Git Workshop

Part 1 of 2

Setup, Clones, Commits and Branches

Workshop Agenda

- Part 1 – This Deck
  - Installation and Setup
  - Concepts
  - Repository Initialization
  - Clone
  - Basic Lifecycle
  - Logs
  - Introduction to Branches
- Part 2 – Another Deck
  - Merge Conflicts
  - Remote Repositories
  - Tags
  - Hosting Services
    - GitHub
    - AWS CodeCommit

This is a "presentation-ification" of the single-page workshop available at

https://github.com/lacounty-isab/workshops/tree/master/git

Installation

- Documentation and install images for most platforms:
  https://git-scm.com/.
  - Bookmark this for your documentation.
  - Download free e-book.
  - Watch videos.
  - Site includes references to GUI clients. This workshop will only work with the command line client.

- Windows Installation Options
  - Line endings
  - SSH Client
  - Bash Shell
  - Make sure `git` is in your PATH variable.

Check your installation:

```
$ git --version
git version 2.20.1
```

First Time Setup

- Git configuration scope options:
  - `--local`: (default) applies to a particular repository
  - `--global`: applies to all repositories for current user
  - `--system`: applies across all users
- For first configuration, set the following at global scope.
  - `user.name`
  - `user.email`

Exercise 1 – Setup

1. Create a new directory on your file system in which to perform the activities for the workshop. This directory will be called `GitWorkshop` for the rest of this workshop.
2. Open a command line terminal to `GitWorkshop`.
3. Verify your Git version with `git --version`. If this fails, you need to fix problems with your PATH or the installation.
4. Set your name and email in the **global** scope using the following commands.

```
git --config global user.name "John Doe"
git --config global user.email "jdoe@somewhere.gov
```

5. Clone the ISAB repository for this workshop to your workstation.

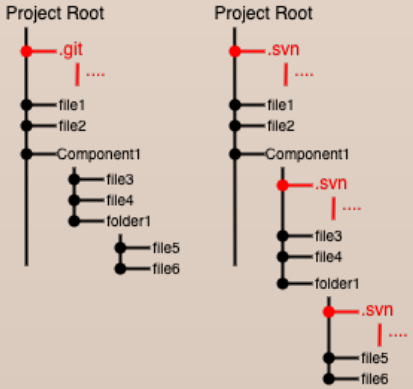
```
git clone https://github.com/lacounty-isab/workshops isabrepo
```

This will clone the Git repository hosted on GitHub to your local file system into a new directory named `isabrepo`. If you omit the `isabrepo` parameter, the new directory will default to the base name of the repository, in this case `workshops`.

Repository Initialization

- A Git repository can be created
  - through clone (previous slide)
  - running `init`
- Running init
  - creates an empty local repository
  - the entire repository is added to a folder named `.git`.
  - compare with SVN.

Git creates a single folder at the root of the working copy. SVN, CVS and others add folders recursively throughout the entire tree. This can be a mess to clean.



Exercise 2 – init

1. From the clone of the workshops repository, copy the `isabrepo/git/samples` directory to `GitWorkshop`. After this, you should have a copy named `GitWorkshop/samples`.
2. Change to `GitWorkshops/samples` directory in your command line. This is a directory of files from which we aim to create a new Git repository.
3. Run git status. It returns a message that simply means we are not in the context of a Git repository.

```
GitWorkshop/samples$ git status
fatal: not a git repository (or any of the parent
directories): .git
```

4. Run `git init`. This will create an empty local repository.

```
GitWorkshop/samples$ git init
Initialized empty Git repository in
GitWorkshop/samples/.git/
```

None of the files in this directory have been placed in the new repository. That will come later. On Windows, the new .git folder is harder to verify. On Linux and macOS it's apparent with `ls -a`.

Repository Initialization

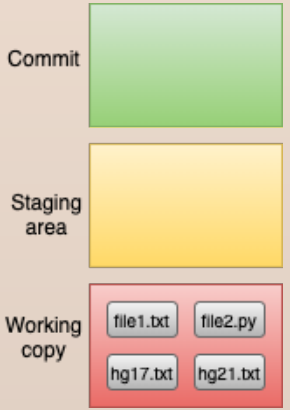
Exercise 2 – init (continued)

5. Run git status.

```
GitWorkshop/samples$ git status
On branch master

No commits yet

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

    file1.txt
    file2.py
    hg17.txt
    hg21.txt

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

The remark "nothing added to commit" is alluding to the staging area. In Git there are three states in which a version of a file can occupy: the working copy, the staging area, and a commit. These are shown in the figure to the right.

The working copy of a file is the one in your directory that you can see and edit. The staging area is a version of a file that is to be committed in the next commit action. Committed versions of the file are in the commit state. These are preserved in the commit history.



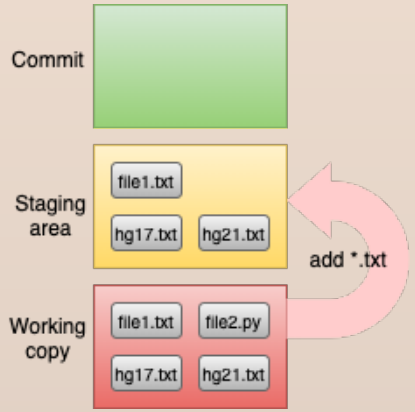
The three logical boxes above are created with the git init command. At the beginning, all files only exist as part of the working copy.

Repository Initialization

6. Commit all the `*.txt` files, but delay the `*.py` file.

```
GitWorkshop/samples$ git add *.txt
GitWorkshop/samples$ git status
On branch master

No commits yet

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

    new file:   file1.txt
    new file:   hg17.txt
    new file:   hg21.txt

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

       file2.py
```

Note the change in the status message. The status of the files went from "untracked" to "to be committed." These are the files that will be added to the repository



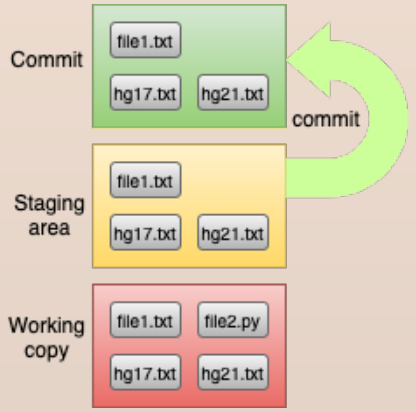
The **add** command adds files and changes to the staging area.

Repository Initialization

7. Run the commit command.

```
GitWorkshop/samples$ git commit -m "Initial version."
[master (root-commit) d5ec68e] Initial version.
 3 files changed, 102 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 hg17.txt
 create mode 100644 hg21.txt
```

This command will commit the changes with the message "`Initial version.`". This avoids a **vi** session for those of you not familiar with the vi editor. But it limits you to commit messages that are a single line.



The **commit** command adds all changes that have been accumulating in the staging area to a commit. There are now files "in the repository."

At the end of this exercise we still have `file2.py` as an untracked file. There is no harm in having files in the directory that are not part of the repository. We can commit them later or never commit them.

```
git1/samples$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    file2.py

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

Clone

The **clone** command copies a repository from one location to another location. We used it in the first exercise to clone the **remote** workshop repository to our **local** workstation. We can also clone locally within a single file system.

Exercise 3 – Local Clone

1. In your command line, change to the GitWorkshop directory. There should be two subdirectories: `isabrepo` and `samples`. Each of these directories holds a Git repository.
2. Clone the samples directory locally.

```
git clone samples samples2
```

3. Change to the `samples2` folder and list the directory.

```
GitWorkshop$ cd samples2
GitWorkshop/samples2$ ls -a
./          ../         .git/
file1.txt   hg17.txt    hg21.txt
```

   Note there is no Python script since we did not commit it to the original repository.

4. Delete the `.git` folder. Now it's no longer a Git repository. But you still keep your working copy.

```
GitWorkshop/samples2$ rm -rf .git
GitWorkshop/samples2$ git log
fatal: not a git repository (or any of the parent
directories): .git
GitWorkshop/samples2$ ls
file1.txt  hg17.txt   hg21.txt
```

5. Change back to the GitWorkshop directory and clone the ISAB repository locally.

```
GitWorkshop$ git clone isabrepo repo1
Cloning into 'repo1'...
done.
GitWorkshop$ ls repo1
crypto/          ds/              octotrooper.png  regex/
distributions/   git/             readme.md
```

6. Make a **bare** clone of the ISAB repository. A *bare* repository is one with no working copy, just the repository itself.

```
GitWorkshop$ git clone --bare isabrepo repo2
Cloning into bare repository 'repo2'...
done.
GitWorkshop$ ls repo2
HEAD         config       hooks/       objects/     refs/
branches/    description  info/        packed-refs
```

Clone Notes

- Bare clones are typically used for hosting on servers. One can fetch from any repository. But only a bare repository may be pushed to from another clone.
- Note how fast the clone operation is locally.
- Local clones are a good way to experiment with new commands without damaging your current repository. Just create a clone and test the new command there.
- The remote clone operation compresses objects before transmission. It's very efficient.

- Step 4 showed how easy it is to strip the repository away from a working copy. Look back at the diagram after Exercise 1 to compare the same task with SVN. This shows how easy it is to
  - turn a set of working files into a Git repository
  - remove the Git repository for a working set of files.

  Git is very agile in this sense. Adding or removing a repository for a working set is quick and easy.

Basic Lifecycle

Once you have a local copy of the repository, the basic lifecycle goes like this.

1. **checkout** – change to a branch; often create a new one.
2. **edit** – edit working copy
3. **stage** – add the changes to a staging area
4. **commit** – commit changes to repository
5. **repeat** – steps 2 through 4
6. **push** – synchronize commits to a remote repository

In the case of a local-only repository, there is no Step 6.

The basic lifecycle will be reinforced in Exercise 4 staring on the next slide.

The following conventions apply to Git Comments

- The first line should be a brief summary with fewer than 50 characters and end with a period. That's because many reporting tools summarize Git commit comments by using the first line only. These reports look nicer if the summary is short.
- If there is more detail to provide, start the detail on the third line. Leave the second line blank.
- Comment lines started with the third line have no conventions for length. But generally it's good to keep them under 100 characters.

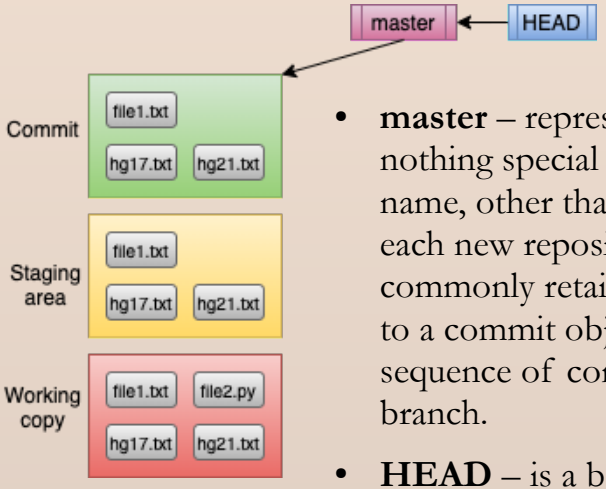
Generally you do **not** need to provide

- the date – this is provided automatically
- the author – this defaults to the committer.
- the changed files – this is obtainable through other Git commands. But you're welcome to list them in the Git comment in order to provide notes about their respective changes.

Exercise 4 – Second Commit

In this exercise we're going to reinforce the basic lifecycle with another commit. We'll continue using the `GitWorkspace/samples` directory. At this point, we should have the the configuration on the right.

As we progress through the exercise, we'll track two pointer objects that are important for understanding branches: **master** and **HEAD**.



- **master** – represents a branch. There is nothing special about this branch or its name, other than Git creates one for us with each new repository. It is, however, commonly retained and used. It's a pointer to a commit object, usually the last one in a sequence of commit objects representing a branch.
- **HEAD** – is a bookmark of sorts. It helps Git determine where to apply its commands. Since we intend for our commands to apply to a particular branch, HEAD usually references a branch pointer rather than directly to a commit. Hence it's usually a pointer to a pointer.

We're going to simulate to edits in this exercise:

- Change a file that has already been committed.
- Add the Python file that was excluded before.

Exercise 4 – Oops

Let's simulate an accidental deletion.

1. Delete `hg17.txt`. This deletes the working copy only.
2. Verify the status.

```
git status
```

3. Of course, `hg17.txt` still exists. The following command will restore `hg17.txt` from the **staging area** to the **working copy**

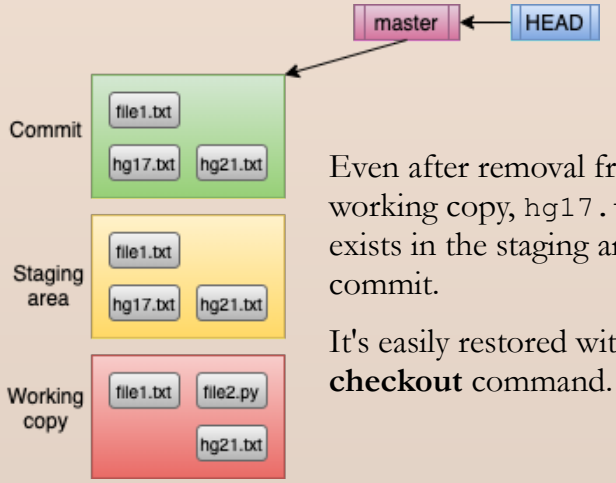
```
git checkout -- hg17.txt
```

   and verify the status.

4. Open hg17.txt inside a text editor.

```
From: <http://www.p2r.se/music/disaster.htm>

Chapter 17

The Hitch Hiker's Guide to the Galaxy notes that
**Disaster Area**, a plutonium rock band from the
```

5. Delete the first two lines so that the first line is "Chapter 17". Then change "Hitch Hiker's" to "Hitchhiker's". Save the file and close the editor.



Even after removal from the working copy, `hg17.txt` still exists in the staging area and as a commit.

It's easily restored with the **checkout** command.

Note on Double-Dashes

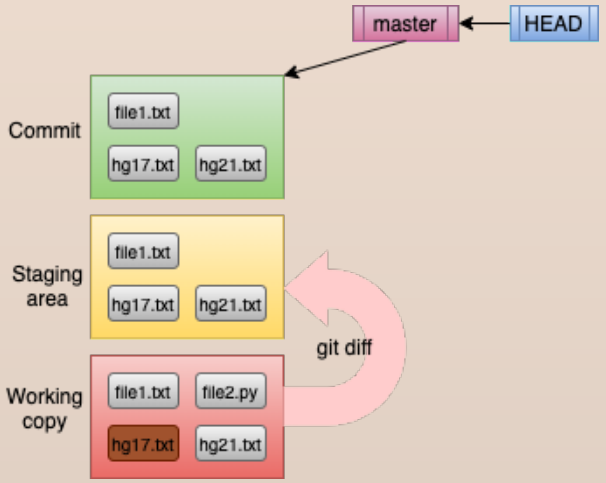
In Step 3, double-dashes seperate the checkout command from the name of the file. In this instance, it's optional. In general, it is used to separate the name of the branch (which comes first) from the name of the file. Both parameters are optional. Without double-dashes, Git will search for a branch named `hg17.txt`. Not finding one, it will then interpret `hg17.txt` as a file or directory name. Using double-dashes forces this interpretation.

Double-dashes are required when a branch namd conflicts with a directory or file name.

Exercise 4 – Diff

6. Run the diff command to verify your changes.

```
GitWorkshop/samples$ git diff
diff --git a/hg17.txt b/hg17.txt
index d330bb2..1e9969d 100644
--- a/hg17.txt
+++ b/hg17.txt
@@ -1,8 +1,6 @@
-From: <http://www.p2r.se/music/disaster.htm>
-
 Chapter 17

-The Hitch Hiker's Guide to the Galaxy notes that
+The Hitchhiker's Guide to the Galaxy notes that
 **Disaster Area**, a plutonium rock band from the
 Gagrakacka Mind Zones, are generally held to be not
 only the loudest rock band in the Galaxy, but in
```



The git diff command, by default, compares the working copy of the file to the staging area (not the committed copy).

Reading **diff** Output

The output from git diff is similar to the Unix diff output. They represent what must happen to convert the old copy of the file to the new copy:

- lines starting with **minus** must be removed
- lines starting with **plus** must be added

other lines are provided for context. In the example above, we see the first two lines removed; the "Hitchhiker" line was changed.

Exercise 4 – Add

7. After verifying the change, add hg17.txt to the staging area.

```
git add hg17.txt
```

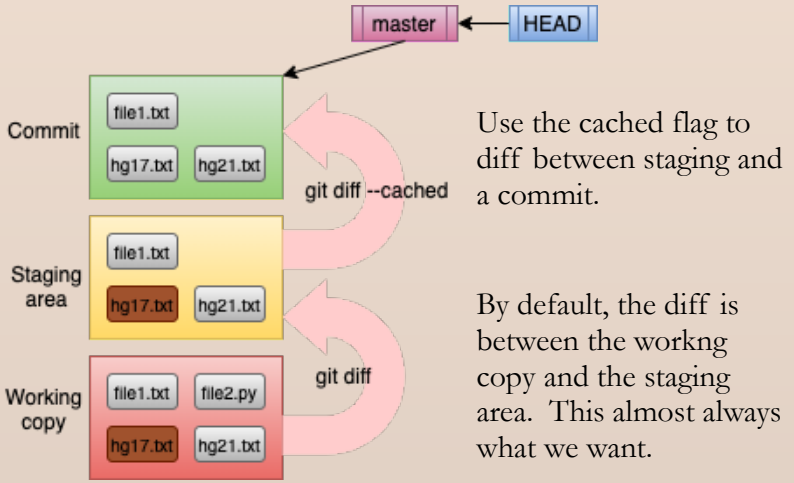
8. Try the **diff** command again. It should show no changes. That's because it only compares the staging area to the working directory copy. If your change has already been added to the staging area, it is consistent with the working copy. To check the difference between the staging area and the latest commit, add the `--cached` flag.

```
git diff --cached
```

9. We'll use a short cut to add the Python program.

```
git add .
```

   This says add everything (recursively) starting with the current directory (the . means current directory). In our case this is what we want.



Use the cached flag to diff between staging and a commit.

By default, the diff is between the workng copy and the staging area. This almost always what we want.

It's always good to check the status befor committing.

Exercise 4 – Commit

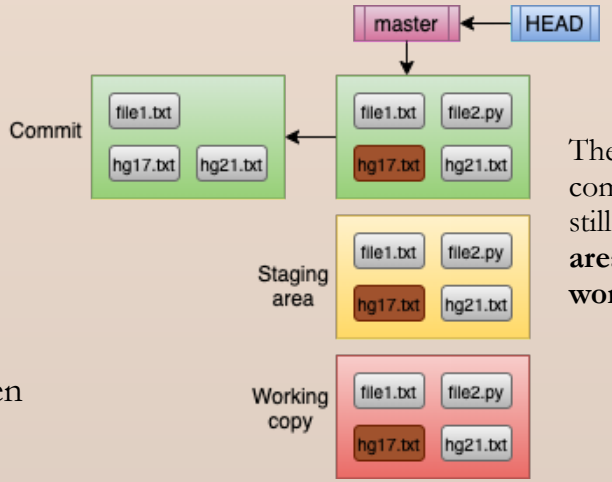
10. Run the commit.

```
git commit -m "Added Python and fixed typos."
```

The result is a new commit object. (There continues to be a single staging area and working copy.

A few things to note:

- The master pointer automatically advances to the next commit.
- Since the HEAD references master, it is implicitly advanced.
- The new commit points backwards in time to the previous commit. This is another subtle but important difference between Git and tools like CVS and SVN.



There are now **two** commits. But there is still a **single staging area** and **single working copy**.

Git Log

The **git log** command displays information about commits. As simple as this sounds, there is a bewildering number of options to customize what you hide, what you see and and how you see it. Here is a basic form of the **git log** command.

```
GitWorkshop/samples$ git log
commit 10f629df8aab162c65f80112d2c2406095d8dfc6 (HEAD -> master)
Author: Paul Glezen <bs193538@gmail.com>
Date:   Mon Mar 30 21:02:13 2020 -0700

    Added Python and fixed typos.

commit b83eb9bed3deb85a32b12e7679a18d28765bb0de
Author: Paul Glezen <bs193538@gmail.com>
Date:   Sun Mar 29 13:15:14 2020 -0700

    Initial version.
```

Note the entries are in **reverse** chronological order.

Our training repository is still very small. In practice there are usually far more commits than you want to see. The following exercise provides practice in filtering log output.

Exercise 5 – Log Filtering

1. In your command line, change to the `GitWorkshop/isabrepo` directory. This repository has too many commits to see on a single screen.
2. List the last four commits.

   ```
   git log -4
   ```

   This is the most common way to limit the output. Forgetting this option usually floods your screen as a lesson to remember it next time.
3. It's common to abbreviate the output to an entry per line. The `--oneline` option does this.

   ```
   git log -5 --oneline
   ```

Git Log

4. While there are many Git log options, it's too cumbersome to type more than a few. Instead, most people collect their favorite options into a Git **alias**. Most modern Git installations include a few log aliases out-of-the-box.

```
git alias | grep log
l      => log --graph --all --
pretty=format:'%C(yellow)%h%C(cyan)%d%Creset %s
%C(white)- %an, %ar%Creset'
ll     => log --stat --abbrev-commit
lol    => log --pretty=format:"%h %s" --graph
```

   The alias is on the left of the =>, the expanded command is on the right.

5. List the last four entries using the `l` (letter el) alias.

```
git l -4
*   3307dd6 (HEAD -> master, origin/master,
origin/HEAD) Merged remote-tracking branch
origin/master. - Paul Glezen, 4 weeks ago
|\
| * 79e14d9 Distribution supplement from last year;
forgot to commit. - Paul Glezen, 6 months ago
* | f2429ae Minor updates to GPG1. - Paul Glezen, 4
weeks ago
|/
* c551df5 Added workshop PDF for GPG 1. - Paul
Glezen, 6 months ago
```

6. Add your own alias named `logdate`.

```
git config --global alias.logdate "log --
pretty=format:'%h %cd [%an] %s' --graph --
date=short"
```

   Recall that adding `--global` to the **config** command makes it available to all your repositories. You can run this alias as

```
git logdate -2
```

7. Another way to restrict the commits is through relative time.

```
git l --since 1.month
* 3307dd6 (HEAD -> master, origin/master,
origin/HEAD) Merged remote-tracking branch
origin/master. - Paul Glezen, 4 weeks ago
* f2429ae Minor updates to GPG1. - Paul Glezen, 4
weeks ago
```

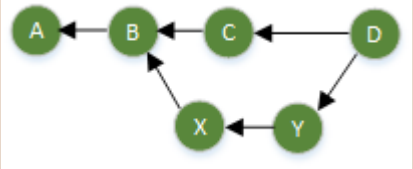
8. You can also restrict to commits applied to a directory hierarchy or to a single file.

```
git l readme.md
* 51a6147 Added workshop PDFs for Crypto Fundamentals -
Paul Glezen, 8 months ago
* eec21ef Documentation notes. - Paul Glezen, 2 years, 8
months ago
* d85ac62 Added section on data analysis workshops. - Paul
Glezen, 2 years, 10 months ago
```

Branches

Git is designed to make branching easy and efficient. The key to understanding branches is to understand commit trees.

A commit tree is a DAG (**D**irected **A**cyclic **G**raph).



- It's a **graph** in that it has vertices and edges (or points and lines if you prefer).
- It's **directed** in that each edge has a direction.
- It's **acyclic** in that you can't traverse a cycle (loop) by following edges in their prescribed direction.

A Git tree is a DAG where the vertices are commits and the directed edges represent changes between two commits.

- Git arrows point to the past.
  - This is the opposite of most other source control tools.
  - Node **B** represents the beginning of a branch.
  - Node **D** represents a *merge*.
- DAG diagrams are good to draw on paper until you get used to thinking of them in your head.
- Branches themselves don't have names. They are referenced though branch pointers. Once the pointers are gone (as in the diagram to the left), the branches are anonymous.

Basic Branch Scenario

- Most work is committed to a branch other than master
  - The branch can be dedicated to a team or a single person.
  - It usually represents a task
- Commits show up on master through a merge.
  - If there have been no intervening commits, the result is a fast-forward operation (the trivial case).
  - Otherwise a merge occurs
    - Conflict Free – still easy
    - Conflicts – conflicts require manual resolution.

Most work is committed to a branch other than master. A branch can be dedicated to a team or a single person. It usually represents a task.

Commits arrive on the master branch through a merge in three ways:

- **Fast-forward** – In this simplest case, there were no intervening merges between branch and merge. So it's as if there was no branch at all.
- **Conflict Free** – In this case there was an intervening merge, but the two sets of changes applied to different lines of source code, so the merge was applied automatically.
- **Conflicts** – In this case, two changes sets attempted to change the same line of code in different ways. This must be manually resolved before the merge can complete.

The following exercise will address the first two of these. The conflict scenario will be addressed in Part 2 of this workshop.

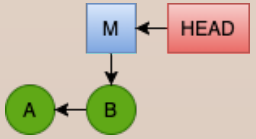
Exercise 6 – Fast Forward Merge

1. To prepare for this exercise, we'll create two clones of the repository. First change to the `GitWorkshop` directory and delete the `samples2` directory we created during Exercise 3.
2. Create a clone and change to its directory.

```
GitWorkshop$ rm -rf samples2
GitWorkshop$ git clone samples samples1
Cloning into 'samples1'...
done.
GitWorkshop$ cd samples1
GitWorkshop/samples1$ git log --oneline
10f629d (HEAD -> master, origin/master, origin/HEAD)
Added Python and fixed typos.
b83eb9b Initial version.
```

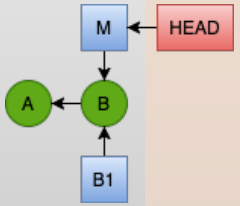
At this point our repository is represented by this diagram.

- **A** is the first commit
- **B** is the second commit
- **M** is the master branch pointer
- **HEAD** is the current branch



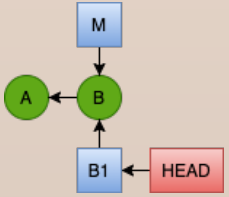
3. Create a branch named B1 **and** make it the current branch.

```
GitWorkshop/samples1$ git branch
* master
GitWorkshop/samples1$ git branch B1
GitWorkshop/samples1$ git branch
  B1
* master
GitWorkshop/samples1$ git checkout B1
Switched to branch 'B1'
GitWorkshop/samples1$ git branch
* B1
  master
```



You can use the `git branch` command at any point to view a list of branches. The asterisk denotes the current branch (HEAD).

Note that the `git branch B1` command merely created the B1 branch (created the B1 box on the upper right), it didn't move HEAD to point to it. For that we had to issue the `git checkout` command.



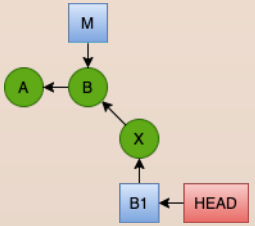
Both these actions could have been effected with a single `checkout` command with the `-b` flag.

```
GitWorkshop/samples1$ git checkout -b B1
Switched to a new branch 'B1'
GitWorkshop/samples1$ git branch
* B1
  master
```

Exercise 6 – Fast Forward Merge

4. Edit `hg21.txt`. Remove the first two lines that contain a dead URL and a blank line. Save the change.
5. Commit this change to the **B1** branch.

```
GitWorkshop/samples1$ git status
On branch B1
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:   hg21.txt

GitWorkshop/samples1$ git add .
GitWorkshop/samples1$ git commit -m "Pruned dead URL from Ch 21."
[B1 c2e8b4e] Pruned dead URL from Ch 21.
```



The asterisk next to **B1** in the `git branch -v` output (below) shows that **B1** is still our current branch. **master** is still pointing to **B1**. But **B1** has advanced to commit **X**.

I drew the **X** commit at an angle to express my intention that this commit is a branch separate from master. But there is nothing in the graph that makes this so. I just drew it this way. Let's reconcile the logs and branch pointers.

```
GitWorkshop/samples1$ git log --oneline
c2e8b4e (HEAD -> B1) Pruned dead URL from Ch 21.
10f629d (origin/master, origin/HEAD, master) Added Python and
fixed typos.
b83eb9b Initial version.
GitWorkshop/samples1$ git branch -v
* B1     c2e8b4e Pruned dead URL from Ch 21.
  master 10f629d Added Python and fixed typos.
```

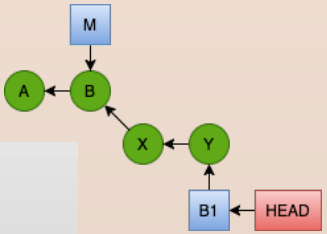
Exercise 6 – Fast Forward Merge

6. Edit `hg17.txt`. Change "songs" to "compositions" in line 17. Save the file.
7. Commit this second change to the **B1** branch.

```
GitWorkshop/samples1$ git diff
diff --git a/hg17.txt b/hg17.txt
--- a/hg17.txt
+++ b/hg17.txt

-Their songs are on the whole very simple and mostly
+Their compositions are on the whole very simple and mostly

GitWorkshop/samples1$ git add .
GitWorkshop/samples1$ git commit -m "Songs to compositions."
[B1 81d60de] Songs to compositions.
```

In the exampe above I display abbreviated output from the **diff** command. Below the log reflects that **HEAD → B1** and is two steps ahead of master.

```
GitWorkshop/samples1$ git log --oneline
81d60de (HEAD -> B1) Songs to compositions.
c2e8b4e Pruned dead URL from Ch 21.
10f629d (origin/master, origin/HEAD, master) Added Python and
fixed typos.
b83eb9b Initial version.
```



Step 8 – Create new Clone

Follow the steps below to create a new clone named `samples2`. Notice how branch **B1** was copied, but not **master**. That's because **HEAD → B1**. We create a new local master based on the value `remote/master` (to be explained in Part 2). Then change back to `samples1` to continue the exercise. We'll come back to `samples2` in Exercise 7.

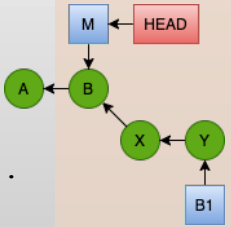
```
GitWorkshop/samples1$ cd ..
GitWorkshop$ ls
isabrepo/ repo1/    repo2/    samples/  samples1/
GitWorkshop$ git clone samples1 samples2
Cloning into 'samples2'...
done.
GitWorkshop$ cd samples2
GitWorkshop/samples2$ git branch -av
* B1                     81d60de Songs to compositions.
  remotes/origin/B1      81d60de Songs to compositions.
  remotes/origin/HEAD    -> origin/B1
  remotes/origin/master 10f629d Added Python and fixed
GitWorkshop/samples2$ git branch master origin/master
Branch 'master' set up to track remote branch 'master'
from 'origin'.
GitWorkshop/samples2$ git branch -v
* B1     81d60de Songs to compositions.
  master 10f629d Added Python and fixed typos.
GitWorkshop/samples2$ cd ../samples1
GitWorkshop/samples1$
```

Exercise 6 – Fast Forward Merge

Now we're ready to merge our changes to the **master** branch. The target for a Git merge operation is **always** the current branch. If we want to merge **B1** into **master**, we first have to make **master** the current branch. The source of the merge will be referenced in the **merge** command.

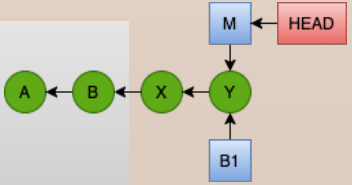
9. Change the current branch from **B1** to **master**.

```
GitWorkshop/samples2$ git branch
* B1
  master
GitWorkshop/samples2$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
GitWorkshop/samples2$ git branch
  B1
* master
```



10. Run the **merge** command referencing the **B1** branch.

```
GitWorkshop/samples2$ git merge B1
Updating 10f629d..81d60de
Fast-forward
 hg17.txt | 2 +-
 hg21.txt | 2 --
 2 files changed, 1 insertion(+), 3 deletions(-)
```

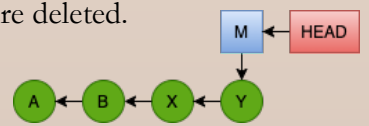


The **X** and **Y** commits constitute the work done on the **B1** branch. Since no other commits had been made to the **master** branch, the merge was a *fast-forward merge*. This happens when the merge is logically equivalent to having applied the commits directly to **master** without branching. In this case, there is **no additional commit**. The **Y** commit becomes the head of the branch for both **master** and **B1**.

Note that immediately after the merge, **master** is still the current branch. If there is no more work to be done on the **B1** branch, we may delete it.

```
GitWorkshop/samples2$ git branch
  B1
* master
GitWorkshop/samples2$ git branch -d B1
Deleted branch B1 (was 81d60de).
GitWorkshop/samples2$ git branch
* master
```

Note that "deleting the **B1** branch" only amounts to deleting the **B1** pointer. No commits were deleted.



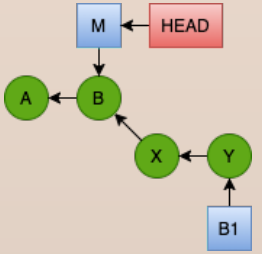
Turn Back the Clock

Next we'll simulate a case where a commit was added to the **master** branch **before** our merge operation. To this end, let's go "back in time" by changing to our `GitWorkshop/samples2` directory. We created this directory in Step 8 of the last exercise.

```
GitWorkshop/samples1$ cd ../samples2
GitWorkshop/samples2$ git log --oneline
81d60de (HEAD -> B1) Songs to compositions.
c2e8b4e Pruned dead URL from Ch 21.
10f629d (origin/master, origin/HEAD, master) Added Python and
fixed typos.
b83eb9b Initial version.
GitWorkshop/samples2$ git branch
* B1
  master
GitWorkshop/samples2$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
GitWorkshop/samples2$ git branch
  B1
* master
GitWorkshop/samples2$
```

In `GitWorkshop/samples2`, **B1** is still the current branch (**HEAD** points to it). We also confirmed this with the **branch** command. We changed the current branch to **master** using the **checkout** command.

After all this, we're back-in-time to before the **B1** merge.




```
GitWorkshop/samples2$ git log --oneline
10f629d (HEAD -> master, origin/master,
origin/HEAD) Added Python and fixed typos.
b83eb9b Initial version.
```

Add a Commit to Master

Edit `hg17.txt`. Note that the "`songs`" → "`compositions`" change on line 17 is no longer there. Edit line 10 to change

`thirty-seven` → `forty-two`

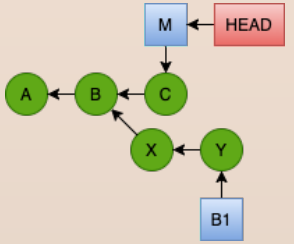
and save the file. Run the **git diff** command to confirm your change.

```
GitWorkshop/samples2$ git diff
diff --git a/hg17.txt b/hg17.txt
-bunkers some thirty-seven miles from the stage,
+bunkers some forty-two miles from the stage,
```

Add this change as a commit.

```
GitWorkshop/samples2$ git add .
GitWorkshop/samples2$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
      modified:   hg17.txt

GitWorkshop/samples2$ git commit -m "Forty-two miles."
[master 4878aee] Forty-two miles.
 1 file changed, 1 insertion(+), 1 deletion(-)
```

This latest change is represented as commit **C** in the updated diagram. When we list the log, we only see entries *reachable* from commit **C**.




```
GitWorkshop/samples2$ git log --oneline
4878aee (HEAD -> master) Forty-two miles.
10f629d (origin/master, origin/HEAD) Added Python
and fixed typos.
b83eb9b Initial version.
```

This concept of *reachable* is important for understanding many Git operations. A node in the Git graph (**D**irected **A**cyclic **G**raph or DAG) is *reachable* from a point C if it may be reached from C by traversing the edges of the graph in the direction of the arrows.

In the diagram above, commits **A**, **B** and **C** are reachable from **C**. **X** and **Y** are not reachable from **C**.

By default, the **log** command only displays commits reachable from **HEAD**.

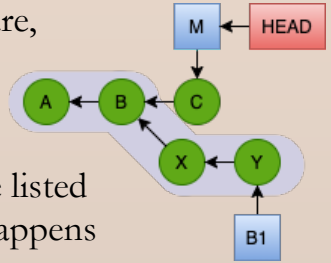
Displaying Branch Commits

By providing a branch name to the log command, we can see all log entries for commits reachable from that branch. To see the commits reachable from **B1**:

```
GitWorkshop/samples2$ git log --oneline B1
81d60de (B1) Songs to compositions.
c2e8b4e Pruned dead URL from Ch 21.
10f629d (origin/master, origin/HEAD) Added Python
and fixed typos.
b83eb9b Initial version.
```

The commits reachable from **B1** are, following the arrows:

Y → X → B → A.



This is the order in which they are listed by the **log** command. This also happens to be reverse chronological order.

In this way, we can view logs from all the commits on any particular branch. We are not restricted to viewing the log history of **HEAD**. **HEAD** is simply the default.

But **all the commits** reachable by any particular node is usually much more than we want. Our sample repository is small. Real life repositories have hundreds of commits. We saw earlier a common way to restrict the output is with **`-N`** where N is a number.

```
GitWorkshop/samples2$ git log --oneline -2 B1
81d60de (B1) Songs to compositions.
c2e8b4e Pruned dead URL from Ch 21.
```

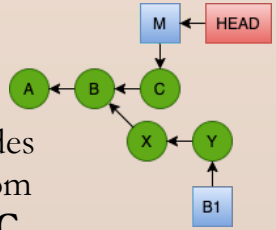
In the example above, we used `-2` to restrict the history to two entries.

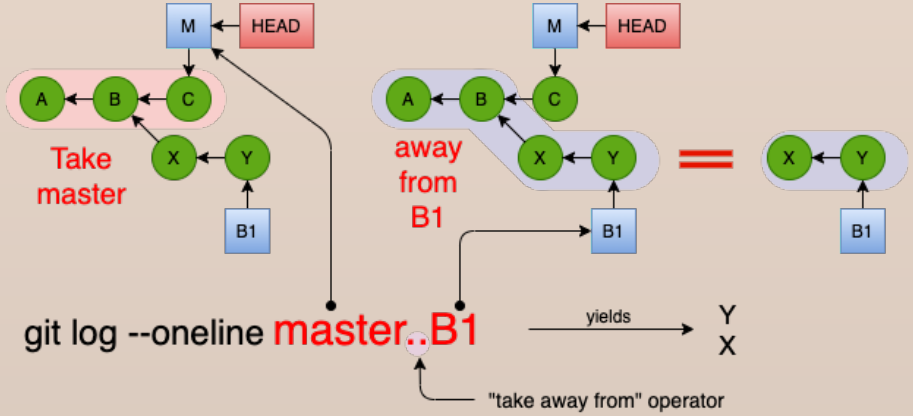
Another common inquiry is: "List the commits on a branch since the branch was created." For this kind of inquiry, certain Git command support *set subtraction*, denoted by the `..` operator (two periods).

Log Set Subtraction – Two Dots

A common inquiry in a branch scenario is to list the commits that have occurred since the branch. In our current example, the branch point is node B. Node C has been added to the master branch and nodes X and Y have been added to the B1 branch.

But how do we obtain this information without a pretty picture drawn for us every time? The answer is *set subtraction*. The nodes unique to **B1** are all the nodes reachable from **Y** *minus* all the nodes reachable from node **C**.

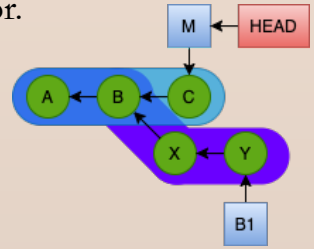





```
GitWorkshop/samples2$ git log --oneline master..B1
81d60de (B1) Songs to compositions.
c2e8b4e Pruned dead URL from Ch 21.
```

We can ask the question in reverse: which commits have been made to master since the B1 branch? We just flip the order of the arguments in the difference operator.

Take all nodes reachable from **Y**	. .	away from all nodes reachable from **C**



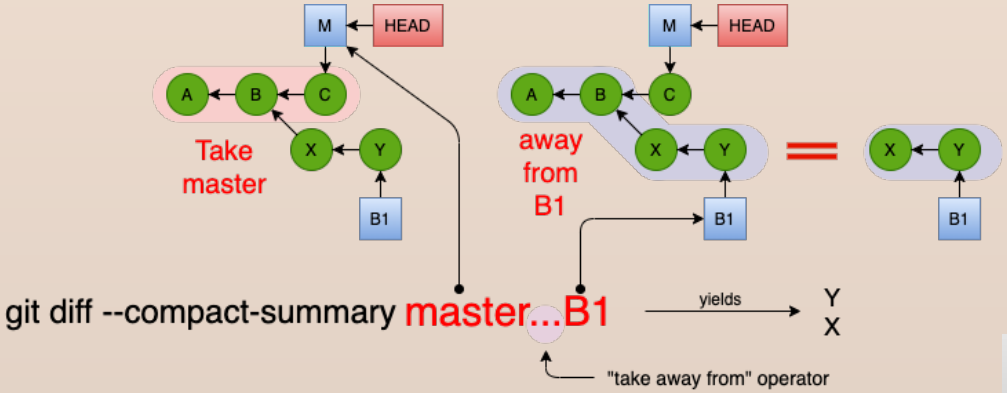

```
GitWorkshop/samples2$ git log --oneline B1..master
4878aee (HEAD -> master) Forty-two miles.
```

The difference operator assumes **HEAD** when the commit is omitted. Since **HEAD** is currently **master**, the last two commands can be abbreviated as follows.

```
GitWorkshop/samples2$ git log --oneline ..B1
81d60de (B1) Songs to compositions.
c2e8b4e Pruned dead URL from Ch 21.
GitWorkshop/samples2$ git log --oneline B1..
4878aee (HEAD -> master) Forty-two miles.
```

Diff Set Subtraction – Three Dots

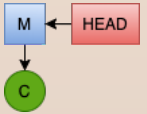
A similar concept applies to the **diff** command, except the set subtraction operator is three dots instead of two.




```
GitWorkshop/samples2$ git diff --compact-summary master...B1
 hg17.txt | 2 +-
 hg21.txt | 2 --
 2 files changed, 1 insertion(+), 3 deletions(-)
```

This shows the difference between the branch-point of **master** and **B1** (node B) and the lastest commit on **B1**.

In the example below, we can see the difference along the master branch. The branch point between **B1** and **master** is still the same, node B. But the change is the node C we made on the master branch.




```
GitWorkshop/samples2$ git diff --compact-summary B1...master
 hg17.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Diff Symmetric Difference – Two Dots

Set symmetric difference is not the same as set subtraction.
The *symmetric difference* between sets A and B is their union take away their intersection.

$$A \Delta B = A \cup B - A \cap B$$

This is everything that is either in A or in B, but not in both.