

Git Workshop

Part 2 of 2

Merge Conflicts, Remote Repositories
and
Hosting Services

Workshop Agenda

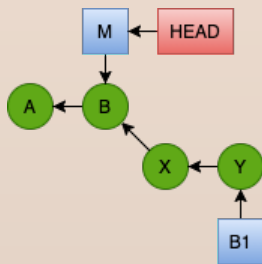
- Part 1 – Another Deck
 - Installation and Setup
 - Concepts
 - Repository Initialization
 - Clone
 - Basic Lifecycle
 - Logs
 - Introduction to Branches
- Part 2 – This Deck
 - Merges
 - Remote Repositories
 - Tags
 - Hosting Services
 - GitHub
 - AWS CodeCommit
 - Commit Signing

This is a "presentation-ification" of the HTML workshop available at
<https://www.workshops.lacounty-isab.org/>

Scenario Continued

This Part 2 continues a scenario that was started in Part 1. If you still have your files and folder from Part 1, you can skip to the next slide now. This slide explains how to recreate where we left-off from Part 1.

Part 1 ended with a merge of the situation shown on the right. It resulted in a *fast-forward* merge where the **master** branch pointer advanced to the **Y** node.



The first exercise of this Part 2 will explore an alternative to the Part 1 ending – the consequence of adding commits to the **master** branch *before* the merge. We need the branch pointers reset to how they were before the merge updated them. Step 8 of Exercise 6 cloned the state of the repository to a `samples2` directory before proceeding. If you no longer have that `samples2` directory available, the steps on the right will show you how to recreate it from a repository hosted on GitHub.

1. Change to your `GitWorkshop` directory. This is a directory in which we undertake these workshop exercises.
2. If you have a `samples2` subdirectory, you may not need to follow these steps. It may have been created during your Part 1 session. If you're unsure, delete the `samples2` directory.

3. Clone the **lacounty-isab/gitwkspex6** repository to `samples2`.

```
GitWorkshop$ git clone
https://github.com/lacounty-isab/gitwkspex6.git samples2
```

4. Change to `samples2`.
5. List branches. You'll find that **master** is the only local branch. Create a *local* **B1** branch from the **origin/B1** remote branch*.

```
GitWorkshop/samples2$ git branch -av
* master                d0d9eea Added Python and fixed typos.
remotes/origin/B1       dbc1acc Songs to compositions.
remotes/origin/HEAD     -> origin/master
remotes/origin/master   d0d9eea Added Python and fixed typos.
GitWorkshop/samples2$ git branch B1 origin/B1
Branch 'B1' set up to track remote branch 'B1' from 'origin'.
GitWorkshop/samples2$ git branch -av
B1                      dbc1acc Songs to compositions.
* master                d0d9eea Added Python and fixed typos.
remotes/origin/B1       dbc1acc Songs to compositions.
remotes/origin/HEAD     -> origin/master
remotes/origin/master   d0d9eea Added Python and fixed typos.
```

* *Remote branches* will be covered as part of *remote repositories*.

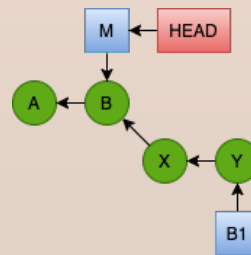
Turn Back the Clock

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,
```

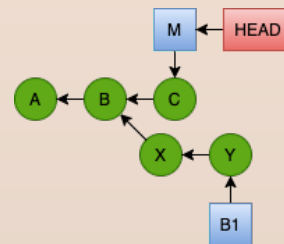
Add this change as a commit.

```
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**:

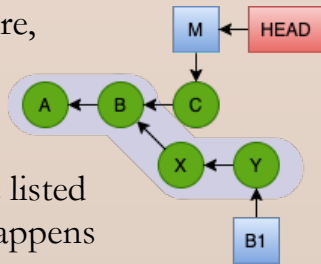
```
GitWorkshop/samples2$ git log --oneline B1
81d60de (B1) Songs to compositions.
c2e8b4e Pruned dead URL from Ch 21.
10f629d (origin/master, origin/HEAD) Added Python
and fixed typos.
b83eb9b Initial version.
```

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

$Y \rightarrow X \rightarrow B \rightarrow A$.

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.
```

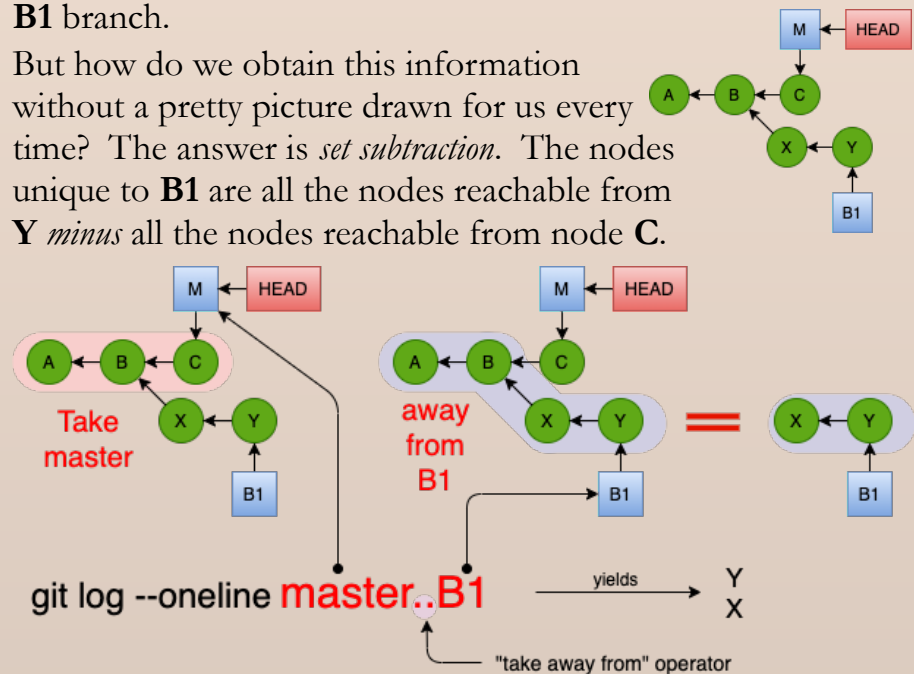
In the example above, we used **-2** to restrict the history to two entries.

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**.



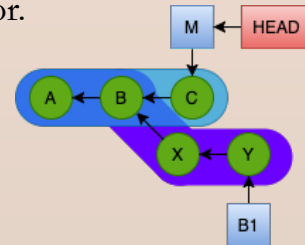
```
GitWorkshop/samples2$ git log --oneline master..B1
81d60de (B1) Songs to compositions.
c2e8b4e Pruned dead URL from Ch 21.
```

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**



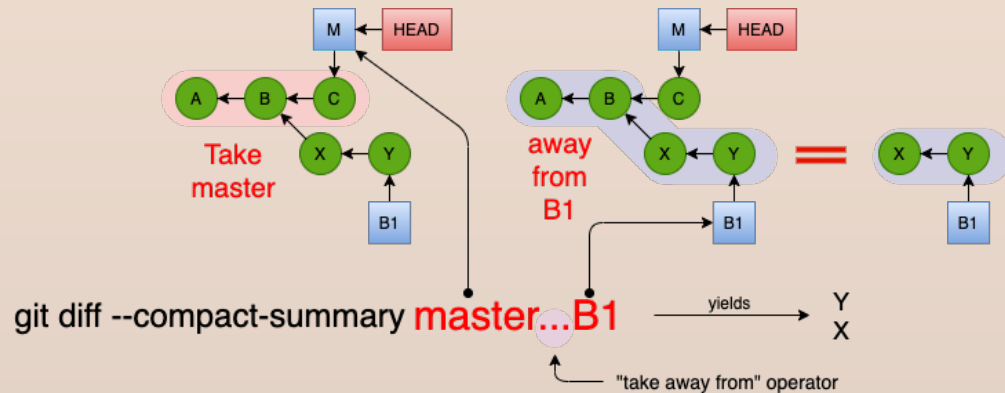
```
GitWorkshop/samples2$ git log --oneline B1..master
4878aee (HEAD -> master) Forty-two miles.
```

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

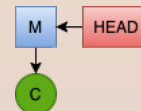
A similar concept applies to the **diff** command, except the set subtraction operator is three dots instead of two.



```
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 latest 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:

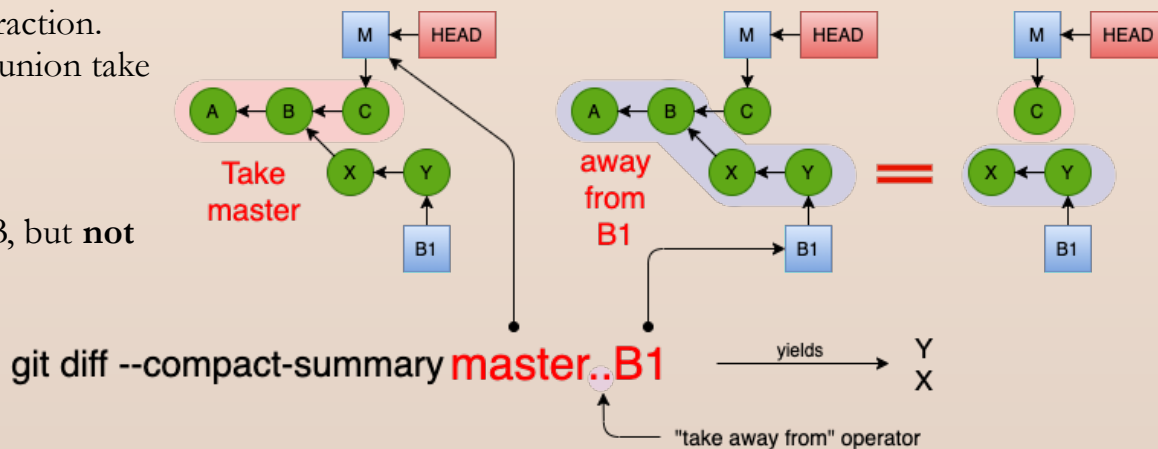
$$A \Delta B = A \cup B - A \cap B$$

The set of all elements that are in either in A or in B, but **not** in both.

Unlike displaying log messages, code changes have a "sign" or "direction". If you made a change by adding two lines to a file, then the reverse direction is removing those two lines from the file.

Despite the set theory term *symmetric difference*, the display of the change is actually **antisymmetric**. It has a distinct direction which is reversed when the order of the parameters to the operator are reversed.

In the diagram, the difference with double dots shows us how to go **from master to B1**. That means **undoing** commit C (changing "forty-two" back to "thirty-seven") and then **applying** commits X and Y.



```
GitWorkshop/samples2$ git diff --compact-summary master..B1
hg17.txt | 4 +--
hg21.txt | 2 --
2 files changed, 2 insertions(+), 4 deletions(-)
GitWorkshop/samples2$ git diff --compact-summary B1..master
hg17.txt | 4 +--
hg21.txt | 2 ++
2 files changed, 4 insertions(+), 2 deletions(-)
```

The opposite order means: start at Y, **undo** Y, **undo** X, and then **apply** C. Without the `--compact-summary` option, all the details of the changes are listed.

Exercise 7 - Merge

It's time to merge the **B1** branch to **master**. It's hard to remember what's happened on either of these two branches since the split occurred; and that's fairly realistic. So we'll start using the **git log** command technique to check what has occurred on each branch before starting the merge.

1. Summarize commits on **master** since **B1** split from **master**.

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

Just the one change on the one line.

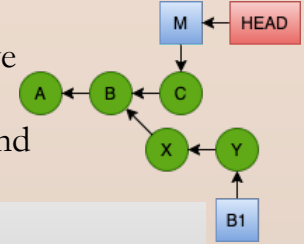
2. Summarize commits on **B1** since **B1** split from **master**.

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

3. With our memory refreshed, we proceed with the merge task. The **merge** command takes a single branch name for the source of the merge. **The target is always the current branch.** Since we want to merge into **master**, we must make **master** the current branch.

```
GitWorkshop/samples2$ git checkout master
Already on 'master'
Your branch is ahead of 'origin/master' by 1 commit.
```

As it happened, we were already on **master**. But it didn't hurt to check. We've confirmed our status depicted in the diagram on the right. The merge command itself is easy.



```
GitWorkshop/samples2$ git merge B1
Auto-merging hgl7.txt
Merge made by the 'recursive' strategy.
 hgl7.txt | 2 +-
 hgl21.txt | 2 --
 2 files changed, 1 insertion(+), 3 deletions(-)
```

4. Run the above git merge command. Git displays an editor window with the contents prepopulated as shown below.

```
Merge branch 'B1'
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

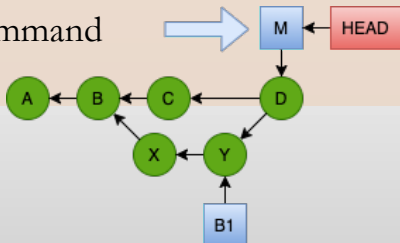
5. Replace the comment or accept it, save and quit. This creates the merge commit.

Merge Result

The merge commit is node **D** in the diagram below. Notice that commit **D** has two children: **C** and **Y**.

However, this is not apparent with the git log command we've been using.

```
GitWorkshop/samples2$ git log --online
6513089 (HEAD -> master) Merge branch 'B1'
4878aee Forty-two miles.
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.
```

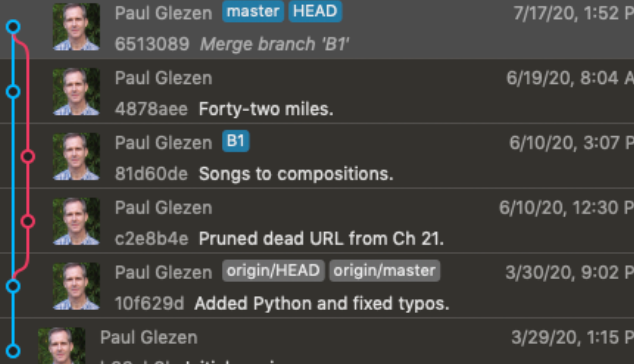


However, by adding the `--graph` option to the command, we can see the relationship of both children to the last commit.

```
GitWorkshop/samples2$ git log --graph --online
* 6513089 (HEAD -> master) Merge branch 'B1'
| \
| * 81d60de (B1) Songs to compositions.
| * c2e8b4e Pruned dead URL from Ch 21.
* | 4878aee Forty-two miles.
| /
* 10f629d (origin/master, origin/HEAD) Added Python and fixed typos.
* b83eb9b Initial version.
```

Without the graph option, the branches are flattened out. Of course, GUI tools make this visualization even nicer.

HEAD



The image shows a Git commit history interface. On the left, a vertical graph displays the commit lineage. A blue line represents the 'master' branch, and a red line represents the 'B1' branch. The blue line starts at the bottom (commit b83eb9b) and goes up to commit 6513089. The red line branches off from commit 10f629d, goes up to commit 81d60de, and then merges back into the blue line at commit 6513089. To the right of the graph is a list of commits, each with a profile picture, a commit hash, a message, the author's name, and the commit date and time.

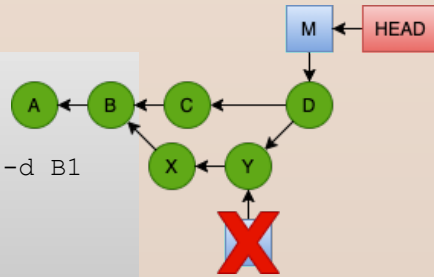
Commit Hash	Message	Author	Date
6513089	Merge branch 'B1'	Paul Glezen	7/17/20, 1:52 PM
4878aee	Forty-two miles.	Paul Glezen	6/19/20, 8:04 AM
81d60de	Songs to compositions.	Paul Glezen	6/10/20, 3:07 PM
c2e8b4e	Pruned dead URL from Ch 21.	Paul Glezen	6/10/20, 12:30 PM
10f629d	Added Python and fixed typos.	Paul Glezen	3/30/20, 9:02 PM
b83eb9b	Initial version.	Paul Glezen	3/29/20, 1:15 PM

Displaying branch histories is one of the better advantages to using a GUI Git tool. Most IDEs have this capability built in or available as a free plugin.

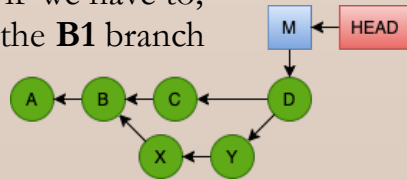
Deleting a Branch (pointer)

We're done with the **B1** branch; but the pointer is still hanging around. If we want to continue to work on a **B1** branch past this point, we probably want* to pick up contributions from the merge to **master**. To this end, we wish to remove the B1 pointer.

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

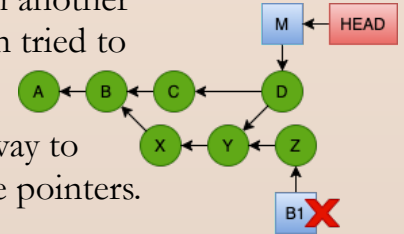


Git makes the deletion of the **B1** branch pointer easy. It was pointing to node **Y** and there was already another element pointing to **Y** (namely node **D**). So we can always reach node **Y** through **D** if we have to; we're not losing access to it by deleting the **B1** branch pointer.



What if we had continued with another commit on branch **B1** and then tried to delete **B1**?

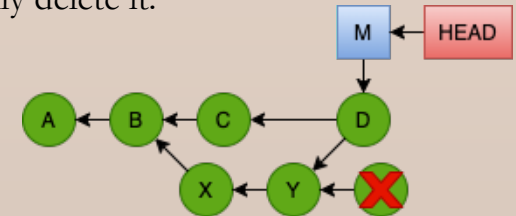
Then there would be no easy way to reach node **Z** through available pointers.



Git will recognize this and refuse the deletion with a warning that the branch is "not fully merged". Sometimes you still want to delete such a branch (for example, you want to discard any changes you made on the branch without a merge). Then you use the same command with a capital **D**.

```
git branch -D B1
```

This will delete the branch pointer regardless of whether it had been merged. After this, node **Z** is essentially unreachable since there is no path to it through available pointers. Git will eventually delete it.



* There are times when we want to continue working on B1 without the master contributions. But doing this for too long increases the risk of complicating subsequent merges.

Merge Conflicts

We are going to introduce some changes in our files on different branches that **conflict**. In this context, it means that different branches change the same line of the same file in different ways. We saw earlier that neither

- changing different files, nor
- changing the same file on different lines

incur a conflict. The **git merge** command will merge these deterministically.

In the following exercise we edit two files: a plain text file and a small Python program.

```
1 # This file contains mappings.
2 #
3 a1 - 20
4 a2 - 43
5
6 b1 - 39
7 b2 - 34
8 b3 - 44
9
10 c1 - 45
11 c2 - 19
```

file1.txt

To keep things distinct from Part 1, we'll create a new directory, sample3, with these files and create a new repository.

```
GitWorkshop/samples3$ ls
file1.txt  file2.py
GitWorkshop/samples3$ git init
Initialized empty Git repository in GitWorkshop/samples3/.git/
GitWorkshop/samples3$ git add .
GitWorkshop/samples3$ git commit -m "Initial version."
[master (root-commit) 4fff5a8] Initial version.
 2 files changed, 22 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.py
GitWorkshop/samples3$ git log --oneline
4fff5a8 (HEAD -> master) Initial version.
```

```
1 def print_usage():
2     usage = """Usage: addAudit.py [-f] [-v] <filename ...>
3         -f - overwrite when duplicate key encountered
4         -v - verbose
5         <filename ...> the name of at least one audit file."""
6
7     print(usage)
8
9 print_usage()
```

file2.py

Exercise 8 – Branch B2

Create a branch **B2** and edit the two files.

1. Create a new branch **B2**.

```
GitWorkshop/samples3$ git checkout -b B2
Switched to a new branch 'B2'
```

2. Edit `file1.txt` as shown on the right.

- a. **Line 6:** Add ",41"
- b. **Line 7:** Change "34" to "36"
- c. **Line 8:** Add a blank space after 44.
- d. **Line 10:** Change "45" to "55"
- e. **Line 11:** Change "19" to "29"

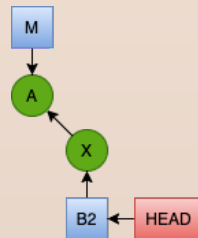
3. Edit `file2.py` by adding a blank space to each line of the `print_usage` function.

4. Check your work with `git diff`.

5. Add the changes to the staging area.

6. Commit with message "B2 changes."

This creates a new commit **X** referenced by branch **B2** as shown in the figure above.



```
1 # This file contains mappings.
2 #
3 a1 - 20
4 a2 - 43
5
6 b1 - 39,41
7 b2 - 346
8 b3 - 44
9
10 c1 - 455
11 c2 - 1929
```

file1.txt

```
1
2 def print_usage():
3     usage = """Usage: addAudit.py [-f] [-v] <filename ...>
4     -f - overwrite when duplicate key encountered
5     -v - verbose
6     <filename ..> the name of at least one audit file."""
7
8     print(usage)
9
10 print_usage()
```

file2.py

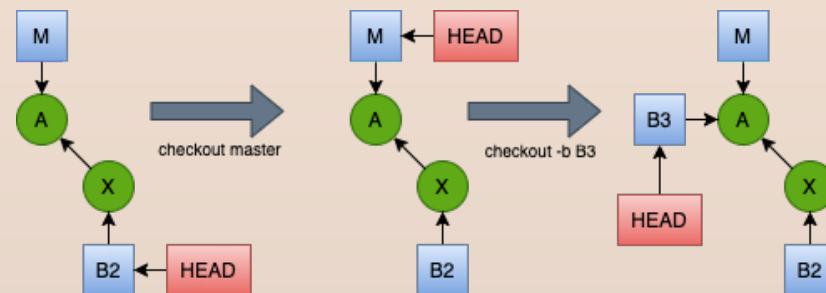
Note that some changes are not as easy to see with the **git diff** command.

Extra Credit: The edit to `file2.py` may seem peculiar.
What is a common cause for this?

Exercise 8 – Branch B3

7. Now we're going switch to a new branch **B3** starting from commit **A** just like **B2** did.

```
GitWorkshop/samples3$ git checkout master
Switched to branch 'master'
GitWorkshop/samples3$ git checkout -b B3
Switched to a new branch 'B3'
```



8. Edit file1.txt.

- a. **Line 3:** change "20" to "30".
- b. **Line 4:** change "43" to "53".
- c. **Line 6:** add ",40".
- d. **Line 7:** change "34" to "35".



```
1 # This file contains mappings.
2 #
3 a1 - 230
4 a2 - 453
5
6 b1 - 39,40
7 b2 - 345
8 b3 - 44
9
10 c1 - 45
11 c2 - 19
```

file1.txt

9. Edit file2.py. Change the triple-quoted string to a series of print statements.



```
1 def print_usage():
2     print("Usage: addAudit.py [-f] [-v] <filename ...>")
3     print("    -f - overwrite when duplicate key encountered")
4     print("    -v - verbose")
5     print("    <filename ..> the name of at least one audit file.")
6
7 print_usage()
```

file2.py

10. Check your work with git diff.
11. Add the changes to the staging area.
12. Commit with message "B3 changes."

Exercise 8 – Branch master

After the B3 commit, we have the branch configuration shown to the right. Since the target of a merge is always the current branch, we're going to change back to the **master** branch. Then merge each of **B2** and **B3**.

13. Change to the **master** branch.

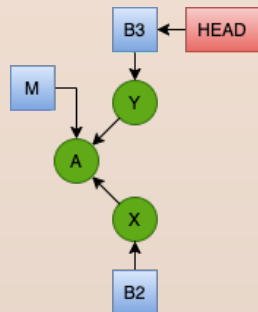
```
git checkout master
```

This moves HEAD to the master branch.

14. Merge **B2**

```
git merge B2
```

This should be a fast-forward merge which simply advances the master branch to **B2**.



15. Finally, issue the command to merge **B3**. This is where the fireworks start.

```
GitWorkshop/samples3$ git merge B3
Auto-merging file2.py
CONFLICT (content): Merge conflict in file2.py
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
Automatic merge failed; fix conflicts and then commit the result.
```

What just happened here?

Git auto-merges on a line-by-line basis. When the same line is changed in two different ways, Git places "merge markers" around those lines to indicate two incompatible changes were made. They must be manually resolved.

The changes are easy to spot. They are bounded by the markers

```
<<<< HEAD – beginning of master
```

```
===== – separates versions
```

```
>>>>> B2 – end of B3 version
```

Resolving the merge conflict amounts to choosing one or the other of these versions to keep.

```
1 # This file contains mappings.
2 #
3 a1 - 30
4 a2 - 53
5
6 <<<<<<< HEAD
7 b1 - 39, 41
8 b2 - 36
9 b3 - 44
10 =====
11 b1 - 39, 40
12 b2 - 35
13 b3 - 44
14 >>>>>>> B3
15
16 c1 - 55
17 c2 - 29
```

This is the HEAD (i.e. master) version of the lines (originally B2)

This is the B3 (i.e. incoming) version of the lines

Exercise 8 – Resolve Conflict 1

We have to decide, based on a larger perspective, how to resolve the conflicts. We could

- pick the left side (**master** branch version)
- pick the right side (**B3** branch version)
- choose something completely different from either side based on some knowledge we might have.

These decisions are carried out in the following manner.

- Edit the lines** within the merge markers based on your decisions
- Delete the merge marker** lines and save the file.
- Add the file** to the Git staging area.

This last step is how Git knows when we've completed the merge activity for the file. We repeat steps a, b and c for each file in which a merge conflict occurred.

Looking at `file1.txt` in the previous slide, things aren't that bad. The top (only changed on **B2**) and the bottom (only changed on **B3**) were auto-merged. Only the middle third, where both **B2** and **B3** changed lines, requires resolution.

16. Edit the lines of `file1.txt`.

- For entry `b1`, branch **B2** added `41` while branch **B3** added `40` to the value. Let's make the decision to add both so that line 7 has `b1 - 39, 40, 41`. Notice how we're implicitly making lines 7 – 9 our "definitive copy."
- For entry `b2`, branch **B2** changed the value to `36` which branch **B3** changed it to `35`. Let's decide to keep `36` so that line 8 remains unchanged.
- Entry `b3` is tricky. It looks the same in both lines 9 and 13. Recall that branch **B2** erroneously added a space at the end of the line. In this case, we wish to accept the **B3** line 13 which left the line unchanged. Remove the last space on line 9.

```
6 <<<<<<<•HEAD␣  
7 b1•-•39, 41␣  
8 b2•-•36␣  
9 b3•-•44•␣  
10 =====␣  
11 b1•-•39, 40␣  
12 b2•-•35␣  
13 b3•-•44␣  
14 >>>>>>>•B3␣
```



```
6 <<<<<<<•HEAD␣  
7 b1•-•39, 40, 41␣  
8 b2•-•36␣  
9 b3•-•44␣  
10 =====␣  
11 b1•-•39, 40␣  
12 b2•-•35␣  
13 b3•-•44␣  
14 >>>>>>>•B3␣
```

Exercise 8 – Resolve Conflicts 2

17. Delete the merge markers. Lines 7 – 9 are now in the form we want to keep. We can delete lines 11 – 13 as well as the merge markers on lines 6, 10 and 14. Then save the file.

```
6 <<<<<< HEAD
7 b1 - 39,40,41
8 b2 - 36
9 b3 - 44
10 =====
11 b1 - 39,40
12 b2 - 35
13 b3 - 44
14 >>>>>> B3
```



```
6 b1 - 39,40,41
7 b2 - 36
8 b3 - 44
```

```
1 def print_usage():
2 <<<<<< HEAD
3     usage = """Usage: addAudit.py [-f] [-v] <filename ...>
4         -f - overwrite when duplicate key encountered
5         -v - verbose
6         <filename ..> the name of at least one audit file."""
7
8     print(usage)
9 =====
10    print("Usage: addAudit.py [-f] [-v] <filename ...>")
11    print("    -f - overwrite when duplicate key encountered")
12    print("    -v - verbose")
13    print("    <filename ..> the name of at least one audit file.")
14 >>>>>> B3
15
16 print_usage()
```

18. Add file1.txt to the Git staging area.

```
git add file1.txt
```



19. Perform the same merge steps a, b and c for file2.py. This is a simpler case where we wish to only accept the **B3** version. The **B2** version had simply added an extra space on each line, which is not uncommon for some editors. Simply delete lines 2 – 9 and line 14. Then save the file and add it to the staging area.

```
1 def print_usage():
2     print("Usage: addAudit.py [-f] [-v] <filename ...>")
3     print("    -f - overwrite when duplicate key encountered")
4     print("    -v - verbose")
5     print("    <filename ..> the name of at least one audit file.")
6
7 print_usage()
```

Exercise 8 - Commit

20. With both file conflicts resolved and added to the Git staging area, we can now create the merge commit. It should be like any other commit; the hard work is over.

```
git commit -m "Merge branch B3"
```

21. Verify the branch activity using **git log**.

```
GitWorkshop/samples3$ git log --oneline --graph
*   55a29e6 (HEAD -> master) Merge branch B3
| \
| * 17c3dc5 (B3) B3 changes.
* | fd4d195 (B2) B2 changes.
|/
* cce1b27 Initial version.
```

And that's all there is to it. It's just a matter of recognizing the merge markers and resolving the merge the proper way.

Note the merge-marker technique only works with text files. This technique does not work for binary files, such as images, executables, compressed archives, etc. Teams must take care not to change these files concurrently. They cannot easily be merged.

Remote Repositories – Terms

A *clone* is another copy of the current repository.

- Sometimes it's called a *remote repository*. It's remote in the sense that it's in a separate directory. It doesn't have to be on a separate machine. We saw examples early in Part 1 where we created multiple remote repositories on the same machine with the `clone` command. The `clone` command is the typical way to create a remote repository. The term *clone* can refer to the remote repository, or it can refer to the Git command.
- A clone is not a strict copy. By default, the `clone` command only copies commits reachable by the master branch. Additional branches may be cloned through additional commands.
- In this presentation, we'll use the term **clone** as
 - a Git command `clone` using monospace font,
 - a noun – referring to a clone of a repository
 - a verb – the act of creating a clone with `clone`.