# Operating system Fundamentals

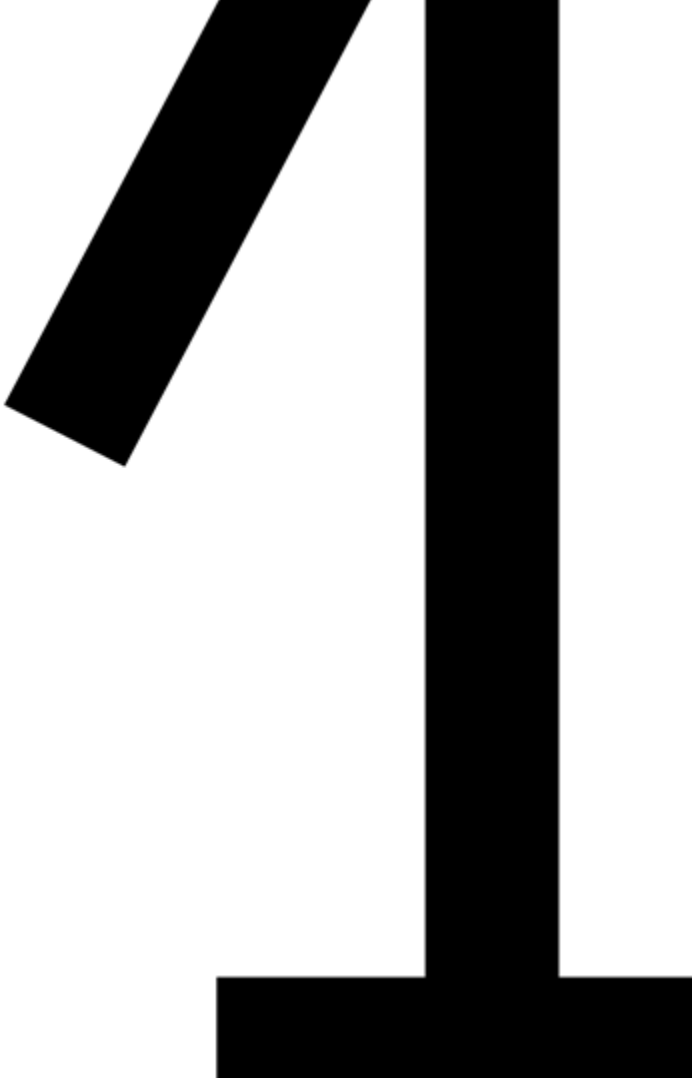
Ch 08 Scripts part 1

KdG Karel de Grote Hogeschool

# Contents

- recap
- simple scripts
- variables
- back quotes
- arguments
- log files

# Recap



# Linux command

- Every command is a programme
- Every command has two inputs
  - standard input (System.in)
  - command-line parameters (and options)  (args)
- Every command has three outputs
  - standard output (System.out)
  - standard error (System.err)
  - exit code (System.exit())

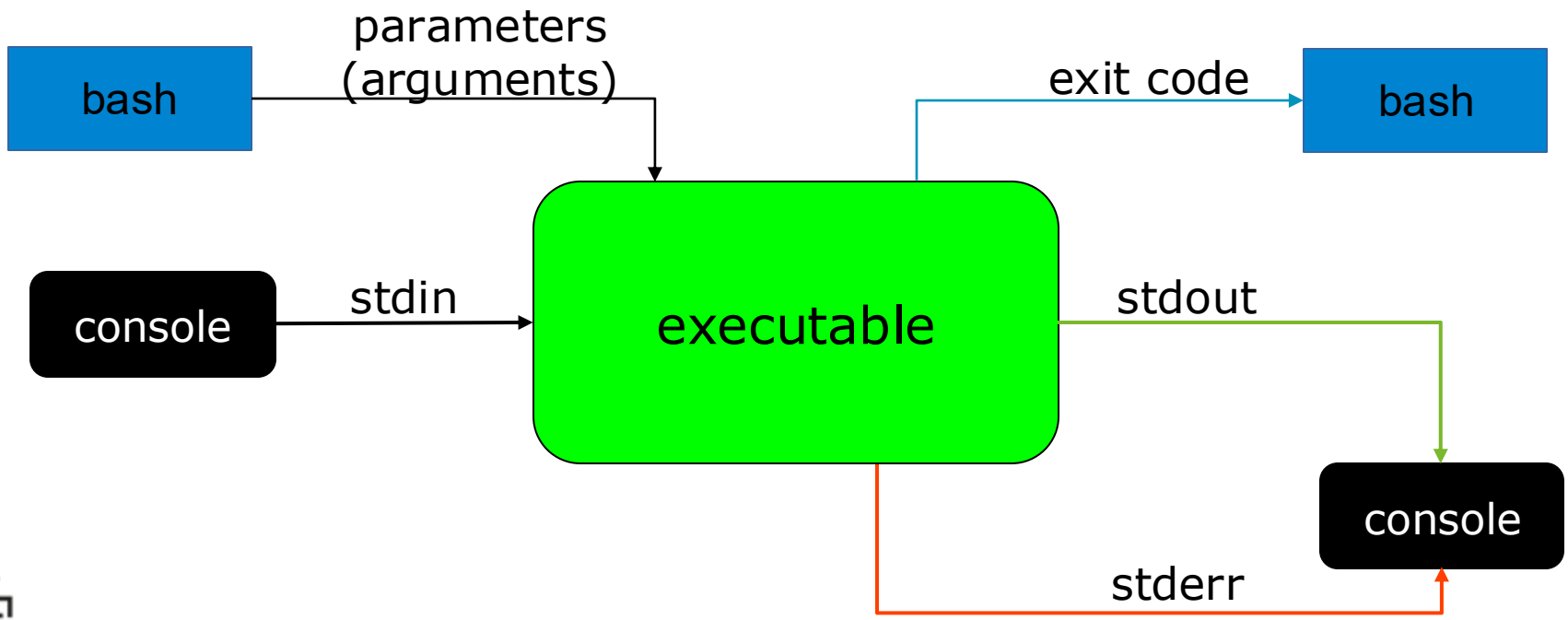
# Example in Java

```
public static void main(String[] args) {
 if (args.length != 0) {
    System.err.println("Error! Do not provide parameters\n");
    System.exit(1);
 }
 String s;
 s = new Scanner(System.in).nextLine();
 System.out.printf("You entered the string '%s'\n", s);
}
```

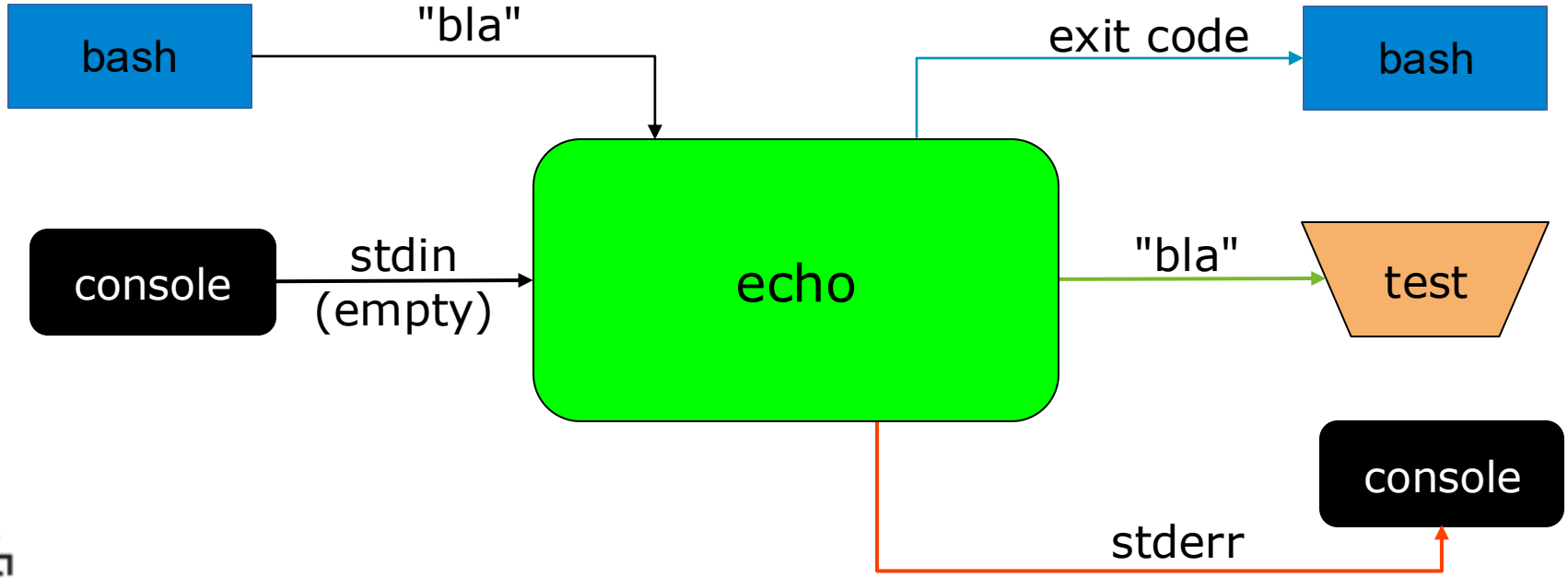
# Example in C

```
#include <stdio.h>
int main(int argc, char** argv) {
  if (argc != 1) {
    fprintf(stderr, "Error! Do not provide parameters\n");
    return 1;
  }
  char s[100];
  scanf("%s", s);
  printf("You entered the string '%s'\n", s);
  return 0;
}
```

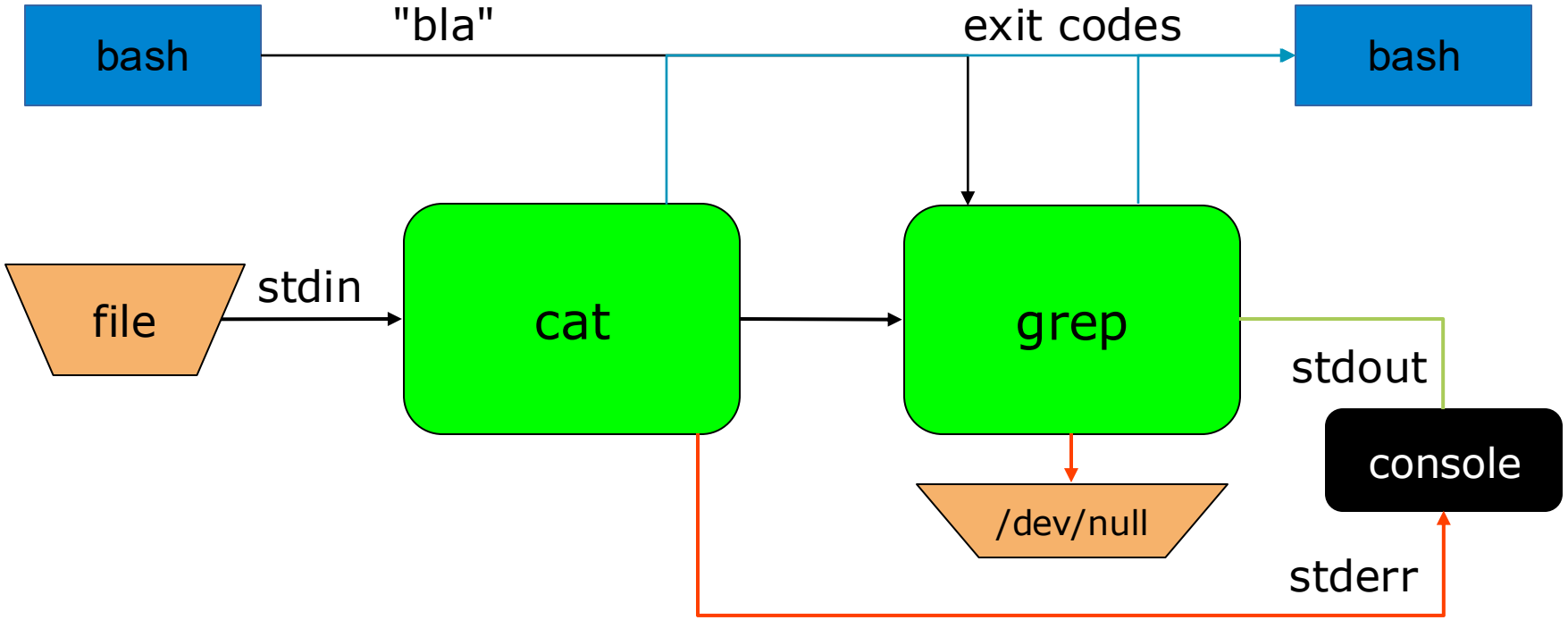
# Unix command



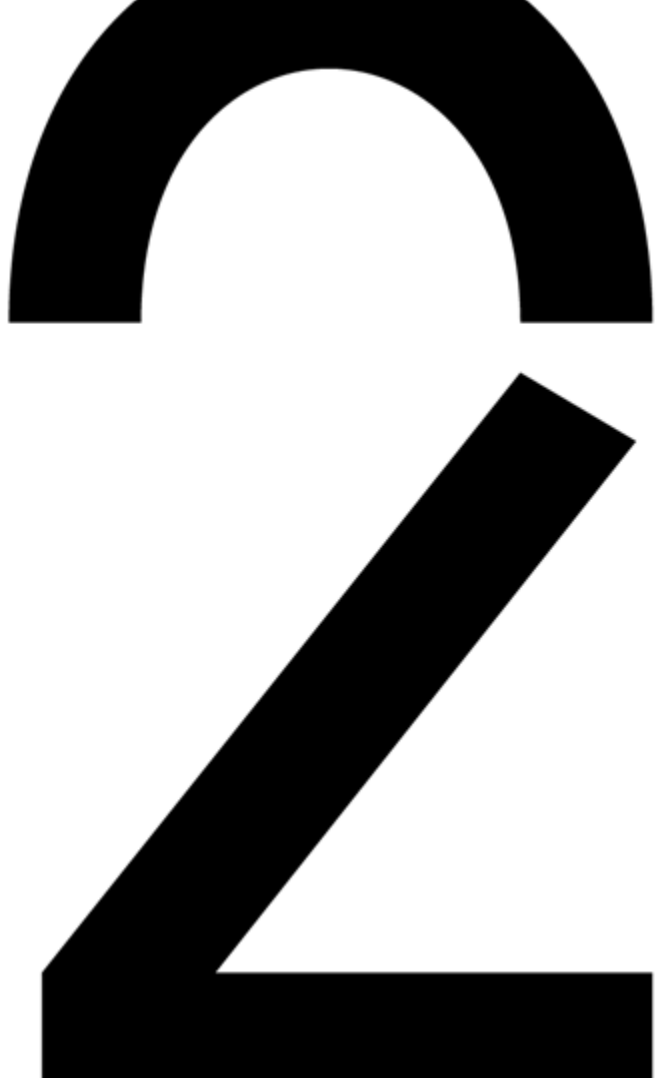
# echo "bla" >test



# cat file | grep -E "bla" 2>/dev/null



# Simple scripts



# Simple scripts

# Simple scripts

- A script is a text file.
- A script can be written in different languages (Bash, R, Python, PHP, Perl, etc.)
  → we write in bash
- simplest script = list of commands
- useful for bundling commands and executing them regularly (automation)

# Example: backup

- Suppose: you want to take regular backups
- Commands:

```
tar zcf /tmp/backup.tgz /root 2>/dev/null
sync
```

- problems:
  - lots of typing
  - risk of typing errors

# Solution: create a script

- Create a text file "~/bin/backup1" containing:

```
echo "Backing up..."
tar zcf /tmp/backup.tgz /root 2>/dev/null
sync
echo "Done."
```

- Make it executable with:

```
chmod +x ~/bin/backup1
```

# Shebang

- When executing, bash will interpret the lines
- What if you call the script from another shell?
- What if the script is written in a different shell/language?

| Shell Name | Full Name / Origin | Key Characteristics | Primary Use Case |
|---|---|---|---|
| **sh** | Bourne Shell | Original Unix shell; basic, foundational. | System scripting (basic), historical. |
| **csh** | C Shell | C-like syntax for scripting; interactive features. | Less common for general scripting; niche use. |
| **ksh** | Korn Shell | Superset of Bourne; blends Csh's interactive features with sh's scripting power; performant. | Enterprise environments, robust scripting. |
| **bash** | Bourne-Again SHell | Most common modern shell; superset of sh; extensive features, good for interactive and scripting. | General purpose user shell, widespread scripting. |
| **zsh** | Z Shell | Extends Bash; very feature-rich, highly customizable (plugins like Oh My Zsh); excellent for interactive use. | Advanced interactive users, modern scripting. |
| **ash** | Almquist Shell | Very lightweight, POSIX-compliant; focus on small size. | Embedded systems, initramfs, base for `dash`. |
| **dash** | Debian Almquist Shell | Fork of `ash`; minimalist, fast, strict POSIX. | System scripting, `/bin/sh` on Debian/Ubuntu, fast boot. |

# Shebang

- Solution:

```
#!/bin/bash
echo "Backing up..."
tar zcf /tmp/backup.tgz /root 2>/dev/null
sync
echo "Done."
```

- Other options:
  - /bin/sh, /usr/bin/perl, /usr/bin/python3, …
  - example: **#!/usr/local/bin/python3**
    example: **#!/usr/bin/env bash**

# Exercise

Create a new text file called hello.sh with the following content:

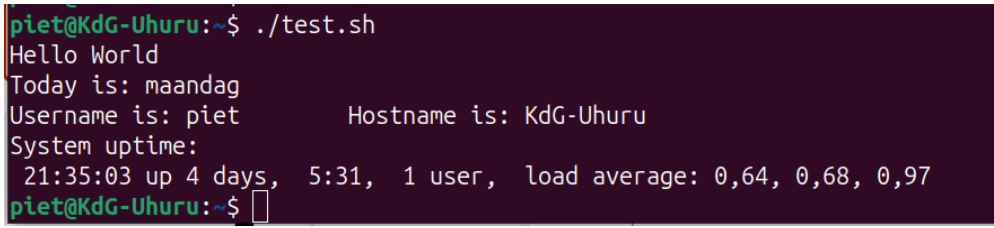
```
#!/bin/bash
echo "Hello world"
```

Make the file executable:

```
chmod +x hello.sh
```

Run the file:

```
./hello.sh
```




Extend the script: Print today's date, your username, hostname the uptime of the system

```
#!/bin/bash

echo "Hello World"
echo -n "Today is: "
#This is a comment: Run the date command
date +%A
echo -e "Username is: $USER\t Hostname is: $HOSTNAME"
echo "System uptime:"
# Run the uptime command
uptime
```

# Variables

# Variables

- What if we want to back up to a different directory?
- What if we want a different file name for the tar file?
- What if we want to back up other files?
- ...

# Variables

- Start with letter (or _)
- Avoid hyphens (-)
- Lowercase with Underscores for Local Variables
- Uppercase  for Constants and Environment Variables
- No CamelCase
- Descriptive: not dir but backup_dir

```
#!/bin/bash
backup_dir=/tmp
backup_file=backup.tgz
files=/root

echo "Backing up to $backup_file..."
tar -zcf "${backup_dir}/${backup_file}" ${files} 2>/dev/null
sync
echo "Done."
```

KdG
Karel de Grote Hogeschool

# Variables – use {}

**${var}**

```
$ var=Good
$ echo $varbye
→ not output
$ echo ${var}bye
Goodbye
```

# Variables – use "${var}"

```
$ mkdir "My Documents"
$ var="My Documents"
$ ls ${var}
ls: cannot access 'My': No such file or directory
ls: cannot access 'Documents': No such file or directory
$ ls "${var}"
$ touch "${var}/document"
$ ls "${var}"
document
$
```

KdG
Karel de Grote Hogeschool

# Variables – use "${var}"

```
$ var="*.txt"
$ echo ${var}
*.txt
$ touch doc.txt
$ echo ${var}
doc.txt
$ echo "${var}"
*.txt
$ ls "${var}"
ls: cannot access '*.txt': No such file or directory
```

KdG
Karel de Grote Hogeschool

# Variables scope – use export

Create a script: var.sh

```
#!/bin/bash

echo ${var}
```

Execute the commands:

```
$ var=Hello

$ ./var.sh

→ no output
$ echo $var

Hello


$ export var
```

# Variables scope – use export

- If you want to change variables outside the script (such as environment variables):

  export PATH=${PATH}:/home/jan/bin

# "read only" Variables

Create readonly variables
(like "final" in Java)

```
$ readonly backup_dir="/root"
$ echo $backup_dir
 /root
$ backup_dir="/etc"
 -bash: backup_dir: readonly variable
```

# Numeric variables - integers

- Variables do not need to be declared: they are always of type String.
- You can put a number in a variable, but it will then be a String
- If you want to perform calculations with these numbers, use the following syntax (only works with integers!):

```
var=1
echo $((var+1))
var=$((var+4))
```

# Numeric variables – decimal numbers

- Install bc → `sudo dnf install bc`
  bash calculator

- bc accepts a string as input
  eg. echo "3.14 * 2* 3.0" | bc –l

- If you want to calculate with variables:

```
pi=3.14
rad=3
echo "$pi * 2 * $rad" | bc –l
echo $(echo "$pi * 2 * $rad" | bc -l)
circ=$(echo "$pi * 2 * $rad" | bc -l)
printf "%.2f\n" "$circ"
```

# Input questions

```
#!/bin/bash
readonly backup_dir=/temporary
readonly files=/root

echo -n "enter backup filename ->"
read -r backup_file
echo "Backing up to ${backup_file}..."
tar -zcf "${backup_dir}/${backup_file}" ${files} 2>/dev/null
sync
echo "Done."
```

# read

read is a bash "built-in", not a separate programme.

There is no man page, but you can request help via:

```
help read
```

```
read -r → do not allow backslashes
read -s → do not echo input coming from a terminal
read -rs → use this for passwords
```

# Exercise

Create a script that asks the user for two numbers and returns the sum and the multiplication

Example ouput:

```
Enter a number: 37
Enter a number: 5
The sum of 37 and 5 is 42 and muliplication is 185
```

# Default values

- ${word} displays the content of "word", but what if it does not exist?

- ${word:=hello}
  - returns the content of the variable "word"
  - if "word" does not exist (or is empty), this returns the word "hello" and the variable also gets this value

# Default values

```
#!/bin/bash

backup_dir=/temporary
files=/root

echo -n "enter backup filename (backup.tgz) ->"
read -r backup_file
echo "Backing up to ${backup_file:=backup.tgz}..."
tar -zcf "${backup_dir}/${backup_file}" ${files} 2>/dev/null
sync
echo "Done."
```

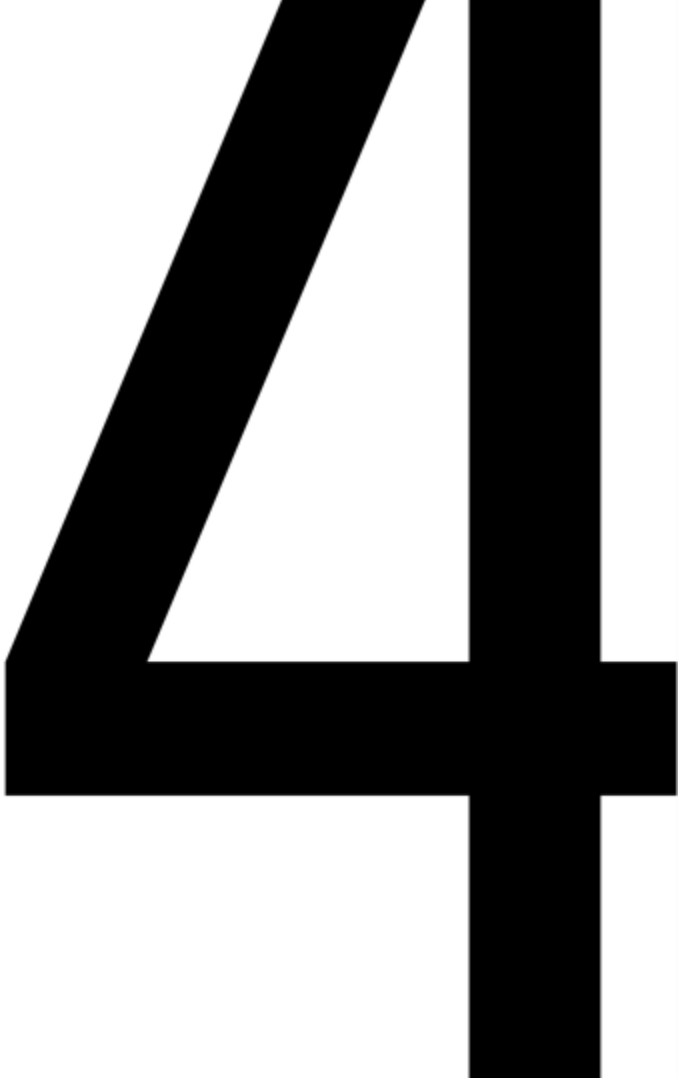
is used if backup_file does not exist or is empty;
the variable is now also changed

# Checking variables

- You can also check whether a variable has a value
  ${word:?variable does not exist!}
  - continues if "word" already has a value
  - gives an error message if "word" has no value
    and stops the script

- example:
  - cd ${JAVA_HOME:?"error JAVA_HOME is empty"}
  - param1=${1:?Parameter missing}

# Quotes



# Quotes

- There are three types of quotations
  - 'single': take the text between them literally
  - 'double': replace any variables with their value
  - back quotes: execute the command within the quotes and replace the variable with the standard output command
    legacy

# Back quotes

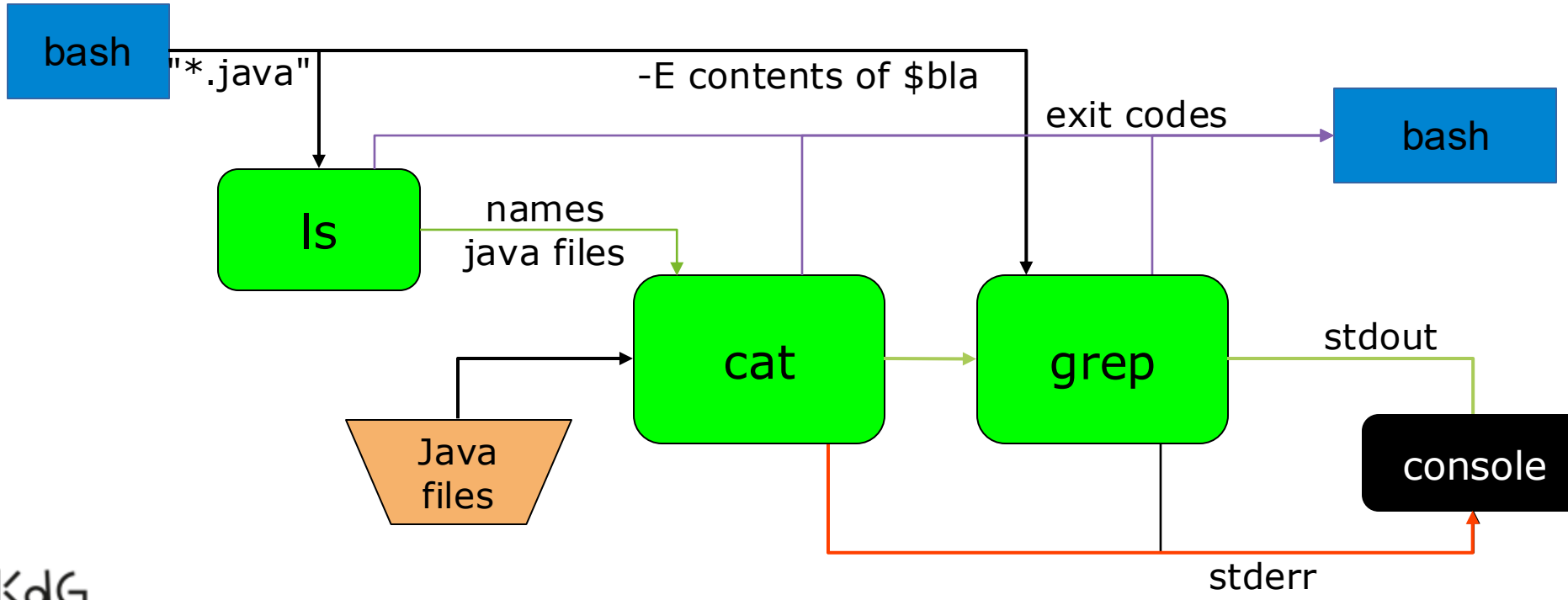
```
hello=hola
echo "date ${hello}"
echo "date \${hello}"
echo 'date ${hello}'
echo `date` ${hello}
```

The use of backquotes for command execution is legacy.
It is not very readable and prone to errors.

Use command substitiution instead

`date`
$(date)

# cat $(ls *.java) | grep -E "$bla"



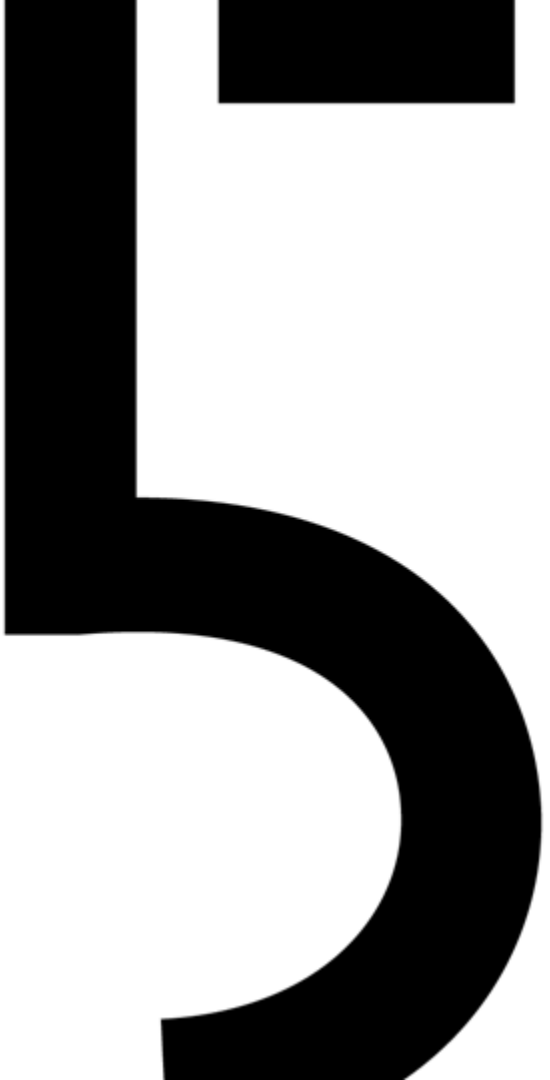
# Backup script

```
#!/bin/bash

#today=`date +%Y%m%d` → legacy notation
today=$(date +%Y%m%d)
backup_dir=/temporary
backup_file=backup${today}.tgz
files=/root

echo "Backing up to $backup_file..."
tar -zcf "${backup_dir}/${backup_file}" ${files} 2>/dev/null
sync
echo "Done."
```

# Arguments



# Arguments

- Every command in Unix can receive command-line parameters (arguments) as input.
- a script can also use these
  - Parameters are separated by spaces in scripts:
    - $0 = name of the script
    - $1, $2, ..., $9 = first to ninth parameter
    - $# = number of parameters
    - $* = all parameters as 1 string
    - $@ = all parameters as an array

# Example

```
#!/bin/bash
echo "number of parameters: $#"
echo "all parameters as 1 string: $*"
echo "all parameters as 1 array: $@"
echo "command = $0"
echo "parameter 1 = $1"
echo "parameter 2 = $2"
echo "parameter 11 = ${11}"
echo "parameter 11 = ${12?Parameter not submitted}"
```

# Backup script

```
#!/bin/bash

today=$(date +%Y%m%d)
backup_dir=/temporary
backup_file=backup$today.tgz
files="/root"

echo "Backing up to $backup_file..."
tar -zcf "${backup_dir}/${backup_file}" ${files} "$@"   \
                                                2>/dev/null
sync
echo "Done."
```

# Using $@ and $* with ""

```
$ cat /opt/share/scripts/params.sh

#!/bin/bash
echo "\$1  : $1"
echo "\$2  : $2"
for i in $*   ; do echo "\$*  : $i"   ; done
for i in "$*" ; do echo "\"$\" : $i"  ; done
for k in $@   ; do echo "\$@  : $k"   ; done
for j in "$@" ; do echo "\"\$@\": $j" ; done

$ /opt/share/scripts/params.sh "My Documents" "My Files"
```

# Exercise – modifiy the previous script:

Create a script that ~~asks the user for two numbers~~
Read the input from command line arguments
and returns the sum and the multiplication

Example ouput:

```
Enter a number: 37
Enter a number: 5
The sum of 37 and 5 is 42 and muliplication is 185
```

# Log files

# Scripts often create log files

```
#!/bin/bash

today=$(date +%Y%m%d)
backup_dir=/temporary
backup_file=backup${today}.tgz
files="/root $@"
logfile=/var/log/backup.log

echo "Backing up to $backup_file..."
tar -zcf "${backup_dir}/${backup_file}" ${files} 2>>${logfile}
sync
echo "${today}: backup successful" >>${logfile}
echo "Done."
```

# Conclusion

- A script is like a Java programme:
  - echo -> System.out.println()
  - read -> keyboard.nextLine()
  - variables -> always type String
  - use $((...)) or bc to calculate
  - exit -> System.exit()
  - $1, $2, ... -> args
  - if, for, while, switch -> next lesson
  - functions (methods) -> next lesson

# Exercises

# Exercises

## Canvas → chapter 8

**8.1** Write a bash script named `multiply.sh`.

The script accepts 2 arguments.

Print the product of the 2 arguments to STDOUT. (`echo`)

**8.2** Write a script `loggedin.sh` that displays a sorted list of all logged-in usernames.

**8.3** Create a script countfiles.sh that counts how many directories and files are in the current user's home directory. Store that count in the variables "filecount" and "dircount".
Print to the screen: "The home dir contains <filecount> files and <dircount> directories."

**8.4** Write a script `hello.sh` that expects a name as a parameter. The script saves this name in a variable `name`. If no parameter was provided, the script stops with an error message. If a parameter was provided, the script says `<date>: Hello, <name>` where the date and name are filled in.

**8.5** Write a script `searchwords.sh` that asks the user for 2 words. The script searches for these words in the file "/usr/share/dict/words" (create this beforehand) and displays every line where one of these words occurs. If one word is empty, it is replaced by "emty" and searched for that instead.

# Exercises

## Canvas → chapter 8

**8.6** write a script archive.sh:

* This script uses the `find` command to search for all files with the `.sh` extension in your home directory (and subfolders).
* Use the `-exec` option with `find` to copy these files into the `/tmp/shellscripts` folder.
* The script then creates the tar archive `shellscripts.tar.gz` in the `/tmp` folder. All copied shell scripts will be included in that archive.
* Verify your result (are all files in the tar file?).
* Ensure that the script at the end writes a line: "x files have been archived in shellscripts.tar.gz" (where x is the count).
* Use a variable so you can easily reuse and change the name `shellscripts.tar.gz`.