DECLARATIVE PIPELINE	SCRIPTED PIPELINE
New way of writing with simple groovy declaratives	Old way / Traditional way of pipeline code with scripted groovy syntax
Code is written in a file and is checked into git (SCM) and then added to jenkins job	Code is written directly in jenkins pipeline job which in UI interface
All code is defined inside pipeline block	Defined within a node block

DECLARATIVE PIPELINE

A valid Declarative pipeline must be defined with the "pipeline" sentence and include the next required sections like **AGENT, STAGES, STAGE, STEPS**

AGENT

- An agent is a directive that can run multiple builds with only one instance of Jenkins.
- This feature helps to distribute the workload to different agents and execute several projects within a single Jenkins instance.
- It instructs Jenkins to **allocate an executor** for the builds.
- A single agent can be specified for an entire pipeline or specific agents can be allotted to execute each stage within a pipeline. Few of the parameters used with agents are:

Any Runs the pipeline/ stage on any available agent.

None This parameter is applied at the root of the pipeline and it indicates that there is no global agent for the entire pipeline and each stage must specify its own agent.

Label Executes the pipeline/stage on the labelled agent.

Docker This parameter uses docker container as an execution environment for the pipeline or a specific stage. In the below example I'm using docker to pull an ubuntu image. This image can now be used as an execution environment to run multiple commands.

```
Pipeline {
      Agent {
           Docker {
                    Image 'ubuntu'
                  }
            }
          }
```

STAGES & STAGE

- This block contains all the work that needs to be carried out. The work is specified in the form of stages.
- There can be more than one stage within this directive. Each stage performs a specific task.

```
pipeline {
     agent any
     stages {
          stage ('build') {
               ...
          }
```

```
            stage ('test: integration-&-quality') {
                ...
            }
            stage ('test: functional') {
                ...
            }
            stage ('test: load-&-security') {
                ...
            }
            stage ('approval') {
                ...
            }
            stage ('deploy:prod') {
                ...
            }
        }
    }
```

STEPS

- A series of steps can be defined within a stage block.
- Steps are carried out in sequence to execute a stage.
- There must be at least one step within a step's directive.

```
Pipeline {
    Agent any
    Stages {
        Stage ('Build') {
            Steps {
                Echo 'Build phase ……..'
            }
        }
    }
```

AVAILABLE DIRECTIVES:

ENVIRONMENT (Defined at stage or pipeline level)

- This directive can be both defined at stage or pipeline level.
- This will determine the scope of its definitions means declared environment variables.
- Definitions of "environment" at the "pipeline" level will be valid for all of the pipeline steps.
- Definitions of "environment" at the within a "stage" will only be valid for the particular stage.

At the "pipeline" level:

```
pipeline {
    agent any
    environment {
        OUTPUT_PATH = './outputs/'
    }
    stages {
```

```
            stage ('build') {
                 ...
            }
        ...
        }
}
```

Here, “environment” is used at a “stage” level:

```
pipeline {
     agent any
     stages {
          stage ('build') {
     environment {
          OUTPUT_PATH = './outputs/'
     }
     ...
          }
          ...
     }
}
```

Setting environment variables dynamically (Optional)

In the case where environment variable need to be set dynamically at run time this can be done with the use of a shell scripts (sh),

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any
    environment {
        // Using returnStdout
        CC = """${sh(
                returnStdout: true,
                script: 'echo "clang"'
            )}"""
        // Using returnStatus
        EXIT_STATUS = """${sh(
                returnStatus: true,
                script: 'exit 1'
            )}"""
    }
    stages {
        stage('Example') {
            environment {
                DEBUG_FLAGS = '-g'
            }
            steps {
                sh 'printenv'
            }
    } } }
```

INPUT (DEFINED AT STAGE LEVEL)

The "input" directive is defined at a stage level and provides the functionality to prompt for an input. The stage will be paused until a user manually confirms it.

The following configuration options can be used for this directive:

- message: This is a required option where the message to be displayed to the user is specified.
- id: Optional identifier for the input. By default, the "stage" name is used.
- ok: Optional text for the Ok button.
- submitter: Optional list of users or external group names who are allowed to submit the input. By default, any user is allowed.
- submitterParameter: Optional name of an environment variable to set with the submitter name, if present.
- parameters: Optional list of parameters to be provided by the submitter.

Simple input:

```
pipeline {
     agent any
     stages {
     stage ('build') {
                    input ('press continue to continue')
                     }
     steps {
          echo "User: ${username} said Ok."
     }
        }}
```

Detailed input:

```
pipeline {
     agent any
     stages {
          stage ('build') {
     input{
          message "Press Ok to continue"
          submitter "user1,user2"
          parameters {
               string(name:'username', defaultValue:
'user', description: 'Username of the user pressing Ok')
          }
     }
     steps {
          echo "User: ${username} said Ok."
     }
          }
     }
}
```

OPTIONS (DEFINED AT STAGE OR PIPELINE LEVEL)

Defined at pipeline level, this directive will group the specific options for the whole pipeline. The available options are:

- buildDiscarder
- disableConcurrentBuilds
- overrideIndexTriggers
- skipDefaultCheckout
- skipStagesAfterUnstable
- checkoutToSubdirectory
- newContainerPerStage
- timeout
- retry
- timestamps

```
pipeline {
     agent any
          options {
                    retry(3)
                  }
          stages {
                    ...
                 }
         }
```

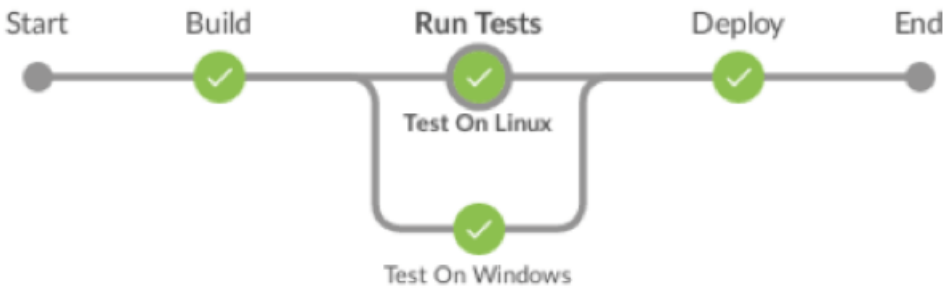
PARALLEL

Jenkins pipeline Stages can have other stages nested inside that will be executed in parallel. This is done by adding the "parallel" directive to your script. An example on how to use it is provided:

Both scripts will run the tests on different nodes since they run specific platform tests. Parallelism can also be used to simultaneously run stages on the same node by the use of multithreading, if your Jenkins server has enough CPU.

Some restrictions apply when using parallel stages:

- A stage directive can have either a parallel or steps directive but not both.
- A stage directive inside a parallel one cannot nest another parallel directive, only steps are allowed.
- Stage directives that have a parallel directive inside cannot have "agent" or "tools" directives defined.




```
stage('run-parallel-branches') {
     steps {
       parallel(
               a: {
                    echo "Tests on Linux"
                  },
               b: {
                    echo "Tests on Windows"
                  }
               )
          }
     }
```

New syntax

```
pipeline {
     agent none
     stages {
          stage('Run Tests') {
          parallel {
               stage('Test On Windows') {
                    agent { label "windows" }
                    steps {
                              bat "run-tests.bat"
                          }
                    }
```

```
                    stage('Test On Linux') {
                         agent { label "linux" }
                         steps {
                                   sh "run-tests.sh"
                              }
                         }
                    }
               }}}
```

PARAMETERS

This directive allows you to define a list of parameters to be used in the script. Parameters should be provided once the pipeline is triggered. It should be defined at a "pipeline" level and only one directive is allowed for the whole pipeline.

String and Boolean are the valid parameter types that can be used.

```
pipeline {
agent any
parameters {
          string(name: 'user', defaultValue: 'John',
     description: 'A user that triggers the pipeline')
          }
     stages {
          stage('Trigger pipeline') {
          steps {
               echo "Pipeline triggered by ${params.USER}"
                }
}}}
```

POST

Post sections can be added at a pipeline level or on each stage block and sentences included in it are executed once the stage or pipeline completes. Several post-conditions can be used to control whether the post executes or not:

- always: Steps are executed regardless of the completion status.
- changed: Executes only if the completion results in a different status than the previous run.
- fixed: Executes only if the completion is successful and the previous run failed
- regression: Executes only if current execution fails, aborts or is unstable and the previous run was successful.
- aborted: Steps are executed only if the pipeline or stage is aborted.
- failure: Steps are executed only if the pipeline or stage fails.
- success: Steps are executed only if the pipeline or stage succeeds.
- unstable: Steps are executed only if the pipeline or stage is unstable.

```
pipeline {
     agent any
     stages {
          stage('Some steps') {
          steps {
```

```
                ...
                }
}}
post {
always {
     echo “Pipeline finished”
     bat ./performCleanUp.bat
          }
}}
```

SCRIPT

This step is used to add Scripted Pipeline sentences into a Declarative one, thus providing even more functionality. This step must be included at "stage" level.

Several times blocks of scripts can be utilized on different projects. These blocks allow you to extend Jenkins functionalities and can be implemented as shared libraries. More information on this can be found at Jenkins shared libraries. Also, shared libraries can be imported and used into the "script" block, thus extending pipeline functionalities.

Next we will provide sample pipelines. The first one will only have a block with a piece of Scripted pipeline text, while the second one will show how to import and use shared libraries:

```
pipeline {
     agent any
     stages {
          stage('Sample') {
          steps {
               echo "Scripted block"
          script {

                  }
}}}}
```

TOOLS

The “tools” directive can be added either at a pipeline level or at the stage one. It allows you to specify which maven, jdk or gradle version to use on your script. Any of these tools, the three supported at the time of writing, must be configured on the “Global tool configuration” Jenkins menu.

Also, Jenkins will attempt to install the listed tool (if it is not installed yet). By using this directive, you can make sure a specific version required for your project is installed.

```
pipeline {
agent any
tools {
     maven 'apache-maven-3.0.1'
}
stages {
... }}
```

TRIGGERS

Triggers allows Jenkins to automatically trigger pipelines by using any of the available ones:

- cron: By using cron syntax, it allows to define when the pipeline will be re-triggered.
- pollSCM: By using cron syntax, it allows you to define when Jenkins will check for new source repository updates. The Pipeline will be re-triggered if changes are detected. (Available starting with Jenkins 2.22).
- upstream: Takes as input a list of Jenkins jobs and a threshold. The pipeline will be triggered when any of the jobs on the list finish with the threshold condition.

Sample pipelines with the available triggers are shown next:

```
pipeline {
     agent any
     triggers {
          //Execute weekdays every four hours starting at
               minute 0
               cron('0 */4 * * 1-5')
               }
     stages {
               ...
          }
      }

pipeline {
     agent any
     triggers {
          //Query repository weekdays every four hours
            starting at minute 0
               pollSCM('0 */4 * * 1-5')
             }
     stages {
               ...
          }
      }

pipeline {
     agent any
     triggers {
          //Execute when either job1 or job2 are
            successful
     upstream(upstreamProjects: 'job1, job2',  threshold:
                    hudson.model.Result.SUCCESS)
               }
     stages {
               ...
          }
      }
```

WHEN

Pipeline steps could be executed depending on the conditions defined in a "when" directive. If conditions match, the steps defined in the corresponding stage will be run. It should be defined at a stage level.

For example, pipelines allow you to perform tasks on projects with more than one branch. This is known as multibranched pipelines, where specific actions can be taken depending on the branch name like "master", "feature*", "development", among others. Here is a sample pipeline that will run the steps for the master branch:

```
pipeline {
     agent any
     stages {
               stage ('Deploy stage') {
          when {
               branch 'master'
               }
     steps {
               echo 'Deploy master to stage'
               ...
           }
     }}}
```

FINAL JENKINS DECLARATIVE PIPELINE TIPS:

- Declarative pipelines syntax errors are reported right at the beginning of the script. This is a nice functionality since you will not waste time until a step fails to realize there is a typo or misspelling.
- As already mentioned, pipelines can be written either declarative or scripted. Indeed, the declarative way is built on top of the scripted way making it easy to extend as explained, by adding script steps.
- Jenkins pipelines are being widely used on CI/CD environments. Using either declarative or scripted pipelines has several advantages. In this post we presented all the syntax elements to write your declarative script along with samples. As we already stated, the declarative way offers a much more friendly syntax with no Groovy knowledge required.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building..'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing..'
            }
        }
```

```
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }
}
```

EPLANATION

1. **BUILD**

For many projects the beginning of "work" in the Pipeline would be the "build" stage. Typically, this stage of the Pipeline will be where source code is assembled, compiled, or packaged. The Jenkinsfile is **not** a replacement for an existing build tool such as GNU/Make, Maven, Gradle, etc, but rather can be viewed as a glue layer to bind the multiple phases of a project's development lifecycle (build, test, deploy, etc) together.

Jenkins has a number of plugins for invoking practically any build tool in general use, but this example will simply invoke make from a shell step (sh). The sh step assumes the system is Unix/Linux-based, for Windows-based systems the bat could be used instead.

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'make'
                archiveArtifacts artifacts: '**/target/*.jar',
                                 fingerprint: true
            }
        }
    }
}
```

- The sh step invokes the make command and will only continue if a zero-exit code is returned by the command. Any non-zero exit code will fail the Pipeline.
- archiveArtifacts captures the files built matching the include pattern (**/target/*.jar) and saves them to the Jenkins master for later retrieval.

2. **TEST**

Running automated tests is a crucial component of any successful continuous delivery process. As such, Jenkins has a number of test recording, reporting, and visualization facilities provided by a number of plugins. At a fundamental level, when there are test failures, it is useful to have Jenkins record the failures for reporting and visualization in the web UI. The example below uses the junit step, provided by the JUnit plugin.

In the example below, if tests fail, the Pipeline is marked "unstable", as denoted by a yellow ball in the web UI. Based on the recorded test reports, Jenkins can also provide historical trend analysis and visualization.

```
Jenkinsfile (Declarative Pipeline)
```

```
pipeline {
    agent any

    stages {
        stage('Test') {
            steps {
                /* `make check` returns non-zero on test
failures,
                * using `true` to allow the Pipeline to continue
nonetheless
                */
                sh 'make check || true'
                junit '**/target/*.xml'
            }
        }
    }
}
```

3. DEPLOY

Deployment can imply a variety of steps, depending on the project or organization requirements, and may be anything from publishing-build artifacts to an Artifactory server, to pushing code to a production system.

At this stage of the example Pipeline, both the "Build" and "Test" stages have successfully executed. In essence, the "Deploy" stage will only execute assuming previous stages completed successfully, otherwise the Pipeline would have exited early.

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any

    stages {
        stage('Deploy') {
            when {
              expression {
                currentBuild.result == null ||
currentBuild.result == 'SUCCESS'
              }
            }
            steps {
                sh 'make publish'
            }
        }
    }
}
```

REAL WORLD SCENARIOS

1. **HANDLING CREDENTIALS**

   For secret text, usernames and passwords, and secret files

   Jenkins' declarative Pipeline syntax has the credentials() helper method (used within the environment directive) which supportssecret text, username and password, as well as secret file credentials. If you want to handle other types of credentials, refer to the For other credential types section (below).

   In this example, two secret text credentials are assigned to separate environment variables to access Amazon Web Services (AWS). These credentials would have been configured in Jenkins with their respective credential IDs jenkins-aws-secret-key-id and jenkins-aws-secret-access-key.

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent {
        // Define agent details here
    }
    environment {
        AWS_ACCESS_KEY_ID     = credentials('jenkins-aws-secret-
key-id')
        AWS_SECRET_ACCESS_KEY = credentials('jenkins-aws-secret-
access-key')
    }
    stages {
        stage('Example stage 1') {
            steps {
                //
            }
        }
        stage('Example stage 2') {
            steps {
                //
            }
        }
    }
}
```

- BITBUCKET_COMMON_CREDS - contains a username and a password separated by a colon in the format username:password.
- BITBUCKET_COMMON_CREDS_USR - an additional variable containing the username component only.
- BITBUCKET_COMMON_CREDS_PSW - an additional variable containing the password component only.

```
environment {
    BITBUCKET_COMMON_CREDS = credentials('jenkins-bitbucket-
common-creds')
}
```

username and password credentials are assigned to environment variables to access a Bitbucket repository in a common account or team for your organization; these credentials would have been configured in Jenkins with the credential ID

Handling parameters

Assuming that a String parameter named "Greeting" has been configuring in the Jenkinsfile, it can access that parameter via ${params.Greeting}:

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any
    parameters {
        string(name: 'Greeting', defaultValue: 'Hello',
description: 'How should I greet the world?')
    }
    stages {
        stage('Example') {
            steps {
                echo "${params.Greeting} World!"
            }
        }
    }
}
```

Handling failure

Declarative Pipeline supports robust failure handling by default via its post section which allows declaring a number of different "post conditions" such as: always, unstable, success, failure, and changed.

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any
    stages {
        stage('Test') {
            steps {
                sh 'make check'
            }
        }
    }
    post {
        always {
            junit '**/target/*.xml'
        }
        failure {
            mail to: team@example.com, subject: 'The Pipeline
failed :('
        }
    }
}
```