ENGG1340 Computer Programming II
COMP2113 Programming Technologies
Module 2: Version Control
Estimated time to complete: 2 hours

Objectives

At the end of this chapter, you should be comfortable with:

- Setting up your own Git repository
- Saving changes to your repository
- Inspecting your repository
- Traversing your repository
- Concepts of Branching, Merging, Pushing and Pulling

1. Introduction

1.1 What is Git?

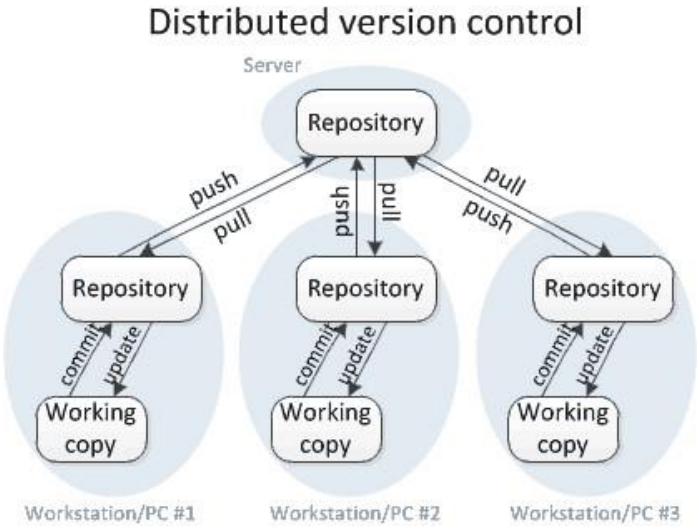
Git is a common and modern version control system for managing and tracing changes in computer files and coordinating work on those files among multiple people. It is primarily used for source-code management in software development. Git is a distributed version control system (DVCS) that has greater characteristics of performance, security and flexibility than most alternate version control systems.

1.2 What is a Version Control System?

- Version control systems, or VCSs, are a category of software instruments that support software development teams, manages changes to source code over time.
- It tracks the history of individual changes by each contributor to code in a special kind of database.
- If a mistake is made or a bug is to be fixed, developers can turn back to an earlier version of the source code to solve the problems without impeding the workflow of other team members.
- If a software team does not use a VCS they are subject to issues such as the creation of incompatible code between two independent parts of a project or ignorance towards the changes that are available to the users.

1.3 Why use Git?

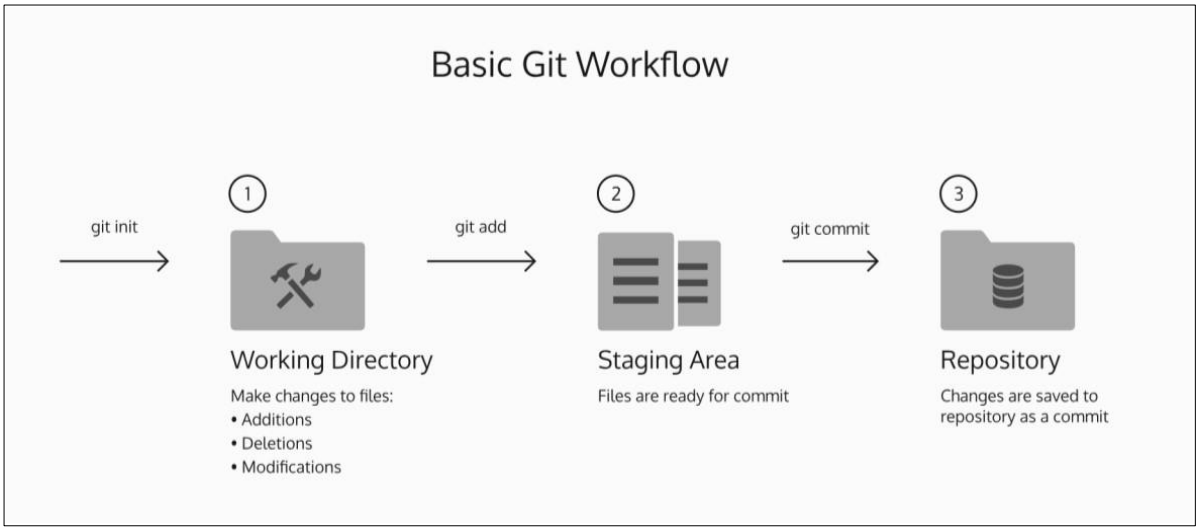
- Git lets developers see the entire timeline of their changes, decisions, and progression of any project in one place.
- With a DVCS like Git, collaboration can happen at any time while maintaining source code integrity. Using branches, developers can safely propose changes to production code.



- Businesses using Git can break down communication barriers between teams and keep them focused on doing their best work

1.4 Basic Git Workflow

The basic Git workflow is shown below. A more detailed discussion will be given in the next section.



- The **Working Directory** is where all changes will be made to the file.
- The **Staging Area** where you will mention all the changes made to the working directory.
- A **Repository** in which Git stores all changes made as different versions of the project.

2. Getting Started with Git

2.1 Installing Git

Before we start using git, we have to make sure it is there on your computer. If it is installed it is good to update it to the latest version.

To check if git is installed on your computer use `git version` command.

```
$ git version
git version 2.17.1
```

If it is not installed, follow the instructions below to install it.

Installing on Linux

To install git on Linux systems copy and paste the commands below to your terminal.

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install git
```

Installing on Mac

To make installation of software easier on Mac we download Homebrew. If you already have Homebrew you can skip the step below. If you do not have Homebrew copy the commands written below and confirm the installation.

```
$ ruby -e "$(curl -fsSL
   https://raw.githubusercontent.com/Homebrew/install/master/install)"
$ brew doctor
```

Copy and paste the code below to download git.

```
$ brew install git
```

Installing on Windows

To install git on Windows, visit the link below and download it.

https://gitforwindows.org/

2.2 Initialising a new repository

A repository, or git repository, contains the entire collection of files, folders, directories, etc., along with a history of the changes made in the project.

There are two ways to create your local git repository. You can either initialise it or clone an existing remote directory.

To Initialise a directory

1. Open your terminal and browse to the directory of the root project folder using the `cd` command.
2. In this folder, we will use the command `git init` to initialize a new repository.

```
$ cd project
```

```
$ git init
Initialized empty Git repository in /home/research/ra/1801/cklai/project/.git/
```

   After executing this command, a new *.git/* subdirectory will be created in your current working directory. The command sets up all the tools Git needs to begin tracking changes made to the project.

3. Now we can create a file that we want to work on. For example, we can create *work.txt* as follow.

```
Welcome to my Git tutorial.
Today we will learn how to get started with Git.
```

`git init` has a few other command line options which you may find useful:

Command	Meaning
`git init`	Creates a new local repository

`git init --quiet`
or
`git init -q`
	Only prints "critical level" messages, errors and warnings. All other output is silenced

`git init --bare`	Creates a bare repository
`git init -template=<template_directory>`	Specifies directory from which templates will be used.
`git init --separate-gitdir=<git dir>`	Creates a text file containing the path to the actual repository

To Clone an existing remote directory

1. If you want to use someone's project as an inspiration for your own project, you can copy their directory.
   NOTE: If a *.git* directory is present, the repositories will be cloned there. If we do not have a *.git* directory then the repository will be cloned to your pwd (present working directory).
2. We type the command `git clone <remote repository URL >` to clone a copy of the remote repository. In the example below, we copy the remote repository from the Internet to the current directory.

```
$ git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORY
Cloning into `Spoon-Knife`...
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (8/8), done.
remove: Total 10 (delta 1), reused 10 (delta 1)
Unpacking objects: 100% (10/10), done.
```

Some additional commands of `git clone` are:

Command	Meaning
`git clone -branch <branch_name>`	Specifies a specific branch to clone instead of the entire master branch
`git clone --bare`	Similar to `git init --bare` creates a copy of the remote repository with an omitted working directory
`git clone --mirror`	Clones all extended references of the remote repository and implicitly calls the `-bare` argument
`git clone --template=<template_directory> <repo location>`	Clones the repo `at <repo location>` and applies the template from `<template_directory>` to the newly created local branch

2.3 Inspecting the working directory

As you keep making changes in your working directory (only *work.txt* at the moment), you can track all the changes made by using the command `git status`.

```
$ git status
On branch master

No commits yet

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

        work.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Notice that your file *work.txt* is in red and under the untracked files. This means that Git can see the file but has not started tracking changes on that file yet.

2.4 Staging changes

In order for Git to start tracking the changes you make in the working directory, you need to add those files to the staging area first. This can be done by using the command `git add filename` , where *filename* is the name of the file you're working on, like the *work.txt* file for us.

This command adds all changes in the working directory to the staging area. It is used to save a copy of the current state of your project.

After adding the file to the staging area, you can check the status of the files again using `git status`

```
$ git add work.txt
```

```
$ git status
On branch master

No commits yet

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

        new file:   work.txt
```

Now you can see how Git shows that a new file was added to the staging area, in green.

Some useful `git add` commands are shown below:

Command	Meaning
`git add .`	Adds all files under the current directory

`git add -A`
or
`git add --all`
	Finds all new or updated files that are present throughout the project and adds them to the staging area

2.5 Tracking changes in the working directory

If we make changes the *work.txt* by adding a third line.

```
Welcome to my Git tutorial.
Today we will learn how to get started with Git.
We start with the Basic Git Workflow.
```

To check the difference between the working directory and the staging area, we can use the command `git diff filename`, where *filename* is the name of the file that you are checking.

```
$ git diff work.txt
diff --git a/work.txt b/work.txt
index 90ac5e6..f0f679f 100644
--- a/work.txt
+++ b/work.txt
@@ -1,2 +1,3 @@
 Welcome to my Git tutorial.
 Today we will learn how to get started with Git.
+We start with the Basic Git Workflow.
```

The text indicated that the first two lines of text in the working directory are also in the staging area, and the last line of text is added in the working directory.

We can then add the changes made to the staging area using the command `git add work.txt`.

Below are some useful commands using `git diff`.

Command	Meaning
`git diff --base <filename>`	View the conflicts against the base file (the point where the two branches started diverting the considered file)
`git diff <sourcebranch> <targetbranch>`	Preview changes before merging

* Both commands are useful before merging. Merging is explained in the next section.

If the output of `git diff` is too long, git will use a pager (using command `more` or `less`), in that case, you need to press `Q` to exit the pager. Alternatively, you can add option `--no-pager` to ask git not to use a pager. This option can be used in most of the git commands.

2.6 Committing changes

After staging all changes, the last step of the Git workflow would be to permanently save the changes from the staging area to the repository. Every time we do this, we create a commit that you can refers to in the future. For this, we use the command `git commit` .

The option `-m` will be used to specify a commit message. The message should be enclosed in a pair of double-quotes("). The message should describe the purpose or the changes in the commit.

```
$ git commit -m "Added an introduction."
[master (root-commit) 8327731] Added an introduction.
    Committer: Chan Tai Man <tmchan@academy11.cs.hku.hk>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 3 insertions(+)
 create mode 100644 work.txt
```

Notice the message above regarding your name and email address configuration. You can follow the instructions to make changes to the configuration file accordingly.

As a good development practice and for a good repository stewardship, always specify a meaningful commit message.

To access older version of the project, you can use the command `git log`. It lets you see the list of all previous commits, filter it, and also search for specific changes. All commits are stored in chronological order in the repository and can be accessed using this command.

```
$ git log
commit 8327731fa6a9108fb6b54d0b38b9b59c7fbf316c (HEAD -> master)
Author: Chan Tai Man <tmchan@academy11.cs.hku.hk>
Date:   Mon Jan 14 10:38:43 2019 +0800

    Added an introduction.
```

In each record, the following information will be given.

- A 40-character code called SHA (hash), that is used to uniquely identify the commit, typically seen in orange.
- The commit author, being yourself.
- The date and time of the commit
- The commit message

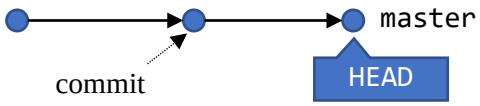
3. Collaborating with others

We have learnt about the basics of setting up a repository, inspecting it and saving the changes in a repository. We will now go over the basics of the mechanism used to collaborate with other users such as **Branching**, **Merging**, **Push**, **Pull**.

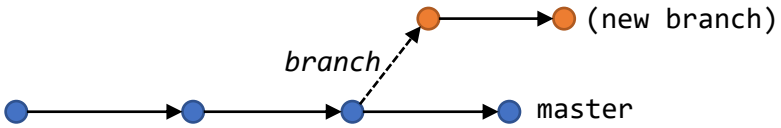
3.1 Branching

What is Branching?

A branch can be thought of as a pointer to the latest commit in your Git repository. When we initialise a repo, we are working on a single branch called the master branch. The commit we are working at is called HEAD, which is usually the latest commit of a branch. We can visualize the branch as a list.

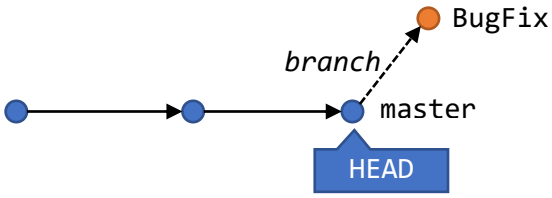


Branches can be created to allow changes to be made on different parts of a project simultaneously. This increases efficiency and allows for abstraction of work.

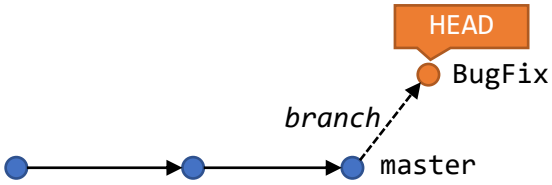


Creating a Branch

We will be using our local repository to create the branch. Type the command `git branch BugFix` to create a new branch called `BugFix`.



Note that we are still working in the master branch. We need to switch to the new branch using the command `git checkout BugFix`.




```
$ git branch BugFix
```

```
$ git checkout BugFix
Switched to branch 'BugFix'
```

Traversing your repository

Above we have created a new branch named `BugFix` and used the command `checkout` to switch to that branch.

We can use the `checkout` command to move our HEAD to a previous commit as well. If we do so, we restore the states of the files in this commit as well. For example, we can move to commit `BugFix^`, which is the previous commit in the branch `BugFix`.

```
$ git checkout BugFix^
Note: checking out 'BugFix^'.

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 8327731 Added an introduction.
```

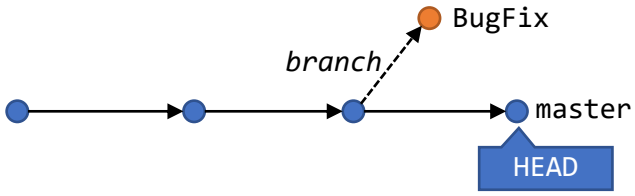
To move to a specific commit, use `git log` to find the hash.

Below are some useful commands involving branches:

Command	Meaning
`git branch`	List all branches
`git branch branchname`	Create a branch
`git checkout branchname`	Change to a branch
`git checkout -b branchname`	Create and change to a new branch
`git branch -m branchname new_branchname`	Rename branch
`git branch --merged`	Show all completely merged branches with current branch
`git branch -d branchname`	Delete merged branch (only possible if not HEAD)
`git branch -D branch_to_delete`	Delete not merged branch

3.2 Merging

Before we continue, let's move back to the master branch and add a new file.



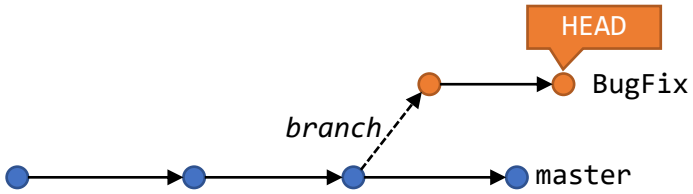

```
$ git checkout master
Switched to branch 'master'
```

```
$ echo "This is some new file in master." > master.txt
```

```
$ git add master.txt
```

```
$ git commit -m "Added master.txt"
[master 832ceba] Added master.txt
 1 file changed, 1 insertions(+)
 create mode 100644 master.txt
```

Next, we add another file in branches `BugFix`.




```
$ git checkout BugFix
Switched to branch 'BugFix'
```

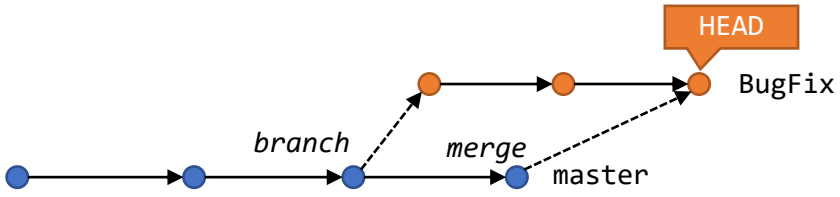
```
$ echo "This is some new file in BugFix" > bugfix.txt
```

```
$ git add bugfix.txt
```

```
$ git commit -m "Added bugfix.txt"
[BugFix ed600da] Added bugfix.txt
 1 file changed, 1 insertions(+)
 create mode 100644 bugfix.txt
```

Here we mimic the case when someone is fixing a bug in branch `BugFix` and at the same time, some other continue the development in the master branch. After the bug is fixed, we would like the change in branch `BugFix` to be reflected in the master branch which is our main project. To do this we can use the `merge` command.

There are two directions of merging, we could merge the changes in master to `BugFix` or the other way around. Depending on the situations, it may be safer to merge the changes in master to `BugFix` so that you can check if the merge is successful without touching the master branch.



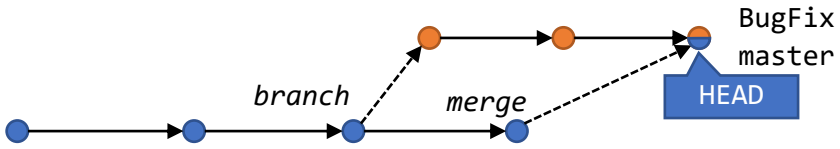
Since we are merging the changes in master to `BugFix` , we stay in the branch `BugFix` and apply the merge. As a new commit is created, you should specify a commit message using option `-m`, otherwise, Git will ask you to input a message.

```
$ git checkout BugFix
Already on 'BugFix'
```

```
$ git merge master -m "Apply changes in master"
Merge made by the 'recursive' strategy.
 master.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 master.txt
```

If the master branch has made a change that affects the change in branch `BugFix`, there will be a **conflict**. Git will ask you to resolve the conflicts before it can continue merging.

Now if the merge is successful, we may apply the bug fix on master. Suppose we checkout the master branch and do a merge again. Since there is only one path from master to `BugFix`, by default Git will simply move the master branch to the last commit in `BugFix` so they both share the same commit. This is called **fast forward**. Note that there is no commit created in this case.




```
$ git checkout master
Switched to branch 'master'
```

```
$ git merge BugFix
Fast-forward
 bugfix.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 bugfix.txt
```

Below are some useful commands involving merge:

Command	Meaning
`git merge branchname`	Merge a branch (will fast forward if possible)
`git merge --ff-only branchname`	Merge a branch only with fast forward
`git merge --no-ff branchname`	Merge a branch without fast forward (force a new commit)
`git merge --abort`	Stop merge in case of conflicts
`git cherry-pick 073791e7`	Merge only one specific commit

There is an alternative to merging. This command is called rebasing. To learn more about it, go to https://git-scm.com/docs/git-rebase.

3.3 Working with Remote Repositories

Till now we have only been working on our local repository. To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere.

Showing Remote Repositories

Use the command `git remote` to see all the remote repositories according to the shortnames you have given them. `git remote -v` allows you to look at the URL of the remote repository as well.

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Enumerating objects: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0), pack-reused 1857
Receiving objects: 100% (1857/1857), 334.04 KiB | 395.00 KiB/s, done.
Resolving deltas: 100% (837/837), done.
```

```
$ cd ticgit
```

```
$ git remote
origin
```

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

Adding Remote Repositories

To add a remote repository, use the command `git remote add <shortname> <URL>`

```
$ git remote add pb https://github.com/paulboone/ticgit
```

```
$ git remote
origin
pb
```

Now, you can see a remote repository “pb” is added to the local repository.

Inspecting a Remote Repository

If you want to see more information about a particular remote, you can use the `git remote show <remote>` command.

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master tracked
    ticgit tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Here are some of the important commands regarding remote repositories:

Command	Meaning
`git remote`	Viewing git remote configurations
`git remote -v`	Viewing git remote configurations along with associated URLs
`git remote add <name> <url>`	Change to branch
`git remote rm <name>`	Remove remote repository
`git remote rename <old-name> <new-name>`	Rename remote repository
`git branch --merged`	Show all completely merged branches with current branch
`git branch -d branchname`	Delete merged branch (only possible if not HEAD)
`git branch -D branch_to_delete`	Delete not merged branch

3.4 Pushing

The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. `git push` is most commonly used to publish an upload local changes to a central repository. After a local repository has been modified a push is executed to share the modifications with remote team members.

```
$ git push origin master
```

This command works only if you cloned from a server to which you have to write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to fetch their work first and incorporate it into yours before you'll be allowed to push.

Here are some of the important commands regarding remote repositories:

Command	Meaning
`git push <remote> <branch>`	Push the specified branch to <remote>
`git push <remote> --force`	Same as the above command, but force the push even if it results in a non-fast-forward merge
`git push <remote> --all`	Push all of your local branches to the specified remote.
`git push <remote> --tags`	Sends all of your local tags to the remote repository

3.5 Pulling

The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration workflows.

```
$ git pull
```

Below are some commonly used commands with pull:

Command	Meaning
`git pull <remote>`	Fetch the specified remote's copy of the current branch and immediately merge it into the local copy.
`git pull --nocommit <remote>`	Fetches the remote content but does not create a new merge commit.
`git pull --verbose`	Gives verbose content while pulling from the remote directory

4. Further Reading

We have introduced the Git here. You will get familiar with them when you spend more time using it. The following webpages contain a very good introduction to working in Git. You are highly recommended to read it once.

Git - gitglossary Documentation - https://git-scm.com/docs/gitglossary
This website provides a comprehensive explanation of Git terminology.

5. References

- Book. (n.d.). Retrieved from https://git-scm.com/book/en/v2