo "B = $B"
echo "C = $C"
echo "D = $D"
```

- bash

```
A = local_a
B =
C = local_c
D =
```

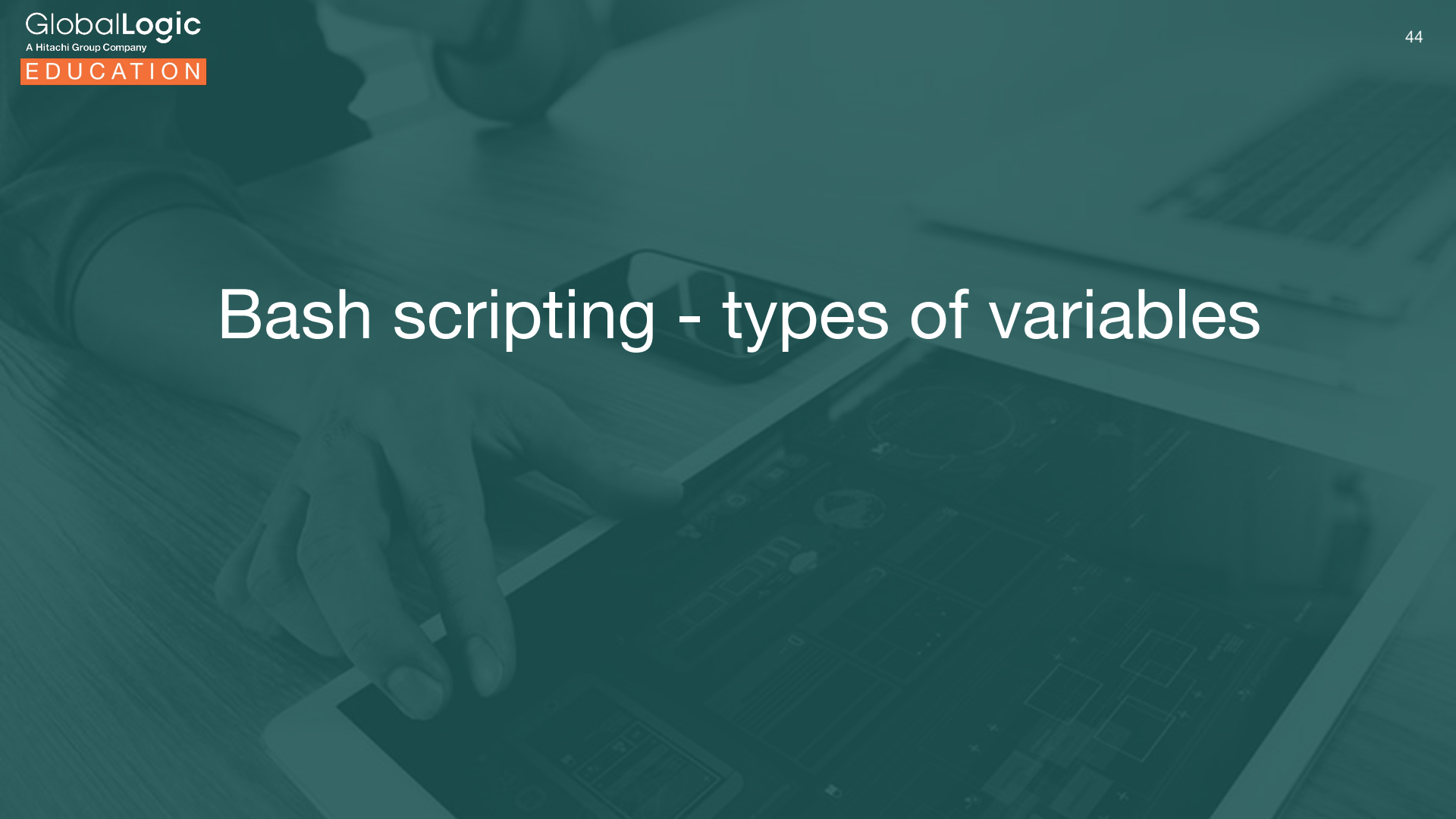
- ksh

```
A = local_a
B = local_b
C = local_c
D =
```

- Bash scripting - functions
  - Exporting functions

```
function myInheritedFunc {
echo inherited
}
export -f myInheritedFunc
```

Bash scripting - types of variables



- Bash scripting - types of variables
  - typeset and declare keywords
    - bash
      - `typeset`
      - `declare`
    - ksh
      - `typeset`

- Bash scripting - types of variables
  - Integers
    - `$ man bash`
      - Section "Arithmetic Expansion"
      - Section "ARITHMETIC EVALUATION"
    - ```
      typeset -i I
      I=0
      let I=$I+1
      I=$(( $I + 1 ))
      ```
    - ```
      typeset -i I=0
      (( I += 1 ))
      ```
    - ```
      typeset -i I=0
      (( I++ ))
      ```
      ```

- Bash scripting - types of variables
  - Indexed arrays
    - `$ man bash`
      - Section "Arrays"
    - ```
      typeset -a ARRAY
      ARRAY[0]="value1"
      ARRAY[1]="value2"
      …
      echo ${ARRAY[0]}
      …
      echo ${ARRAY[*]} # all elements
      echo ${#ARRAY[*]} # number of elements
      ```
    - ```
      # example
      typeset -a ARRAY
      ARRAY=( value1 value2 value3 value4 )
      ARRAY=( $(cat array.txt) ) # reading data from file to the array
      ```

- Bash scripting - types of variables
  - Indexed arrays
    - ```
      # example
      typeset -a A
      typeset -i I
      I=1
      while read A[$I]
      do
              I=$(( $I + 1 ))
      done < array.txt # reading data from file to the array

      for VALUE in ${A[*]} # passing array to "for" loop
      do
              echo "Value is: $VALUE"
      done
      ```
      ```

- Bash scripting - types of variables
  - Indexed arrays
    - Accessing all elements via `*` and `@`.

```
A=( value1 "value2 value3" value4 )
```

- Using `${A[*]}` expression (WITHOUT QUOTES):

```
for VALUE in ${A[*]}
do
      echo $VALUE
done
value1
value2
value3
value4
```

- Using `"${A[*]}"` expression (WITH QUOTES):

```
for VALUE in "${A[*]}"
do
      echo $VALUE
done
value1 value2 value3 value4
```

- Bash scripting - types of variables
  - Indexed arrays
    - Using `${A[@]}` expression (WITHOUT QUOTES):

```
for VALUE in ${A[@]}
do
      echo $VALUE
done
value1
value2
value3
value4
```

    - Using `"${A[@]}"` expression (WITH QUOTES):

```
for VALUE in "${A[@]}"
do
      echo $VALUE
done
value1
value2 value3
value4
```

- Bash scripting - types of variables
  - Associative arrays
    - `$ man bash`
      - Section "Arrays"
    - ```
      typeset -A ARRAY
      ARRAY[qqq]="value1"
      ARRAY[www]="value2"
      …
      echo ${ARRAY[q]}
      …
      echo ${ARRAY[*]} # all elements
      echo ${#ARRAY[*]} # number of elements
      ```
    - ```
      # example
      typeset -a ARRAY=( [key1]="value1" [key2]="value2" [key3]="value3" )
      ```
    - Accessing all elements via `*` and `@`.
      - The same as for indexed arrays with one difference: the values are not sorted in associative arrays.

String operations in bash

- String Manipulation Operations
  - Variable-slicing syntax”
    - $ var="something wicked this way comes..."
    - $ echo ${var:10} # *wicked this way comes...*
    - $ echo ${var:10:6} # wicked
    - $ echo ${var:(-4)} # last 4 digits (s…)
    - $ echo ${var:(-4):2} # 2x 1st digits from the last 4 digits.
    - *# Extract the digits after the decimal point.*
      fractionalPart="${fvalue#*\.}" (#echo "$var" | cut -f2 -d.)
    - $ VAR1=abc
    - echo "$VAR1xyz" #Empty string
    - echo "${VAR1}xyz" # abcxyz

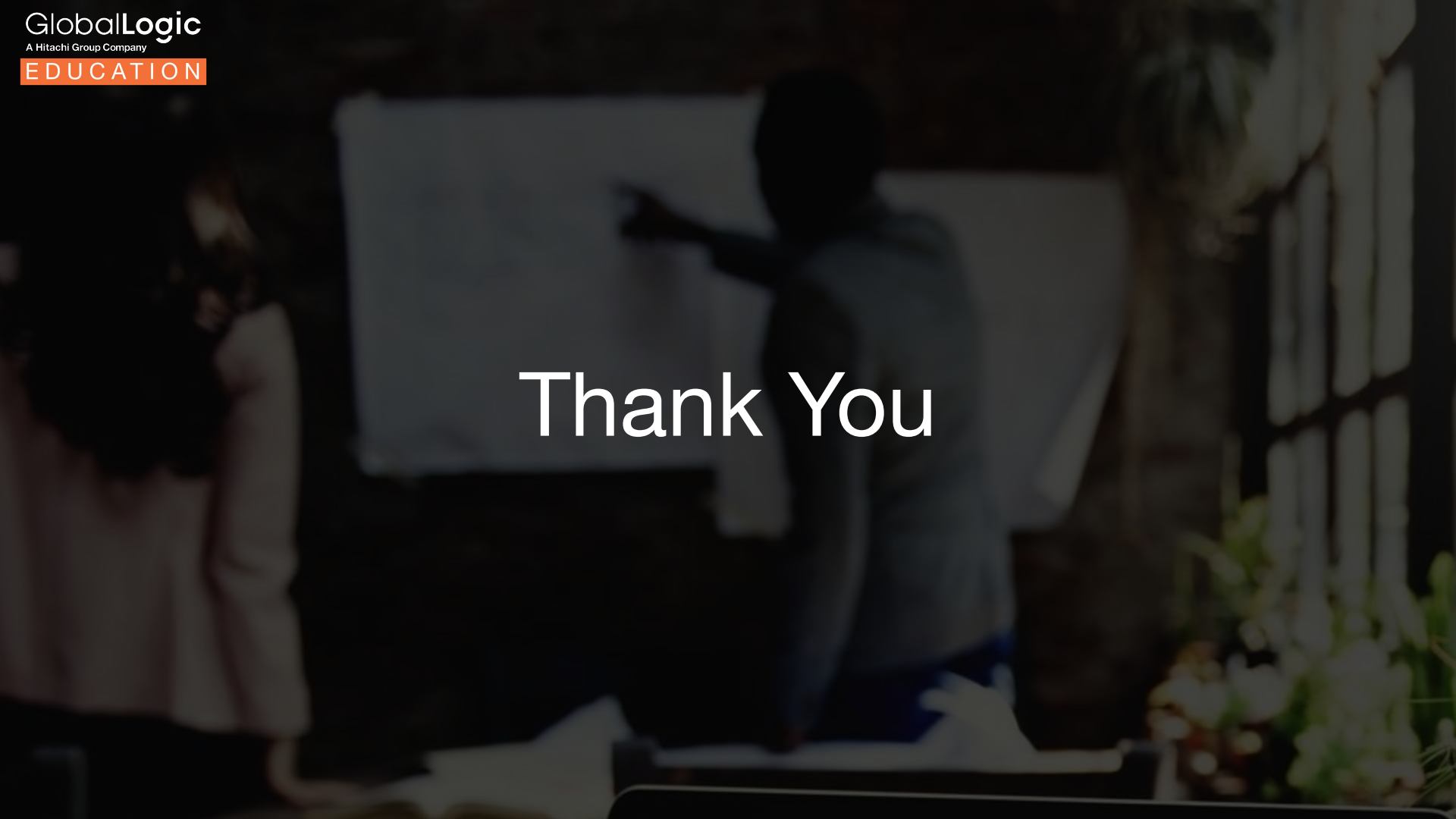
Bash command substitution

- Command Substitution
  - Command substitution `# allows us to execute a command and substitute it with its standard output. Note this command executes within a subshell, which means it has its own environment and so it will not affect the parent shell's environment.`
    - `$ var_date=$(date) && echo $var_date`

      `вівторок, 21 січня 2025 15:59:55 +0200`

      `# In this case, we are using the $(..) form where the command to be executed is enclosed between the parentheses. This form is the recommended syntax of command substitution.`
    - `$ var_date=`date` && echo $var_date`

      `вівторок, 21 січня 2025 16:00:05 +0200`


GlobalLogic
A Hitachi Group Company
EDUCATION
Thank You