Week 3 Shell Scripting & Build Tools

`/etc`

A directory in the file system which contains all the system configuration files

The name `/etc` stands for "et cetera"
The name reflects its purpose as a repository for a variety of system settings and configuration files that did not fit into other directories in the filesystem hierarchy standard.

`env`

look through the PATH and tries to find the program specified and runs it

1. **For portable POSIX shell scripts:** `#! /bin/sh`
    - This is **a direct path** to the `sh` shell, which is the standard command interpreter for the system.
    - The existence of `/bin/sh` is part of the POSIX standard, so you can expect it to be present on any POSIX-compliant system, making the script portable across such systems.
2. **For less portable BASH scripts:** `#! /usr/bin/env bash`
    - `env` is used to invoke the first `bash` interpreter in the system's `PATH` environment variable.
    - because `bash` **can be in different locations on different systems** (`/bin/bash` , `/usr/bin/bash` , `/usr/local/bin/bash` , etc.)
    - By using `env` , the script doesn't rely on `bash` being in a specific location, increasing portability across systems where the path to `bash` may vary.
    - However, the script is considered "less portable" because not all systems may have the `bash` shell installed, whereas `/bin/sh` is almost always available.

`chown`

used to change the owner of files or directories.
It is a command that **requires administrative privileges**, so it is often used with `sudo` to grant such permissions for the duration of the command.

basic syntax

```
chown [OPTION]... [OWNER][:[GROUP]] FILE...
```

- **OWNER**: The user name or user ID (UID) of the **new owner** you want to set for the file or directory.
- **GROUP**: The name or group ID (GID) of the new group you want to set. This is optional and if not specified, **the file's group ownership** is not changed.
- **FILE**: The target file(s) or directory(ies) whose ownership you want to change.
- **OPTION**: There are various options you can use with `chown` to modify its behavior, such as `-R` for recursive change.

examples

1. **Changing the Owner of a File**
   `sudo chown xxxx filename`
2. **Changing BOTH the Owner and Group of a File**
   `sudo chown xxxx:xxxx filename`
   (This changes the owner to "jane" and the group to "users")
3. **Changing the Owner of a Directory Recursively**
   `sudo chown -R jane /path/to/directory`
   (This will change the owner to "jane" for the directory and all its contents)
4. **Changing the Group Only**
   `sudo chown :users filename`
   (leave the owner part blank)
5. **Using User ID (UID) and Group ID (GID)**
   `sudo chown 1001:1002 filename`
   (This changes the owner to the user with UID 1001 and the group to GID 1002)

bash scripts example

```
#!/bin/bash
input="the hardest question"
echo $input | sed s/hard/easi/

# output --> the easiest question
```

1. `#!/bin/bash` :
   This is called a shebang line and it tells the system that this script should be executed using the Bash shell
2. `input="the hardest question"` :
   This line assigns the string "the hardest question" to the **variable** `input`
3. `echo $input | sed s/hard/easi/`
   This line echoes the value of `input`, piping it into `sed` to perform the text transformation. Specifically, `sed s/hard/easi/` means:
   - `s` : **Substitute** command
   - `hard` : The **pattern** to match in the input text.
   - `easi` : The text to **replace** the first occurrence of the pattern with.
   - The final forward slash `/` marks the end of the substitution command.

#sed

stands for **stream editor**, and it is used for filtering and transforming text.
`s/pattern/replacement/flags`

- `s` indicates a substitution operation.
- `pattern` is the text to be searched for
- `replacement` is the text to replace the matched pattern with.
- `flags` are optional and can be used to modify the behavior of the substitution.

Practice

1. functions include `compile()` / `run()` / `build()`
2. The `%` operator in `"${1%.c}"` is used to **remove a trailing substring** from the value of the parameter. Specifically, the `.c` is the substring that will be removed if it is found at the end of `${1}`.
   i.e. hello.c / hello --> both will be compiled
3. `[ -f "$1" ]` checks if the file specified by the first positional parameter (`$1`) exists and is a regular file.
4. `if [ -z "$2" ]; then` in a shell script is used to check if the second positional parameter (`$2`) passed to the script or function is empty
   `-z` : A test option that returns true (0) if the length of the string is zero.

```
#!/bin/sh

# Function to compile a C file
compile() {
    # Check if the file exists with or without the .c extension
    if [ -f "$1.c" ]; then
        filename="$1.c"
    elif [ -f "$1" ]; then
        filename="$1"
    else
        echo "Error: Source file $1.c does not exist."
        return 1
    fi

    # Compile the file
    gcc -Wall -std=c11 -g "$filename" -o "${1%.c}"
}

# Function to run a program
run() {
    if [ -f "$1" ]; then
        ./"$1"
    else
        echo "Error: Program $1 does not exist."
        return 1
    fi
}

# Function to build and then run a program
build() {
    compile "$1" && run "${1%.c}"
}
```

```
# Main script logic
case "$1" in
    compile)
        if [ -z "$2" ]; then
            echo "Error: No filename provided for compile."
            exit 1
        else
            compile "$2"
        fi
        ;;
    run)
        if [ -z "$2" ]; then
            echo "Error: No filename provided for run."
            exit 1
        else
            run "${2%.c}"
        fi
        ;;
    build)
        if [ -z "$2" ]; then
            echo "Error: No filename provided for build."
            exit 1
        else
            build "$2"
        fi
        ;;
    *)
        echo "Usage: ./b COMMAND NAME"
        echo "Commands:"
        echo "  compile NAME - Compiles the specified C file."
        echo "  run NAME     - Runs the specified program."
        echo "  build NAME   - Compiles and runs the specified C file."
        exit 1
        ;;
esac
```

Build tools - maven

Java

- The `javac` compiler turns source files (`.java`) into `.class` files;
- The `jar` tool packs class files into `.jar` files;
- The `java` command runs class files or jar files.

A **Java Runtime Environment (JRE)** contains only the `java` command, which is all you need to run java applications if you don't want to do any development. Many operating systems allow you to double-click jar files (at least ones containing a special file called a `manifest`) to run them in a JRE.

A **Java Development Kit (JDK)** contains the `javac` and `jar` tools as well as a JRE. This is what you need to develop in java.

maven

`maven` is a **Java package manager and build tool**. It is not part of the Java distribution, so you will need to install it separately

The `pom.xml` File

In the context of a Maven project,

1. a `pom.xml` file (Project Object Model) is the fundamental unit of work in Maven.

- It describes the project's configuration, including its dependencies.

2. A dependency in a `pom.xml` file represents an external library or project that your project needs to compile, run, or both.
    - **Dependencies** are a crucial aspect of most Java projects because they allow you to *use code from other projects* without having to include the source code directly in your project.

artifacts identifier

```
<groupId>org.example</groupId> /* Defines the group or organization that the
project belongs to */
<artifactId>project</artifactId>  /* The name of the project artifact */
<version>0.1</version>
```

build properties

determine what version of Java to compile against

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

dependencies

The dependencies section is **where you add libraries you want to use**.

`Junit`
the JUnit library, which **provides ==the testing framework==** used in your Java project

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

plugins

contains the plugins that maven **uses to compile and build your project**.

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>3.0.0</version>
    <configuration>
        <mainClass>org.example.App</mainClass>
    </configuration>
</plugin>
```

CML

`mvn compile` compiles the project
`mvn clean` will remove all compiled files
`mvn exec:java` run the compiled project

- lines coming from maven itself will start with `[INFO]` or `[ERROR]` or similar, so lines without any prefix like that are printed by your program itself

`mvn test` runs the tests in `src/test/java`
`mvn package` creates a jar file of your project in the `target/` folder

If store java project in git repositories

create a file `.gitignore` in the same folder as the `pom.xml` and add the line `target/` to it

- since you don't want the compiled classes and other temporary files and build reports in the repository.
- The `src/` folder, the `pom.xml` and the `.gitignore` file itself should all be checked in to the repository