STANDARD JENKINS FILE

Declarative Pipeline

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                //
            }
        }
        stage('Test') {
            steps {
                //
            }
        }
        stage('Deploy') {
            steps {
                //
            }
        }
    }
}
```

Scripted Pipeline

```
Jenkinsfile (Scripted Pipeline)
node {
    stage('Build') {
        //
    }
    stage('Test') {
        //
    }
    stage('Deploy') {
        //
    }
}
```

Pipeline

A Pipeline is a user-defined model of a CD pipeline. A Pipeline's code defines your entire build process, which typically includes stages for building an application, testing it and then delivering it, pipeline is Declarative Pipeline-specific syntax that defines a "block" containing all content and instructions for executing the entire Pipeline.
Also, a `pipeline` block is a key part of Declarative Pipeline syntax.

Node

A node is a machine which is part of the Jenkins environment and is capable of executing a Pipeline, `node` is Scripted Pipeline-specific syntax that instructs Jenkins to execute this Pipeline (and any stages contained within it), on any available agent/node. This is effectively equivalent to `agent` in Declarative Pipeline-specific syntax.
Also, a `node` block is a key part of Scripted Pipeline syntax.

Agent

Agent is Declarative Pipeline-specific syntax that instructs Jenkins to allocate an executor (on a node) and workspace for the entire Pipeline.

Stage

A `stage` block defines a conceptually distinct subset of tasks performed through the entire Pipeline (e.g. "Build", "Test" and "Deploy" stages), which is used by many plugins to visualize or present Jenkins Pipeline status/progress.

Step

A single task. Fundamentally, a step tells Jenkins *what* to do at a particular point in time (or "step" in the process). For example, to execute the shell command `make` use the `sh` step: `sh 'make'`. When a plugin extends the Pipeline DSL, that typically means the plugin has implemented a new *step*.

Note :- implementing `stage` blocks in a Scripted Pipeline provides clearer visualization of each `stage's subset of tasks/steps in the Jenkins UI. Be aware that both stages and steps (above) are common elements of both Declarative and Scripted Pipeline syntax.

Agent slave naming Scheme

Slave Naming Scheme
<company>-<slave_type><number>

- **company:** The company who provided the slave
- **slave type:** The type of the slave. It could be build, pod, virtual, and test. If a new type of slave needs to be added, this must be confirmed with Releng.
- **number:** The slaves will be numbered based on which company provided the slave and slave type.

Here are the examples based on above scheme.

Slave Type	Examples
build	ericsson-build1 ericsson-build2
pod jumphost	intel-pod5 intel-pod6 lf-pod1 lf-pod2
virtual	ericsson-virtual1 lf-virtual1
test	huawei-test1 huawei-test2

Global Library

Library structure and contents

jenkins-shared-library
|____vars
|____src
|____resources
vars: This directory holds all the global shared library code that can be called from a pipeline. It has all the library files with a .groovy extension. It also supports .txt files for the documentation of shared library code. For example, if you have a file named maven-build.groovy, you can have a help file named maven-groovy.txt. In this file, you can write the respective shared library function help documentation in markdown format. The help file can be viewed from <your-jenkins-url>/pipeline-syntax/globals page.

src: It is a regular java source directory. It is added to the classpath during every script compilation. Here you can add custom groovy code to extend yous shared library code. Also, you can import existing Jenkins and plugins classes using an import statement.

There will be scenarios where your groovy DSL's will not be flexible enough to achieve some functionality. In this case, you can write custom groovy functions in src and call it in your shared library code.

resources: All the non-groovy files required for your pipelines can be managed in this folder. One such example is, you might need a common JSON template to make an API call during the build. This JSON template can be stored in the resources folder and called can be called in the shared library using libraryResource function.

Since these libraries will be globally usable, any Pipeline in the system can utilize functionality implemented in these libraries.
These libraries are considered "trusted:" they can run any methods in Java, Groovy, Jenkins internal APIs, Jenkins plugins, or third-party libraries. This allows you to define libraries which encapsulate individually unsafe APIs in a higher-level wrapper safe for use from any Pipeline. Beware that **anyone able to push commits to this SCM repository could obtain unlimited access to Jenkins**. You need the *Overall/RunScripts* perm ission to configure these libraries (normally this will be granted to Jenkins administrators).

Using libraries

Shared Libraries marked *Load implicitly* allows Pipelines to immediately use classes or global variables defined by any such libraries. To access other shared libraries, the `Jenkinsfile` needs to use the `@Library` annotation, specifying the library's name:

```
@Library('my-shared-library') _
/* Using a version specifier, such as branch, tag, etc */
@Library('my-shared-library@1.0') _
/* Accessing multiple libraries with one statement */
@Library(['my-shared-library', 'otherlib@abc1234']) _
```

The annotation can be anywhere in the script where an annotation is permitted by Groovy. When referring to class libraries (with `src/` directories) , conventionally the annotation goes on an `import` statement:

```
@Library('somelib')
import com.mycorp.pipeline.somelib.UsefulClass
```

If you are only interested in using global variables/functions (from the `vars/` directory), the syntax is quite simple:

```
library 'my-shared-library'
```

Library versions

The "Default version" for a configured Shared Library is used when "Load implicitly" is checked, or if a Pipeline references the library only by name, for example`@Library('my-shared-library') _`. If a "Default version" is**not**defined, the Pipeline must specify a version, for example `@Library('my-shared-library@master') _`.

When using the`library`step you may also specify a version:

```
library 'my-shared-library@master'
```

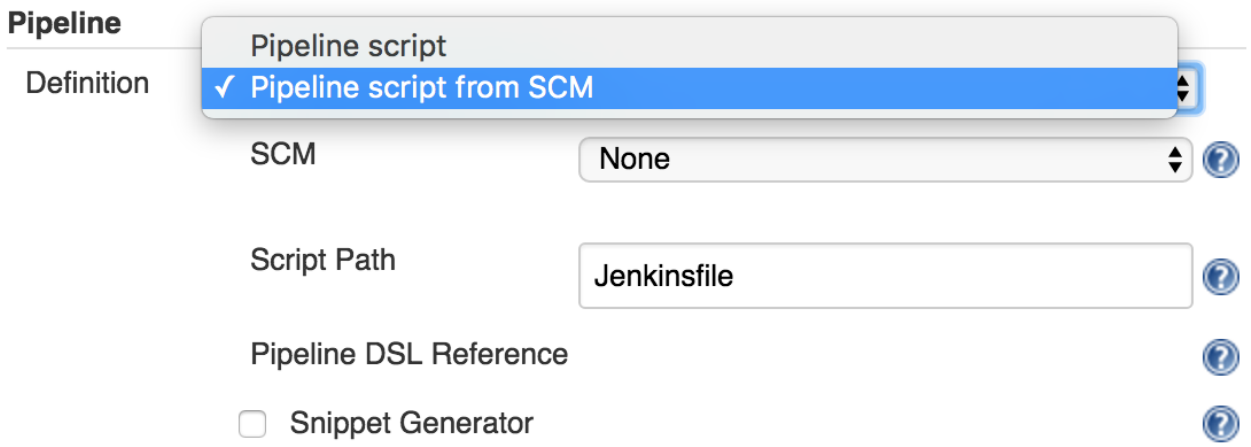
Since this is a regular step, that version could be *computed* rather than a constant as with the annotation; for example:

```
library "my-shared-library@$BRANCH_NAME"
```

Best Practices for Jenkins Pipeline

1. **Do: Develop your pipeline as code**

   Use the feature to store your Jenkinsfile in SCM then version and test it like you do other software.
   - Why? Treating your pipeline as code enforces good discipline and also opens up a new world of features and capabilities like multi-branch, pull request detection and organization scanning for GitHub and BitBucket.



- You should also call your Pipeline script the default name: Jenkinsfile and start the following script header, so your IDE, GitHub and other tooling recognize it as Groovy and enable code highlighting:

```
#!groovy
```

2. **Do: All work within a stage**

- Why? Stages are the logical segmentation of a pipeline. Separating work into stages allows separating your pipeline into distinct segments of work.

Example:

```
stage 'build'
//build
stage 'test'
//test
```

3. **Do: All material work within a node**

   Any material work within a pipeline should occur within a node block.

- Why? By default, the Jenkinsfile script itself runs on the Jenkins master, using a lightweight executor expected to use very few resources. Any material work, like cloning code from a Git server or compiling a Java application, should leverage Jenkins distributed builds capability and run an agent node.Example,

```
stage 'build'
node{
    checkout scm
    sh 'mvn clean install'
}
```

4. **Do: Acquire nodes within parallel steps**

Why? One of the main benefits of parallelism in a pipeline is: to do more material work (see Best Practice #4)! You should generally aim to acquire a node within the parallel branches of your pipeline. Example,

```
parallel 'integration-tests':{
    node('mvn-3.3'){ ... }
}, 'functional-tests':{
    node('selenium'){ ... }
}
```

5. **Don’t: Use input within a node block**

   While you can put an input statement within a node block, you definitely shouldn’t.

- Why? The input element pauses pipeline execution to wait for an approval - either automated or manual. Naturally these approvals could take some time. The node element, on the other hand, acquires and holds a lock on a workspace and heavy weight Jenkins executor - an expensive resource to hold onto while pausing for input. So, create your inputs outside your nodes.Example,

```
stage 'deployment'
input 'Do you approve deployment?'
node{
    //deploy the things
}
```

6. **Do: Wrap your inputs in a timeout**

Pipeline has an easy mechanism for timing out any given step of your pipeline. As a best practice, you should always plan for timeouts around your inputs.

- Why? For healthy cleanup of the pipeline, that's why. Wrapping your inputs in a timeout will allow them to be cleaned-up (i.e., aborted) if approvals don't occur within a given window.Example,

```
timeout(time:5, unit:'DAYS') {
    input message:'Approve deployment?', submitter: 'it-ops'
}
```

7. **Don't: Set environment variables with env global variable**

   While you can edit some settings in the env global variable, you should use the withEnv syntax instead.

- Why? because the env variable is global, changing it directly is discouraged as it changes the environment globally, so the withEnv syntax is recommended.Example,

```
withEnv(["PATH+MAVEN=${tool 'm3'}/bin"]) {
    sh "mvn clean verify"
}
```

8. **Do: Prefer stashing files to archiving**

Before the stash capability was added to Pipeline DSL, archives were the best way to share files between nodes or stages in a pipeline. If you just need to share files between stages and nodes of your pipeline, you should use stash/unstash instead of archive.

- Why? Stash and unstash are designed for sharing files, for example your application's source code, between stages and nodes. Archives, on the other hand, are designed for longer term file storage (e.g., intermediate binaries from your builds).

```
stash excludes: 'target/', name: 'source'
unstash 'source'
```