File Organizer file link
file-organizer
To run the file-organizer script we can use one of the following commands: the first command is for debugging and the second one is for normal execution.

sudo bash -x file-organizer4.sh /home/habib/Downloads/
bash file-organizer4.sh /home/habib/Downloads/

set -euo pipefail

set -e
The set -e option instructs bash to immediately exit if any command has a non-zero exit status. You wouldn't want to set this for your command-line shell, but in a script it's massively helpful. In all widely used general-purpose programming languages, an unhandled runtime error

set -u
Affects variables. When set, a reference to any variable you haven't previously defined - with the exceptions of $* and $@ - is an error, and causes the program to immediately exit

set -x (For disable set +x)
Enables a mode of the shell where all executed commands are printed to the terminal. In your case it's clearly used for debugging, which is a typical use case for set -x : printing every command as it is executed may help you to visualize the control flow of the script if it is not functioning as expected.

set -o pipefail
By default, when you use a pipeline (commands connected by |), the exit status of the entire pipeline is the exit code of the last command only.

Example (without pipefail):
cat nonexistentfile | grep "hello"
echo $?
cat nonexistentfile → fails (returns non-zero exit code)
grep "hello" → succeeds (returns 0 because it runs, even with no input)
The pipeline's overall exit code = 0 (success), even though the first command failed.
This can hide real errors in your script.

With pipefail:
set -o pipefail
cat nonexistentfile | grep "hello"
echo $?
Now, the exit code of the pipeline becomes the exit code of the first failed command in the chain.
So here:

cat nonexistentfile fails → pipeline exit code = non-zero (error).
Script detects the failure correctly.

~~IFS=$'\n\t'~~
~~Only split words at newlines (\n) or tabs (\t), but NOT spaces.~~
~~That makes the script much safer for filenames with spaces.~~

```
# === Enable safe globbing ===
shopt -s nullglob    # empty matches expand to nothing
# shopt -s dotglob   # uncomment to include hidden files
```

shopt -s nullglob
When you use a wildcard pattern like *.txt, Bash expands it to all matching files.
By default, if there are no matches, the pattern remains literal (unchanged).
nullglob changes that behavior.
nullglob makes an empty match expand to nothing, instead of leaving the literal *.txt

shopt -s dotglob
By default, Bash ignores hidden files (those starting with a dot .) when using wildcards like *.
dotglob makes patterns like * also include hidden files.

Conditional Statement

```
if [ $# -ne 1 ]; then
   echo "Usage: $0 /home/habib/Downloads"
   exit 1
fi
```

$# → number of arguments passed to the script.
-ne → means "not equal".
$0 → Represents the name of the currently running script file (e.g., file-organizer.sh).
If the number of passing arguments is not equal to one (1) then it will print usage message and exit the conditional statement. “Usage: $0 /home/habib/Downloads”

```
if [ ! -d "$dir" ]; then
   echo "Error: $dir is not a directory."
   exit 1
fi
```

Conditional Statement
-d → Checks if the file is a directory (true if the $dir is a directory)
! → means “not”

The statement goes inside if the passing argument ($dir) will not a directory.

```
cd "$dir" || exit 1   # Move into the target directory (exit if it fails)
```

Change the directory to $dir directory
$dir → The argument you pass when running the script will be the directory where you'd like to organize files into structured sub-directories.

```
# === Logging setup ===
timestamp=$(date +"%Y-%m-%d/%H:%M:%S") #2025-10-21/10:04:24
log_file="$dir/organize_${timestamp}.log"
```

Displaying the date and time in a customized way we use like that.

```
exec > >(tee "$log_file") 2>&1 # tee -a
```

This line redirects all output from the script — both standard output (stdout) and standard error (stderr) to your log file and terminal.

```
# === Define categories (extend easily) ===
declare -A types=(
  ["Images"]="jpg jpeg png gif bmp svg webp"
  ["Docs"]="pdf docx doc txt odt xlsx csv"
  ["Archives"]="zip tar gz bz2 rar 7z tar.gz"
  ["Videos"]="mp4 mkv avi mov"
  ["Audio"]="mp3 wav flac m4a"
)
```

declate -A → Defines an associative array --- i.e., a key value mapping
Key = Category name (e.g., Images, Docs, Archives)
Value = List of file extensions under that category.

```
# === Total files count for progress reporting ===
files=( * )
regular_files=()
for f in "${files[@]}"; do
  [[ -f "$f" ]] && regular_files+=("$f")
done
printf "Regular files detected: %d\n\n" "${#regular_files[@]}"
```

files(*)

It expands to match zero or more characters in filenames or directories. It stores(lists) all files and directories, separated by spaces.

for f in "${files[@]}"; do
${files[@]}
It retrieves all the values from the variable “files”
$f
It is a variable that will take on each element one by one from the variable files until all elements have been processed.
[[-f "$f"]] && regular_files+=("$f")

-f
Checks whether the file is a regular file (returns true if it is) and increments the count each time.
regular_files+=("$f")
Addition assignment operator is used to add the value on the right side to variable $f.

printf "Regular files: %d\n" "${#regular_files[@]}"

printf function is used to display results in the terminal. it formats and prints output in a predictable way.
Format string:
%d → decimal integer for count and total
%s → string for the filename
\n → newline
"${#regular_files[@]}" → counts the number of array elements.

You can add padding or alignment if you want prettier output:
printf "[%02d/%02d] Processing: %-20s\n" "$count" "$total" "$file"
%02d → pad numbers with zeros (e.g., 01, 02, 03)
%-20s → left-align the filename within 20 characters

```
# === File Processing Loop ===
# Iterate through each file, detect its type, and move it to the proper folder.
for file in "${regular_files[@]}"; do

  # Skip the log file itself to prevent self-movement
  [[ "$file" == organize_*.log ]] && continue
```

```
  filename="$file"
  ext=""
  moved=false
```

"${regular_files[@]}
[@] This is used to retrieve all the values from the variable $regular_files

[["$file" == organize_*.log]] && continue
Continue statement jumps to the next iteration, skipping the current one when a match is found.
filename="$file"
$file contains one file for each iteration.

```
 # --- Determine File Extension ---
  # Extract file extension and normalize to lowercase.
  if [[ "$filename" == *.* ]]; then
    ext="${filename##*.}"
    ext="${ext,,}"
  fi
```

"$filename" == *.*
This condition tests whether the filename contains at least one dot (.) anywhere
Filenames that start with a dot (hidden files), e.g. .bashrc, also match *.* because they contain a . (the leading dot). For .bashrc

"${filename##*.}"
##*. removes the longest match of anything up to and including the last dot
filename="report.final.txt"
echo "${filename##*.}" # txt

```
file="/etc/nginx/conf.d/nginx.conf"
filename="${file##*/}"   # nginx.conf
basename="${filename%.*}"  # nginx
ext="${filename##*.}"     # conf
```

"${ext,,}"
Convert all uppercase letters in the variable to lowercase.

for cat in "${!types[@]}"; do
"${!types[@]}" Retrieves all the keys of the associative array (Images, Docs, Archives).
cat variable goes through each key one by one. It contains all keys only.
 for e in ${types[$cat]}; do
types[$cat] returns the value of keys like Images, Docs etc.
Example: for cat=Images, it returns "jpg jpeg png gif"

```
    # Compare extension; if matched, move file
      if [[ "$ext" == "$e" ]]; then
        mkdir -p -- "$cat"
```

If the file's extension matches one of the extensions in the list(value of keys' string), then:
Create a folder for that category (if it doesn't already exist).
Move the file there
The -- ensures that if $file starts with a dash (like -test.pdf),
it's treated as a filename, not a command option.

```
dest="$cat/$filename"
          [[ -e "$dest" ]] && {
            base="${filename%.*}"
            ext_part="${filename##*.}"
            ts=$(date +%s)
            dest="$cat/${base}_${ts}.${ext_part}"
            echo "Collision detected, renamed to: $dest"
          }
```

-e means "exists" (can be a file or directory).
`base="${filename%.*}"`
Removes the extension part (everything after the last dot) from the filename
Example:
filename="photo.png"
base="${filename%.*}" # → "photo"

`ext_part="${filename##*.}"`
Extracts only the extension (everything after the last dot)
Example:
filename="photo.png"
ext_part="${filename##*.}" # → "png"
`ts=$(date +%s)`
echo "$ts" # → 1739921183 (changes every second)