**⚙️ Build and Shell Steps**

| **Function** | **Description** |
| --- | --- |
| sh 'command' | Runs a shell command (Linux/macOS). |
| bat 'command' | Runs a batch command (Windows). |
| echo 'message' | Prints a message to the console. |
| checkout scm | Checks out source code from version control. |
| tool 'ToolName' | Uses a configured tool (e.g., JDK, Maven). |

**sh 'command'**

In a Jenkins Pipeline, the sh step is used to execute shell commands on the agent where the pipeline is running. This is particularly useful in **Declarative** or **Scripted Pipelines** when you need to run Unix/Linux shell commands (or Windows commands using bat on Windows agents).

Here’s a **detailed explanation** with examples:

**🔧 Basic Syntax**

sh 'your-shell-command-here'

This runs the command in a shell and returns the output to the Jenkins console log.

**✅ Example in a Declarative Pipeline**

pipeline {

    agent any

    stages {

        stage('Build') {

            steps {

                echo 'Running build script...'

                sh 'echo "Hello from shell!"'

            }

        }

    }

}

**Capturing Output from sh**

If you want to **capture the output** of a shell command into a variable, you need to use returnStdout: true:

pipeline {

    agent any

    stages {

        stage('Capture Output') {

            steps {

                script {

                    def result = sh(script: 'echo "Hello, Jenkins!"', returnStdout: true).trim()

                    echo "The result is: ${result}"

                }

            }

        }

    }

}

* returnStdout: true tells Jenkins to return the output.
* .trim() removes any trailing newline characters.
* **Handling Errors**
* By default, if the shell command fails (non-zero exit code), the pipeline will fail. You can override this with returnStatus: true:

def status = sh(script: 'exit 1', returnStatus: true)

echo "Command exited with status: ${status}"

This is useful for **conditional logic** based on command success/failure.

**Example with Conditional Logic**

pipeline {

    agent any

    stages {

        stage('Check File') {

            steps {

                script {

                    def status = sh(script: 'test -f myfile.txt', returnStatus: true)

                    if (status == 0) {

                        echo "File exists."

                    } else {

                        echo "File does not exist."

                    }

                }

            }

        }

    }

}

**bat 'command'**

In a Jenkins Pipeline, the bat step is used to execute **Windows batch commands** on a Windows-based agent. It works similarly to the sh step used for Unix/Linux systems.

**🔧 Basic Syntax**

bat 'your-windows-command-here'

This runs the command in a Windows command shell (cmd.exe) and prints the output to the Jenkins console log.

**Example in a Declarative Pipeline**

pipeline {

    agent {

        label 'windows' // Ensure the agent is a Windows machine

    }

    stages {

        stage('Run Batch Command') {

            steps {

                bat 'echo Hello from Windows!'

            }

        }

    }

}

**Capturing Output from bat**

To capture the output of a batch command:

def result = bat(script: 'echo Hello, Jenkins!', returnStdout: true).trim()

echo "Output: ${result}"

* returnStdout: true captures the output.
* .trim() removes extra whitespace or newlines.

**❌ Handling Errors Gracefully**

To get the exit code without failing the pipeline:

def status = bat(script: 'exit /b 1', returnStatus: true)

echo "Exit code: ${status}"

* returnStatus: true returns the exit code instead of failing the build.

**Example with Conditional Logic**

pipeline {

    agent {

        label 'windows'

    }

    stages {

        stage('Check File') {

            steps {

                script {

                    def status = bat(script: 'if exist myfile.txt (exit /b 0) else (exit /b 1)', returnStatus: true)

                    if (status == 0) {

                        echo "File exists."

                    } else {

                        echo "File does not exist."

                    }

                }

            }

        }

    }

}

**Checkout scm**

In a Jenkins Pipeline, the checkout scm step is used to **check out the source code** from the same repository that triggered the pipeline. This is especially useful in **Multibranch Pipelines** or **Pipeline jobs connected to a Git repository**, where Jenkins automatically provides the scm variable.

checkout scm

This tells Jenkins to use the **SCM configuration** defined for the job (like Git, SVN, etc.) and check out the code.

**✅ Example in a Declarative Pipeline**

pipeline {

    agent any

    stages {

        stage('Checkout Code') {

            steps {

                checkout scm

            }

        }

        stage('Build') {

            steps {

                sh 'ls -la' // or bat 'dir' on Windows

            }

        }

    }

}

**📌 When to Use checkout scm**

* In **Multibranch Pipelines**, Jenkins automatically sets up the scm variable based on the branch that triggered the build.
* In **Pipeline jobs** configured with a Git repository in the UI, checkout scm uses that configuration.
* **🛠️ Advanced: Custom Checkout**
* If you want to check out a different repository or customize the checkout, use the git step or checkout with a full SCM block:

checkout([$class: 'GitSCM',

          branches: [[name: '\*/main']],

          userRemoteConfigs: [[url: 'https://github.com/your/repo.git']]

])

**✅ Updated Scripted Pipeline Example for Private GitHub**

node {

    stage('Checkout') {

        checkout([$class: 'GitSCM',

                  branches: [[name: '\*/sunil']],

                  userRemoteConfigs: [[

                      url: 'https://github.com/your-org/your-repo.git',

                      credentialsId: 'your-credentials-id'

                  ]]

        ])

    }

}

**tool 'ToolName'**

In a Jenkins Pipeline, the tool 'ToolName' step is used to **declare and use a tool** that has been configured in Jenkins' **Global Tool Configuration** (like JDK, Maven, Gradle, etc.). This step ensures the tool is available in the environment and returns the path to the tool's installation directory.

**🔧 Basic Syntax**

def toolPath = tool 'ToolName'

* 'ToolName' must match the name of a tool configured in **Manage Jenkins → Global Tool Configuration**.
* The returned toolPath is the **absolute path** to the tool's home directory.

**✅ Example: Using Maven**

pipeline {

    agent any

    tools {

        maven 'Maven 3.8.6' // Optional: declarative way to auto-install

    }

    stages {

        stage('Build') {

            steps {

                script {

                    def mvnHome = tool 'Maven 3.8.6'

                    env.PATH = "${mvnHome}/bin:${env.PATH}"

                    sh 'mvn clean install'

                }

            }

        }

    }

}

If you've already configured tools like **JDK**, **Maven**, or **Gradle** in Jenkins under **Manage Jenkins → Global Tool Configuration**, then:

**✅ In Declarative Pipelines:**

You **can** use the tools block to automatically install and add the tool to the PATH. For example:

pipeline {

    agent any

    tools {

        maven 'Maven 3.8.6'

        jdk 'JDK 11'

    }

    stages {

        stage('Build') {

            steps {

                sh 'mvn clean install' // Maven is already in PATH

            }

        }

    }

}

In this case, you **do not need to use** the tool step manually.

**✅ In Scripted Pipelines:**

You **must use** the tool step if you want to get the path to the tool and manually add it to the PATH:

node {

    def mvnHome = tool 'Maven 3.8.6'

    env.PATH = "${mvnHome}/bin:${env.PATH}"

    sh 'mvn clean install'

}