

267424 Custom Git-Gradle

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Starting Out With Git.....	3
Lab 2 - Branching, Merging and Working with Remotes.....	22
Lab 3 - Experimenting with Workflows.....	28
Lab 4 - Build Script Basics.....	39
Lab 5 - Build Java Project, Management Package Dependencies, and Run Unit Tests with Gradle.....	48
Lab 6 - Integrate Eclipse with GitLab and Gradle.....	56
Lab 7 - Appendix - Rebasing and Rewriting History.....	72

Lab 1 - Starting Out With Git

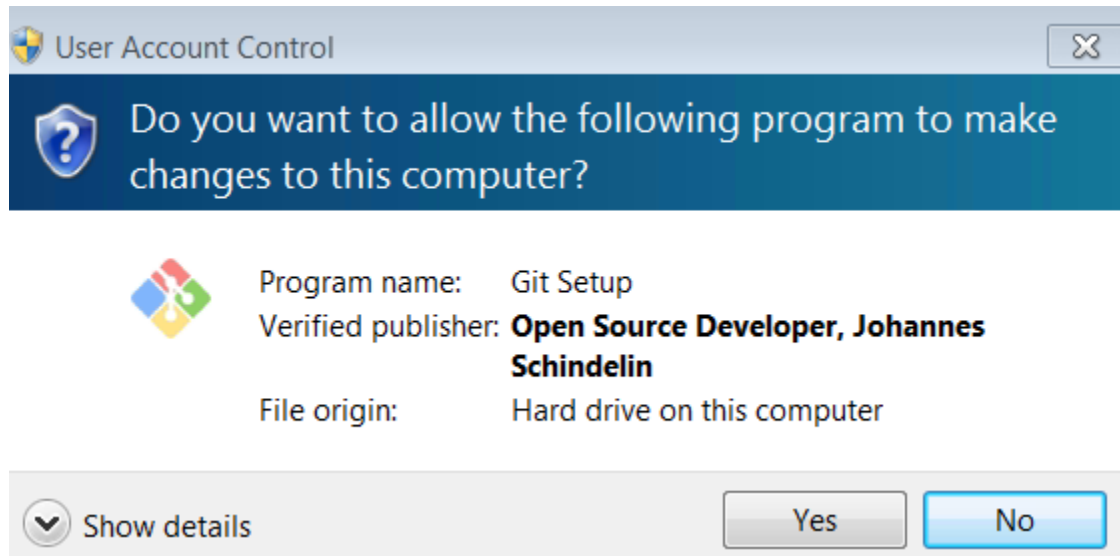
At the end of this lab you will be able to:

1. Install git on windows
2. Create a git repository
3. Perform basic git functions on the repository

Part 1 - Install Git

First, we're going to install "Git For Windows". For the lab, we have provided the download file for you. For your real installations, you can retrieve the download from "<http://git-scm.com/download/win>". It is free software, licensed under the GPL.

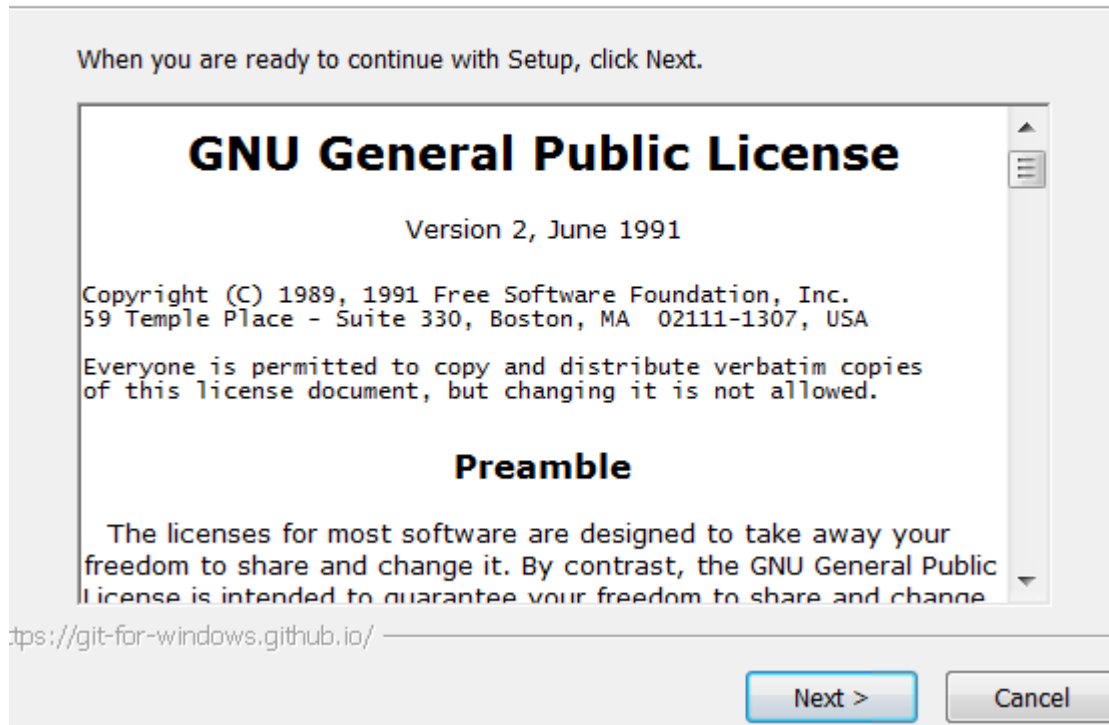
- __1. Using Windows Explorer, navigate to **C:\Software**.
- __2. Locate the file **Git-2.8.1-32-bit.exe**, and double-click on the file to run it.
- __3. Windows may show a message as below. Click Yes.



__4. The installer shows the licensing dialog. Click **Next**.

Information

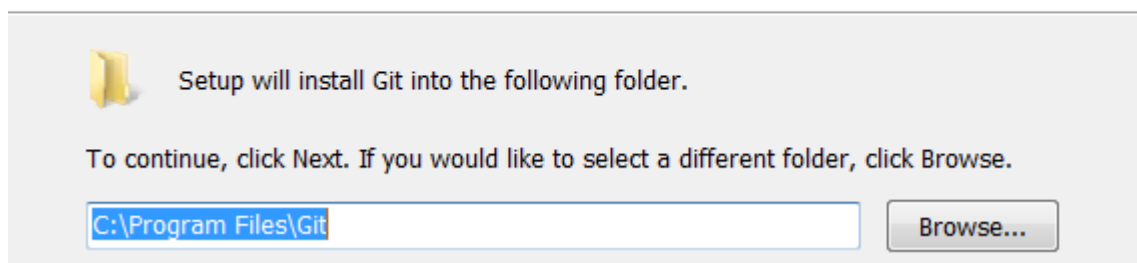
Please read the following important information before continuing.



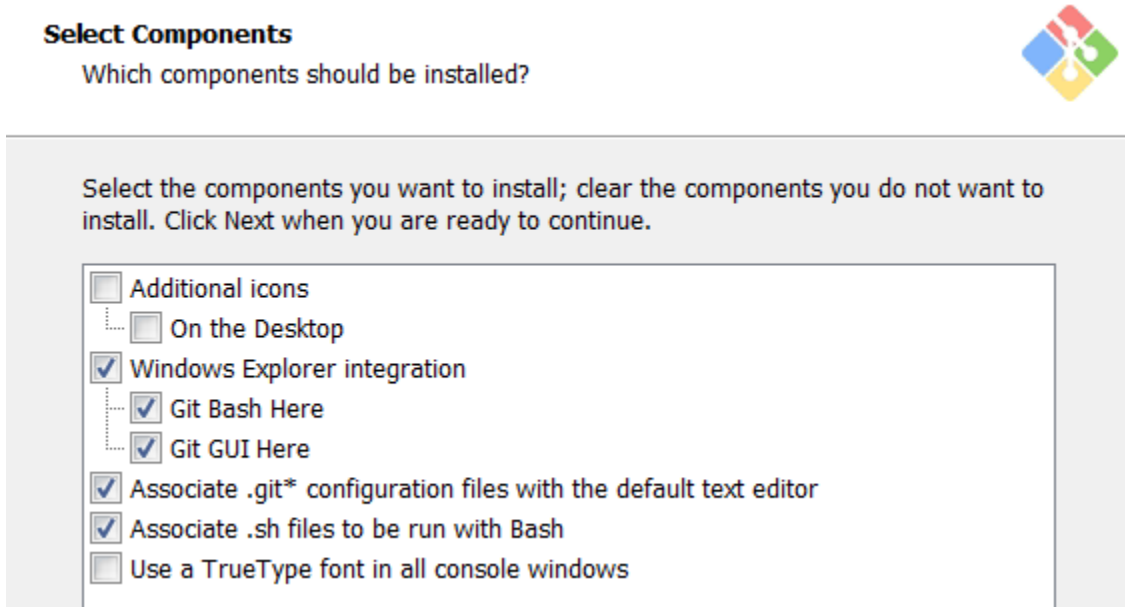
__5. Accept the default for the installation location by clicking **Next**.

Select Destination Location

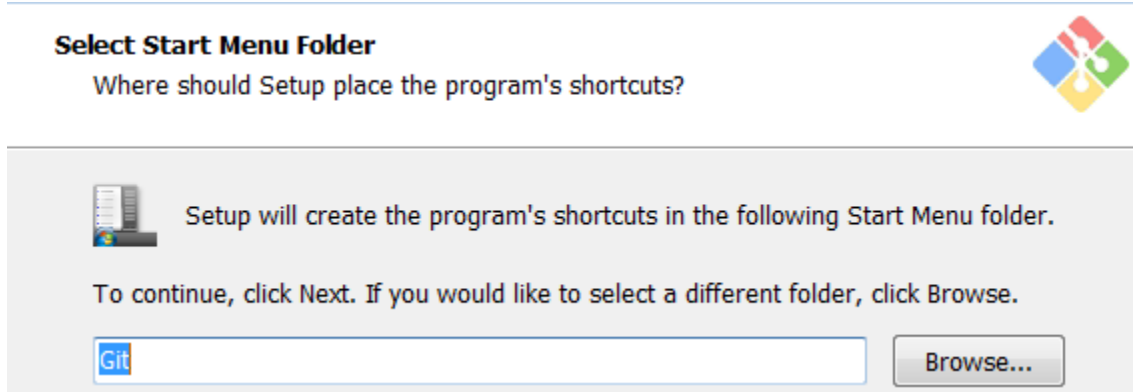
Where should Git be installed?



6. The installer shows the **Select Components** dialog. Leave all the defaults as-is, and then click **Next**.



7. The installer will display the **Select Start Menu Folder** dialog. Leave the default as-is and then click **Next**.



__8. The installer will display the **Adjusting your PATH environment** dialog. Select the radio button for **Use Git from the Windows Command Prompt**, and then click **Next**.

Adjusting your PATH environment

How would you like to use Git from the command line?



☐ **Use Git from Git Bash only**

This is the safest choice as your PATH will not be modified at all. You will only be able to use the Git command line tools from Git Bash.

☒ **Use Git from the Windows Command Prompt**

This option is considered safe as it only adds some minimal Git wrappers to your PATH to avoid cluttering your environment with optional Unix tools. You will be able to use Git from both Git Bash and the Windows Command Prompt.

☐ **Use Git and optional Unix tools from the Windows Command Prompt**

Both Git and the optional Unix tools will be added to your PATH.

Warning: This will override Windows tools like "find" and "sort". Only use this option if you understand the implications.



__9. The installer displays the **Configuring the line ending conversion** dialog. Leave the default as-is, and then click **Next**.

Configuring the line ending conversions

How should Git treat line endings in text files?



☒ **Checkout Windows-style, commit Unix-style line endings**

Git will convert LF to CRLF when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Windows ("core.autocrlf" is set to "true").

☐ **Checkout as-is, commit Unix-style line endings**

Git will not perform any conversion when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Unix ("core.autocrlf" is set to "input").

☐ **Checkout as-is, commit as-is**

__10. Click **Next** to leave the default.

Configuring the terminal emulator to use with Git Bash

Which terminal emulator do you want to use with your Git Bash?



☒ **Use MinTTY (the default terminal of MSYS2)**

Git Bash will use MinTTY as terminal emulator, which sports a resizable window, non-rectangular selections and a Unicode font. Windows console programs (such as interactive Python) must be launched via ``winpty`` to work in MinTTY.

☐ **Use Windows' default console window**

__11. Click **Install**.

Configuring extra options

Which features would you like to enable?



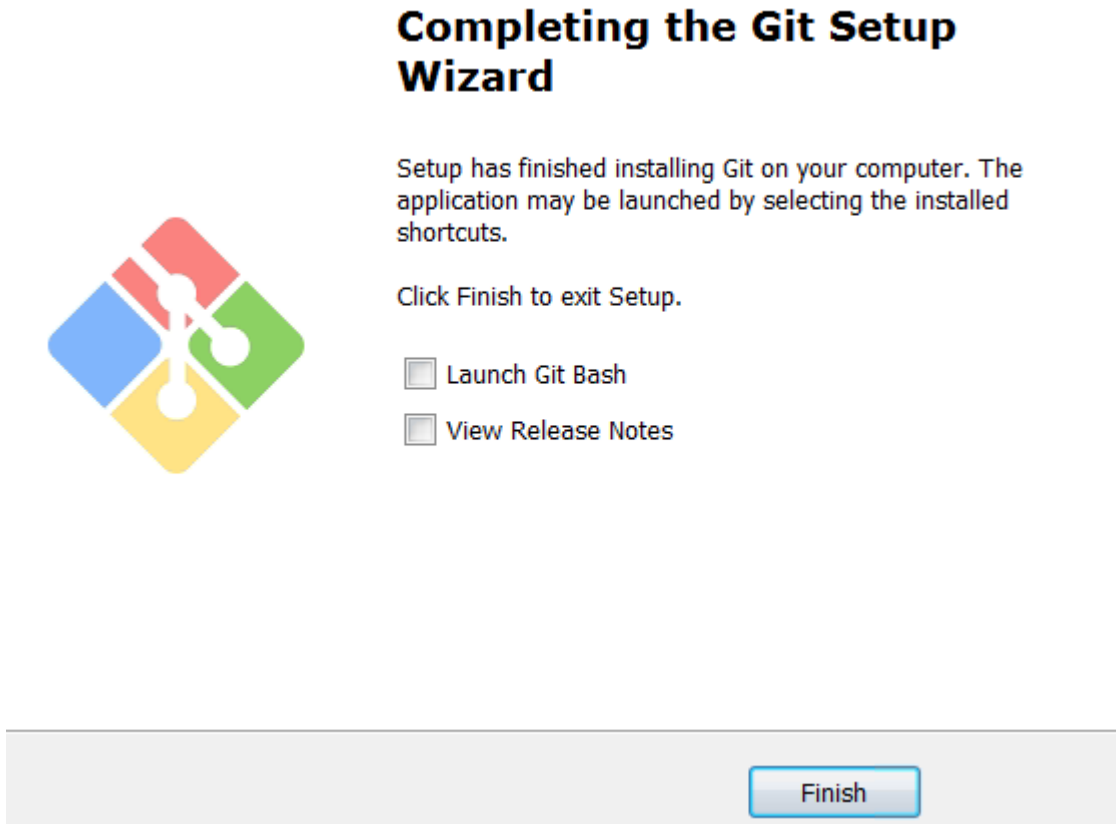
☒ **Enable file system caching**

File system data will be read in bulk and cached in memory for certain operations ("`core.fscache`" is set to "`true`"). This provides a significant performance boost.

☐ **Enable Git Credential Manager**

The installer will run for a few seconds installing Git.

__12. The installer will display the **Git Setup** dialog. Un-check the checkbox for **View Release Notes**, and then click **Finish**.



That's it! We've installed Git for Windows

Part 2 - Set Windows Explorer View Options

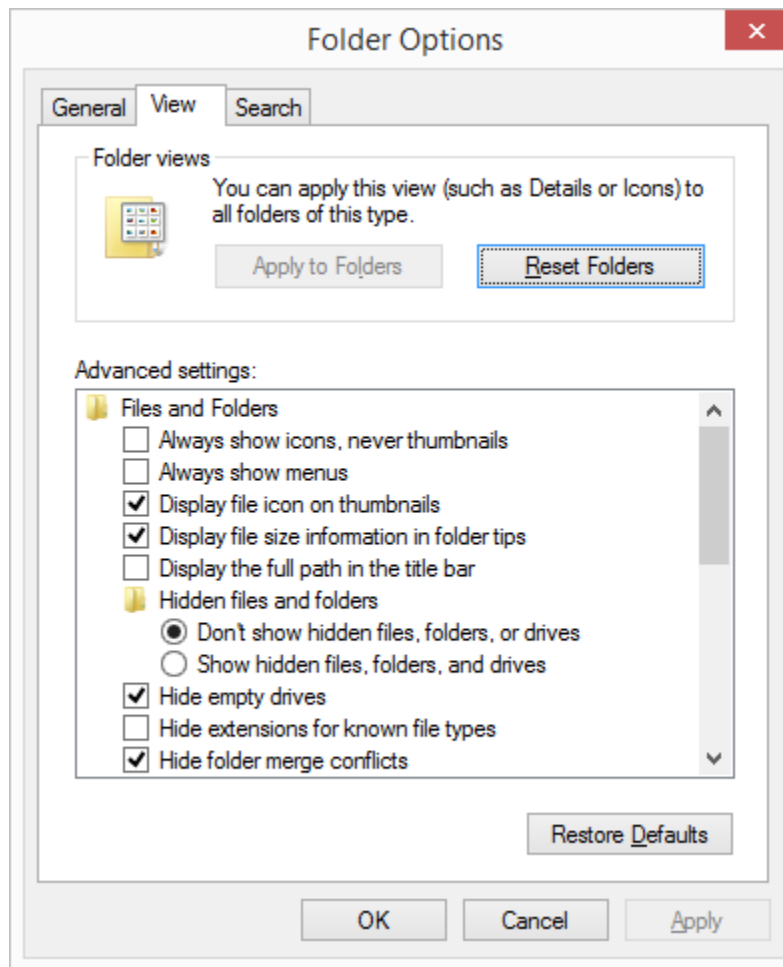
By default, Windows Explorer hides the extensions of files when there is an application associated with that file. We're going to shut that off so that we can see the entire file name in Windows Explorer.

__1. Open Windows Explorer.

Note. The following steps are for Windows 7 and may vary on other OS.

__2. From the menu, select **Tools** → **Folder Options**.

___3. Click the **View** tab to select it.



___4. Find the entry for “Hide extensions for known file types”. Make sure it is unchecked, and then click **OK**.

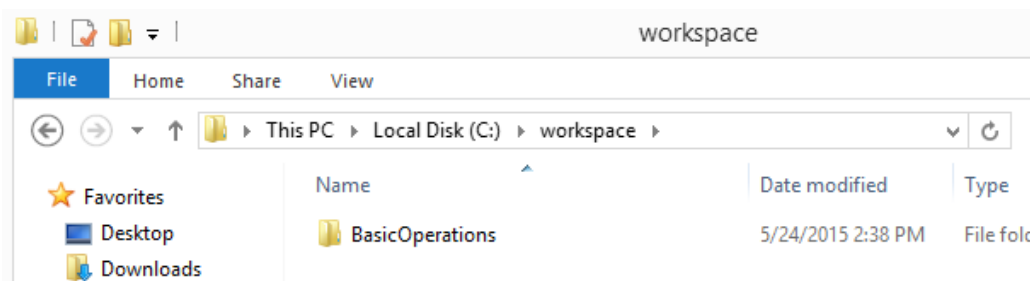
___5. Leave the Windows Explorer window open – we'll be using it in the next part of the lab.

Part 3 - Create a Workspace Folder with a Repository

___1. Use Windows Explorer to create a folder in the **C:** drive called **C:\workspace**

___2. Double-click on the new folder to open it.

___3. Create a new folder called **BasicOperations**



___4. Open a Windows Command Prompt by going to the start menu and then type 'com'.

The search box will appear, and one of the entries in it should be the "Command Prompt". Click the command prompt to open it.

__5. At the command prompt, enter

```
cd \workspace\BasicOperations
```

__6. At the command prompt, enter

```
git init
```

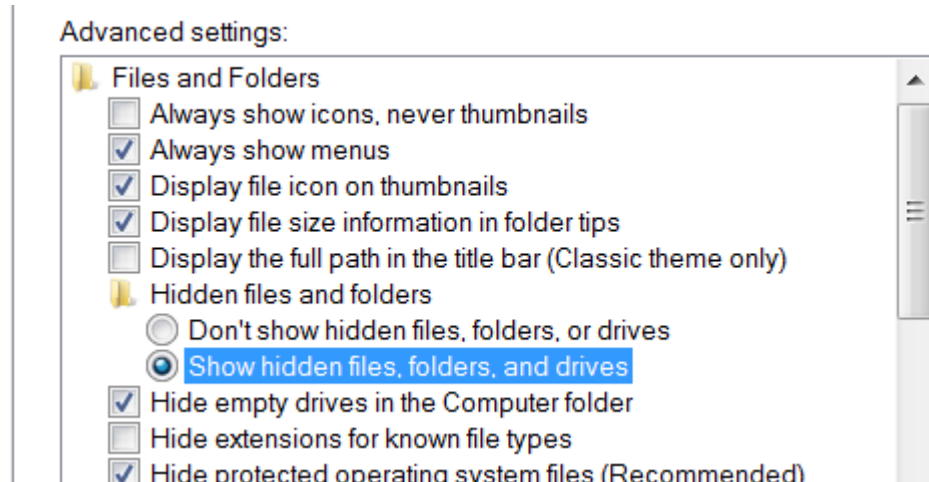
__7. You should see git create the repository, as shown:

```
C:\workspace\BasicOperations>git init
Initialized empty Git repository in C:/workspace/BasicOperations/.git/
```

__8. Go back to the Window Explorer. Double-click on the **BasicOperations** folder to enter it.

There probably appears to be nothing in this folder, but there is actually a hidden folder that contains the git repository. Let's change the view options so we can see it.

__9. In the Explorer main menu, click **Tools** → **Folder Options**. Then click the **View** tab and select **Show hidden files, folders, and drives** and click **OK**.



__10. Now we can see that there is a folder called '.git' inside the **BasicOperations** folder. This folder is the git repository. The name comes from the Unix convention where the shell's 'ls' command considers a file 'hidden' if it starts with a '.' character.

Name	Date
.git	5/2

Part 4 - Tell Git Who You Are

One interesting aspect of Git is that it separates user identity in the repository from any sort of authentication or authorization. Because a distributed repository will generally be maintained by many separate individuals or systems, the identity of the committer must be contained in the repository. A system like 'Subversion' would ask you to authenticate against its authentication database when you did a “commit” to its remote repository, but Git has some challenges in this area: First, the act of doing a “commit” is separate from the act of “pushing” changes to a remote repository. Second, there is no central repository or authentication database. Third, when we eventually send changes to a remote, those changes could include changes that we got from other developers, and we want to preserve information on “who did what changes”. So, even if we're not connected to any central repository, we need to tell Git who we are. The identity that we supply will be recorded whenever we commit to a repository. This identity is part of Git's configuration. As such, you could have a global identity (i.e. for every repository you have), or you could have a different identity in each local repository. We're going to set up a global identity.

For the purposes of this lab, we're going to use an imaginary user named “Alice Smith”, who has an email address “alice@smith.com”.

__1. Ensure that the Command Prompt is showing the **BasicOperations** folder from the last part of the lab.

__2. Enter the following two lines in the Command Prompt:

```
git config --global user.name "Alice Smith"
git config --global user.email alice@smith.com
```

___3. Ensure that there were no error messages. The output should be similar to:

```
C:\workspace\BasicOperations>git config --global user.name "Alice Smith"
C:\workspace\BasicOperations>git config --global user.email alice@smith.com
C:\workspace\BasicOperations>
```

Part 5 - Create Files and Commit Them to the Repository

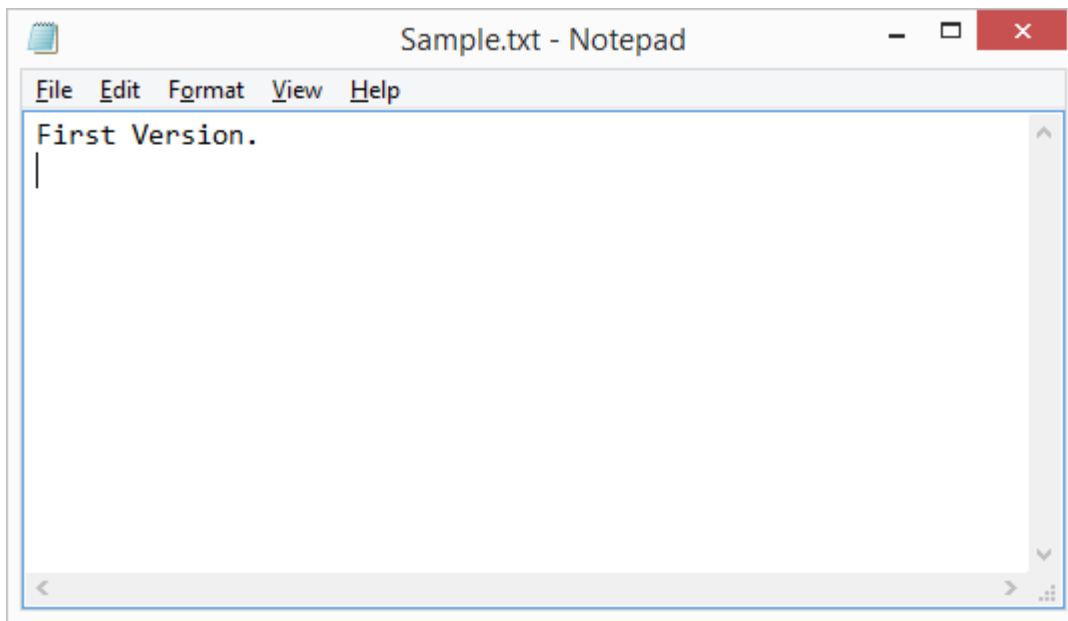
Now we'll create a file or two and do our first commit. For the sake of simple tools, we're going to just use **Notepad** here, since it's already installed with Windows, but feel free to install and use a different editor if you prefer.

___1. Ensure that Windows Explorer is showing the **BasicOperations** folder from the last part of the lab.

___2. Right-click in an empty area of the main panel, and select **New → Text Document**.

___3. Windows will create the new file, and should leave the file name selected for you to edit. Change the file name to **Sample.txt**

___4. Double-click on **Sample.txt** to open it in **Notepad**. Change the contents to read "First Version." and hit enter.



___5. Save the file by clicking **File → Save**, and then close the file by clicking the close button.

At this point, the file is there, but Git isn't tracking it yet (even though Git certainly knows it's there, after all it's in the file system). Let's add the file into Git's staging area.

___6. In the command prompt, enter:

```
git status
```

Git will tell us about the files it sees in the folder:

```

C:\workspace\BasicOperations>git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Sample.txt

nothing added to commit but untracked files present (use "git add" to track)
C:\workspace\BasicOperations>_

```

As we can see, git is aware of the new file, but it is "untracked". Let's change that.

___7. At the command prompt, enter:

```
git add .
```

The '.' says to add the contents of the current directory. If you preferred, you could have called out the name of the file itself.

___8. At the command prompt, enter:

```
git status
```

Git should now report that the file is "staged" as a file to be committed.

```

C:\workspace\BasicOperations>git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   Sample.txt

```

Before we do the commit, let's talk about the default editor. A Commit needs to be accompanied by a 'commit message' that goes into the repository's log to explain the commit. We could provide a commit message with the '-m' option to git's command line, however, that tends to lead to short, one-line commit messages. Normally, we'd like to be a little more descriptive with the commit message. As a result, if we don't specify a commit message, git will open up an editor with a default message from a template.

By default, the editor is a Windows version of 'vim' that was installed with the git package. If you're familiar with 'vim' then go ahead and use it (it's an excellent programmer-oriented editor), but if you're not already a 'vim' user, it's a little complicated to explain here. So, we'll change the default editor to Wordpad in the next step.

__ 9. At the command prompt, enter (all this should be on one line):

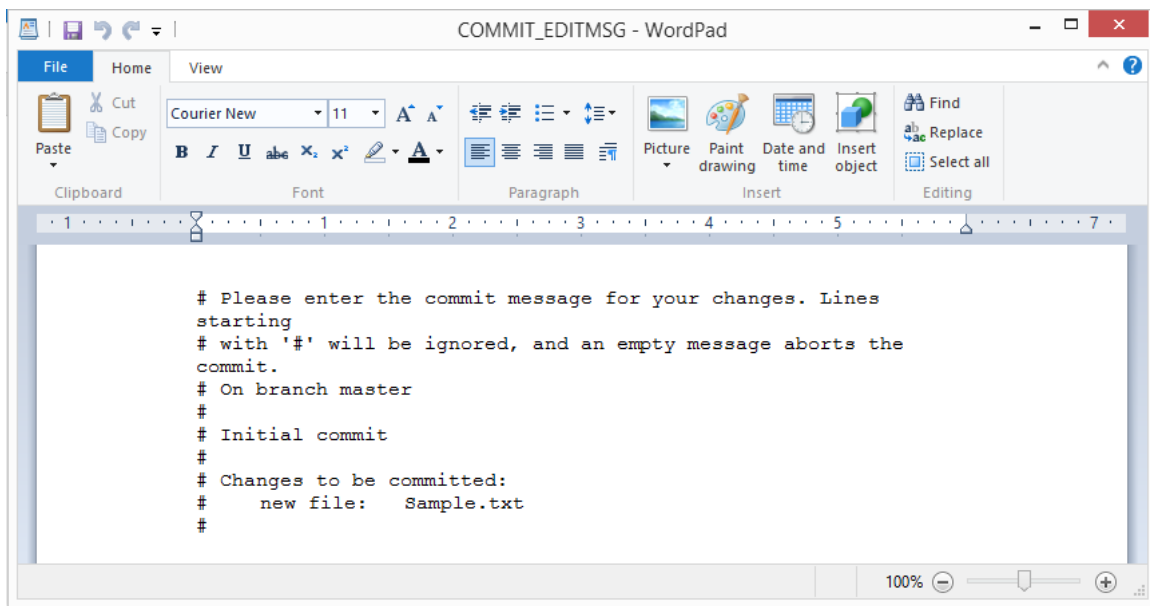
```
git config --global core.editor "'C:\Program  
Files\Windows NT\Accessories\wordpad.exe'"
```

Note the single quotes and double-quotes in the above line. They need to be typed exactly as shown, because the path to the wordpad executable contains spaces.

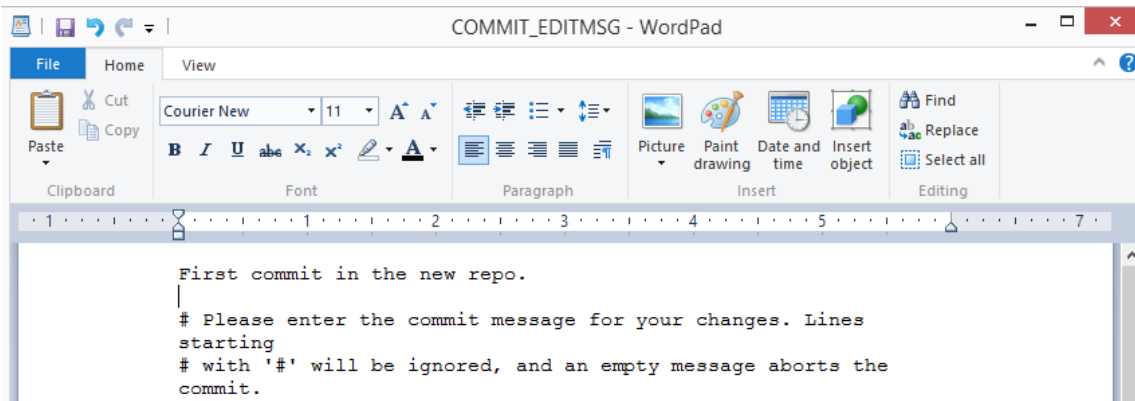
__ 10. At the command prompt, enter:

```
git commit
```

__ 11. An instance of Wordpad should open, with the default commit message in it.



__12. Enter a message at the top of the file. For instance, “First commit in the new repo.”



__13. Save the file by selecting **File** → **Save**, and then exit Wordpad by clicking the close button.

__14. The command prompt should reflect the completed commit:

```
C:\workspace\BasicOperations>git commit
[master (root-commit) f8fd0df] First commit in the new repo.
1 file changed, 1 insertion(+)
create mode 100644 Sample.txt
```

__15. Open the 'Sample.txt' file with Notepad again, and then change the contents to read 'Second Version'.

__16. Save and close the file.

__17. Once again, go to the command prompt and enter:

```
git status
```

Git should tell us that 'Sample.txt' was modified, but is not currently staged for commit.

```
C:\workspace\BasicOperations>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Sample.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

If we were to commit now, we would actually be committing without any changes. In fact git will tell us that there are no changes staged, and we need to add files before committing.

__18. At the command prompt, enter:

```
git diff
```

Git will show us what' changed in the folder.

```
C:\workspace\BasicOperations>git diff
diff --git a/Sample.txt b/Sample.txt
index b7a9a66..c08d855 100644
--- a/Sample.txt
+++ b/Sample.txt
@@ -1,1 @@
-First Version.
+Second Version.
```

__19. Having reviewed the 'diff', we can decide to go ahead and commit all the changes. At the command prompt, add all the updated files into the staging area by entering:

```
git add .
```

__20. At the command line, enter:

```
git commit
```

__21. Git opens up Wordpad with the proposed commit message. Note that the file now appears under '**Changes to be committed**'.

__22. The commit message is intended for us to give a description of the commit that will be useful when we review the logs. Enter 'We're on the second version.', and then save the file and close Wordpad.

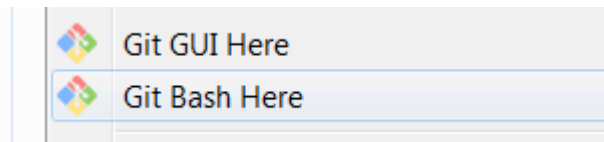
```
C:\workspace\BasicOperations>git commit
[master 1c07c33] We're on the second version.
1 file changed, 1 insertion(+), 1 deletion(-)
```

Part 6 - Review the Change History

At this point, we've now committed two different versions of the file '**Sample.txt**'. Let's see if we can find any evidence of this history.

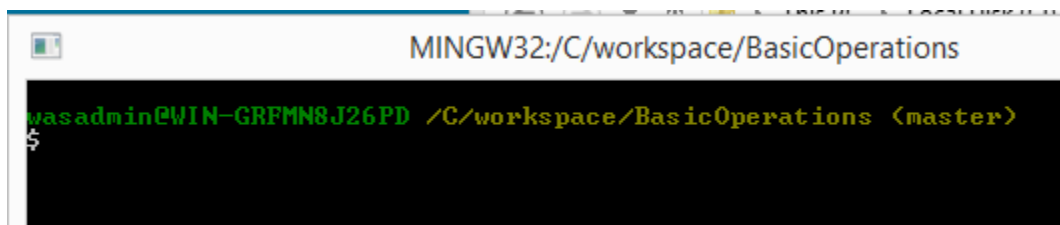
Just to be different, let's try out Git's command shell.

- __1. Ensure that the Windows Explorer window is still open and displaying the **C:\workspace\BasicOperations** folder.
- __2. Right-click in an empty area of the window, and then click on '**Git Bash Here**'.



Git will display a new terminal window with a '\$' prompt. If you've used Unix or Linux, you may recognize this as the Bourne shell or 'bash' prompt. In fact, Git for Windows includes a variant of the 'bash' shell derived from the 'Cygwin' system. Cygwin is a software package that provides a traditional Unix-style shell and the full set of Unix-style utilities in a Windows environment. Git for Windows includes a subset of these utilities.

The git commands are available at a command prompt, or through the 'Git Bash' facility, or also in Windows PowerShell. The commands and arguments are identical in all three access modes, so choosing one is a matter of individual taste.



- __3. Type 'git log' and press return.


```
wasadmin@WIN-GRFMN8J26PD /C:/workspace/BasicOperations <master>
$ git log
commit 0a0505e9c4735434f0c7c9bb57f8c78c8976a32a
Author: Alice Smith <alice@smith.com>
Date: Sun May 24 19:17:22 2015 -0400

    We're on the second version.

commit 06ffbff869abe47c7e0315fa98371e3b39b2ad55
Author: Alice Smith <alice@smith.com>
Date: Sun May 24 19:03:25 2015 -0400

    First Commit in the new repo.

wasadmin@WIN-GRFMN8J26PD /C:/workspace/BasicOperations <master>
$
```

Git displays a list of all the commits that we have done, in reverse chronological order.

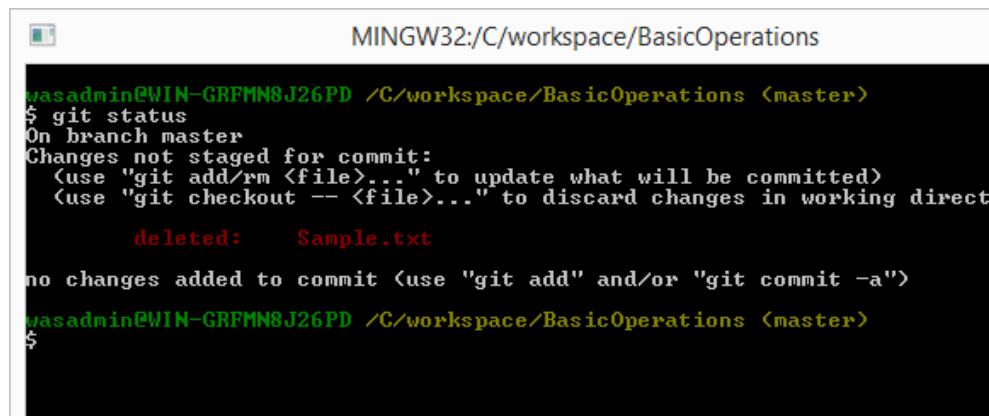
__4. Leave the **Git Bash** window open for the next part of the lab.

Part 7 - Delete a File and Recover It

In this section, we'll demonstrate that files are actually stored in the repository, even if we delete the file in our folder.

__1. Using Windows Explorer, delete the file '**Sample.txt**'. Right-click on the file and select **Delete**.

__2. In the **Git Bash** window, type 'git status' and press return.



```
MINGW32:/C:/workspace/BasicOperations

wasadmin@WIN-GRFMN8J26PD /C:/workspace/BasicOperations (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

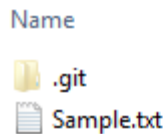
        deleted:    Sample.txt

no changes added to commit (use "git add" and/or "git commit -a")
wasadmin@WIN-GRFMN8J26PD /C:/workspace/BasicOperations (master)
$
```

Notice that git sees the removal as a change that has not been staged or committed. If we really wanted the change to stick, we could say 'git rm Sample.txt' to stage the removal, and then we could commit the change. Instead, we'll recover the file from the repository.

__3. Type 'git checkout -- Sample.txt' and then press return.

'Sample.txt' has re-appeared in Windows Explorer.



Note: The command 'git checkout -- Sample.txt' looks a little funny – what's the double-dash for? The answer will make more sense after we've talked about branches. Basically, it's possible to pull a file in from a different branch than the one that we're currently on, so the 'git checkout' command actually allows you to specify the branch name and the file path we want to recover. '--' separates the parameters from the list of files, and basically indicates the current branch.

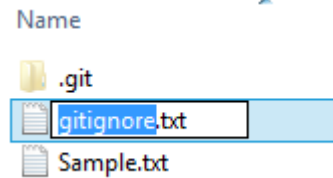
Part 8 - Use a '.gitignore' File

It's common to have files in a working directory that we don't really want to put into the repository. For example, a compiler might take "hello.c" and generate "hello.o". Since "hello.o" is a generated file, and will be regenerated every time we execute the build, it makes no sense to put it into version control. We can create a file called '.gitignore' that specifies a set of patterns to specify files that should be excluded from the repository.

__1. Ensure that our Windows Explorer is still displaying the 'C:\workspace\BasicOperations' folder.

__2. Right-click in an empty area of the Explorer window, and then select **New** → **Text Document**.

__3. Explorer will show the new file, with the base part of the name (excluding '.txt') selected. Type '**gitignore**'. Right now we are calling the file 'gitignore.txt'.



Note: We really want the file to be called '.gitignore', but Windows Explorer doesn't like that style of name. We're going to edit the file in a format that Explorer likes, and then rename it in the bash prompt window.

Alternately, if you are familiar with the 'vi' editor, the git bash utility includes a version of 'vim' that you can open from the bash prompt, with command 'vi .gitignore'.

__4. Double-click on '**gitignore.txt**' to open it in Notepad.

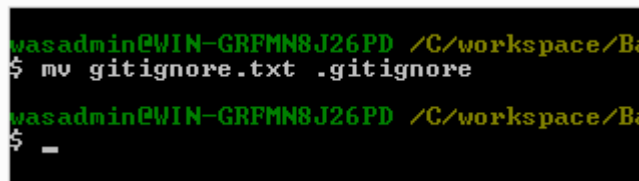
__5. Add the following three lines to '**gitignore.txt**':

```
*.o
local.properties
temp/**
```

__6. Save and close the file.

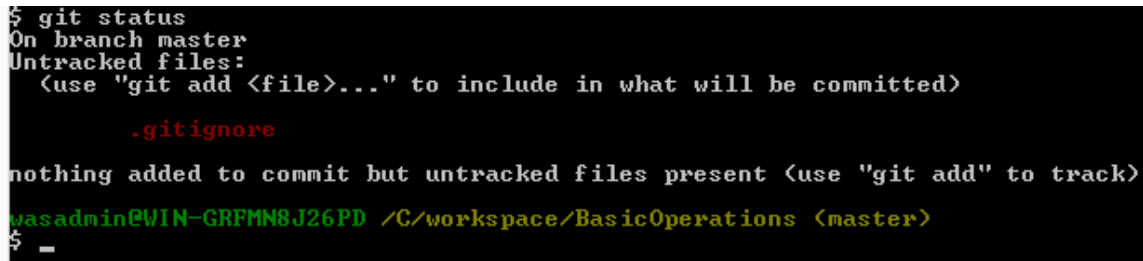
__7. In the **git bash** window, enter:

```
mv gitignore.txt .gitignore
```

A screenshot of a terminal window with a black background and green text. The prompt is 'wasadmin@WIN-GRFMN8J26PD /C/workspace/Ba'. The command '\$ mv gitignore.txt .gitignore' has been entered and executed. The next prompt shows a cursor on a new line.

__ 8. In the **git bash** window, enter:

```
git status
```

A terminal window showing the output of the 'git status' command. The text is as follows:

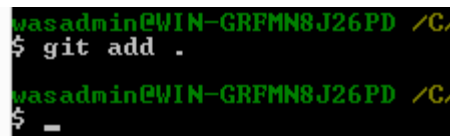
```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

nothing added to commit but untracked files present (use "git add" to track)
wasadmin@WIN-GRFMN8J26PD /C:/workspace/BasicOperations (master)
$ _
```

Notice that git is telling us that we have unstaged changes – the '.gitignore' file that we just created. This time we'll use the command line to add and commit them.

__ 9. In the **git bash** window, enter '**git add .**' (note the 'dot' at the end) and then press return.

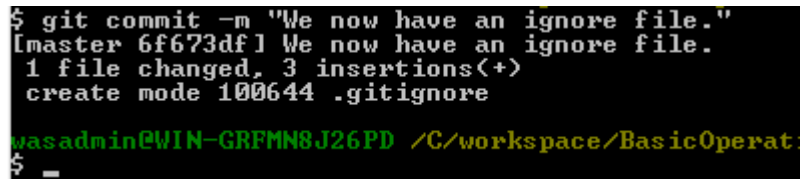
A terminal window showing the execution of the 'git add .' command. The text is as follows:

```
wasadmin@WIN-GRFMN8J26PD /C/
$ git add .
wasadmin@WIN-GRFMN8J26PD /C/
$ _
```

__ 10. In the **git bash** window, enter the following line, and then press return:

```
git commit -m "We now have an ignore file."
```

In this example, we've used a shortcut "-m" option to specify the commit message. If you leave out that option on the command line, git will open up an editor (in this case 'vi') to let you edit the commit message.

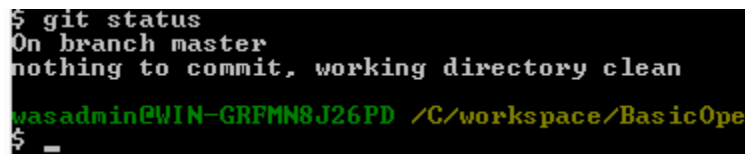
A terminal window showing the output of the 'git commit -m' command. The text is as follows:

```
$ git commit -m "We now have an ignore file."
[master 6f673df] We now have an ignore file.
1 file changed, 3 insertions(+)
create mode 100644 .gitignore
wasadmin@WIN-GRFMN8J26PD /C:/workspace/BasicOperations
$ _
```

__ 11. Enter the following line and then press return, to create a file called 'hello.o':

```
echo 'Hi there' > hello.o
```

__ 12. Enter '**git status**' and press return.

A terminal window showing the output of the 'git status' command after creating the 'hello.o' file. The text is as follows:

```
$ git status
On branch master
nothing to commit, working directory clean
wasadmin@WIN-GRFMN8J26PD /C:/workspace/BasicOperations
$ _
```

Notice that git reports nothing to commit. Because the file name 'hello.o' matches the pattern in the line '*.o' in our '.gitignore' file, git ignores it. It would also ignore a file called 'local.properties' and a folder called 'temp'.

Part 9 - Review

In this lab, we started off by installing Git for Windows, and then created a repository and executed a few simple operations on it.

One concept to note is the idea of the separate areas of a project that we're managing under git: the working directory, the staging area, and the repository. The working directory is the area that is currently in the operating system's file system. We make changes to files in the file system, and then tell git which changes we want to record by copying the changes to the staging area (or 'staging' them) using 'git add'. Then we put the changed snapshot into the repository by 'committing' them. Once a file is in the repository, it is there forever, even if we remove it from the working directory. We can recover any version of the file that we've previously committed.

Lab 2 - Branching, Merging and Working with Remotes

One of the main themes of Git is the idea that we should isolate work on branches and use those branches to coordinate the work of multiple developers. In this lab, we'll examine the use of branches and the act of merging.

In addition, Git works on the basis of distributed repositories. We'll explore the use of more than one repository, or Remote repositories.

At the end of this lab you will be able to:

1. Clone an existing repository
2. Create a branch
3. Merge a branch onto another branch.

Part 1 - Clone a Repository

For this lab, we will clone a pre-existing repository that has been provided to you

- __1. Using Windows Explorer, navigate to C:\workspace
- __2. Open a command prompt window, and then enter:

```
cd \workspace
```

- __3. In the command prompt window, enter:

```
git clone \LabFiles\BranchMergeRemotes
```

This command creates a clone of the repository that is located in the file system at 'C:\LabFiles\BranchMergeRemotes'.

```
C:\workspace>git clone \LabFiles\BranchMergeRemotes
Cloning into 'BranchMergeRemotes'...
done.
C:\workspace>
```

Notice that in the **Windows Explorer** window, there is a new folder called **BranchMergeRemotes**.

In this case, we're cloning from a repository that is in the file system, but the exact same syntax can be used to clone from a repository on a remote server. The only difference is the URL that's used to identify the original repository.

For instance, 'git clone https://github.com/angular/angular.js.git' would create a local copy of the repository that contains the popular 'AngularJS' framework, which is hosted on GitHub.

- __4. In the **Command Prompt** window, type the following and then press return:

```
cd BranchMergeRemotes
```

Part 2 - Create a Branch to Work on a Feature.

In many version control systems, particularly older systems, creating a branch is a

difficult and often expensive operation. In Git, however, creating a branch is a very lightweight operation, and in general, we want to create a branch for every feature that we work on. When a feature reaches a stable state, we'll merge it back to the master branch. In this way, the master branch only contains stable features, and any developer's work is isolated in its own feature branch.

__1. In the **Command Prompt** window, type the following, pressing return after each line:

```
git branch add-text  
git checkout add-text
```

The lines above create a new branch called 'add-text' (which is a description of what we're about to do), and then switch to that branch.

Note: The two commands above could have been shortened to 'git checkout -b add-text'.

__2. Type 'git status'.

```
C:\workspace\BranchMergeRemotes>git checkout add-text  
Switched to branch 'add-text'  
  
C:\workspace\BranchMergeRemotes>git status  
On branch add-text  
nothing to commit, working directory clean  
  
C:\workspace\BranchMergeRemotes>
```

Notice that the status message indicates that we're on the branch called 'add-text'.

That's all that's required to create a branch in Git!

__3. Type 'git branch'.

```
C:\workspace\BranchMergeRemotes>git branch  
* add-text  
master
```

The branch command on its own simply lists the branches that are defined in the repository. Notice that the 'add-text' branch is both highlighted and has an asterisk next to its name, indicating that we're on that branch.

Part 3 - Work on the Branch

When we're on a branch, we can go ahead and make some changes and then stage and commit those changes.

- __1. In Windows Explorer, navigate to 'C:\workspace\BranchMergeRemotes'.
- __2. Right-click on '**index.html**', and then select **Open With** → **Notepad**.
- __3. Replace the text 'Hello World' with 'This is the text that we really want.'

```
File Edit Format View Help
<!DOCTYPE html>

<html>
<body>
This is the text that we really want.
</body>
</html>
```

__ 4. Save and close the file.

__ 5. Back in the **Command Prompt** window, type 'git status'.

```
C:\workspace\BranchMergeRemotes>git status
On branch add-text
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

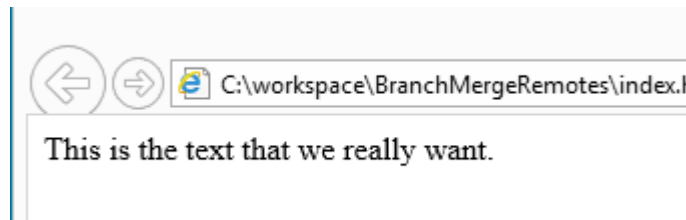
Notice that git now tells us that the file '**index.html**' has been updated, but not staged. We could use the 'git add' command to stage the changed file, but instead we're going to add an option to 'git commit' that will automatically stage all the changes.

__ 6. Type the following and then press return:

```
git commit -a -m "We now have the text we want."
```

```
C:\workspace\BranchMergeRemotes>git commit -a -m "We now have the text we want."
[add-text c28029d] We now have the text we want.
 1 file changed, 1 insertion(+), 1 deletion(-)
C:\workspace\BranchMergeRemotes>_
```

__ 7. In **Windows Explorer**, double-click on '**index.html**' to open it in a web browser.



Notice that the newer text is displayed.

__ 8. Close the web browser.

Part 4 - Merge Back to the Master Branch

We have committed changes to the feature branch that we created and called 'add-text'. Now let's say that we're happy with the feature and want to 'promote' it to the master branch.

The general procedure is as follows:

- Switch to the master branch

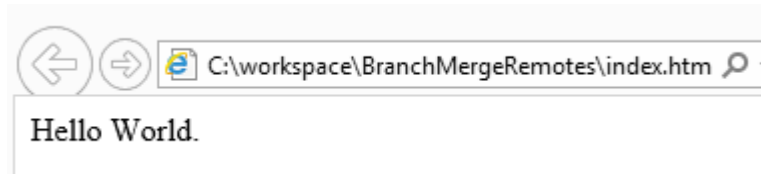
- Merge the changes from the feature branch

- Optionally, delete the feature branch

___1. In the **Command Prompt** window, enter the following line, and then press return:

```
git checkout master
```

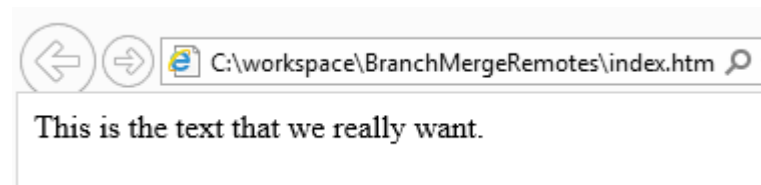
___2. As above, open '**index.html**' with a web browser, but notice that the text now reads 'Hello World' again.



___3. In the Command Prompt window, type 'git merge add-text'.

```
C:\workspace\BranchMergeRemotes>git merge add-text
Updating 152674c..c28029d
Fast-forward
 index.html | 2 + -
 1 file changed, 1 insertion(+), 1 deletion(-)
```

___4. As above, open '**index.html**' with a web browser, but notice that we have the updated text.



___5. We're done with this feature branch, so let's delete it. In the **Command Prompt** window, type 'git branch -d add-text' and then press return.

```
C:\workspace\BranchMergeRemotes>git branch -d add-text
Deleted branch add-text (was c28029d).
C:\workspace\BranchMergeRemotes>_
```

There's no cost the keeping the branch around (remember that a branch is really just a reference to a particular committed snapshot), but it will generally prevent confusion if we keep the branch set clean by deleting branches that we're not actively using.

Part 5 - Push to Another Remote Repository

Let's say we wanted to create a new repository and store all our changes there. In essence, this is what we would do when we first create a project.

As it stands, we have a local repository that contains a number of snapshots. The basic procedure to create a remote repository will be:

- Create an empty remote repository

- Push our changes to that repository

For simplicity, we'll use another repository in the local file system, but as with pulling

from a remote server, the only difference would be the url used for the repository.

__1. In the **Command Prompt** window, enter the following, and then press return:

```
mkdir \workspace\AnotherRemote
```

__2. In the **Windows Explorer** window, navigate to C:/workspace. Notice that there is a new folder called '**AnotherRemote**'.

__3. In the **Command Prompt** window type the following, and then press return:

```
git init --bare \workspace\AnotherRemote
```

The '--bare' option tells git that this repository will never have a working copy. It will only be used as a git repository. This is usually the case for 'remote' repositories that are used to share code. Setting up the repository in this way allows us to "push" changes to it freely, even if our changes would make a working directory obsolete if it existed. Otherwise, we would need to go into that repository and "pull" changes into it.

If you have a look at the contents of **C:\workspace\AnotherRemote**, you'll notice that it's different from the repositories we've already seen – there is no hidden '**.git**' folder. Rather, the contents that would have been in that folder are present in the **AnotherRemote** folder.

__4. In the **Command Prompt** window, type the following on one line, and then press return:

```
git remote add another-remote  
\workspace\AnotherRemote
```

This command tells git to add a reference to a remote repository at 'C:/workspace/AnotherRemote', and to call it 'another-remote' locally.

__5. We can display the remote repository's status. Type '**git remote show another-remote**' and then press return:

```
C:\workspace\BranchMergeRemotes>git remote show another-remote  
* remote another-remote  
Fetch URL: \workspace\AnotherRemote  
Push URL: \workspace\AnotherRemote  
HEAD branch: <unknown>
```

__6. Finally, let's push our data to the remote. Enter '**git push another-remote master**'.

```
C:\workspace\BranchMergeRemotes>git push another-remote master  
Counting objects: 6, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (4/4), done.  
Writing objects: 100% (6/6), 566 bytes | 0 bytes/s, done.  
Total 6 (delta 0), reused 0 (delta 0)  
To \workspace\AnotherRemote  
* [new branch] master -> master
```

__7. Once again, enter '**git remote show another-remote**'

```
C:\workspace\BranchMergeRemotes>git remote show another-remote
* remote another-remote
Fetch URL: \workspace\AnotherRemote
Push URL: \workspace\AnotherRemote
HEAD branch: master
Remote branch:
  master tracked
Local ref configured for 'git push':
  master pushes to master (up to date)
```

Notice that this time, the remote status indicates that the remote HEAD branch is 'master' and that we are tracking the remote master branch.

__8. Close the git command and all browsers.

Part 6 - Review

In this lab, we tried out a common workflow, the "feature branch" workflow. We checked out a project stored in a remote repository. We created a new branch, did some work on it, committed it, and then merged it back onto the master branch. Finally, we created yet another remote repository and pushed our changes to it.

The thing to notice here is that using remote repositories is natural in git. In fact, the "remote" repository is basically identical to the "local" repository – in this case, we used simple repositories in the file system, but they could equally well have been accessed remotely using secure-shell or a repository manager.

Lab 3 - Experimenting with Workflows

In git, it's easy to create multiple repositories that contain a given project, and to move information between them. As a result, there are a variety of possible workflows, or ways of coordinating the efforts of more than one developer. This lab explores two of those workflows – the 'centralized' workflow and the 'integration manager' workflow.

At the end of this lab you will be able to:

1. Use a centralized repository in the same way as you would with non-distributed version control system
2. Use a "pull-request" style of development, as is common with git.

Part 1 - Simulate Two Developers

We're going to pretend that in addition to our initial developer, "Alice Smith", we now have a second developer, "Chuck Eastman". Chuck's email address will be "chuckeast@hotmail.com". To simulate this, we'll setup repository-specific configuration on two separate repositories.

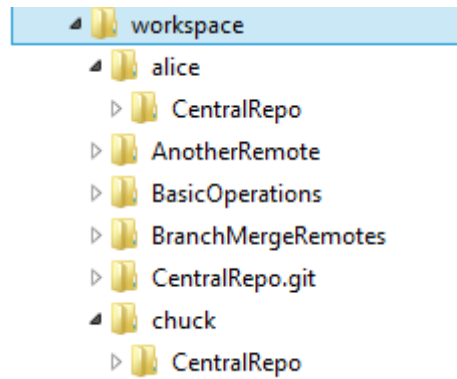
- __ 1. Open a **Command Prompt** window.
- __ 2. In the **Command Prompt** window, type the following lines, pressing return after each line:

```
cd \workspace  
mkdir alice  
mkdir chuck
```

These commands create folders named '**alice**' and '**chuck**' inside the '**workspace**' folder. Inside each of these folders, we're going to clone a central repository, and then perform some experiments.

- __ 3. Type '**git clone --bare \LabFiles\CentralRepo**'. This will create a copy of the CentralRepo that is supplied in the LabFiles folder, so that we can commit to it without changing the supplied repository (in case you want to repeat the lab at some point).
- __ 4. In the **Command Prompt** window, enter '**cd alice**', to change to Alice's directory.
- __ 5. Type '**git clone \workspace\CentralRepo.git**'.
- __ 6. Type '**cd ..**' to return to the workspace folder.
- __ 7. Type '**cd chuck**' to change to Chuck's directory.
- __ 8. Type '**git clone \workspace\CentralRepo.git**'.
- __ 9. Type '**cd ..**' to change back to the workspace folder.

We now have a folder structure similar to the following:



The last thing we need to do in order to setup this simulation is to set the user's identity in each repository.

__10. Enter the following lines, pressing return after each line:

```
cd alice\CentralRepo
git config user.name "Alice Smith"
git config user.email alice@smith.com
cd ..\..
cd chuck\CentralRepo
git config user.name "Chuck Eastman"
git config user.email chucked@hotmai.com
```

Notice that we used '**git config**' but without the '--global' flag, to change the settings just for the current repository.

Part 2 - Try Out the Centralized Work Flow

It's possible to use git in a way that's very much like the centralized workflow that you would use with older version control systems. Even if you manage your repository in a centralized fashion, git still has advantages – primarily, that developers can work and continue to commit changes to their repository even while disconnected from the internet or any central repository. For example, we could have a repository that is available on the office network, but not the public Internet; developers would still be able to commit to their local repository if working from home, or on an airplane.

In this scenario, you (a developer) commit your work to your local repository, and then on a regular basis, you also push your working branch to the central repository. If another developer has made an incompatible change, then your push is rejected, and you have to pull down the other developer's changes and merge them manually. After that, you can attempt again to push to the central repository.

Recall that in the previous part of the lab, we setup 'Alice' and 'Chuck', each with a clone of the 'CentralRepository'.

- __1. In Windows Explorer, navigate to C:\workspace\alice\CentralRepo
- __2. Right-click on **index.html** and then select **Open With** → **Notepad**.
- __3. Edit the file so that the line inside the <body> element reads 'This is how Alice wants it.'

```
<!DOCTYPE html>

<html>
<body>
This is how Alice wants it.
</body>
</html>
```

__4. Save and close the file.

__5. In the **command prompt** window, go to Alice's CentralRepo folder by typing the following:

```
cd \workspace\alice\CentralRepo
```

__6. Enter the following, to commit Alice's changes:

```
git commit -a -m "Alice now has her way."
```

Git reports the successful commit...

```
C:\workspace\alice\CentralRepo>git commit -a -m "Alice now has her way."
[master bfaf161] Alice now has her way.
1 file changed, 1 insertion(+), 1 deletion(-)
```

At this point, Alice has made a change and committed to her local repository. Now let's push this to the CentralRepository.

__7. In the **Command Prompt** window, enter:

```
git push
```

Git will print out some information about the default push behavior, and then push the changes to the central repository.

```
C:\workspace\alice\CentralRepo>git push
warning: push.default is unset; its implicit value is changing in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the current behavior after the default changes, use:

    git config --global push.default matching

To squelch this message and adopt the new behavior now, use:

    git config --global push.default simple

When push.default is set to 'matching', git will push local branches
to the remote branches that already exist with the same name.

In Git 2.0, Git will default to the more conservative 'simple'
behavior, which only pushes the current branch to the corresponding
remote branch that 'git pull' uses to update the current branch.

See 'git help config' and search for 'push.default' for further information.
(the 'simple' mode was introduced in Git 1.7.11. Use the similar mode
'current' instead of 'simple' if you sometimes use older versions of Git)

Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 305 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To \workspace\CentralRepo.git
 bdb807b..bfaf161 master -> master
```

__8. We'll take git's advice and set the global push behavior to 'simple' in order to avoid this message in the future. Enter the following:

```
git config --global push.default simple
```

__9. At this point, Alice has pushed her changes to the central repository. Now let's have Chuck do some work.

__10. In Windows Explorer, navigate to C:\workspace\chuck\CentralRepo

__11. Right-click on **index.html** and then select **Open With** → **Notepad**.

__12. Edit the file so that the line inside the <body> element reads 'This is how Chuck wants it.'

```
<!DOCTYPE html>

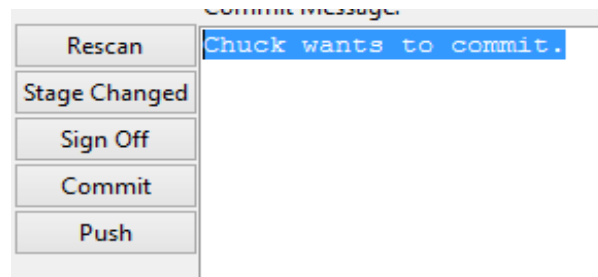
<html>
<body>
This is how Chuck wants it.
</body>
</html>
```

__13. Save and close the file.

__14. Git began life as a Linux command line program, and arguably, the command line is the preferred way of interacting with git. But just once, let's try out the GUI interface. Right-click in the empty area of **Windows Explorer** and then select **Git Commit Tool**. Notice that the change to **index.html** shows as an unstaged change.

__15. Click the **Stage Changed** button.

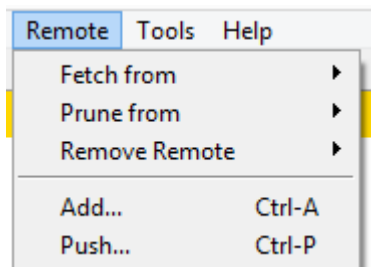
__16. Enter a commit message like "Chuck wants to commit.", and then click **Commit**.



The **Commit Tool** window will close. At this point, Chuck has made a change and committed to his local repository. Now let's push this to the CentralRepository.

__17. Right-click in the empty area of **Windows Explorer**, and then select **Git GUI**.

__18. The Git GUI window is displayed. From the main menu, select **Remote** → **Push...**



__19. The **Push** dialog appears. Leave the defaults as-is, and then click **Push**.

__20. Now we have a bit of a problem. Git reports a command failure.

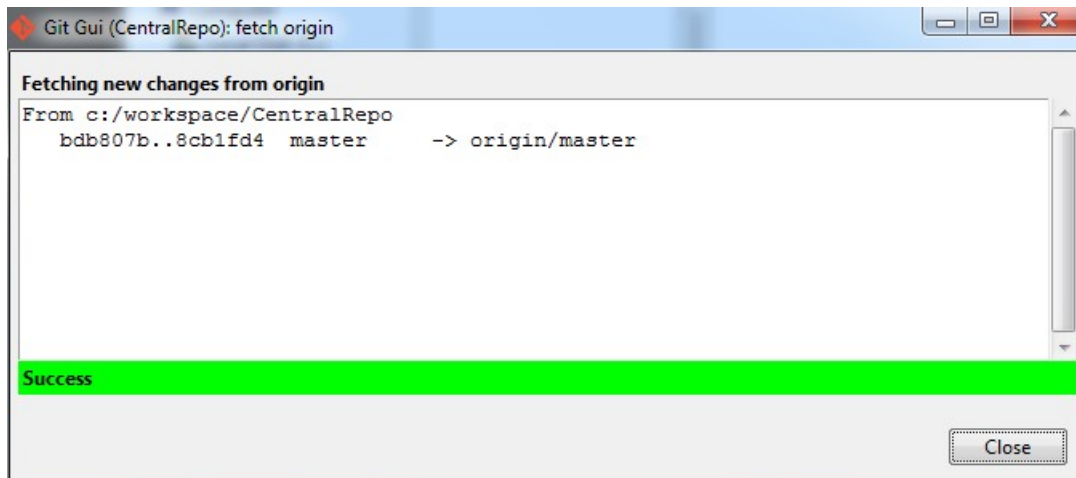


The problem here is that Chuck's changes are based on the version that was checked out of the CentralRepository, but in the meantime, Alice got in with a conflicting change to the file. As a result, Chuck's push to the CentralRepository is disallowed. He needs to fetch the changes, merge then, and then commit again.

__21. Click **Close** in the dialog that was reporting the merge failure.

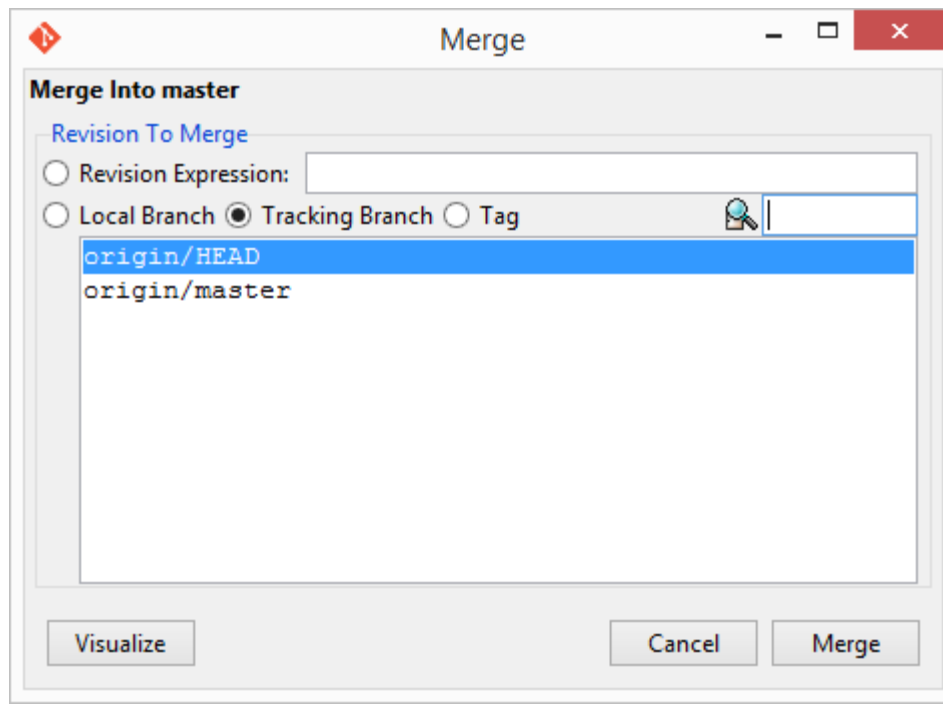
__22. In the Git GUI, select Remote → Fetch From → origin

__23. Close the resulting 'Success' window

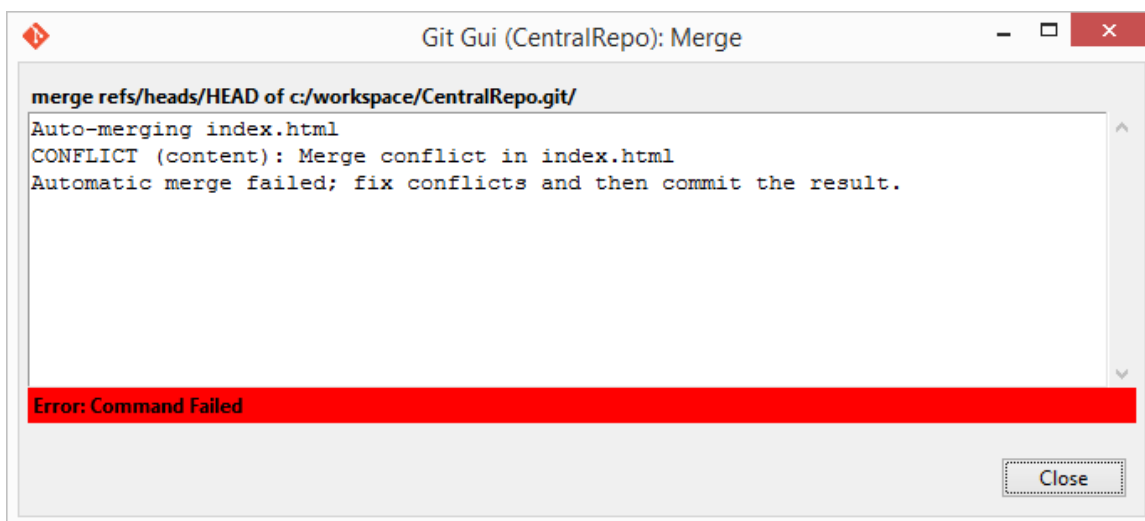


__24. Select Merge → Local Merge...

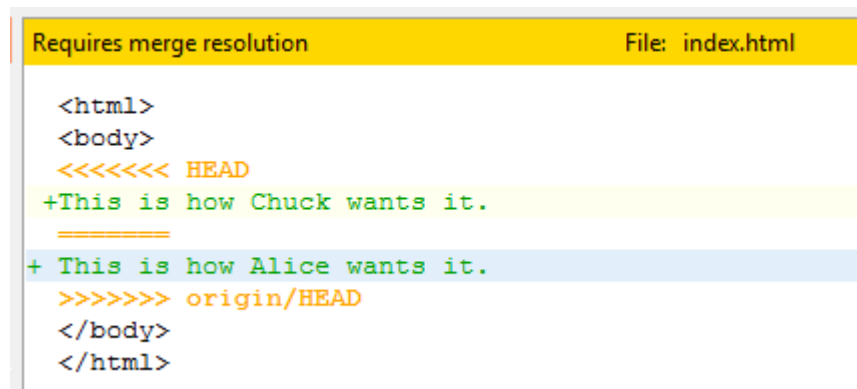
__25. The **Merge** dialog is displayed. Notice that it is defaulting to merge the remote branch '**origin/HEAD**' to the local branch '**master**'. This is what we want, so click on **Merge**.



26. The GUI reports a merge conflict:



27. Click **Close**. The GUI will now show that we have a conflict and invite us to resolve the conflict.



__28. Close Git GUI.

In this case, we'll simply replace the body with 'This is a compromise between Chuck and Alice.'

__29. Open the file again with Notepad, and edit the body so it looks like:

```
<!DOCTYPE html>

<html>
<body>
This is a compromise between Chuck and Alice.
</body>
</html>
```

__30. Save and close the file.

__31. In the **Command Prompt** window, enter the following to navigate to Chuck's copy of the CentralRepo repository:

```
cd \workspace\chuck\CentralRepo
```

__32. Type '**git add index.html**' and hit enter.

Note: If you're used to other version control systems like 'cvs' or Subversion, the command 'git add' may seem a little odd. In many VCS, you would use an 'add' command to tell the system to begin tracking changes on the file. In that case, it wouldn't make sense to tell the VCS to add the file again.

However, Git does things a little differently. Git isn't tracking files individually, it's taking snapshots of the entire working copy. It uses the "index" to list which files are added to the record for each commit. So, here we're saying "add index.html to the list of files in this commit". In case of a merge like this, adding the file to the index tells git that "this is the version we want to commit". In other words we're using 'git add' to tell git that we've resolved the conflict.

__33. Type '**git commit**' and press enter.

__34. Git will open WordPad with the commit message. Notice that git has already filled in a message saying that this commit is a "merge", with details of the merge. Close WordPad to let the commit continue.

```
C:\workspace\chuck\CentralRepo>git commit
[master 34b6893] Merge branch 'HEAD' of \workspace\CentralRepo.git
```

Now we have resolved the merge conflict locally, but we still need to push to the central repository.

__35. In the **Command Prompt** window, type '**Git push**' and then press enter (Recall that this was the step that failed when we tried it last time in the GUI).

```
C:\workspace\chuck\CentralRepo>git push
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 689 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To \workspace\CentralRepo.git
bfaf161..34b6893 master -> master
```

Git now reports a successful push.

The thing to notice in this example is that the developers had to do their own merge. In this case, Chuck had to integrate his changes on top of Alice's, and then push the final merge to the central repository. Indirectly, this means Chuck needs to be aware of Alice's changes, and they end up indirectly working together.

Part 3 - Try Out the Integration Manager Work Flow

In this work flow, the developers do their work independently, and then their changes are incorporated into a third repository by someone acting as an Integration Manager. To simulate this scenario, we won't worry about the identity of the integration manager, we'll just demonstrate the idea of "pulling" changes into a third repository.

__1. In Command Prompt, navigate to C:\workspace\alice\CentralRepo by typing:

```
cd \workspace\alice\CentralRepo
```

__2. First, let's make sure we're up-to-date with the **CentralRepo** (remember that in the previous part of the lab, Chuck made and integrated some changes after Alice made changes). In the **Command Prompt** window, type '**git status**' and press return.

```
C:\workspace\alice\CentralRepo>git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

'git status' appears to report that we're up-to-date, but just to be sure, let's do a fetch from the **CentralRepo**.

__3. Enter '**git pull origin**' and then press return.

```
C:\workspace\alice\CentralRepo>git pull origin
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From \workspace\CentralRepo
bfaf161..34b6893 master -> origin/master
Updating bfaf161..34b6893
Fast-forward
 index.html | 2 + -
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Pulling from the original repository has actually caused a file to be changed. Had we made local changes, we would have been working from an outdated starting point. That is actually manageable in most cases, but it's simpler if we start from the latest updates.

__4. In Windows Explorer, right-click on **c:\workspace\alice\CentralRepo\index.html** and then select Open With → Notepad.

__5. Add a line before **</body>** that reads "Another change by Alice." When the file

appears as below, save and close the file.

```
<!DOCTYPE html>

<html>
<body>
This is a compromise between Chuck and Alice.
Another change by Alice.
</body>
</html>
```

__6. Back in the **Command Prompt** window, enter

```
git commit -a -m "Alice added a comment."
```

Because we're just committing to the local repository, there's no possibility of a merge conflict, so the commit is successful.

```
C:\workspace\alice\CentralRepo>git commit -a -m "Alice added a comment."
[master 33b97b3] Alice added a comment.
1 file changed, 1 insertion(+)
```

__7. We'll now do a change to Chuck's working copy. In the **Command Prompt** window, enter:

```
cd \workspace\chuck\CentralRepo
```

__8. In the Windows Explorer window, navigate to C:\workspace\chuck\CentralRepo

__9. In the **Command Prompt** window, enter '**git pull origin**' and press return. This command will ensure that Chuck also is starting from the latest updates.

__10. In **Windows Explorer**, open the file **index.html** with Notepad, and add a line just after <body> that says "Here's a Chuck line." When the file appears as below, save and close the file.

```
<!DOCTYPE html>

<html>
<body>
Here's a Chuck line.
This is a compromise between Chuck and Alice.
</body>
</html>
```

__11. In the **Command Prompt** window, commit the changes, by entering:

```
git commit -a -m "Chuck added a line."
```

Again, there is no possibility of a merge failure, so the commit succeeds.

Now, let's assume that both Alice and Chuck have talked to the person acting as

Integration Manager, and essentially said "Please pull the changes from the main branch of my repository.

__12. In the **Command Prompt** window, enter the following lines, pressing return after each line:

```
cd \workspace
mkdir integration
cd integration
git clone \workspace\CentralRepo.git
cd CentralRepo
```

These steps create a folder called 'integration' and create clone of the CentralRepo in that folder, then move into that clone. Now we can pull the changes from both Alice's and Chucks' repositories into this repo.

__13. Enter the following in the **Command Prompt** window

```
git remote add alice \workspace\alice\CentralRepo
```

This line adds Alice's repository as a remote source in the integration manager's clone of CentralRepo

__14. Enter the following:

```
git pull alice master

C:\workspace\integration\CentralRepo>git pull alice master
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From \workspace\alice\CentralRepo
 * branch            master       -> FETCH_HEAD
 * [new branch]      master       -> alice/master
Updating 34b6893..33b97b3
Fast-forward
 index.html | 1 +
 1 file changed, 1 insertion(+)
```

Notice that the pull indicates one file changed. That's Alice's work successfully integrated. Let's go ahead and push that to the parent repository.

__15. In the **Command Prompt** window, enter '**git push origin master**' and press return.

Now let's integrate Chuck's work.

__16. Enter the following in the **Command Prompt** window:

```
git remote add chuck \workspace\chuck\CentralRepo
```

This line adds Chuck's repository as a remote source in the integration manager's clone of CentralRepo

__17. Enter the following:

```
git pull chuck master
```

```

C:\workspace\integration\CentralRepo>git pull chuck master
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From \workspace\chuck\CentralRepo
 * branch          master      -> FETCH_HEAD
 * [new branch]    master      -> chuck/master
Auto-merging index.html
Merge made by the 'recursive' strategy.
index.html | 1 +
1 file changed, 1 insertion(+)

```

Notice that this time, there appears to have been a merge-commit performed. This is because Chuck's changes were made against an earlier version of the central repo, and need to be merged in.

Interestingly, there is no merge conflict. If you have a look at '**index.html**', you'll find the following:

```

<!DOCTYPE html>
<html>
<body>
Here's a Chuck line.
This is a compromise between Chuck and Alice.
Another change by Alice.
</body>
</html>

```

Notice that git was able to merge Chuck and Alice's changes without a conflict, since they were trivial changes made at different spots in the file (Chuck's change was after <body>; Alice's change was before </body>).

Let's go ahead and push the final result to the parent repository

___18. In the **Command Prompt** window, enter '**git push origin master**' and press return.

So the end result is the same: we had two developers do some work that was successfully integrated into the central repository. In this case, however, any merge issues were evaluated and handled by a person acting as Integration Manager, so the developers, Alice and Chuck, did not have to deal with each other's work.

Part 4 - Review

In this lab, we sampled two common workflows: Central Repository and Integration Manager. The Central Repository workflow is well-suited to development groups that are used to using older version control tools, as well as smaller groups. The Integration Manager approach (as well as more complicated variants) can be used to manage and integrate changes from large numbers of developers with relative ease.

Lab 4 - Build Script Basics

Gradle was already configured for you under C:\Software\gradle-3.2.1 folder and added in the environment variables; Gradle can be downloaded by following this URL: <https://gradle.org/gradle-download/>. **Complete Distribution** contains the binary files, documentation, and source code. If you prefer, you can just download the binary files.

In this lab you will build a Java Project, manage dependencies, and run unit tests with Gradle.

Part 1 - Gradle installation verification

- __1. Open a Command Prompt.
- __2. Verify Gradle is installed.

```
gradle -version
```

In this lab you will create build scripts and utilize various features.

Part 2 - Create directory structure for storing build script

In this part you will create directory structure for storing build script.

- __1. Switch to the **Software** directory.

```
cd C:\Software
```

- __2. Create the directory structure.

```
mkdir labs\gradle\basics
```

- __3. Switch to the newly created directory.

```
cd labs\gradle\basics
```

Part 3 - Create Gradle build script and use shortcut task definition technique for defining a task

In this part you will create a Gradle build script and add a task using shortcut task definition technique.

__1. Create build.gradle file.

```
notepad build.gradle
```

__2. Click Yes to create the file.

__3. Enter following code.

```
// let's add a simple task
task hello1 << {
    println 'Hello World!'
}
```

Here you have added a single-line comment and a task which displays a message on the screen.

__4. Save the file.

__5. Run Gradle.

```
gradle
```

Notice it's asking us to pass it more parameters.

```
C:\Software\labs\gradle\basics>gradle
The Task.leftShift(Closure) method has been deprecated and is scheduled to be removed in Gradle 5.0. Please use Task.doLast(Action) instead.
    at build_etnkc1jr8c6i2lq9u6bhluc5e.run(C:\Software\labs\gradle\basics\build.gradle:2)
:help

Welcome to Gradle 3.2.1.

To run a build, run gradle <task> ...

To see a list of available tasks, run gradle tasks

To see a list of command-line options, run gradle --help

To see more detail about a task, run gradle help --task <task>

BUILD SUCCESSFUL
```

__6. Get task list.

```
gradle tasks
```

Notice it lists hello1 under "Other tasks" section.

__7. Execute hello1 task.

```
gradle hello1
```


Notice it displays "Hello World!" message on the screen, but there are additional messages logged as well.

__ 8. Execute the task by suppressing the log.

```
gradle -q hello1
```

Notice this time it just displays the message.

In this part you used << when defining the task. It's shortcut task definition technique. It's deprecated feature and won't be there in future versions. Instead of using shortcut definition technique you will want to use action based technique. You will use the action based technique in the next part of this lab.

Part 4 - Create a task with actions

In this part you will create another task which uses actions.

__ 1. In the Notepad window, which contains build.gradle code, append following code.

```
// add another task with actions
task hello2 {
    doFirst {
        print 'This is '
    }
    doLast {
        println 'a test!'
    }
}
```

Note: doFirst and doLast are the predefined actions. As the name suggests, doFirst code is executed before doLast. You don't need to use both together. You can use one or the other.

__ 2. Save the code.

__ 3. Execute hello2 task.

```
gradle -q hello2
```

Notice it displays "This is a test!" message on the screen.

Part 5 - Define Task Dependency

In this part you will define task dependency. A task will make use of another task.

__ 1. In the Notepad window, which contains build.gradle code, append following code.

```
task welcome(dependsOn: hello1) {
    doLast {
```

```
        println "Welcome Bob!"
    }
}
```

Note: welcome task depends on hello1. It will execute hello1 then welcome task. In this case hello1 task already exists. In case if you define 'welcome' task before defining the dependent hello1 task, then use following syntax.

```
task welcome(dependsOn: 'hello1')
```

Notice here you have used quotes for defining the dependent task. This technique is also called Lazy dependsOn.

__2. Save the code.

__3. Execute welcome task.

```
gradle -q welcome
```

Notice it displays Hello World! followed by Hello Bob!

Part 6 - Alternative technique for defining task dependency

In this part you will use a different technique for defining task dependency.

__1. In the Notepad window, which contains build.gradle code, append following code.

```
task hello {
}

hello.dependsOn hello1, hello2
```

Note: hello task depends on hello1 and hello2. It will execute hello1, hello2, then if hello task.

__2. Save the code.

__3. Execute hello task.

```
gradle -q hello
```

Notice it displays Hello World! followed by This is a test!

Part 7 - Using Task Properties

In this part you will define task properties and reuse them in other tasks.

__1. In the Notepad window, which contains build.gradle code, append following code.

```
task myProperties {  
    ext.greeting = "Hello, "  
    ext.userName = "Bob"  
}
```

Note: Here you have created a custom task named myProperties and added 2 properties to it. Task name can be anything, but you must use ext object to add properties to the task.

__2. Append follow code to reuse the properties.

```
task useProperties {  
    doLast {  
        println myProperties.greeting + myProperties.userName  
    }  
}
```

Note: Here you have used the properties by specifying the task name.

__3. Save the code.

__4. Execute useProperties task.

```
gradle -q useProperties
```

Notice it displays Hello, Bob message on the screen.

Part 8 - Create a method and reuse it in Gradle build script

In this part you will create a method and then reuse it in the Gradle script.

__ 1. In the Notepad window, which contains build.gradle code, append following code.

```
String toUpper(String userName) {  
    userName.toUpperCase()  
}
```

Note: This method takes string based input, converts it to upper case, and returns it.

__ 2. Append following code to reuse the method.

```
task callMethod {  
    doLast {  
        println myProperties.greeting + toUpper(myProperties.userName)  
    }  
}
```

Note: The custom toUpper method is called by the doLast action of custom task.

__ 3. Save the code.

__ 4. Execute the callMethod task.

```
gradle -q callMethod
```

Notice it displays Hello,BOB message on the screen.

Part 9 - Defining default tasks

In this part you will set default tasks so they should execute even without specifying them as part of command-line arguments.

__ 1. In the Notepad window, which contains build.gradle code, append following code.

```
task defaultTask1 {  
    doLast {  
        println 'Default Task 1!'  
    }  
}  
  
task defaultTask2 {  
    doLast {  
        println 'Default Task 2!'  
    }  
}
```

Note: Here you have defined 2 custom tasks. Next, you will set them as default tasks.

__ 2. In the beginning of the file, add following code.

```
defaultTasks 'defaultTask1', 'defaultTask2'
```

- __ 3. Save the code.
- __ 4. Execute the build script without specifying any task name.

```
gradle -q
```

Notice it displays Default Task 1! Default Task 2! message on the screen.

Part 10 - Create Tasks Dynamically

In this part you will create tasks dynamically by using a loop.

- __ 1. In the Notepad window, which contains build.gradle code, append following code.

```
3.times { i ->
    task "task$i" {
        doLast {
            println "Task #: $i"
        }
    }
}
```

Note: Here you are using groovy syntax to define a loop which runs 3 times. The current index location is stored in variable i. String interpolation is using to retrieve value of i and append it to "task".

- __ 2. Append following code.

```
task allTasks {
}

allTasks.dependsOn task0, task1, task2
```

Note: Here you have created allTasks which aggregates task0, task1, and task2.

- __ 3. Save the file.
- __ 4. Execute allTasks.

```
gradle -q allTasks
```

Notice it displays Task #0, Task #1, Task #2 message on the screen.

Part 11 - Using list and .each loop

In this part you will create a list, use each loop to go over it, create directories, and delete directories.

- __ 1. In the Notepad window, which contains build.gradle code, append following code.
- __ 2. Append following code.

```

task createDirs {
    doLast {
        ext.myDirList = ["dir1", "dir2"]
        ext.myDirList.each() {
            new File("${it}").mkdir()
        }
    }
}

```

Note: You have used ext object to create a custom list of string then used each loop to iterate over the items and created directory for each item. `${it}` is a special variable which contains the current item being processed by the each loop.

__3. Save the file.

__4. Execute the task.

```
gradle -q createDirs
```

__5. Get directory list.

```
dir
```

Notice dir1 and dir2 are created.

__6. In the Notepad window, which contains build.gradle code, append following code.

```

task removeDirs {
    doLast {
        ext.myDirList = ["dir1", "dir2"]
        ext.myDirList.each() {
            new File("${it}").delete()
        }
    }
}

```

Note: In the code you are removing the directories which you created previously.

__7. Save the file.

__8. Execute the task.

```
gradle -q removeDirs
```

__9. Get the directory list.

```
dir
```

Notice dir1 and dir2 are removed.

__10. Close all.

Part 12 - Review

In this lab you utilized core features of groovy in build scripts.

Lab 5 - Build Java Project, Management Package Dependencies, and Run Unit Tests with Gradle

Part 1 - Create directory structure for storing project files

In this part you will create directory structure for storing Java files. It's the same structure used by Maven as well.

- __ 1. Open a Command Prompt.
- __ 2. Switch to the **Software\labs\gradle** directory.

```
cd C:\Software\labs\gradle
```

- __ 3. Create directory structure for storing Java code.

```
mkdir MyProject\src\main\java\hello
```

MyProject is the name of your project. It's the base directory you will use for storing your project files.

main and java directories must exist with same names. hello is custom Java package name.

- __ 4. Switch to MyProject directory.

```
cd MyProject
```

- __ 5. View directory tree.

```
tree
```

```
├── src
│   └── main
│       └── java
│           └── hello
```

Part 2 - Create a Java project

In this part you will write simple Java code. There will be 2 files. One will contain a custom class and second will contain the main function which make use of the custom class.

__1. Create Greeter.java by using Notepad. (Note: Click 'Yes', if prompted to do so)

```
notepad src\main\java\hello\Greeter.java
```

__2. Enter following code.

```
package hello;

public class Greeter {
    public String sayHello() {
        return "Hello world!";
    }
}
```

__3. Save the file and close Notepad window.

__4. Create HelloWorld.java file by using Notepad. (Note: Click 'Yes', if prompted to do so)

```
notepad src\main\java\hello\HelloWorld.java
```

__5. Enter following code.

```
package hello;

public class HelloWorld {
    public static void main(String[] args) {
        Greeter greeter = new Greeter();
        System.out.println(greeter.sayHello());
    }
}
```

__6. Save the file and close Notepad window.

Part 3 - Create Gradle build script

In this part you will create a Gradle build script.

__1. Run Gradle, without creating the build script, and get tasks list.

```
gradle tasks
```

Notice there is no section labeled "Build tasks" containing build, clean, clean, ... tasks.

__2. Create build.gradle file. (Note: Click 'Yes', if prompted to do so)

```
notepad build.gradle
```

__3. Enter following code.

```
apply plugin: 'java'
```

You have included Java plugin which will make more tasks available to Gradle.

__ 4. Save and close the file.

__ 5. Get Gradle tasks list again.

```
gradle tasks
```

Notice there's a new section available with bunch of useful Java related tasks.

```
Build tasks
-----
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles test classes.
```

Part 4 - Build and Clean the Gradle generated files

In this part you will build the Java project with Gradle, run the generated compiled code, and clean the files.

__ 1. Build the project using Gradle.

```
gradle build
```

__ 2. View "build" directory tree.

```
tree build
```

```
├── classes
│   ├── main
│   │   └── hello
├── dependency-cache
├── libs
├── tmp
│   ├── compileJava
│   └── jar
```

Here are some important directories:

* classes: compiled .class files are generated here.

* libs: jar file is stored here.

__ 3. Run the compiled class in the jar file.

```
java -cp build\libs\MyProject.jar hello.HelloWorld
```

Notice it displays Hello World! message on the screen.

__4. Clean the build generated files.

```
gradle clean
```

__5. Get directory list.

```
dir
```

Notice build directory is gone.

Part 5 - Dependency Management

In this part you will include package dependency and then manage it using the Gradle build script.

__1. Edit HelloWorld.java file.

```
notepad src\main\java\hello\HelloWorld.java
```

__2. Below "package hello;" add following code.

```
import org.joda.time.LocalTime;
```

Note: Although you can use native Java classes for obtaining date and time, but just so you can see dependency management you will utilize a 3rd party package, Joda, for obtaining the date and time.

__3. Above "Greeter greeter = new Greeter();" add following code.

```
LocalTime currentTime = new LocalTime();  
System.out.println("The crrent time is: " + currentTime);
```

__4. Save and close the file.

__5. Try to build the project with Gradle.

```
gradle build
```

Notice the build has failed. It's unable to find LocalTime class dependency.

__6. Edit the build script.

```
notepad build.gradle
```

__7. Append following code.

```
repositories {  
    mavenLocal()  
    mavenCentral()  
}
```

Note: Here you are adding local and online Maven Central repository (<http://search.maven.org>) from where the dependencies will be downloaded from. You can also use other repositories, such as Jcenter(<https://bintray.com/bintray/jcenter>). Custom repositories can also be defined like this:

```
repositories {  
    maven {  
        url "http://repo.mycompany.com/maven2"  
    }  
}
```

__8. Below repository add following code to define source and target Java version. (Note: this step is optional)

```
sourceCompatibility = 1.7  
targetCompatibility = 1.7
```

__9. Add following code to add Joda time 2.2 package dependency.

```
dependencies {  
    compile "joda-time:joda-time:2.2"  
}
```

__10. Save and close the file.

__11. Run Gradle build.

```
gradle build
```

Notice there are no errors this time.

Part 6 - Using Application Plugin

In this part you will use Application plugin. It makes more tasks available to Gradle. You can use it for executing your application.

__1. Edit the build script file.

```
notepad build.gradle
```

__2. Add following line in the beginning of the file.

```
apply plugin: 'application'
```

__3. Add following code in the end of the file.

```
mainClassName = "hello.HelloWorld"
```

Note: Here you have specified the main class which should be executed.

__4. Save and close the file.

__5. Execute the build script.

```
gradle build
```

__6. Run the application.

```
gradle -q run
```

Notice you have used the run task made available by Application plugin. Also notice it displays current date & time and Hello world! message on the screen.

Part 7 - Running unit tests with Grade

In this part you will write a JUnit test and run it with Gradle.

__1. Create directory structure for storing unit test.

```
mkdir src\test\java\hello
```

__2. Create a unit test. (Note: Click 'Yes', if prompted to do so)

```
notepad src\test\java\hello\TestGreeting.java
```

__3. Enter following code.

```
package hello;

import org.junit.Assert;
import org.junit.Test;

import hello.Greeter;

public class TestGreeting {
    @Test
    public void testGreeter() {
        Greeter msg = new Greeter();
        Assert.assertEquals("Hello world!",
msg.sayHello());
    }
}
```

__4. Save and close the file.

__5. Run the test.

```
gradle test
```

Notice it displays error message that JUnit is not recognized. You need to add JUnit dependency in order to make it work. You will do it in the next step.

__6. Open build.gradle file.

```
notepad build.gradle
```

__7. In dependencies section, after compile "joda-time:joda-time:2.2", add following line.

```
testCompile "junit:junit:4.12"
```

__8. Save and close the file.

__9. Run test again.

```
gradle test
```

Notice test is executed successfully and no error is displayed.

__10. View test result.

```
notepad build\test-results\test\TEST-hello.TestGreeting.xml
```

Notice testGreeter test case is listed. It also shows time it took to run the test.

__11. Close the Notepad window and command prompt.

```
exit
```

Part 8 - Review

In this lab you built a Java Project, managed dependencies, and ran unit tests with Gradle.

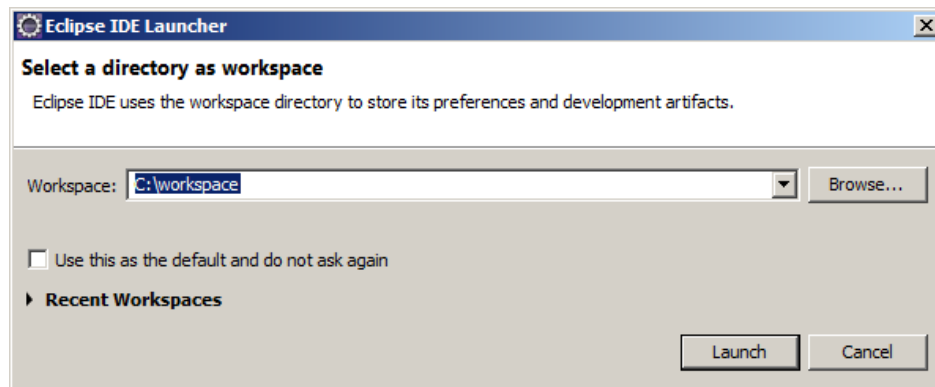
Lab 6 - Integrate Eclipse with GitLab and Gradle

In this you will take an existing application that uses Gradle and import it into Eclipse. Next you will use Eclipse Gradle tools (Buildship) to build, test and run the application. Next you will use Eclipse Git tools (Egit) to integrate with GitLab.com

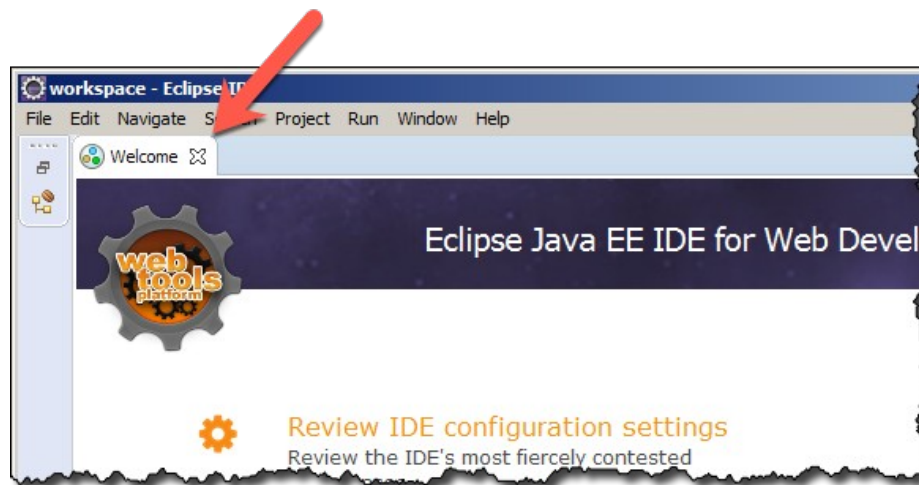
Part 1 - Import the application into Eclipse

In this part you will import the application, that you built in Lab 5, into Eclipse

- __1. Use Windows Explorer and open Eclipse: C:\Software\eclipse\eclipse.exe (check with your instructor if you cannot find eclipse location).
- __2. Select the workspace directory C:\workspace and click **Launch**.

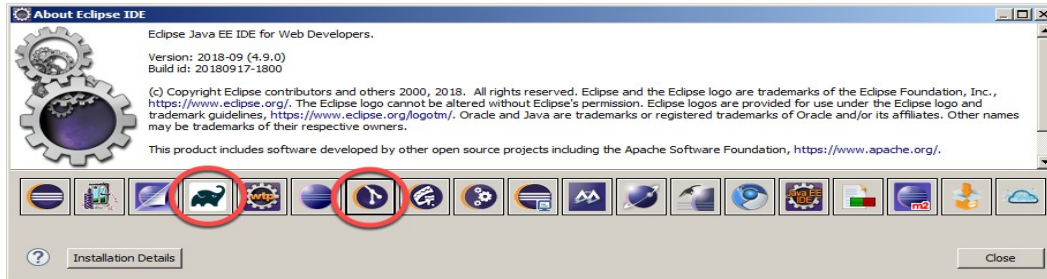


- __3. Click the 'x' to close the Welcome screen



- __4. Choose **Help** → **About Eclipse** from the menu.

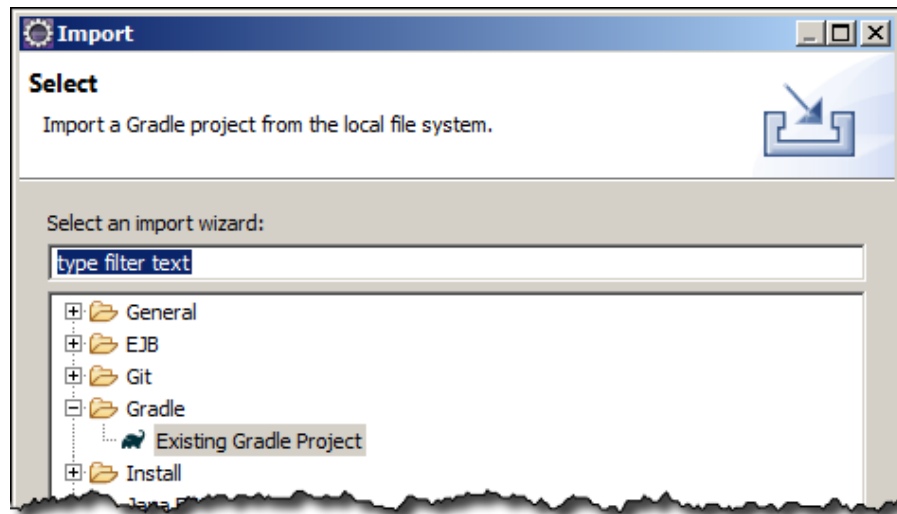
__5. Click on the two icons circled below to verify that *Buildship* (Gradle tools) and *Egit* (Git tools) are installed.



__6. Close the About Eclipse dialog.

__7. Choose **File** → **Import ...** from the menu.

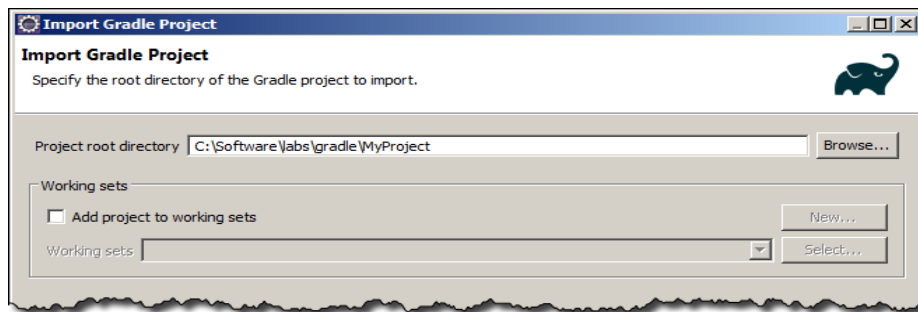
__8. Select **Gradle** → **Existing Gradle Project** and click **Next**.



__9. You may ignore the *How to Experience the Best Gradle Integration* page, click **Next**.

__10. Select the *Project root directory* (the directory you used in a previous Gradle Lab)

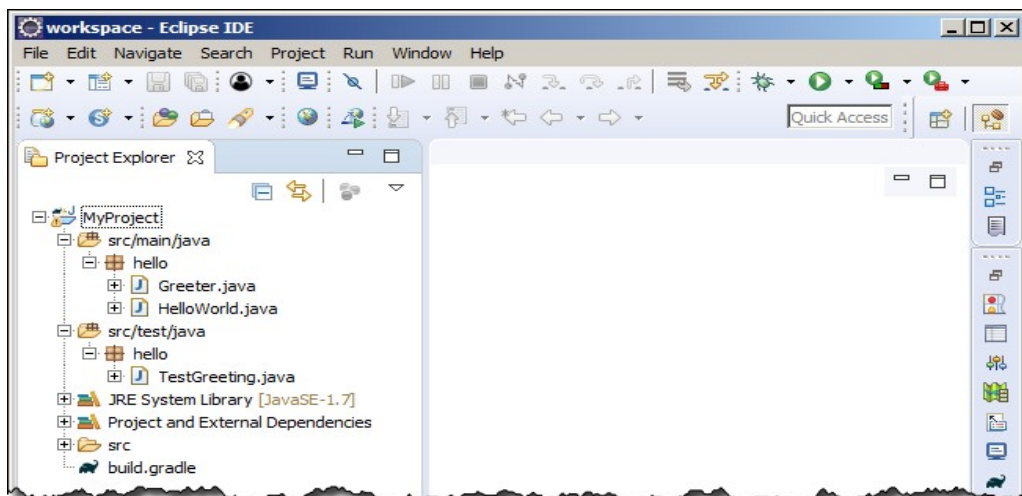
C:\Software\labs\gradle\MyProject



__11. Click **Finish**.

After a few seconds you should see **MyProject** in the **Project Explorer** pane.

__12. Expand **src/main/java** and **src/test/java** to see the project.

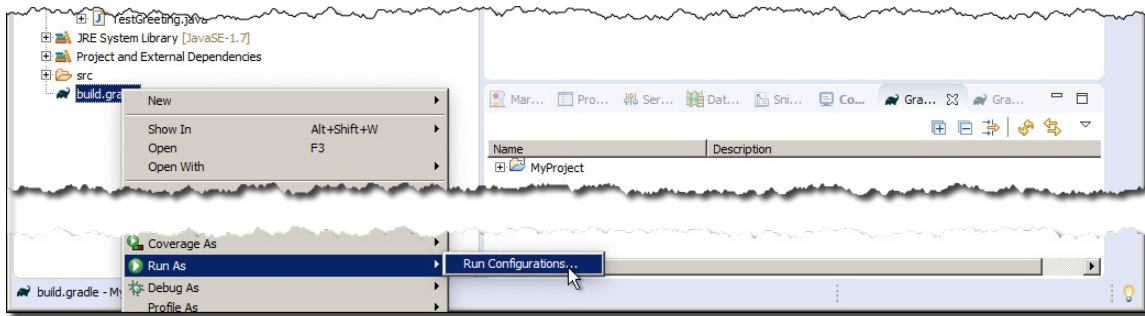


Part 2 - Build, Test and Run the Application with Gradle in Eclipse

In this section you will use Eclipse and Gradle to build, test and run the application.

__ 1. Right click on **build.gradle** in the *Project Explorer* window.

__ 2. Select **Run As** → **Run Configurations ...**



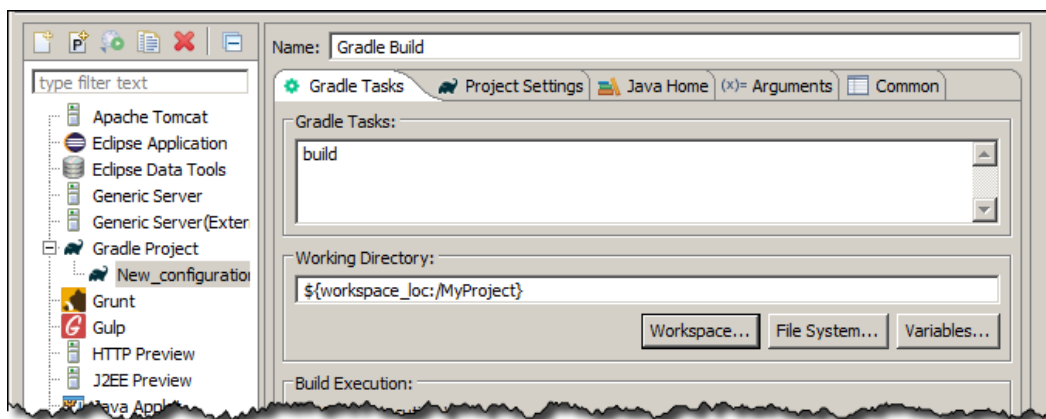
__ 3. Double click on **Gradle Project**.

__ 4. Set the **Name** to **Gradle Build**

__ 5. Type **build** in the **Gradle Tasks** text area.

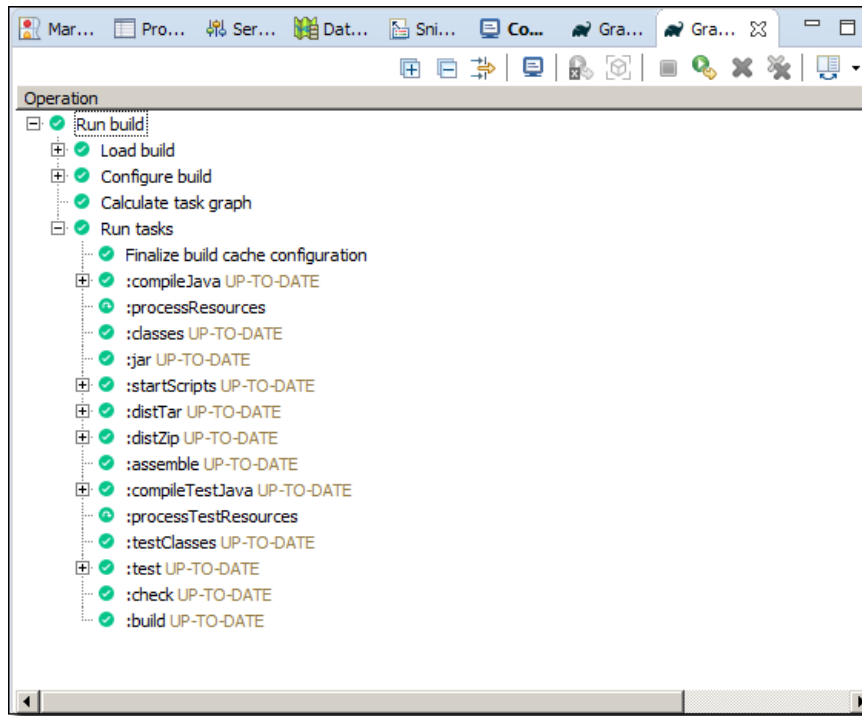
__ 6. Click the **Workspace** button and select **MyProject** to set the **Working Directory** and click **OK**.

Verify your screen looks like this:



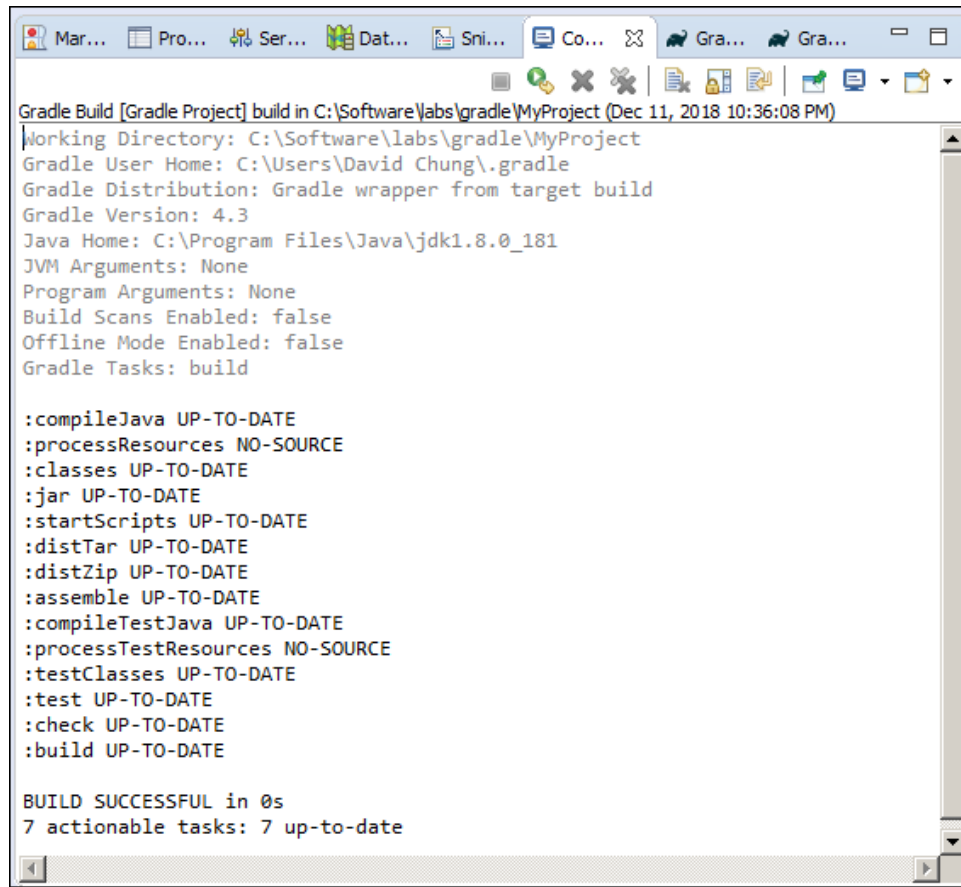
__ 7. Click **Run**.

You should see the build process in the Gradle Execution Window.



__8. Click the **Console** tab.

You should see the build output and status.



```
Gradle Build [Gradle Project] build in C:\Software\labs\gradle\MyProject (Dec 11, 2018 10:36:08 PM)
Working Directory: C:\Software\labs\gradle\MyProject
Gradle User Home: C:\Users\David Chung\gradle
Gradle Distribution: Gradle wrapper from target build
Gradle Version: 4.3
Java Home: C:\Program Files\Java\jdk1.8.0_181
JVM Arguments: None
Program Arguments: None
Build Scans Enabled: false
Offline Mode Enabled: false
Gradle Tasks: build

:compileJava UP-TO-DATE
:processResources NO-SOURCE
:classes UP-TO-DATE
:jar UP-TO-DATE
:startScripts UP-TO-DATE
:distTar UP-TO-DATE
:distZip UP-TO-DATE
:assemble UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources NO-SOURCE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build UP-TO-DATE

BUILD SUCCESSFUL in 0s
7 actionable tasks: 7 up-to-date
```

You may see some warnings.

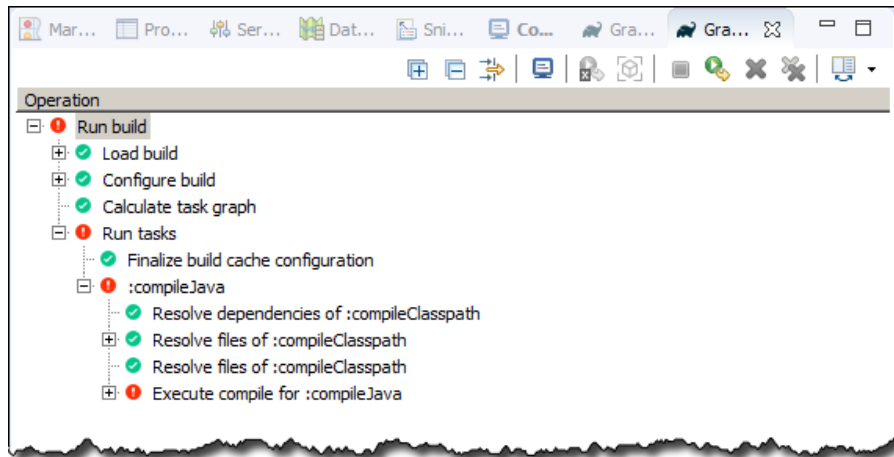
Next you will make a change that breaks *Greeter.java* (e.g. remove a semi-colon).

__ 9. Open **src/main/java/hello/Greeter.java**

__ 10. Remove a semi-colon in the return line.

__ 11. Save the file.

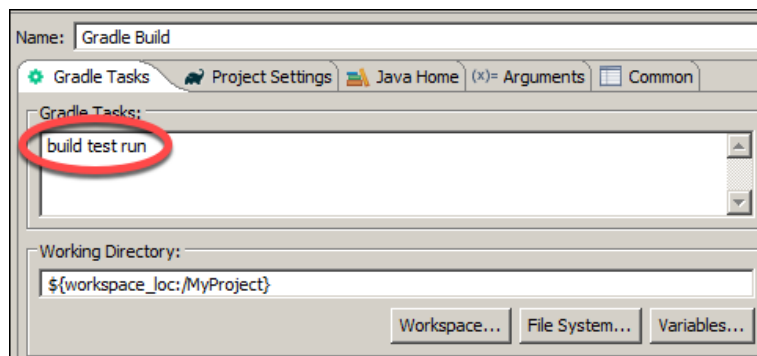
__12. Execute the run configuration again. You should see errors.



__13. Fix the code and build again, it should compile without errors.

__14. Go back to **Run Configurations**.

__15. Update the **Gradle Tasks** to use **build test run**.



__16. Click **Run**.

__17. Open a file browser.

__18. Open **C:\Software\labs\gradle\MyProject\build\test-results\test\TEST-hello.TestGreeting.xml** in a text editor

__19. You should see that one test ran with no failures.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="hello.TestGreeting" tests="1" skipped="0"
failures="0" errors="0" timestamp="2016-12-12T05:07:19"
hostname="WIN-3CLLIM6JDKO" time="0.003">
  <properties/>
  <testcase name="testGreeter" classname="hello.TestGreeting"
time="0.003"/>
  <system-out><![CDATA[]]></system-out>
  <system-err><![CDATA[]]></system-err>
</testsuite>
```

__20. Close the file.

__21. Click the **Console** tab and you should see the output from the program.

```
:che...-TO-DATE
:build UP-TO-DATE
:run
The crrnt time is: 23:02:52.451
Hello world!

BUILD SUCCESSFUL in 0s
8 actionable tasks: 1 executed, 7 up-to-date
```

Part 3 - Integrate with GitLab

In this part you will configure a project at GitLab.com. You will then create a local Git repository for the application. Next you will commit the application locally and push it to GitLab.com. Finally you will make a modification to the code an push an update to GitLab.com.

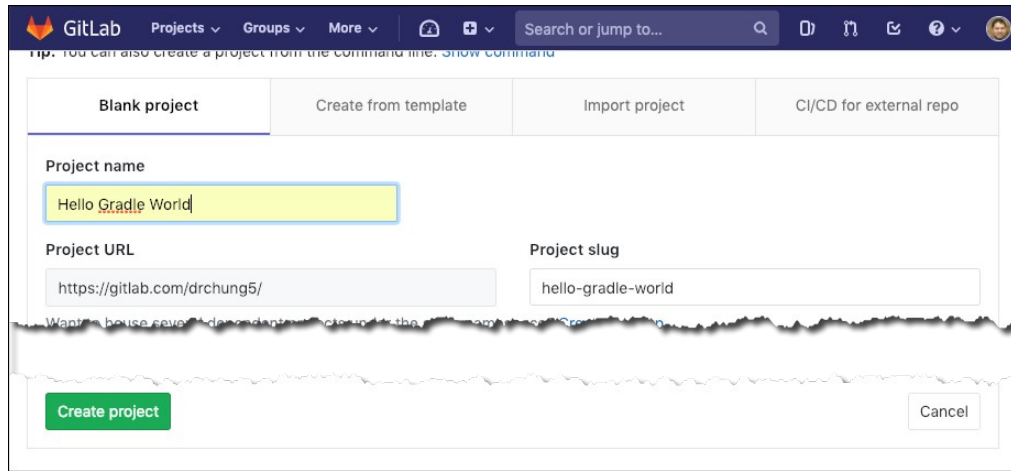
__1. You should have received an email address, username and password for GitLab.com. You will need them to configure GitLab and Git. Write them down here:

Email: _____

Username: _____

Password: _____

- __2. Open a web browser and navigate to **https://www.GitLab.com**
- __3. Sign in with the username/password above.
- __4. Click **Create a project**.
- __5. Make sure the **Blank project** tab is selected.
- __6. Set the **Project name** to **Hello Gradle World** and click **Create Project**.



- __7. Scroll down and then copy the project URL to the clipboard.

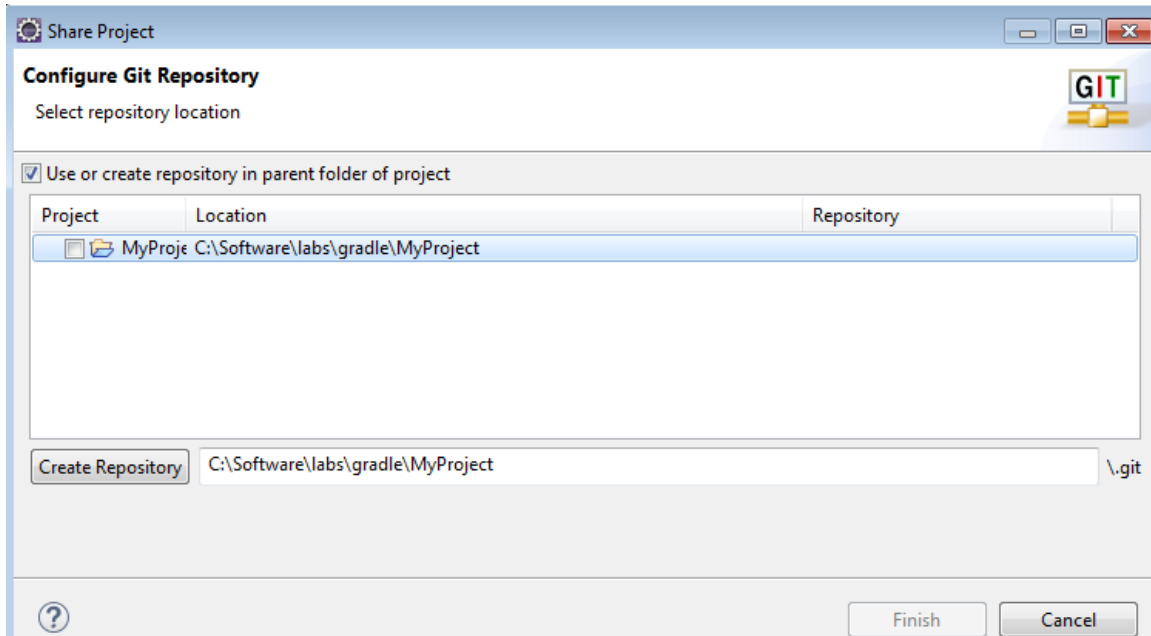
Create a new repository

```
git clone https://gitlab.com/alex.jimenez/hello-gradle-world.git
cd hello-gradle-world
touch README.md
```

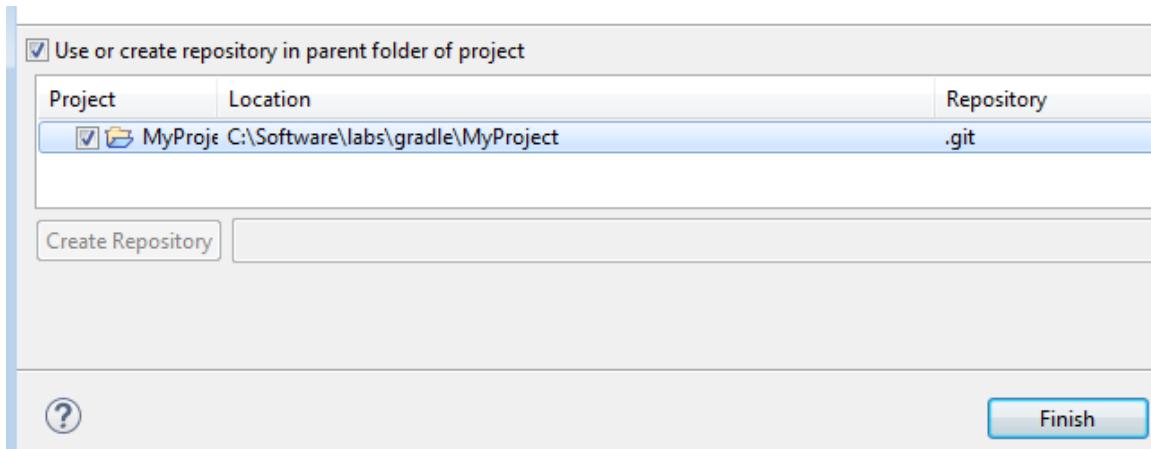
- __8. Open a notepad editor and paste the URL that you just copied so you don't miss it later when you will use it.
- __9. In eclipse, right click **MyProject** and select **New → File**.
- __10. Enter **.gitignore** and click **Finish**.
- __11. Add the following contents:

```
/.gradle
/.settings
/.classpath
/.project
/bin
/build
```

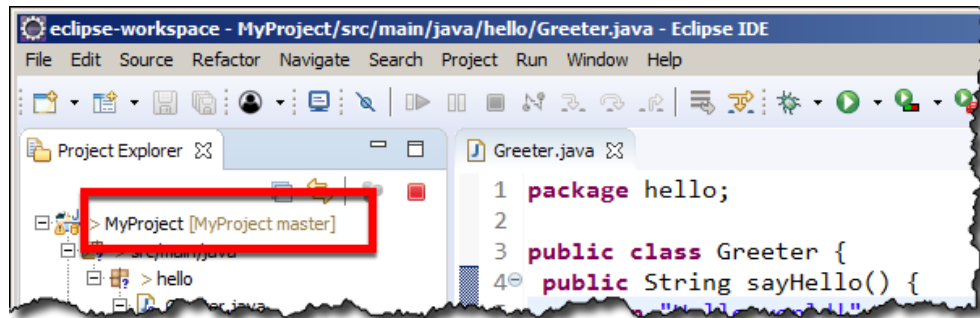

- __12. Save the file.
- __13. Right click on **MyProject** and select **Team** → **Share Project ...**
- __14. On the **Configure Git Repository** page, check **Use or create a repository in the parent folder of project**.
- __15. Click on **MyProject** and you will see that the text next to **Create repository** button is auto filled.



- __16. Click the **Create repository** button.
- __17. The screen will be updated, click **Finish**.



__18. Notice that the project is now decorated with **[MyProject master]**.



__19. Right click on **MyProject** and add all the project files to staging by selecting **Team** → **Add to Index**.

__20. Open a DOS Shell to the project directory and run **git status** to verify that file have been staged.

```
cd C:\Software\labs\gradle\MyProject
git status
```

```
C:\Software\labs\gradle\MyProject>git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

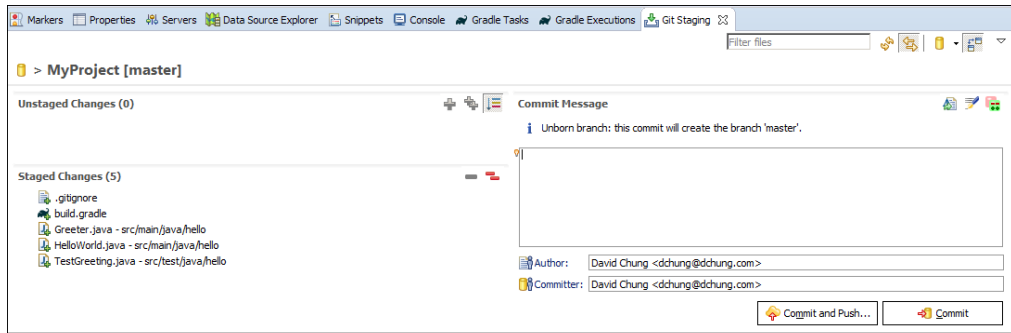
        new file:   .gitignore
        new file:   build.gradle
        new file:   src/main/java/hello/Greeter.java
        new file:   src/main/java/hello/HelloWorld.java
        new file:   src/test/java/hello/TestGreeting.java

C:\Software\labs\gradle\MyProject>_
```

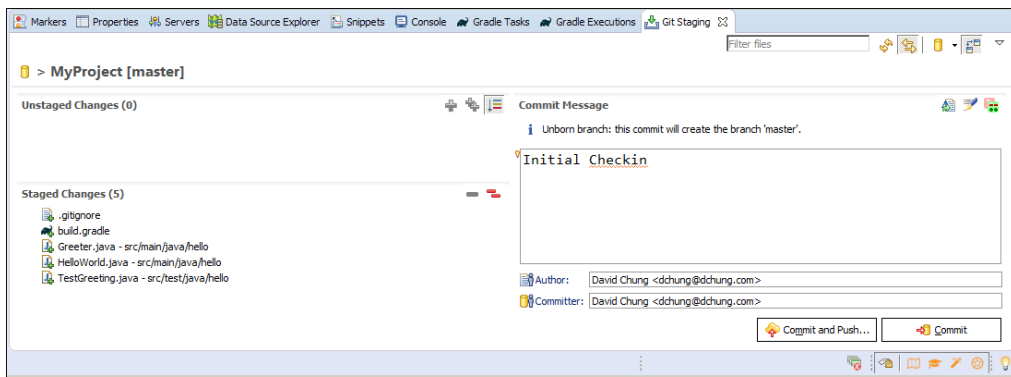
__21. Right click on **MyProject** and select **Team** → **Commit ...**

You should see the **Git Staging** view in Eclipse, maximize the tab.

(Hint if you can't find this tab, select **Window** → **Show View** → **Other ...** from the main menu then select **Git** → **Git Staging**)



22. Add a **Commit Message** and click **Commit and Push ...**



__23. Fill in **Destination Git Repository** dialog using the URL you copied before and the credentials you wrote down earlier in this lab.

Push Branch master

Destination Git Repository

Enter the location of the destination repository.

Remote name:

Location

URI:

Host:

Repository path:

Connection

Protocol:

Port:

Authentication

User:

Password:

☒ Store in Secure Store

__24. Click **Next**.

__25. In the Login dialog, enter your credentials and click OK.

__26. On the **Pull to branch in remote** dialog, accept the default values and click **Next**.

__27. On the **Push Confirmation** page click **Finish**.


__28. If prompt again, then enter your credentials and click OK.

__29. Review the **Push Results** page and click **Close**.

- __30. Go back to GitLab.com browser.
- __31. Refresh the project page and you should see the new project.

master




hello-gradle-world / +

 **Initial Checkin**
wasadmin authored 10 minutes ago

+ Add README

+ Add CHANGELOG

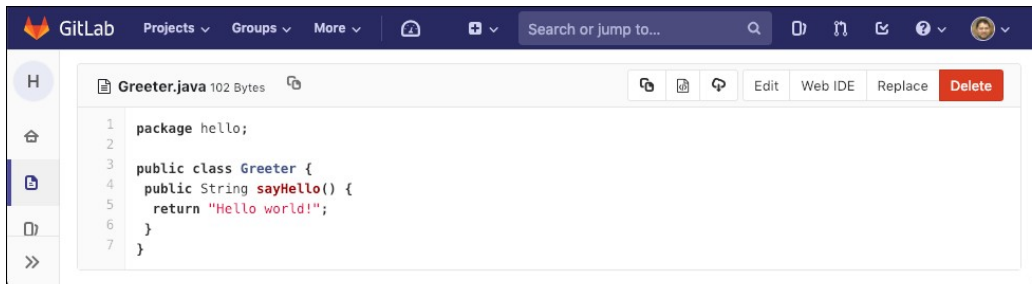
+ Add CONTRIBUTING

Name	Last commit
 src	Initial Checkin
 .gitignore	Initial Checkin
 build.gradle	Initial Checkin

__32. Click **src** and navigate to **/main/java/hello/**

__33. Click on **Greeter.java**

Verify the the content of the file.



```
1 package hello;
2
3 public class Greeter {
4     public String sayHello() {
5         return "Hello world!";
6     }
7 }
```

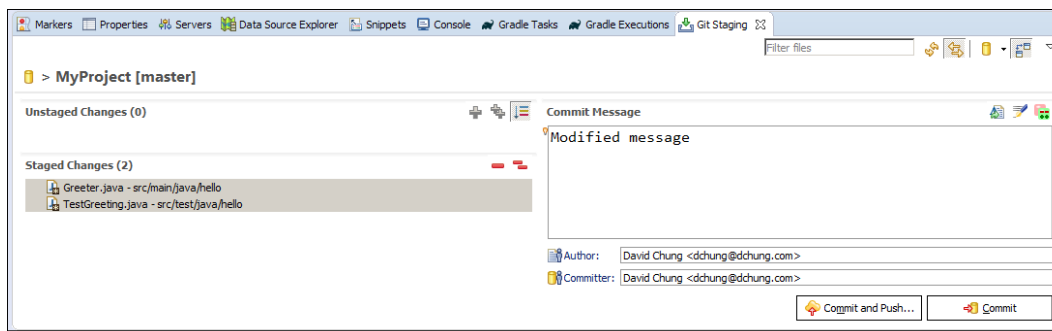
- __ 34. Go back to Eclipse.
- __ 35. Double click on the Git Staging tab to return to the normal size.
- __ 36. Open **src/main/java/hello/Greeter.java**
- __ 37. Modify the **sayHello()** function as follows:

```
"Hello Gradle/GitLab world!"
```

- __ 38. Save the file.
- __ 39. Open **src/test/java/hello/TestGreeting.java**
- __ 40. Modify the message as follows:

```
"Hello Gradle/GitLab world!"
```

- __ 41. Save the file.
- __ 42. Right click on **MyProject** and select **Team** → **Commit** to stage changes.
- __ 43. You should see changes staged in the **Git Staging** view.
- __ 44. Enter a **Commit Message** and click **Commit and Push ...**



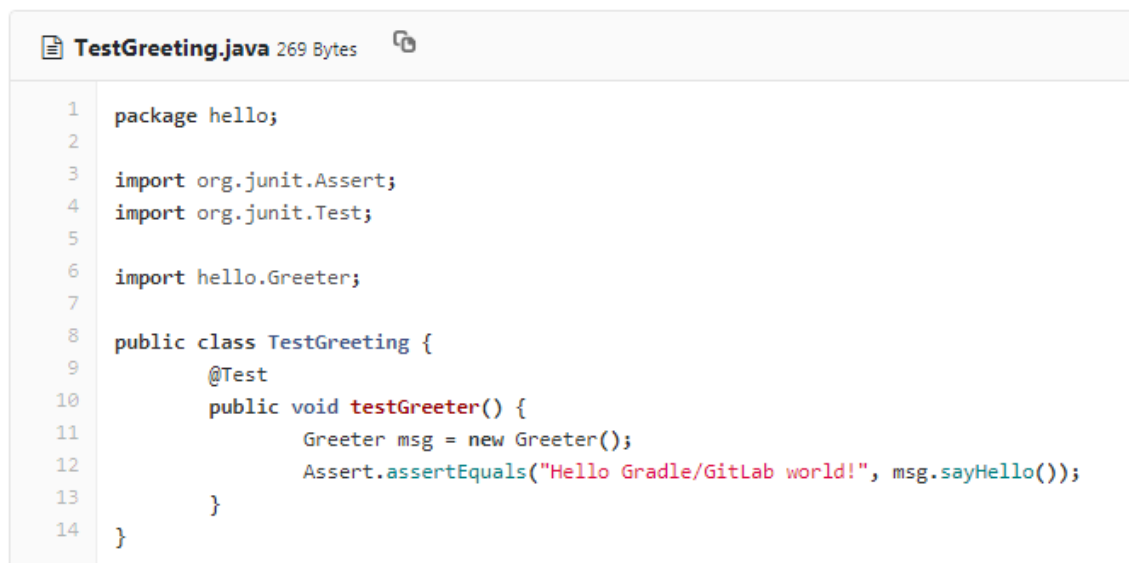
- __ 45. Enter again your credentials and click OK.
- __ 46. Close the confirmation dialog.
- __ 47. Go back to the browser in GitLab.com.

__48. Refresh the page and you should see the changes.



```
1 package hello;
2
3 public class Greeter {
4     public String sayHello() {
5         return "Hello Gradle/GitLab world!";
6     }
7 }
```

__49. Open **hello-gradle-world/src/test/java/hello/TestGreeting.java** and verify the changes.



```
1 package hello;
2
3 import org.junit.Assert;
4 import org.junit.Test;
5
6 import hello.Greeter;
7
8 public class TestGreeting {
9     @Test
10    public void testGreeter() {
11        Greeter msg = new Greeter();
12        Assert.assertEquals("Hello Gradle/GitLab world!", msg.sayHello());
13    }
14 }
```

Part 4 - Review

In this you took an existing application that uses Gradle and imported it into Eclipse. Next you used Eclipse Gradle tools (Buildship) to build, test and run the application. Finally you used Eclipse Git tools (Egit) to integrate with GitLab.com.

Lab 7 - Appendix - Rebasing and Rewriting History

When we have a set of distributed repositories and developers, we are likely to have parallel streams of work going on that would otherwise conflict with each other. As well, the distributed nature of git encourages developers to commit often to their local repositories, even if their work is not yet ready to share. Indeed, the whole concept of "what to share" needs special consideration when we have both a local and a remote repository. As a result, git gives us some interesting capabilities to rewrite history, and thus determine what exactly we share. Also, we need a way to apply a set of changes as though they were applied at a different starting point.

At the end of this lab you will be able to:

1. Use 'git rebase' to move work from one branch to another
2. "Squash" a commit using the interactive rebase tool.

Part 1 - Pull From the Starting Point Repository

__1. Open a command prompt window if you don't have one already available, and navigate to [C:\workspace](#) by typing:

```
cd \workspace
```

__2. In the **Command Prompt** window, type the following lines, pressing return after each line

```
git clone \LabFiles\SquashRepo.git
cd SquashRepo
```

These commands checkout a clone of the "starting point" repository for this lab, and then change directory into that clone.

__3. Type 'git log --format=oneline', and then press return.

```
C:\workspace\SquashRepo>git log --format=oneline
6d5f5be16ad30900c2f19d0f1886b0b6e3a41dd3e Changed count file.
96f46c3440b645ffa22b1337dc229198451fb9a9 Changed count file.
548ed278a8c0e5e5b07962dc10c21beb2b7b5747 Changed count file.
8a40fc1134070c907586818514a8f2bed5c71c9d Changed count file.
03f739d6b499d0834018f9bfcd2b6c998ecec29b Added count file.
bdb807b0c99991c646aa8a0d5dfd2197aa9d8146 We now have the text we want.
152674c04f49834ca9d7a0574ca6d3f54399190f The web site has a basic index.html fil
```

__4. Press q to return to the prompt.

The '--format=oneline' option causes 'git log' to display each commit on a single line. You can see that this repository contains a number of very similar commits. Also we should note that these commits are on the main branch. Let's imagine that we've been working for a while on this repository, making these commits, and now we realize that we really should have been doing this on a feature branch.

Part 2 - Rebase Commits Onto a Different Branch

In the above commit list, notice that each commit is identified using a long string of hex characters. This is actually an SHA-1 hash of the index for each commit, but really all

we need to care about is that each commit essentially has a unique identifier.

In order to put these items onto a different branch, we essentially need to do the following:

- Go back to an earlier commit state (we'll go back to the commit that had the message 'We now have the text we want'; the id for this commit starts with 'bdb807b0')
- Create a branch that starts with that commit
- replay the changes onto this new branch
- Reset the master branch to the earlier commit state.

__ 1. Type '**git checkout bdb807b0**'.

```
C:\workspace\SquashRepo>git checkout bdb807b0
Note: checking out 'bdb807b0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b new_branch_name

HEAD is now at bdb807b... We now have the text we want.
```

Notice that git warns us that we're in a "detached head" state, where we can discard commits that we make in this state. In our case it doesn't matter, because we're about to make a branch.

__ 2. In the Git Bash window, enter '**git checkout -b new-count**'.

__ 3. Now the big one – we're going to tell git to take all the changes that have been done since the 'new-count' branch was created, and put them onto the new-count branch. Type '**git rebase master**'

__ 4. Type '**git checkout master**'.

Now we're on the master branch. We'll reset the HEAD to point to the commit where we branched off.

__ 5. Type '**git reset bdb807b0**' and press return.

__ 6. Now type '**git log --format=oneline**', press return and then press q.

```
C:\workspace\SquashRepo>git log --format=oneline
bdb807b0c99991c646aa8a0d5dfd2197aa9d8146 We now have the text we want.
152674c04f49834ca9d7a0574ca6d3f54399190f The web site has a basic index.html fil
```

Notice that the history ends at the commit that we mentioned. Essentially, we have changed history to read that we branched off onto a branch called 'new-count' at that point, and then did our work on that branch.

Warning:

In this case, we have rewritten history that already exists in a different repository ('/C/LabFiles/SquashRepo'). This is actually very bad practice, because if we pushed these changes to a remote repository, there might be essentially two views of history out there, which can cause a lot of confusion.

We only did this to simulate a case where you have been working on a local repository for a while, then said "Oops!", without actually requiring you, the student, to make and commit several changes.

In reality, you should follow the rule "Never rewrite history on any commits that you've published, anywhere!"

Part 3 - Squash A Series of Commits

__1. In the **Command Prompt** window, type '**del count.txt**' (this file should not actually exist on this branch).

__2. Type '**git checkout new-count**'.

__3. Type '**git log --format=oneline**', press return and then press 'q'.

```
C:\workspace\SquashRepo>git log --format=oneline
6d5f16ad30900c2f19d0f1886b0b6e3a41dd3e Changed count file.
96f46c3440b645ffa22b1337dc229198451fb9a9 Changed count file.
548ed278a8c0e5e5b07962dc10c21beb2b7b5747 Changed count file.
8a40fc1134070c907586818514a8f2bed5c71c9d Changed count file.
03f739d6b499d0834018f9bfc2b6c998ecec29b Added count file.
bdb807b0c99991c646aa8a0d5dfd2197aa9d8146 We now have the text we want.
152674c04f49834ca9d7a0574ca6d3f54399190f The web site has a basic index.html fil
```

Notice that we have a few commits with the same message: 'Changed count file'. Obviously we changed the count file several times before we settled on the final version. If we're publishing our work to an external repository, there's no need to show all the changes; we'd rather just publish one final change. So we'll 'squash' the commits.

__4. We want to start from the commit that has id starting with bdb807b. Type '**git rebase -i bdb807b**' and press return.

__5. git will open an editor with a script that will perform the rebase operation. Edit the file to read as follows:

```
pick 03f739d Added count file.
squash 8a40fc1 Changed count file.
squash 548ed27 Changed count file.
squash 96f46c3 Changed count file.
squash 6d5f16a Changed count file.

# Rebase bdb807b..6d5f16a onto bdb807b
#
```

Recall that earlier on, we set the editor configuration variable 'core.editor' to use WordPad, so the interactive rebase file should open in WordPad.

__6. Save the file and exit the editor.

__7. Git does the changes, and opens an editor to let you edit the final commit message. By default, it simply concatenates the commit messages that we squashed. We'll leave that for this example, so just close WordPad

```
C:\workspace\SquashRepo>git rebase -i bdb807b
[detached HEAD b9e65c8] Added count file.
1 file changed, 1 insertion(+)
create mode 100644 count.txt
Successfully rebased and updated refs/heads/new-count.
```

Finally, git reports success.

__8. Just to see the results, type '**git log --oneline**'.

```
C:\workspace\SquashRepo>git log --format=oneline
b9e65c85f6f15590164d7bd7858351693f006656 Added count file.
bdb807b0c99991c646aa8a0d5dfd2197aa9d8146 We now have the text we want.
152674c04f49834ca9d7a0574ca6d3f54399190f The web site has a basic index.html fil
<END>
```

We now have a far simpler commit history, that we don't mind publishing to the outside world.

In a way, squashing a commit is similar to the act of preparing a diff, or a patch, that we might have done with older version control systems. We are essentially editing the final version of work that we want to publish to downstream users of the work.

__9. Close all.

Part 4 - Review

In this lab, we looked at two ways to rewrite history in git. There are many more, but the items we looked at are fairly common situations in standard git workflows. In particular, it is common for centralized repositories (e.g. open-source projects) to ask for "squashed commits", so as to see simplified work products and history.

