**Chapter 7. Declarative Pipelines**

In this part of the book, we’re going to be talking about another evolution in Jenkins pipelines—Declarative Pipelines. Declarative Pipelines allow users to define a pipeline in a way similar to how they would define jobs in the traditional Jenkins web forms. By this we mean:

* There is a well-defined, enforced structure. (You can think of this like the sections on the pages of a Jenkins web form.)
* Defining a pipeline section is more about declaring the high-level steps/goals than defining the logic to accomplish it. (This is similar to filling in the fields in a Jenkins web form.)
* Familiar Jenkins processing constructs are provided and don’t have to be emulated with programming. (For example, you have a way to do post-build processing and send notifications, as opposed to having to use try-catch-finally Groovy programming to handle this.)
* All of the above enable better validation and error checking. (Errors are identified and presented in the context of the expected structure and keywords, not just Groovy tracebacks.)

These features distinguish Declarative Pipelines from the alternative way of creating a pipeline that ties DSL steps and sections together with programming constructs (assignments, conditionals, etc.)—essentially writing a program. That style of free-form coding for a pipeline is what we call a “Scripted Pipeline.”

Both types of pipelines have their place, with advantages and disadvantages to each. Broadly speaking, Declarative Pipelines are easiest for someone new to using the pipeline functionality. This is because they more closely resemble what was done and available in the web forms, and they have clearer, more contextual validation and error checking.

Scripted Pipelines provide more flexibility and the ability to mix in programming constructs to execute logical flows, decision handling, assignments, etc. that are not available in Declarative Pipelines. For more experienced users or advanced applications, Scripted Pipelines can be the best option.

It is also worth noting that not all plugins that support Scripted Pipelines have interfaces and flows that support Declarative Pipelines directly.

One last general note about Declarative Pipelines: you may be wondering how support for them is integrated with Jenkins. Like nearly every piece of additional functionality in Jenkins, they’re supported via plugins. The set of plugins that support Declarative Pipelines and the new Blue Ocean interface (described in [Chapter 9](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch09.html#CH_The_Blue_Ocean_Interface)) are largely tied together.

Now, let’s start diving into the world of Declarative Pipelines by taking a look at the motivation behind them.

# Motivation

To understand why we can benefit from another way to structure pipelines in Jenkins, it’s helpful to understand some of the shortcomings specifically associated with the traditional Scripted Pipeline creation and model.

## Not Intuitive

As we’ve discussed, moving from a web interface (with specific forms, help buttons, and UI elements that guide you in setting up jobs) to creating scripts, is not intuitive. One key part of the original UI job pages was the separation into sections, such as post-build processing, that guided users through the various phases. When moving to scripts, the elements for the different phases are available, but it’s not clear how to structure or order them out of the box. Worse, some familiar processes don’t have corresponding constructs in the nondeclarative DSL.

## Getting Groovy

While it’s not a requirement to be able to program in Groovy to create DSL scripts, sometimes it can feel that way to users. For missing functionality, Groovy constructs may be the only alternative. Verification such as syntax checking is done at the Groovy level. Also, errors are surfaced as Groovy errors (tracebacks) and not as DSL-specific ones.

## Additional Assembly Required

Building on a point raised earlier, additional code can be required to get the familiar Jenkins constructs we had in the web forms version. For example, the simple task of sending email after a failed build has to be handled with something like a try-catch-finally construct, instead of the familiar built-in post-build functionality.

The following code highlights the contrast between sending emails after a failure in a Scripted Pipeline versus the way this was typically handled in traditional Jenkins, as shown in

node {

**try** {

sendEmailStarted()

stage('Source') {...}

stage('Build') {...}

...

sendEmailSuccess()

} **catch** (err) {

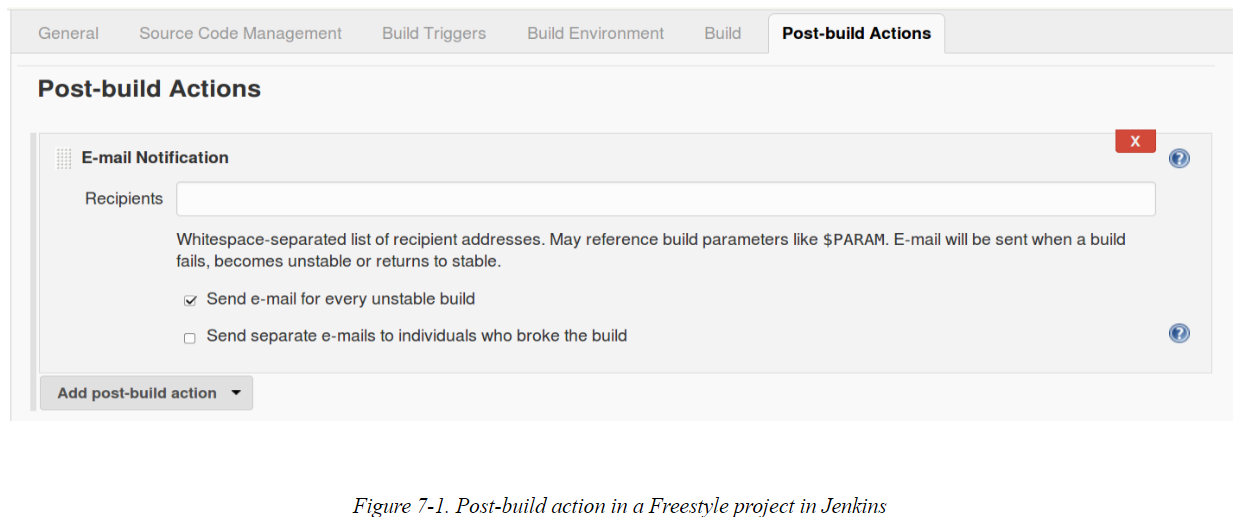
currentBuild.result = "FAILED"

sendEmailFail()

**throw** err

}

}



For these and other reasons, the CloudBees staff, as part of the Jenkins community, created an expanded DSL and simpler environment for programming pipelines. Note that Declarative Pipelines are still pipelines-as-code. We are still using the same environment to code our pipelines; we enter the Declarative Pipeline syntax in the Pipeline tab script window or in Jenkinsfiles, just as we would for any other pipeline code. However, as we’ve noted, the Declarative Pipeline syntax is more structured and the environment provides improved DSL-specific validation and error checking. We’ll explore that structure next and discuss script checking and error reporting later in this chapter.

# The Structure

A declarative is made up of an outer block that contains directives and sections. Each section in turn can contain other sections, directives, and steps, and in some cases conditionals. The distinction between blocks, sections, and directives is somewhat arbitrary, but since they’re used in the formal documentation, we’ll define those and the other terms more clearly.

## Block

A block here is really just any set of code that has a beginning and end. In Groovy, this translates to a closure (a section of code where the beginning and end are bracketed with { and }).

While many parts of the pipeline are technically blocks, that term is used primarily to describe the overall pipeline block, which contains all of the code associated with a Declarative Pipeline.

It looks like this:

pipeline {

*// code in declarative syntax*

}

## Section

Sections in a Declarative Pipeline are a way to collect items that need to be executed at particular points during the overall flow of the pipeline. The grouped items may include directives, steps, and conditionals (defined in the following sections). As the pipeline is executed, it looks for sections to define the various groupings and phases.

Currently, there are three areas we refer to as sections:

stages

This section wraps all of the individual stage definitions (directives) that define the main body and logic for the pipeline.

steps

This section wraps a set of DSL steps within a stage definition. It serves to separate the collection of steps from other items within a stage, such as environment definitions.

posts

This section wraps around steps and conditions to be done or checked at the end of a pipeline run or at the end of a stage.

An example layout with sections identified in bold font is shown here:

pipeline {

agent any

**stages** {

stage('name1') {

**steps** {

...

}

**post** {

...

}

}

stage('name2') {

**steps** {

...

}

}

}

**post** {

...

}

## Directives

A directive can be thought of as a statement or block of code that does any of the following in a pipeline:

Defines values

An example of this is the agent directive, which allows us to specify a node or container to run an entire pipeline or a stage in. If we wanted to run our pipeline on a node named worker, we could use agent ('worker').

Configures behavior

An example of this is the triggers directive that lets us configure how often Jenkins checks for source updates or triggers our pipeline. If we wanted it to retrigger our pipeline at 7 a.m. every weekday, we could use triggers { cron ('0 7 0 0 1-5') }.

Specifies actions to be done

An example of this is the stage directive, which is expected to have a steps section containing DSL steps to be executed.

## Steps

The label steps itself is a section title with in a stage of the pipeline. However, within the steps section, we can have any valid DSL statement, such as git, sh, echo, etc. You can think of a step here as corresponding to one of these statements.

## Conditionals

Conditionals supply a condition or criteria under which an action should occur. These are optional. There are two cases you may encounter/use:

* When: Strictly speaking, this is a directive. It resides within a stage definition and defines criteria forwhether or not a stage should be executed. For example:

stage ('build') {

when {

branch 'foo'

}

<steps>

}

* *Conditions* blocks in the post section that define the criteria for doing post-processing. The criteria (conditions) here refer to the status of the build, such as success or failure.

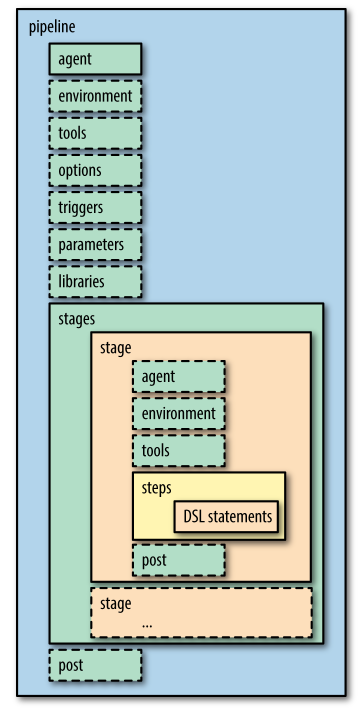
Now that we have a basis for the terminology, let’s look at the different building blocks in more detail.

# The Building Blocks

In this section we’ll cover specifics on each of the sections and directives available to you to use in a Declarative Pipeline, including syntax, parameters, and example usage.

At a high level, the blocks stack up as shown in [Figure 7-2](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#fig_oview_DPS). Here, each box represents the specific section or directive indicated by its text, and their placement indicates where they can be in the Declarative Pipeline structure. For example, pipeline is the outermost block, and all of your other sections and directives must be inside of it.

Those with dotted lines around them are optional in that part of the structure. Those with solid lines are required in that part of the structure. Note that there are some directives that can occur at both the pipeline and stage level. They may be required in one area and optional in another.



## pipeline

The pipeline block is required in a Jenkins Declarative Pipeline. It is the outermost section and signals that this is a Pipeline project. The syntax is simply pipeline {} with the rest of the code within the closure:

pipeline {

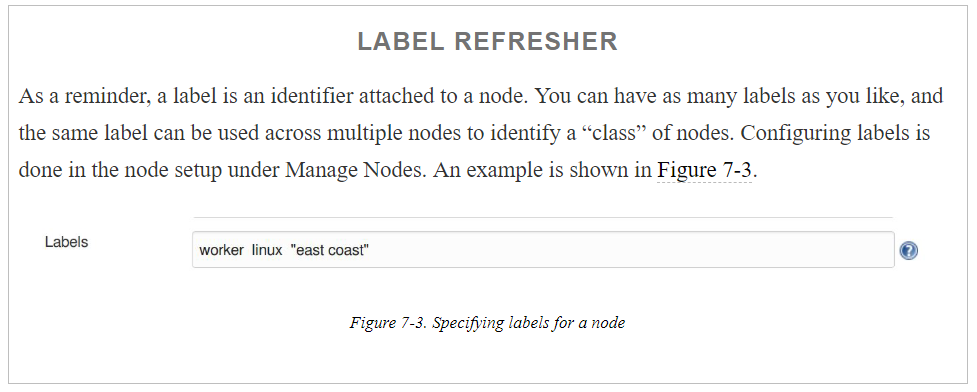
*// pipeline code*

}

## agent

The agent directive specifies where the entire pipeline or a specific stage runs. This is similar to how the nodedirective is used in Scripted Pipelines. In fact, you can reasonably think of an agent as a node, except that the master node is not an agent.

An agent directive near the top of the pipeline block is required as a “default” place for execution. However, individual agent directives can optionally be specified at the beginning of individual stages to indicate wherethe code in those stages should be run.



What the agent directive actually does is indicate which (if any) nodes to use in the execution of the pipeline or stage. It does this by mapping the argument supplied to it to the label(s) specified for the nodes in your Jenkins system. The format of the argument can be a single predefined type, an indicator with a specific label, or a label block with additional characteristics, such as for Docker containers. The possible options are summarized in the following sections.

agent any

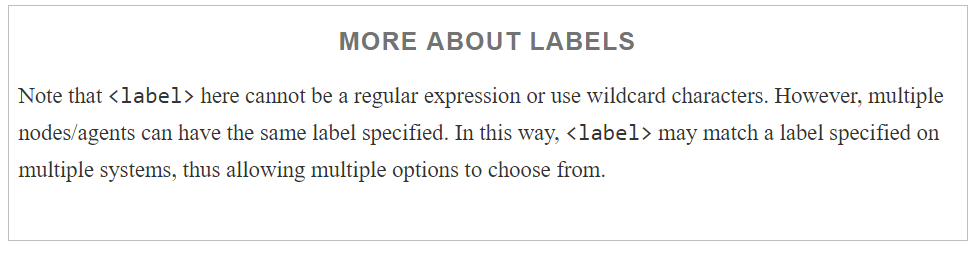
This syntax tells Jenkins that the pipeline or stage can run on *any* agent that is defined, without regard to what label it has.

agent none

When used at the top level, this indicates that we are not specifying an agent globally for the pipeline. The implication is that an agent will be specified, if needed, for individual stages.

agent { label “<label>”}

This indicates that the pipeline or stage can run on any agent that has the label <label>.



### LABELS AND CUSTOM WORKSPACES

A recent addition to the label syntax for agents allows us to specify a custom workspace for a pipeline or stage. Given an agent definition, we can include the customWorkspace directive to specify where the workspace that the agent uses should live. The syntax looks like this:

agent {

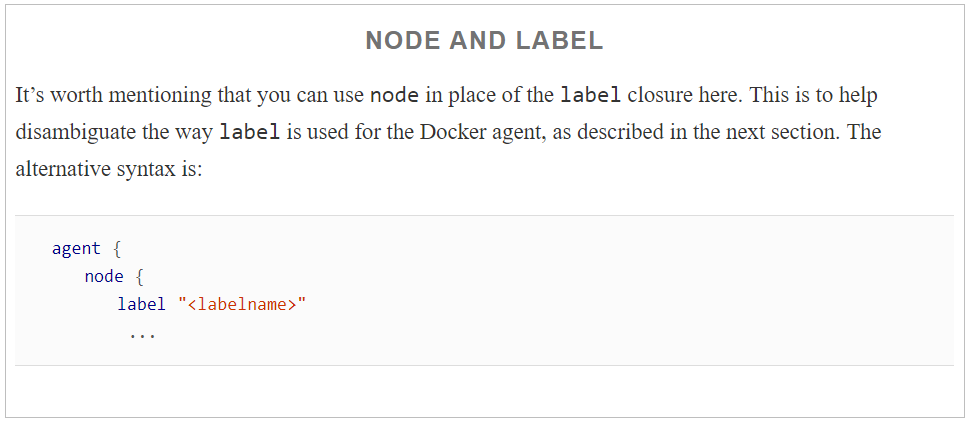
label {

label "<labelname>"

customWorkspace "<desired directory>"

}

}



### AGENTS AND DOCKER

The final agent options we’ll look at are Docker containers. There are two shorthand ways to get a Docker image—specifying an existing image or creating an image from a Dockerfile—in the agent declaration. Alternatively, the longer version of the declaration can be used to specify additional elements, such as a node to use for thecontainer, and arguments for the container.

First, we’ll look at the formats for using an existing Docker image:

agent { docker '<image>' }

This short syntax tells Jenkins to pull the given image from Docker Hub and run the pipeline or stage in acontainer based on the image, on a dynamically provisioned node.

agent { docker { <elements> } }

This long syntax allows for defining more specifics about the Docker agent. There are three additional elements that you can add in the declaration (within the { } block):

image '<image>'

Like the short form, this tells Jenkins to pull the given image and use it to run the pipeline code.

label '<label>'

If this element is present in the declaration, it tells Jenkins to instantiate the container and “host” it on anode matching <label>.

args '<string>'.

If this element is present in the declaration, it tells Jenkins to pass these arguments to the Docker container; the syntax here should be the same as you would normally pass to a Docker container.

Here’s an example declaration using the long form:

agent {

   docker {

image "image-name"

      label "worker-node"

      args "-v /dir:dir"

   }

}

agent { dockerfile true }

This short syntax is intended to be used when you have a source code repository, that you retrieve, that has a Dockerfile in its root (note that dockerfile here is a literal). In that case, this will tell Jenkins to build a Docker image using that Dockerfile, instantiate a container, and then run the pipeline (or the stage’s code if run in a stage) in that container.

agent { dockerfile { <elements> } }

This long syntax allows for defining more specifics about the Docker agent you are trying to create from a Dockerfile. There are three additional elements that you can add in the declaration (within the { } block):

filename '<path to dockerfile>'

This allows for specifying an alternate path to a Dockerfile, including a different name. Jenkins will try to build an image from the Dockerfile, instantiate a container, and use it to run the pipeline code.

label '<label>'

If this element is present in the declaration, it tells Jenkins to instantiate the container and “host” it on a node matching <label>.

args '<string>'

If this element is present in the agent Dockerfile declaration, it tells Jenkins to pass these arguments to the Docker container; the syntax here should be the same as you would normally pass to a Docker container.

An example of specifying a Docker agent via a Dockerfile using the long form is shown here:

agent {

dockerfile {

      filename "<subdir/dockerfile name>"

      label "<agent label>"

      args "-v /dir:dir"

   }

}

#### **Using the same node for Docker and non-Docker stages**

There is one other aspect associated with using Docker agents. Suppose you define a particular non-Docker agent at the top of your pipeline:

pipeline {

agent {label 'linux'}

Later, in a particular stage, you want to run the code in a Docker container—but you also want to use the same node and workspace that you defined for the pipeline. To enable this, the pipeline has a directive you can use with the Docker specification: reuseNode. It would look something like the following in practice:

stage 'abc' {

agent {

docker {

image 'ubuntu:16.6'

reuseNode **true**

This tells Jenkins to reuse the same node and workspace that were defined for the original pipeline agent to “host” the resulting Docker container.

Next, we’ll look at how to configure environment values for a pipeline.

## **environment**

This is an optional directive for your Declarative Pipeline. As the name implies, this directive allows you to specify names and values for environment variables that are then accessible within the scope of your pipeline. Like agent, you can have an instance of environment in the main pipeline definition and/or in individual stages. An environment definition in the top-level pipeline block will make the variable accessible to all steps in the pipeline. An environment definition within a stage will make the variable accessible to only the scope of the stage.

Here is an example of defining an environment variable in this way:

environment {

               TIMEZONE = "eastern"

}

Environment variable definitions can also incorporate variables that are already defined. The syntax for this isjust to include the existing variable in the definition string in ${<variable>}:

environment {

               TIMEZONE = "eastern"

               TIMEZONE\_DS = "${TIMEZONE}\_daylight\_savings"

}

### CREDENTIALS AND ENVIRONMENT VARIABLES

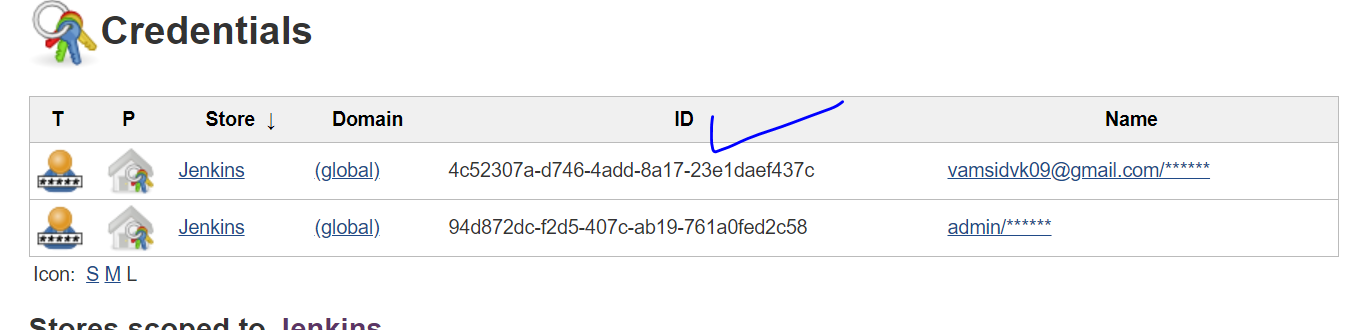
We talked in [Chapter 5](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch05.html#CH_Access_and_Security) about the different kinds of credentials that can be used with pipelines. Each of those methods required the identifier of a set of credentials that had been defined in Jenkins. In the environmentblock, you can assign a global variable to a particular credentials ID. Then you can use that variable throughoutyour pipeline in place of the ID. This can simplify things if you need to specify the ID in multiple places. The syntax is to assign the variable name to the string credentials('<credentials-id>'). For example:

environment {

ADMIN\_USER = credentials('admin-user')

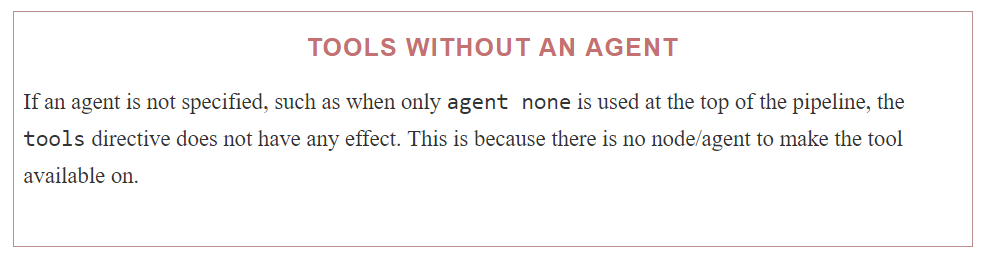
}

In this case, we would have admin-user previously defined as the specified ID for some set of credentials. If you hadn’t explicitly specified a named string (admin-user) as the ID, you would use the identifying string that Jenkins automatically generates during the creation of the credentials.

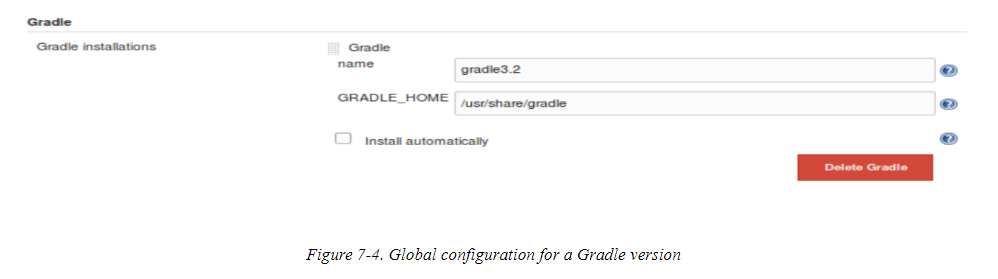


## **tools**

Jenkins users are familiar with using the Global Tool Configuration screen to configure versions, paths, and installers for tools. Once configured there, the tools directive allows us to specify which of these we want to have auto installed and made available in the path on the agent we’ve chosen.



For example, suppose we had the configuration shown in [Figure 7-4](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#fig_glob_config_Gradle_vers).



Then, in our tools block, we could refer to Gradle via:

tools {

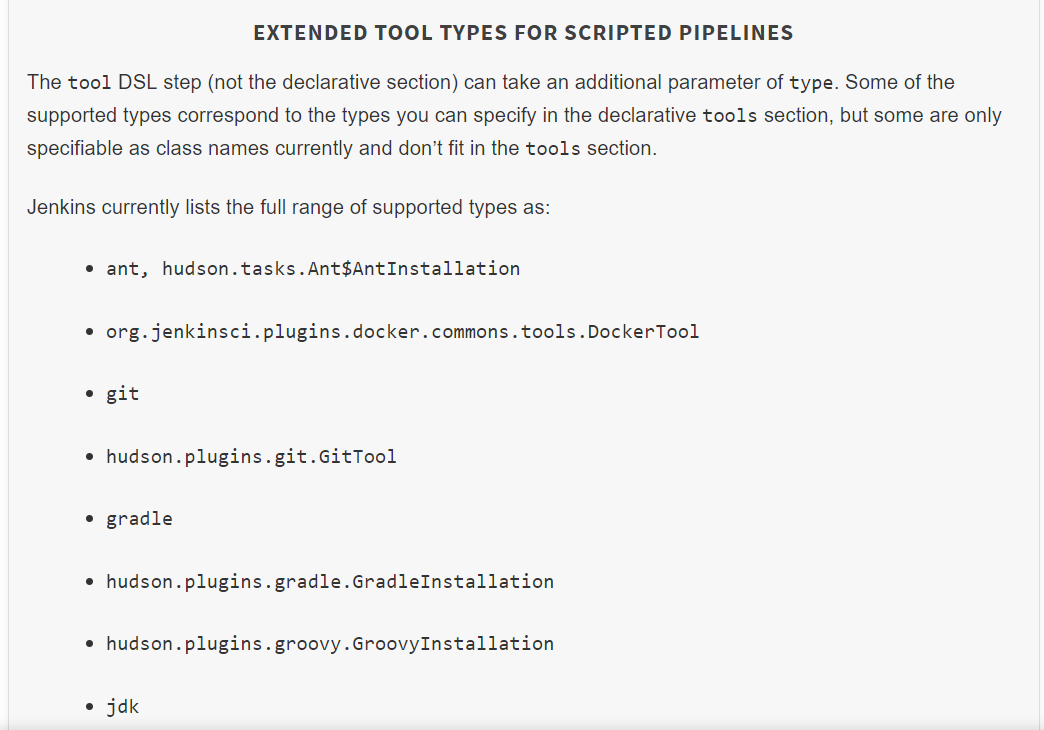
   gradle "gradle3.2"

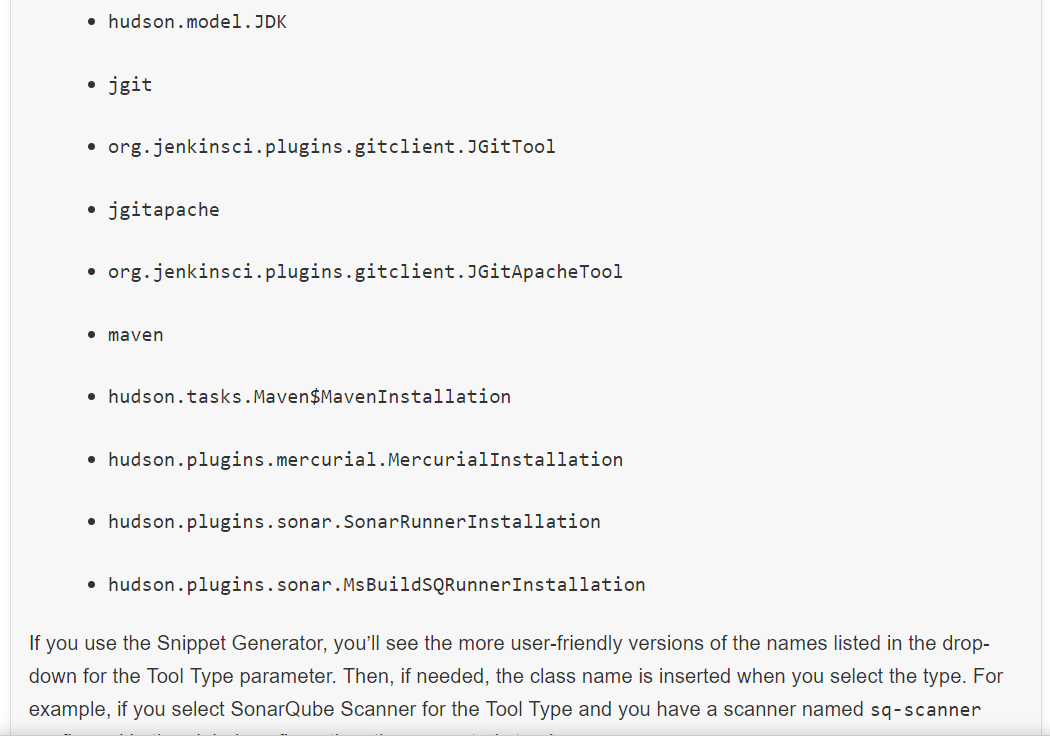
}

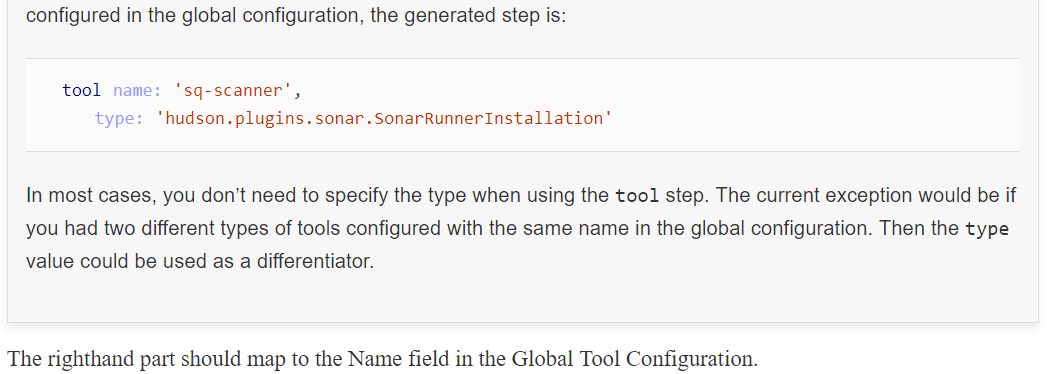
The lefthand part of this declaration is a specific string defined in the pipeline model. As of this writing, the valid tool types you can specify in declarative syntax are:

* ant
* git
* gradle
* jdk
* jgit
* jgitapache (JGit with Apache HTTP client)
* maven

Attempts to use other types that are not yet valid will result in an “Invalid tool type” error when running your pipeline.







Once this is set up, the tool is autoinstalled and put on the path. We can then simply use the string gradle in place of the GRADLE\_HOME path in our pipeline steps and Jenkins will map it back to this Gradle installation on our system. For example:

steps {

sh 'gradle clean compile'

}

Also, it’s worth noting that the tools directive can use the value of a parameter if you need to input a particular version to use. Here’s an example:

pipeline {

    agent any

    parameters {

       string(name: 'gradleTool', defaultValue: 'gradle3',

       description: 'Gradle Version')

    }

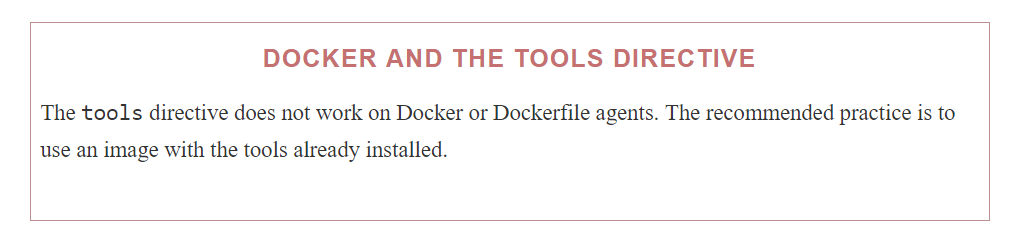
    tools {

       gradle  "${params.gradleTool}"

    }

Just keep in mind that there is currently a limitation with the declarative syntax such that Jenkins doesn’t recognize that a build requires a parameter the first time the pipeline is run.

tools is another directive that can be used either in the pipeline block or separately in a stage.

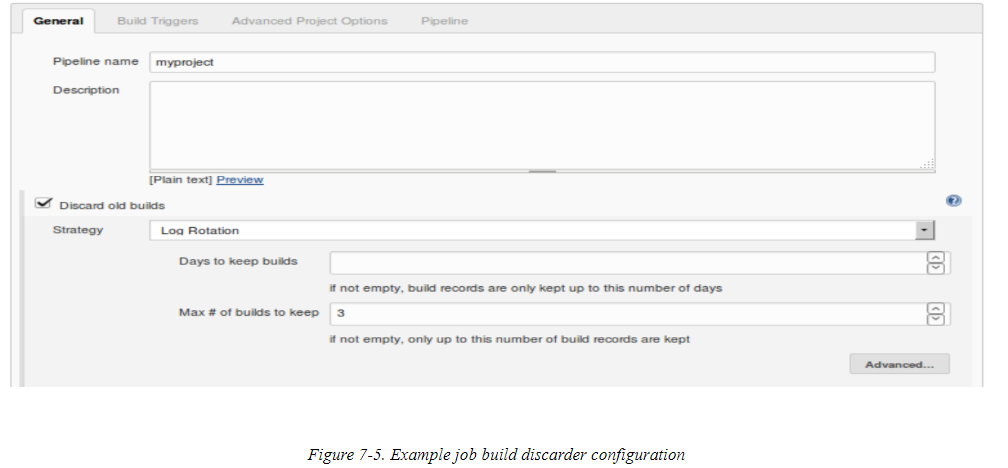


In addition to the tools directive to allow us to access the globally defined tools, we also have the options directive to allow us to set project-level options.

## **options**

This directive can be used to specify properties and values for predefined options that should apply across the pipeline. These would be the type of things that we would set on the General tab of a project in the Jenkins web forms (other than parameters, which have their own section). You can think of it as a place to set Jenkins-defined job options.

A simple example is the option to discard builds. Assume we had the setup in [Figure 7-5](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#fig_examp_job_build_discard) in our Jenkins job.



We could use the following code to achieve the same behavior in our Declarative Pipeline:

options {

buildDiscarder(logRotator(numToKeepStr:’3’))

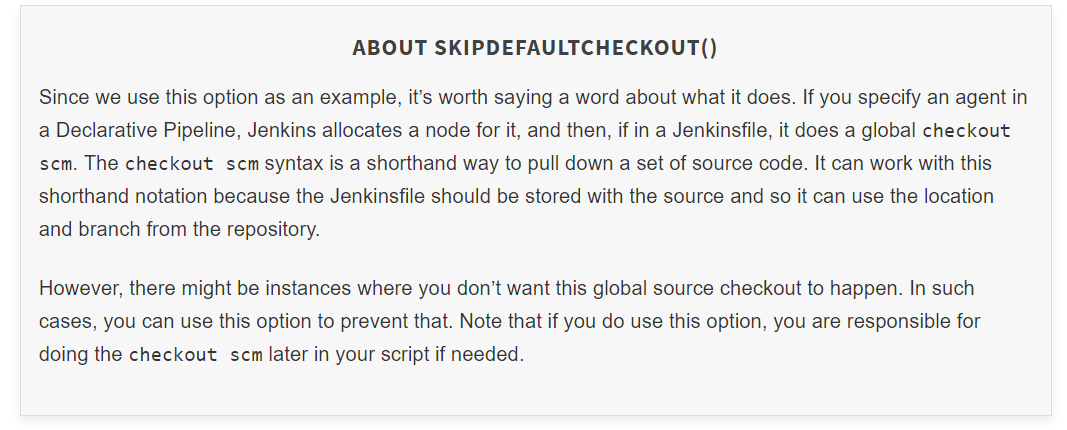
}

As well, there can be specific options for the declarative structure. Here’s an example of one:

options {

               skipDefaultCheckout()

}



**Options summary**

The following list below enumerates the available options and, briefly, their meaning and usage:

buildDiscarder

Keep the console output and artifacts for the specified number of executions of the pipeline.



disableConcurrentBuilds

Prevent Jenkins from starting concurrent executions of the same pipeline. The use case could be for preventing simultaneous access to shared resources or preventing a faster concurrent execution from overtaking a slower one. (This option is also discussed in [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow).)

options { disableConcurrentBuilds() }

retry

If the pipeline execution fails, retry the entire pipeline the specified number of times.

options { retry(2) }

skipDefaultCheckout

As just explained in the “[“About skipDefaultCheckout()”](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#About_skipDefaultCheckout_sidebar)” sidebar, this removes an implied checkout scmstatement, thus skipping the automatic source code checkout from a pipeline defined in a Jenkinsfile.

skipStagesAfterUnstable

If a stage of the pipeline renders the pipeline unstable, don’t process the remaining stages.

options { skipStagesAfterUnstable()}

timeout

Sets a timeout value for an execution of the pipeline. If this timeout value is passed, Jenkins will abort the pipeline.

options { timeout(time: 15, unit: 'MINUTES') }



timestamps

Add timestamps to the console output. This option requires the *Timestamper* plugin. Note that this option applies globally to the whole pipeline execution.

options { timestamps() }

## **triggers**

This directive allows you to specify what kinds of triggers should initiate builds in your pipeline. Note that these do not apply to Multibranch Pipeline or GitHub organization or Bitbucket team/project jobs that are marked by Jenkinsfiles and triggered otherwise—such as by a webhook that notifies Jenkins when a change is made.

There are four different (SCM-neutral) triggers currently available: cron, pollSCM, upstream, and githubPush.

cron

Refers to executing the pipeline at a specified regular interval, and pollSCM is for checking for source code updates (polling the source control management system) at a specified regular interval. If a source change is detected, the pipeline will be executed.

upstream

Takes a comma-separated string of Jenkins jobs and a condition to check. When a job in the string finishesand the result matches the treshold, the current pipeline will be retriggered. For example:

triggers {

upstream(upstreamProjects: 'jobA,jobB', threshold:

hudson.model.Result.SUCCESS)

}

githubPush

Refers to the same kind of behavior as the “GitHub hook trigger for GitSCM polling” setting in the Build Triggers section of a project in the Jenkins application. That is, if a webhook is set up on the GitHub side for events related to the GitHub repository, then when the payload is sent to Jenkins, it will trigger SCM polling for that repo from the Jenkins job to pick up any changes. The syntax should be simply:

triggers { githubPush() }

Both pollSCM and cron can use the cron syntax, a summary of which was given in an earlier chapter and which is repeated here for convenience.

### CRON SYNTAX

The cron syntax used in Jenkins is a specification of when and/or how often to do something based on five fields, separated by spaces. Each of the fields represents a different unit of time. The five fields are:

MINUTES

The desired minutes value within the hour (0–59).

HOURS

The desired hours value within the day (0–23).

DAYMONTH

The desired day of the month (1–31).

MONTH

The desired month of the year (1–12).

DAYWEEK

The desired day of the week (0–7). Here, 0 and 7 both represent Sunday.

Also, the \*/<value> syntax can be used in a field to mean “every <value>” (as in \*/5 meaning “every 5 minutes”).

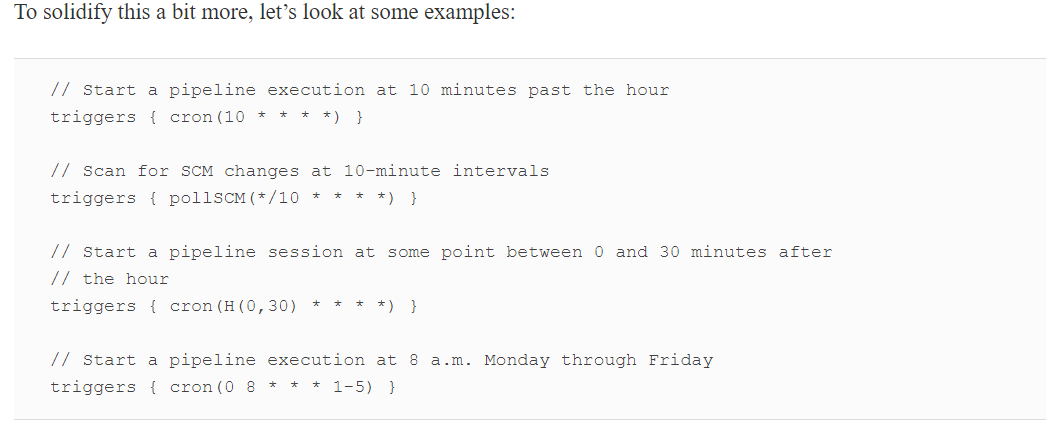
Additionally, the symbol H can be used in any of the fields. This symbol has a special meaning to Jenkins. It tells Jenkins to, within a range, use the hash of the project name to come up with a unique offset value. This value is then added to the lowest value of the range to define when the activity actually starts, within the range of values.

The idea here is not to have all projects that have the same cron values specified, starting at the same time. The offset from the hash serves to “stagger” the execution of projects that have the same cron timing.

Use of the H symbol is encouraged to avoid having projects start executing at the same time. Note that since the value is a hash of the project name, each value will be different from all others, but will remain the same for that project over time.

The H symbol can also have a range attached to it to specify limits on the interval it can pick. The syntax is H(<start range>, <end range>).

To solidify this a bit more, let’s look at some examples:



## parameters

This directive allows us to specify project parameters for a Declarative Pipeline. The input values for these parameters can come from a user or an API call. You can think of these parameters as being the same sort that you would specify in the web form with the “This build is parameterized” option.

You can get an idea of the syntax for these from the Snippet Generator by selecting the input step and then selecting the parameters and values you want to use.

The valid parameter types, with a description and example of each, are listed here (these are the same kinds ofparameters we discussed in conjunction with the input step in [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow)):

booleanParam

This is the basic true/false parameter. The subparameters for a booleanParam are name, defaultValue, and description.

parameters { booleanParam(defaultValue: **false**,

description: 'test run?', name: 'testRun')}

choice

This parameter allows selection from a list of choices. The subparameters for a choice are name, choices, and description. Here, choices refers to a list of choices you enter, separated by newlines, to present to the user. The first one in the list will be the default.

parameters{ choice(choices: 'Windows-1\nLinux-2', description:

'Which platform?', name: 'platform')}

file

This parameter allows for choosing a file to use with the pipeline. The subparameters include fileLocation and description.

The selected file location specifies where to put the file that is selected and uploaded. The location will be relative to the workspace.

parameters{ file(fileLocation: '', description: 'Select the file to

upload')}

text

This parameter allows the user to input multiple lines of text. The subparameters include name, defaultValue, and description.

parameters{ text(defaultValue: 'No message', description:

'Enter your message', name: 'userMsg')

password

This parameter allows the user to enter a password. For passwords, the text entered is hidden. The available subparameters are name, defaultValue, and description.

parameters{ password(defaultValue: "userpass1", description:

'User password?', name: 'userPW')}

This parameter allows the user to select a particular run from a job. This might be used, for example, in a testing environment. The subparameters available include name, project, description, and filter.

The project subparameter is the job that you want to allow the user to select a run from. The default run will be the last one. You also have access to certain environment variables in the script from whichever project you select. These include:

* PARAMETER\_NAME=<jenkins\_url>/job/<job\_name>/<run\_number>/
* PARAMETER\_NAME\_JOBNAME=<job\_name>
* PARAMETER\_NAME\_NUMBER=<run\_number>
* PARAMETER\_NAME\_NAME=<display\_name>
* PARAMETER\_NAME\_RESULT=<run\_result>

The filter subparameter allows you to filter the type of runs to offer based on the overall build status. Choices include:

* All Builds—including “in-progress” ones
* Completed Builds
* Successful Builds—this includes stable and unstable ones
* Stable Builds Only

parameters{ run(name: "Last success", description:

'Last successful project', project: 'project1',

filter: 'Successful Builds')}

string

This parameter allows for entering a string. (This is not hidden like a password parameter is.) The subparameters include description, defaultValue, and name.

parameters{ string(defaultValue: "Linux",

description: 'What platform?', name: 'platform')}

### USING PARAMETERS IN A PIPELINE

Once you define a parameter in the parameters block, you can reference it in your pipeline via the paramsnamespace, as in params.<parameter\_name>. Here’s a simple example using a string parameter in a Declarative Pipeline:

pipeline {

agent any

parameters{

string(defaultValue: "maintainer",

description: 'Enter user role:', name: 'userRole')

}

stages {

stage('listVals') {

steps {

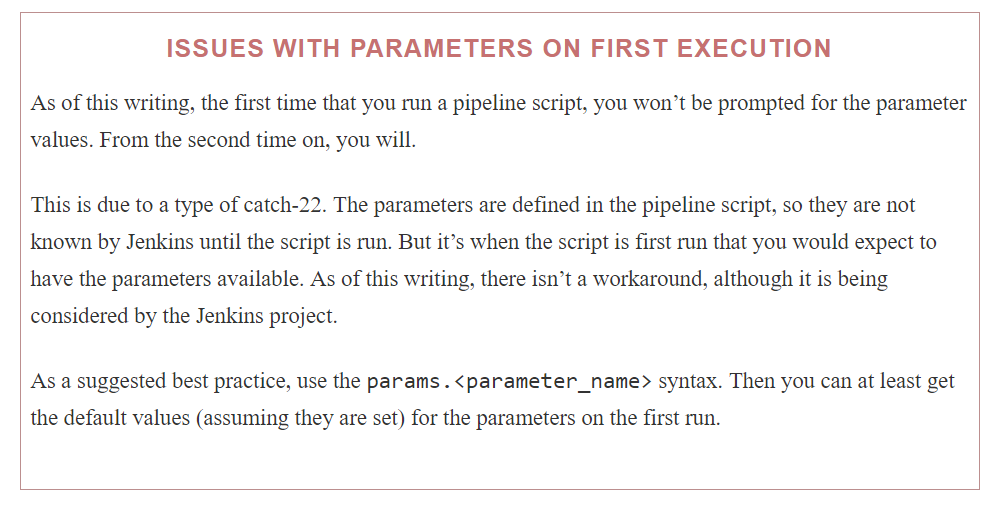
echo "User's role = ${params.userRole}"

}

}

}

}



## libraries

Read it from the book if required

## stages

Whether in a Scripted Pipeline or a Declarative Pipeline, Jenkins wants our code steps to be contained in one or more stages. In a Declarative Pipeline, the collection of individual stages is wrapped by the stages section. This makes our Declarative Pipeline more structured and tells Jenkins where the stages begin and end, as opposed to the pipeline-level directives that we’ve been looking at. stages is a required section, and you must have at least one stage within it. A section of a pipeline demonstrating this syntax is shown here:

pipeline {

  agent any

  stages {

     stage('name1') {

       steps {

          ...

       }

### STAGE

Within the stages section are the individual stages. Each stage has at least a name and one or more DSL steps. Within a stage, you may also have local environment, tools, and agent directives. If there are also corresponding global directives that define values with the same names, then the value defined in the directive in the stage will override the global one.

An example of this situation could be having the same environment variable defined in both an environment directive at the pipeline level and an environment directive in a stage.

If additional values (with different names) are defined in a directive at the pipeline level and the same directive in a stage, the additional settings in the stage are just added to the set already defined globally for the pipeline.

Other than the stage closure itself, the only required element in a stage (for a Declarative Pipeline) is the steps section.

### STEPS

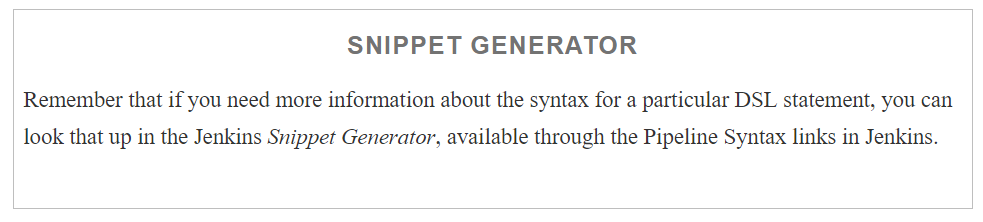
The steps block is required and indicates the actual work that will happen in the stage. It has the form:

steps {

<individual steps - i.e., DSL statements>

}

The individual steps can be any valid DSL statements, such as echo, archiveArtifacts, git, mail, etc. The syntax at this level is the same for Scripted or Declarative Pipelines in terms of using DSL statements. You cannot, however, use Groovy non-DSL statements or constructs, such as if-then or assignments.



Execution of the steps section can also be done conditionally in a pipeline, based on a set of conditions defined at the start of the stage. Let’s take a look at how that works.

#### **Conditional execution of a stage**

In any stage, you can have conditional execution. That is, you can have Jenkins decide whether or not to execute the steps in the stage based on one or more conditions evaluating to true. This is an optional construct that is not available at the top level of the script.

There are several different conditions that you can work with. The choices are:

branch "<name>"

Only proceed if the branch name is <name> or matches the (Ant-style) pattern.

stage('debug\_build') {

when {

branch 'test'

}

...

}

environment name: <name>, value: <value>

Only proceed if the specified environment variable <name> has the specified environment variable <value>.

stage('debug\_build') {

when {

environment name: "BUILD\_CONFIG", value: "DEBUG"

}

...

}

expression <valid Groovy expression>

Only proceed if the specified Groovy expression evaluates to true (meaning not false and not null).

stage('debug\_build') {

   when {

      expression {

echo "Checking for debug build parameter..."

  expression { **return** params.DEBUG\_BUILD }

   }

   ...

}

#### **Conditional execution with and, or, not**

In addition to using these conditions one at a time only when they are true, we can also use logical operators to check multiple conditions, or the inverse of one. The Declarative Pipeline syntax provides keywords that allow us to use the equivalent of “and,” “or,” and “not” logical operations with the three types of conditions we just discussed. The keywords for the three logical operators are:

allOf

When used in a when statement for conditional stage execution, the allOf keyword functions like an “and.” In order for the stage to proceed with its processing, “all of” the conditions included must be true.

when {

   allOf {

environment name: "BUILD\_CONFIG", value: "DEBUG"

branch 'test'

   }

}

anyOf

When used in a when statement for conditional stage execution, the anyOf keyword functions like an “or.” In order for the stage to proceed with its processing, “any of” the conditions included must be true.

when {

   anyOf {

environment name: "BUILD\_CONFIG", value: "DEBUG"

branch 'test'

   }

}

not

When used in a when statement for conditional stage execution, the not keyword functions just as the name implies. In order for the stage to proceed with its processing, the specified conditions must not be true.

when {

   not {

branch 'prod'

   }

}

There is one additional part of a stage that can also execute based on conditionals: post, for processing at the end of a stage. This is a powerful way to emulate the traditional post-build processing type of behavior within a stage.



## **Post**

post is another section available for use in the pipeline or in a stage. It is optional in both places. If present, it gets executed at the end of a pipeline or stage if the conditions are met. You can think of it like post-build actions for a traditional Jenkins Freestyle job or set of jobs.

The conditions in the post block are based on the build status. The syntax is as follows:

post {

<condition name> {

  <valid DSL statements>

}

<condition name> {

<valid DSL statements

}

...

The available conditions are:

always

Always execute the steps in the block.

changed

If the current build’s status is different from the previous build’s status, then execute the steps in the block.

success

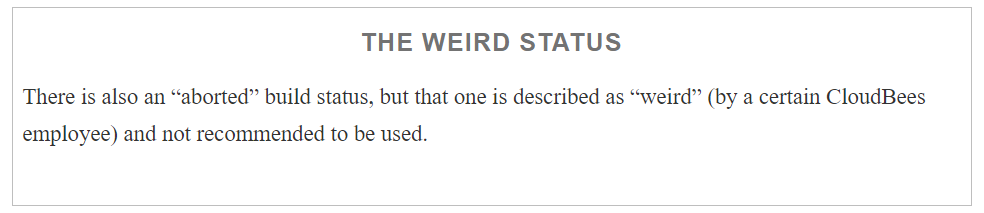
If the current build was successful, then execute the steps in the block.

failure

If the current build failed, then execute the steps in the block.

unstable

If the current build’s status was unstable, then execute the steps in the block.





# Dealing with Nondeclarative Code

The Declarative Pipeline syntax is great for simplifying the way we define pipelines. However, if you need to do something that can’t be expressed declaratively, it can be challenging to figure out how to accomplish that within the declarative structure.

Let’s take, for example, cases where you may need to do a simple assignment operation, or multiple ones. Here are some sample assignments needed to use Artifactory with Gradle in Scripted Pipeline code:



The problem here is that these assignment statements are trying to directly modify values via the DSL and are not declarative. While these statements are legal to use in Scripted Pipelines, they are not in Declarative Pipelines

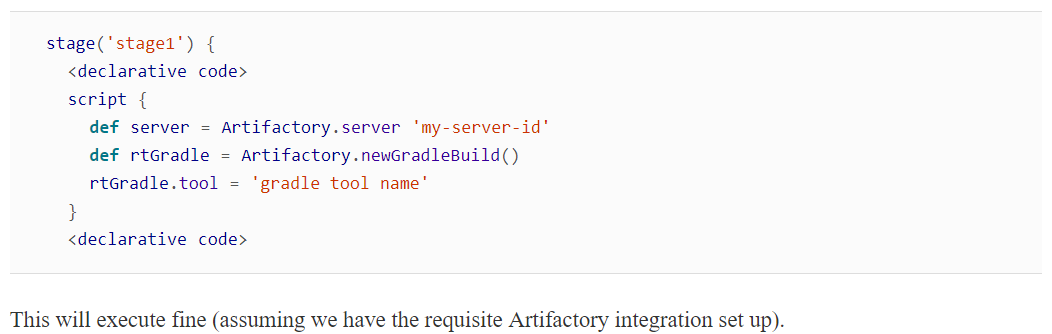
## The script Statement

To solve above problem

The script DSL statement is a special statement intended just for use in Declarative Pipelines; it allows you to define a block/closure that can house any nondeclarative code. As you may have guessed, the name is a reference to “Scripted” Pipelines.

The statement is put inside your Declarative Pipeline wherever you have to have non-declarative code. This method is likely the best way to handle this sort of situation, if you must use nondeclarative code and don’t want to create a shared library.

Turning back to our example assignment statements, wrapping them in a script statement would look like this:



# Using parallel in a Stage

We covered the parallel syntax for declarative syntax in [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow). With regard to using parallel in Declarative Pipelines, you can use it in a stage if it’s the only step in that stage. Note that the paralleldefinition itself can be of the traditional style (using a mapping to define the different parallel “branches”) or a newer style (as of Declarative Pipelines 1.2) that allows for the branches to be defined by stages. Code snippets of both are shown here (refer to [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow) for more details and complete examples):

stage ('Unit Test') {

   steps  {

      parallel(

         set1 : {

              ...

 stage('Unit Test') {

           parallel{

               stage ('set1') {

                  agent { label 'worker\_node2' }

                  steps {

# Script Checking and Error Reporting

As mentioned at the beginning of the chapter, one of the other nice features of Declarative Pipelines is that the formal structure allows for better script checking and more precise error reporting. That is, the checking and reporting are expressed in terms of the DSL and not just Groovy code with stacktraces.

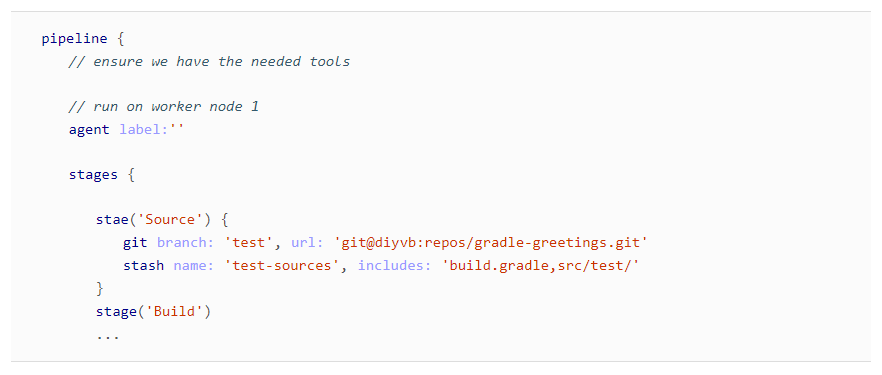
The verification is done at the start, in the editor, and errors are clearly identified, including line numbers. Argument types are also validated, and the environment is checked to make sure the necessary tools are available. If a required tool or tool version isn’t installed, the script will stop with an error.

The following code shows a Scripted Pipeline listing with a syntax error (stae instead of stage). [Figure 7-6](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#fig_err_rep_SP_syn_err)shows the resulting Groovy stacktrace that serves as error identification.





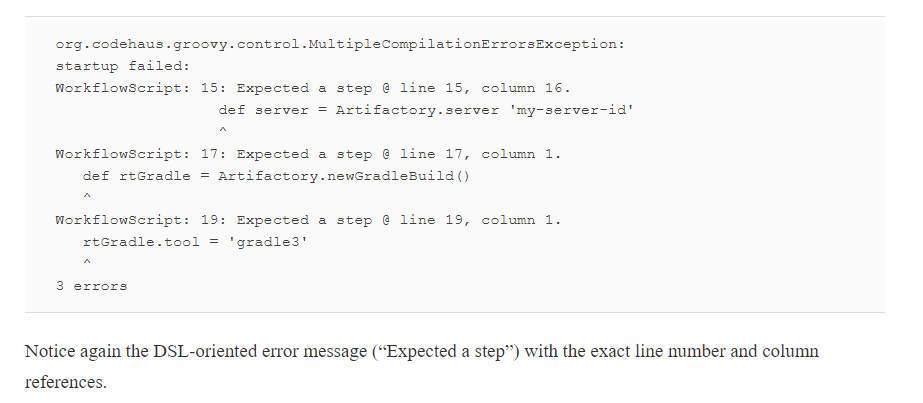
The following code shows a corresponding Declarative Pipeline listing. [Figure 7-7](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#fig_err_rep4dse) shows the clearer error checking that surfaces from using it.





Notice how much clearer and more precise the error message is, in terms of the Jenkins pipeline DSL, in the second example.

You may also recall the error messages we saw in the section [“Dealing with Nondeclarative Code”](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#challenges_ch_seven) when we looked at trying to put nondeclarative code into a Declarative Pipeline:



# Declarative Pipelines and the Blue Ocean Interface

Before we leave our detailed discussion of Declarative Pipelines, we should note one other aspect of them—they are uniquely suited for working with the new Jenkins Blue Ocean interface and the associated visual pipeline editor that it provides. This visual interface is regularly being enhanced and updated by CloudBees and the Jenkins community, and it presents an interesting new way to work with and create pipelines.

Blue Ocean plugins and Declarative Pipeline plugins go hand in hand. The well-defined structure of a Declarative Pipeline lends itself well to being parsed for presentation in a visual form. The limited structure also makes it easier to do the reverse: create a simple visual interface with specific selections that can be transformed into a pipeline.

That’s not to say that Scripted Pipelines can’t be used with the Blue Ocean interface—they will have a visual representation of separate stages, and point-and-click interfaces to view logs and errors. However, trying to dive deeper into the code in the visual interface will result in an error message, because Scripted Pipelines do not have “step” sections that contain the DSL statements. Likewise, Scripted Pipelines cannot be created or edited through the editor, since it expects to have a pipeline block (closure) encompassing all of the pipeline code.

[Chapter 9](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch09.html#CH_The_Blue_Ocean_Interface) is devoted to the Blue Ocean interface, and the interactive features such as the editor.