Advanced Git: Functionality and Features

Brent Laster (author of Professional Git)

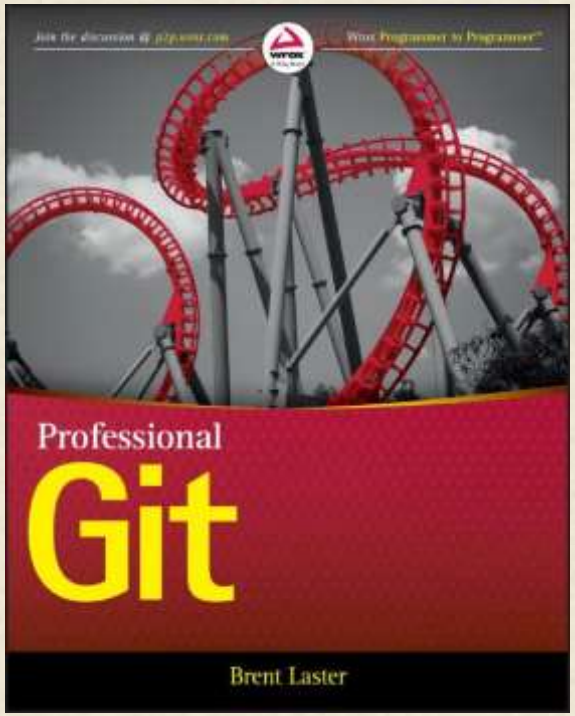
Open Source 101 Conference
February 4, 2017

About me

- Senior Manager, R&D at SAS in Cary, NC
- Global Trainer and Speaker
- Git, Gerrit, Gradle, Jenkins, Pipelines
- Author - NFJS magazine, Professional Git book
- LinkedIn https://www.linkedin.com/in/brentlaster
- Twitter @BrentCLaster

Professional Git

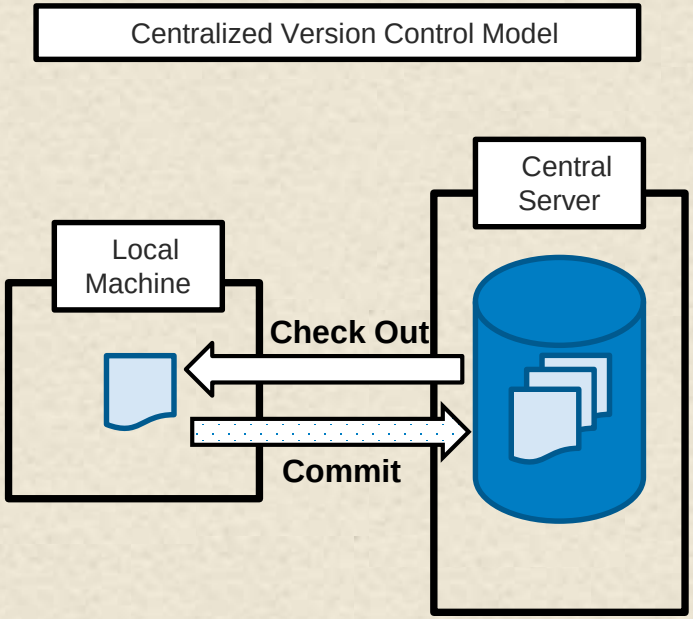
- **Available on:**
  - **Amazon.com**
  - **Wiley.com**

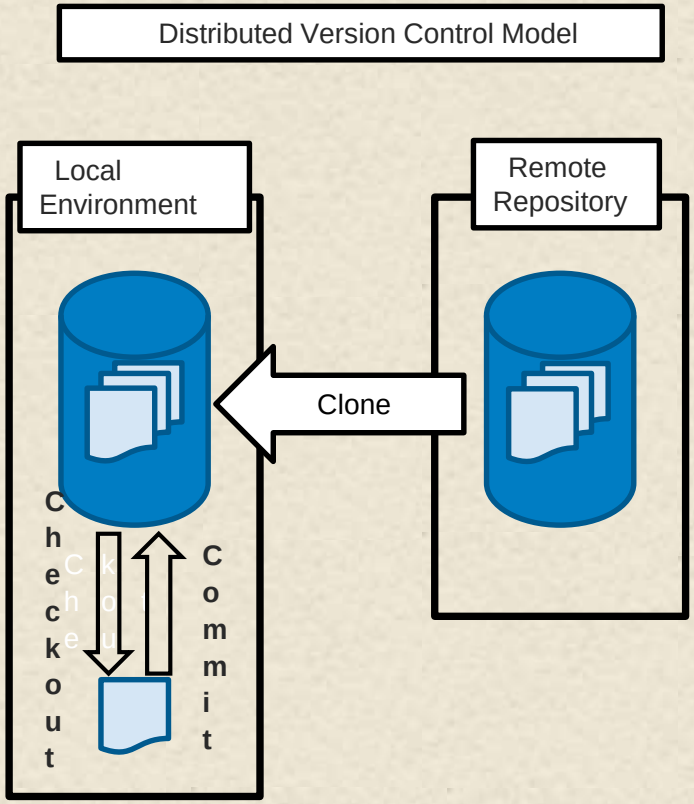


Agenda

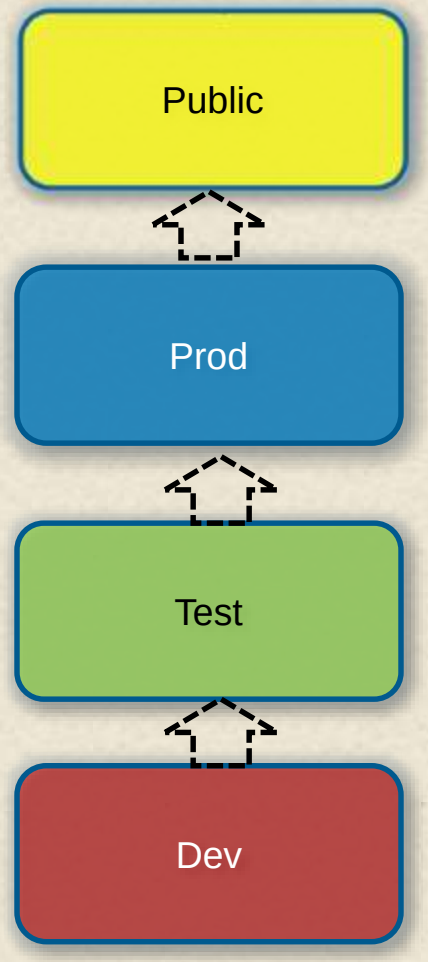
- Core concepts refresh
- Merging and Rebasing
- Stash
- Reset and Revert
- Rerere
- Bisect
- Worktrees
- Submodules
- Subtrees
- Interactive Rebase
- Notes
- Grep

Centralized vs. Distributed VCS

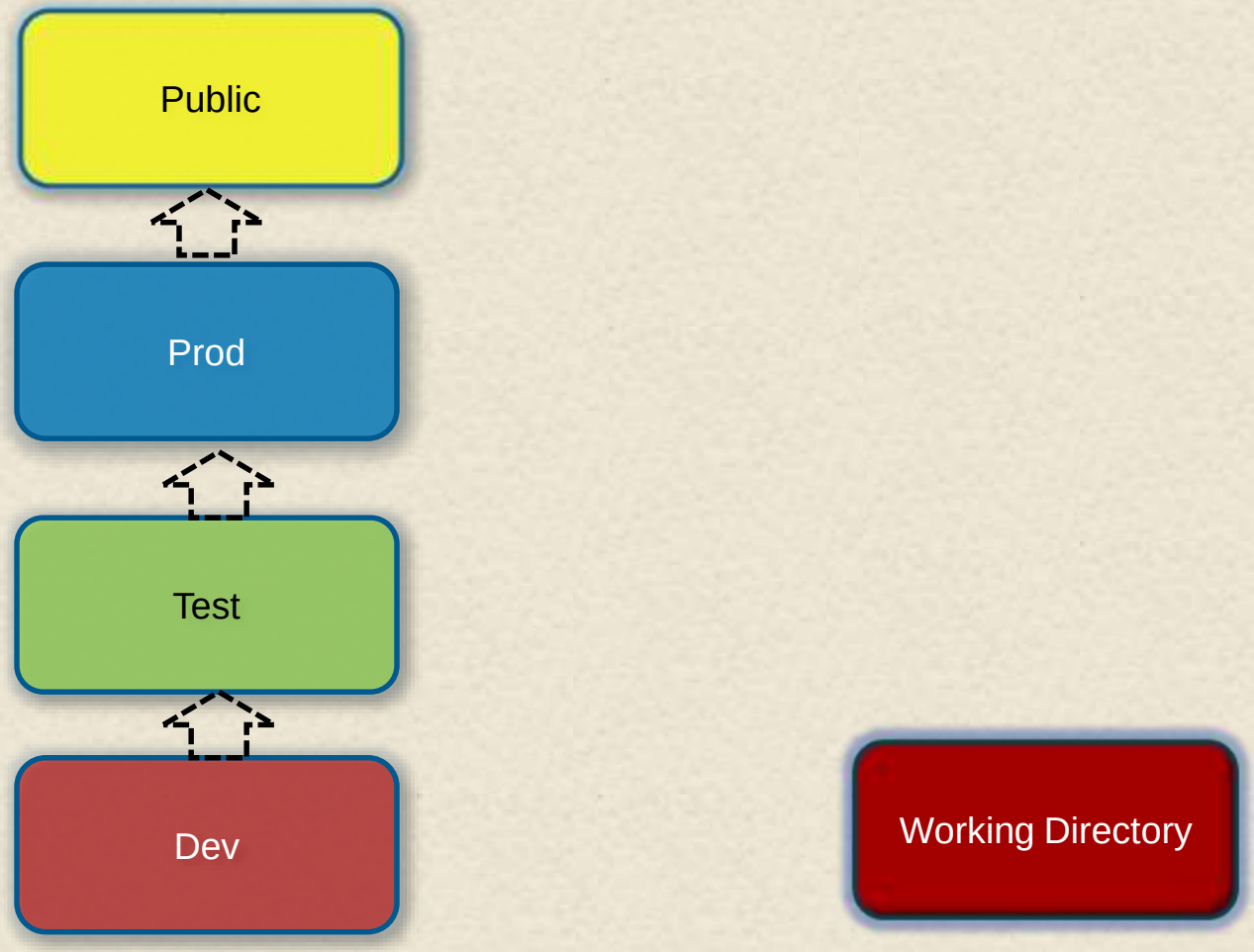




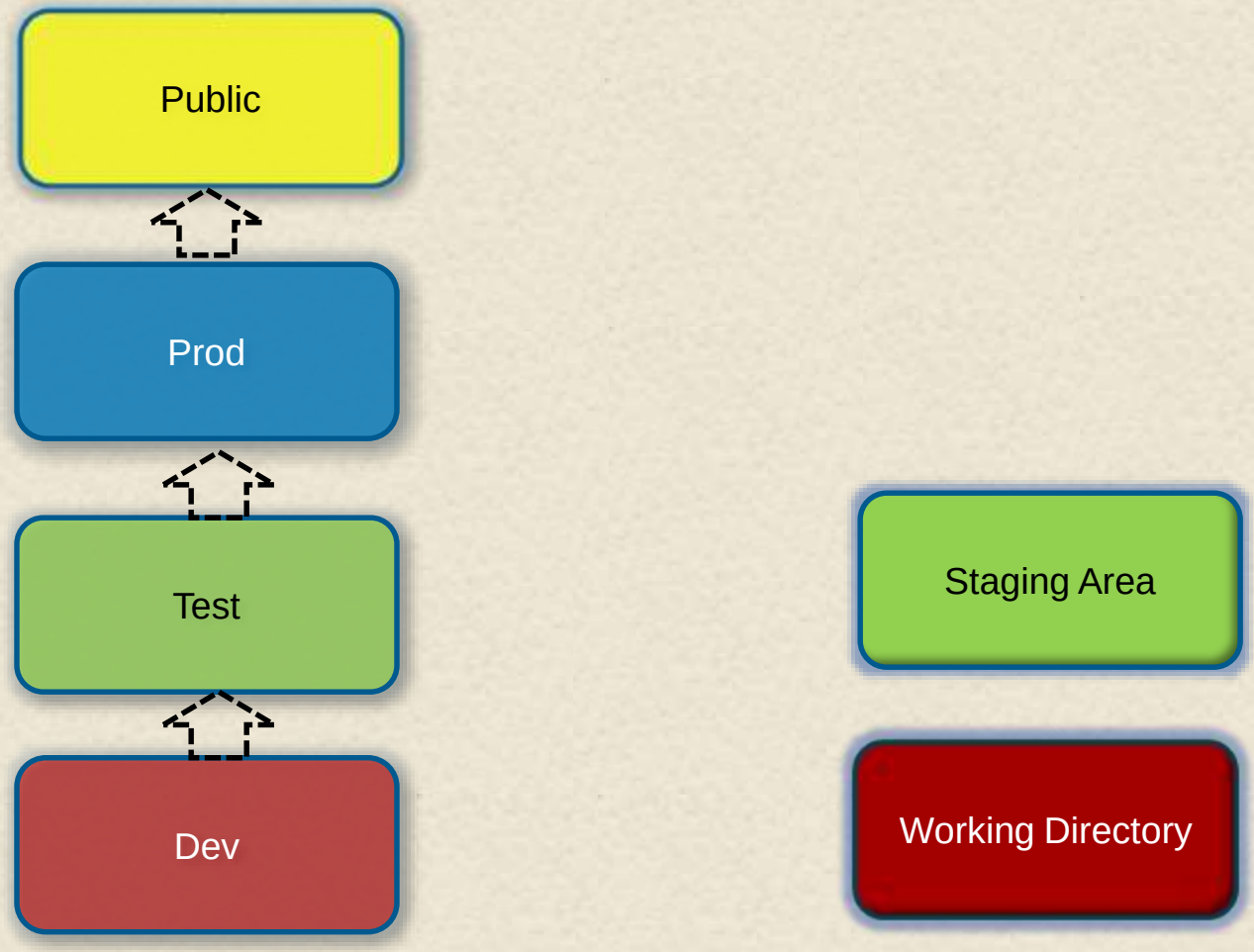
Git in One Picture



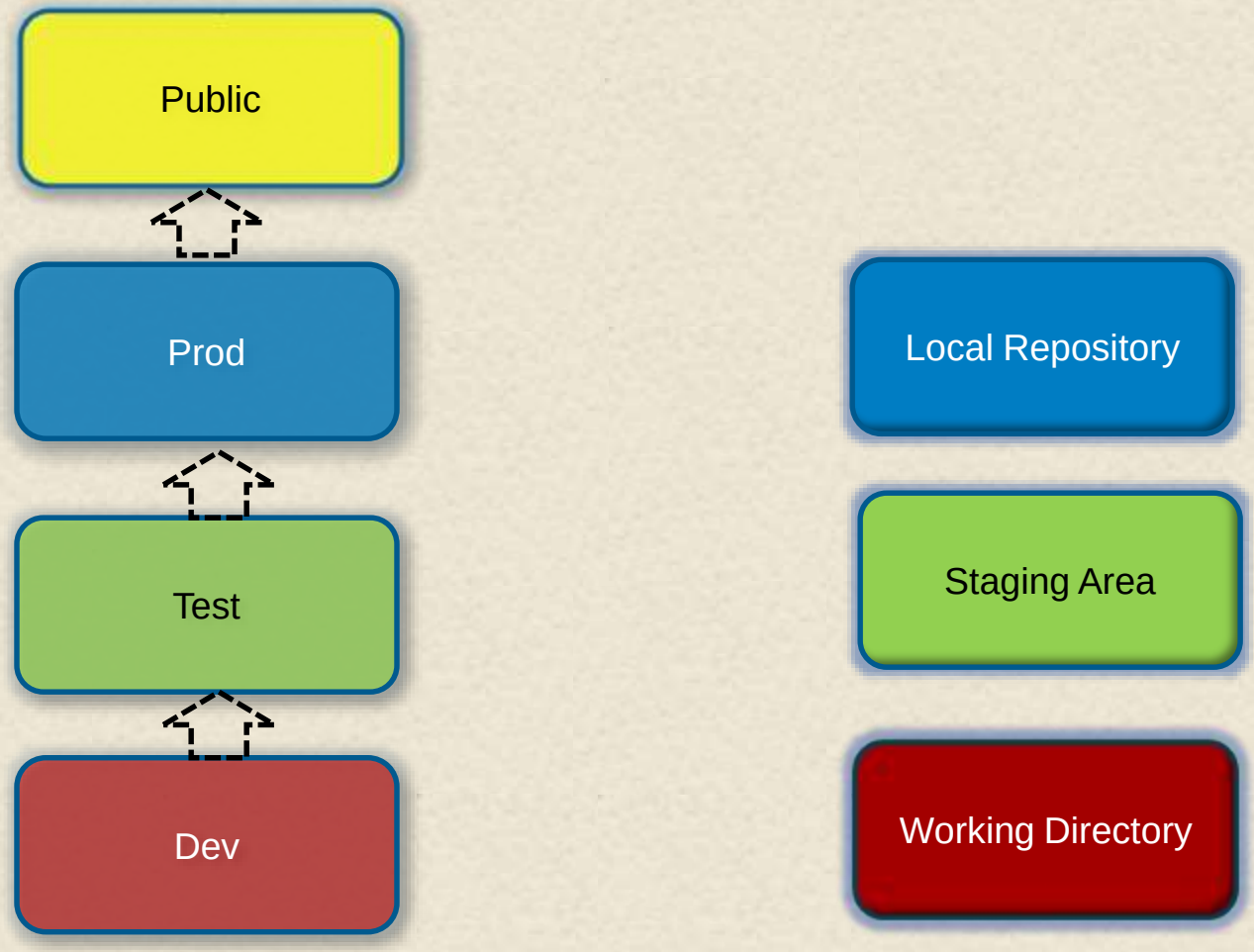
Git in One Picture



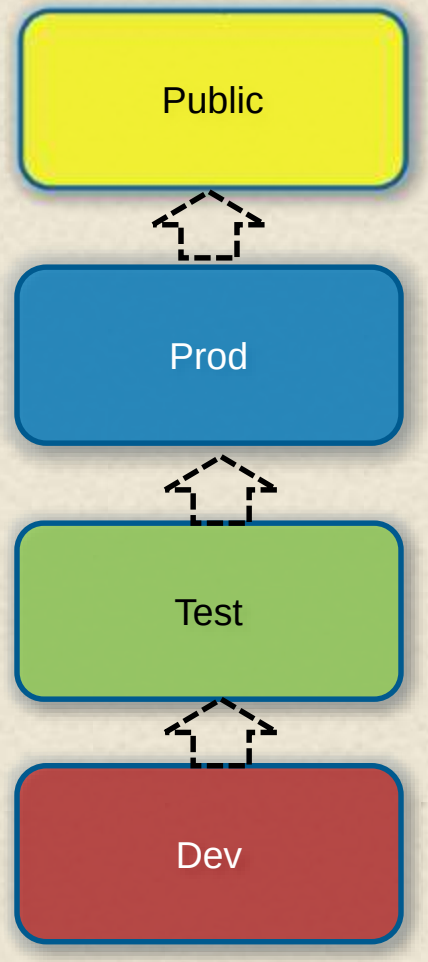
Git in One Picture



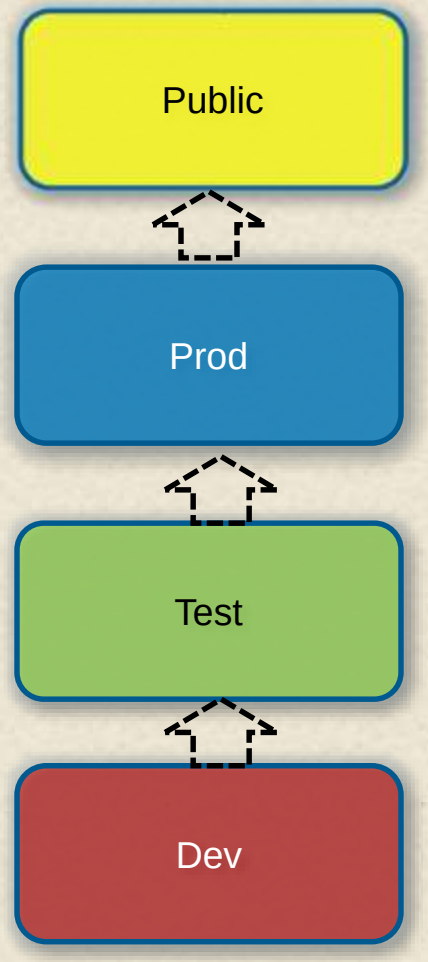
Git in One Picture

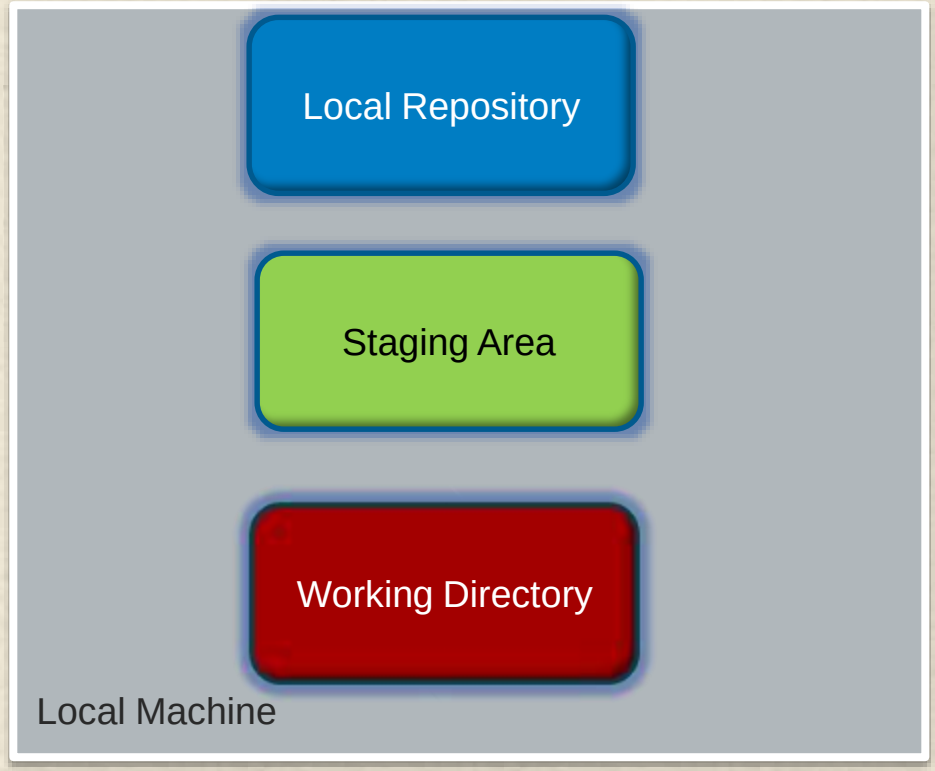


Git in One Picture

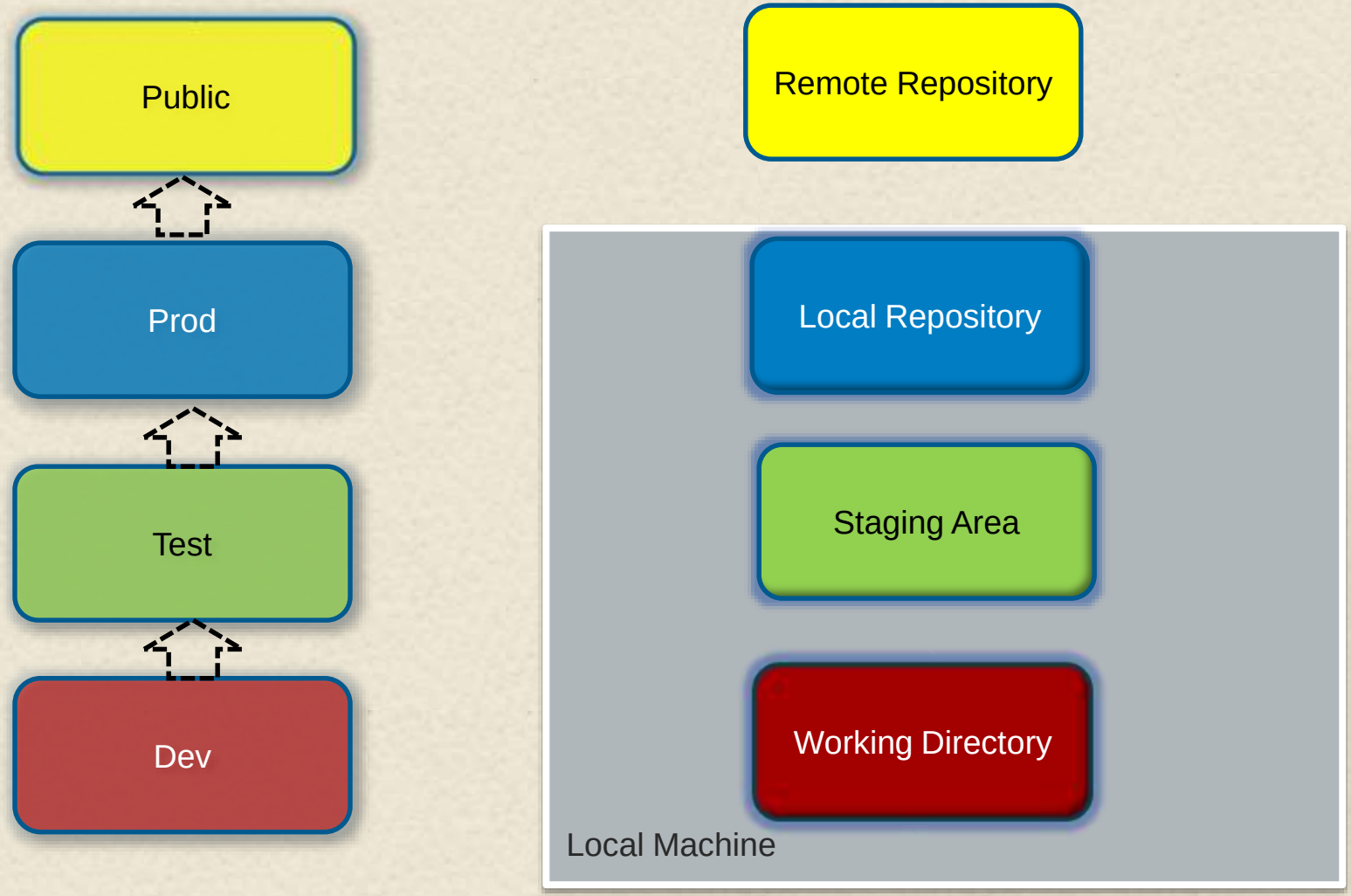


Git in One Picture

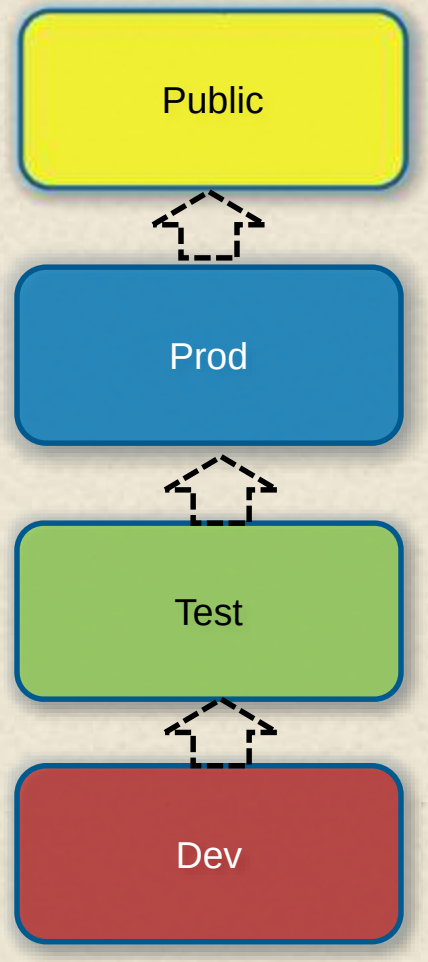




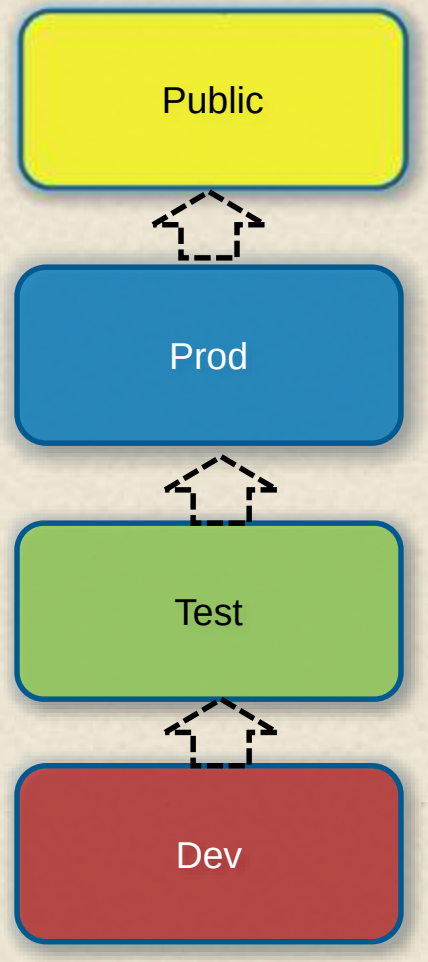
Git in One Picture



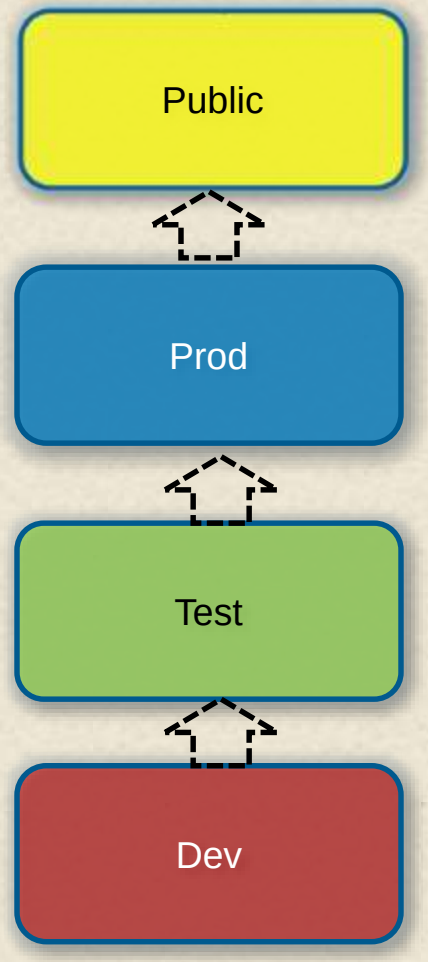
Git in One Picture

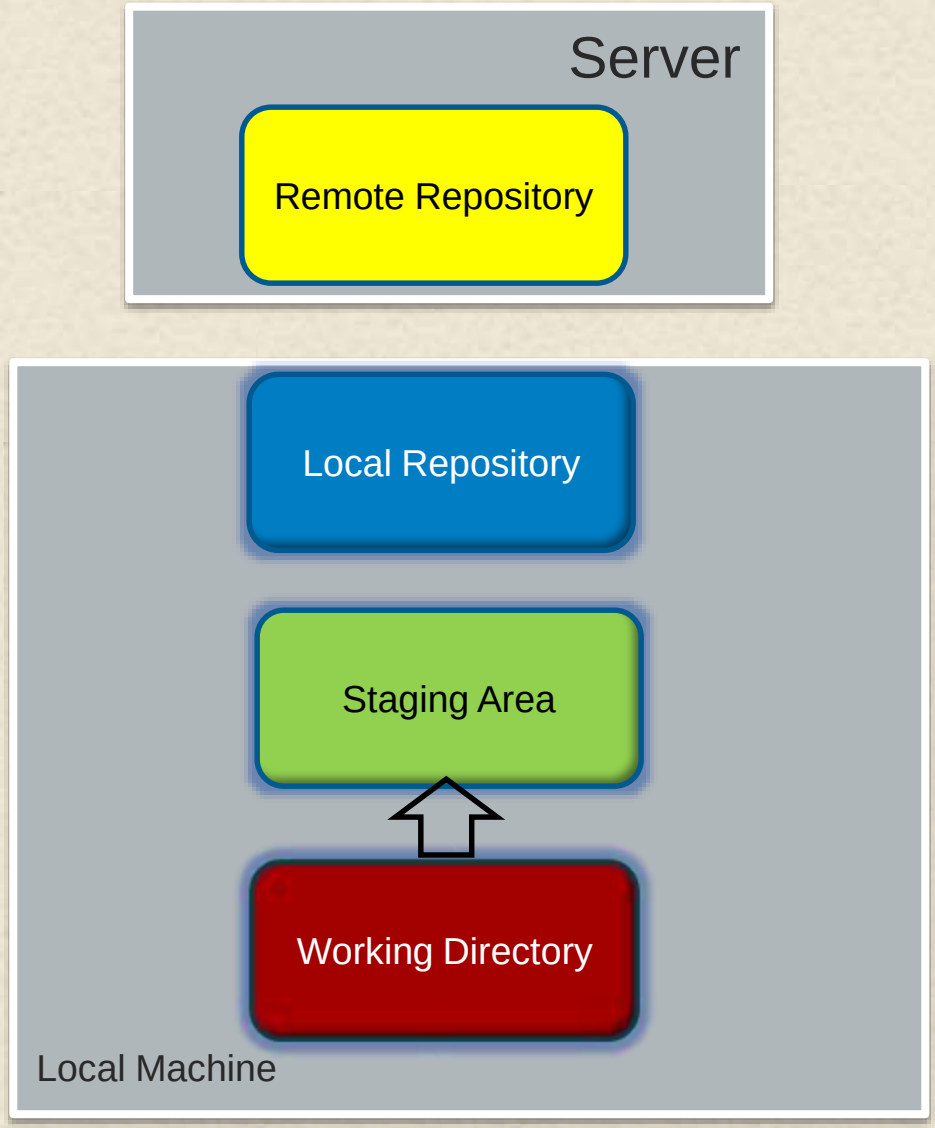


Git in One Picture

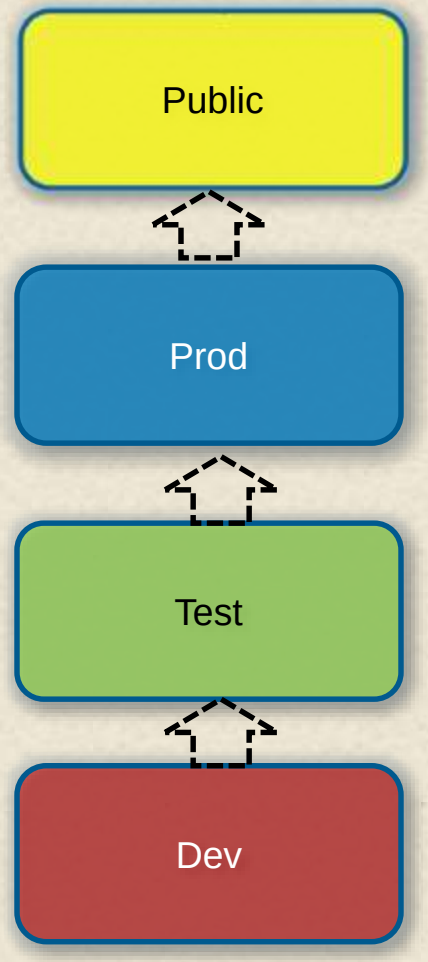


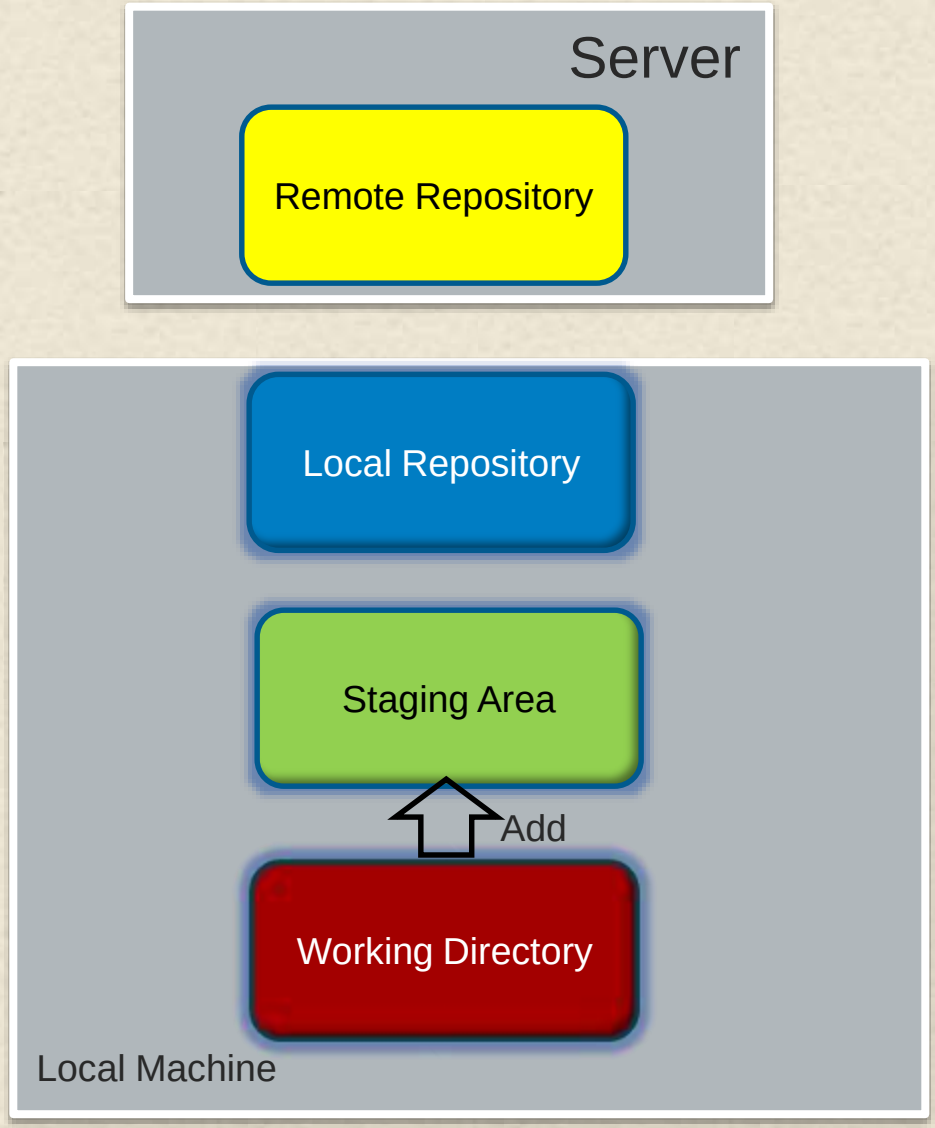
Git in One Picture



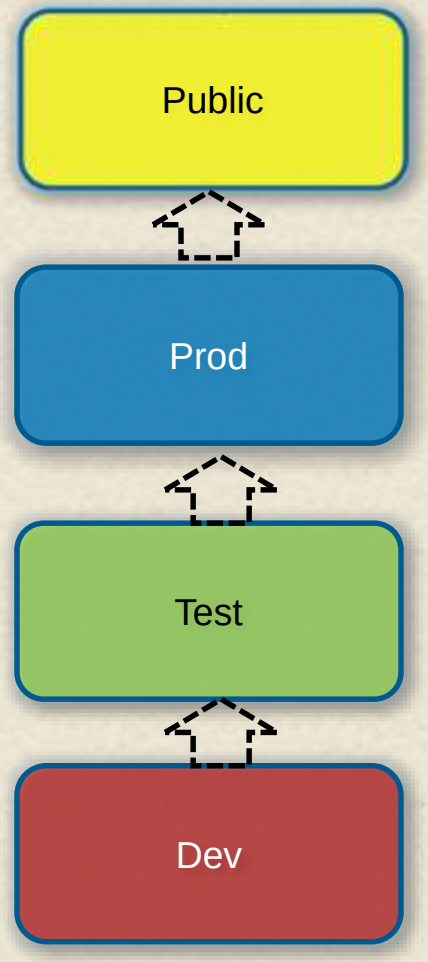


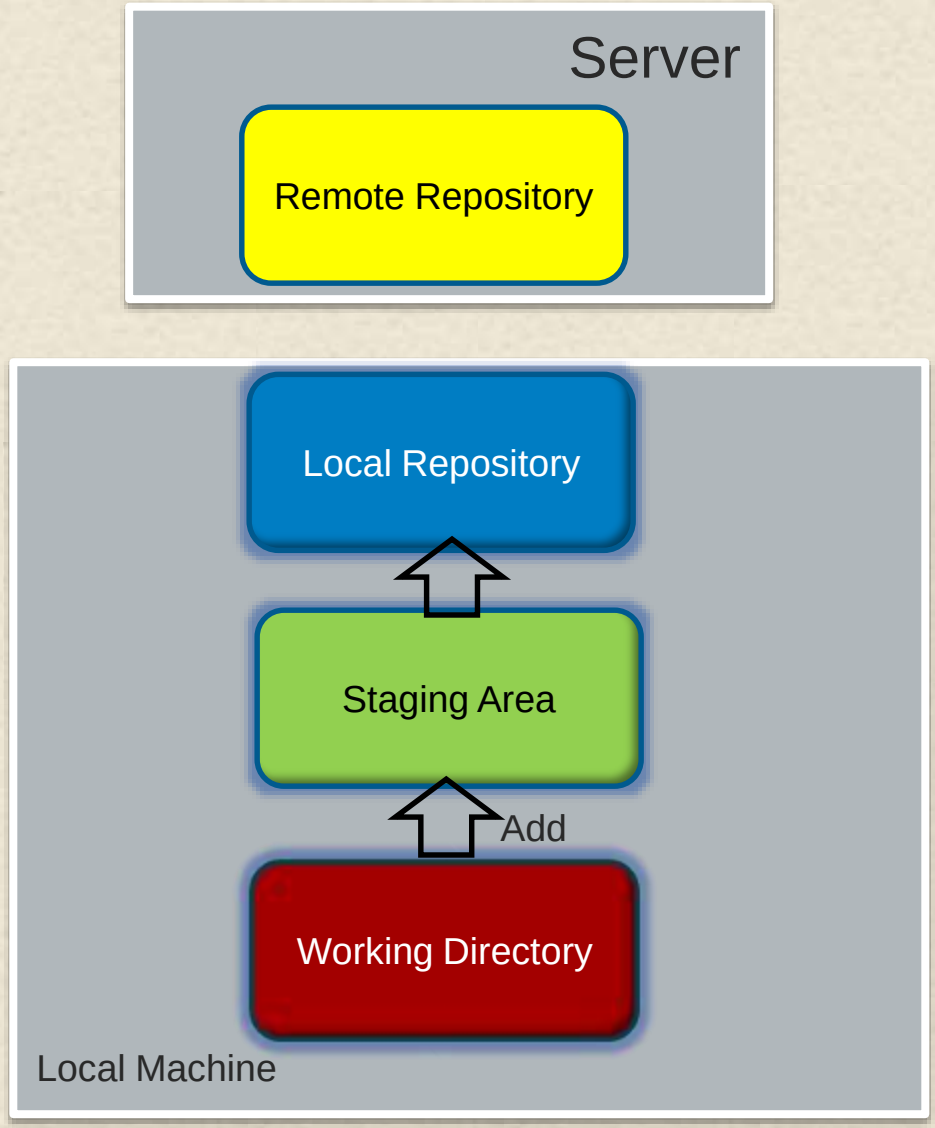
Git in One Picture



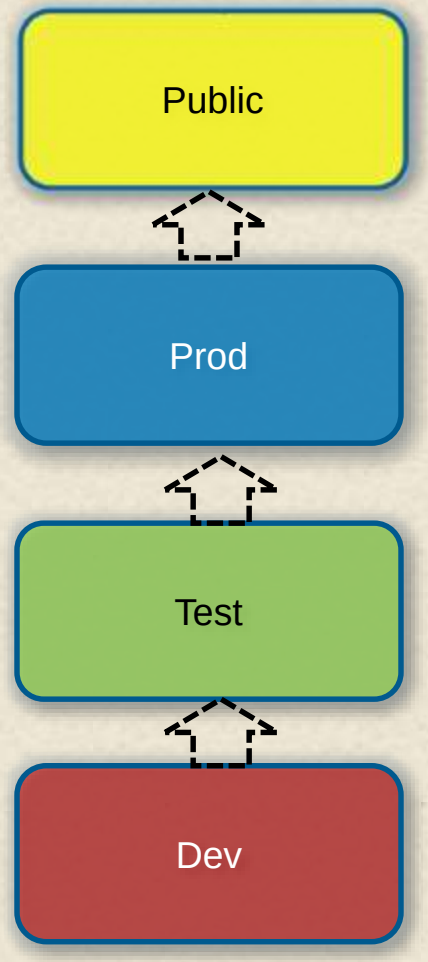


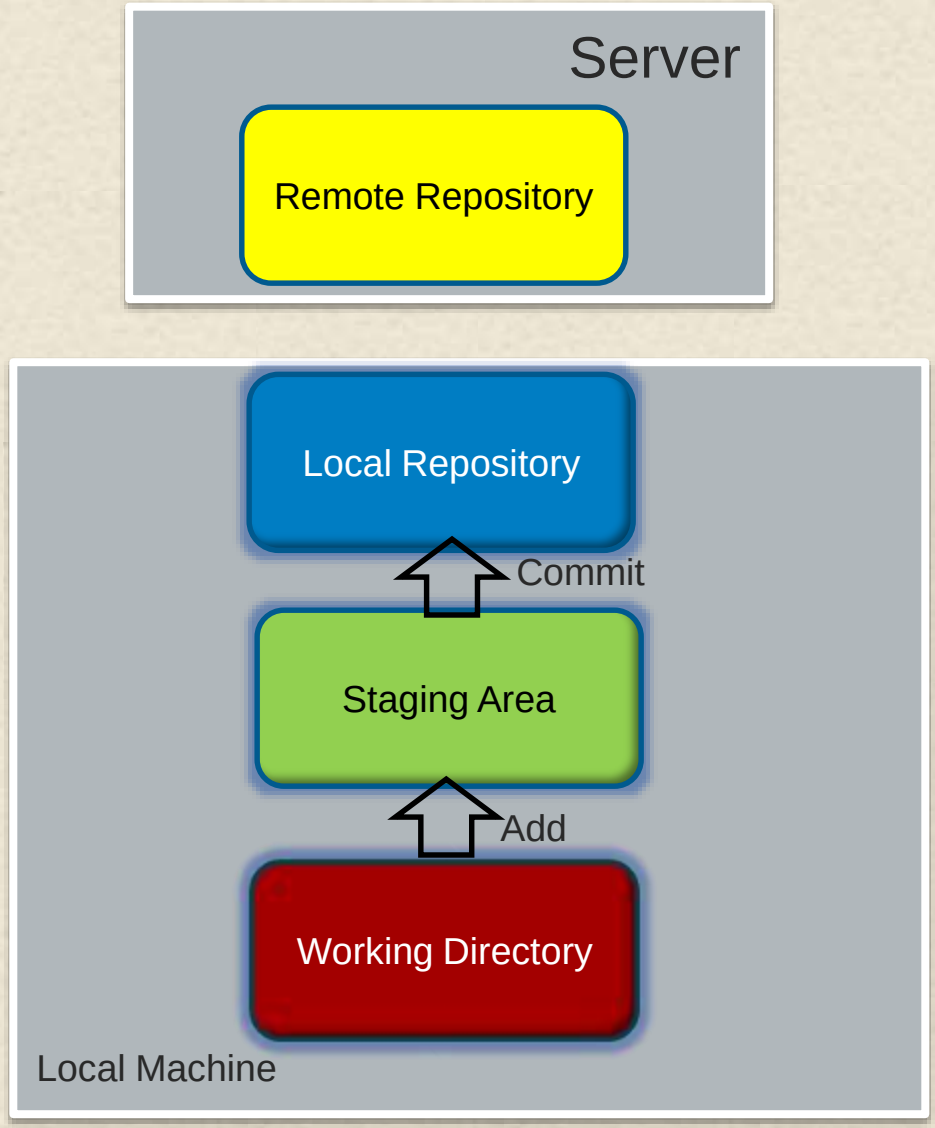
Git in One Picture



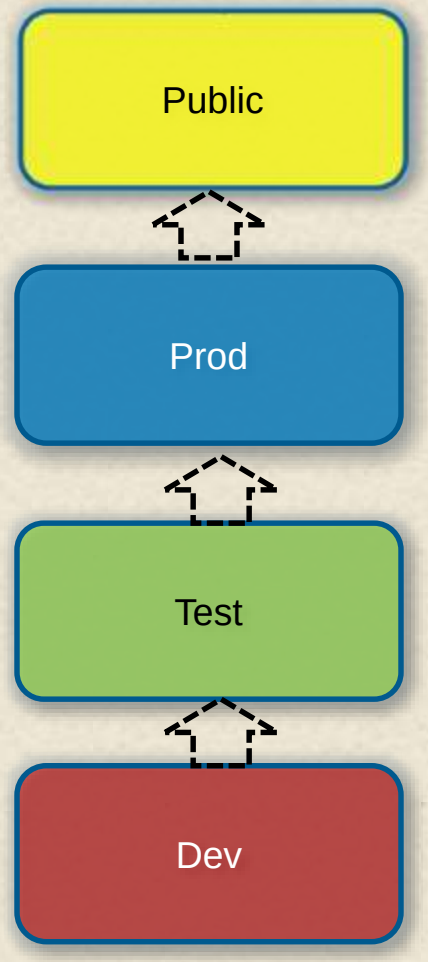


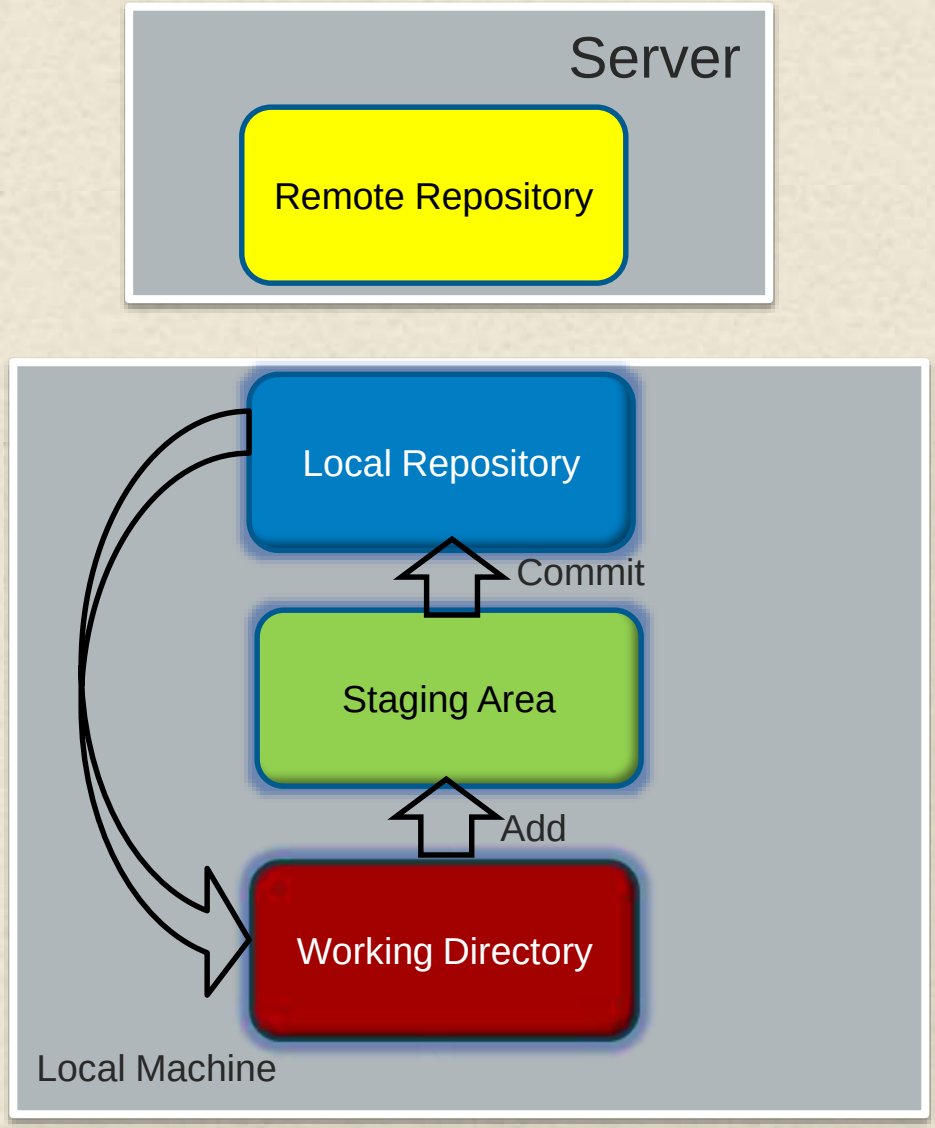
Git in One Picture



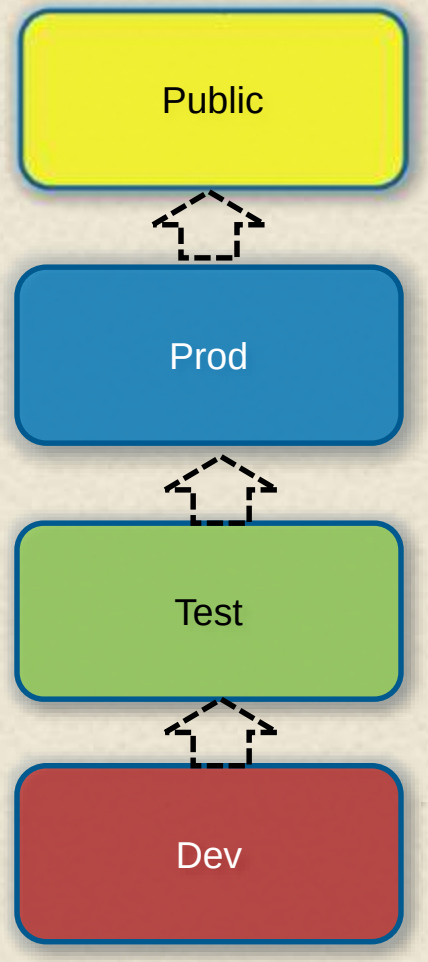


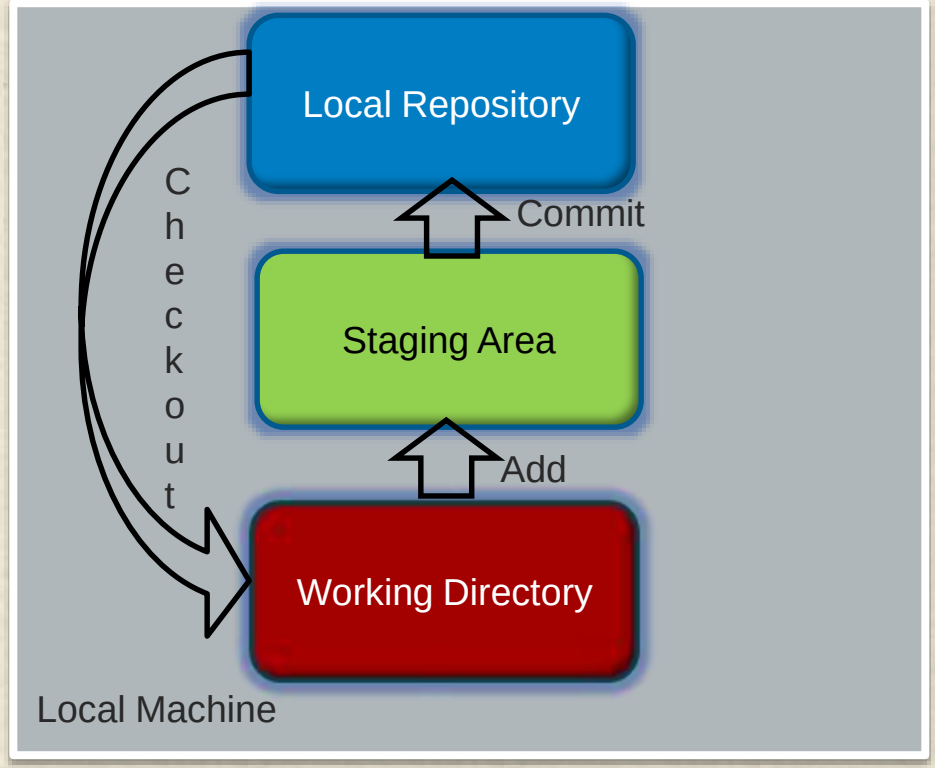
Git in One Picture



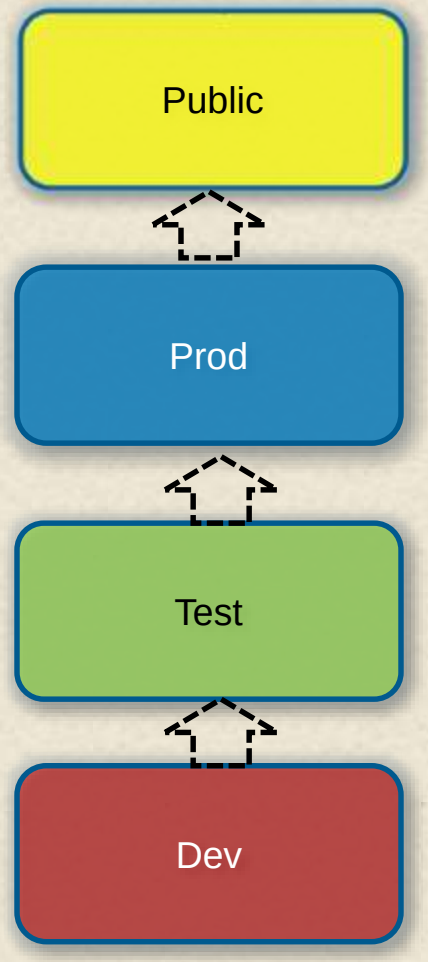


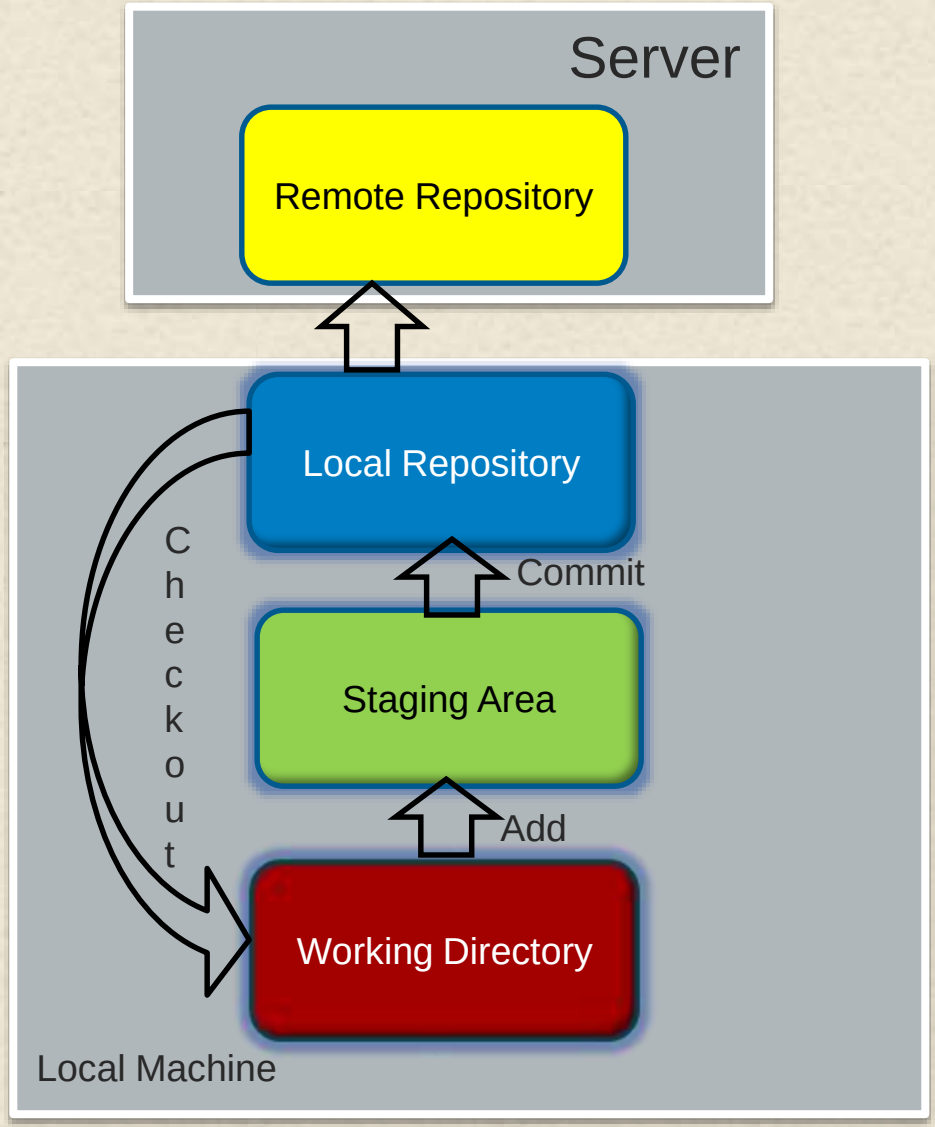
Git in One Picture



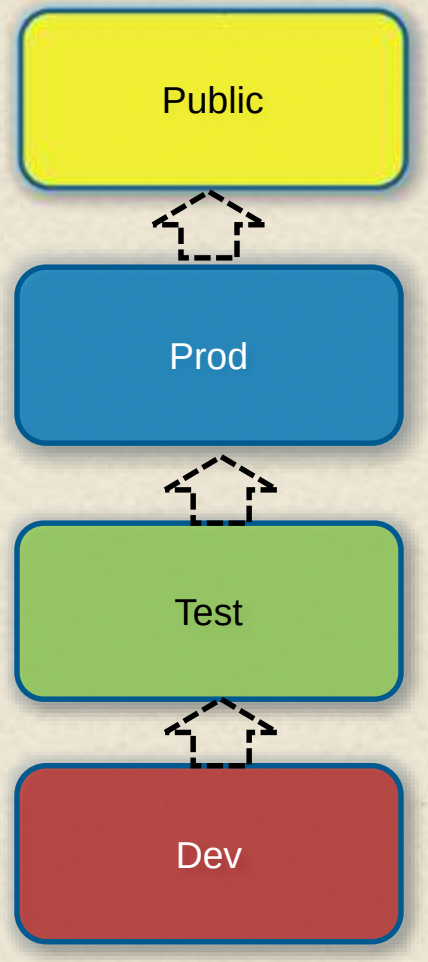


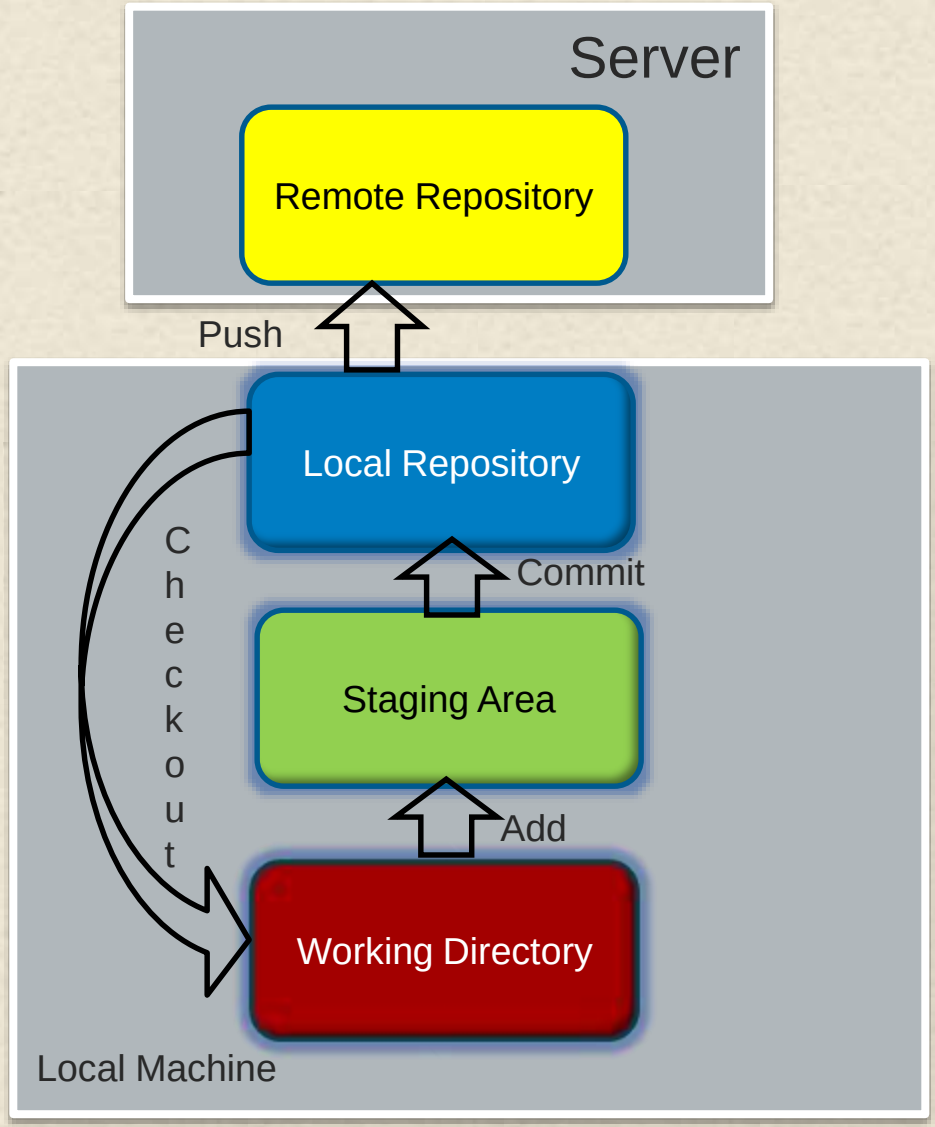
Git in One Picture



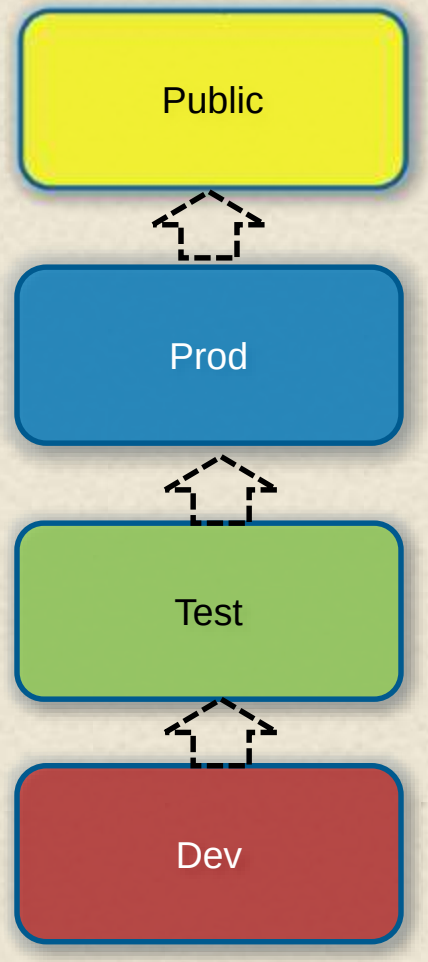


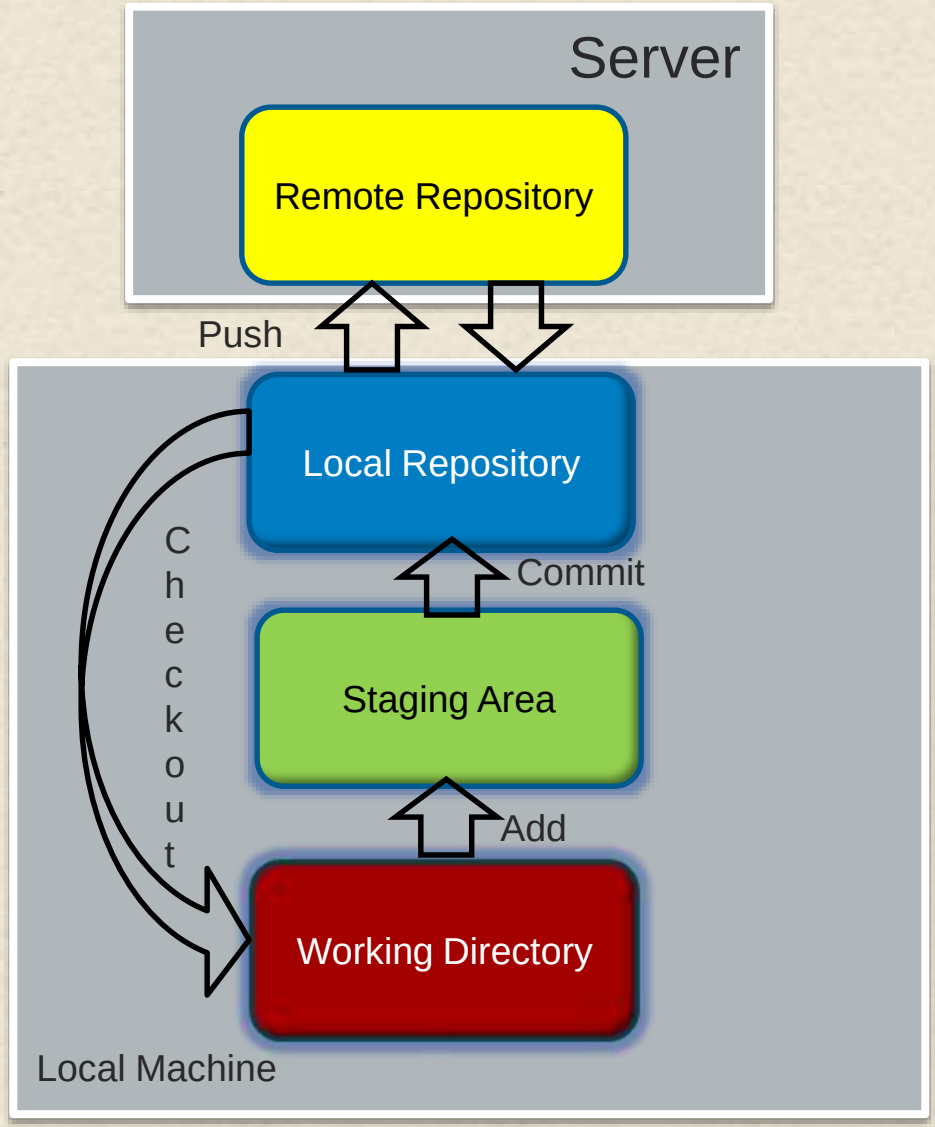
Git in One Picture



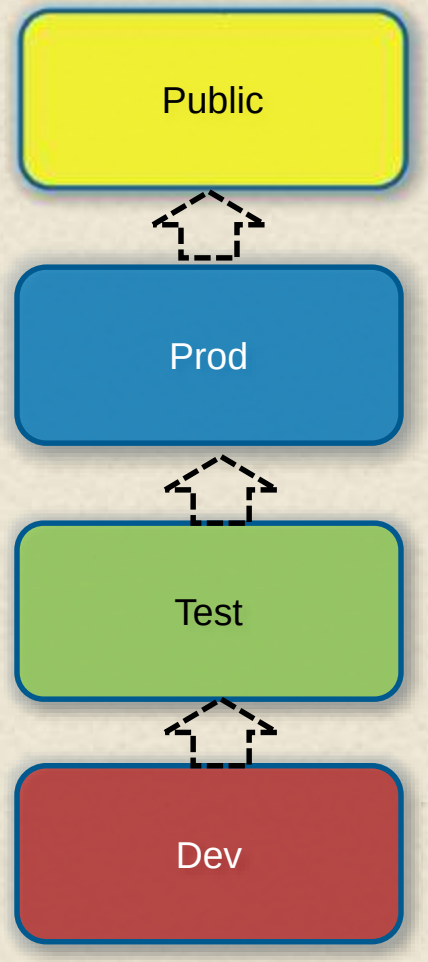


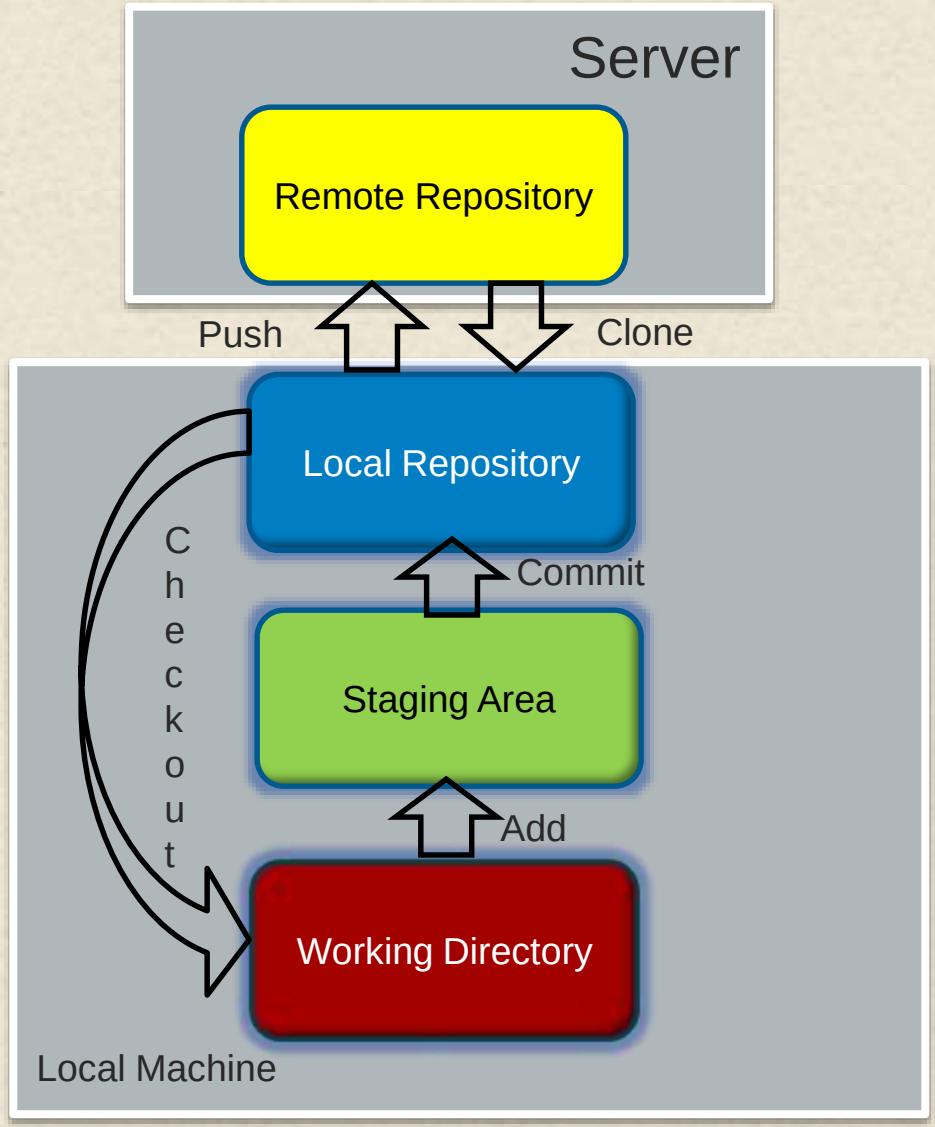
Git in One Picture



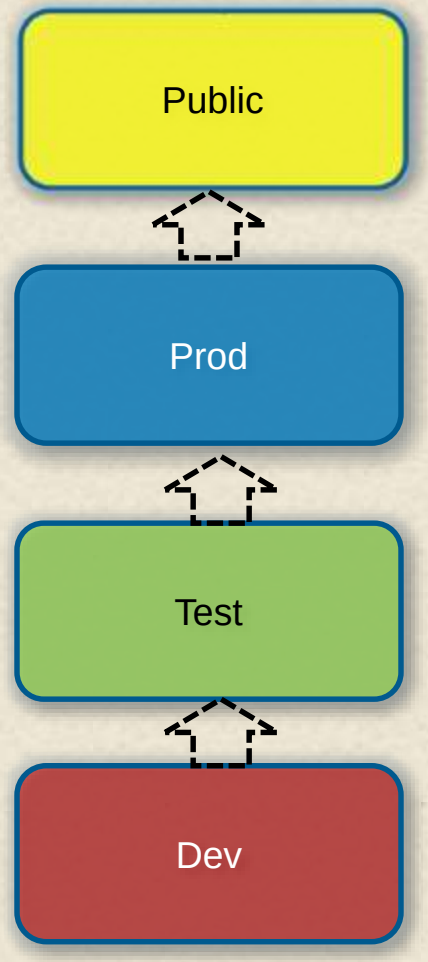


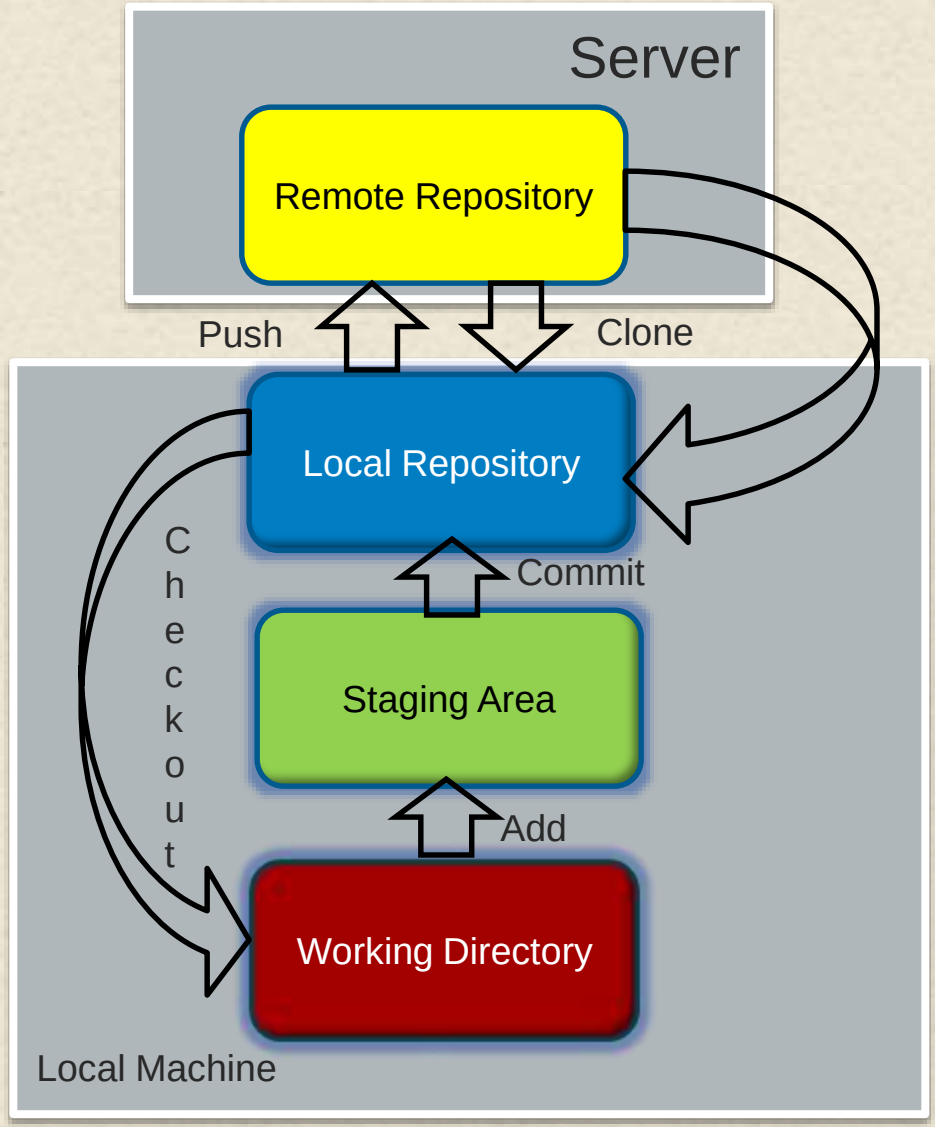
Git in One Picture



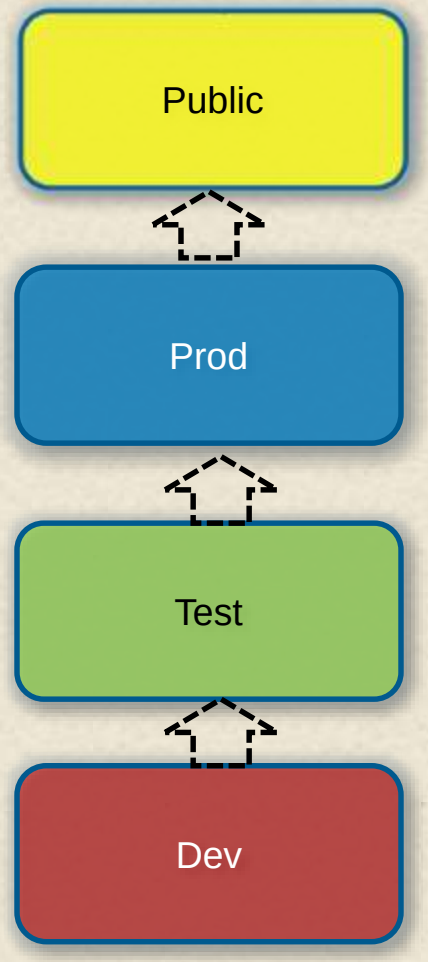


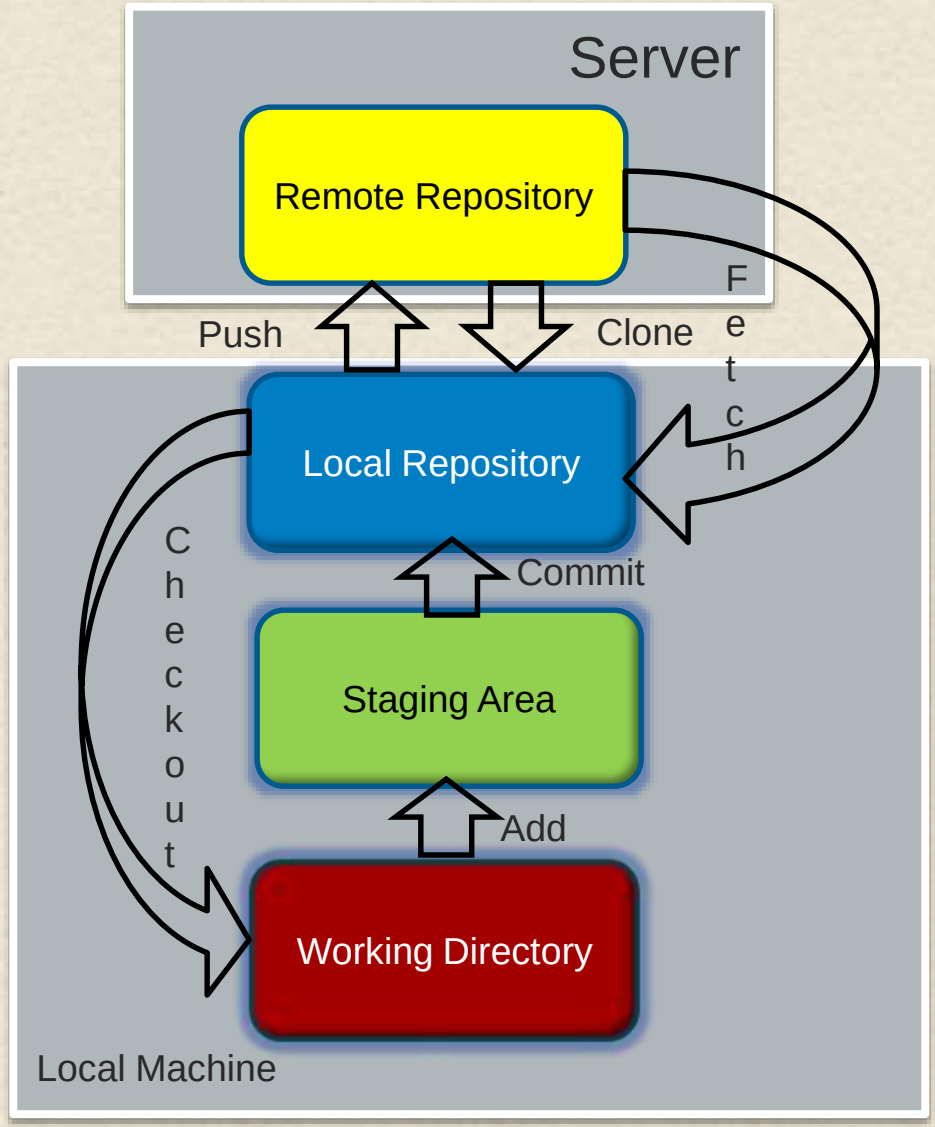
Git in One Picture



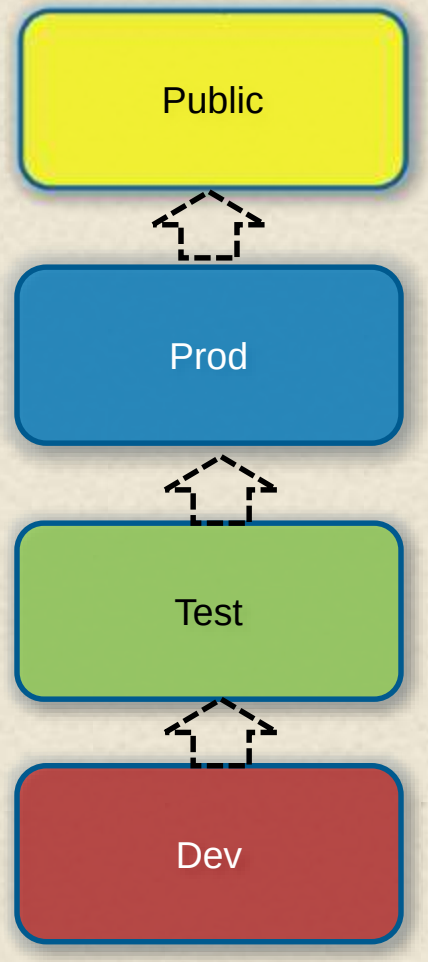


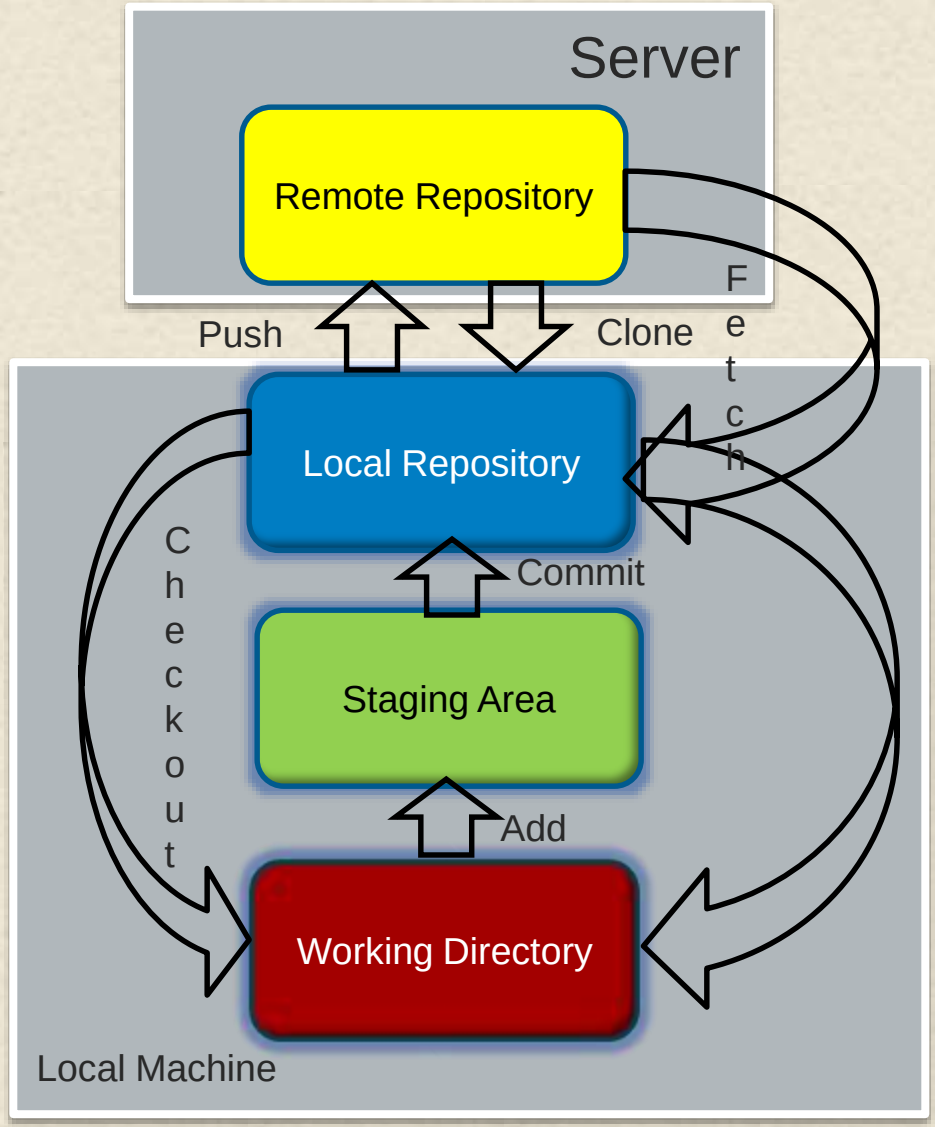
Git in One Picture



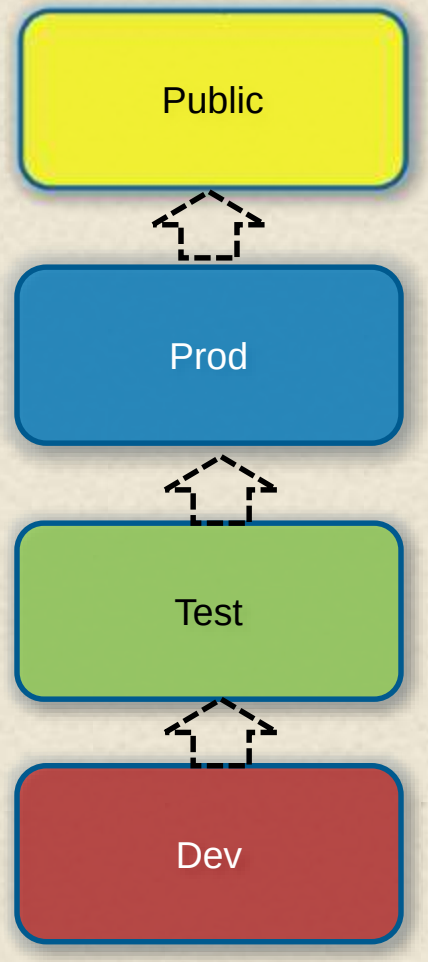


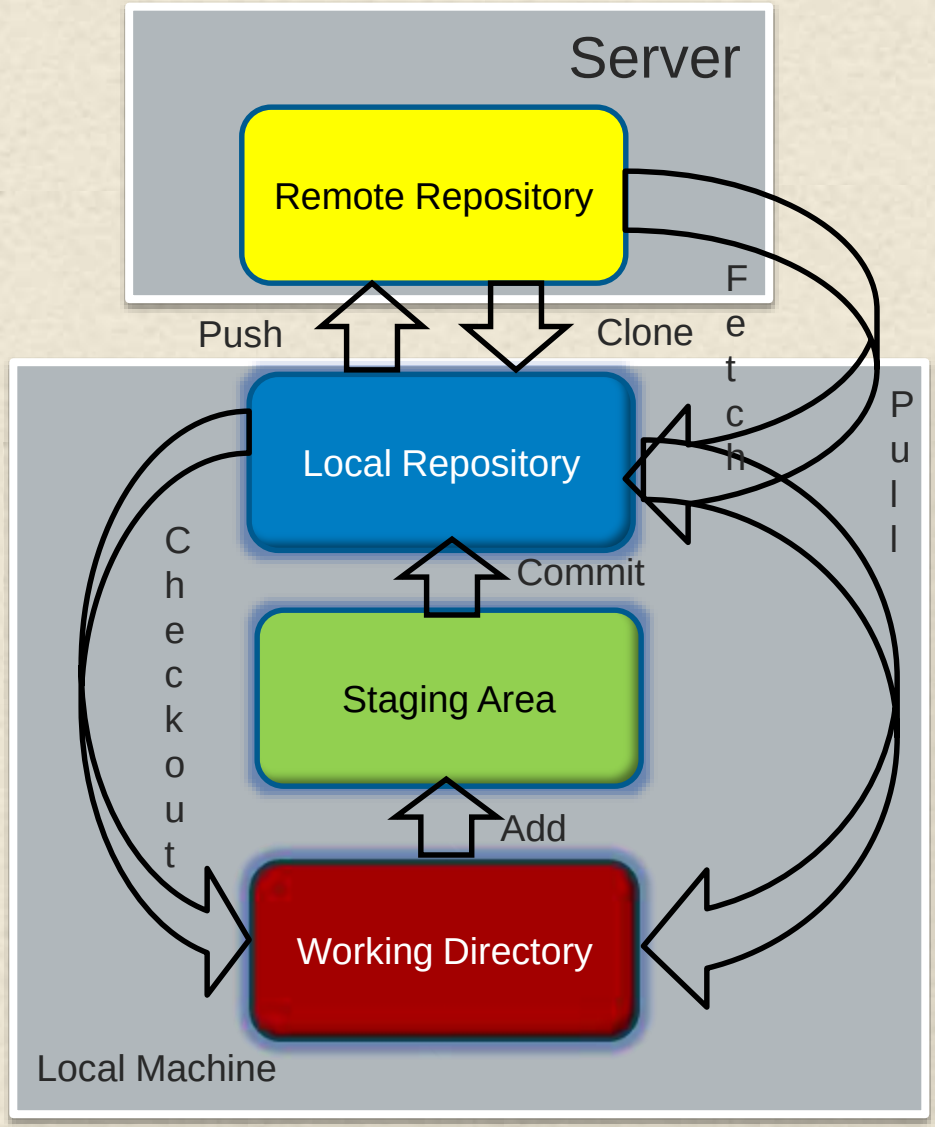
Git in One Picture





Git in One Picture





Git Granularity (What is a unit?)

Git Granularity (What is a unit?)

- In traditional source control, the unit of granularity is usually a file

Git Granularity (What is a unit?)

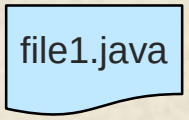
- In traditional source control, the unit of granularity is usually a file

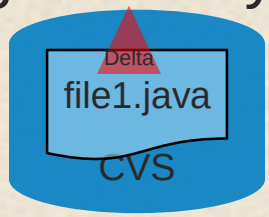
Git Granularity (What is a unit?)

- In traditional source control, the unit of granularity is usually a file

Git Granularity (What is a unit?)

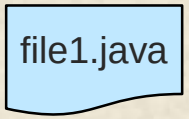
- In traditional source control, the unit of granularity is usually a file

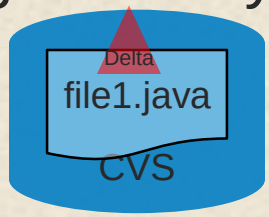




Git Granularity (What is a unit?)

- In traditional source control, the unit of granularity is usually a file

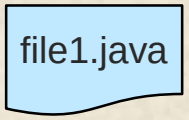


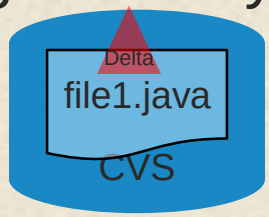


- In Git, the unit of granularity is usually a tree

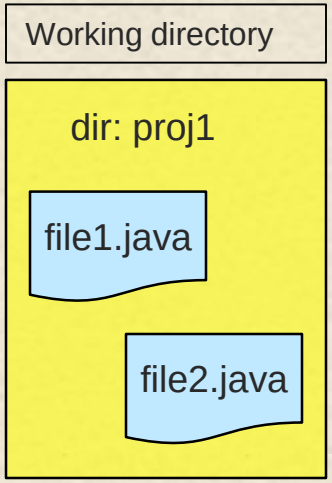
Git Granularity (What is a unit?)

- In traditional source control, the unit of granularity is usually a file



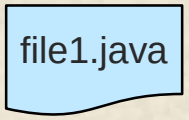


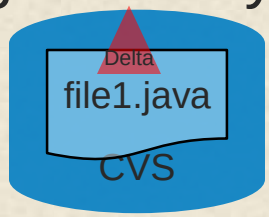
- In Git, the unit of granularity is usually a tree



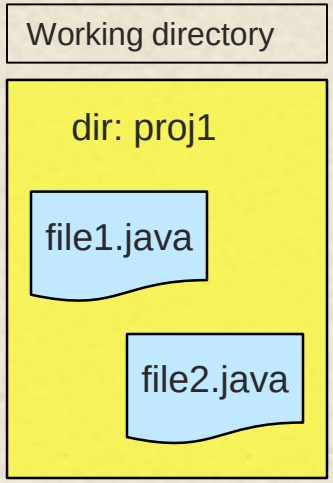
Git Granularity (What is a unit?)

- In traditional source control, the unit of granularity is usually a file



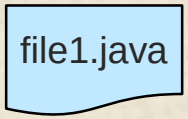


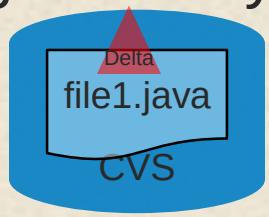
- In Git, the unit of granularity is usually a tree



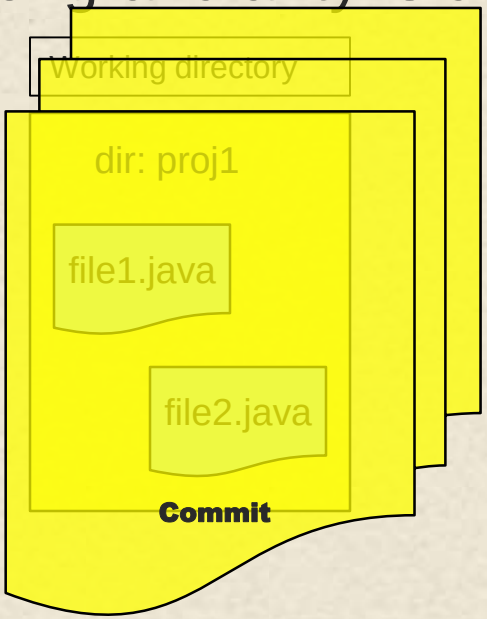
Git Granularity (What is a unit?)

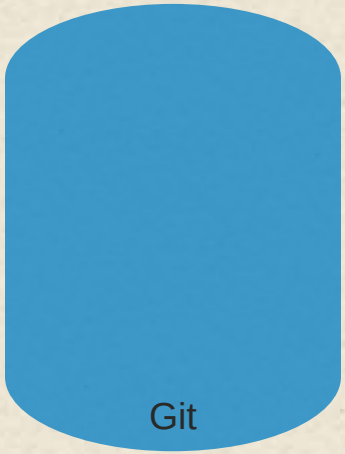
- In traditional source control, the unit of granularity is usually a file





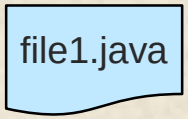
- In Git, the unit of granularity is usually a tree

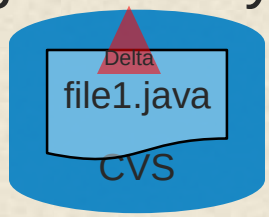




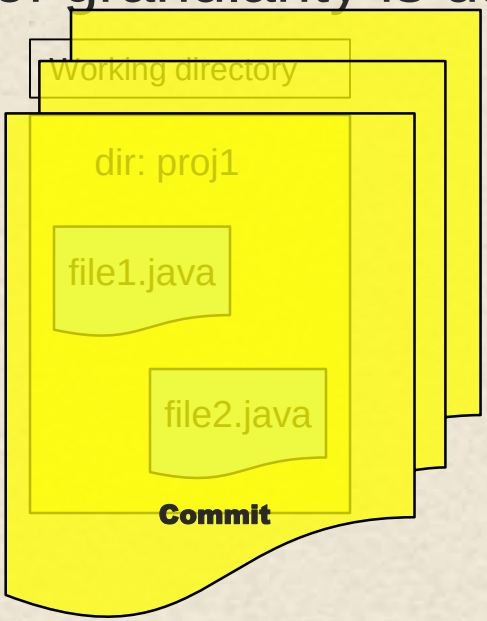
Git Granularity (What is a unit?)

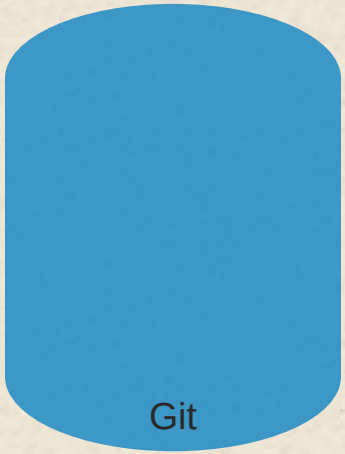
- In traditional source control, the unit of granularity is usually a file





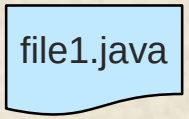
- In Git, the unit of granularity is usually a tree

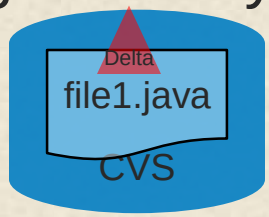




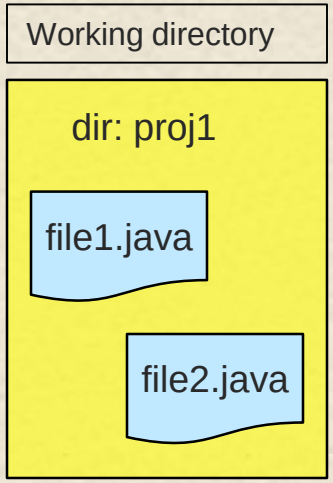
Git Granularity (What is a unit?)

- In traditional source control, the unit of granularity is usually a file



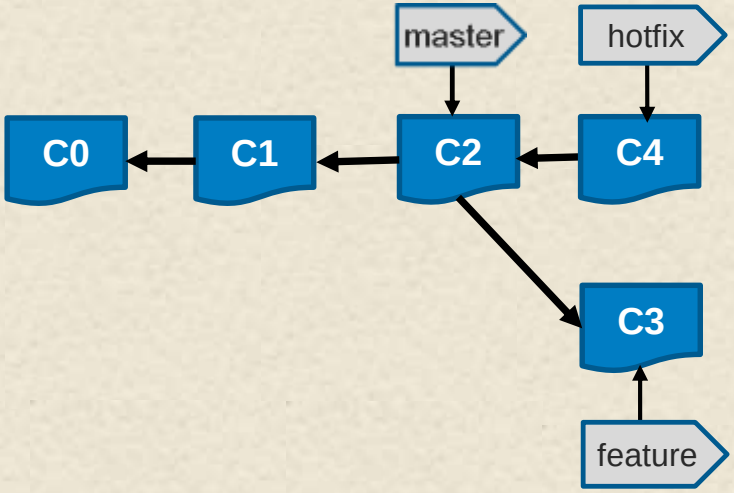


- In Git, the unit of granularity is usually a tree



Merging: What is a Fast-forward?

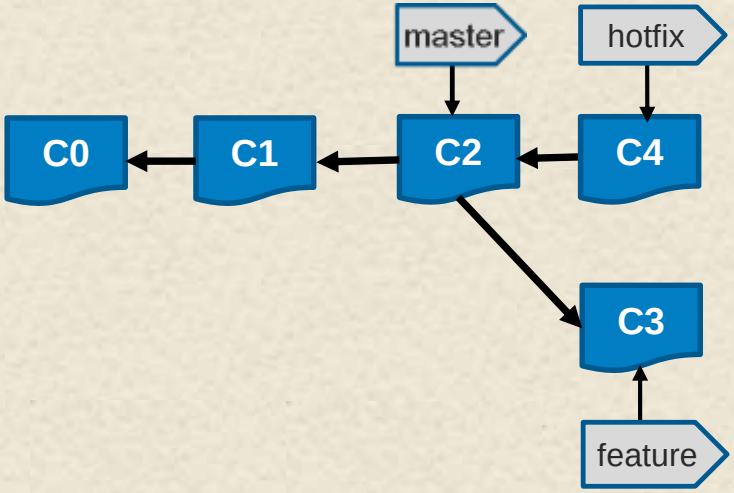
- Assume you have three branches as below
- You want to merge hotfix into master (so master will have your hotfix for future development)



Merging: What is a Fast-forward?

- Assume you have three branches as below
- You want to merge hotfix into master (so master will have your hotfix for future development)

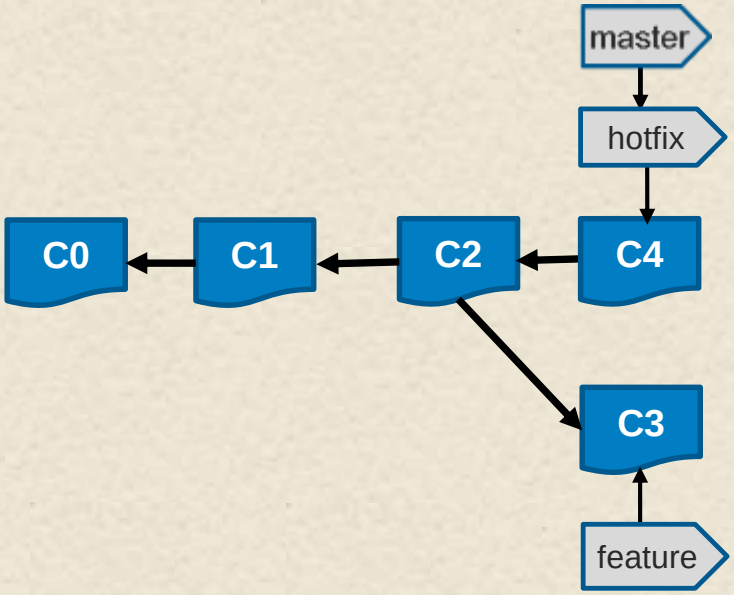
$ git checkout master



Merging: What is a Fast-forward?

- Assume you have three branches as below
- You want to merge hotfix into master (so master will have your hotfix for future development)

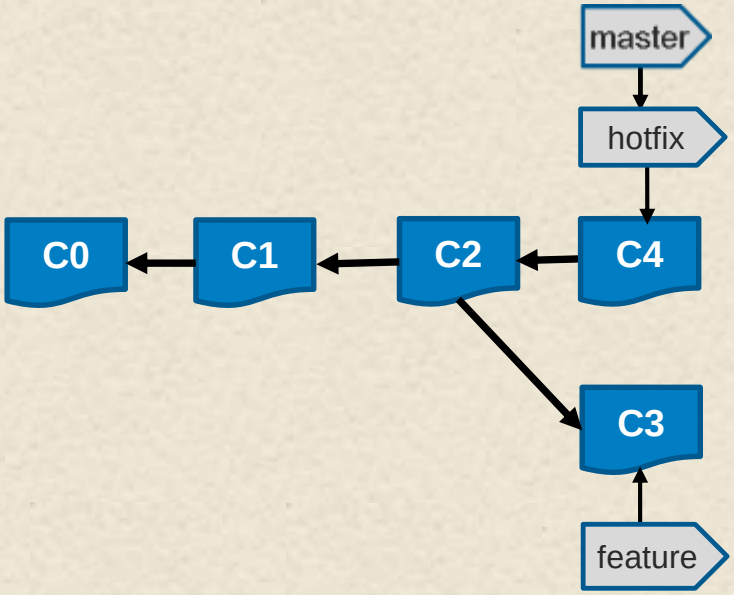
```
$ git checkout master
$ git merge hotfix
```



Merging: What is a Fast-forward?

- Assume you have three branches as below
- You want to merge hotfix into master (so master will have your hotfix for future development)

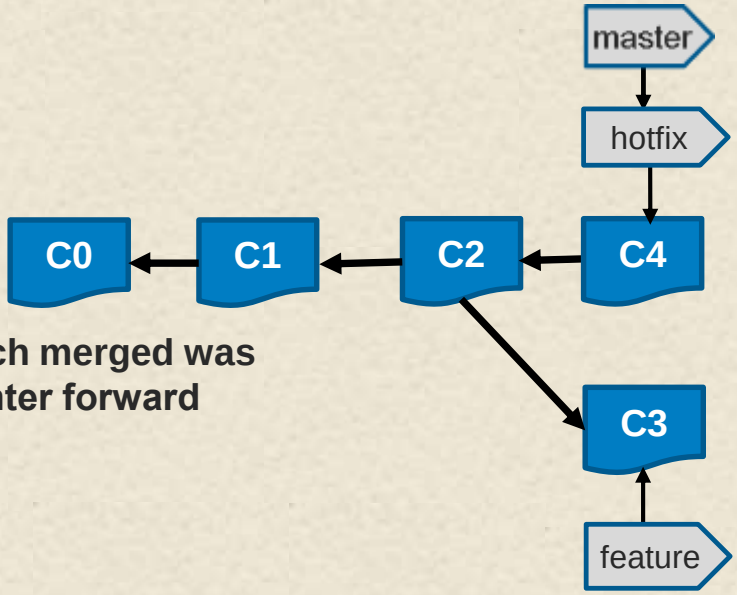
```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast Forward
README  |  1-
1 files changed, 0 insertions(+) 1 deletions (-)
```



Merging: What is a Fast-forward?

- Assume you have three branches as below
- You want to merge hotfix into master (so master will have your hotfix for future development)

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast Forward
README  |  1-
1 files changed, 0 insertions(+) 1 deletions (-)
```

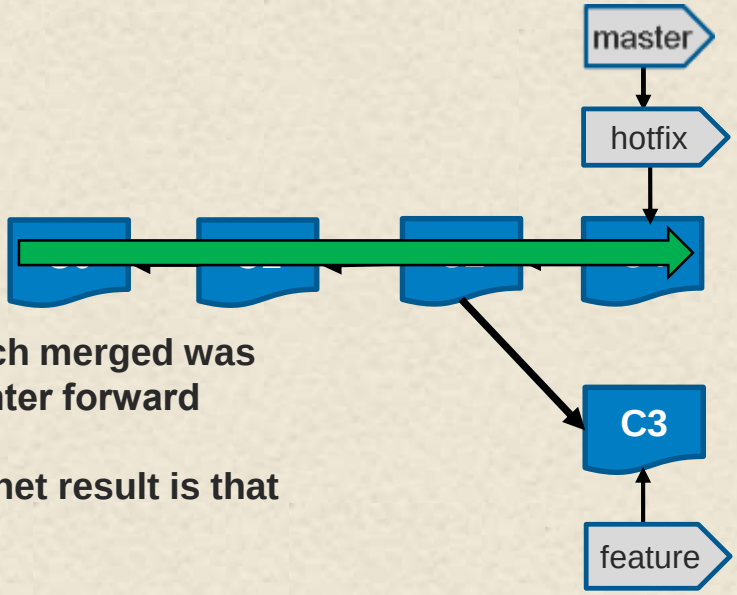


About "Fast Forward" – because commit pointed to by branch merged was directly "upstream" of the current commit, Git moves the pointer forward

Merging: What is a Fast-forward?

- Assume you have three branches as below
- You want to merge hotfix into master (so master will have your hotfix for future development)

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast Forward
README  |  1-
1 files changed, 0 insertions(+) 1 deletions (-)
```

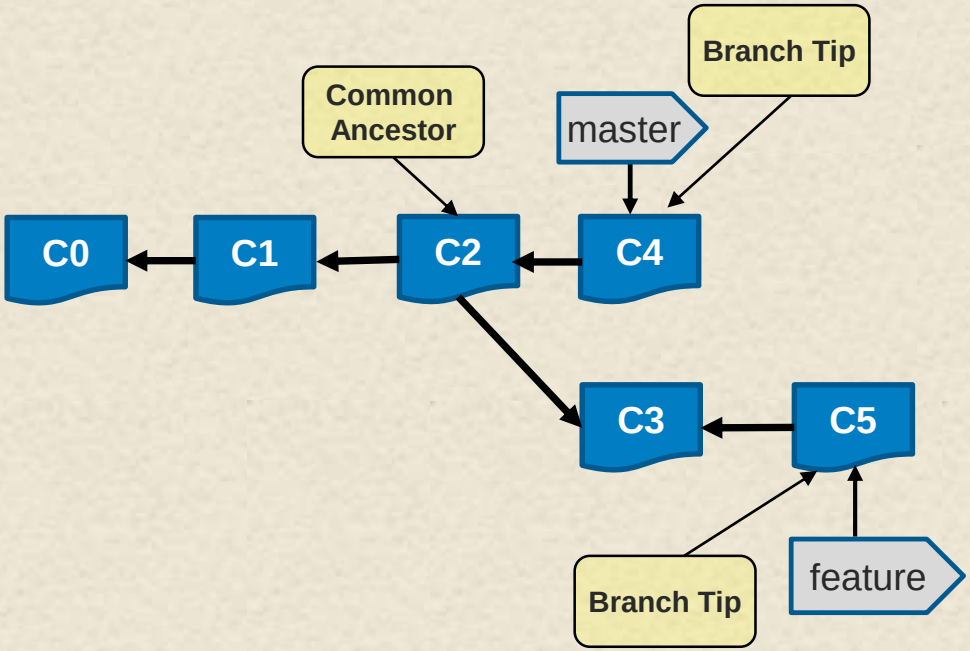


About "Fast Forward" – because commit pointed to by branch merged was directly "upstream" of the current commit, Git moves the pointer forward

(Both branches were in the same line of development, so the net result is that master and hotfix point to the same commit)

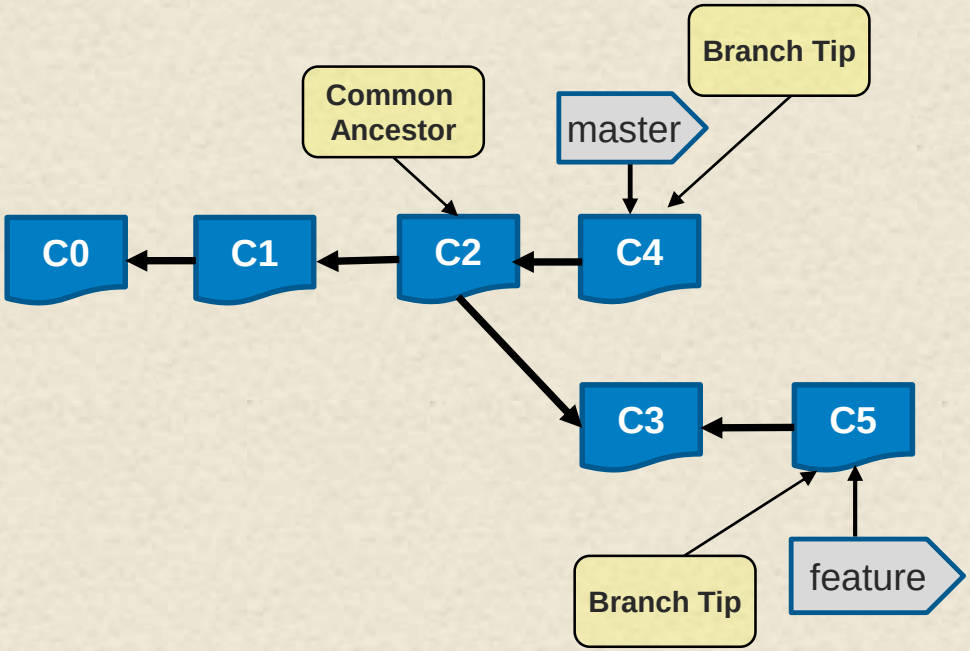
Merging: What is a 3-way Merge?

- **Assume branching scenario below**



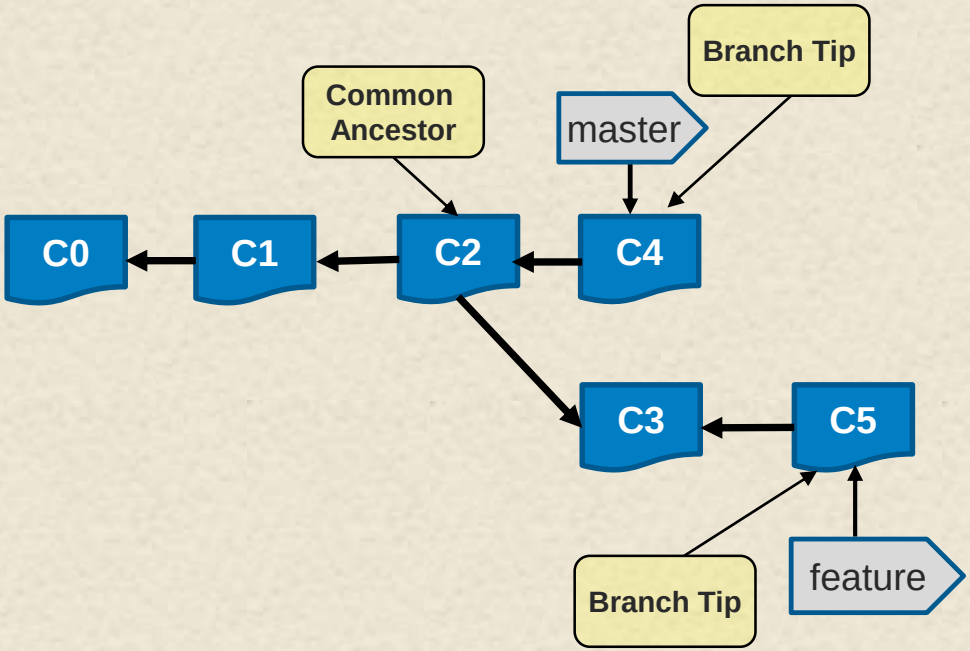
Merging: What is a 3-way Merge?

- **Assume branching scenario below**
  - master and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)



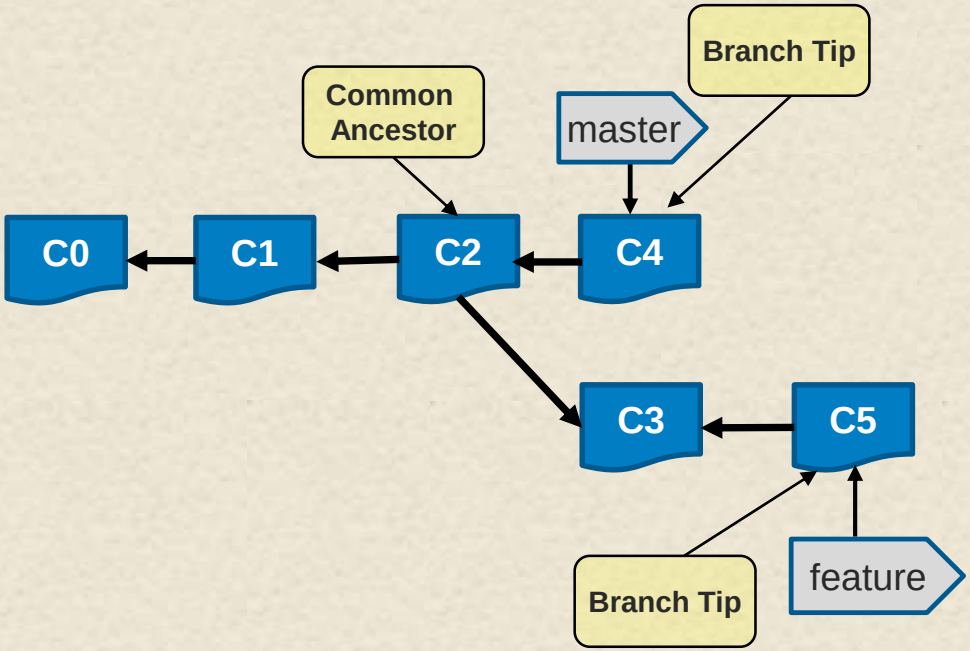
Merging: What is a 3-way Merge?

- **Assume branching scenario below**
  - master and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)
- **Intent is to change to master and merge in feature**



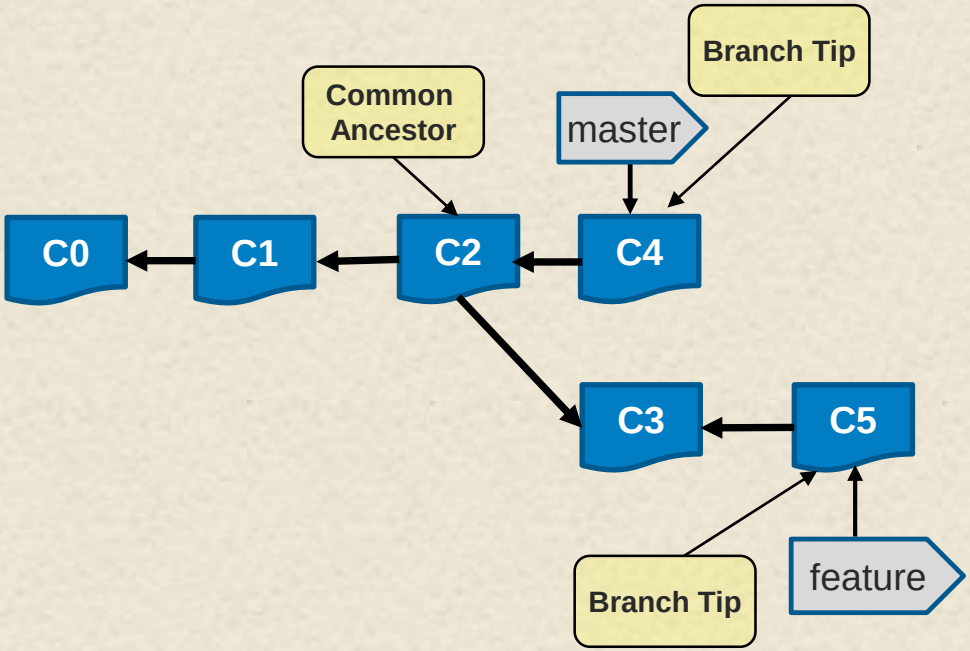
Merging: What is a 3-way Merge?

- **Assume branching scenario below**
  - master and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)
- **Intent is to change to master and merge in feature**
- **Current commit on target branch isn't a direct ancestor of current commit on branch you're merging in (i.e. C4 isn't on the same line of development as C5)**



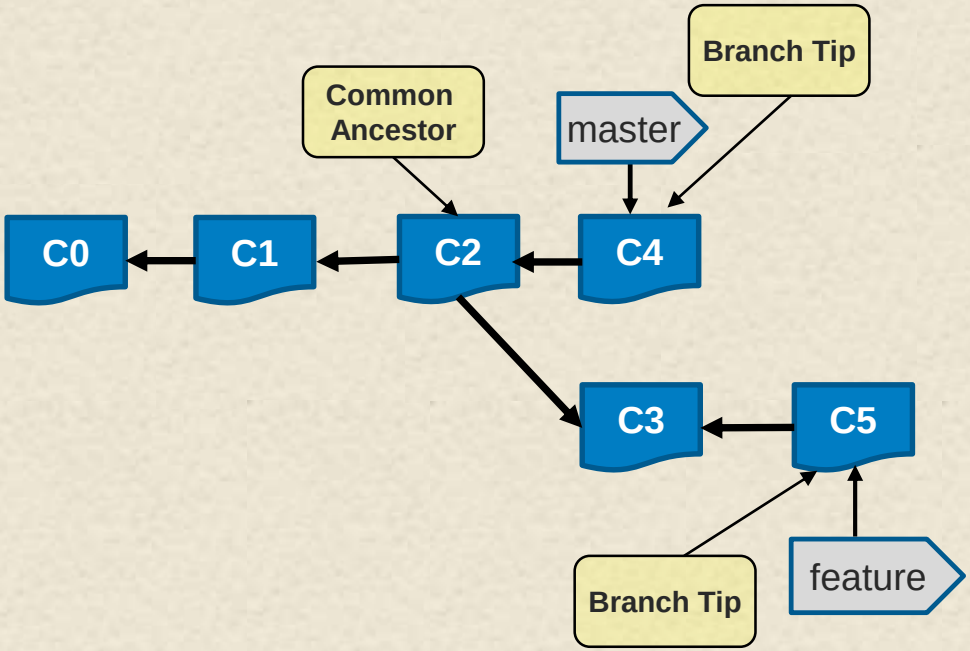
Merging: What is a 3-way Merge?

- **Assume branching scenario below**
  - master and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)
- **Intent is to change to master and merge in feature**
- **Current commit on target branch isn't a direct ancestor of current commit on branch you're merging in (i.e. C4 isn't on the same line of development as C5)**
- **Git does 3-way merge using common ancestor**



Merging: What is a 3-way Merge?

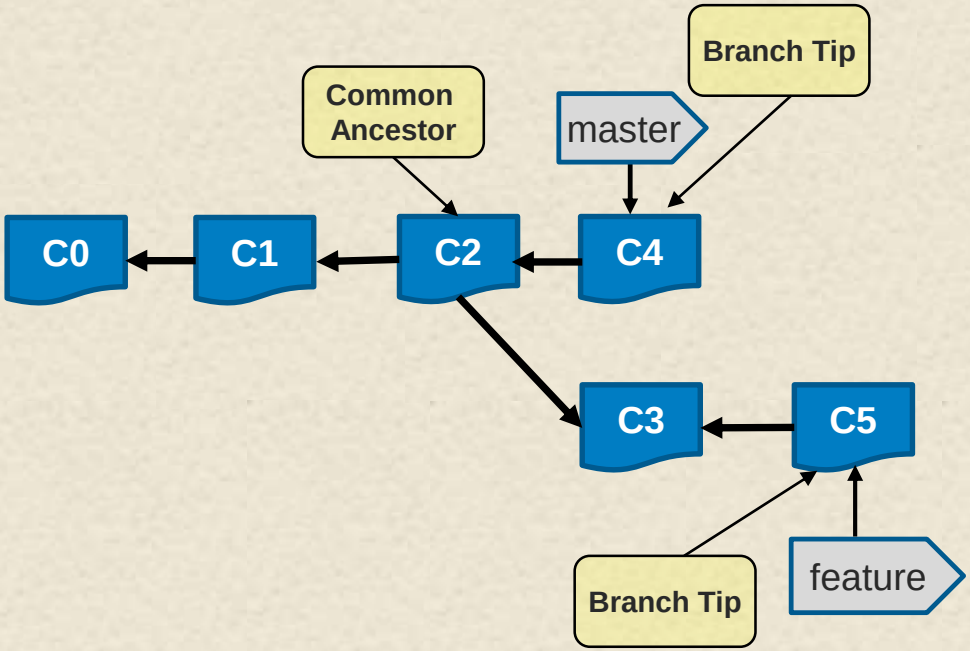
- **Assume branching scenario below**
  - master and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)
- **Intent is to change to master and merge in feature**
- **Current commit on target branch isn't a direct ancestor of current commit on branch you're merging in (i.e. C4 isn't on the same line of development as C5)**
- **Git does 3-way merge using common ancestor**



$ git checkout master

Merging: What is a 3-way Merge?

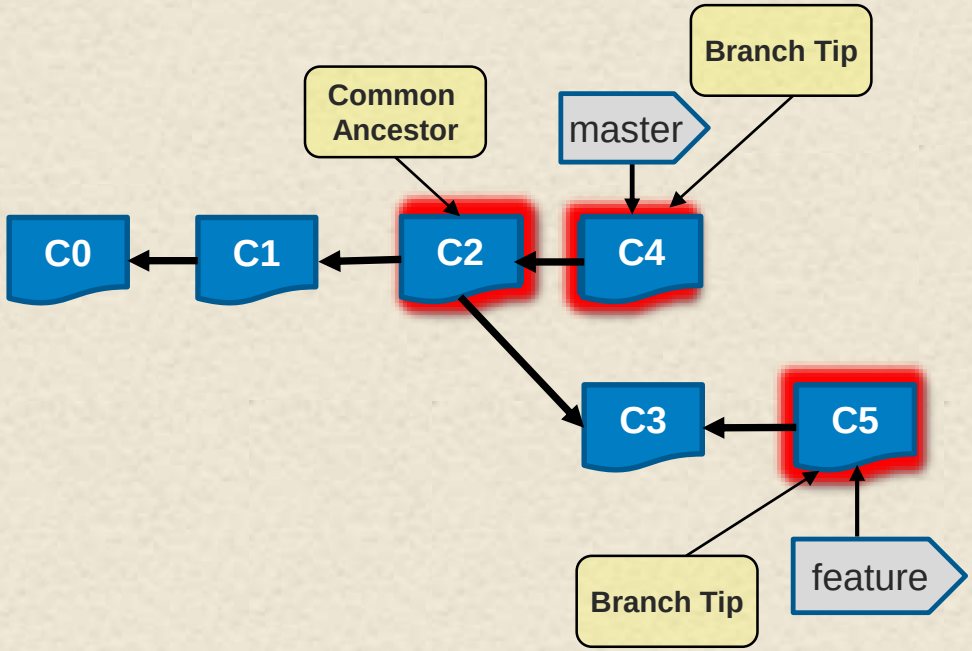
- **Assume branching scenario below**
  - master and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)
- **Intent is to change to master and merge in feature**
- **Current commit on target branch isn't a direct ancestor of current commit on branch you're merging in (i.e. C4 isn't on the same line of development as C5)**
- **Git does 3-way merge using common ancestor**




```
$ git checkout master
$ git merge feature
```

Merging: What is a 3-way Merge?

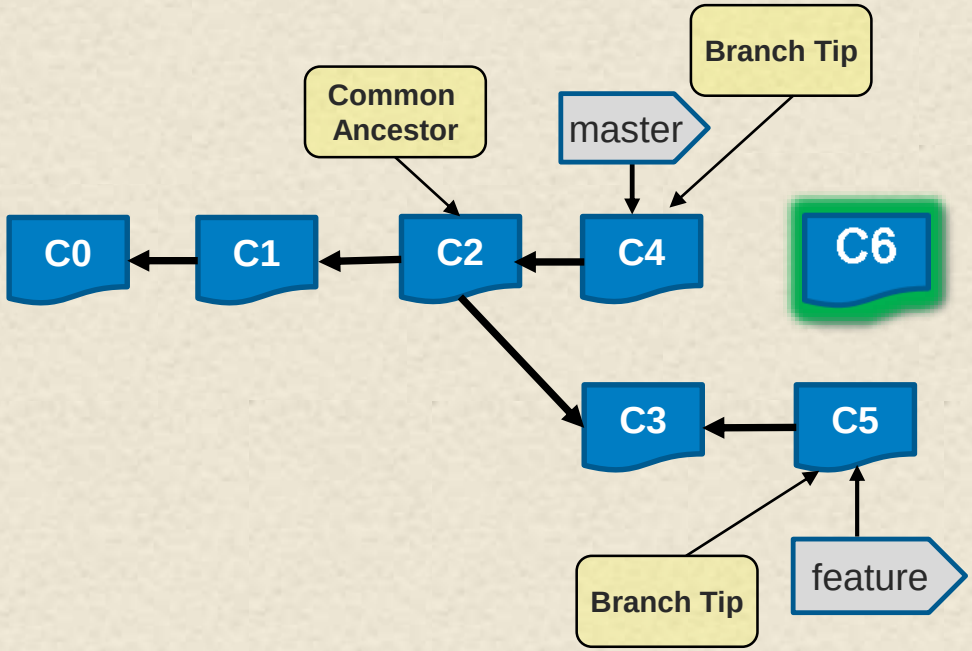
- **Assume branching scenario below**
  - master and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)
- **Intent is to change to master and merge in feature**
- **Current commit on target branch isn't a direct ancestor of current commit on branch you're merging in (i.e. C4 isn't on the same line of development as C5)**
- **Git does 3-way merge using common ancestor**




```
$ git checkout master
$ git merge feature
```

Merging: What is a 3-way Merge?

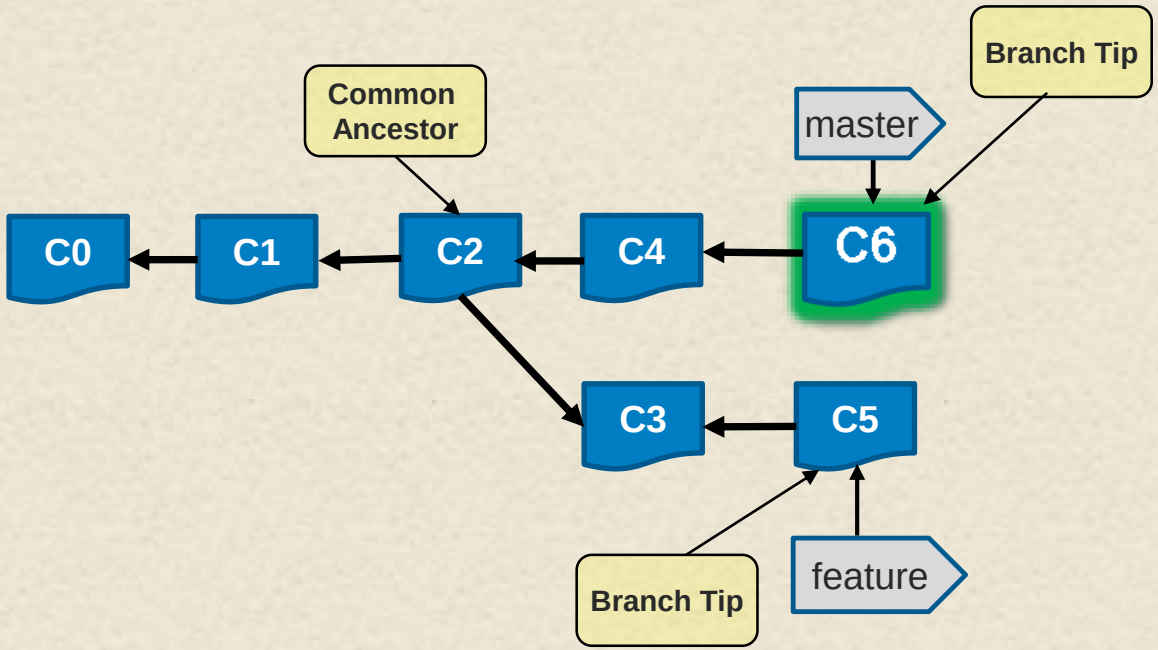
- **Assume branching scenario below**
  - master and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)
- **Intent is to change to master and merge in feature**
- **Current commit on target branch isn't a direct ancestor of current commit on branch you're merging in (i.e. C4 isn't on the same line of development as C5)**
- **Git does 3-way merge using common ancestor**




```
$ git checkout master
$ git merge feature
```

Merging: What is a 3-way Merge?

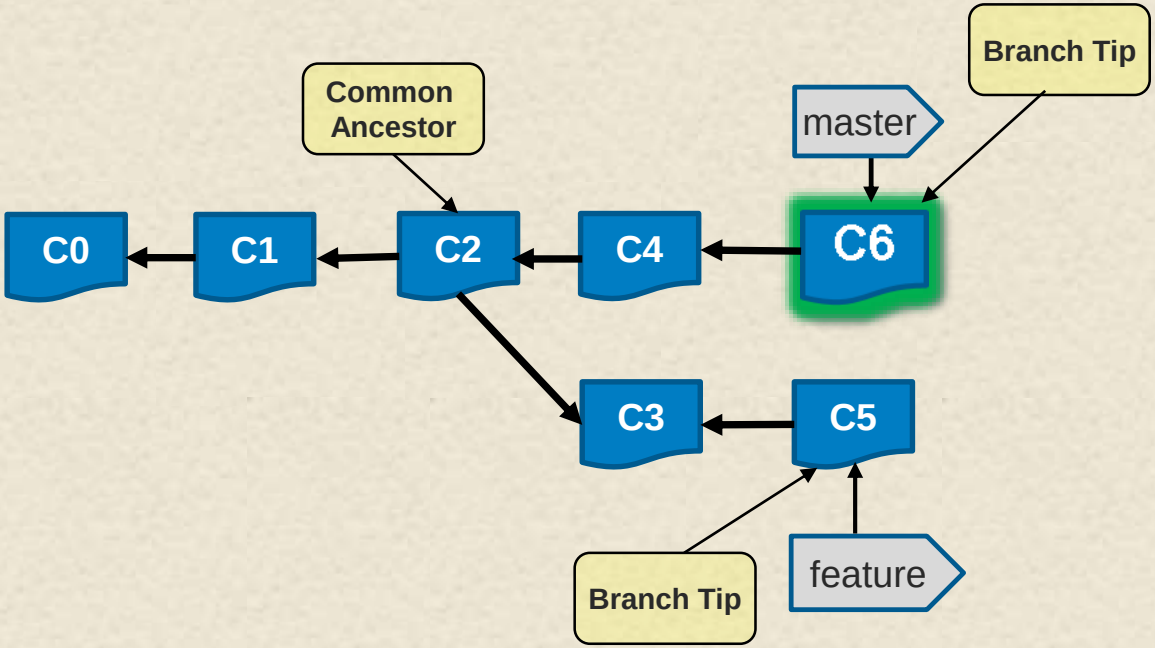
- **Assume branching scenario below**
  - master and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)
- **Intent is to change to master and merge in feature**
- **Current commit on target branch isn't a direct ancestor of current commit on branch you're merging in (i.e. C4 isn't on the same line of development as C5)**
- **Git does 3-way merge using common ancestor**



$ git checkout master
$ git merge feature

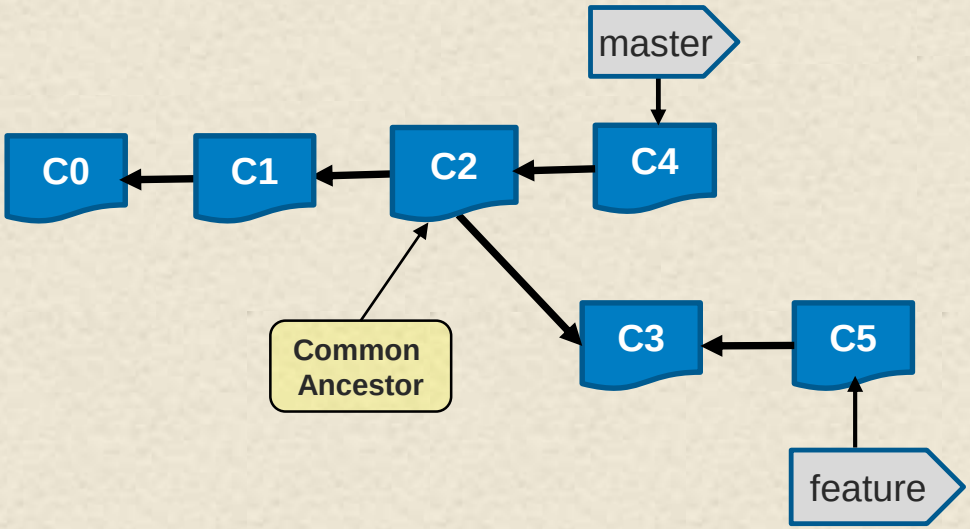
Merging: What is a 3-way Merge?

- **Assume branching scenario below**
  - master and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)
- **Intent is to change to master and merge in feature**
- **Current commit on target branch isn't a direct ancestor of current commit on branch you're merging in (i.e. C4 isn't on the same line of development as C5)**
- **Git does 3-way merge using common ancestor**
- **Instead of just moving branch pointer forward, Git creates a new snapshot and a new commit that points to it called a "merge commit"**



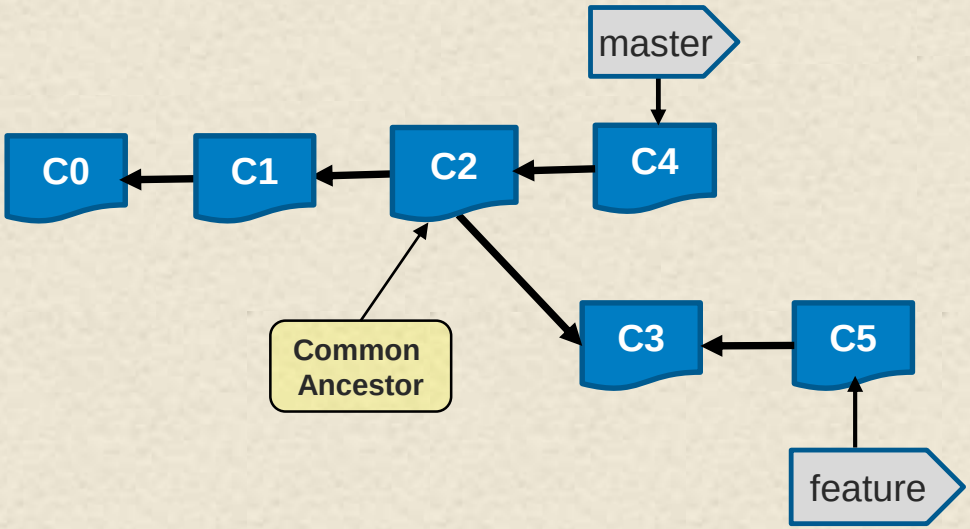
$ git checkout master
$ git merge feature

Merging: What is a Rebase?



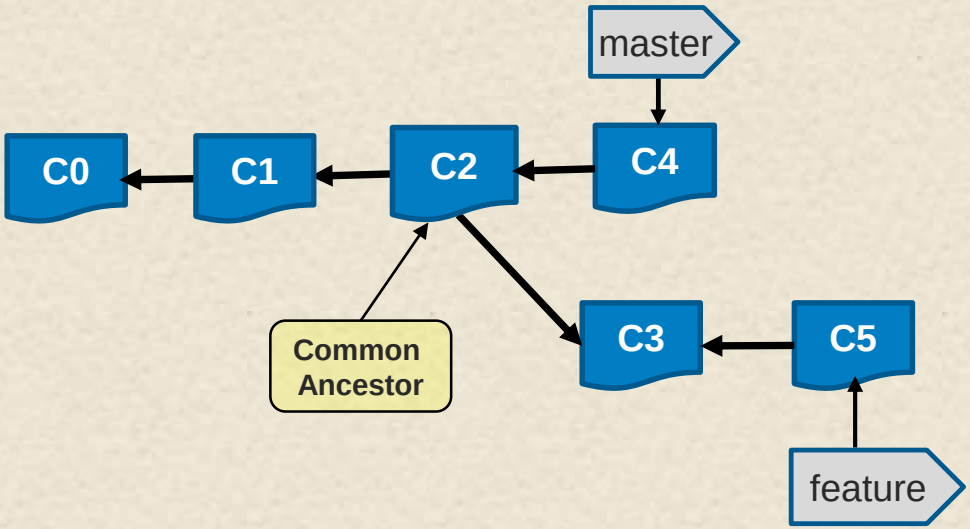
Merging: What is a Rebase?

- **Rebase – take all of the changes that were committed on one branch and replay them on another one.**



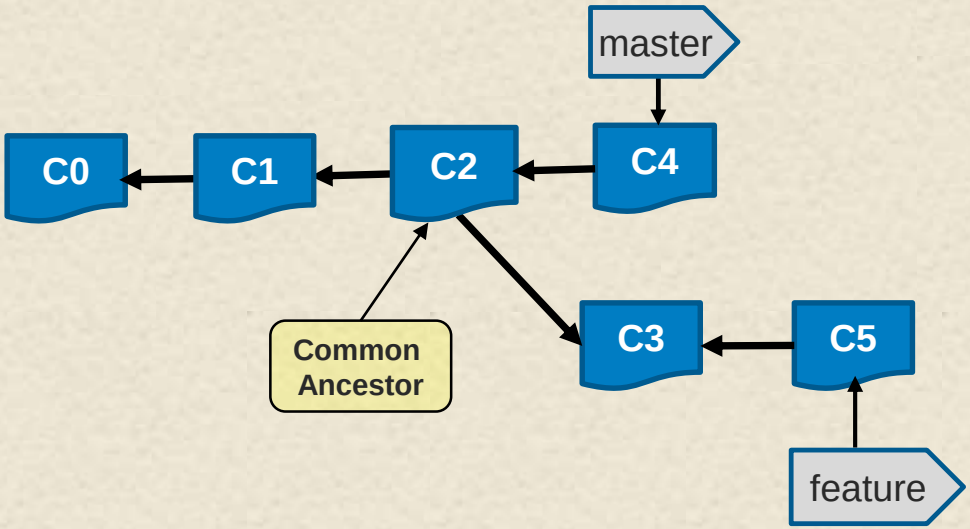
Merging: What is a Rebase?

- **Rebase – take all of the changes that were committed on one branch and replay them on another one.**
- **Concepts in simple syntax (git rebase branch2 [branch1]):**



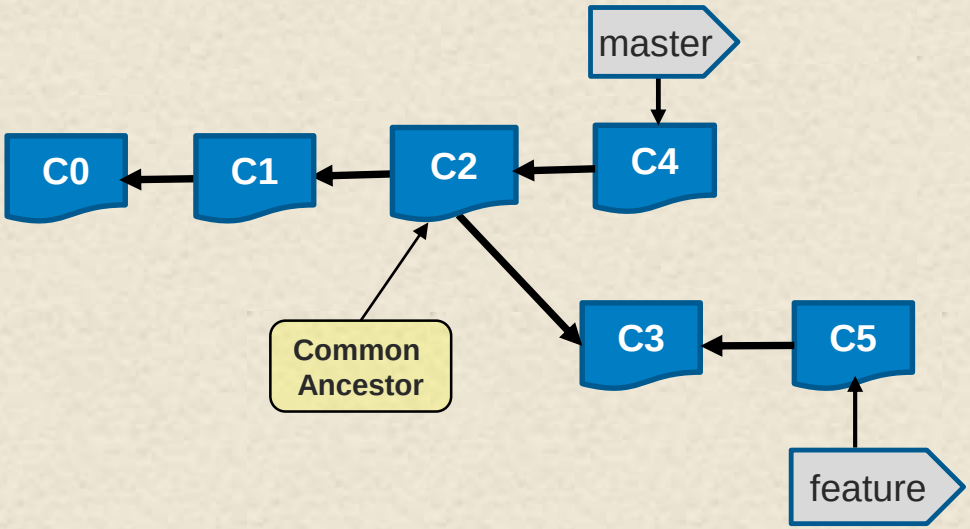
Merging: What is a Rebase?

- **Rebase – take all of the changes that were committed on one branch and replay them on another one.**
- **Concepts in simple syntax (git rebase branch2 [branch1]):**
  - **Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)**



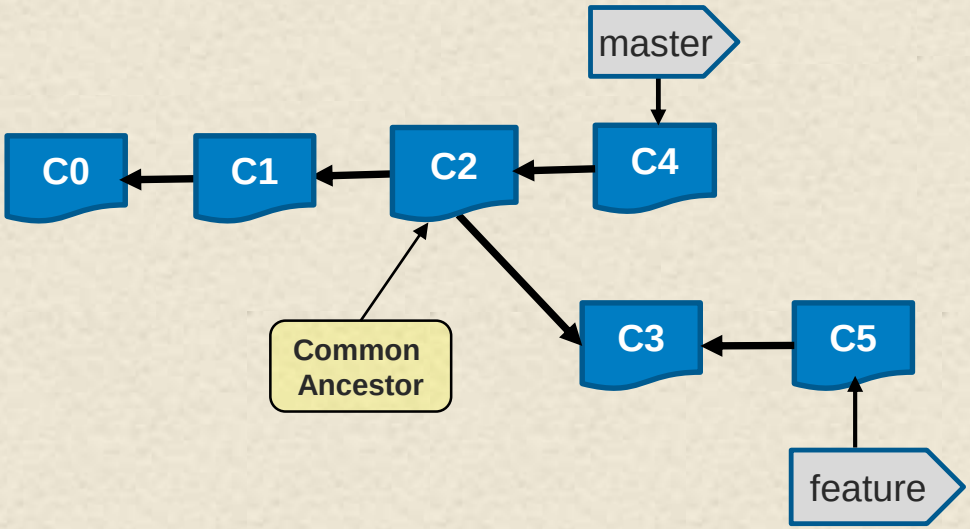
Merging: What is a Rebase?

- **Rebase – take all of the changes that were committed on one branch and replay them on another one.**
- **Concepts in simple syntax (git rebase branch2 [branch1]):**
  - **Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)**
  - **Current branch (or branch1 if supplied) is one having its branchpoint moved**



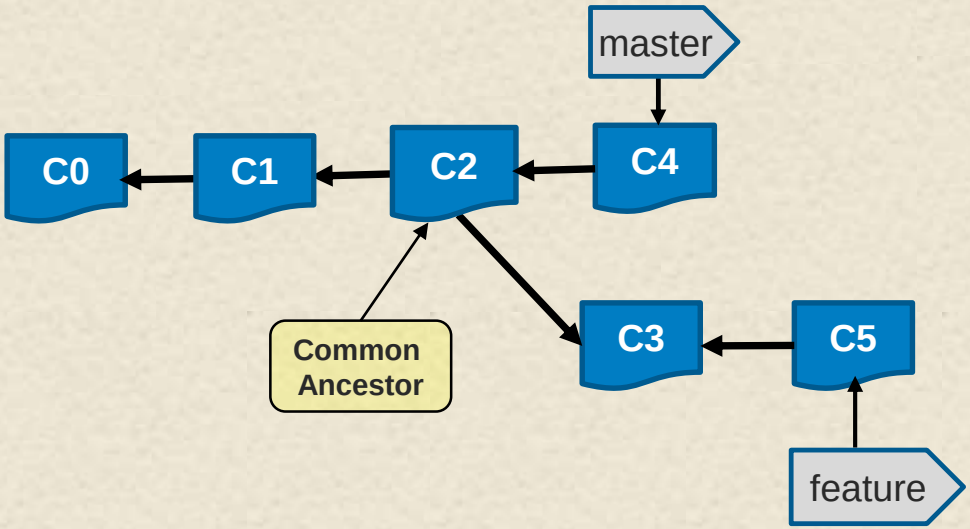
Merging: What is a Rebase?

- **Rebase – take all of the changes that were committed on one branch and replay them on another one.**
- **Concepts in simple syntax (git rebase branch2 [branch1]):**
  - **Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)**
  - **Current branch (or branch1 if supplied) is one having its branchpoint moved**
  - **Branch2 (or commit) provided as first/only argument is the new branchpoint**



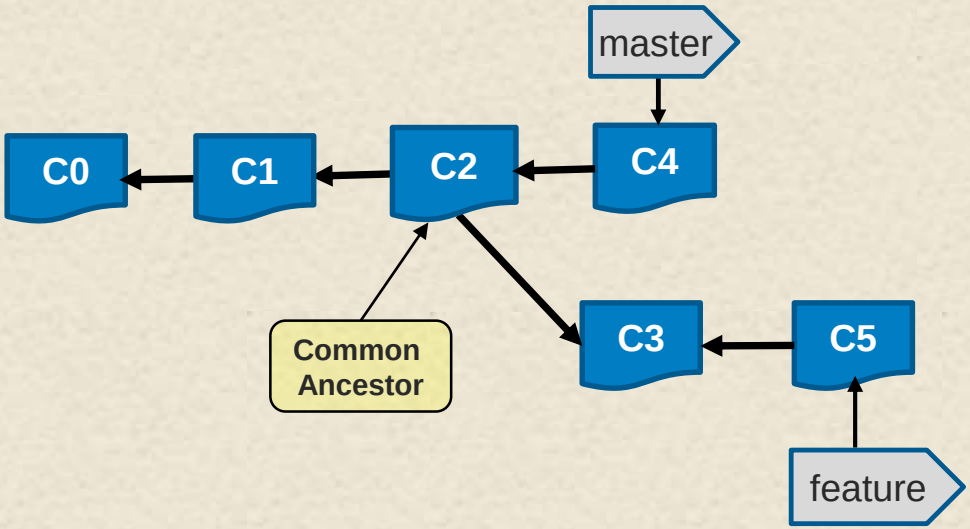
Merging: What is a Rebase?

- **Rebase – take all of the changes that were committed on one branch and replay them on another one.**
- **Concepts in simple syntax (git rebase branch2 [branch1]):**
  - **Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)**
  - **Current branch (or branch1 if supplied) is one having its branchpoint moved**
  - **Branch2 (or commit) provided as first/only argument is the new branchpoint**
  - **In simple case, think of it as "pick up branch1 entirely and move its branchpoint to be after the tip of branch2)"**



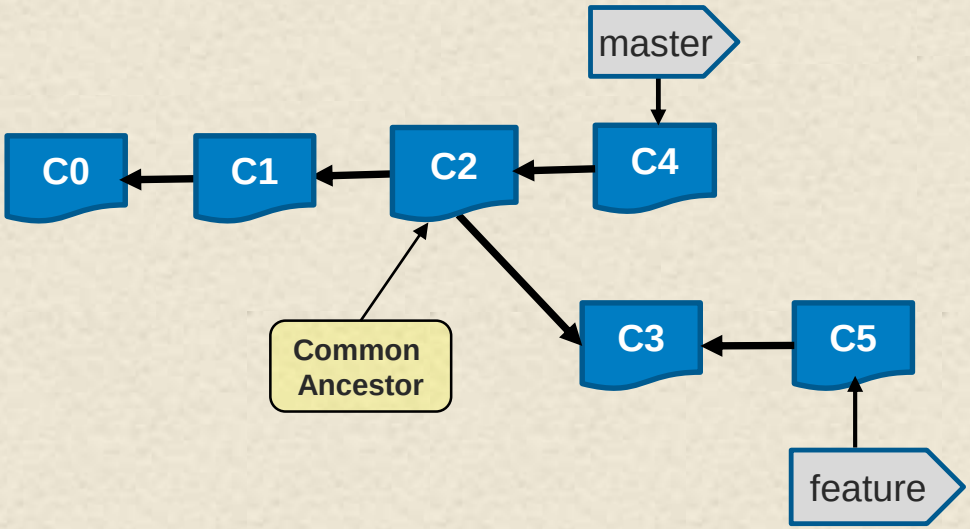
Merging: What is a Rebase?

- **Rebase – take all of the changes that were committed on one branch and replay them on another one.**
- **Concepts in simple syntax (git rebase branch2 [branch1]):**
  - **Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)**
  - **Current branch (or branch1 if supplied) is one having its branchpoint moved**
  - **Branch2 (or commit) provided as first/only argument is the new branchpoint**
  - **In simple case, think of it as "pick up branch1 entirely and move its branchpoint to be after the tip of branch2)"**
- **Process:**



Merging: What is a Rebase?

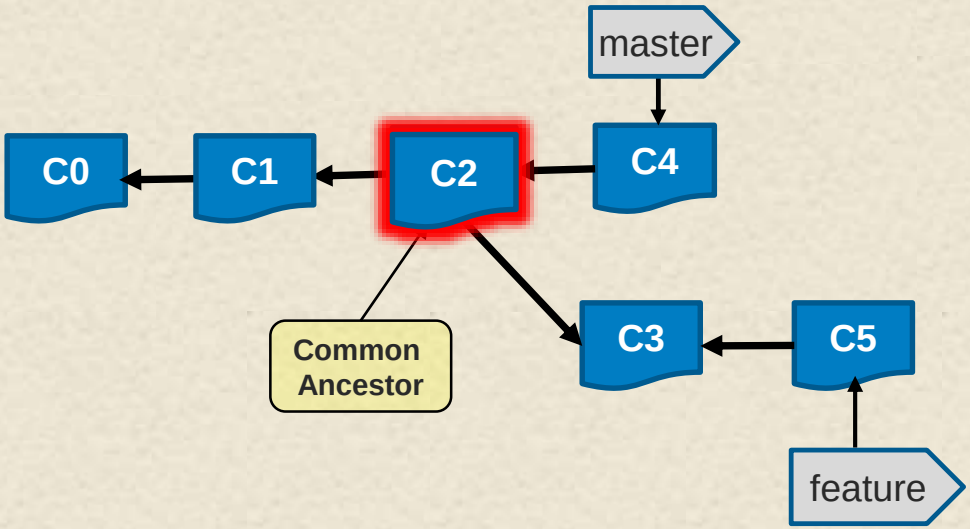
- Rebase – take all of the changes that were committed on one branch and replay them on another one.
- Concepts in simple syntax (git rebase branch2 [branch1]):
  - Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)
  - Current branch (or branch1 if supplied) is one having its branchpoint moved
  - Branch2 (or commit) provided as first/only argument is the new branchpoint
  - In simple case, think of it as "pick up branch1 entirely and move its branchpoint to be after the tip of branch2)"
- Process:




```
$ git checkout feature
$ git rebase master
```

Merging: What is a Rebase?

- Rebase – take all of the changes that were committed on one branch and replay them on another one.
- Concepts in simple syntax (git rebase branch2 [branch1]):
  - Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)
  - Current branch (or branch1 if supplied) is one having its branchpoint moved
  - Branch2 (or commit) provided as first/only argument is the new branchpoint
  - In simple case, think of it as "pick up branch1 entirely and move its branchpoint to be after the tip of branch2)"
- Process:
  - Goes to the common ancestor of the two branches (the one you are on and the one you are rebasing onto)

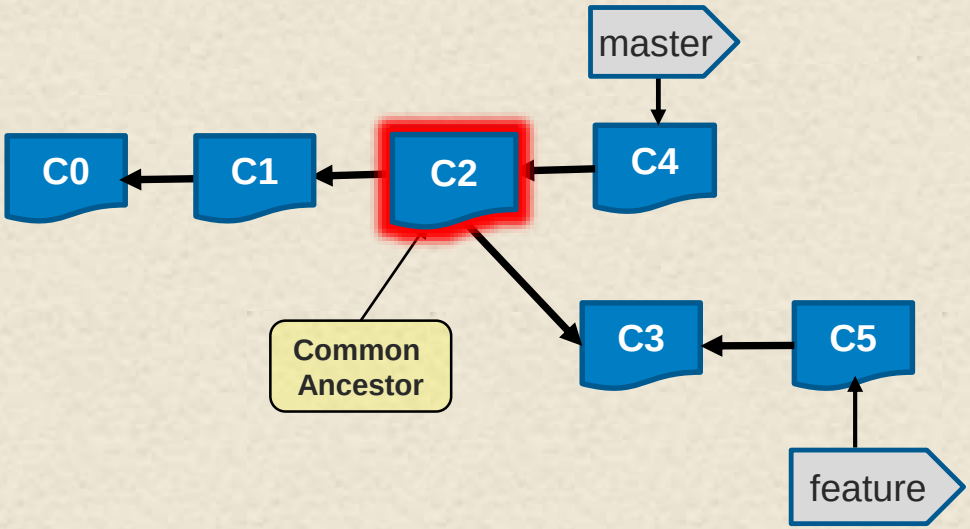



```
$ git checkout feature
$ git rebase master
```

Merging: What is a Rebase?

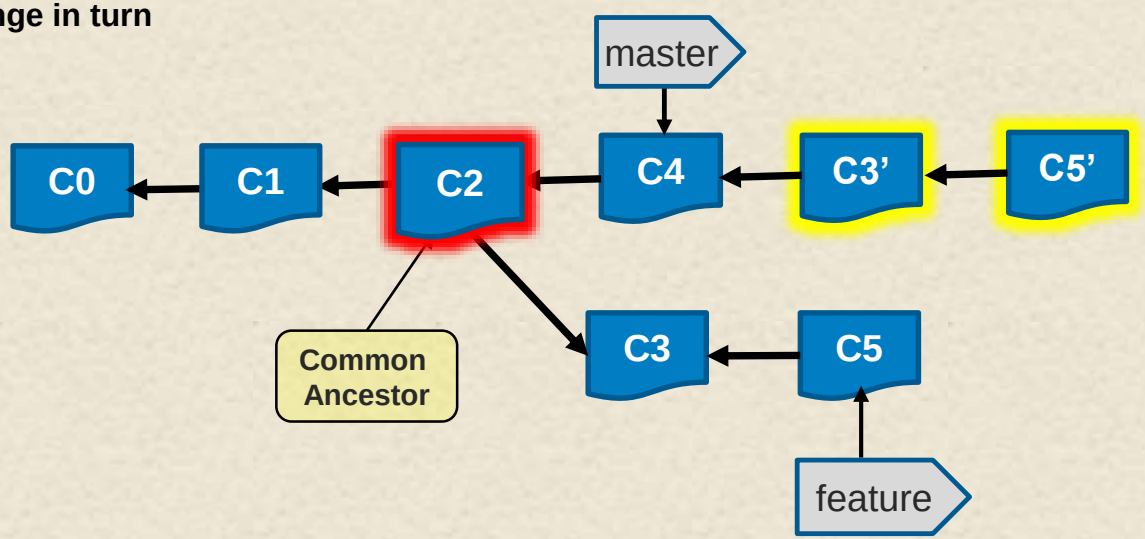
- Rebase – take all of the changes that were committed on one branch and replay them on another one.
- Concepts in simple syntax (git rebase branch2 [branch1]):
  - Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)
  - Current branch (or branch1 if supplied) is one having its branchpoint moved
  - Branch2 (or commit) provided as first/only argument is the new branchpoint
  - In simple case, think of it as "pick up branch1 entirely and move its branchpoint to be after the tip of branch2)"
- Process:
  - Goes to the common ancestor of the two branches (the one you are on and the one you are rebasing onto)
  - Gets the diff introduced by each commit of the branch you are on, saving them to temporary files

```
$ git checkout feature
$ git rebase master
```



Merging: What is a Rebase?

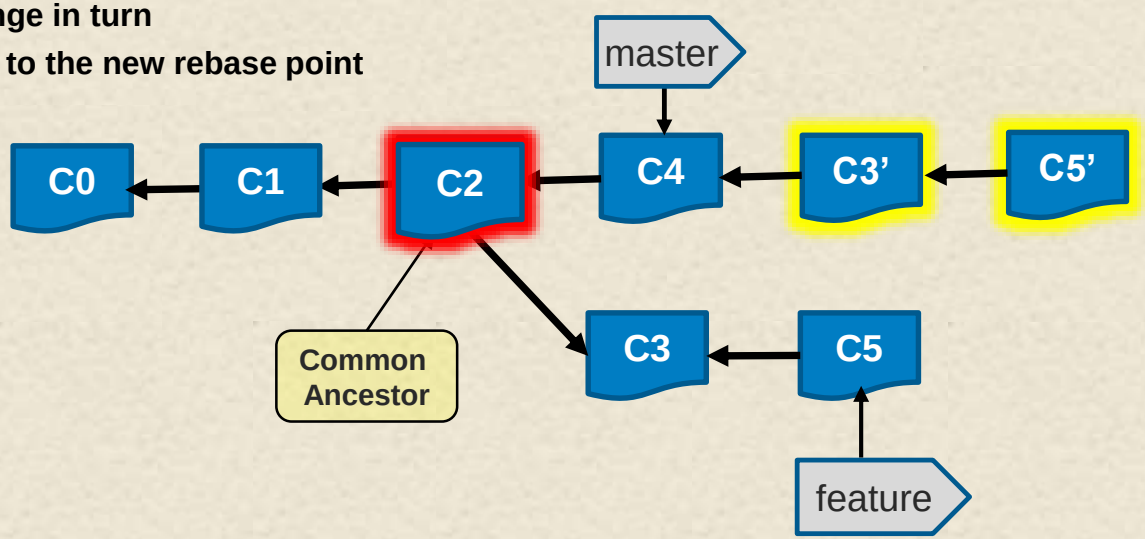
- Rebase – take all of the changes that were committed on one branch and replay them on another one.
- Concepts in simple syntax (git rebase branch2 [branch1]):
  - Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)
  - Current branch (or branch1 if supplied) is one having its branchpoint moved
  - Branch2 (or commit) provided as first/only argument is the new branchpoint
  - In simple case, think of it as "pick up branch1 entirely and move its branchpoint to be after the tip of branch2)"
- Process:
  - Goes to the common ancestor of the two branches (the one you are on and the one you are rebasing onto)
  - Gets the diff introduced by each commit of the branch you are on, saving them to temporary files
  - Applies each change in turn




```
$ git checkout feature
$ git rebase master
```

Merging: What is a Rebase?

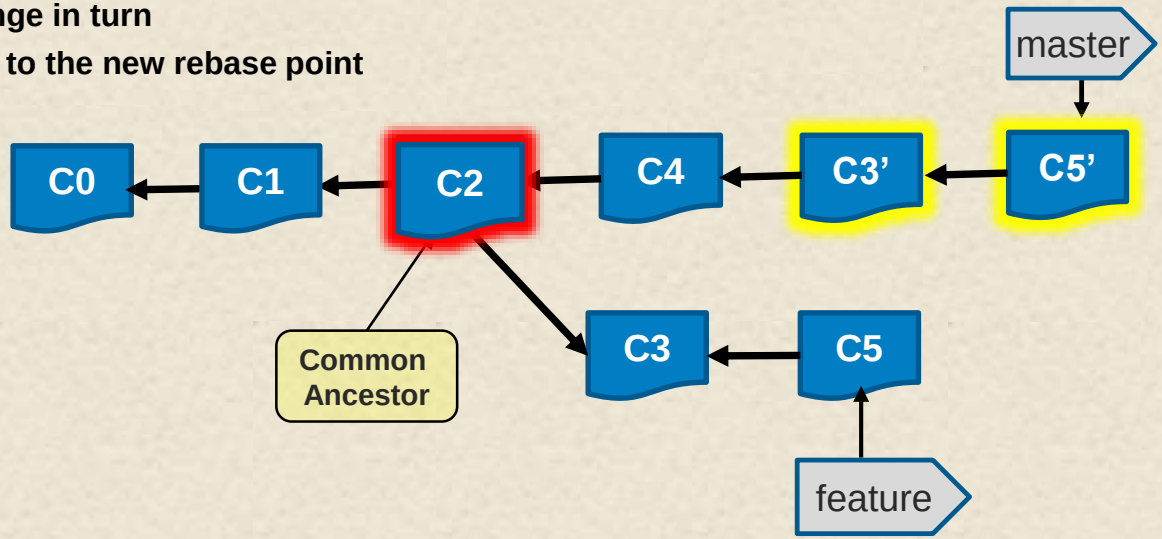
- Rebase – take all of the changes that were committed on one branch and replay them on another one.
- Concepts in simple syntax (git rebase branch2 [branch1]):
  - Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)
  - Current branch (or branch1 if supplied) is one having its branchpoint moved
  - Branch2 (or commit) provided as first/only argument is the new branchpoint
  - In simple case, think of it as "pick up branch1 entirely and move its branchpoint to be after the tip of branch2)"
- Process:
  - Goes to the common ancestor of the two branches (the one you are on and the one you are rebasing onto)
  - Gets the diff introduced by each commit of the branch you are on, saving them to temporary files
  - Applies each change in turn
  - Moves the branch to the new rebase point




```
$ git checkout feature
$ git rebase master
```

Merging: What is a Rebase?

- Rebase – take all of the changes that were committed on one branch and replay them on another one.
- Concepts in simple syntax (git rebase branch2 [branch1]):
  - Move branchpoint of branch (carrying along all commits) to be off of a different commit (new base)
  - Current branch (or branch1 if supplied) is one having its branchpoint moved
  - Branch2 (or commit) provided as first/only argument is the new branchpoint
  - In simple case, think of it as "pick up branch1 entirely and move its branchpoint to be after the tip of branch2)"
- Process:
  - Goes to the common ancestor of the two branches (the one you are on and the one you are rebasing onto)
  - Gets the diff introduced by each commit of the branch you are on, saving them to temporary files
  - Applies each change in turn
  - Moves the branch to the new rebase point




```
$ git checkout feature
$ git rebase master
```

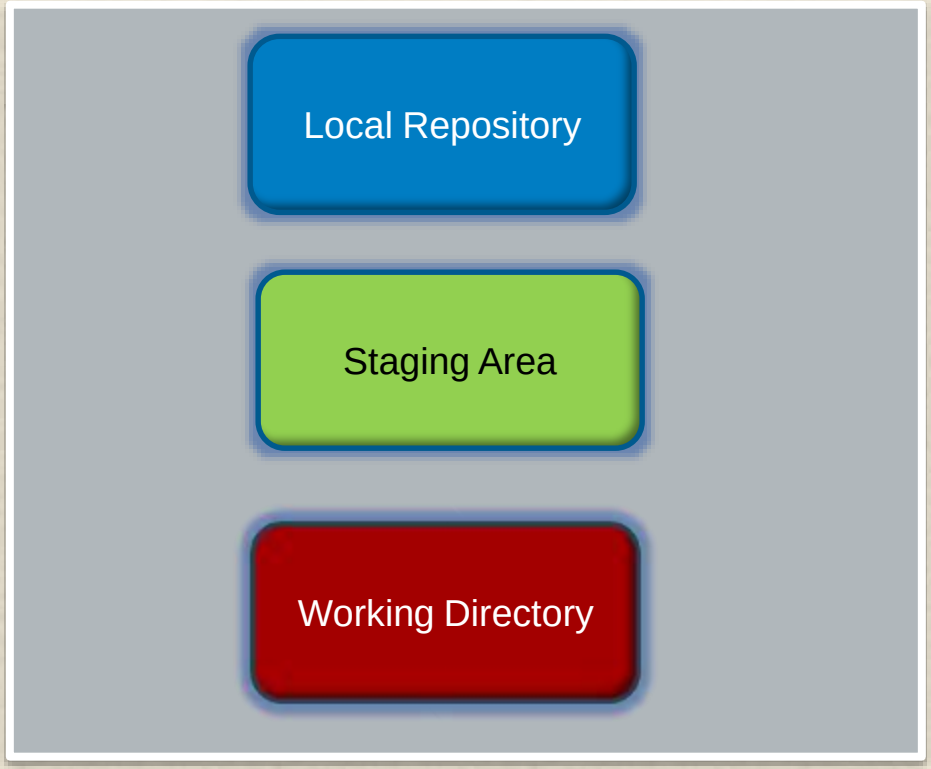
Command: Git Stash

- Purpose -- allow you to keep a backup copy of your work that hasn't been committed yet
- Use case - you want to switch branches but don't want to lose work that hasn't been committed; you want to save something you've tried and may want to come back to
- Syntax:

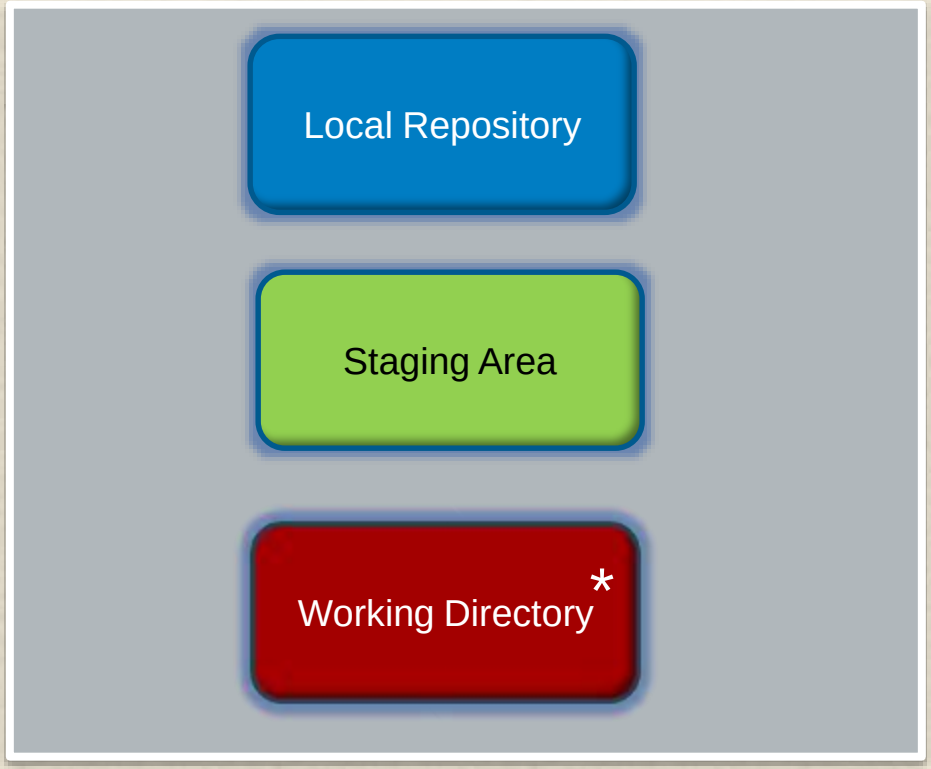
```
git stash list [<options>]
git stash show [<stash>]
git stash drop [-q|--quiet] [<stash>]
git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>]
git stash branch <branchname> [<stash>]
git stash [save [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet]
          [-u|--include-untracked] [-a|--all] [<message>]]
git stash clear
git stash create [<message>]
git stash store [-m|--message <message>] [-q|--quiet] <commit>
```

Stash

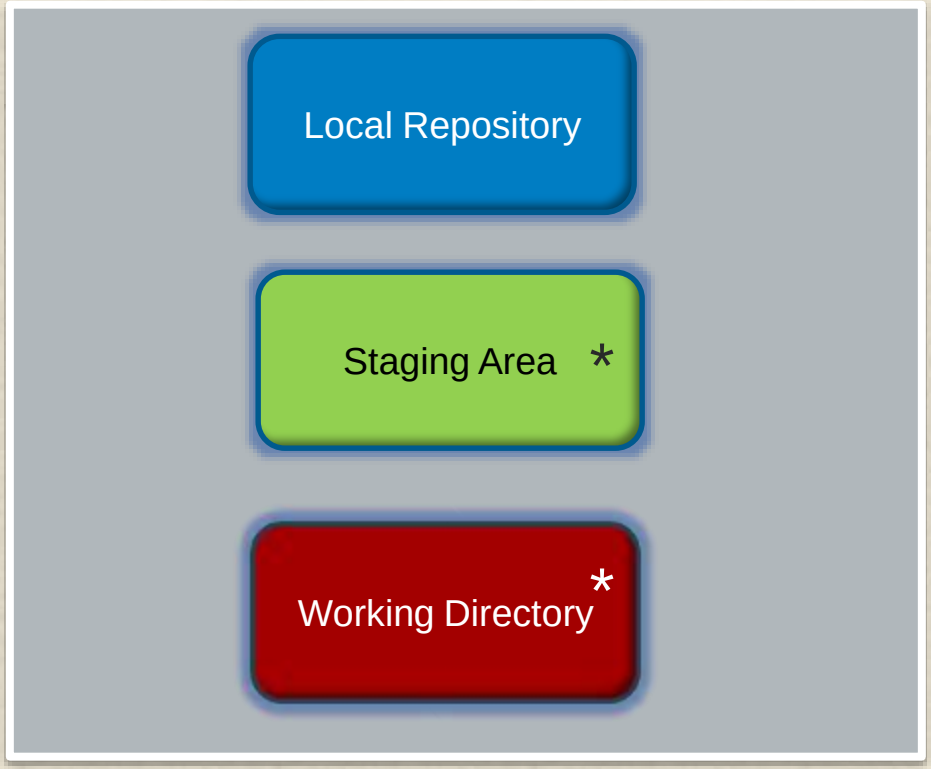
Stash



Stash

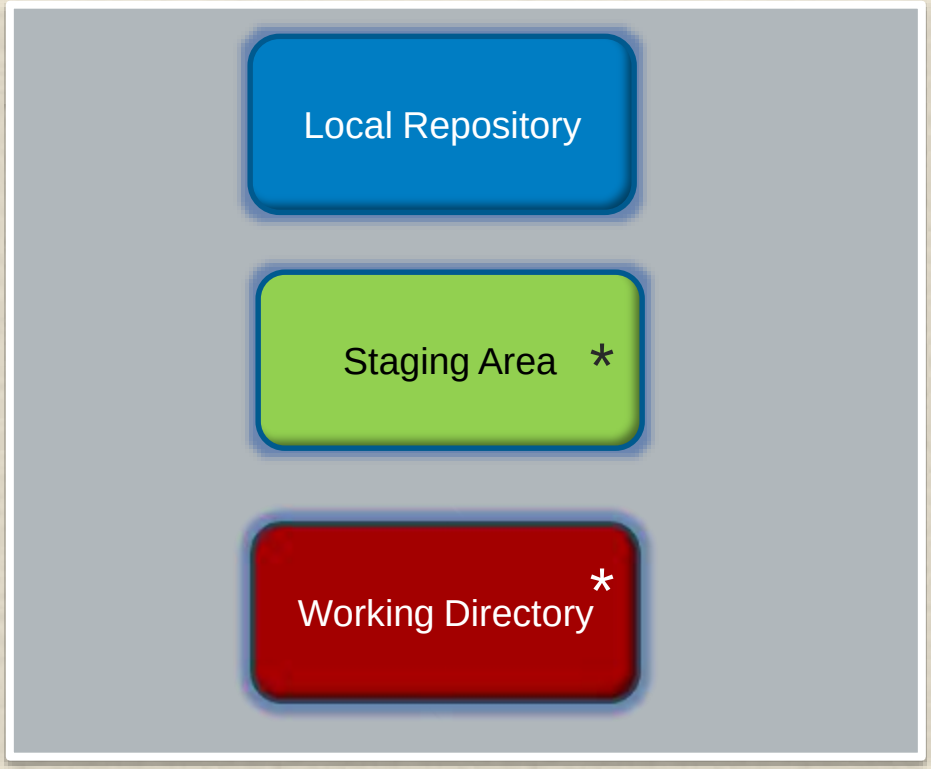


Stash



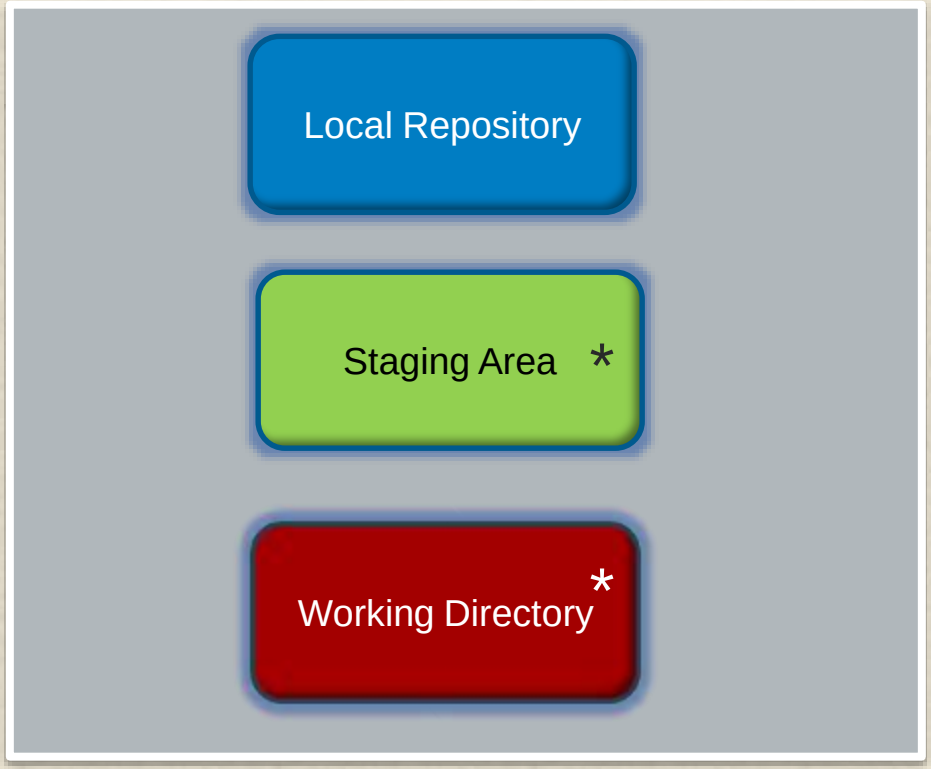
Stash

> git stash



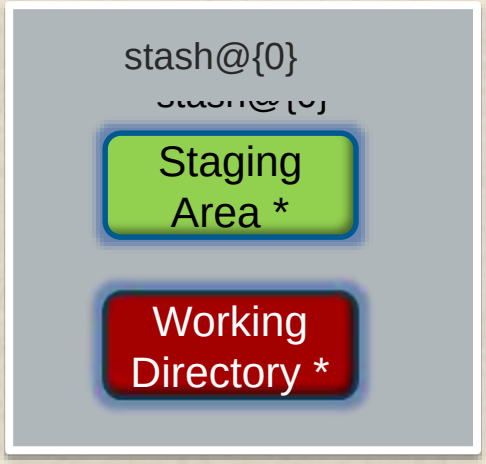
Stash

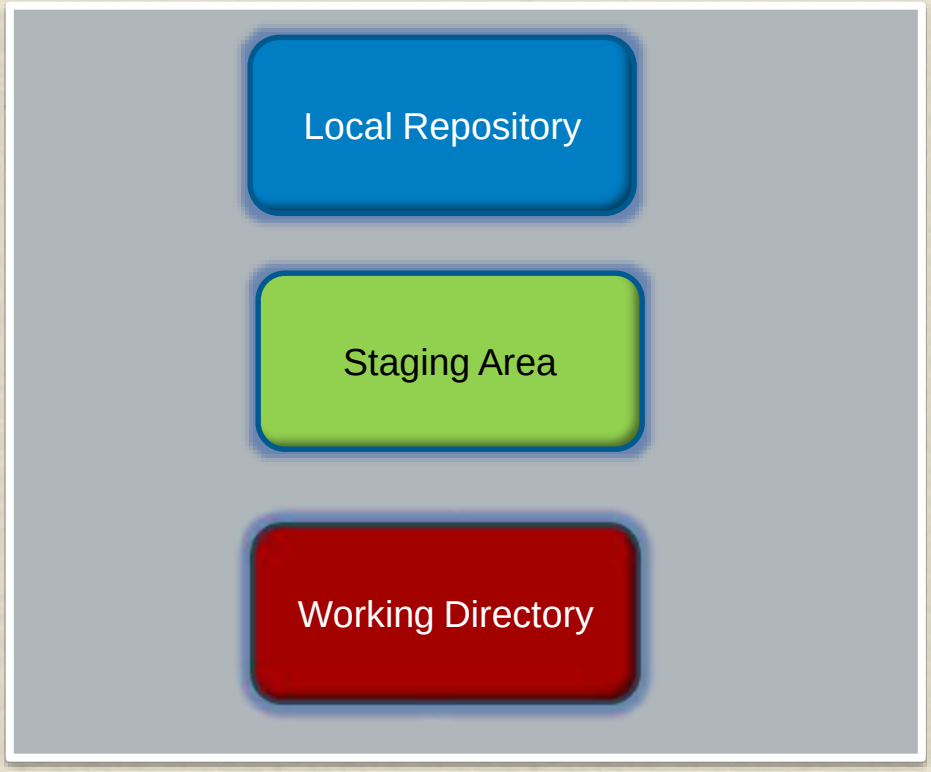
> git stash



Stash

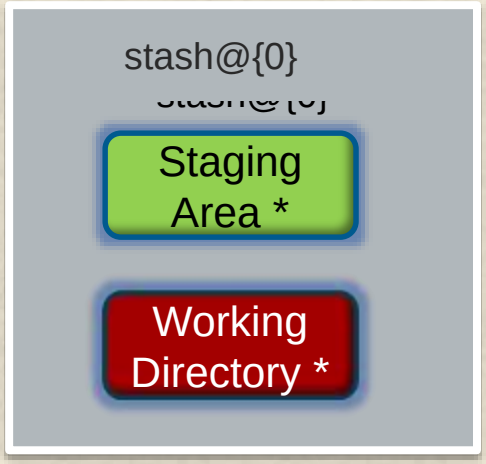
> git stash

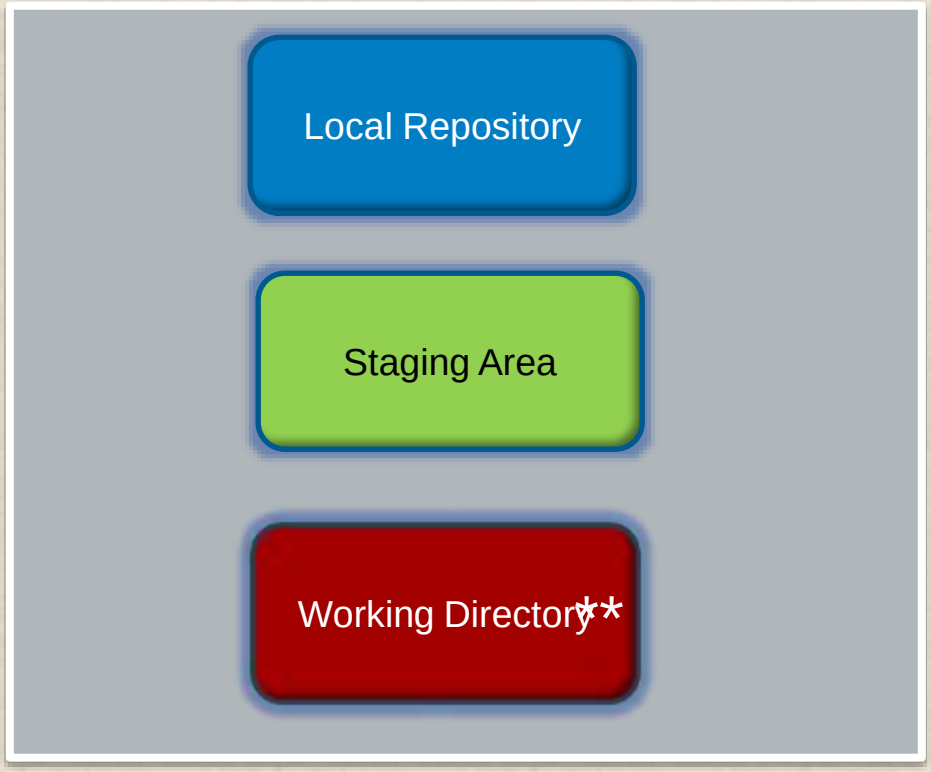




Stash

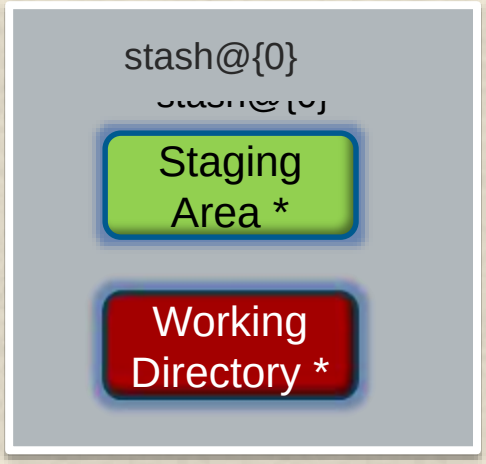
> git stash

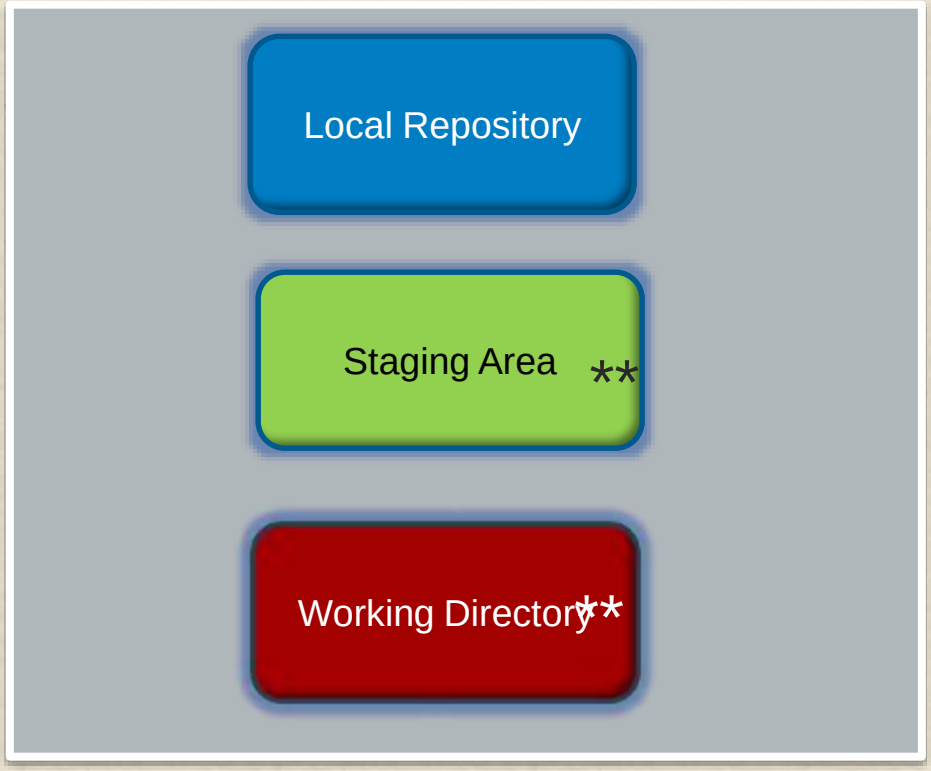




Stash

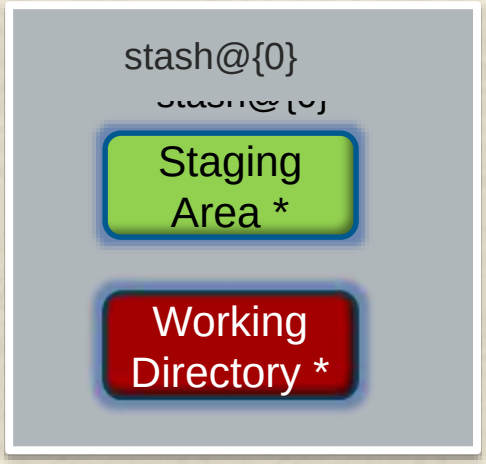
> git stash

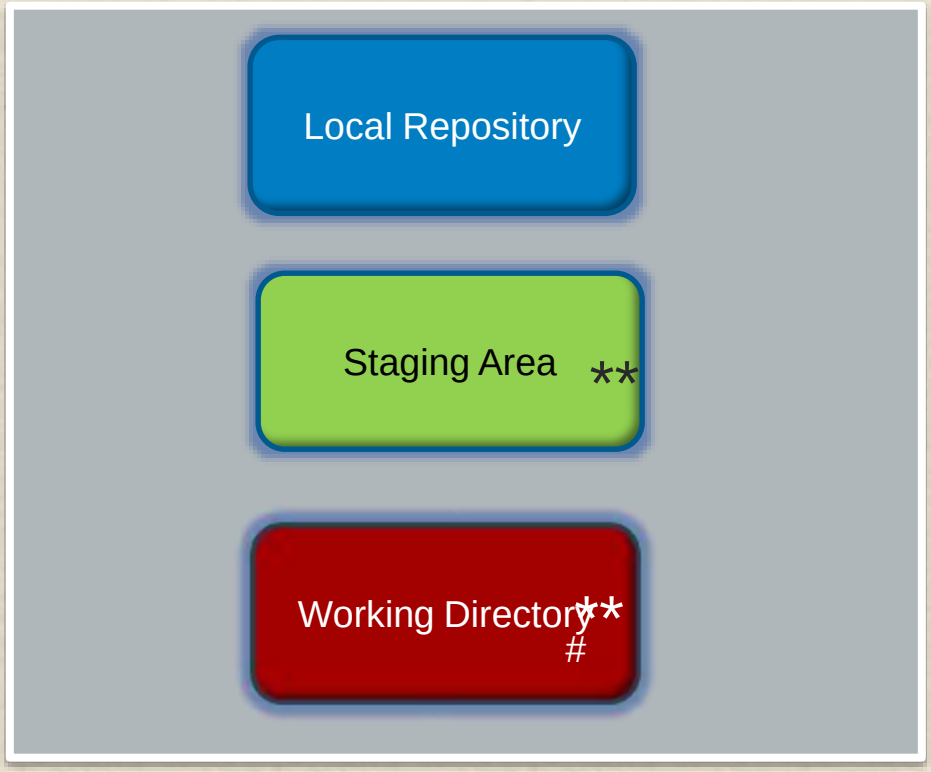




Stash

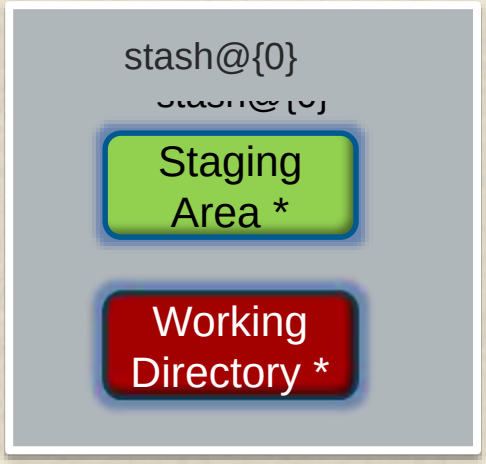
> git stash

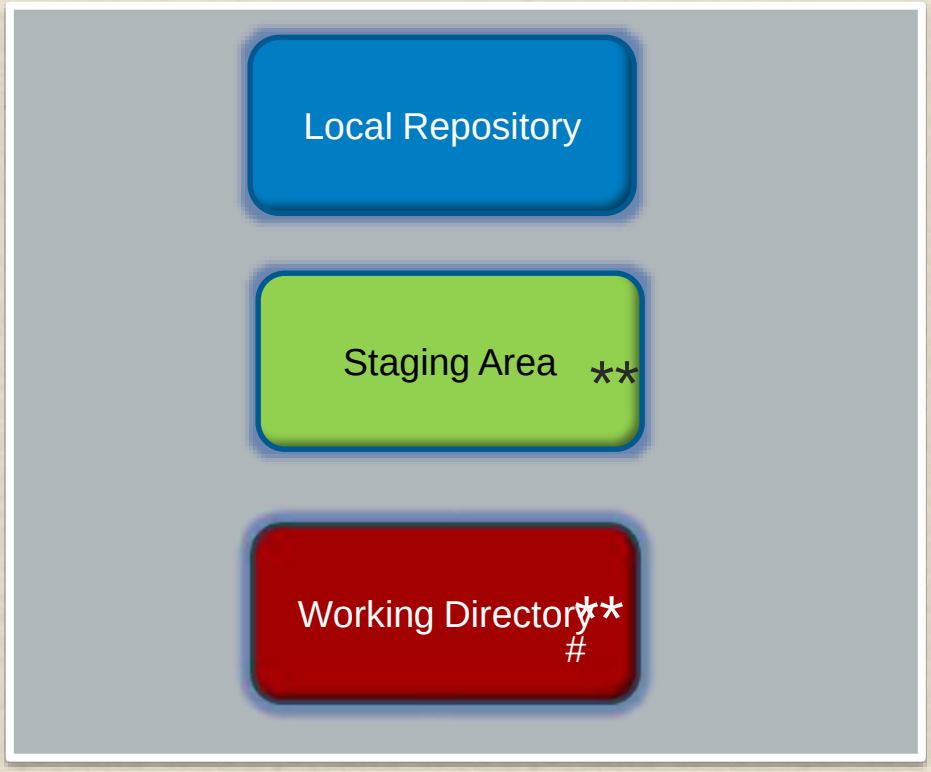




Stash

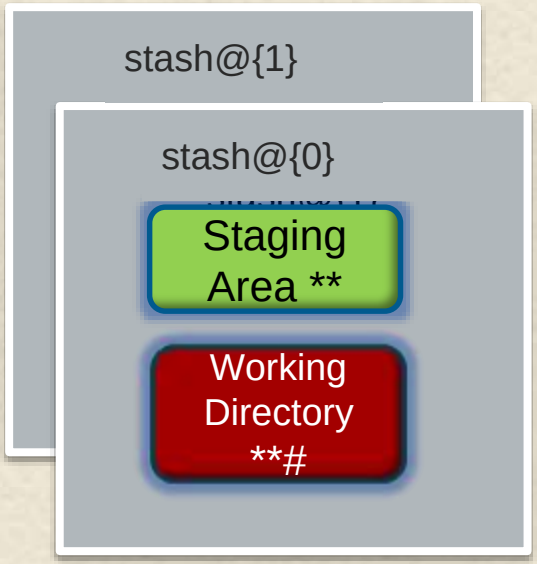
> git stash -u

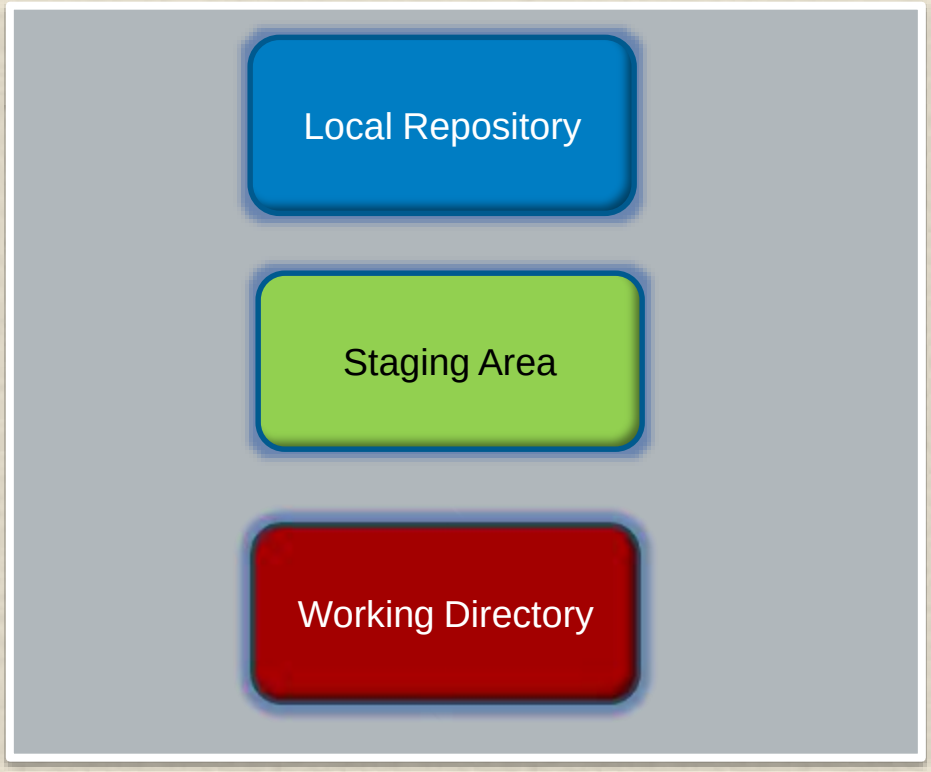




Stash

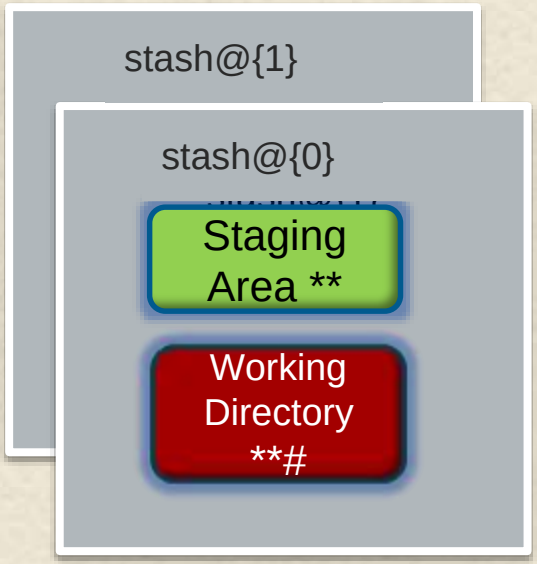
> git stash -u

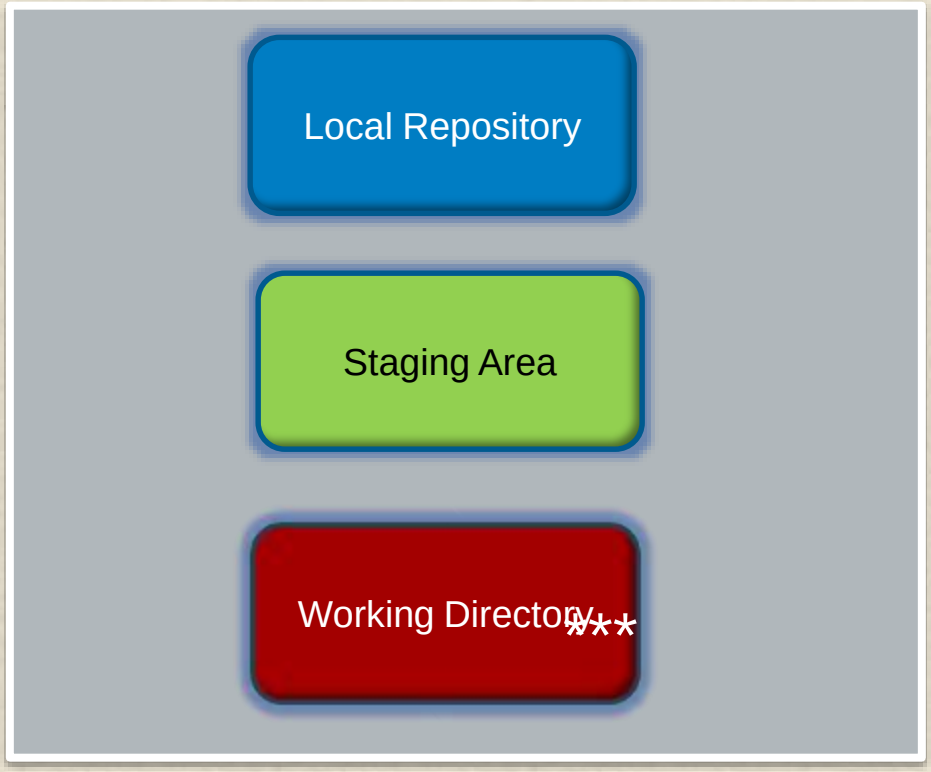




Stash

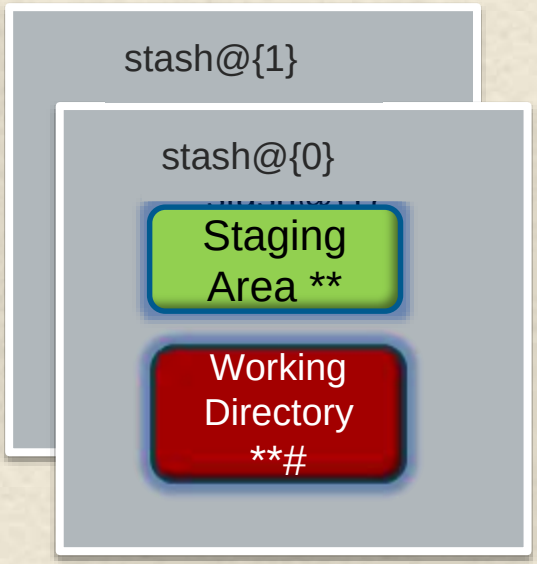
> git stash -u

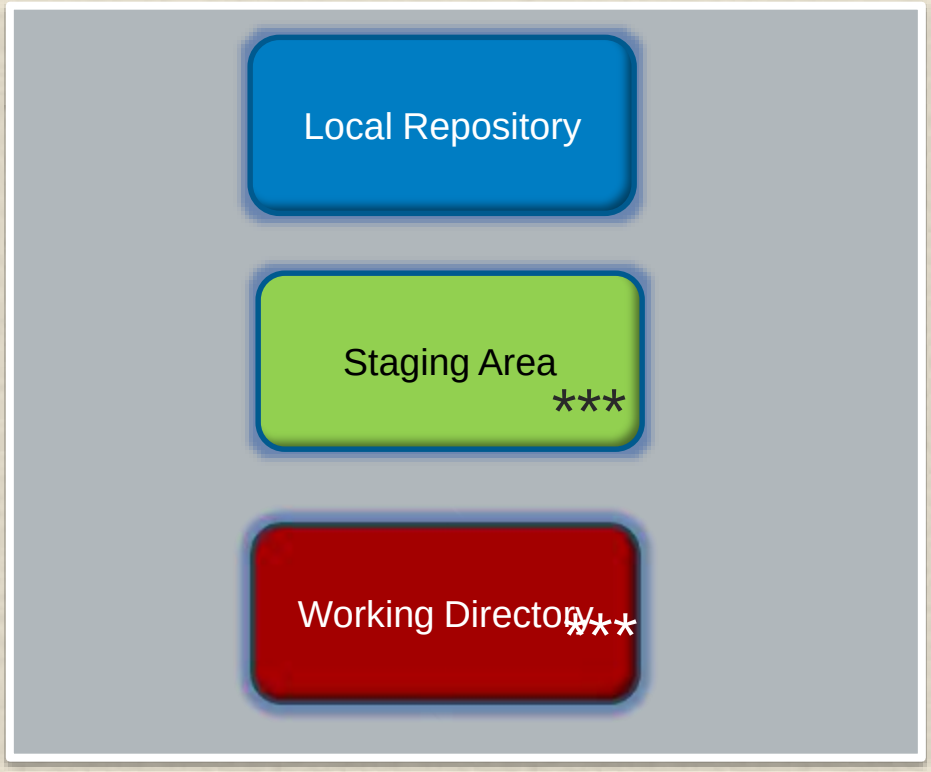




Stash

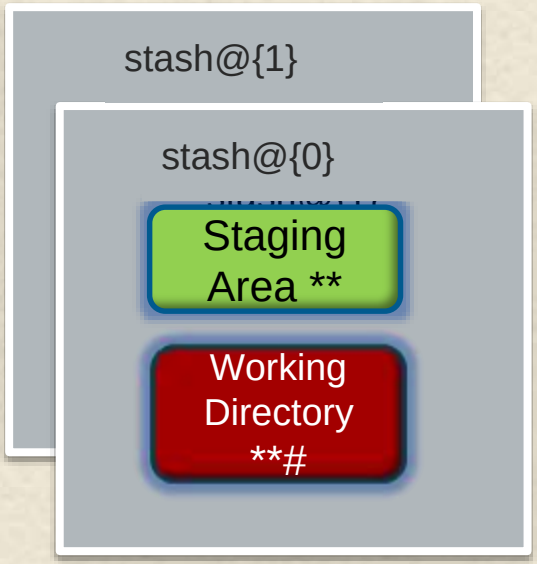
> git stash -u

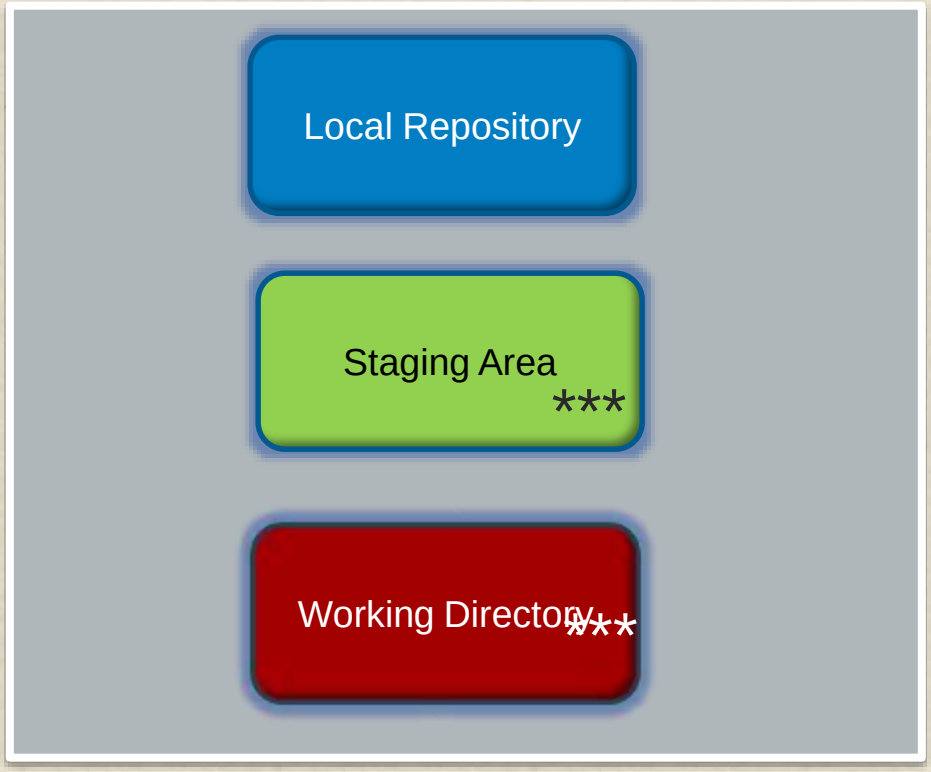




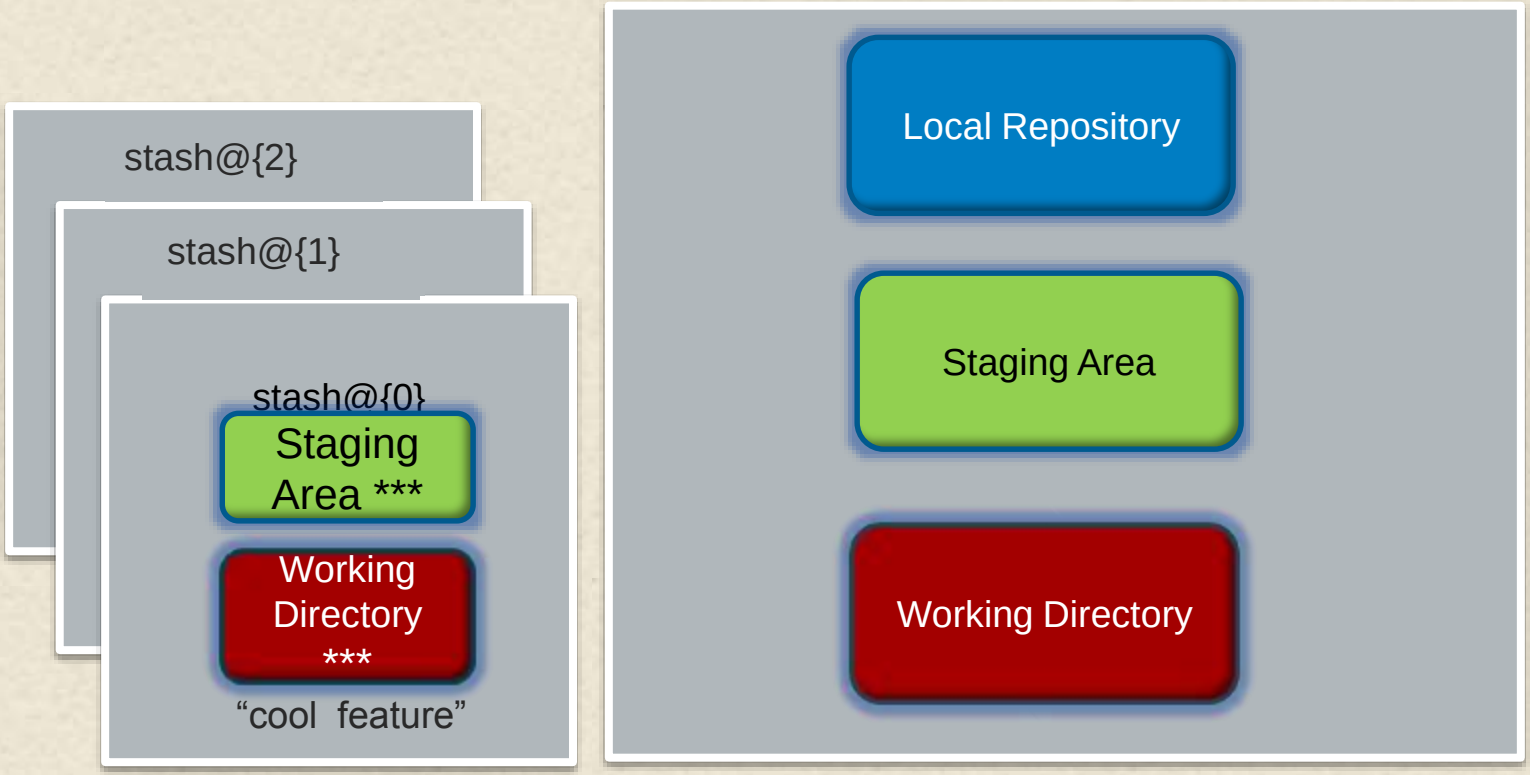
Stash

> git stash save “cool feature”



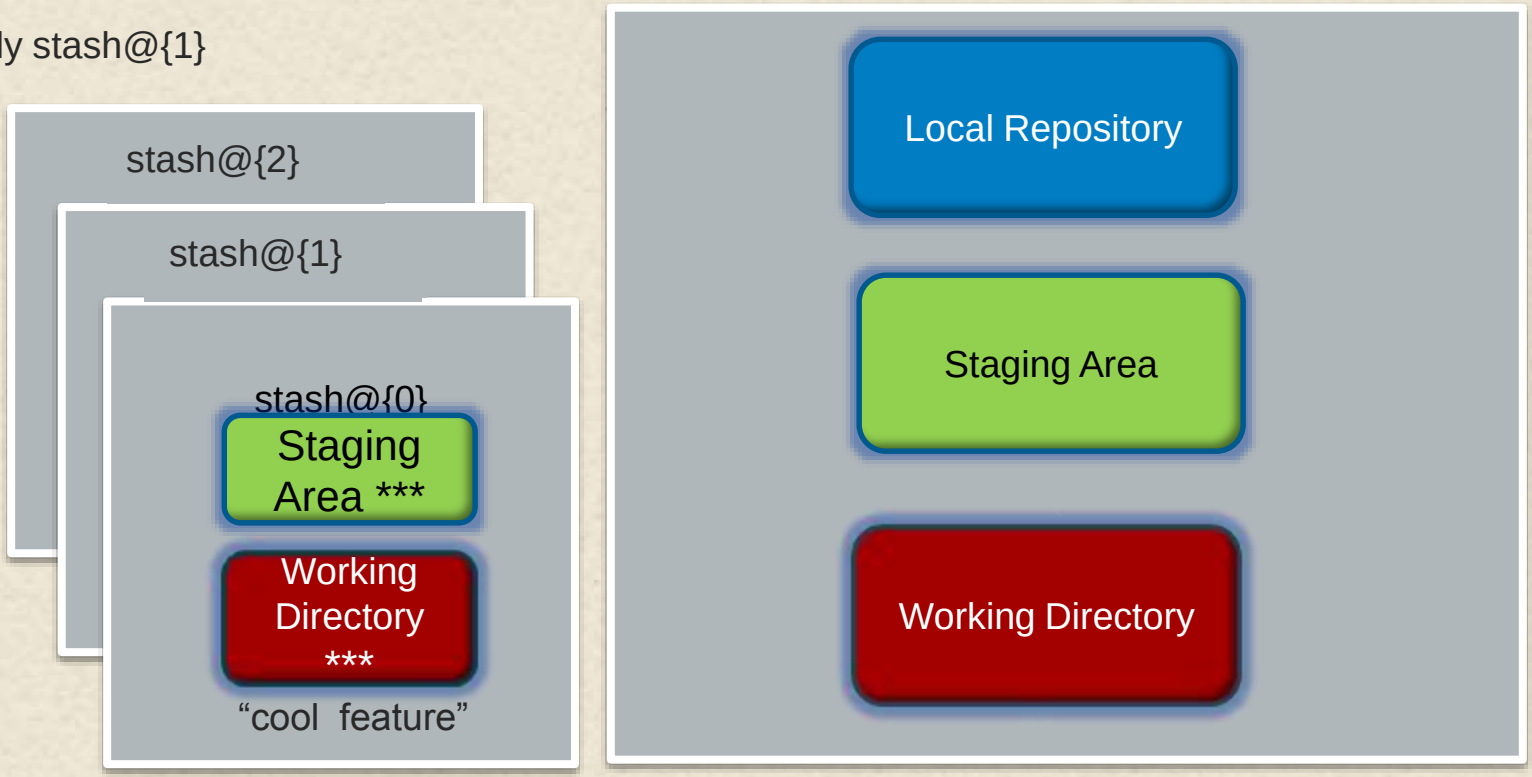


Stash



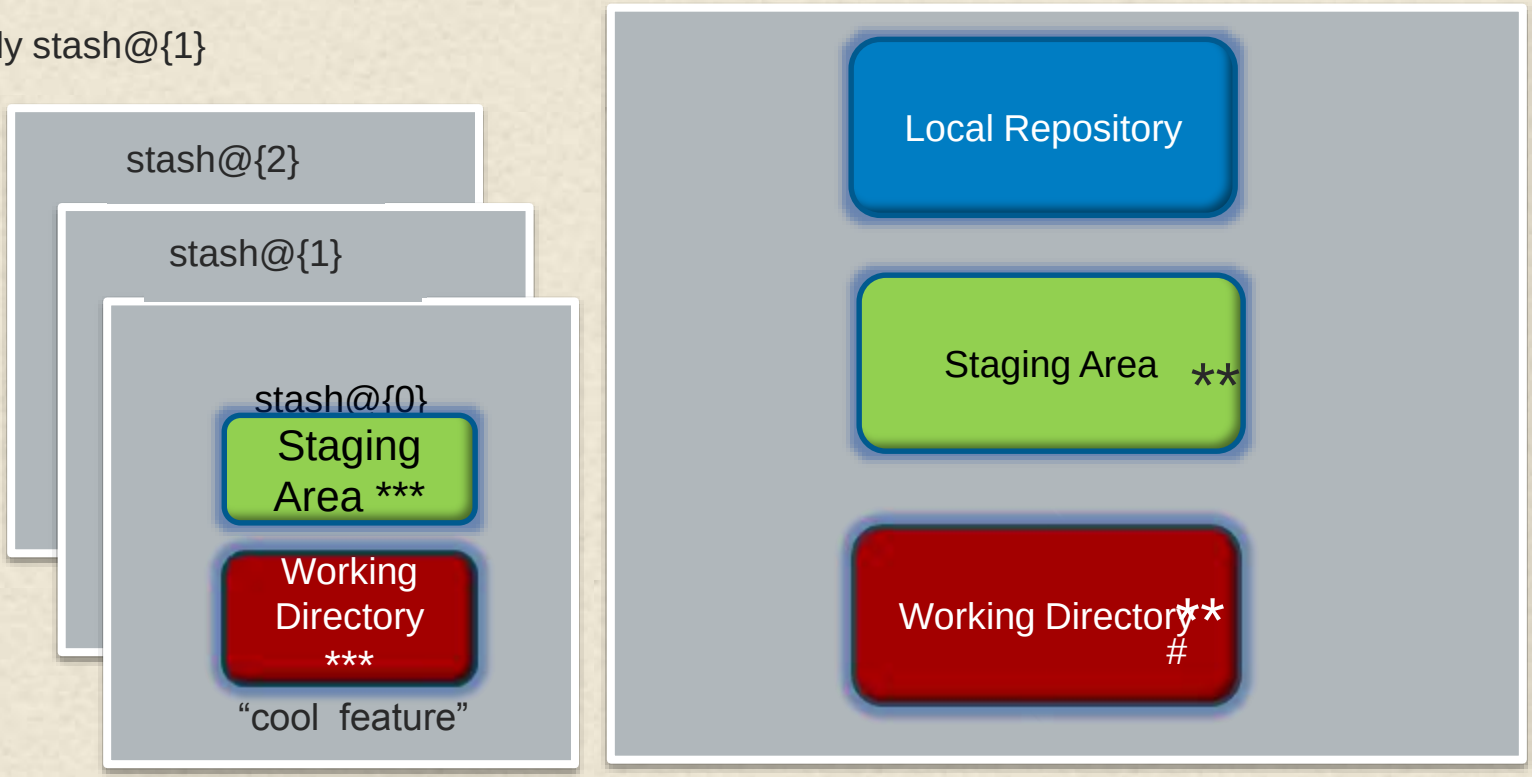
Stash

> git stash apply stash@{1}



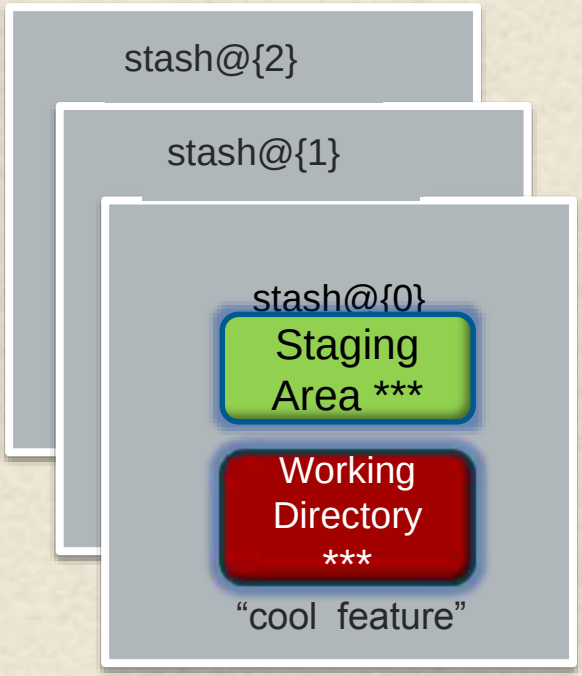
Stash

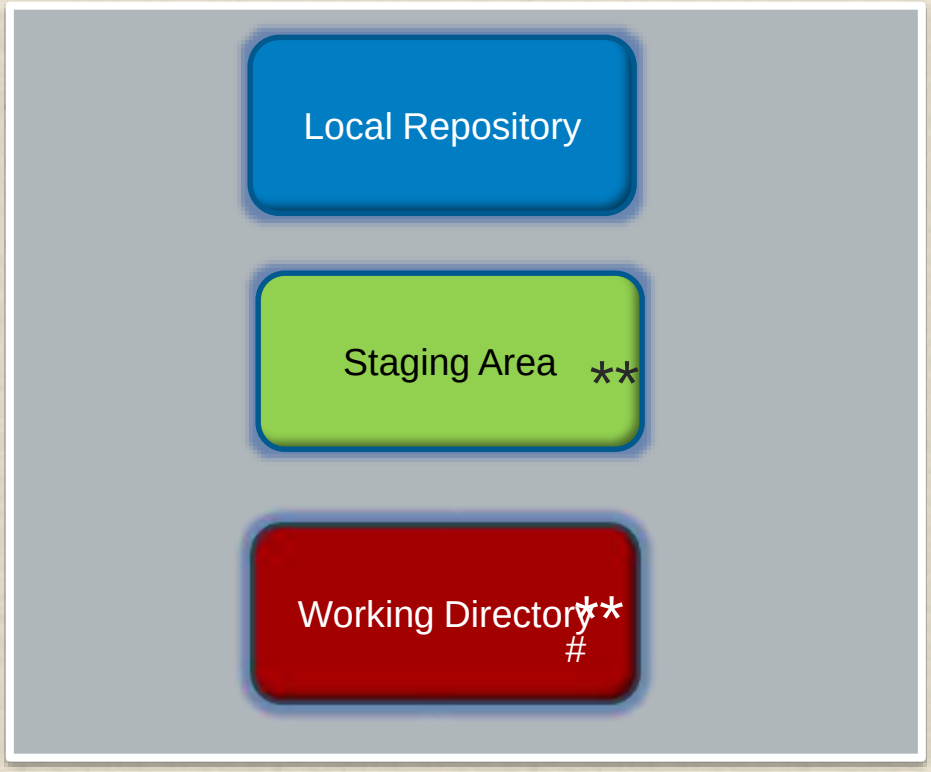
> git stash apply stash@{1}



Stash

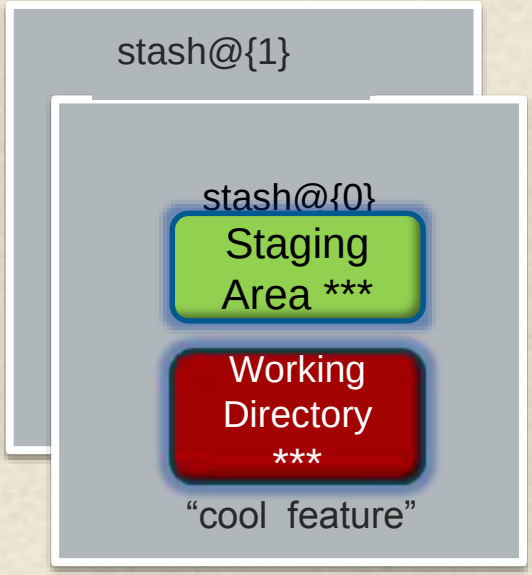
$ git stash pop stash@{2}

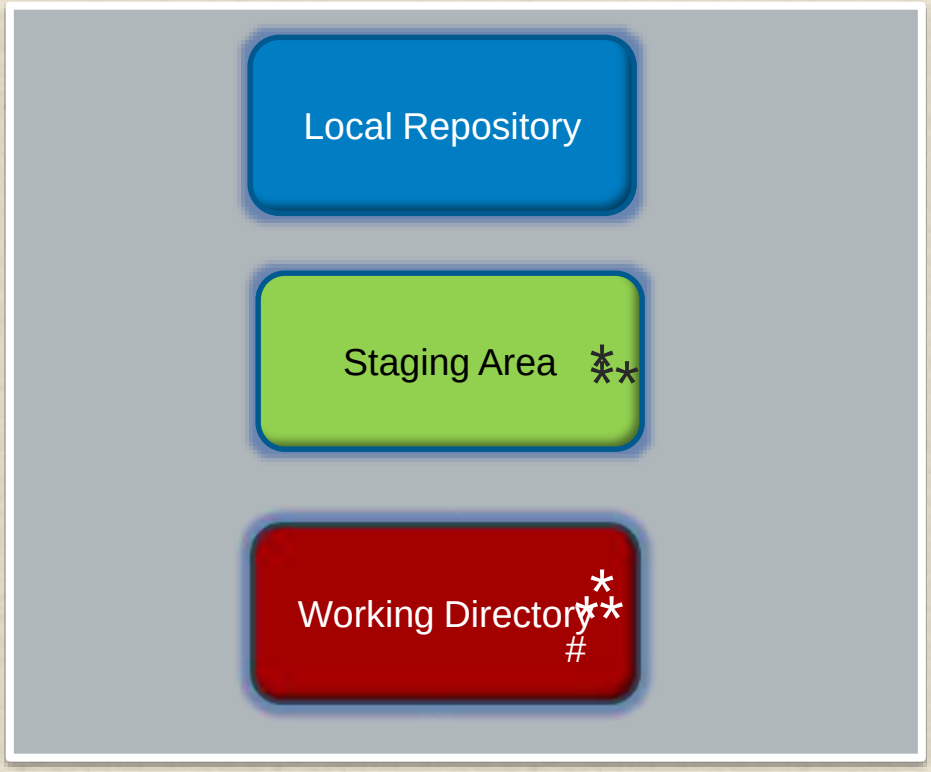




Stash

$ git stash pop stash@{2}





Command: Git Reset

- Purpose -- allow you to “roll back” so that your branch points at a previous commit ; optionally also update staging area and working directory to that commit
- Use case - you want to update your local environment back to a previous point in time; you want to overwrite or a local change you’ve made
- Syntax:

```
git reset [-q] [<tree-ish>] [--] <paths>...
git reset (--patch | -p) [<tree-ish>] [--] [<paths>...]
EXPERIMENTAL: git reset [-q] [--stdin [-z]] [<tree-ish>]
git reset [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]
```

- Warning: --hard overwrites everything

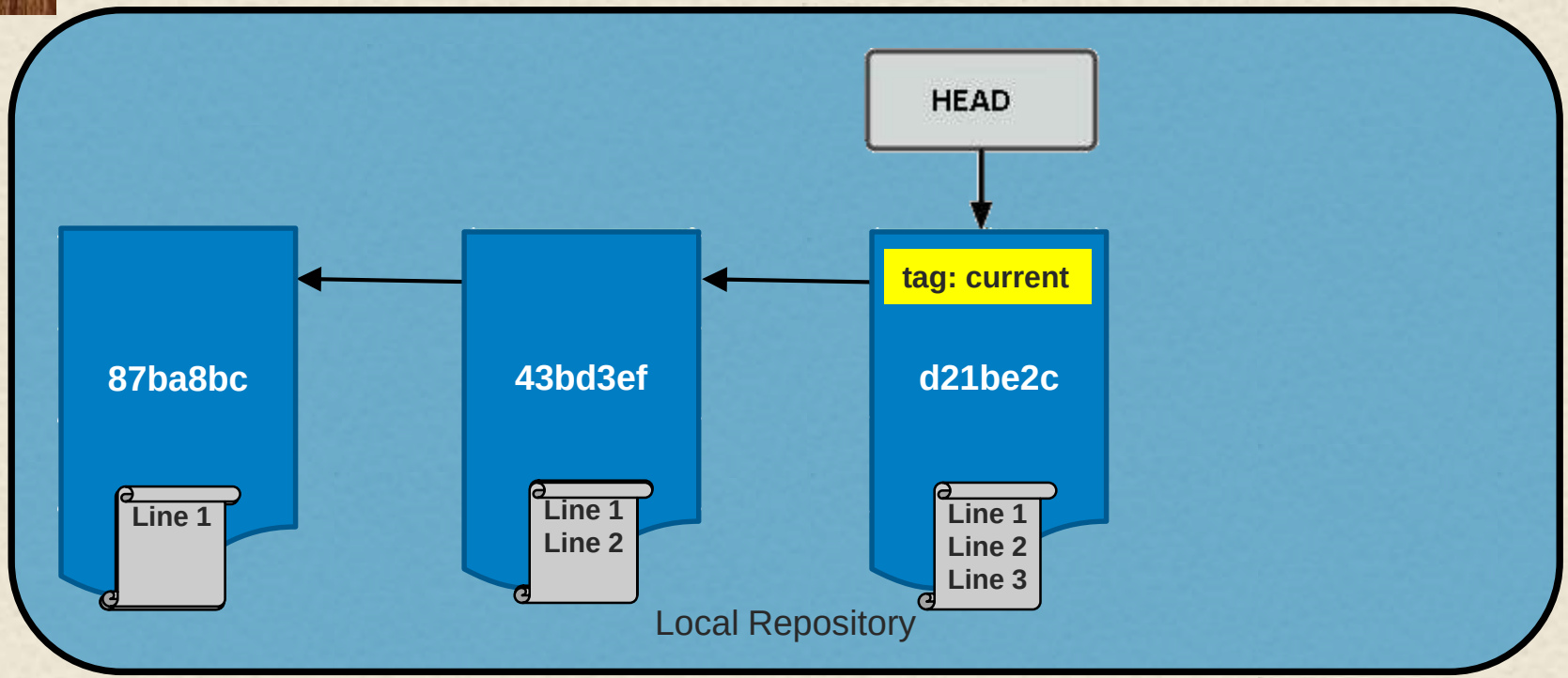
Command: Git Revert

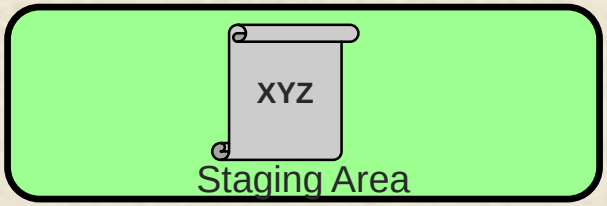
- Purpose -- allow you to “undo” by adding a new change that cancels out effects of previous one
- Use case - you want to cancel out a previous change but not roll things back
- Syntax:

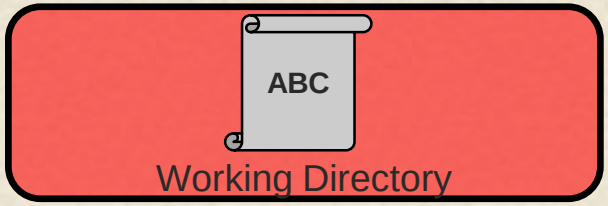
git revert [--[no-]edit] [-n] [-m parent-number] [-s] [-S[<keyid>]] <commit>...
git revert --continue
git revert --quit
git revert --abort

- Note: The net result of using this command vs. reset can be the same. If so, and content that is being reset/revert has been pushed such that others may be consuming it, preference is for revert.

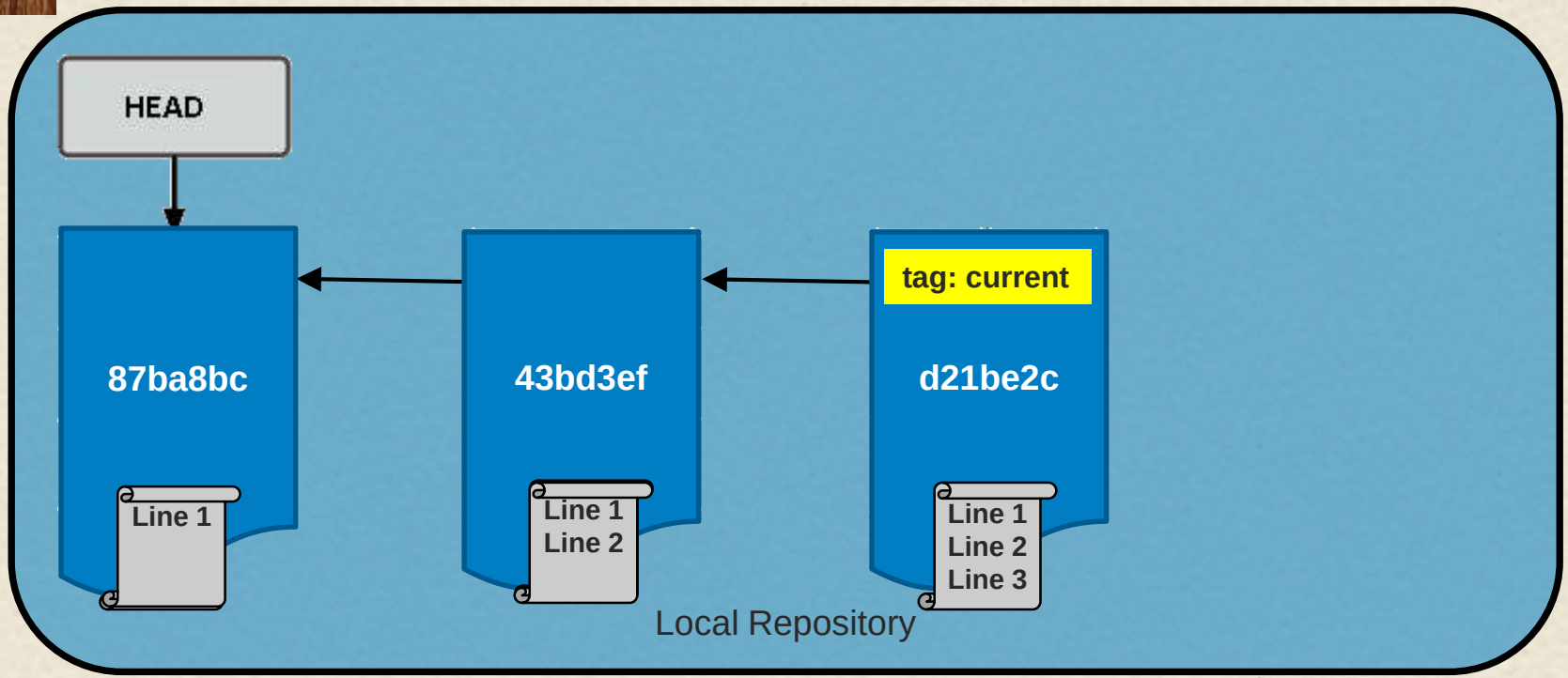
Reset and Revert



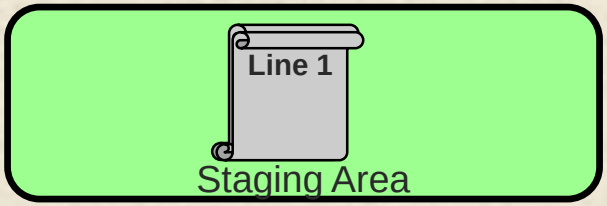


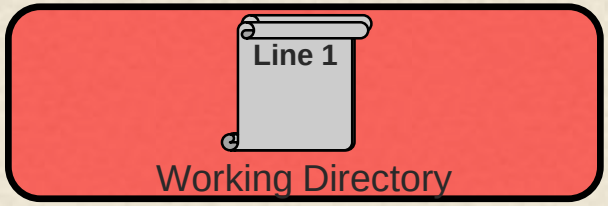


Reset and Revert

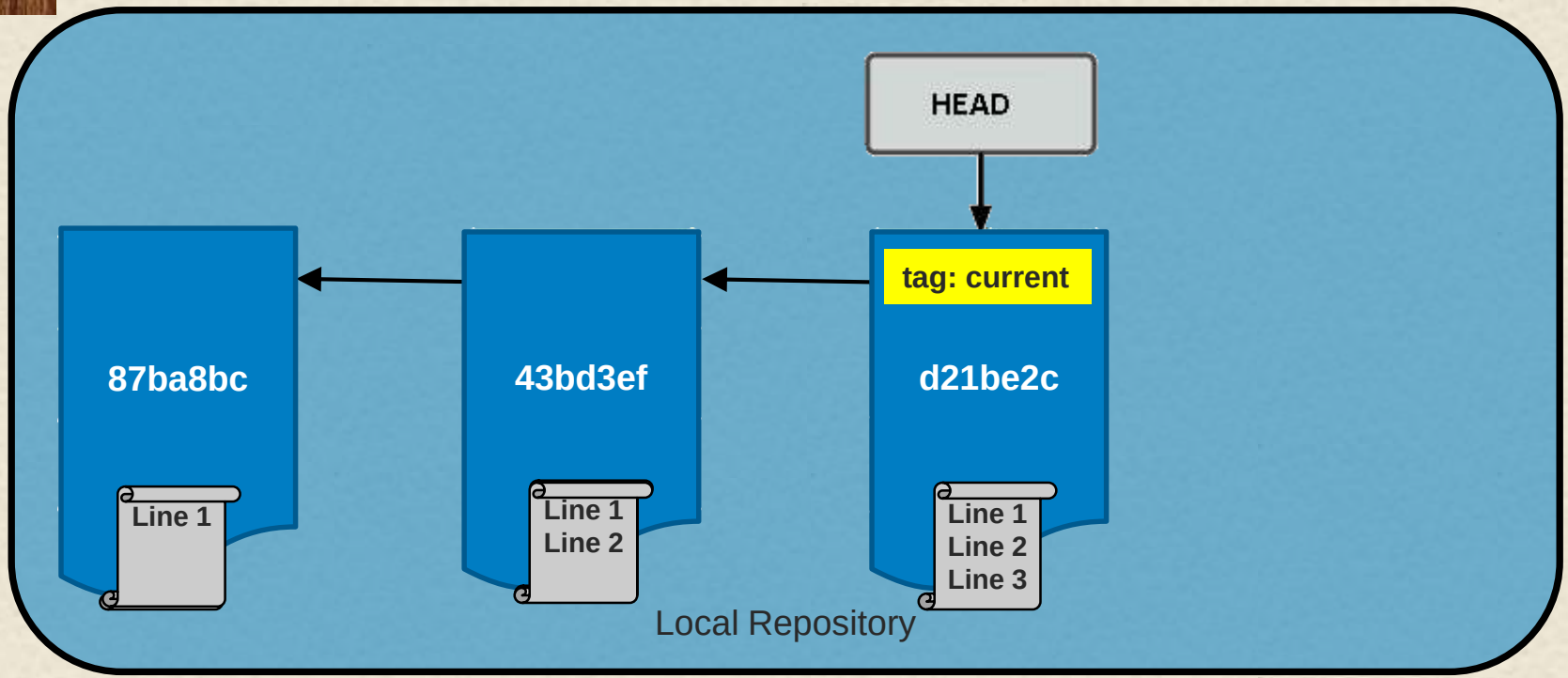


git reset --hard 87ba8bc

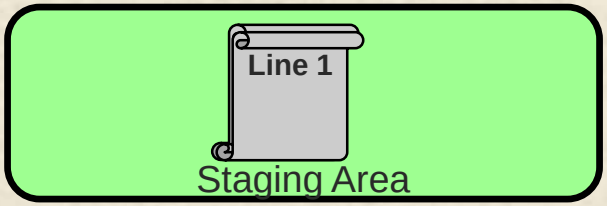


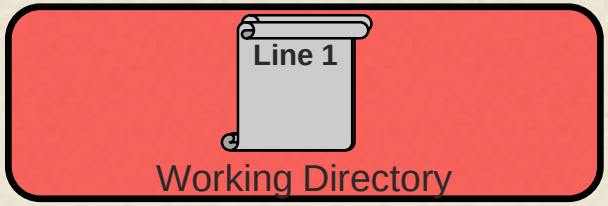


Reset and Revert

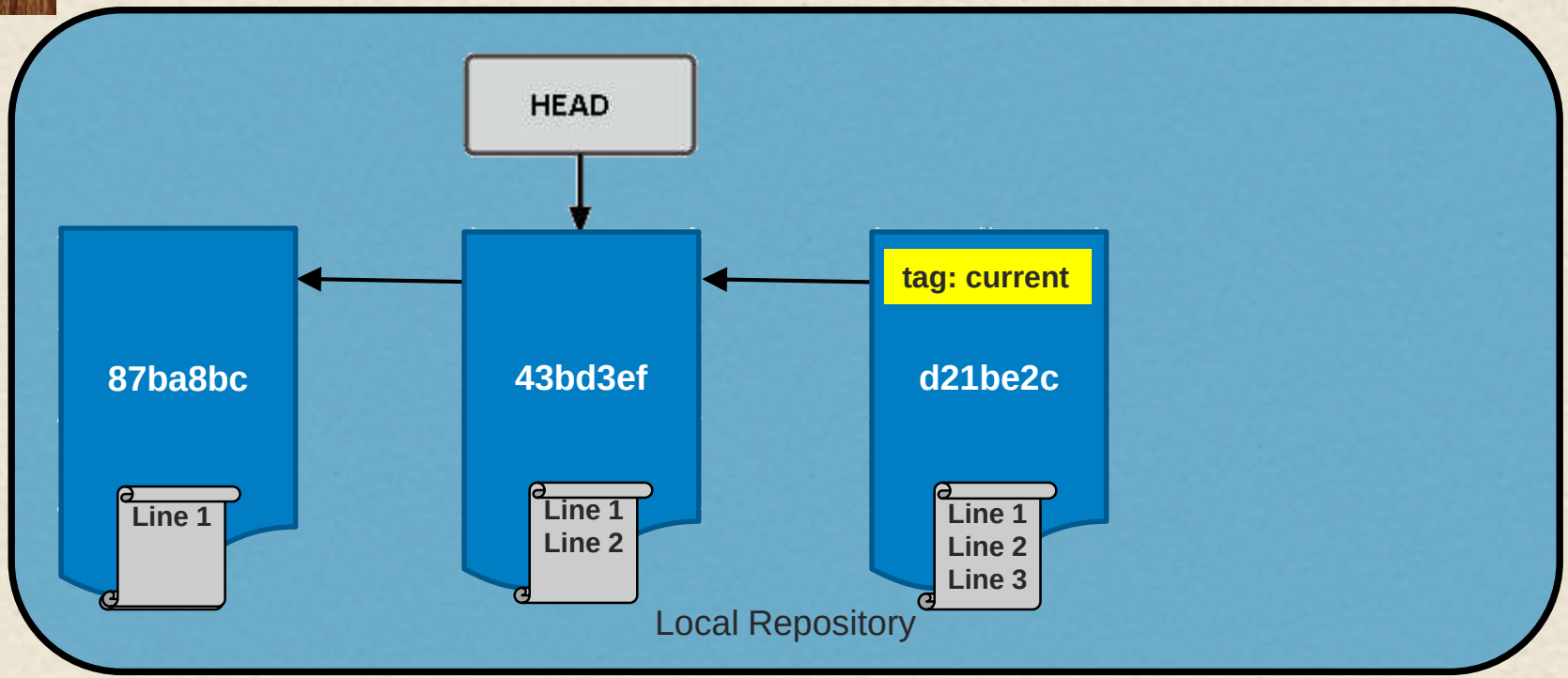


git reset --hard 87ba8bc

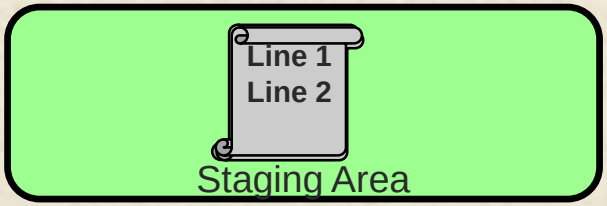


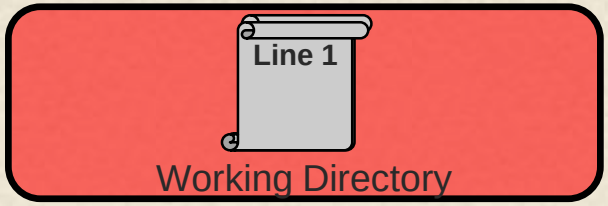


Reset and Revert

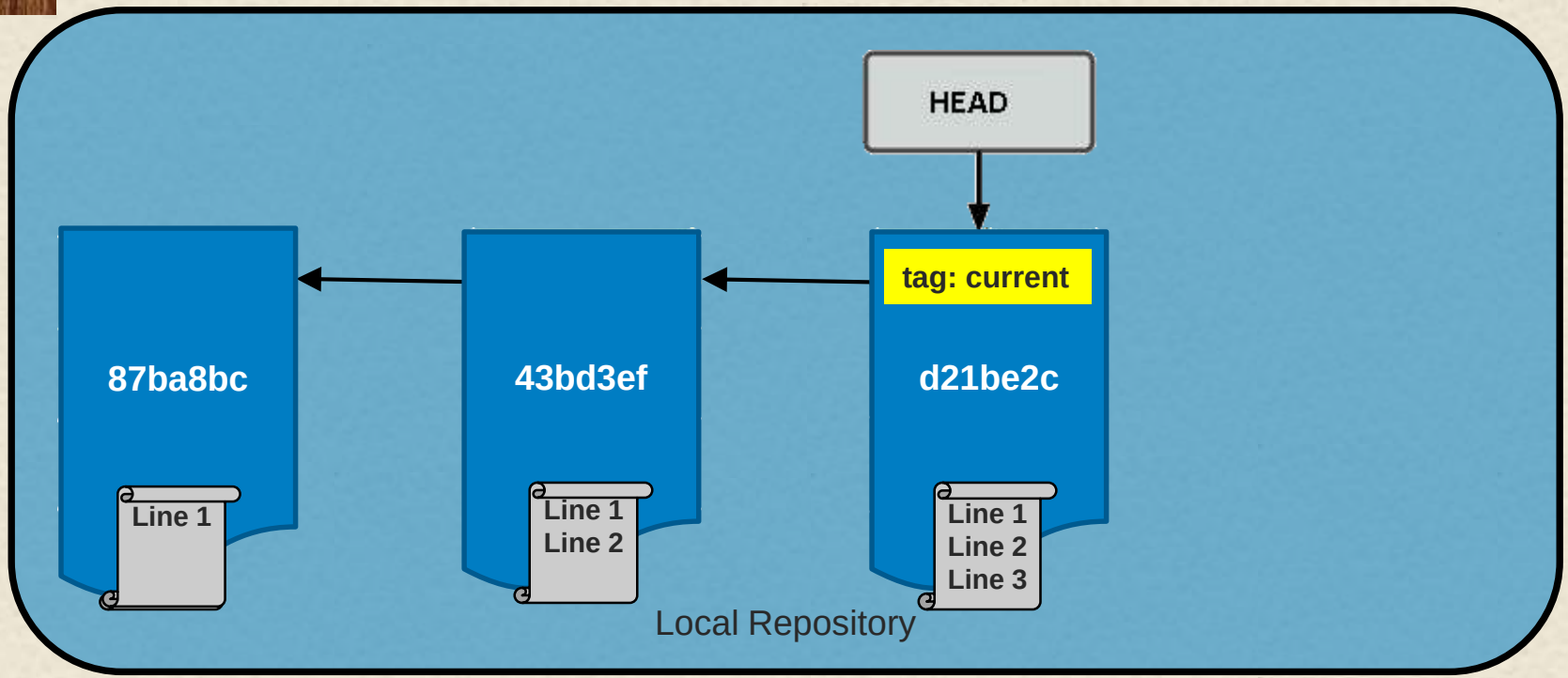


git reset current~1 [--mixed]

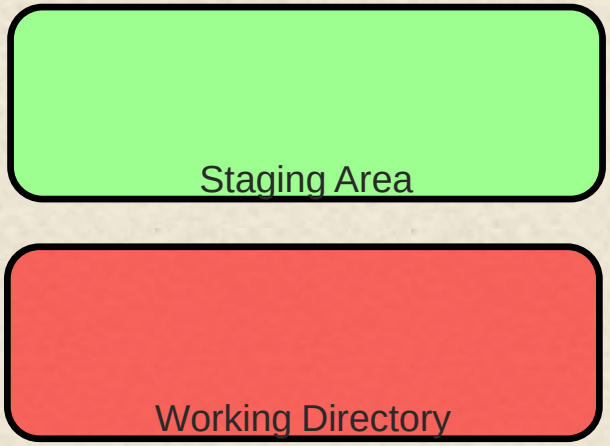




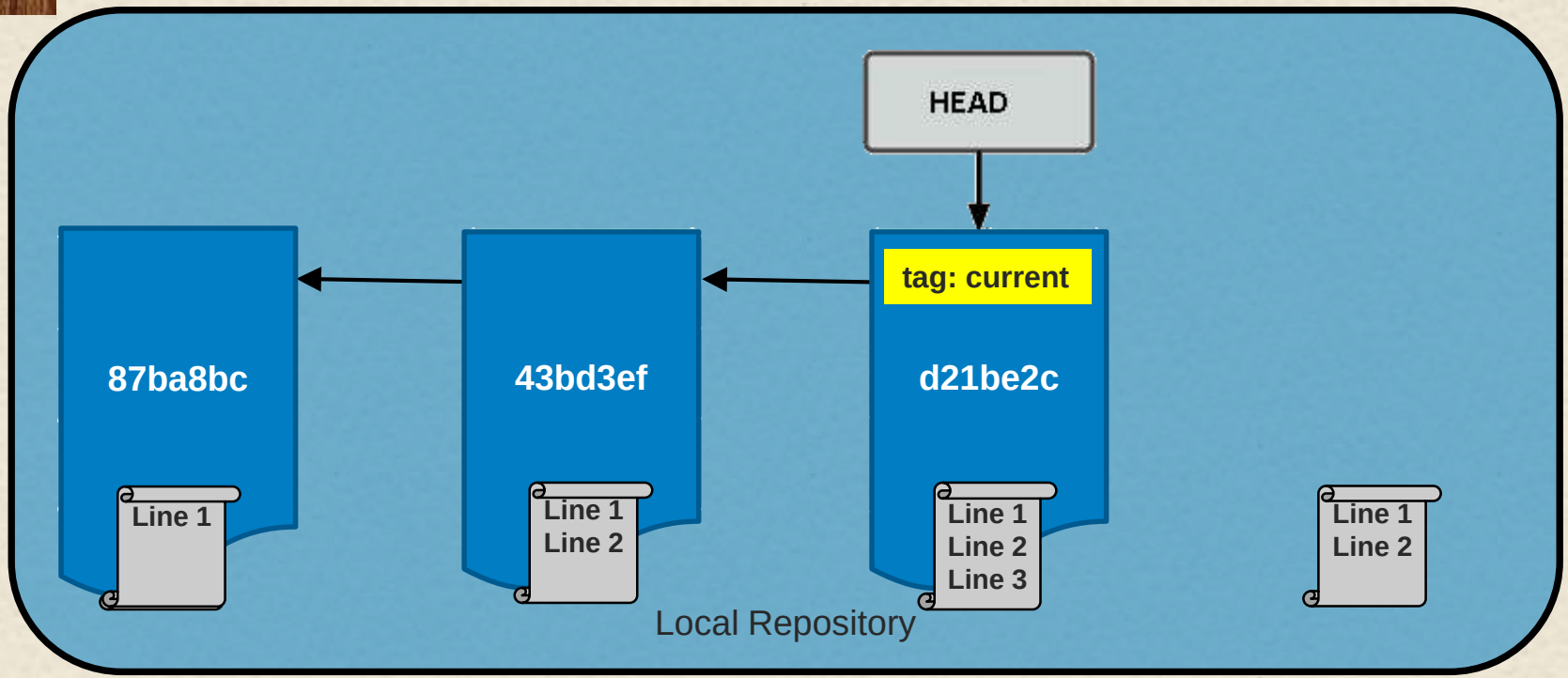
Reset and Revert

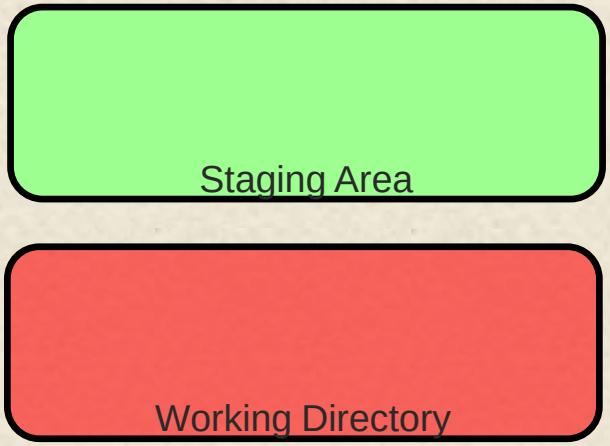


git reset current~1 [--mixed]



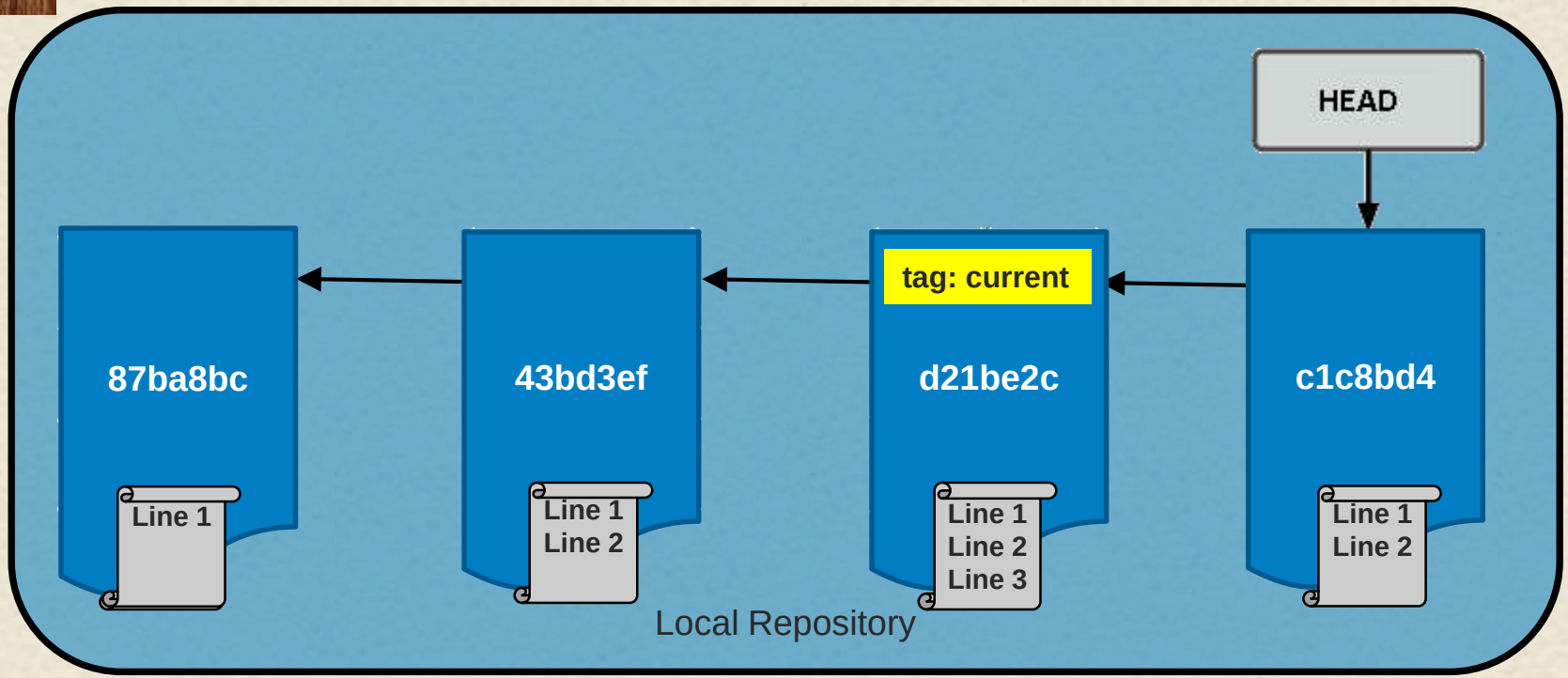
Reset and Revert

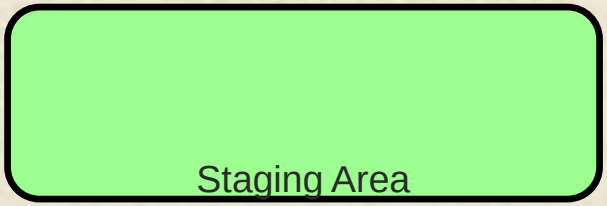


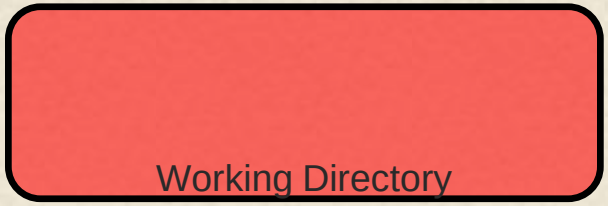


git revert HEAD~1

Reset and Revert

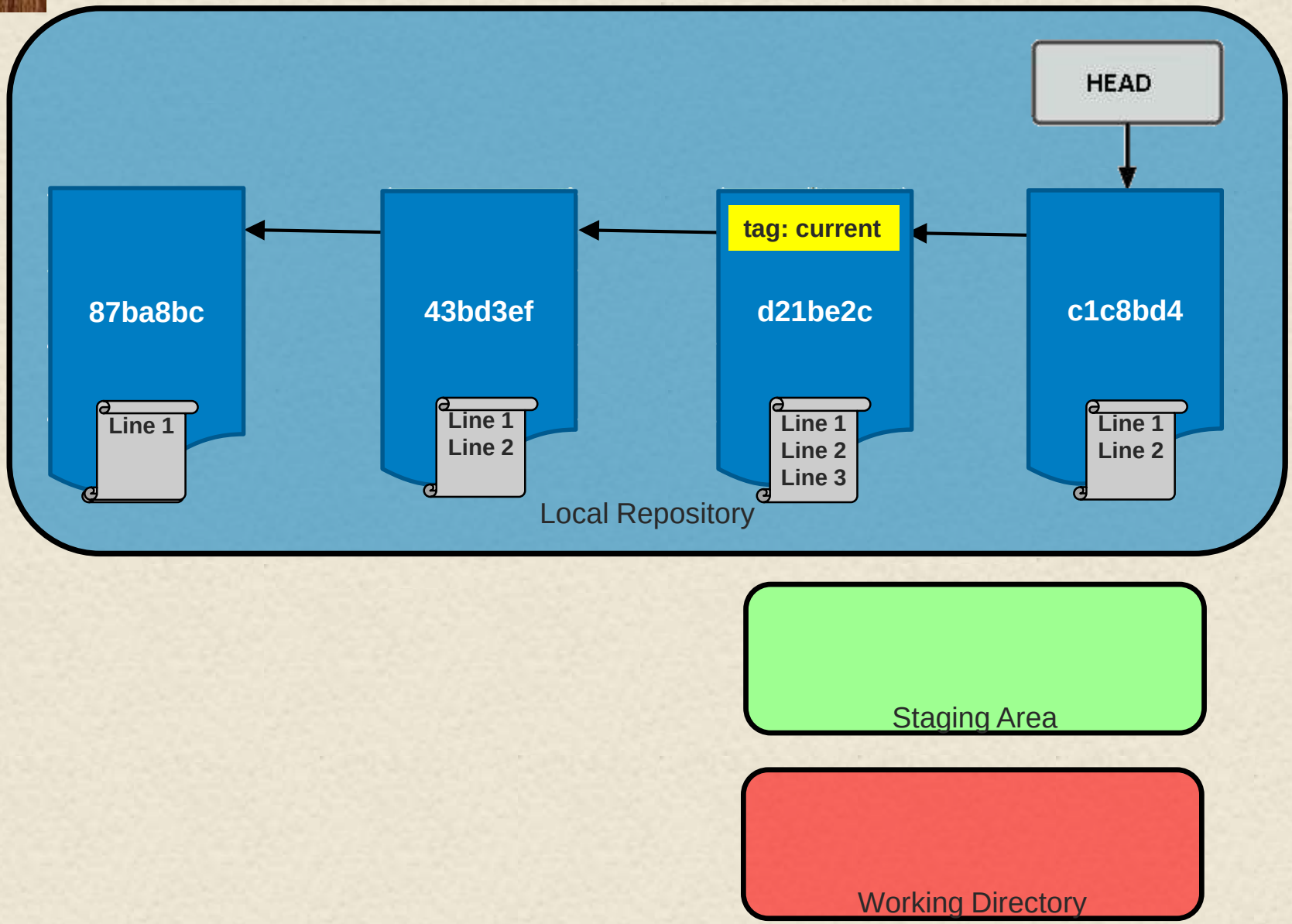






git revert HEAD~1

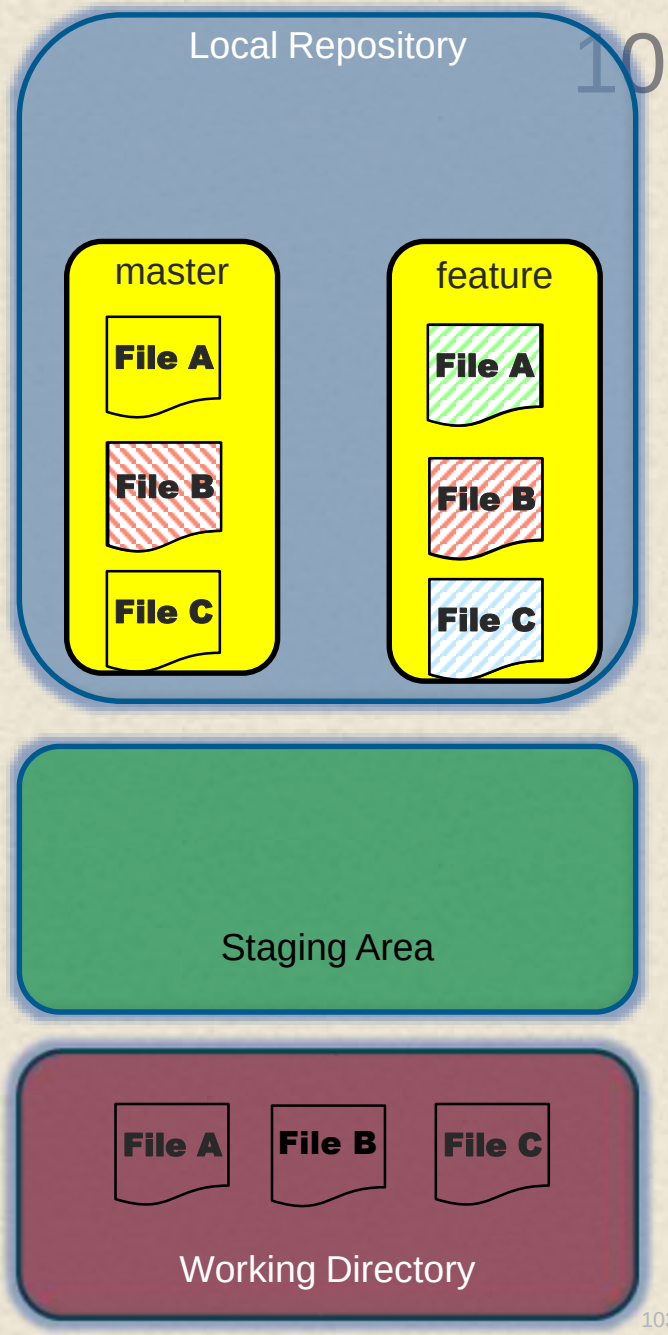
Reset and Revert



Command: Git Rerere (Reuse Recorded Resolution)

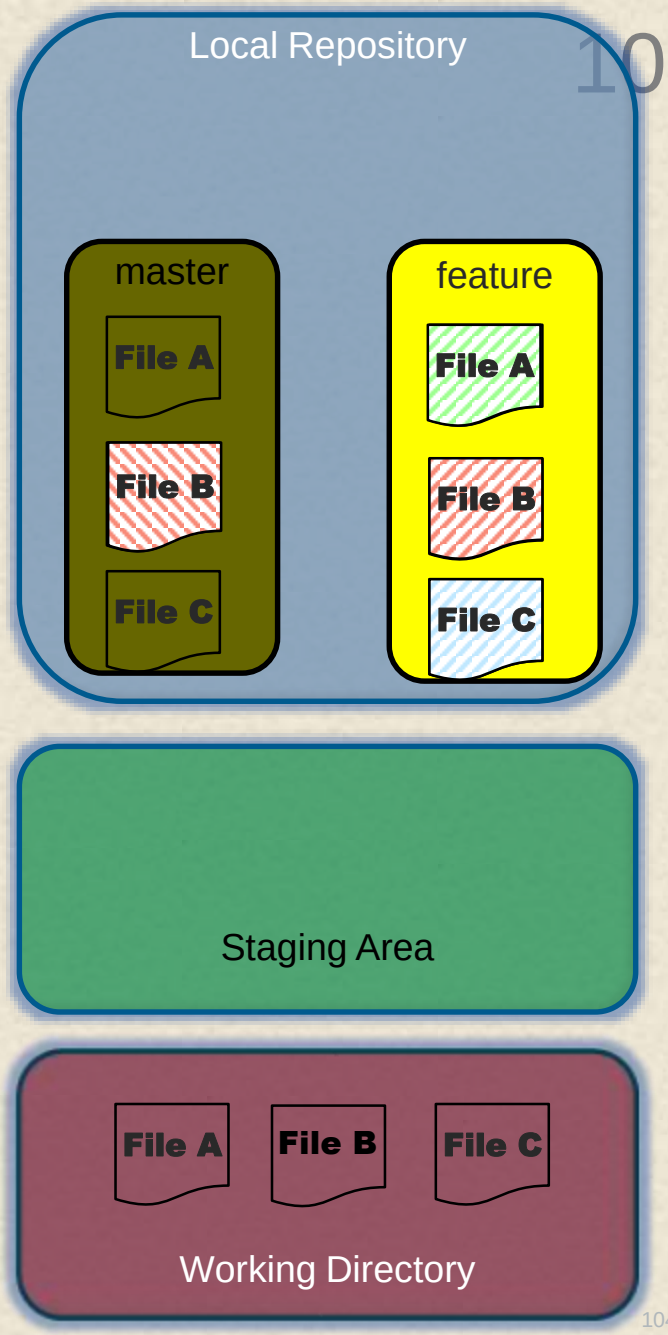
- Purpose -- allows recording of how you solve a merge situation and then can automatically resolve the same situation in the same way if needed later
- Use case - trial and repeated merges; merging newer versions of a branch with the same conflicts into a newer one periodically; resolve conflicts after reset or revert; applicable to any repeated merge case: rebase, merge
- Syntax: *git rerere* [*clear*|*forget* <pathspec>|*diff*|*remaining*|*status*|*gc*]
- Note: This is a “state” command. Enabled by turning on a state in Git, rather than just running a command
  - **Enabled via** *git config --global rerere.enabled 1*
    - » **Then runs automatically**
  - **Invoked directly via** *git rerere* **for related commands or options**

Git Rerere 1



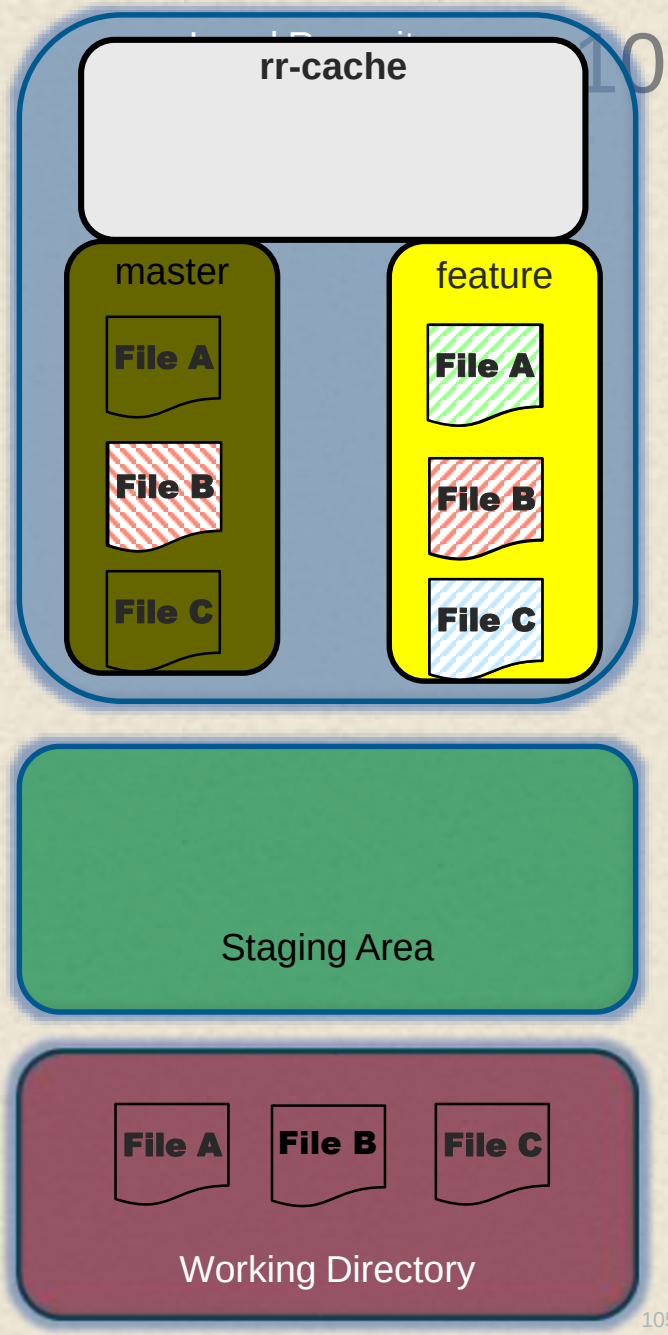
Git Rerere 1

$ git checkout master



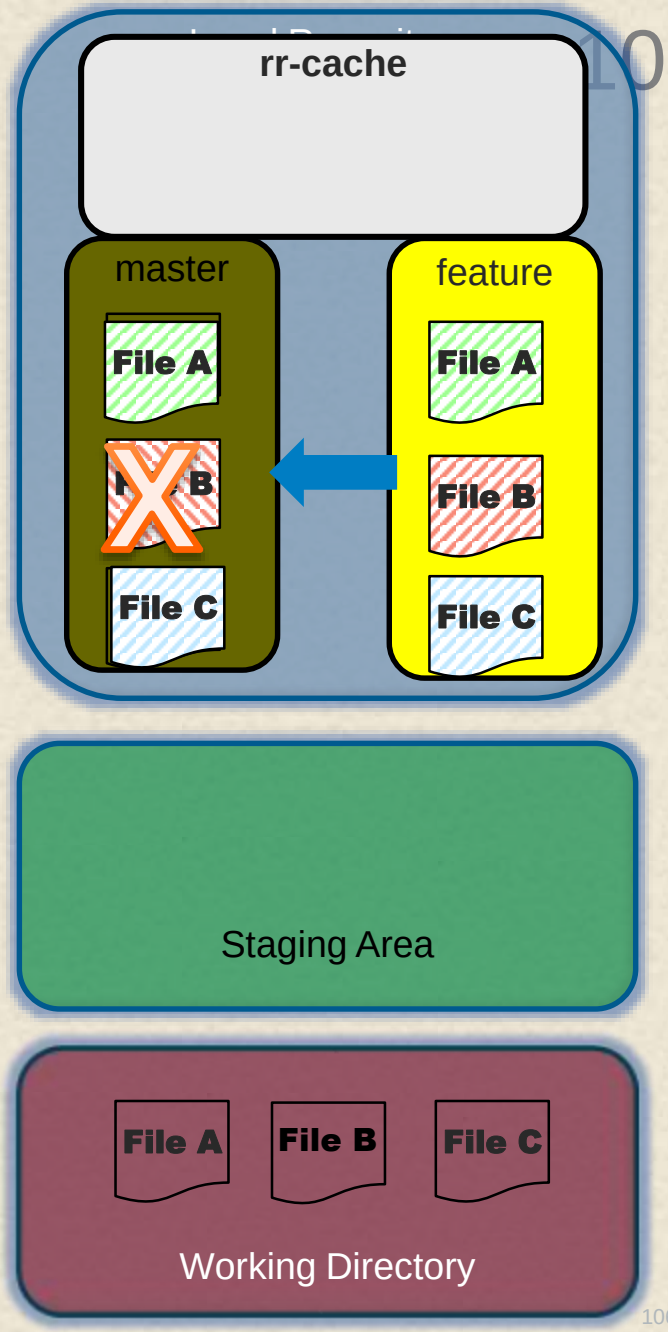
Git Rerere 1

```
$ git checkout master
$ git config --global rerere.enabled 1
```



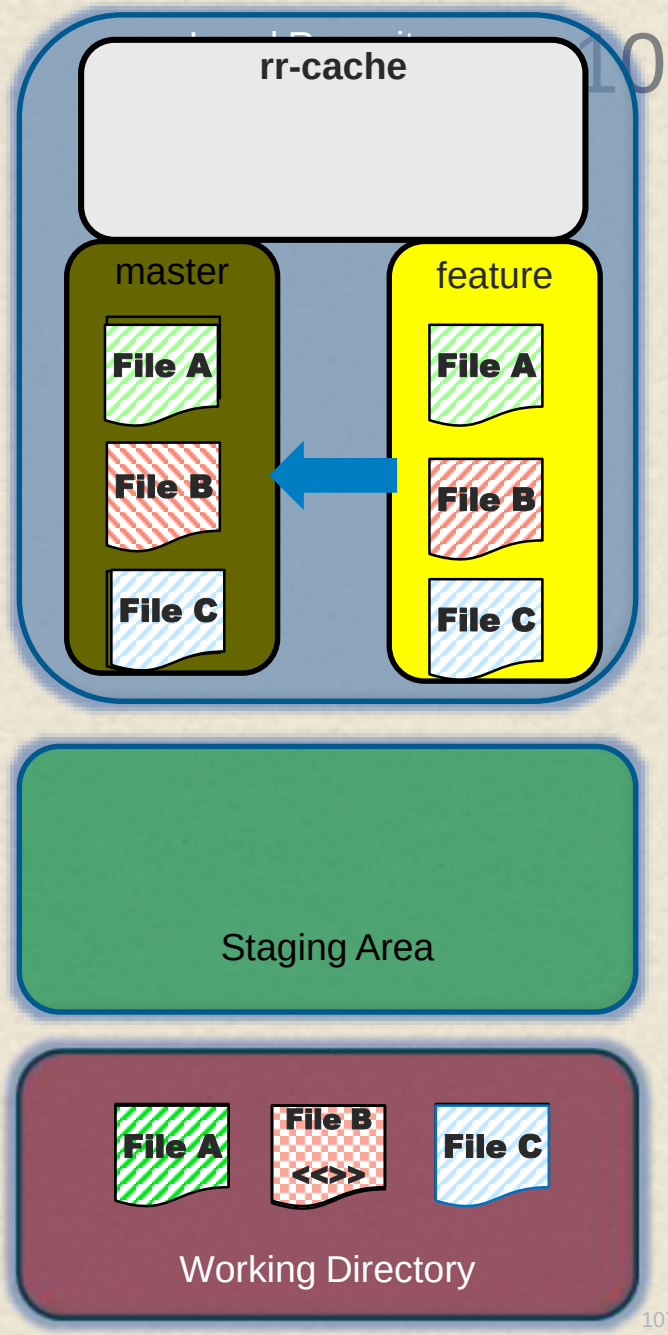
Git Rerere 1

```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
```



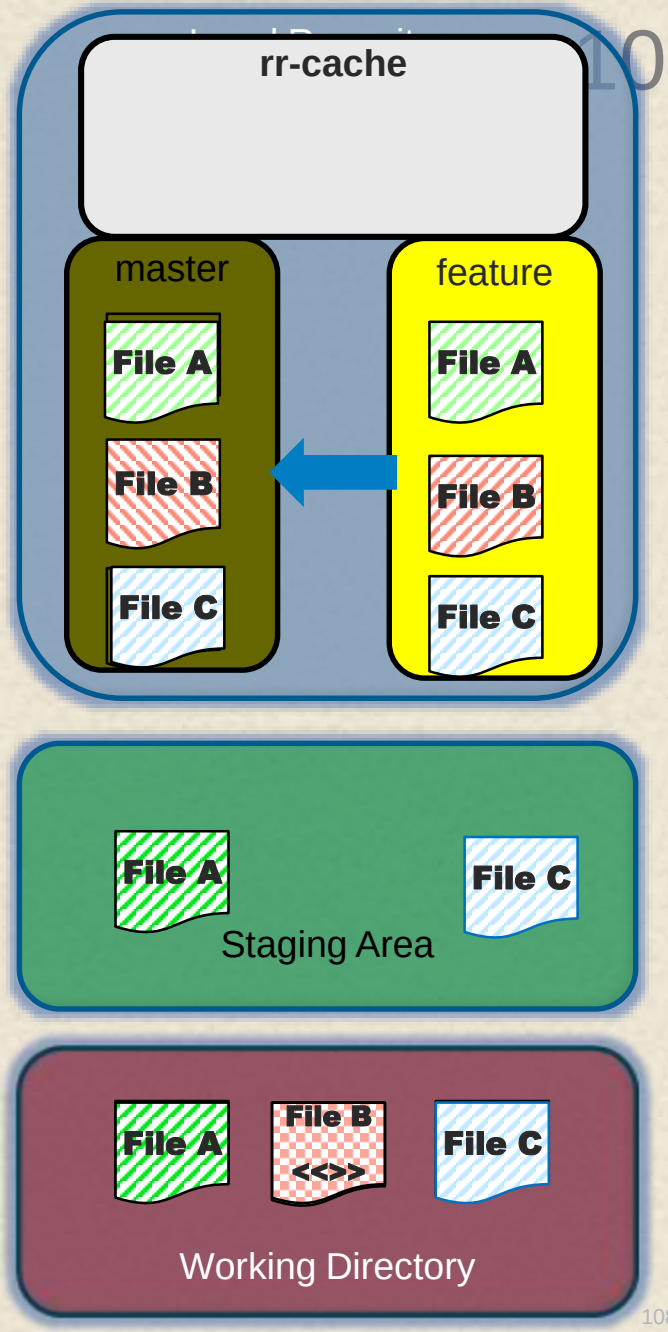
Git Rerere 1

```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
```



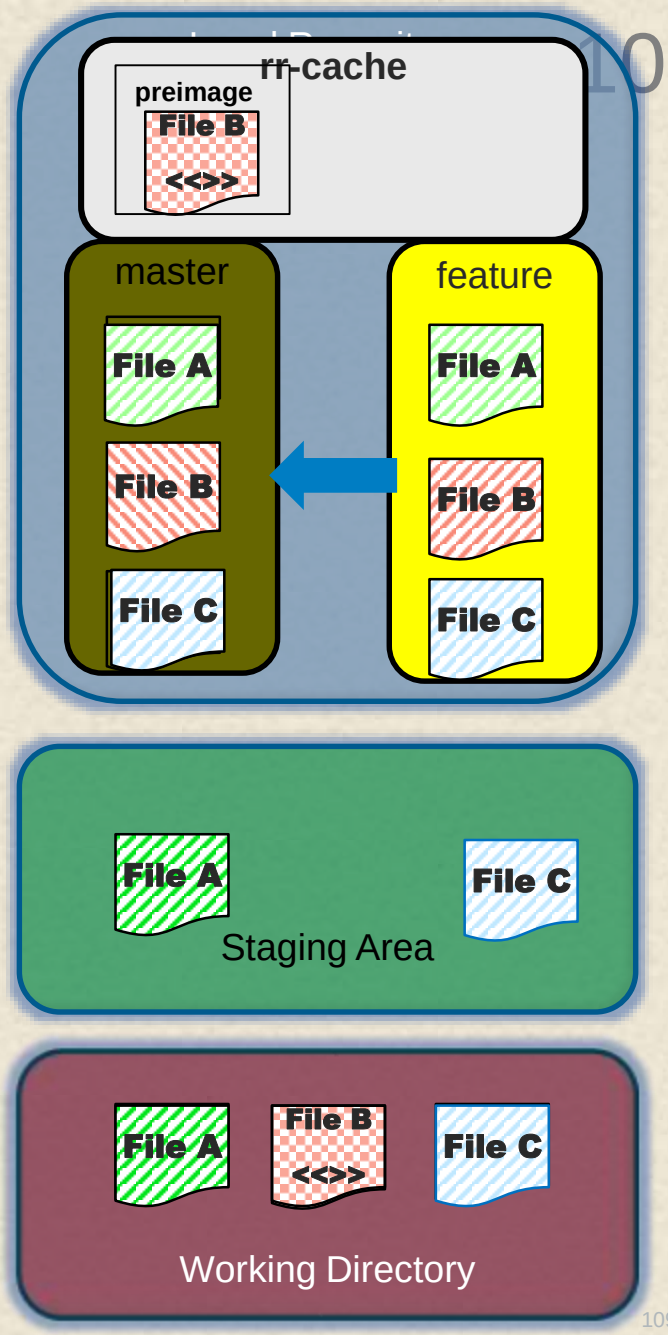
Git Rerere 1

```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
```



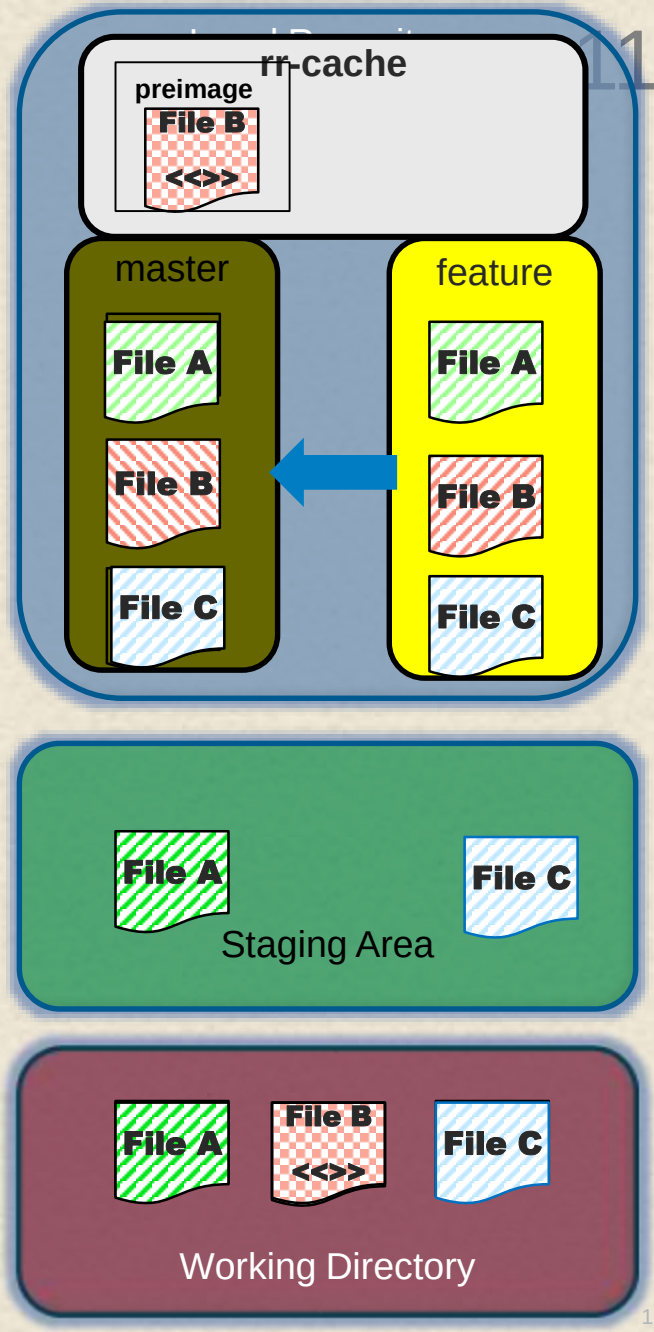
Git Rerere 1

```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
```



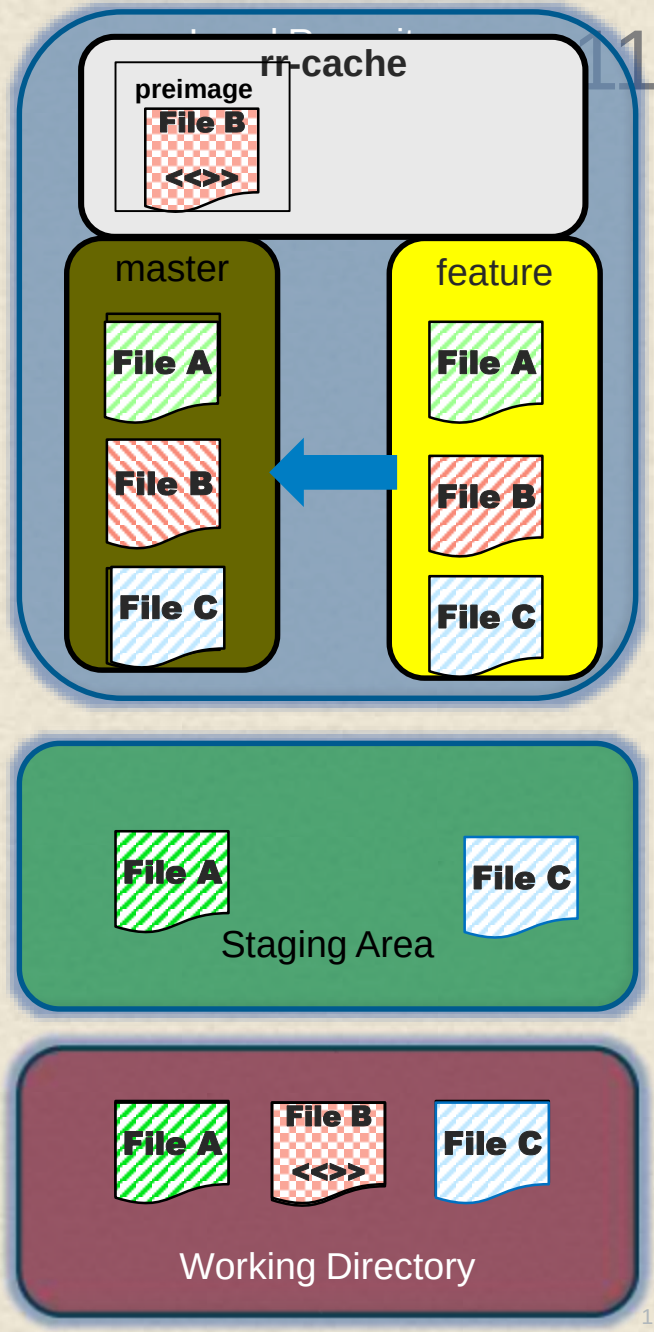
Git Rerere 1

```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
    Recorded preimage for 'File B'
```



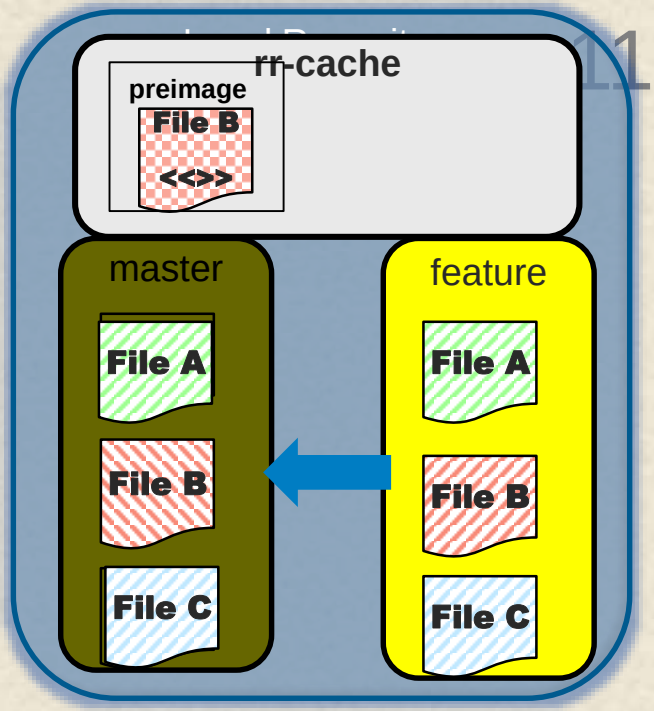
Git Rerere 1

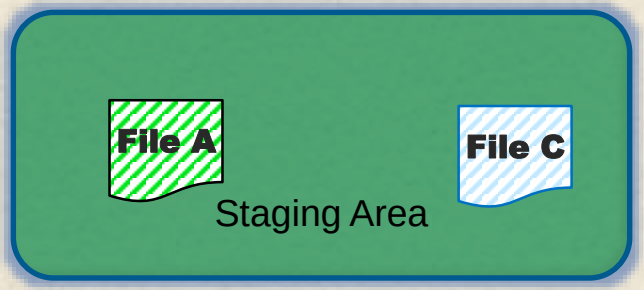
```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
    Recorded preimage for 'File B'
$ git status
    Changes to be committed:
        modified: File A
        modified: File C
    Unmerged paths
        both modified: File B
```



Git Rerere 1

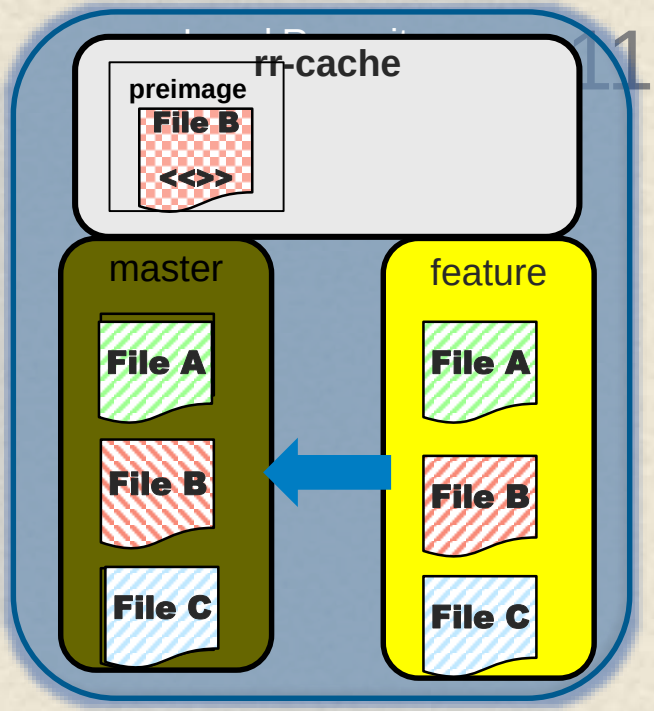
```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
    Recorded preimage for 'File B'
$ git status
    Changes to be committed:
        modified: File A
        modified: File C
    Unmerged paths
        both modified: File B
[fix conflicts]
```





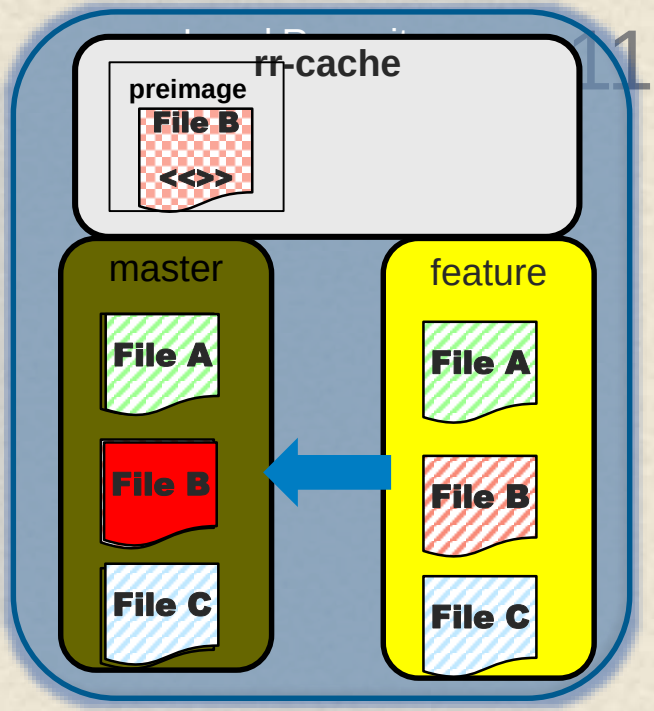
Git Rerere 1

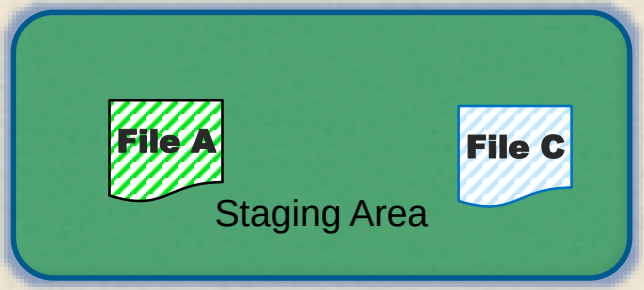
```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
    Recorded preimage for 'File B'
$ git status
    Changes to be committed:
        modified: File A
        modified: File C
    Unmerged paths
        both modified: File B
[fix conflicts]
$ git add .
```



Git Rerere 1

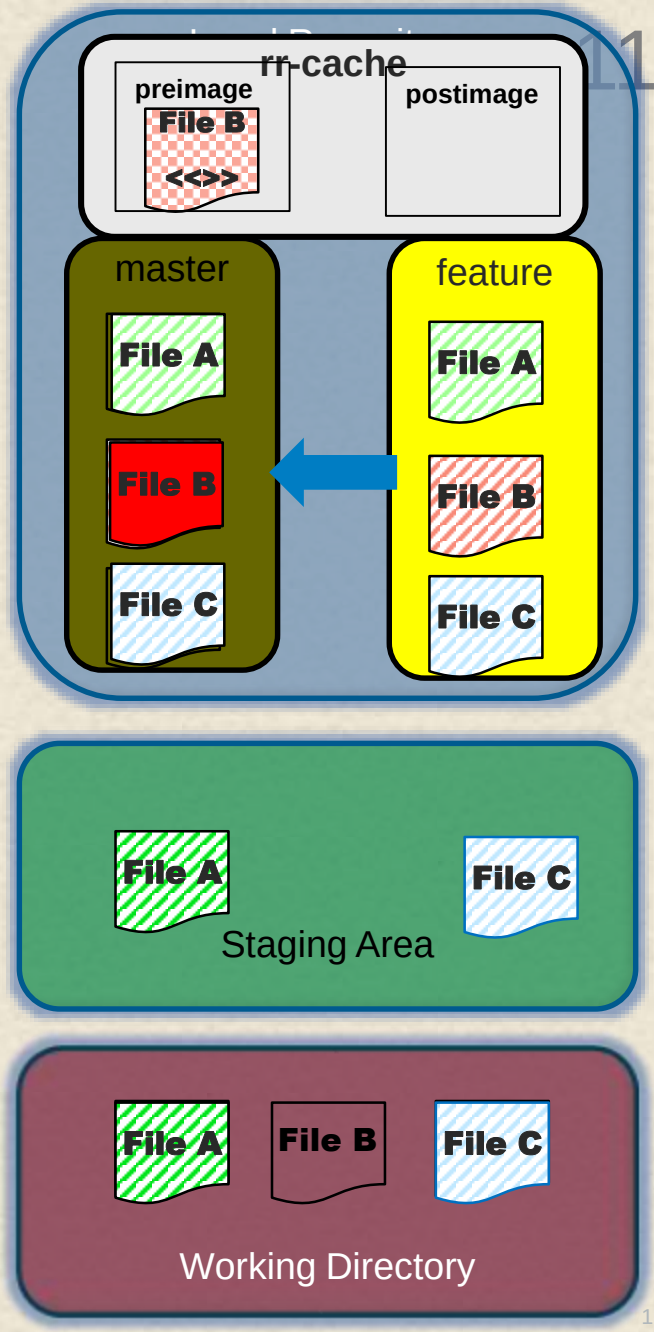
```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
    Recorded preimage for 'File B'
$ git status
    Changes to be committed:
        modified: File A
        modified: File C
    Unmerged paths
        both modified: File B
[fix conflicts]
$ git add .
$ git commit –m
"finalize merge"
```





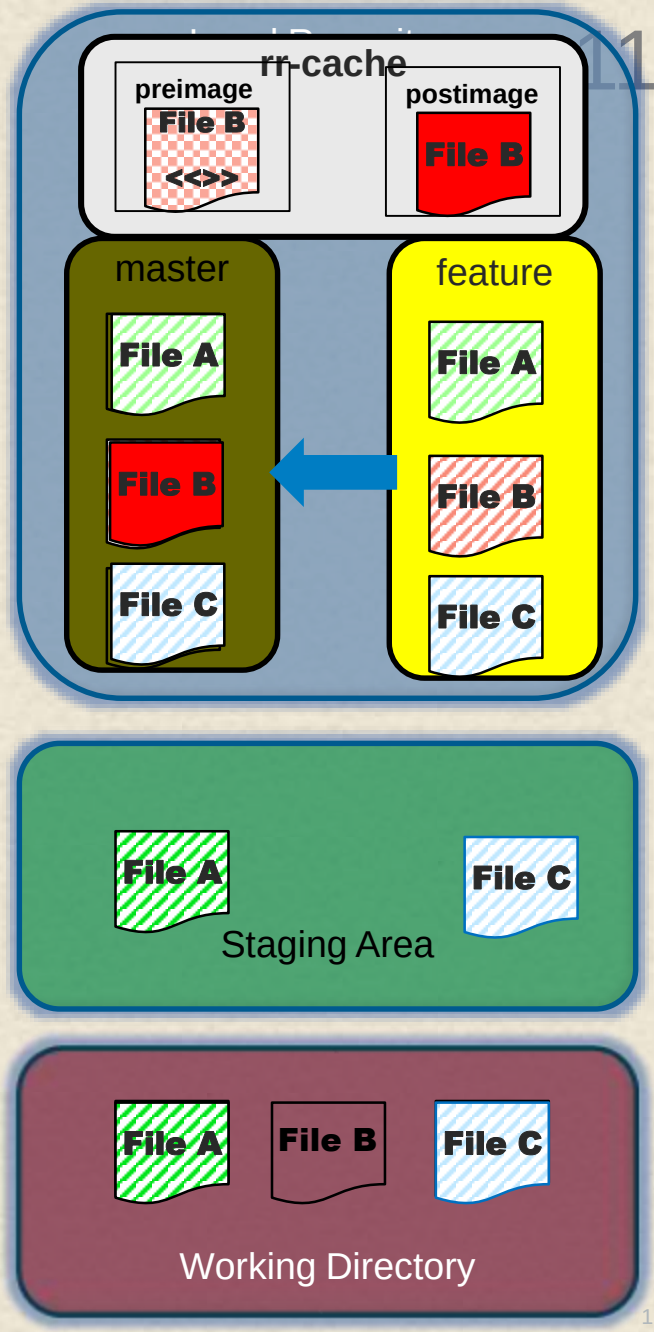
Git Rerere 1

```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
    Recorded preimage for 'File B'
$ git status
    Changes to be committed:
        modified: File A
        modified: File C
    Unmerged paths
        both modified: File B
[fix conflicts]
$ git add .
$ git commit –m
"finalize merge"
```



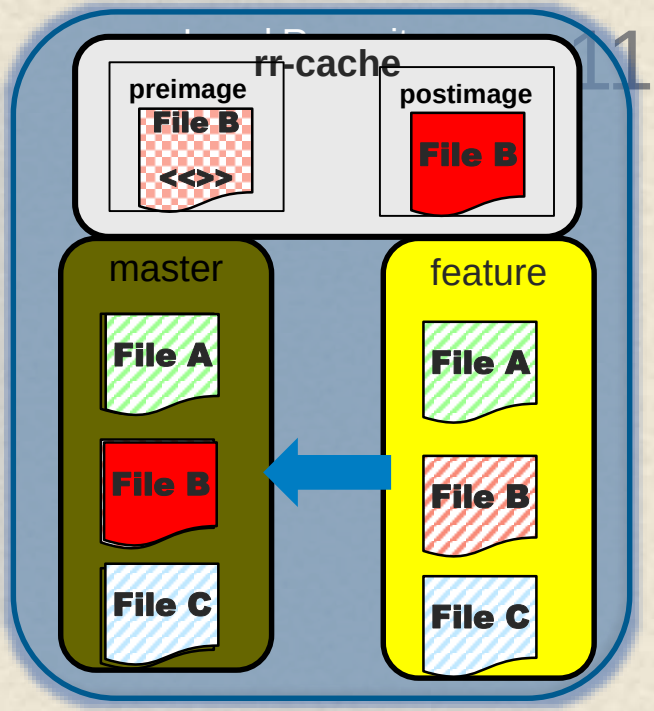
Git Rerere 1

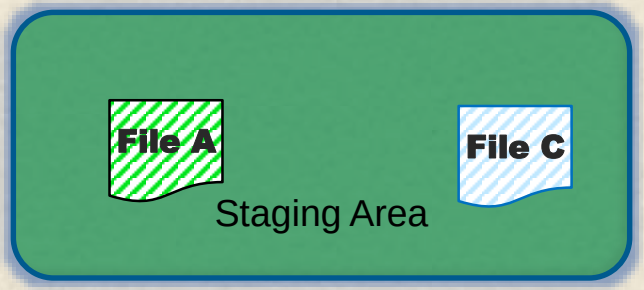
```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
    Recorded preimage for 'File B'
$ git status
    Changes to be committed:
        modified: File A
        modified: File C
    Unmerged paths
        both modified: File B
[fix conflicts]
$ git add .
$ git commit –m
"finalize merge"
```



Git Rerere 1

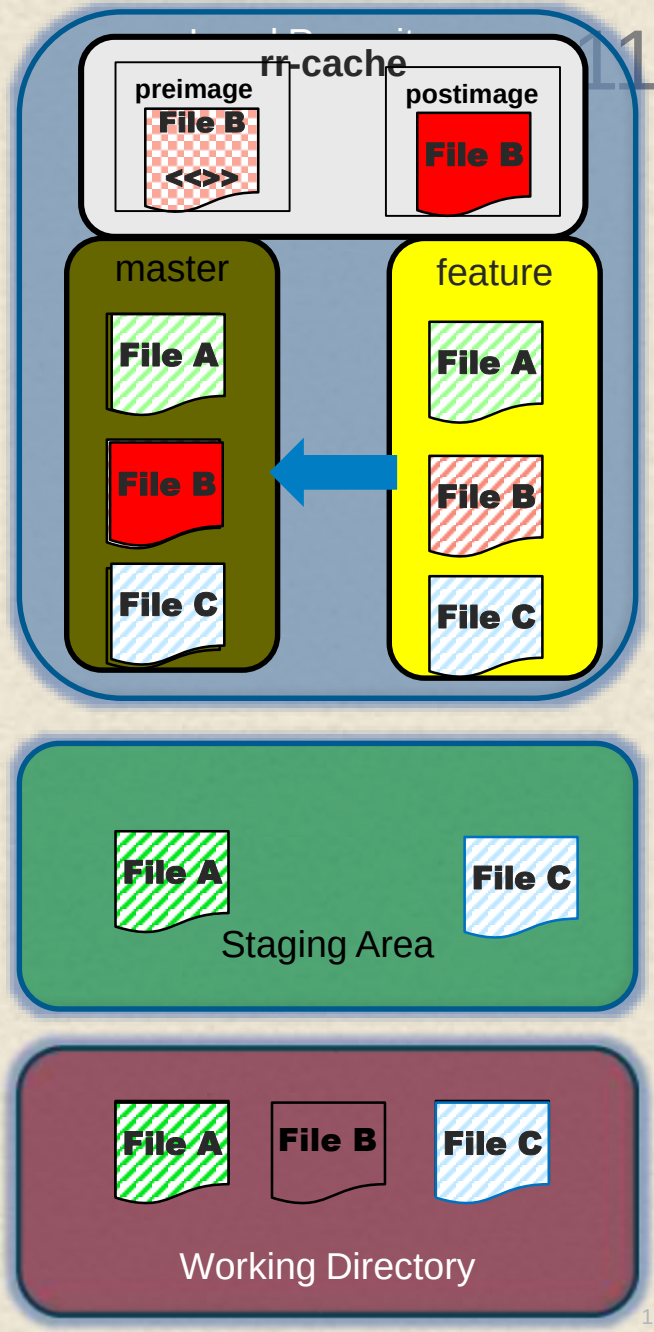
```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
    Recorded preimage for 'File B'
$ git status
    Changes to be committed:
        modified: File A
        modified: File C
    Unmerged paths
        both modified: File B
[fix conflicts]
$ git add .
$ git commit –m
"finalize merge"
    Recorded resolution for 'File B'
```





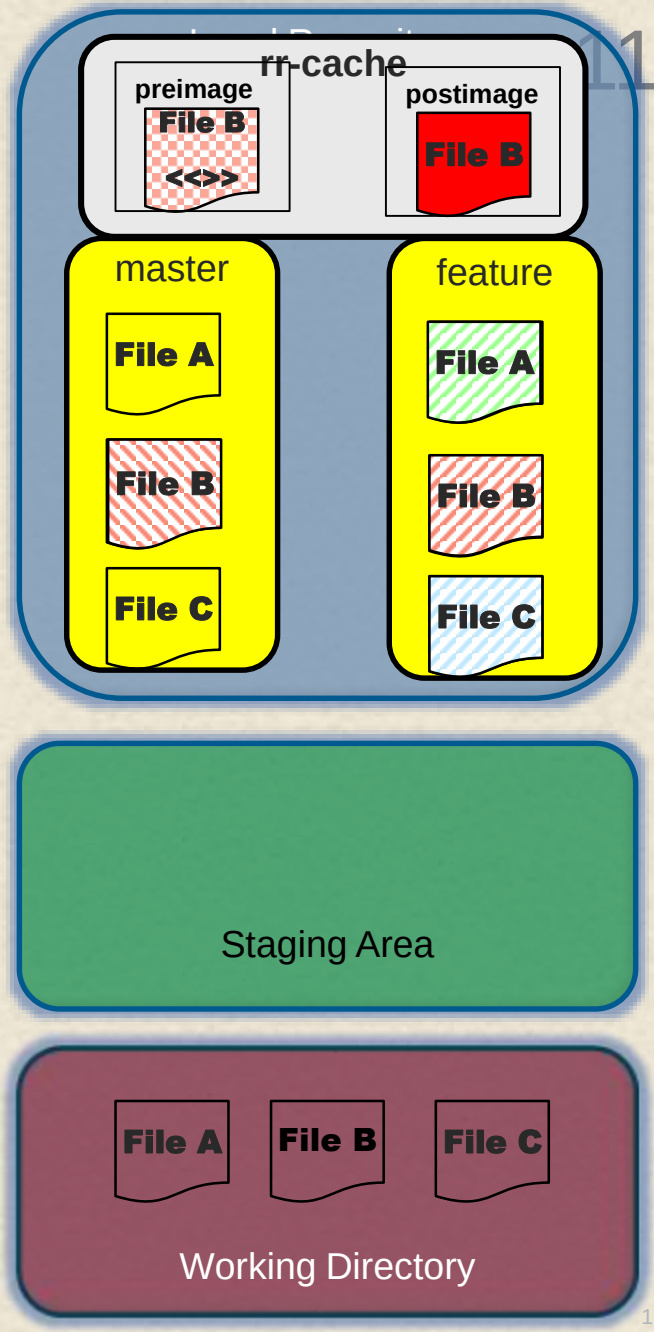
Git Rerere 1

```
$ git checkout master
$ git config --global rerere.enabled 1
$ git merge feature
    Recorded preimage for 'File B'
$ git status
    Changes to be committed:
        modified: File A
        modified: File C
    Unmerged paths
        both modified: File B
[fix conflicts]
$ git add .
$ git commit –m
"finalize merge"
    Recorded resolution for 'File B'
```



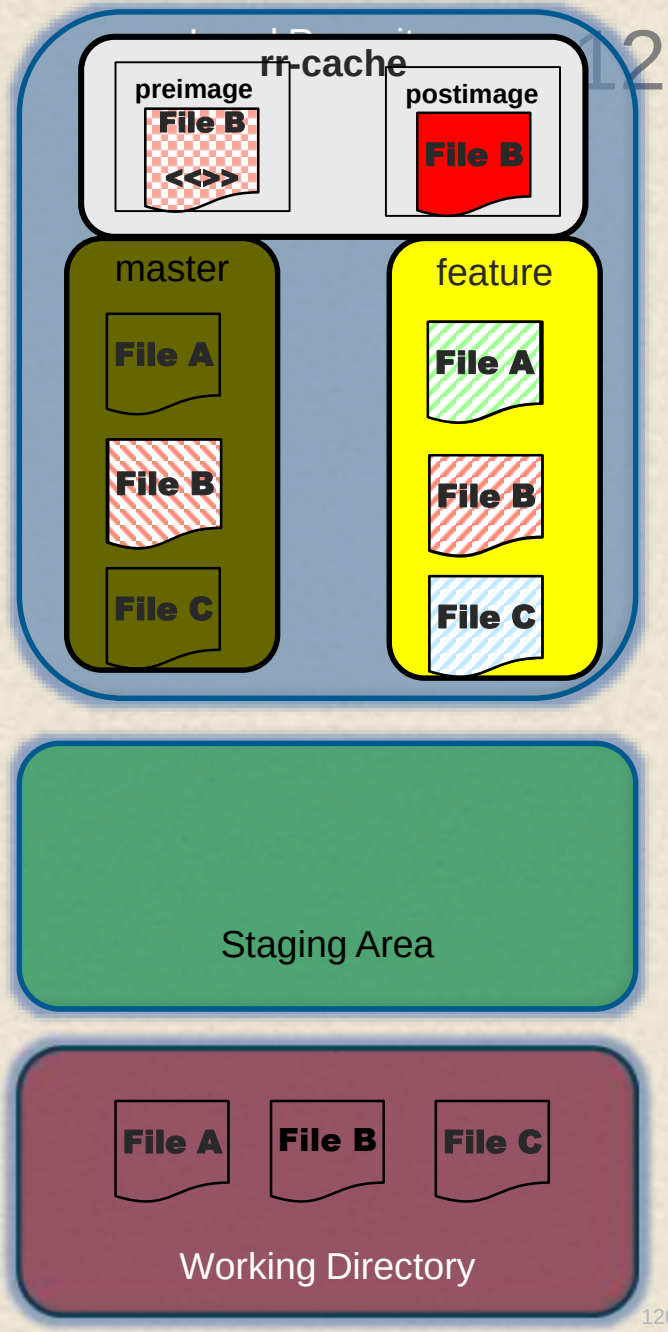
Git Rerere 2

$ git reset --hard HEAD~1



Git Rerere 2

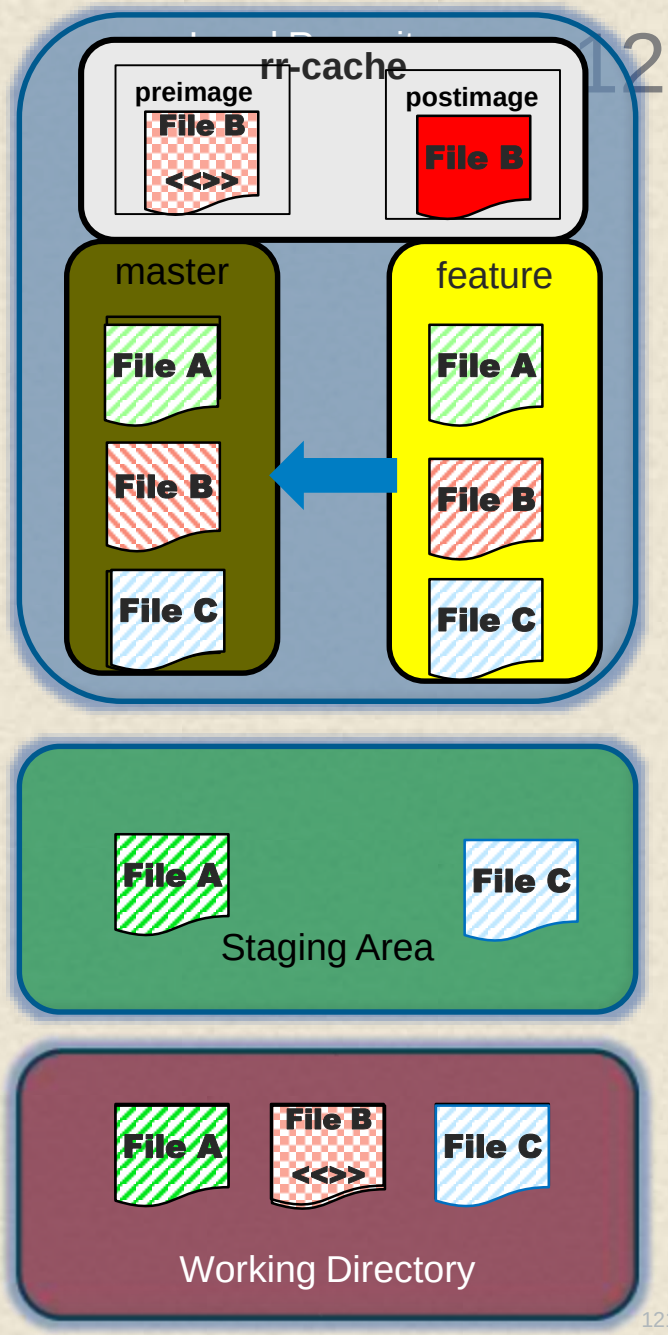
```
$ git reset --hard HEAD~1
$ git checkout master
```



Git Rerere 2

```
$ git reset --hard HEAD~1
$ git checkout master
$ git merge feature
```

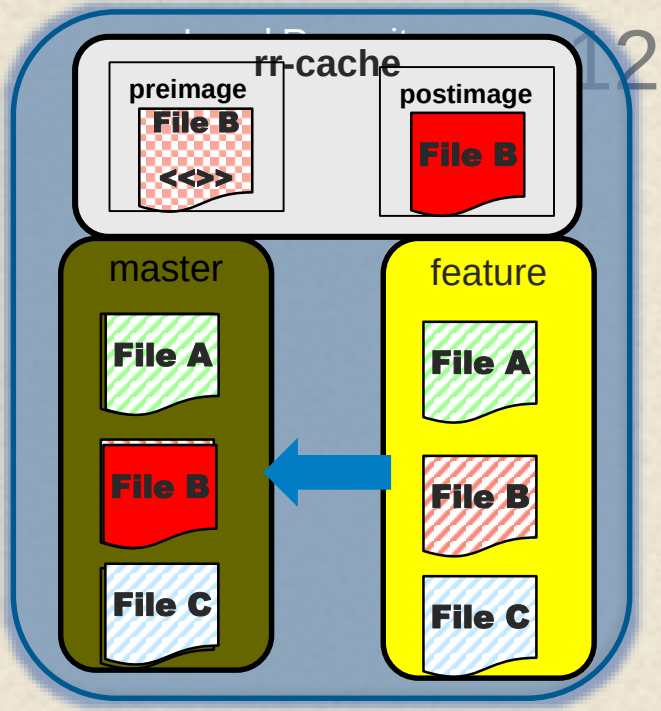
Merge conflict in 'File B'

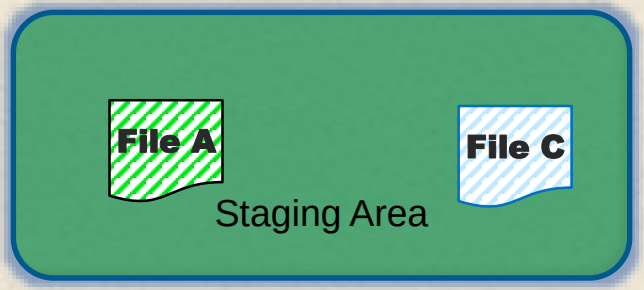


Git Rerere 2

```
$ git reset --hard HEAD~1
$ git checkout master
$ git merge feature
```

Merge conflict in 'File B'
Resolved 'File B' using previous resolution



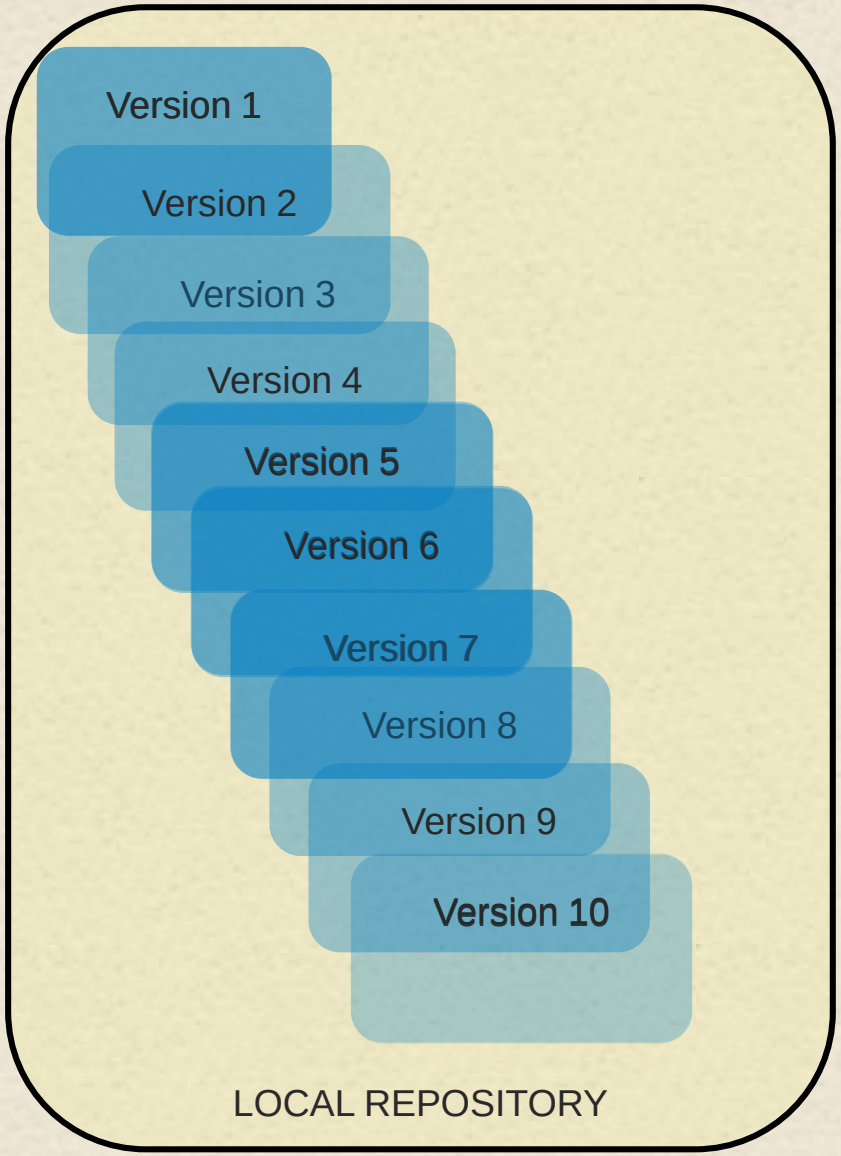


Command: Bisect

- Purpose - Use “automated” binary search through Git’s history to find a specific commit that first introduced a problem (i.e. “first bad commit”)
- Use case - Quickly locate the commit in Git’s history that introduced a bug
- Syntax:

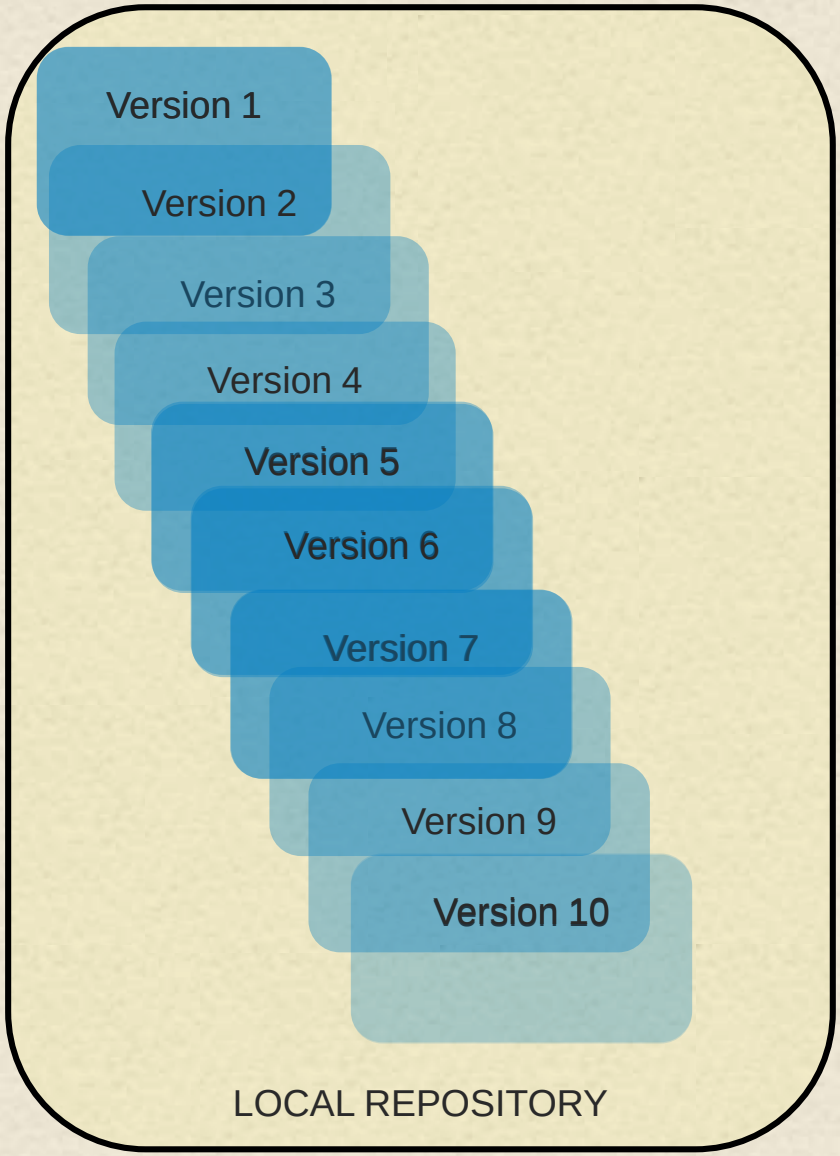
```
git bisect start [--term-{old,good}=<term> --term-{new,bad}=<term>]
          [--no-checkout] [<bad> [<good>...]] [--] [<paths>...]
git bisect (bad|new) [<rev>]
git bisect (good|old) [<rev>...]
git bisect terms [--term-good | --term-bad]
git bisect skip [(<rev>|<range>)...]
git bisect reset [<commit>]
git bisect visualize
git bisect replay <logfile>
git bisect log
git bisect run <cmd>...
git bisect help
```

Bisect



WORKING DIRECTORY

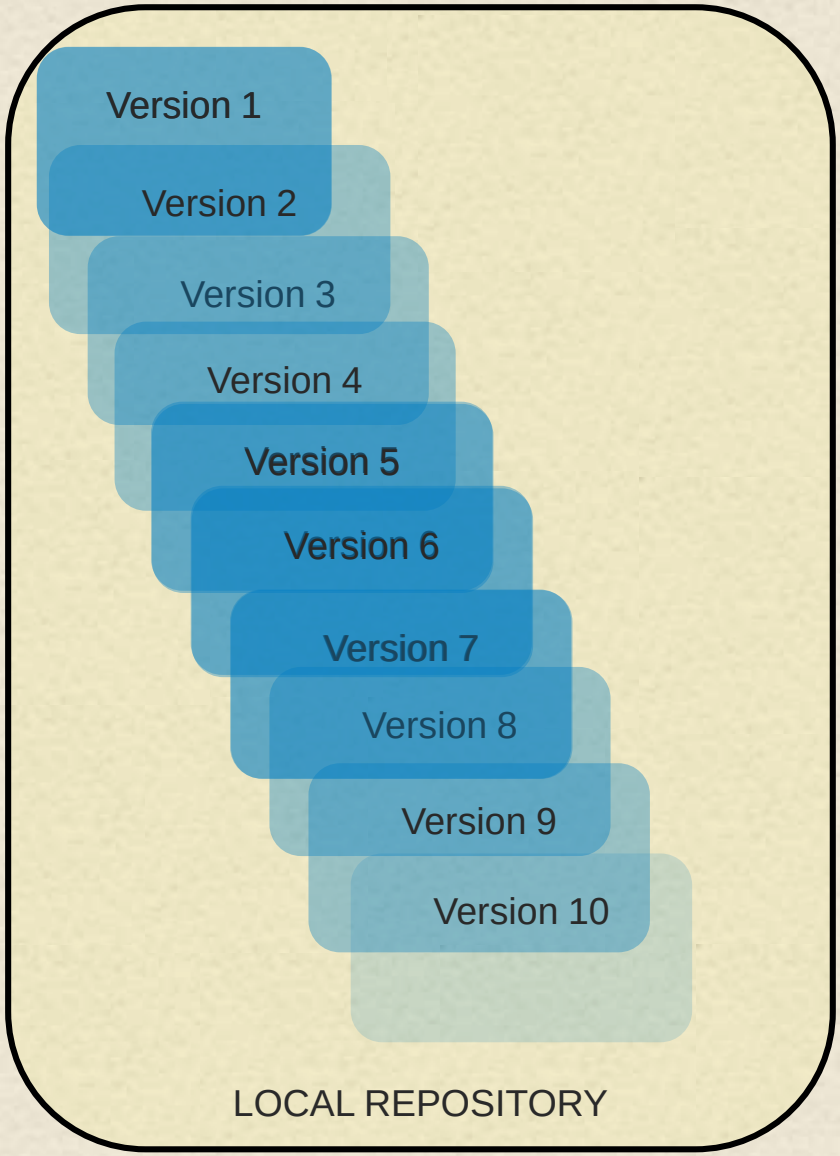
Bisect

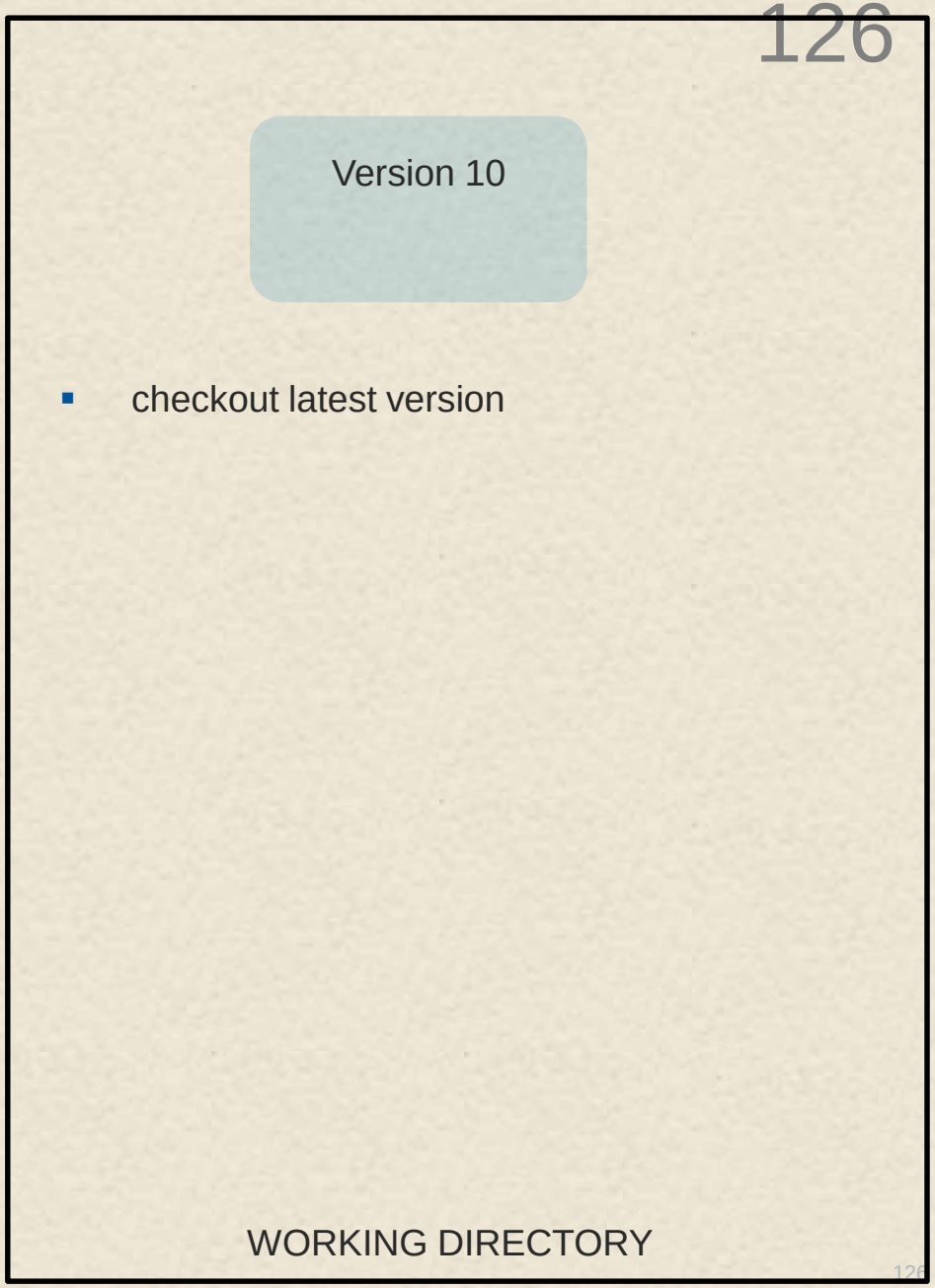


- checkout latest version

WORKING DIRECTORY

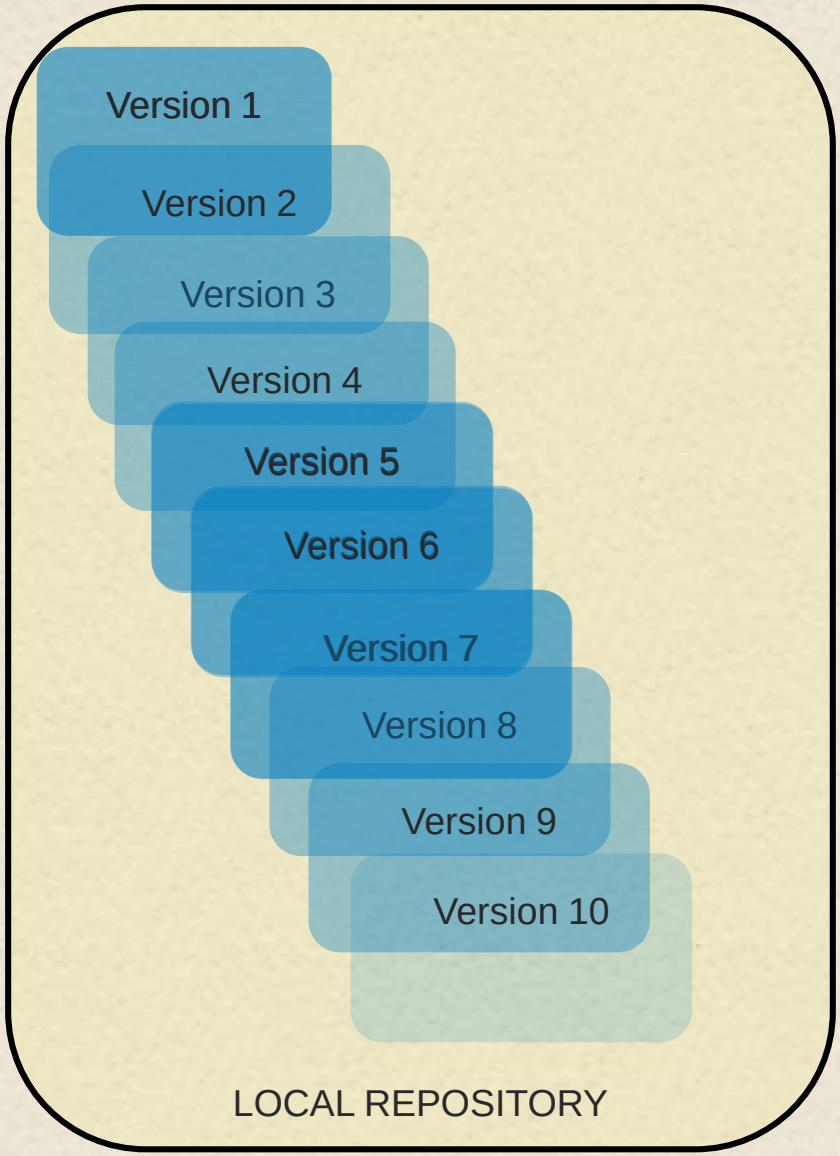
Bisect

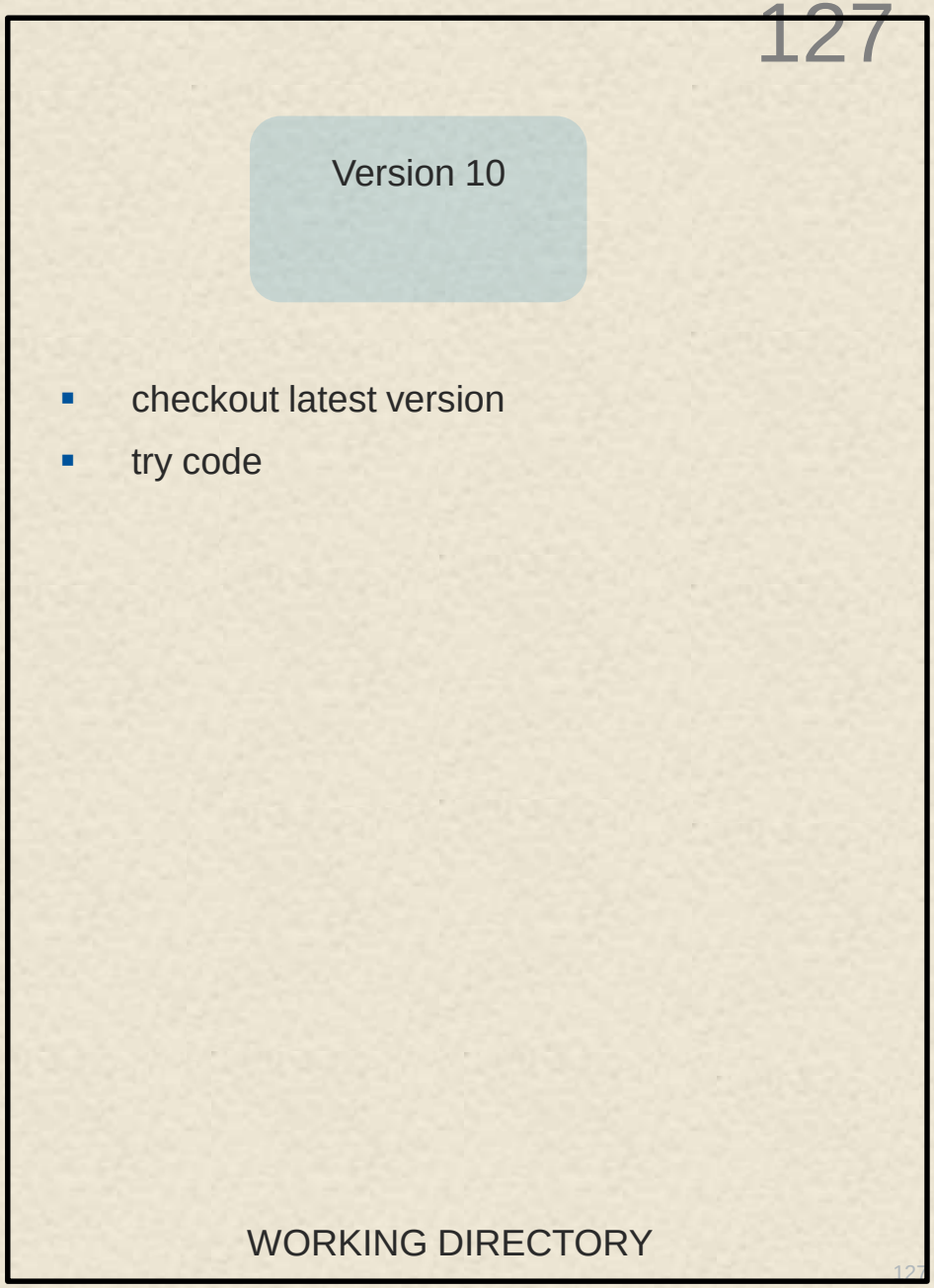




- checkout latest version

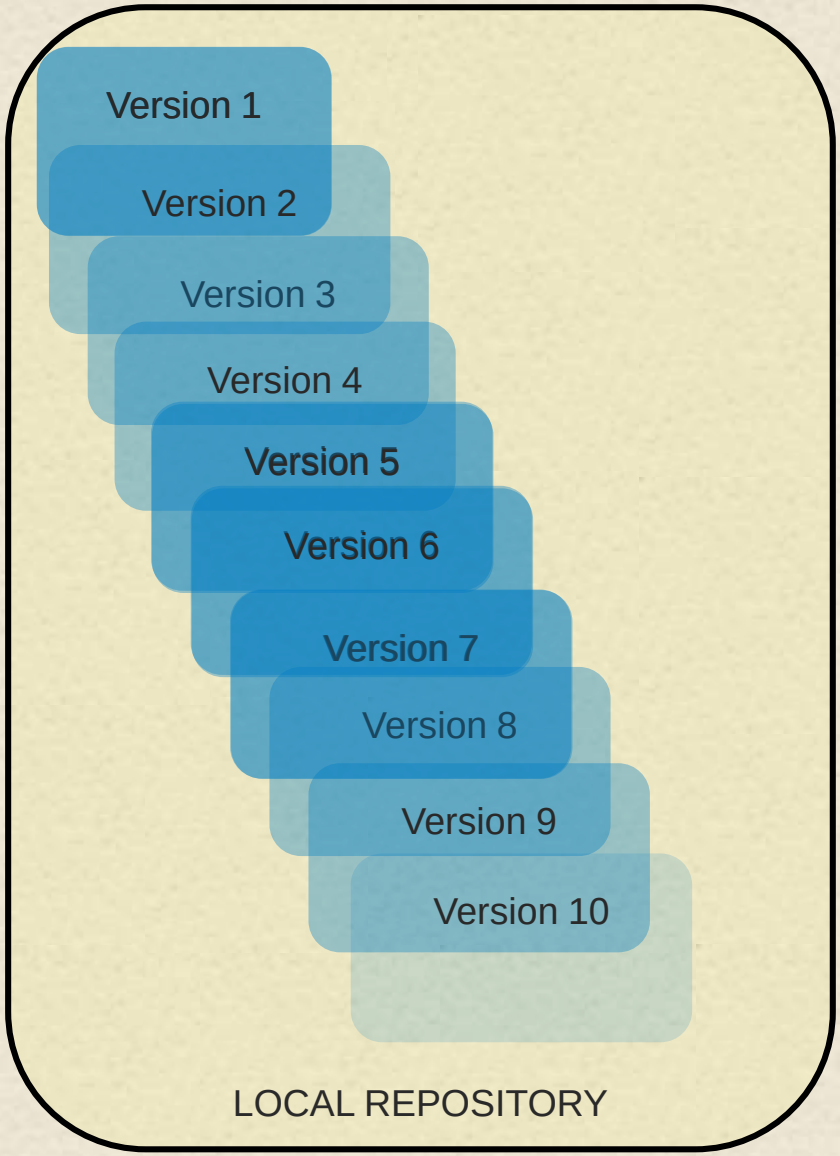
Bisect

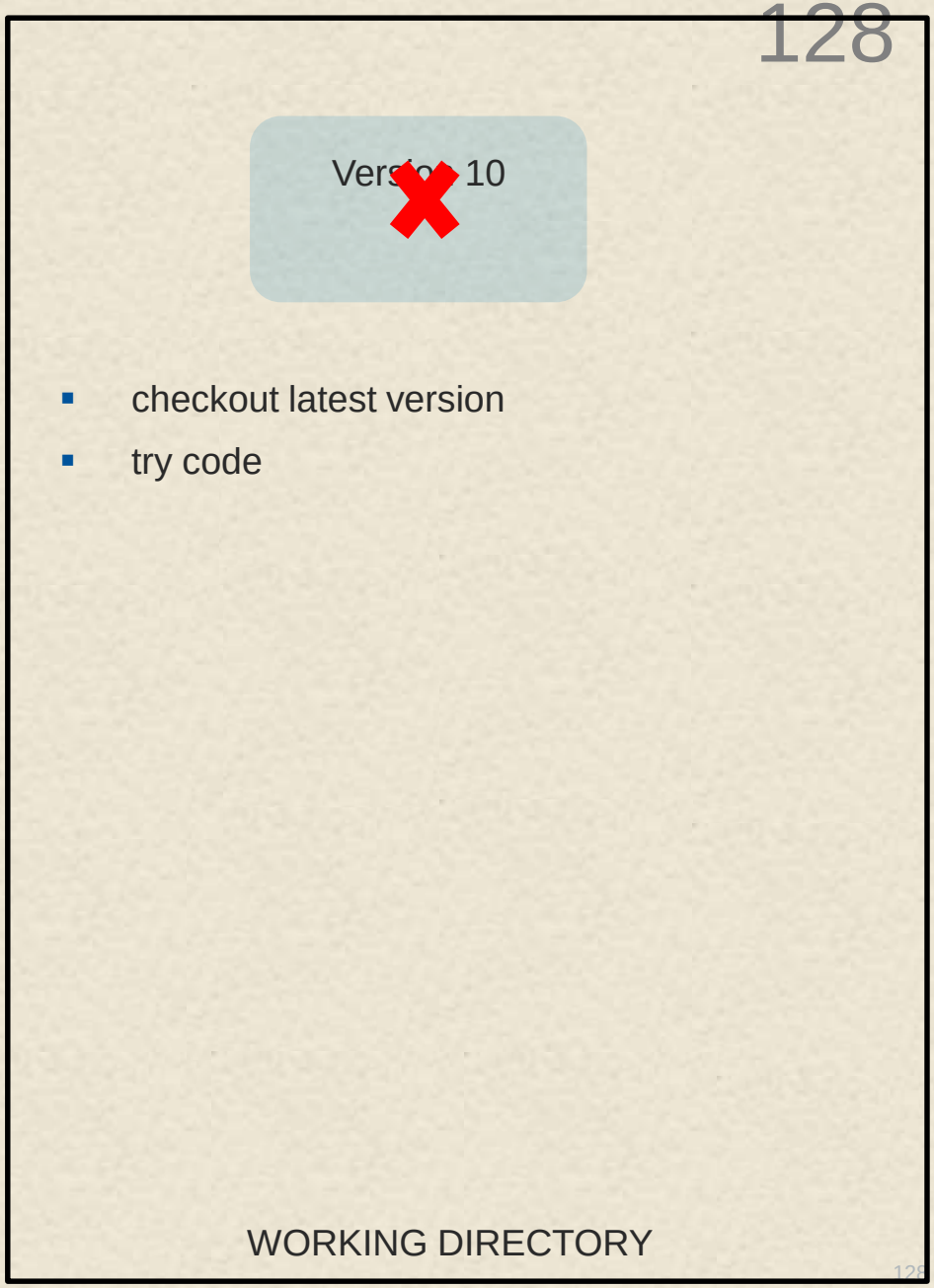




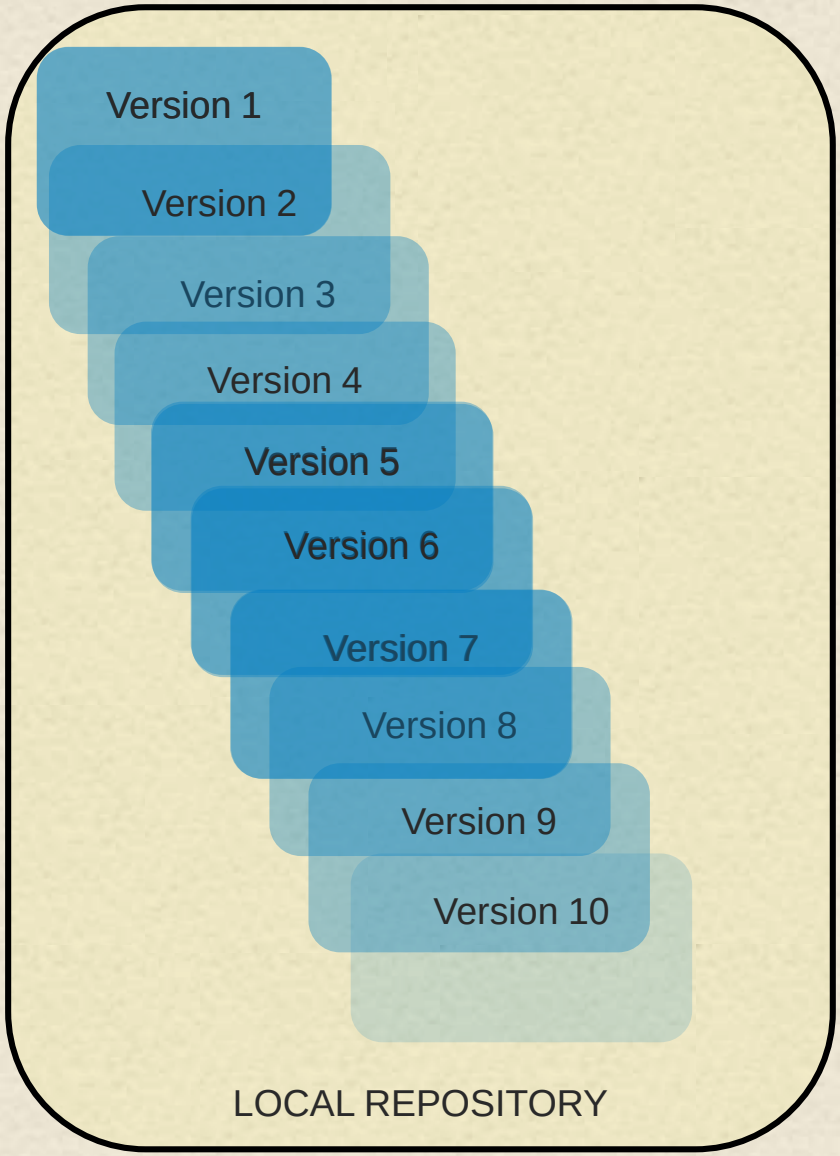
- checkout latest version
- try code

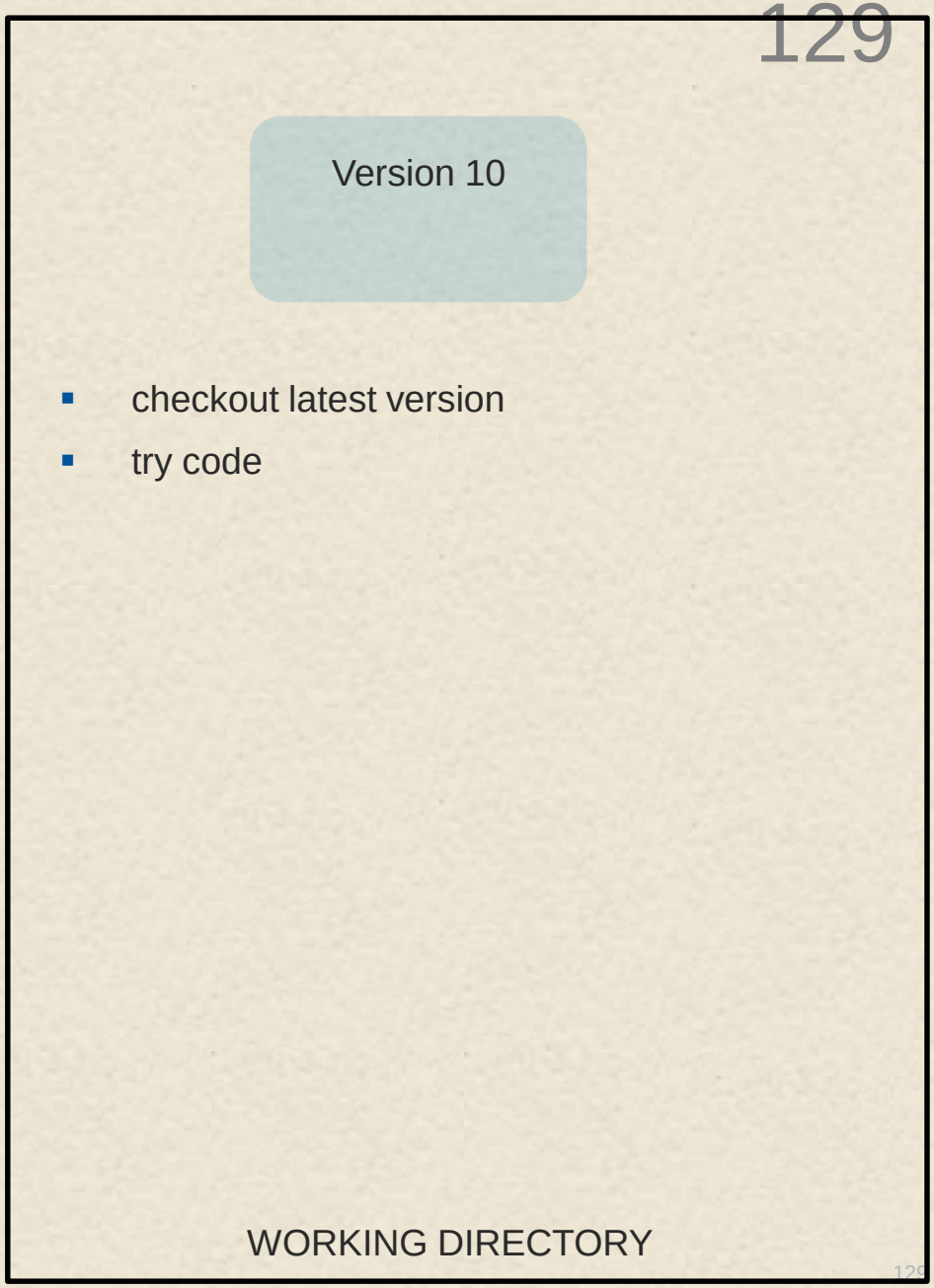
Bisect





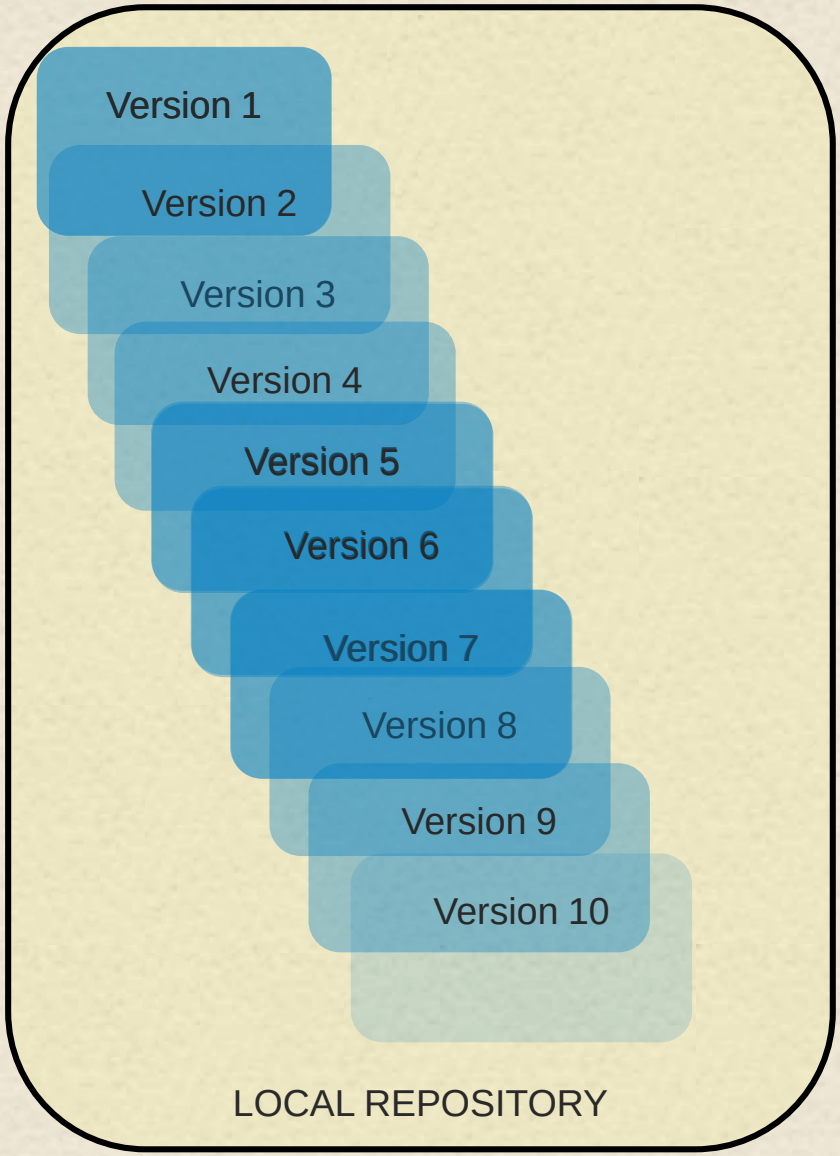
Bisect

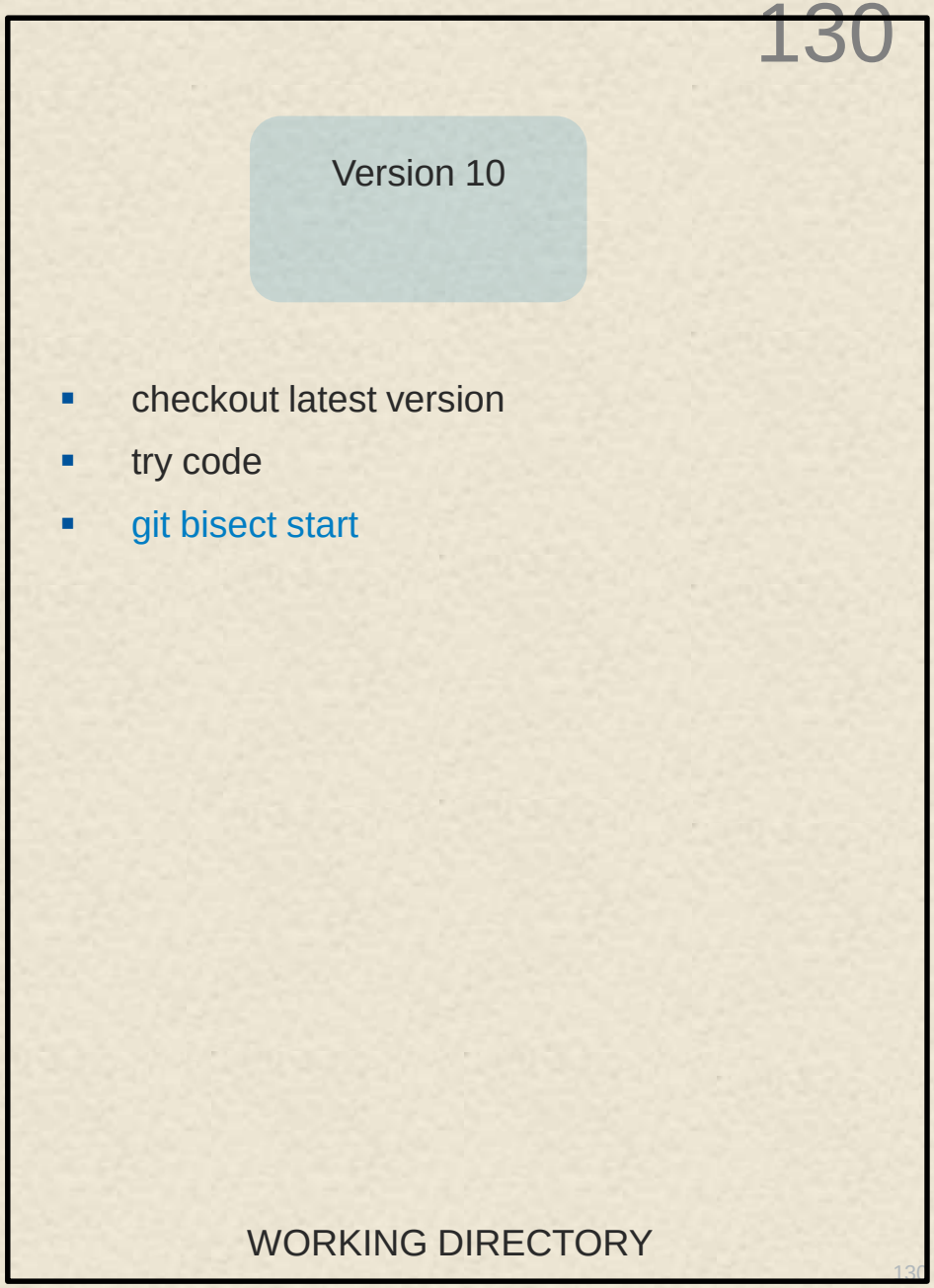




- checkout latest version
- try code

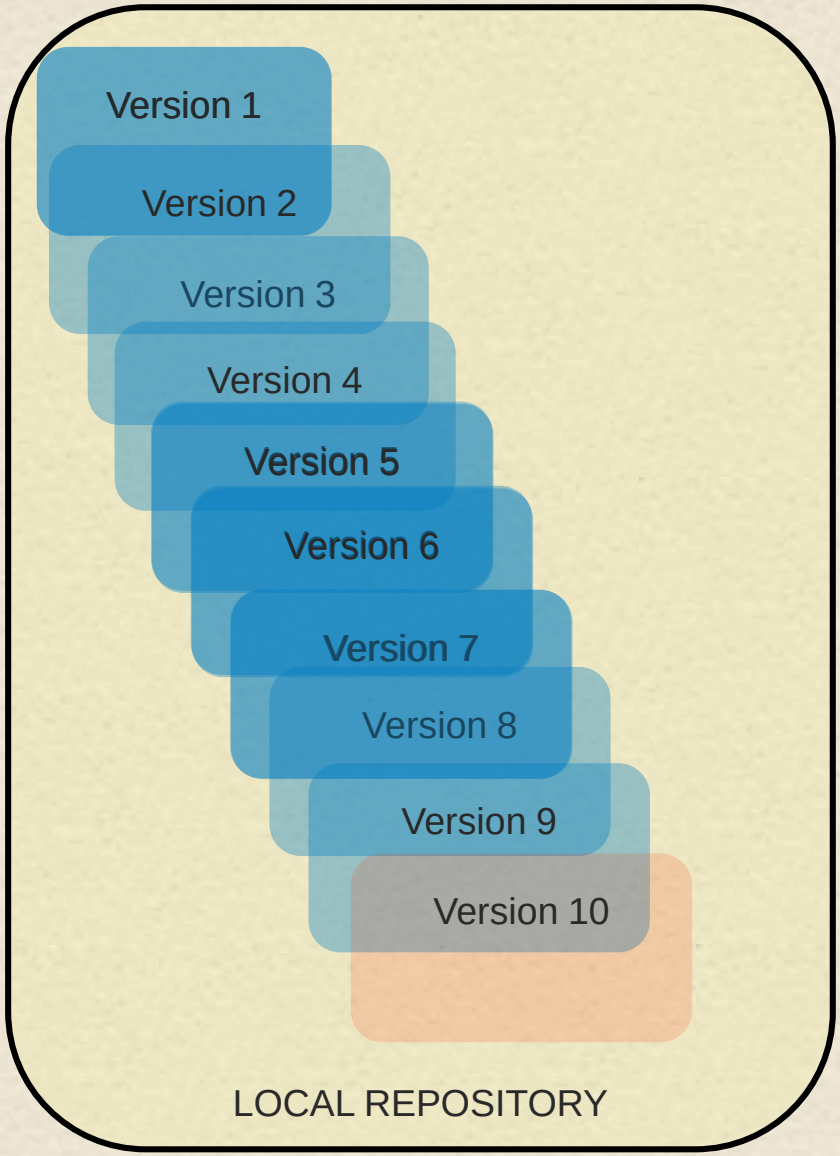
Bisect





- checkout latest version
- try code
- git bisect start

Bisect

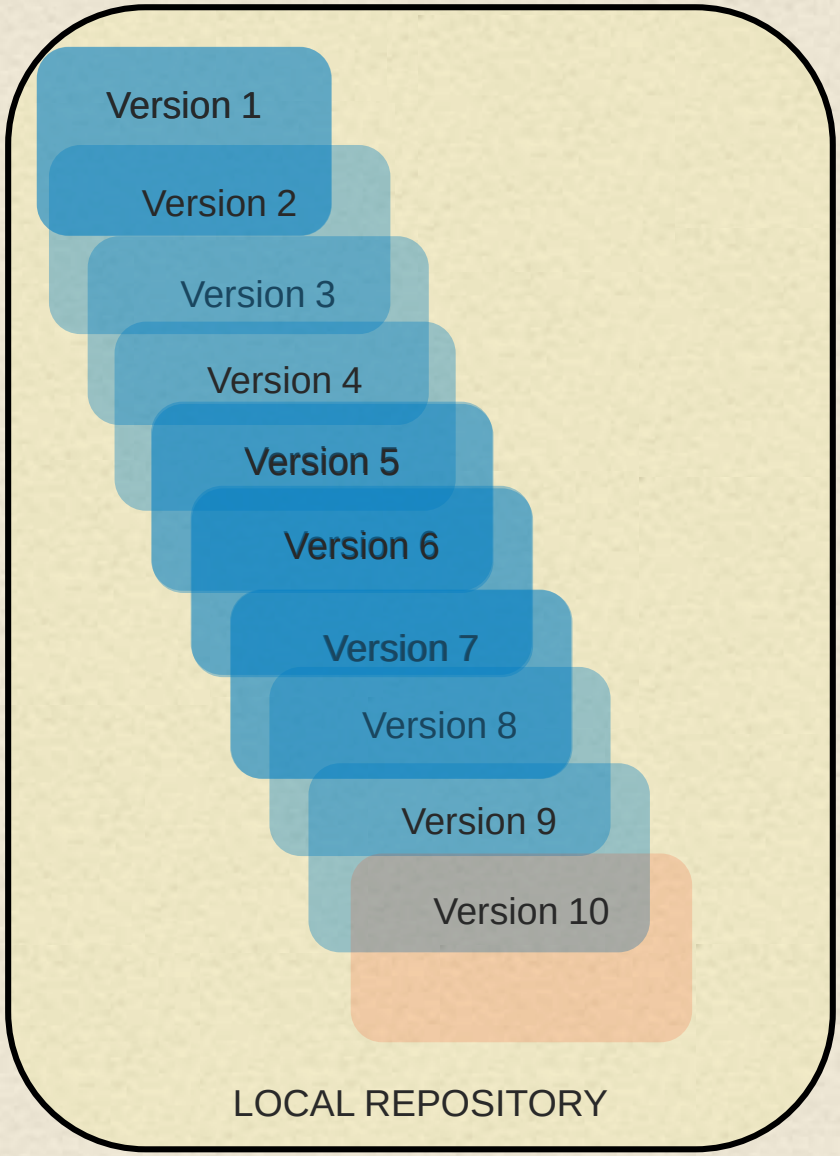


Version 10

- checkout latest version
- try code
- git bisect start
- git bisect bad

WORKING DIRECTORY

Bisect

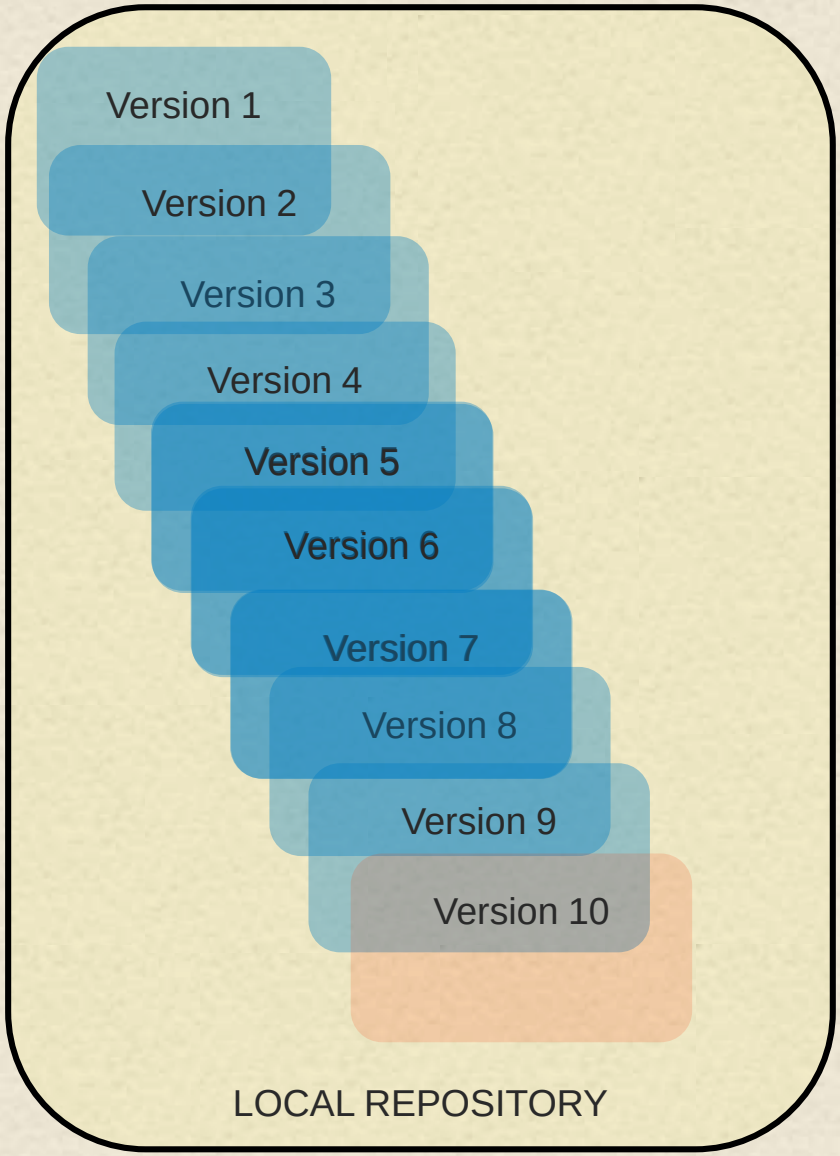


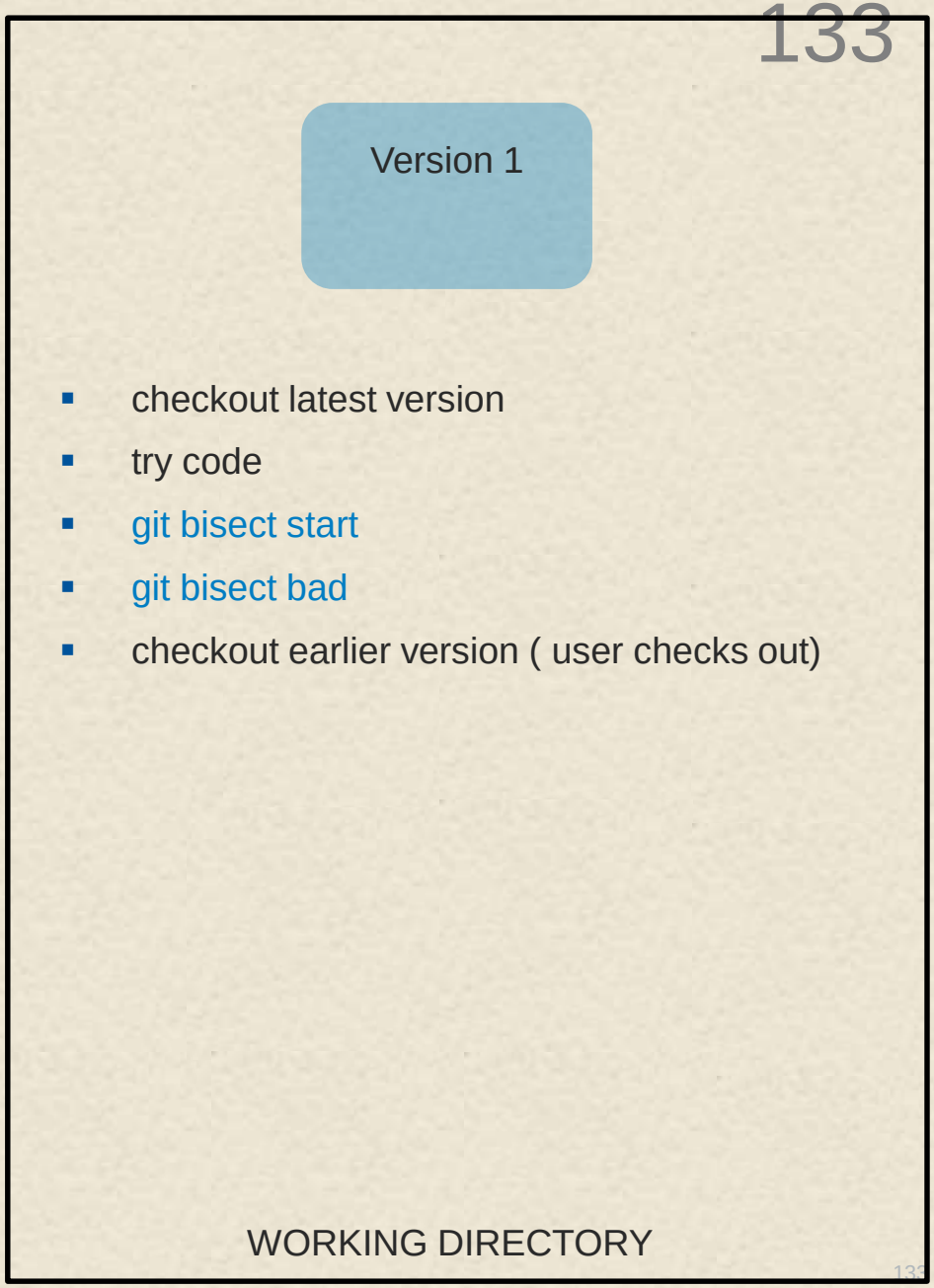
Version 10

- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)

WORKING DIRECTORY

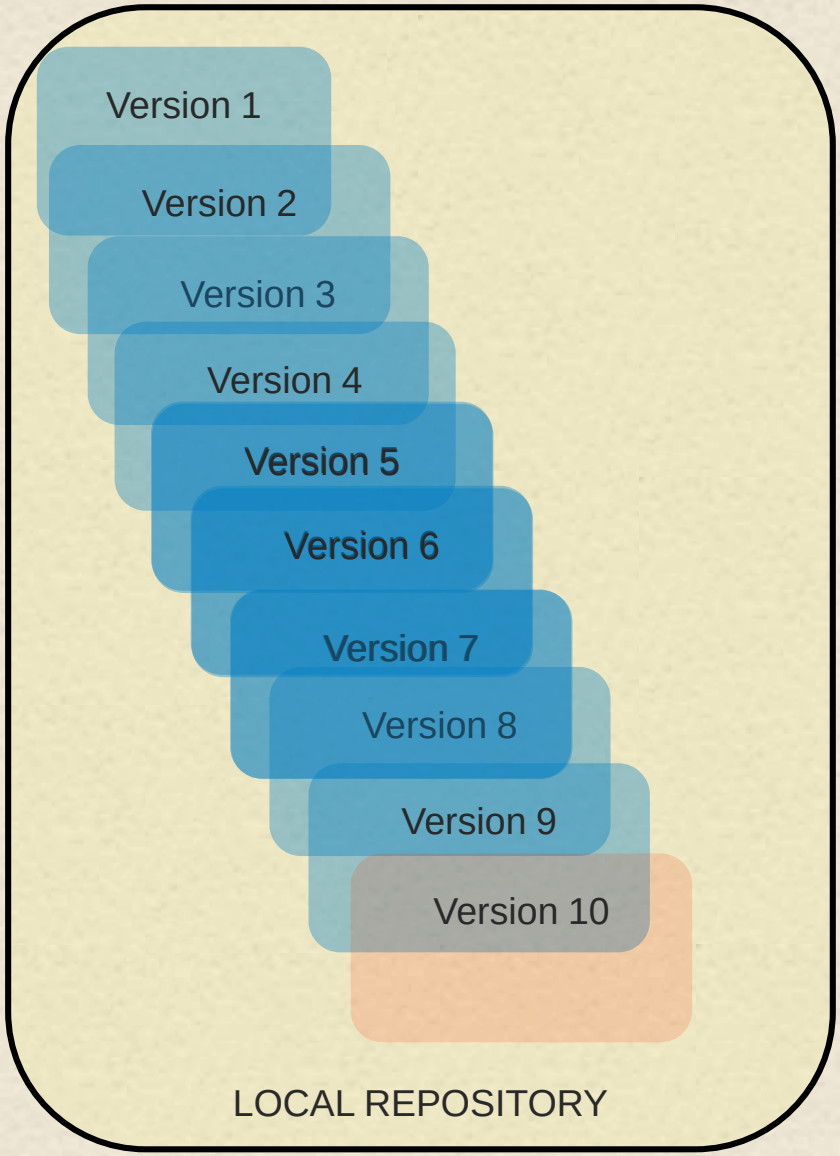
Bisect

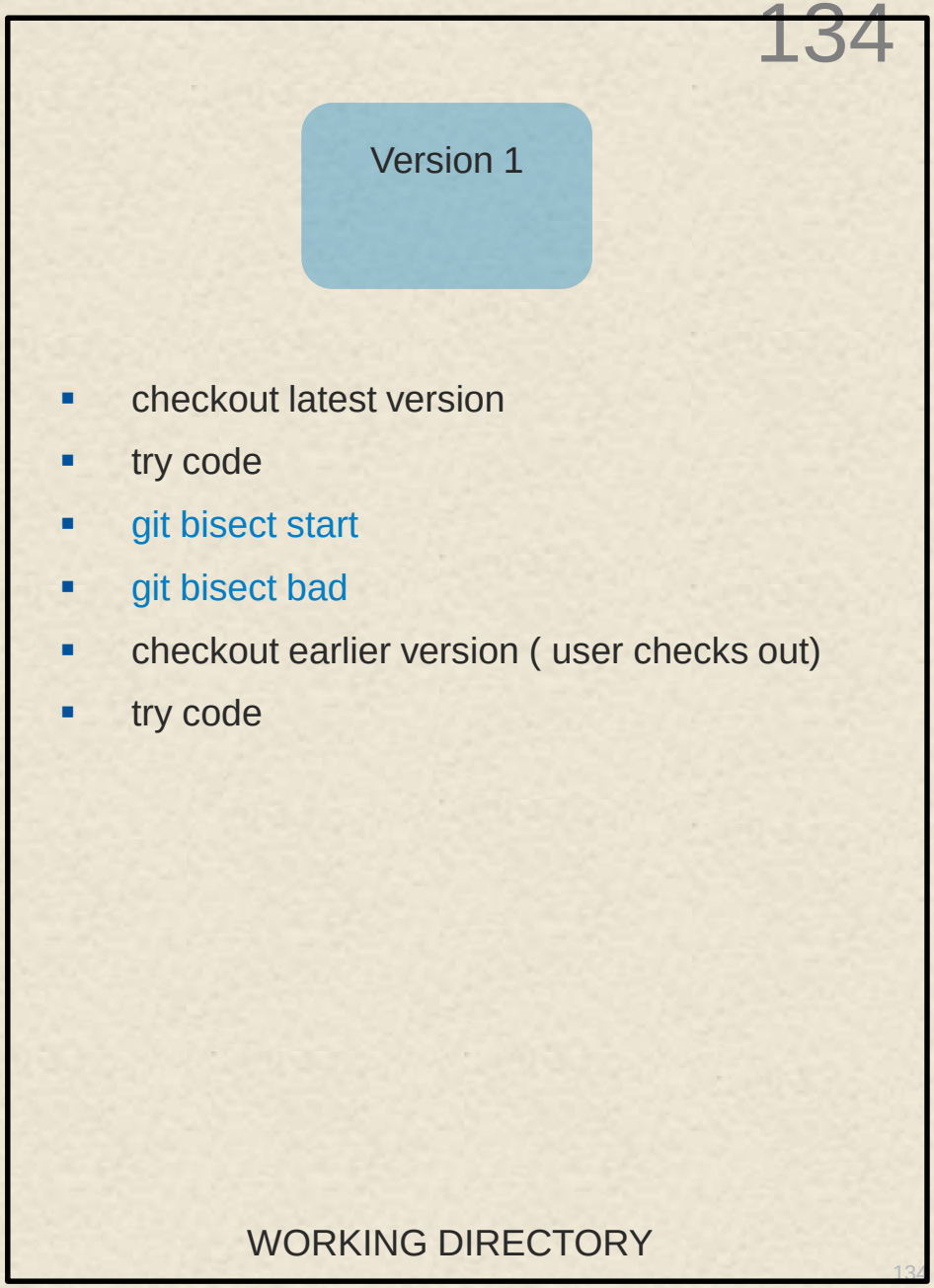




- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)

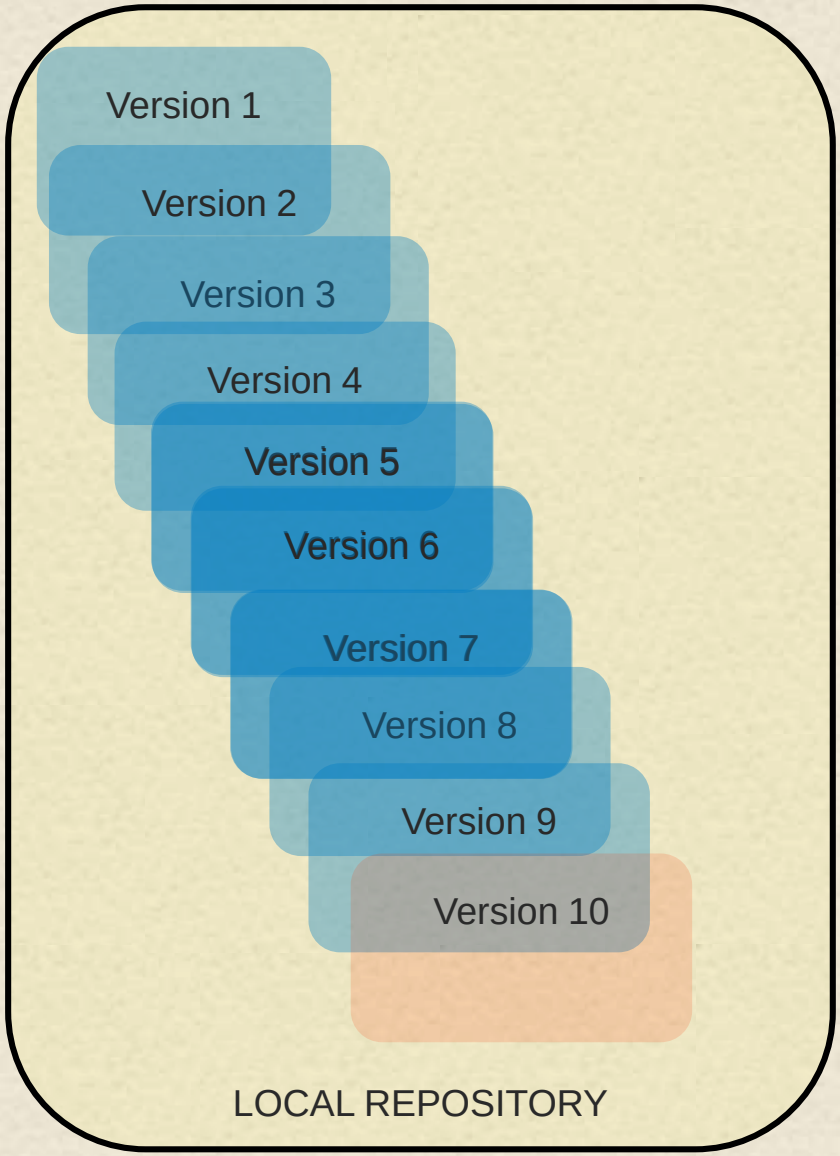
Bisect





- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code

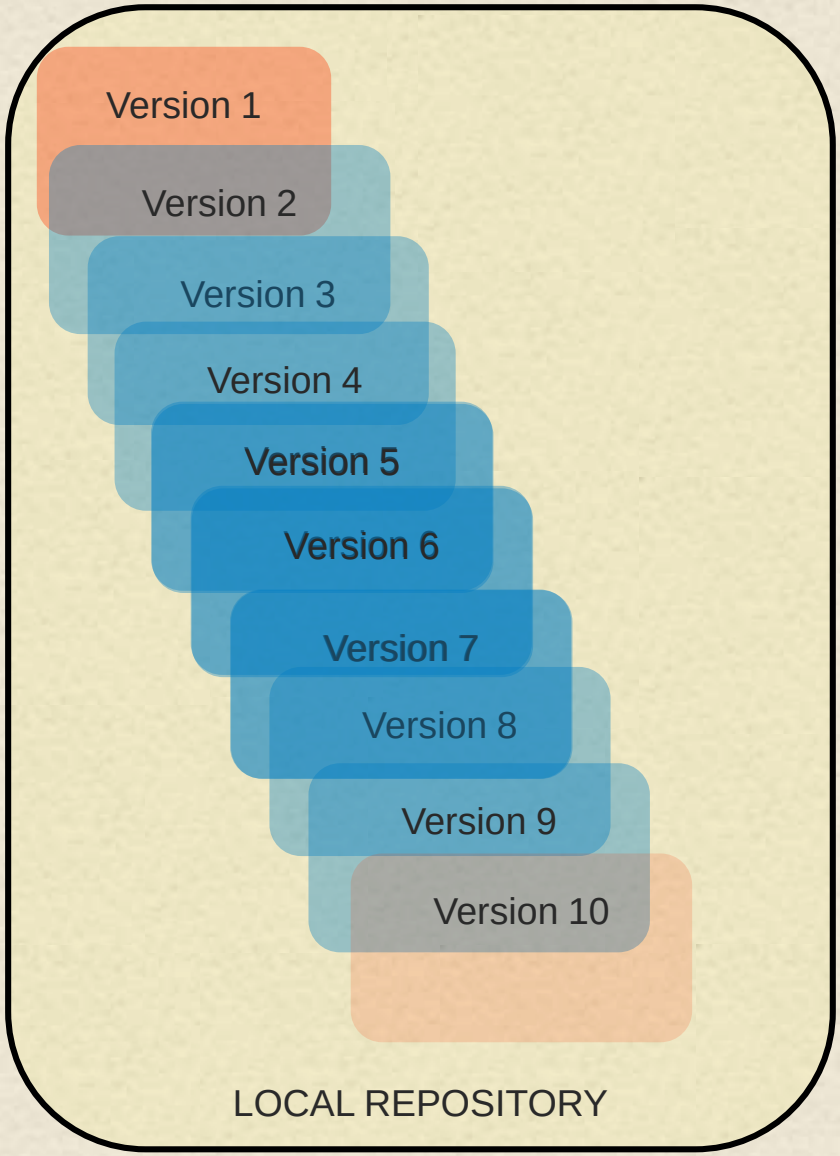
Bisect

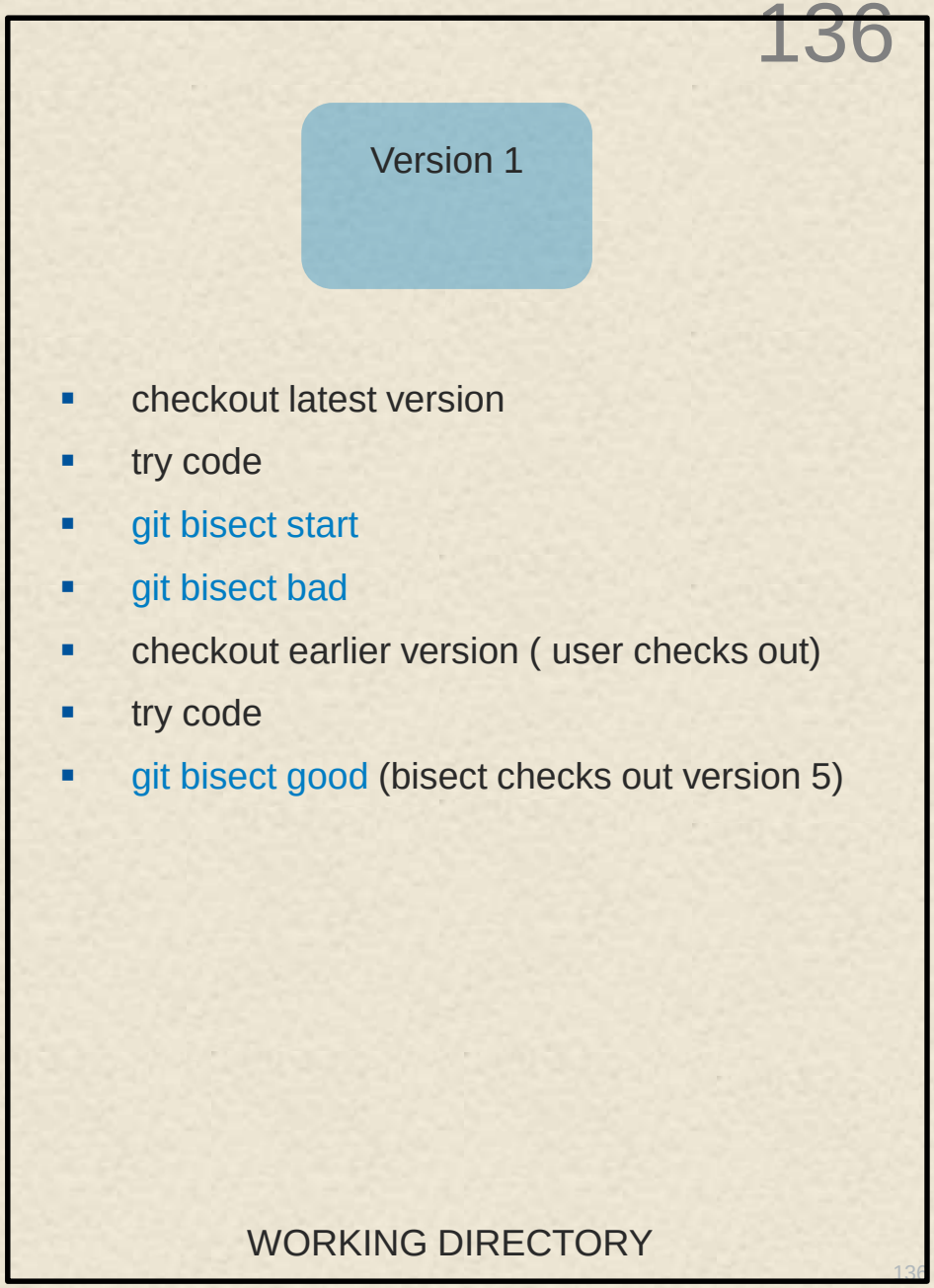


- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code

WORKING DIRECTORY

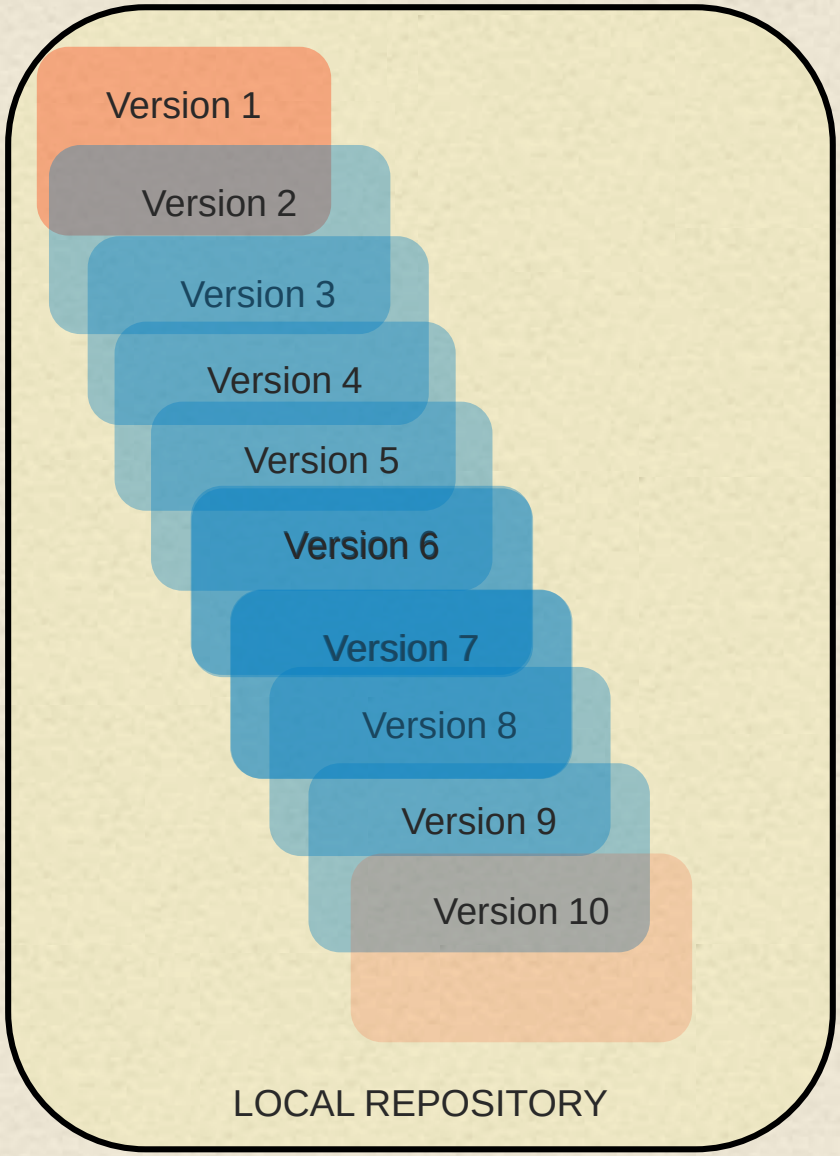
Bisect





- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)

Bisect

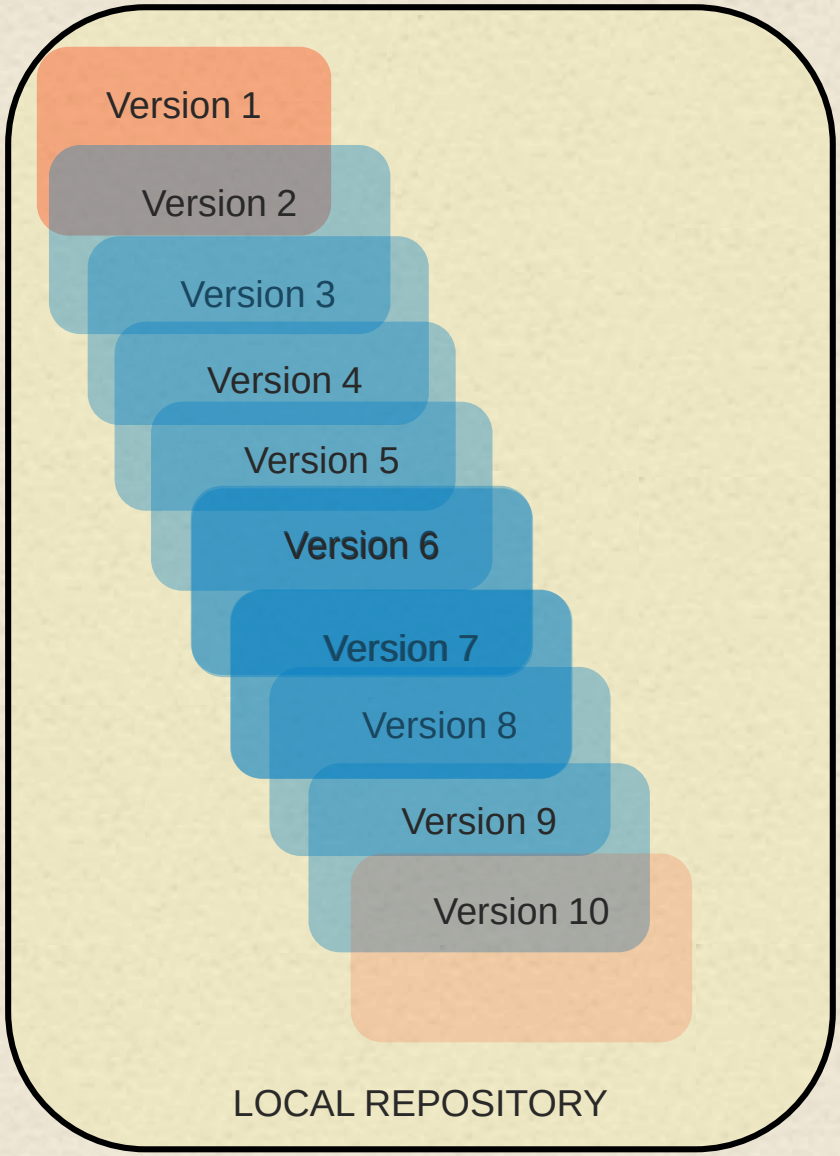


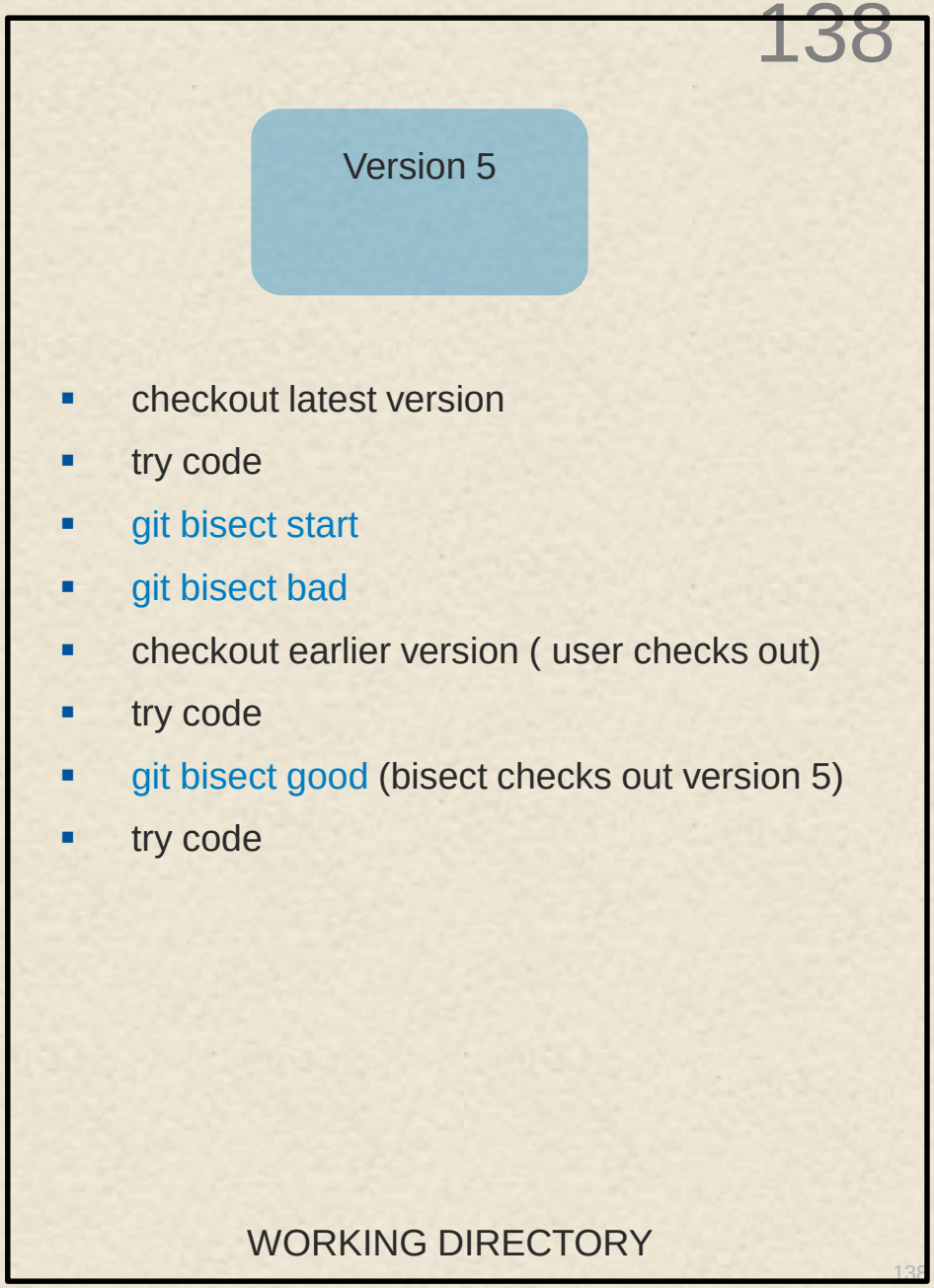
Version 5

- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)

WORKING DIRECTORY

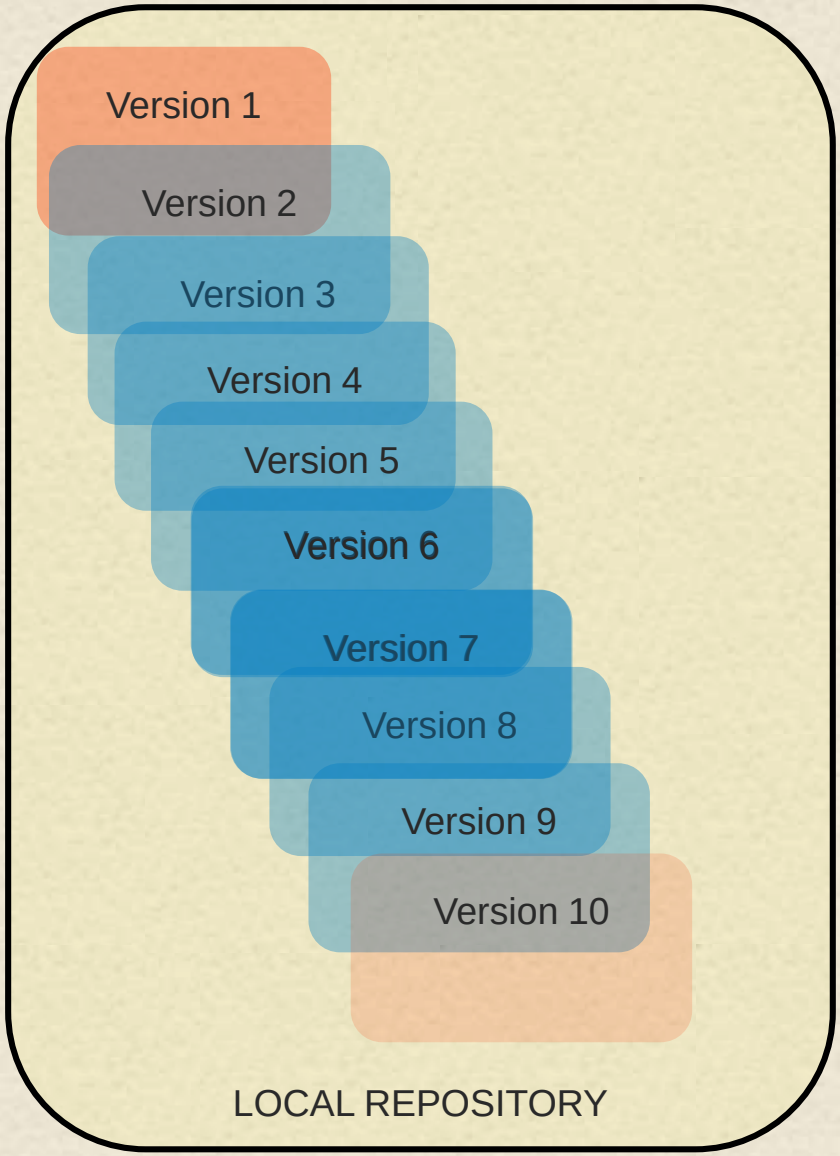
Bisect

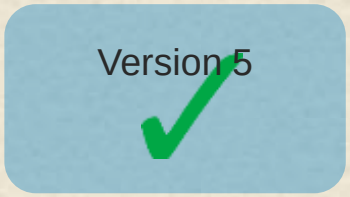




- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code

Bisect

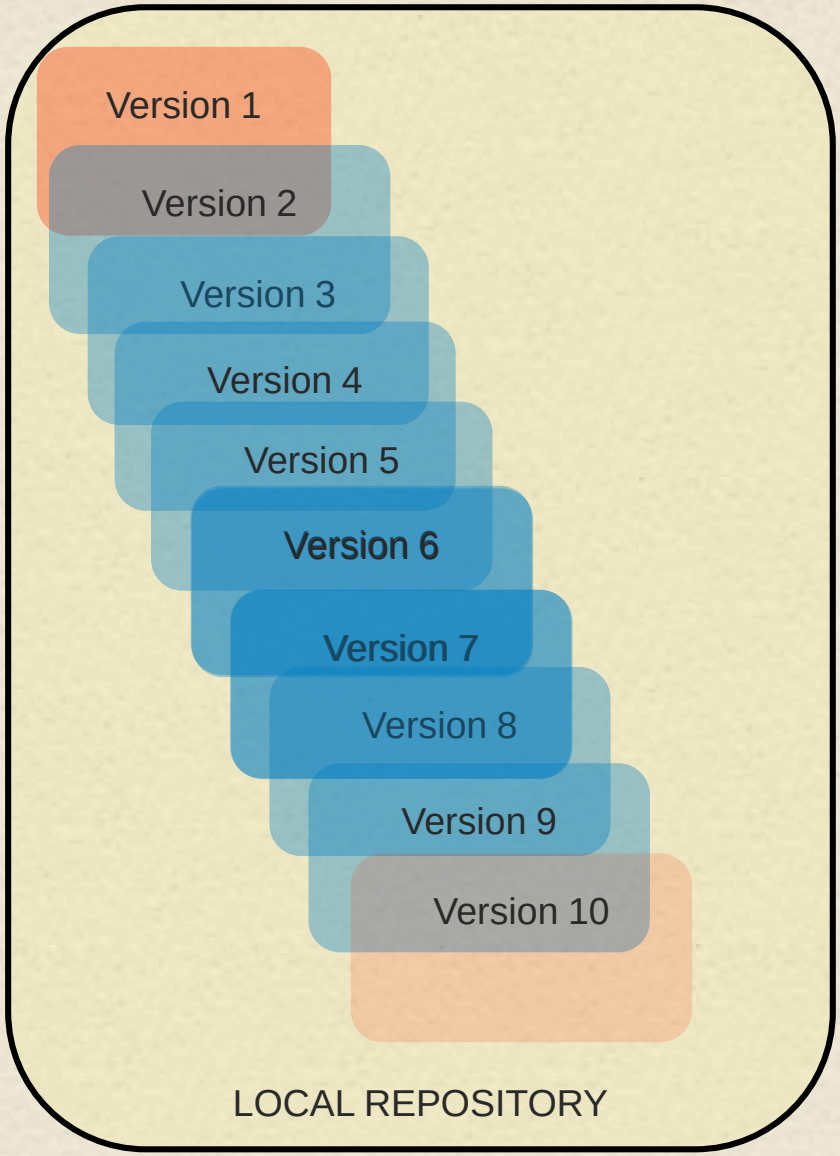


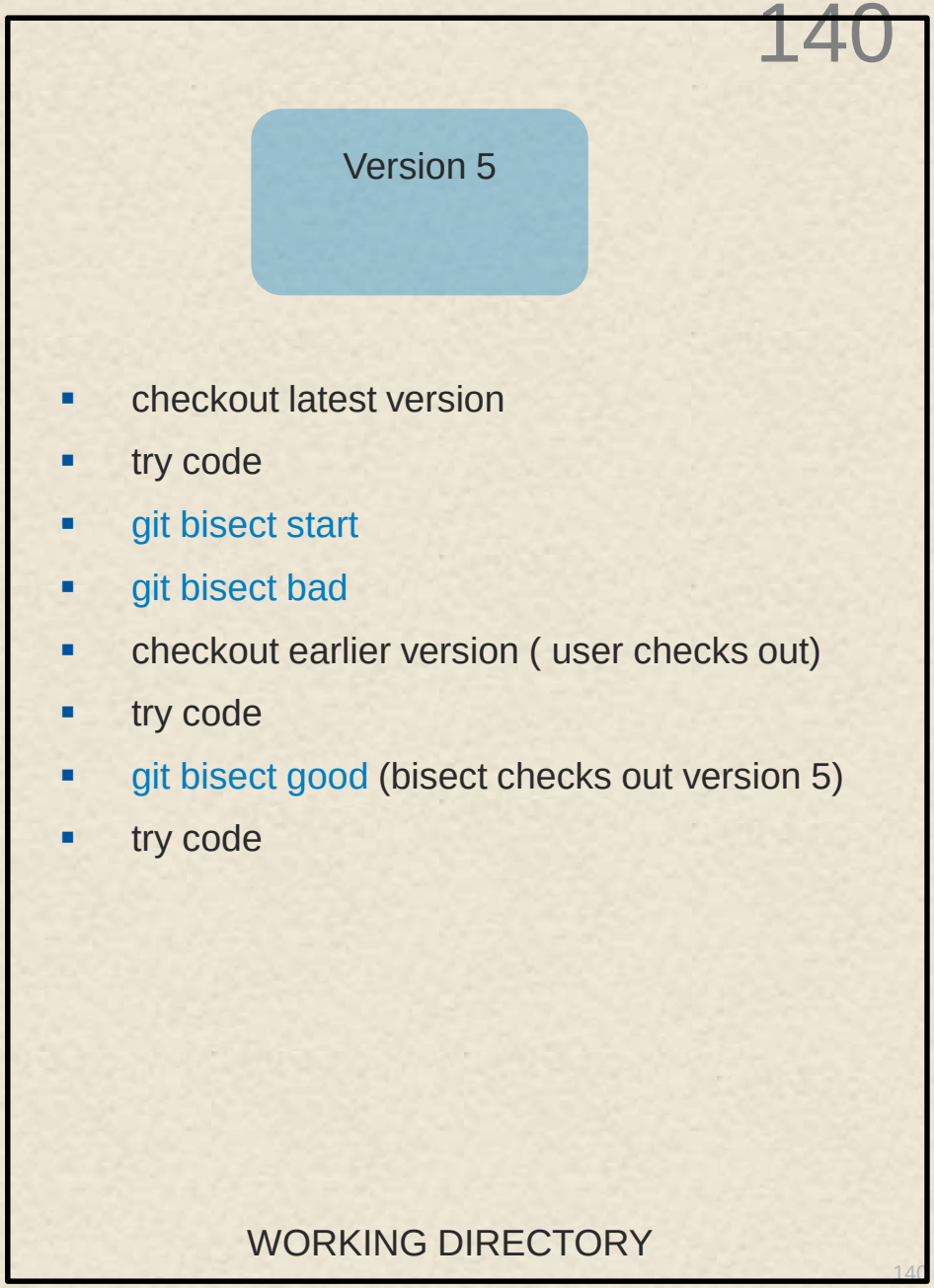


- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code

WORKING DIRECTORY

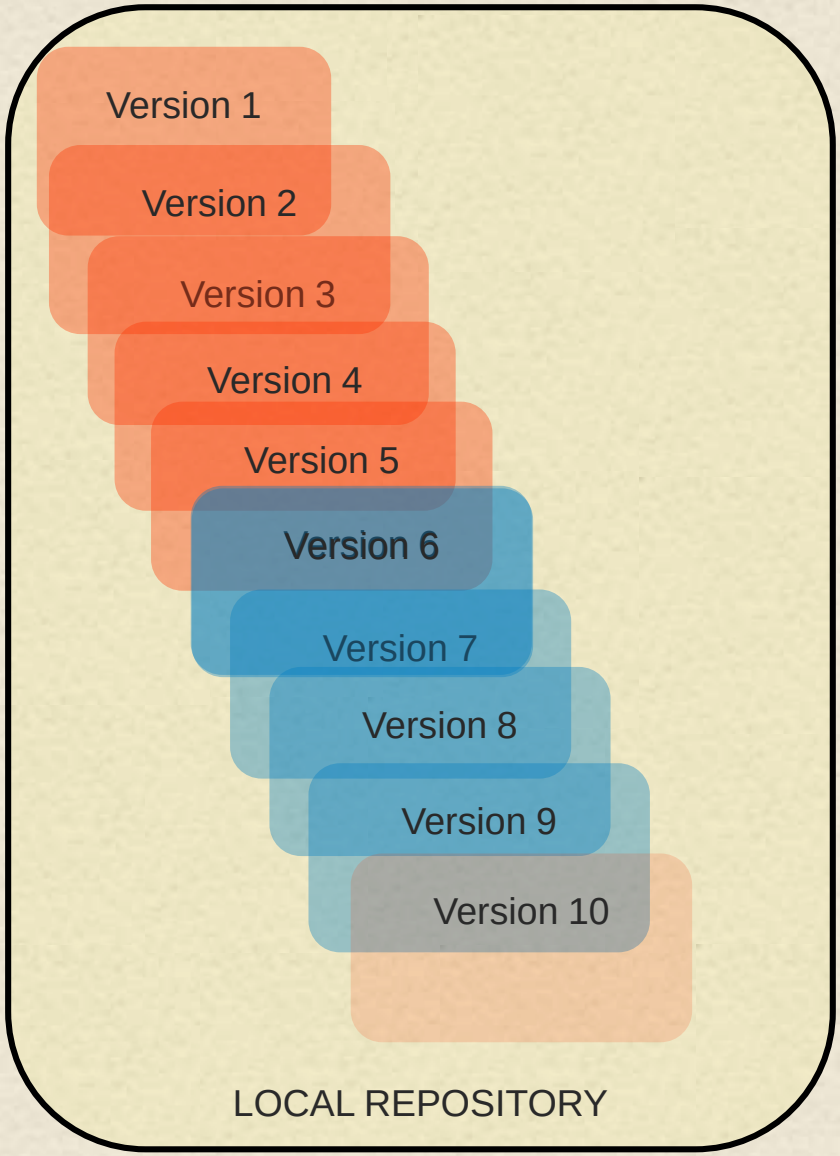
Bisect

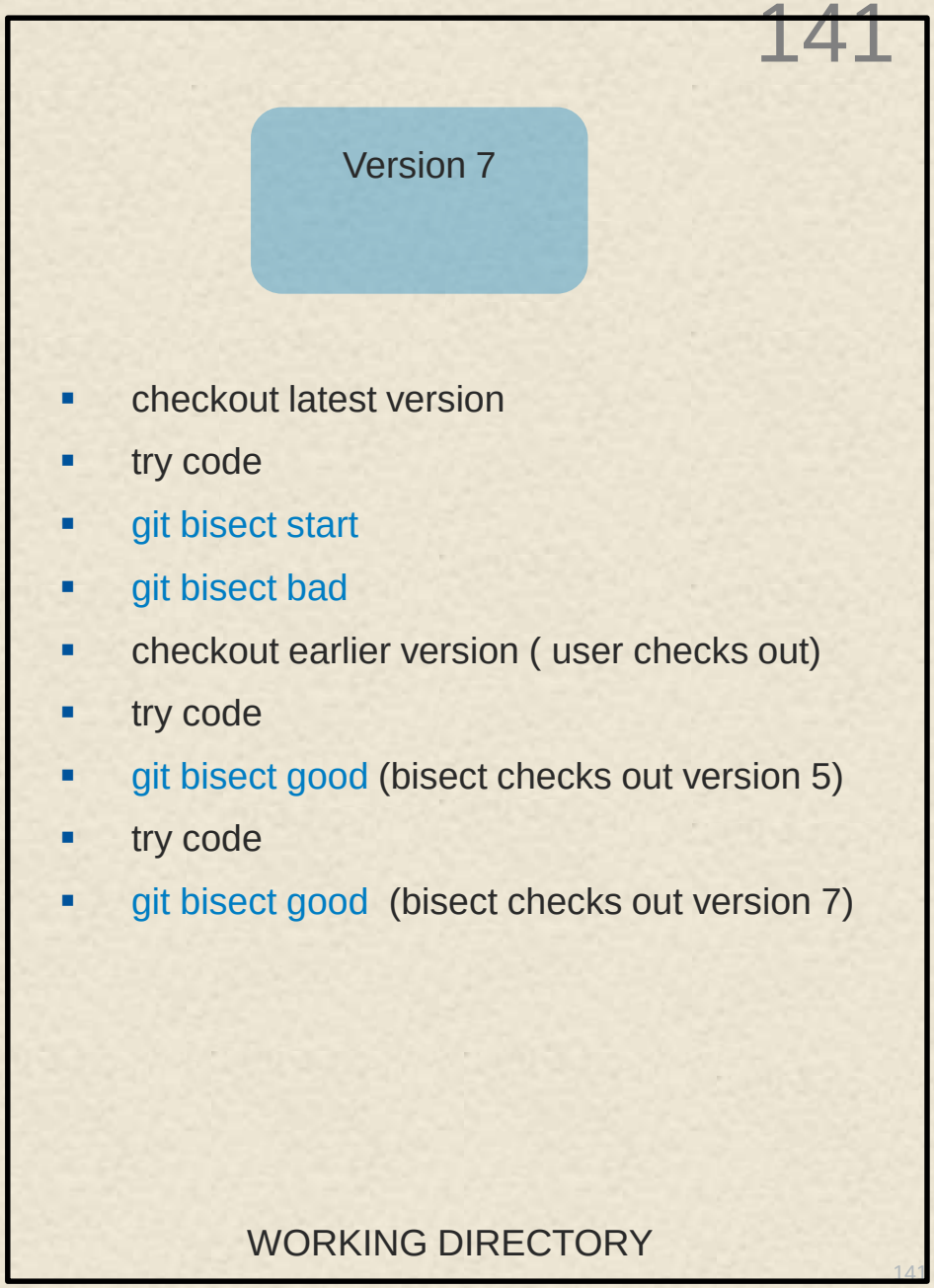




- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code

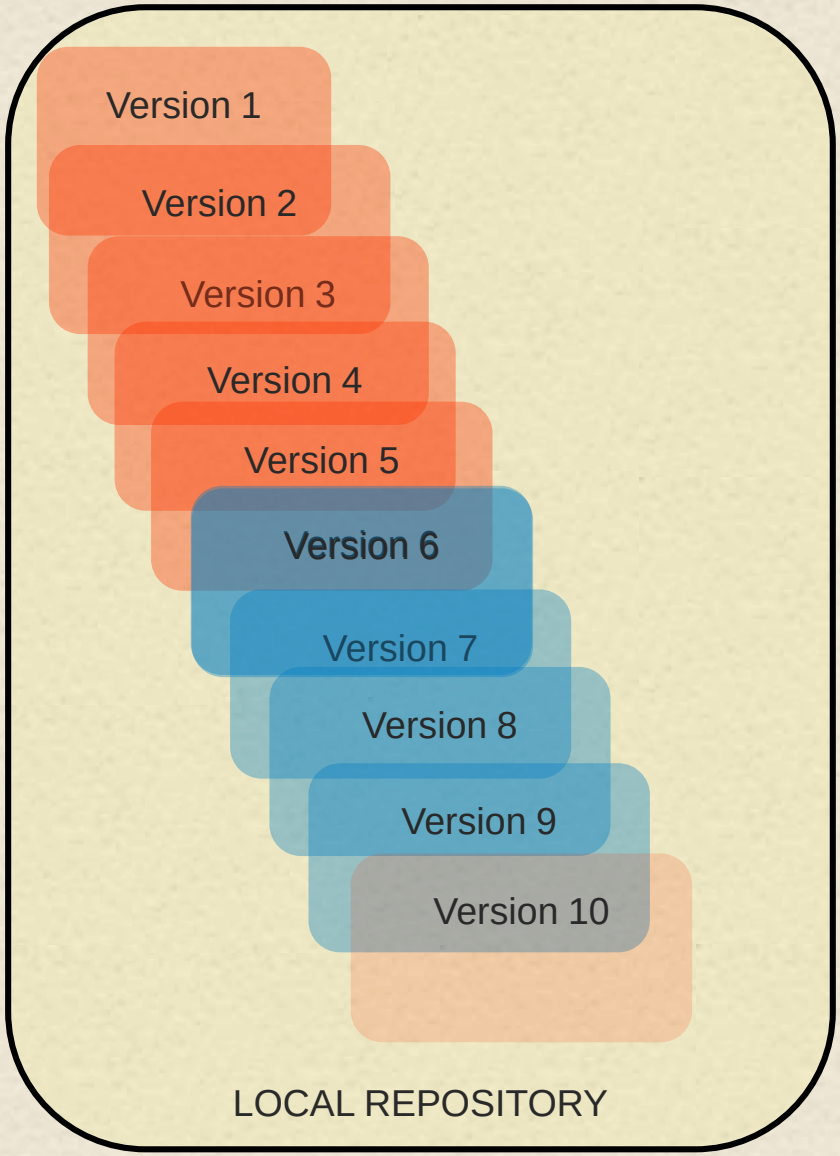
Bisect





- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code
- git bisect good (bisect checks out version 7)

Bisect

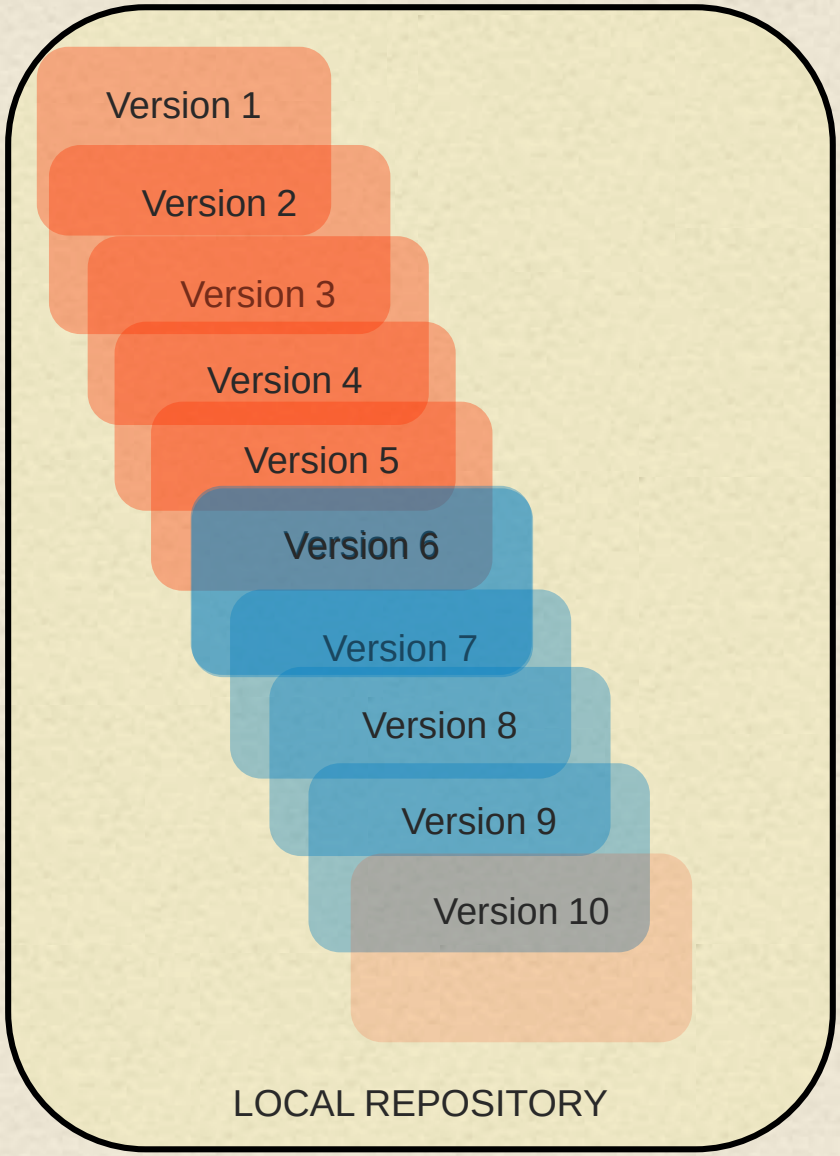


Version 7

- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code
- git bisect good (bisect checks out version 7)
- try code

WORKING DIRECTORY

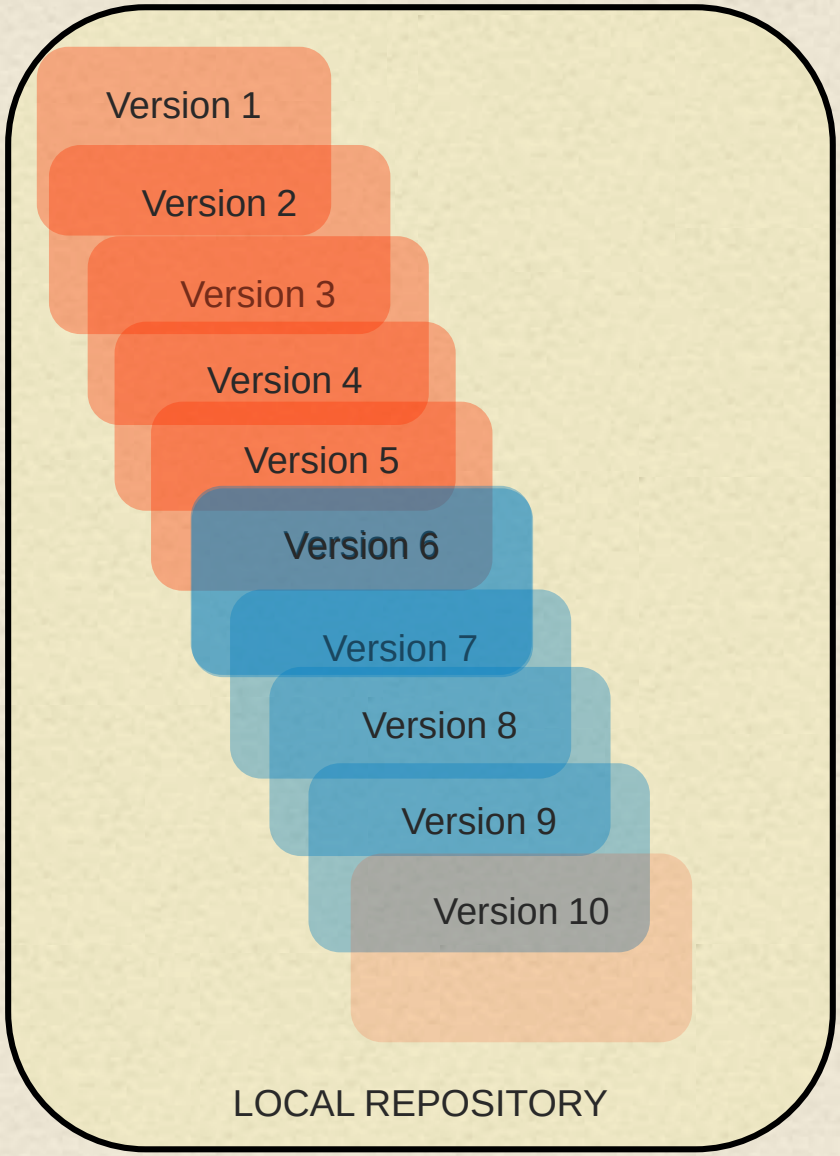
Bisect



- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code
- git bisect good (bisect checks out version 7)
- try code

WORKING DIRECTORY

Bisect

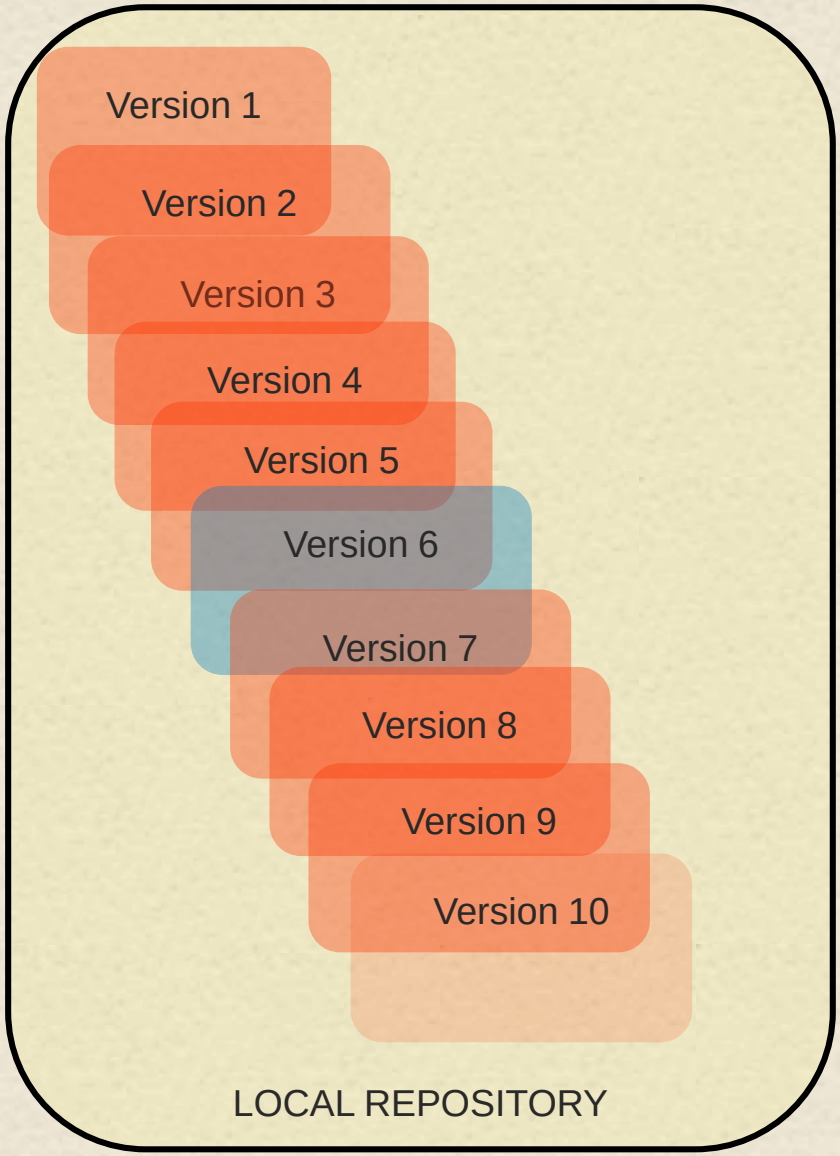


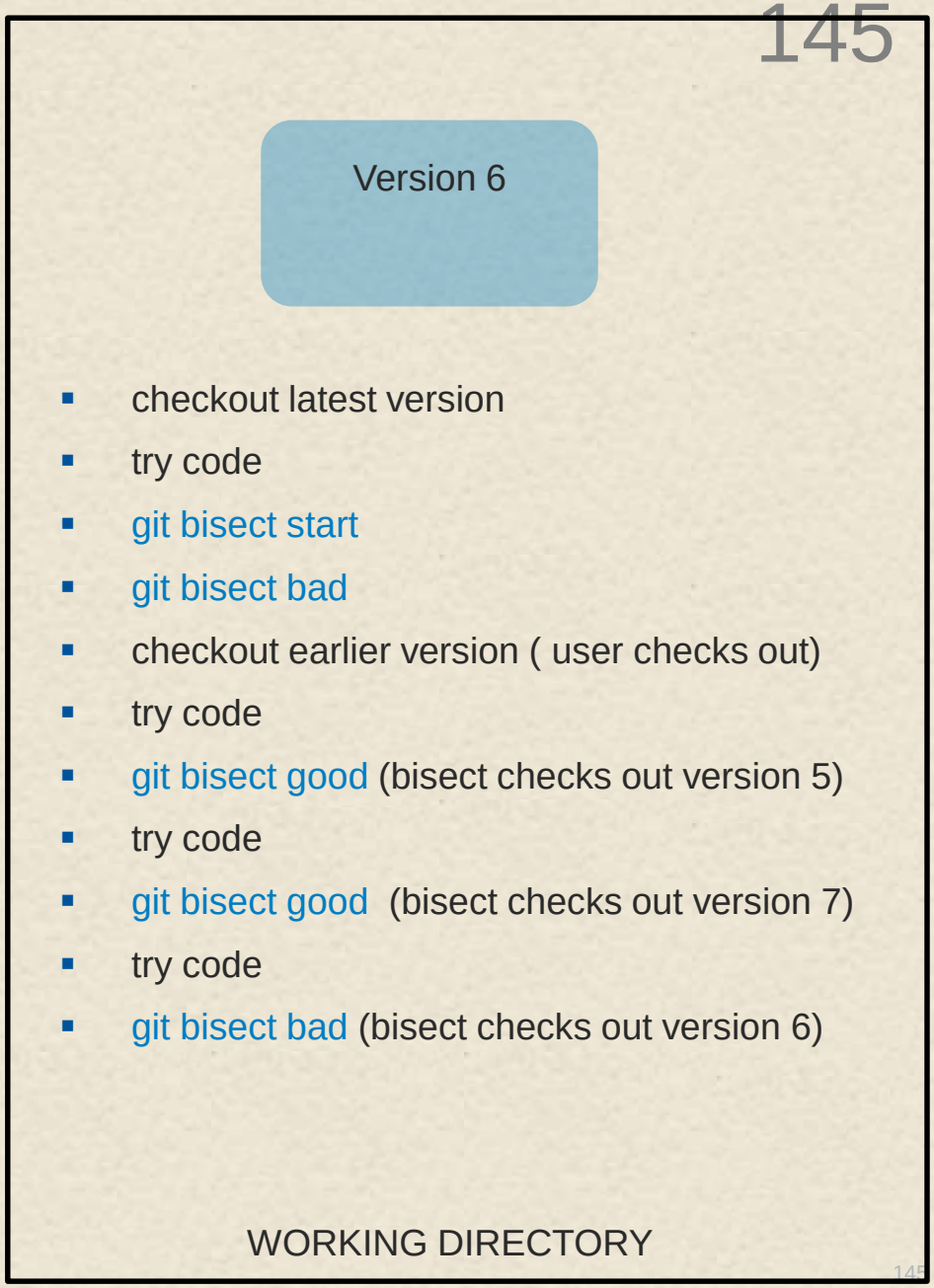
Version 7

- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code
- git bisect good (bisect checks out version 7)
- try code

WORKING DIRECTORY

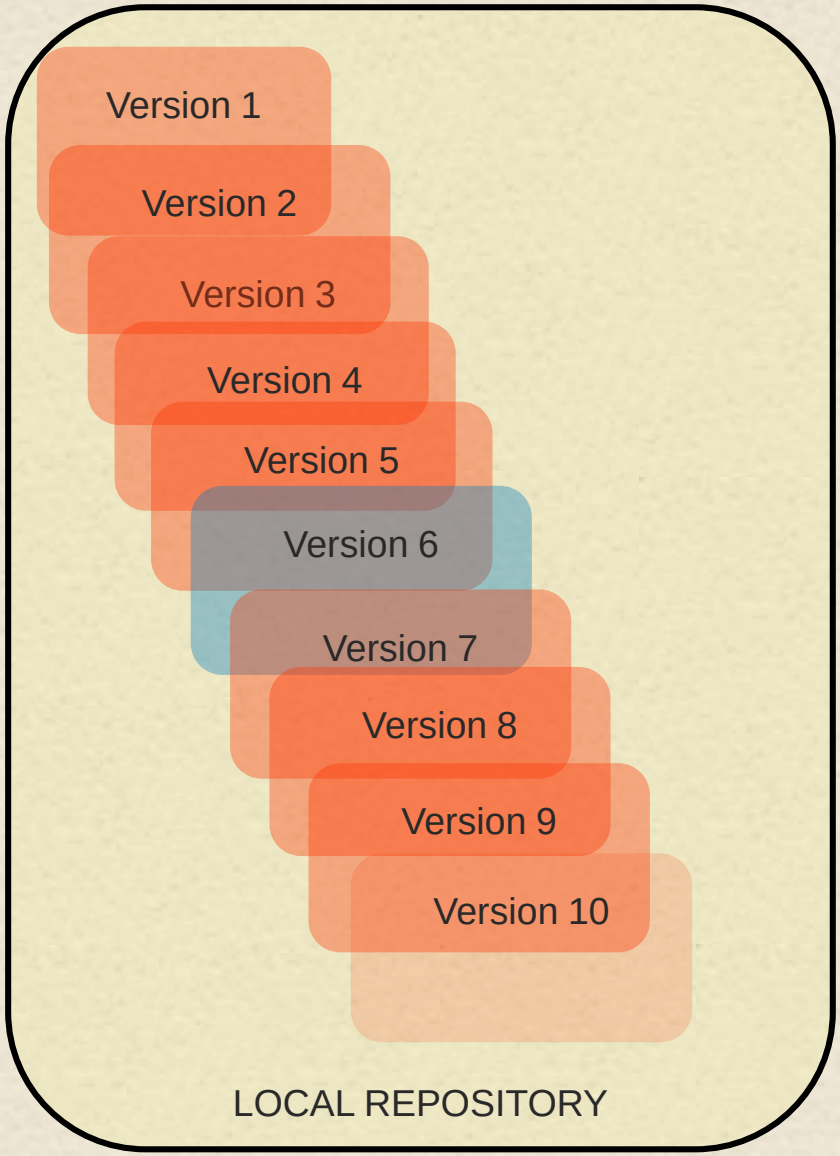
Bisect

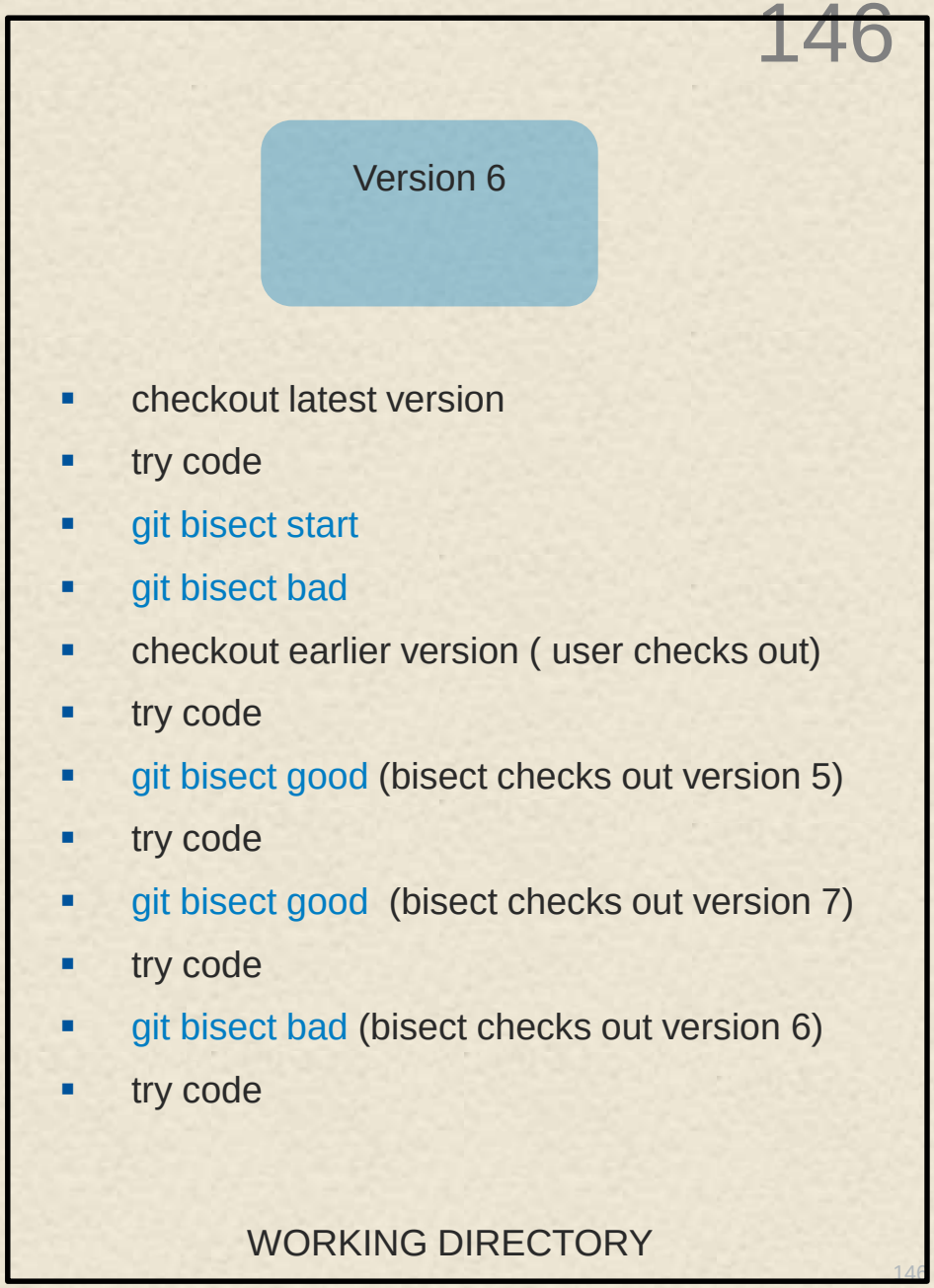




- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code
- git bisect good (bisect checks out version 7)
- try code
- git bisect bad (bisect checks out version 6)

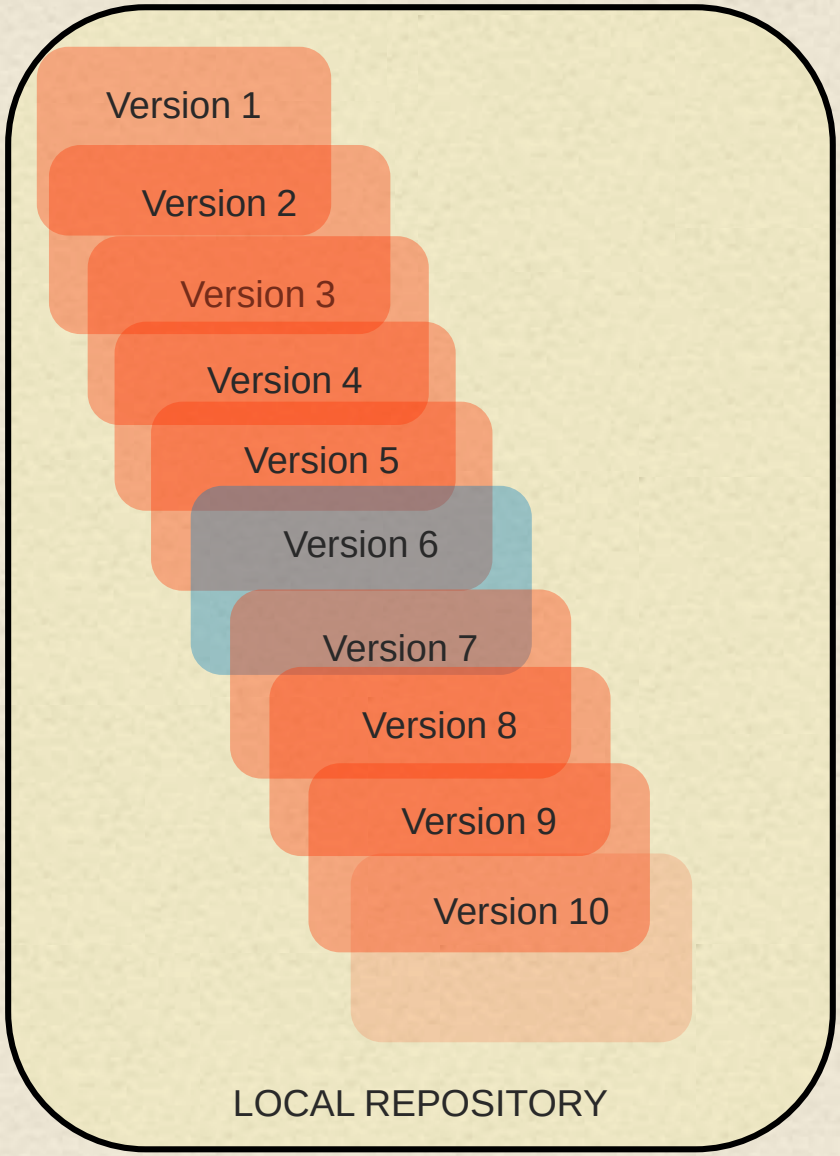
Bisect





- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code
- git bisect good (bisect checks out version 7)
- try code
- git bisect bad (bisect checks out version 6)
- try code

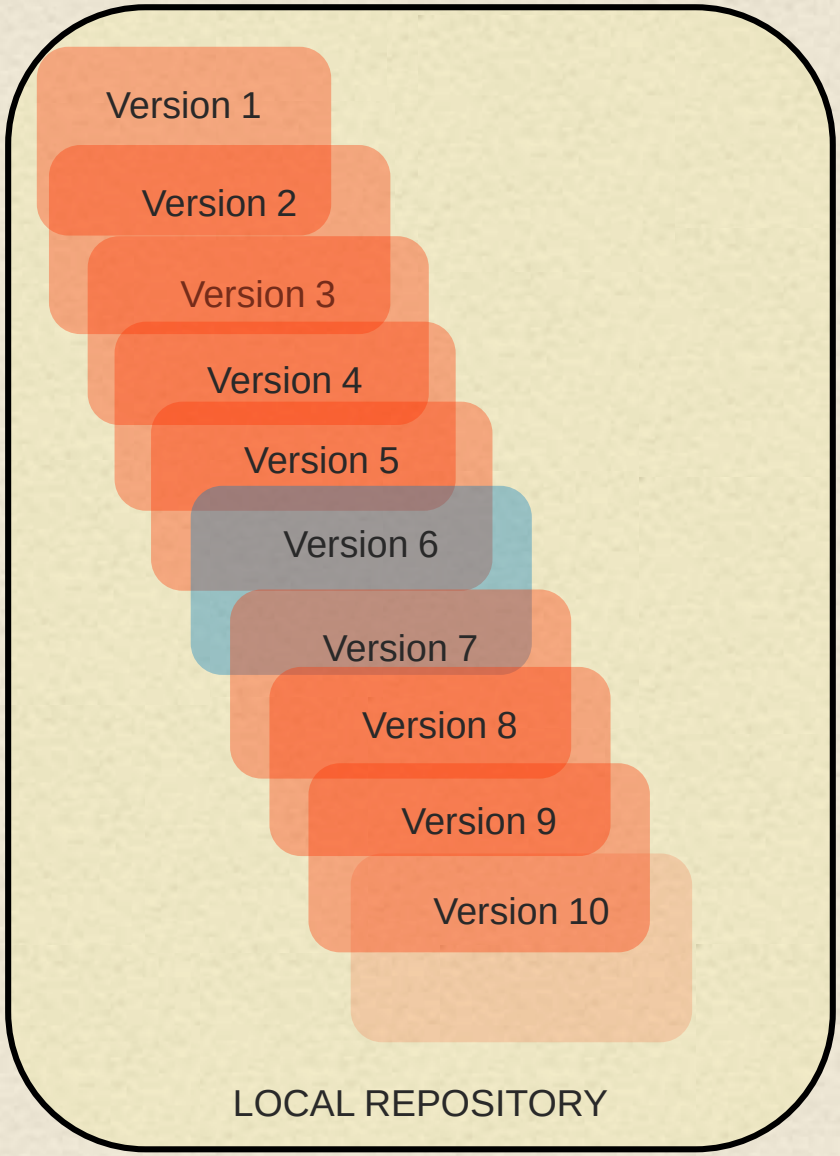
Bisect

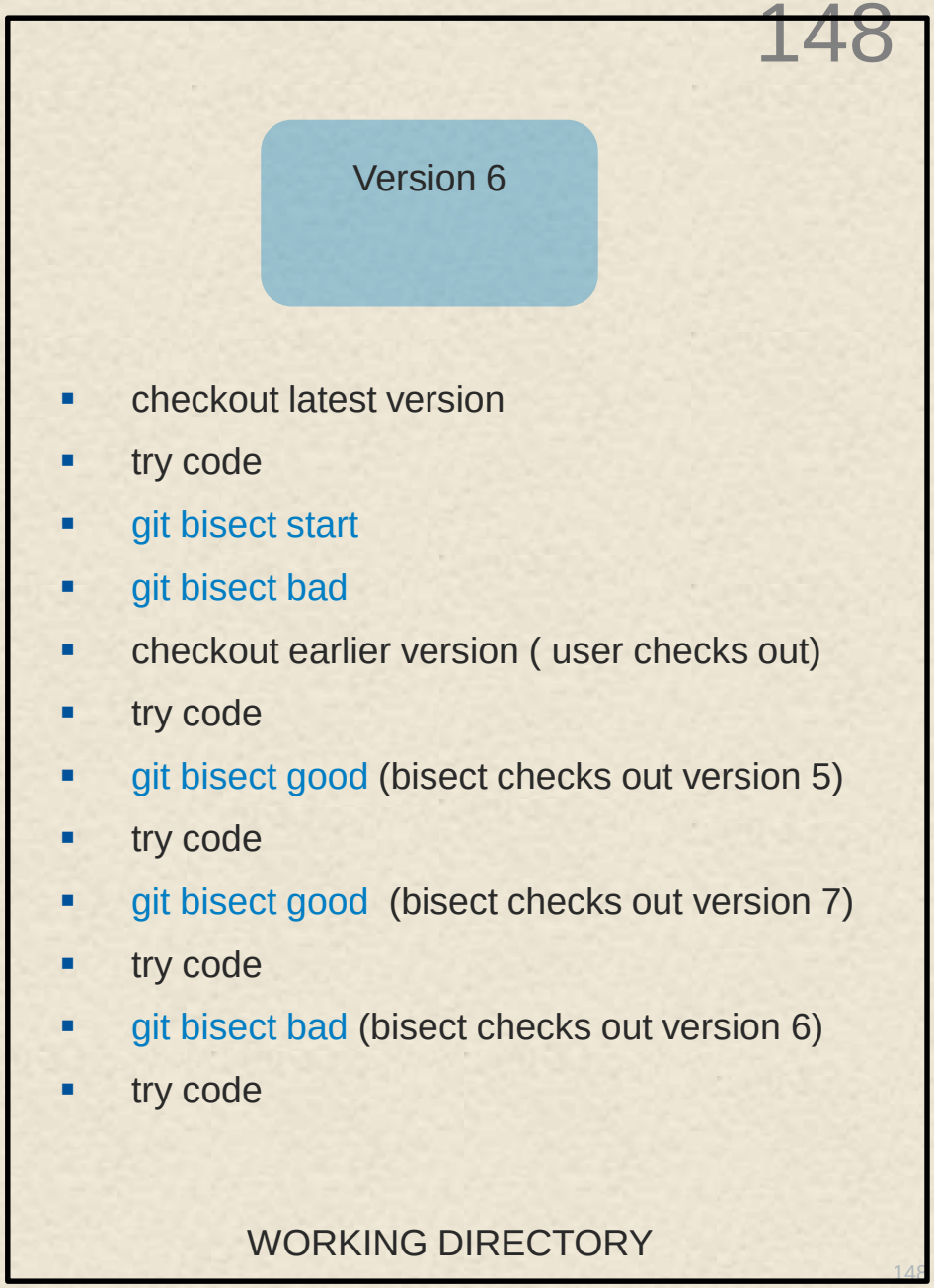


- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code
- git bisect good (bisect checks out version 7)
- try code
- git bisect bad (bisect checks out version 6)
- try code

WORKING DIRECTORY

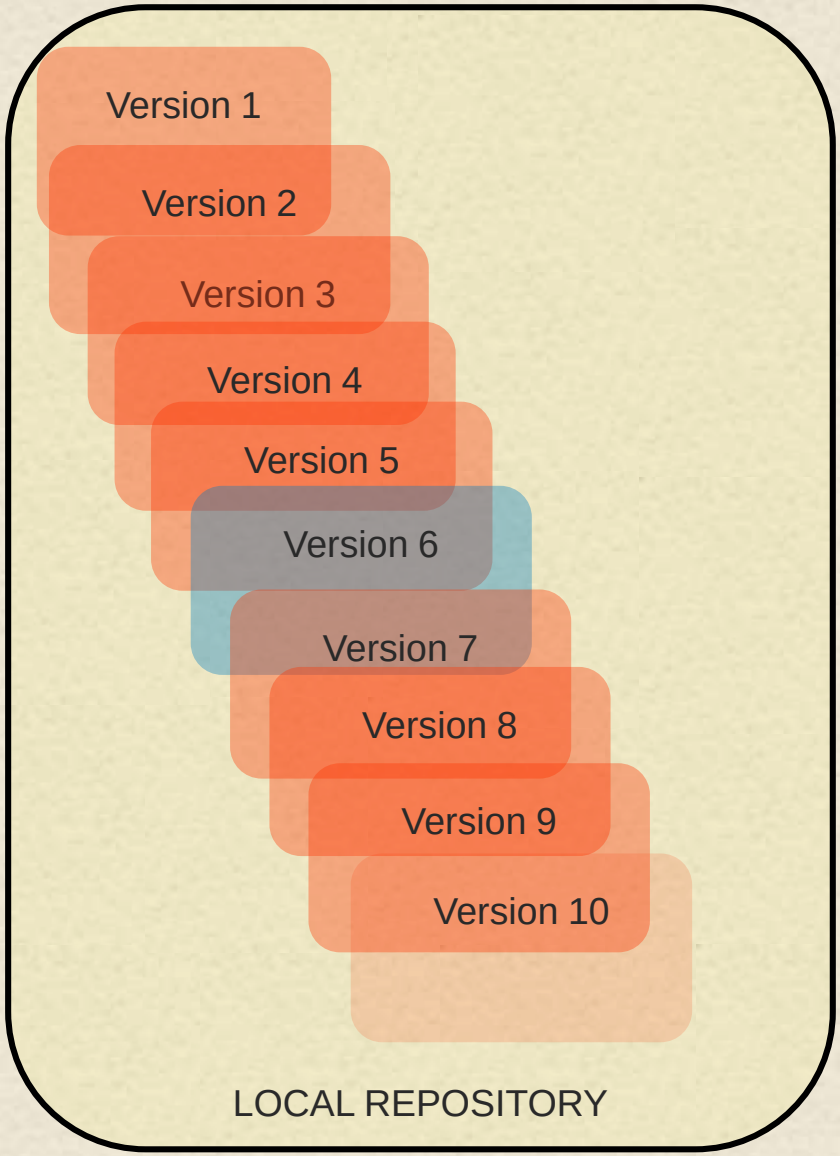
Bisect

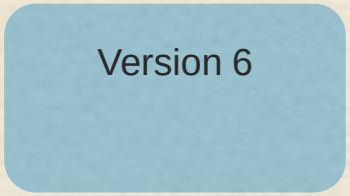




- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code
- git bisect good (bisect checks out version 7)
- try code
- git bisect bad (bisect checks out version 6)
- try code

Bisect

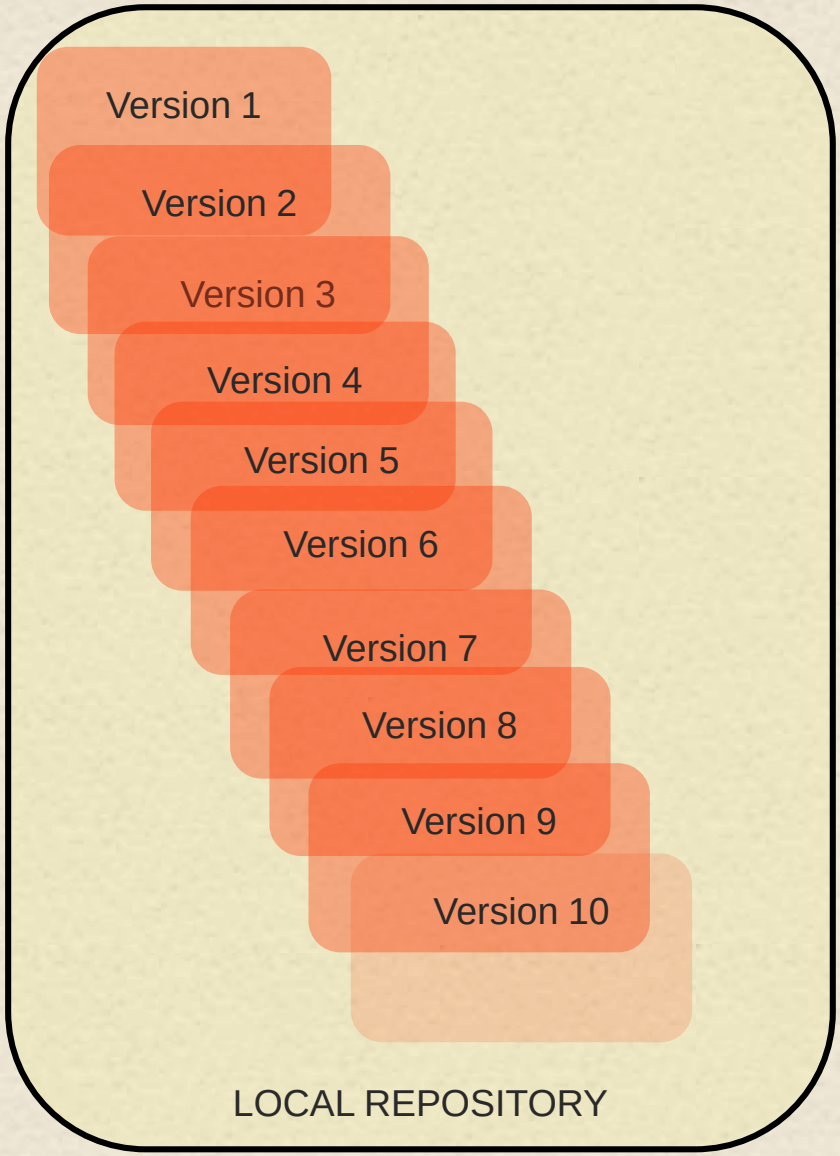


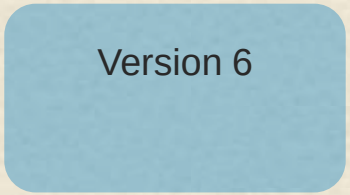


- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code
- git bisect good (bisect checks out version 7)
- try code
- git bisect bad (bisect checks out version 6)
- try code
- git bisect bad (git reports version 6 as the first bad commit)

WORKING DIRECTORY

Bisect



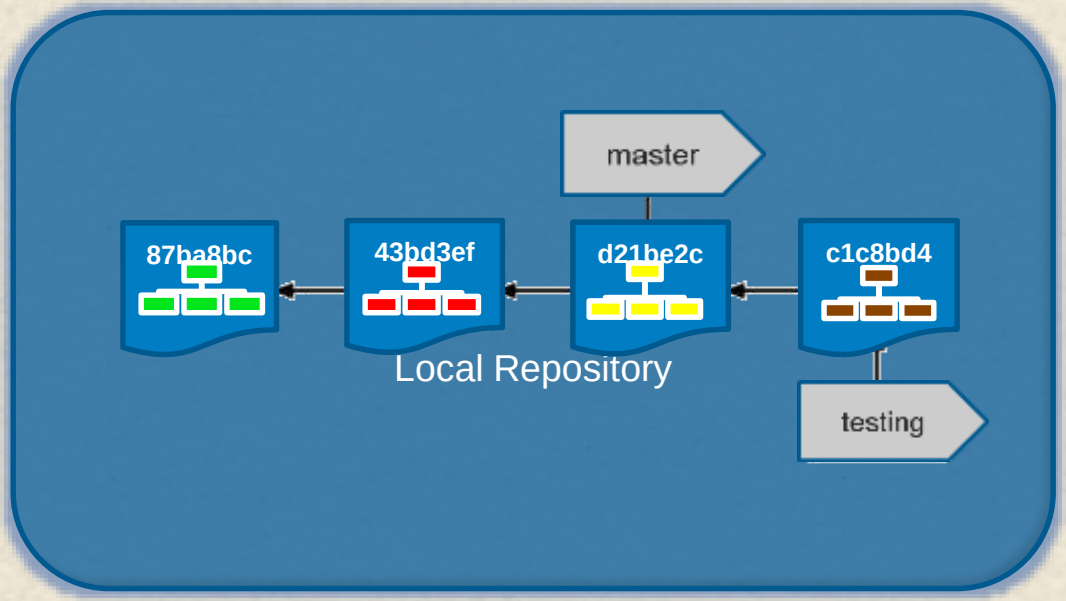


FIRST BAD COMMIT

- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code
- git bisect good (bisect checks out version 7)
- try code
- git bisect bad (bisect checks out version 6)
- try code
- git bisect bad (git reports version 6 as the first bad commit)

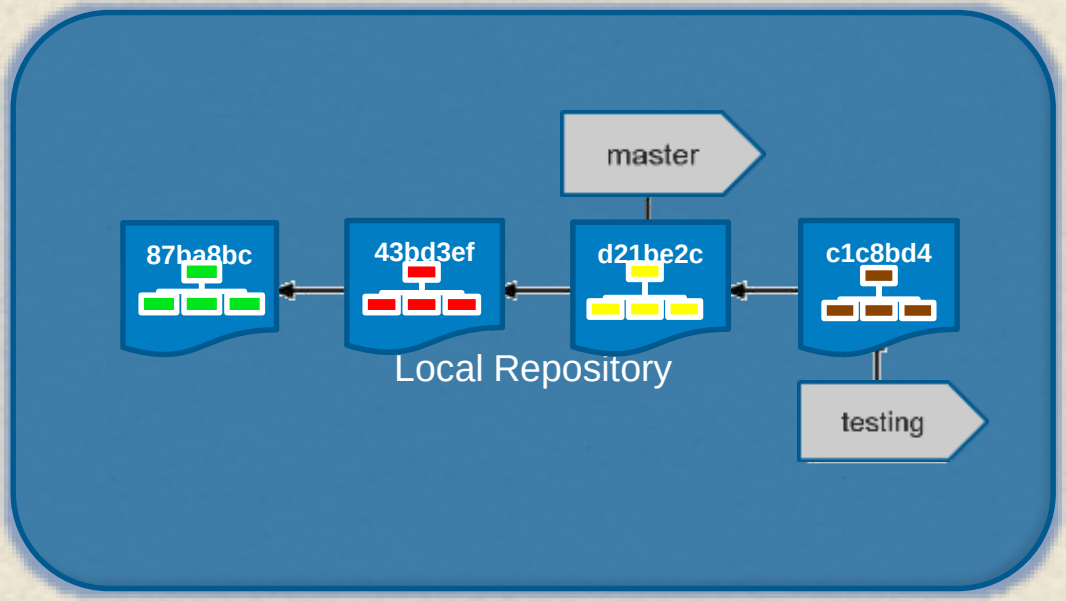
WORKING DIRECTORY

Background: Switching between Branches



Background: Switching between Branches

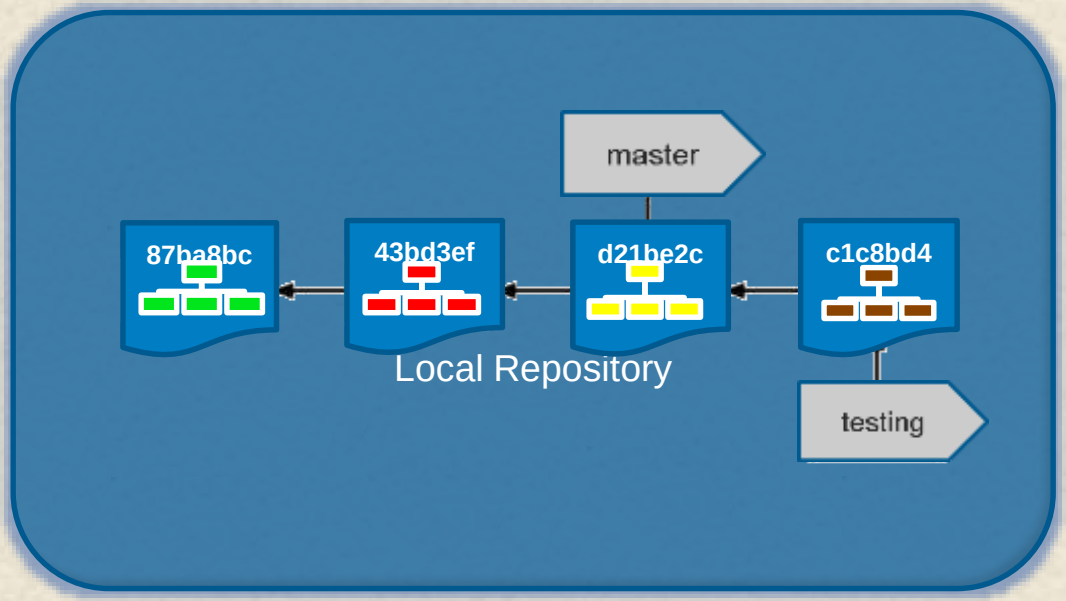
Command: git checkout <branch>



Background: Switching between Branches

Command: git checkout <branch>

git checkout master

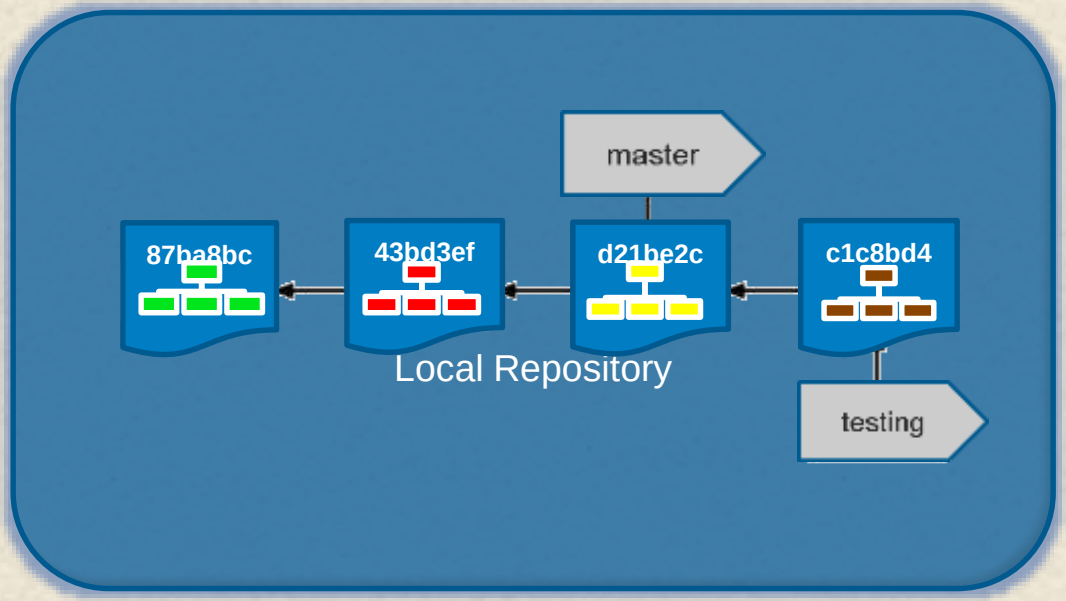


Background: Switching between Branches

Command: git checkout <branch>

git checkout master

- **Does three things**

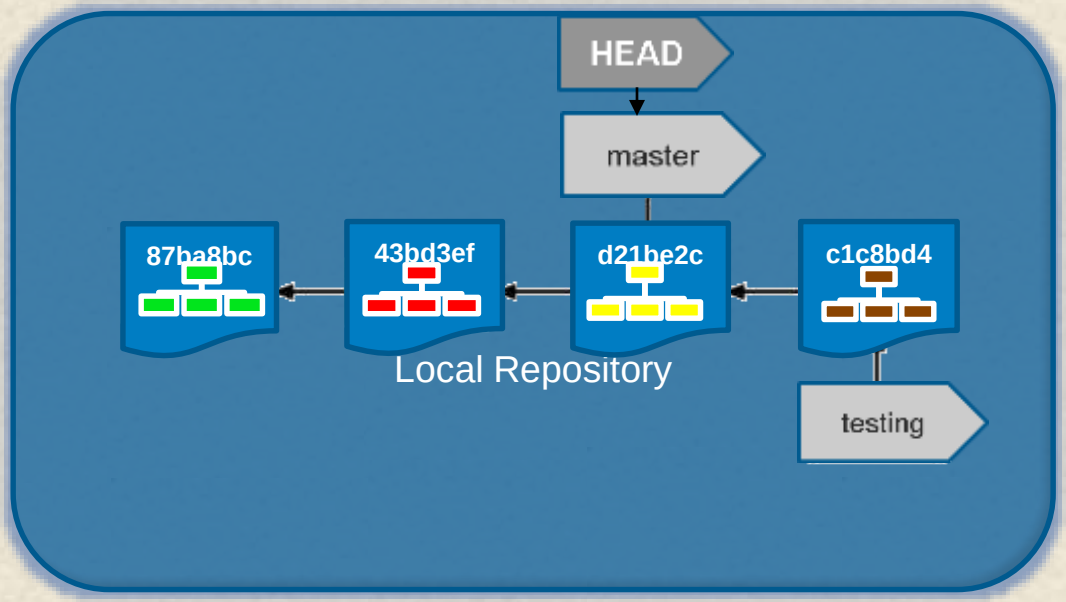


Background: Switching between Branches

Command: git checkout <branch>

git checkout master

- **Does three things**
  - **Moves HEAD pointer back to <branch>**

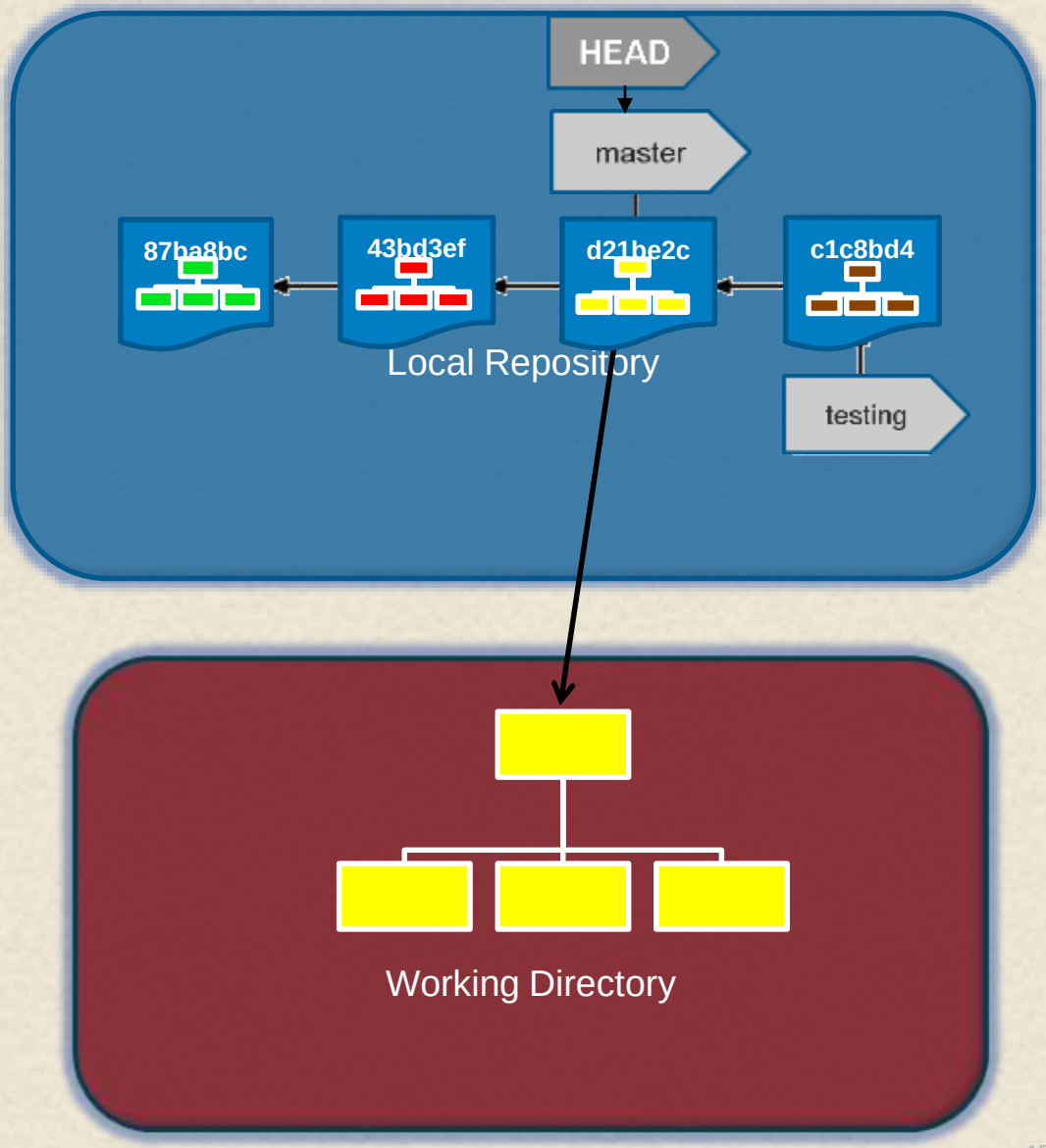


Background: Switching between Branches

Command: git checkout <branch>

git checkout master

- **Does three things**
  - **Moves HEAD pointer back to <branch>**
  - **Reverts files in working directory to snapshot pointed to by <branch>**

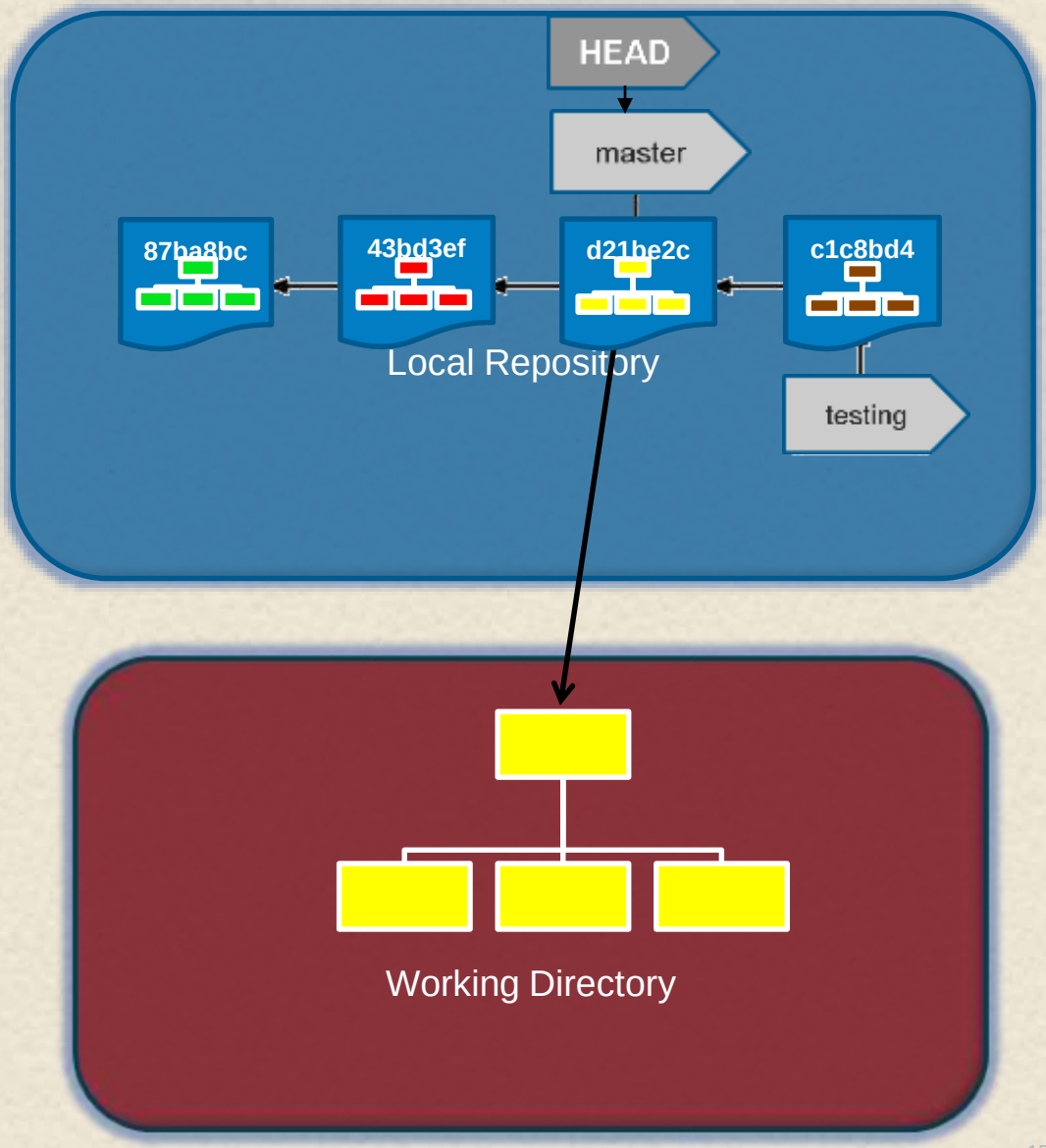


Background: Switching between Branches

Command: git checkout <branch>

git checkout master

- **Does three things**
  - **Moves HEAD pointer back to <branch>**
  - **Reverts files in working directory to snapshot pointed to by <branch>**
  - **Updates indicators**

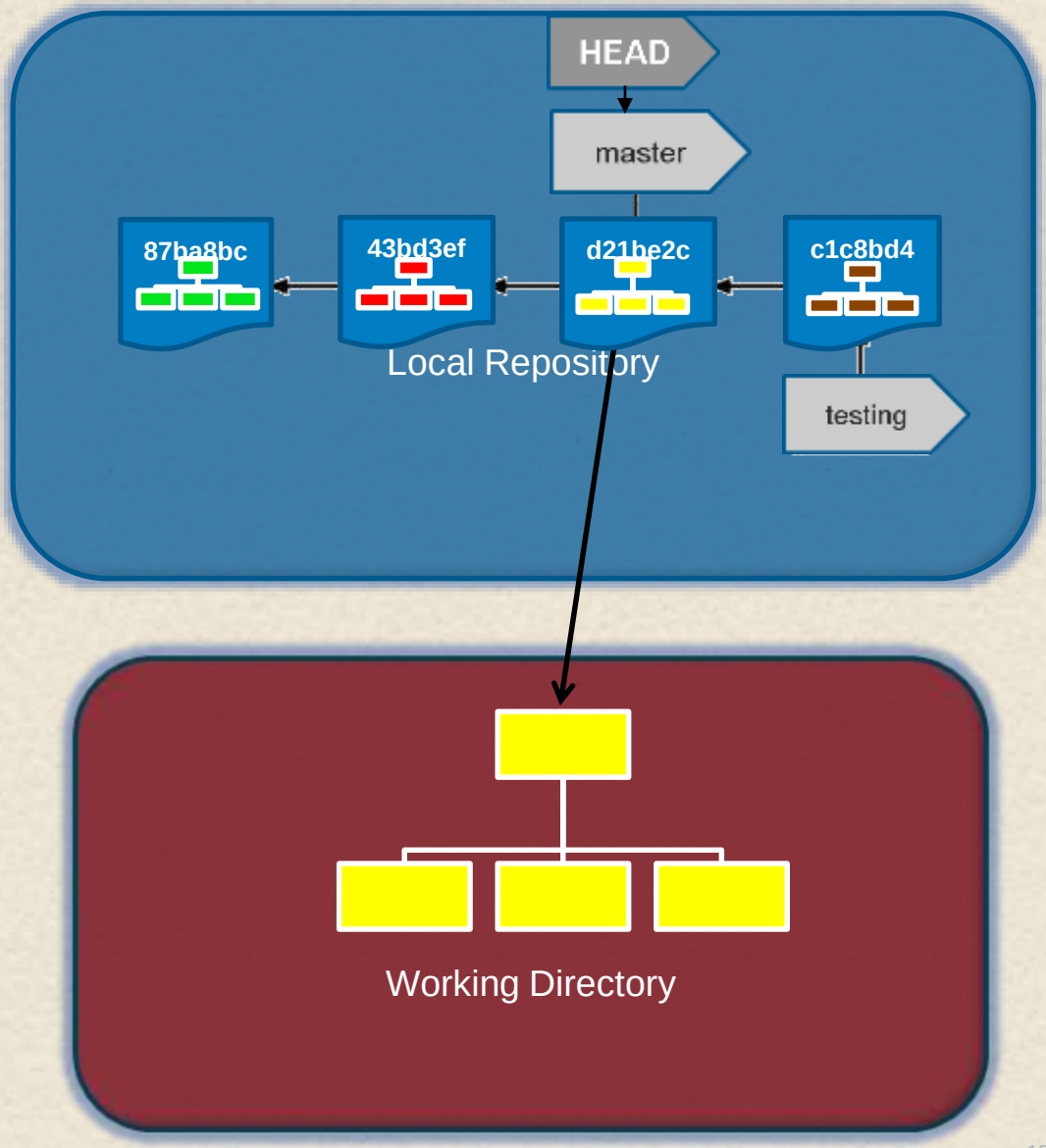


Background: Switching between Branches

Command: git checkout <branch>

git checkout master

- **Does three things**
  - **Moves HEAD pointer back to <branch>**
  - **Reverts files in working directory to snapshot pointed to by <branch>**
  - **Updates indicators**



- **git branch**

  * master
  testing

Background: Switching between Branches

Command: git checkout <branch>

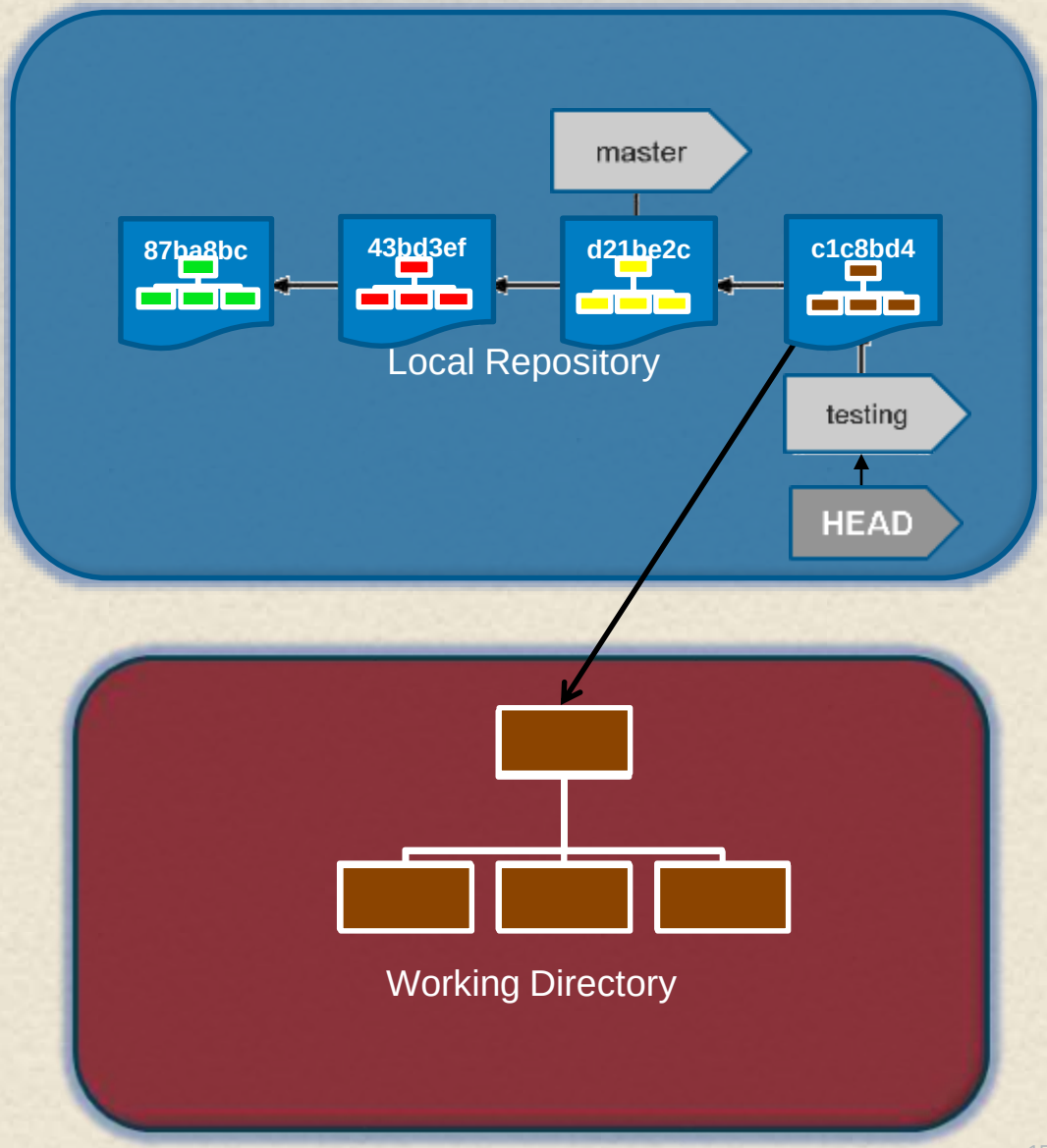
git checkout master

- **Does three things**
  - **Moves HEAD pointer back to <branch>**
  - **Reverts files in working directory to snapshot pointed to by <branch>**
  - **Updates indicators**

git checkout testing

- **git branch**

```
  master
* testing
```



Background: Switching between Branches

Command: git checkout <branch>

git checkout master

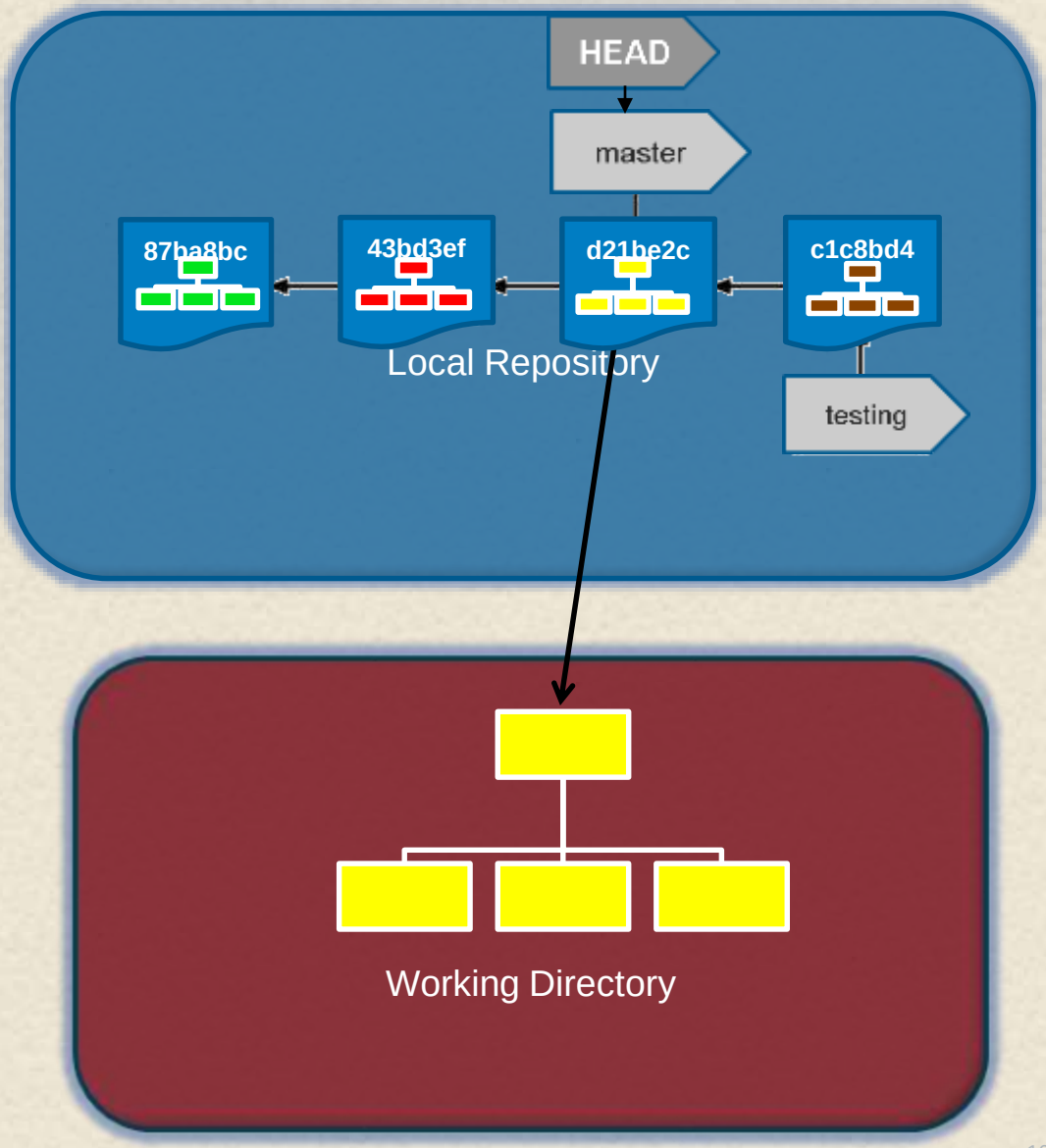
- **Does three things**
  - **Moves HEAD pointer back to <branch>**
  - **Reverts files in working directory to snapshot pointed to by <branch>**
  - **Updates indicators**

git checkout testing

git checkout master

- **git branch**

```
* master
  testing
```



Background: Switching between Branches

Command: git checkout <branch>

git checkout master

- **Does three things**
  - **Moves HEAD pointer back to <branch>**
  - **Reverts files in working directory to snapshot pointed to by <branch>**
  - **Updates indicators**

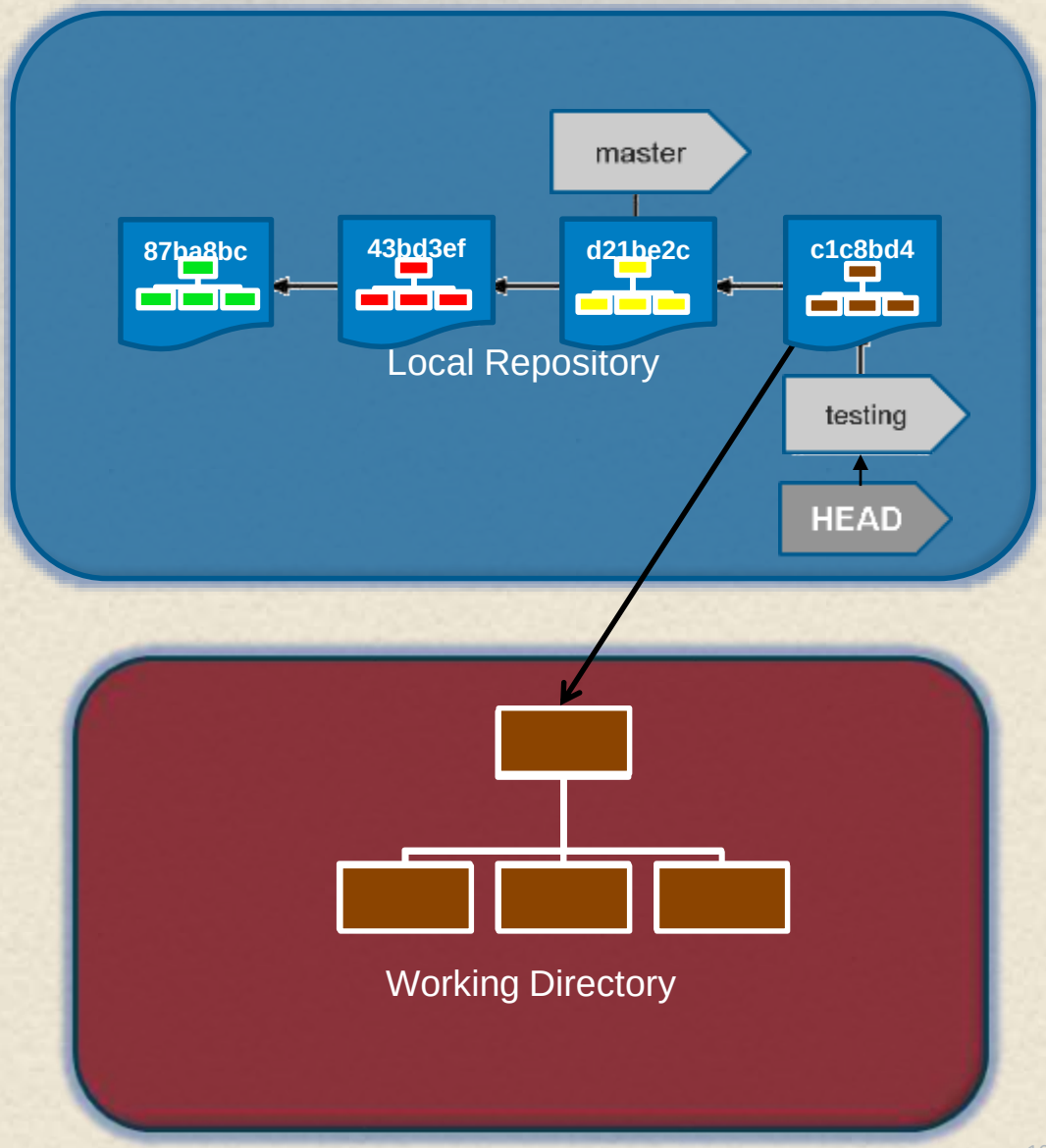
git checkout testing

git checkout master

git checkout testing

- **git branch**

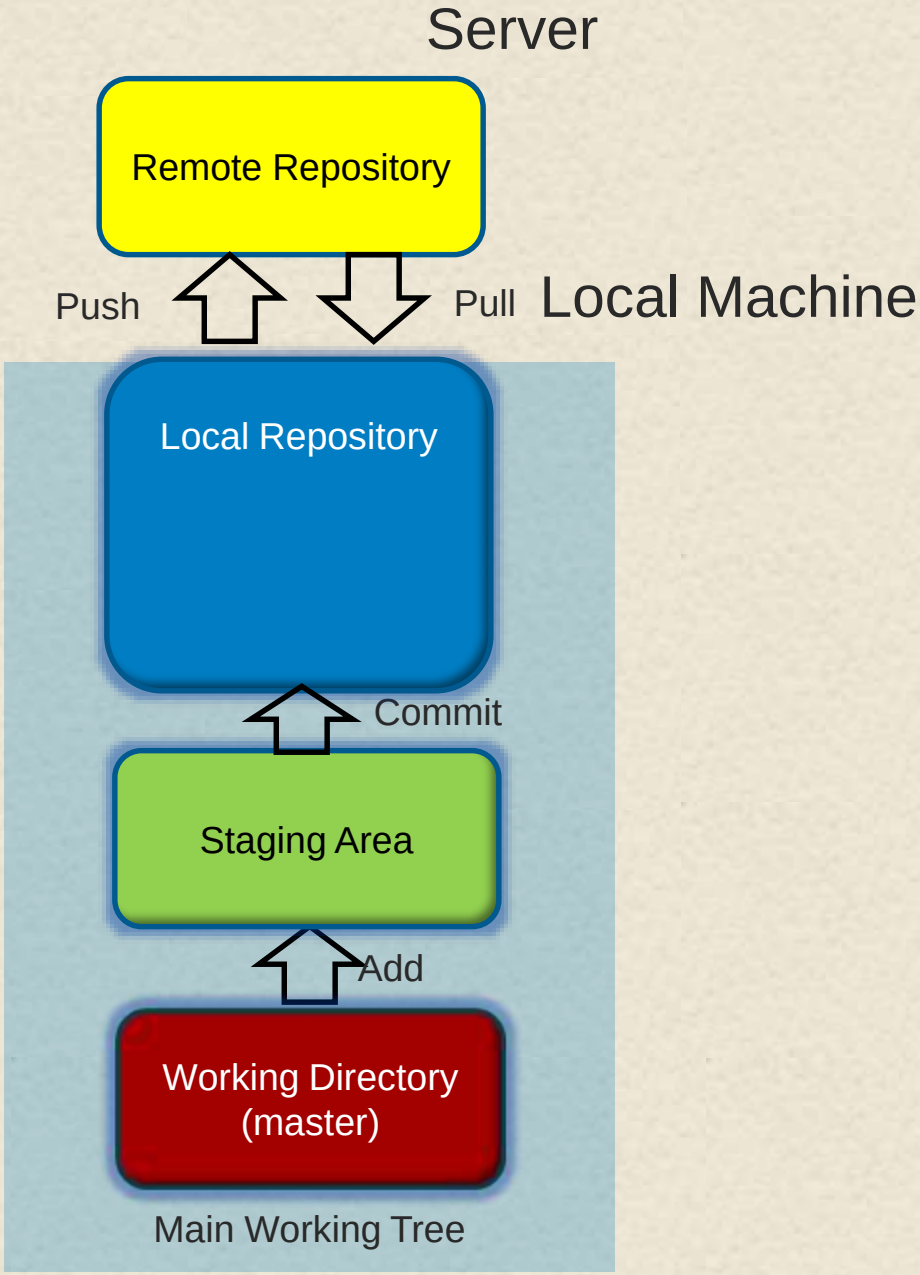
```
  master
* testing
```



Command: Worktrees

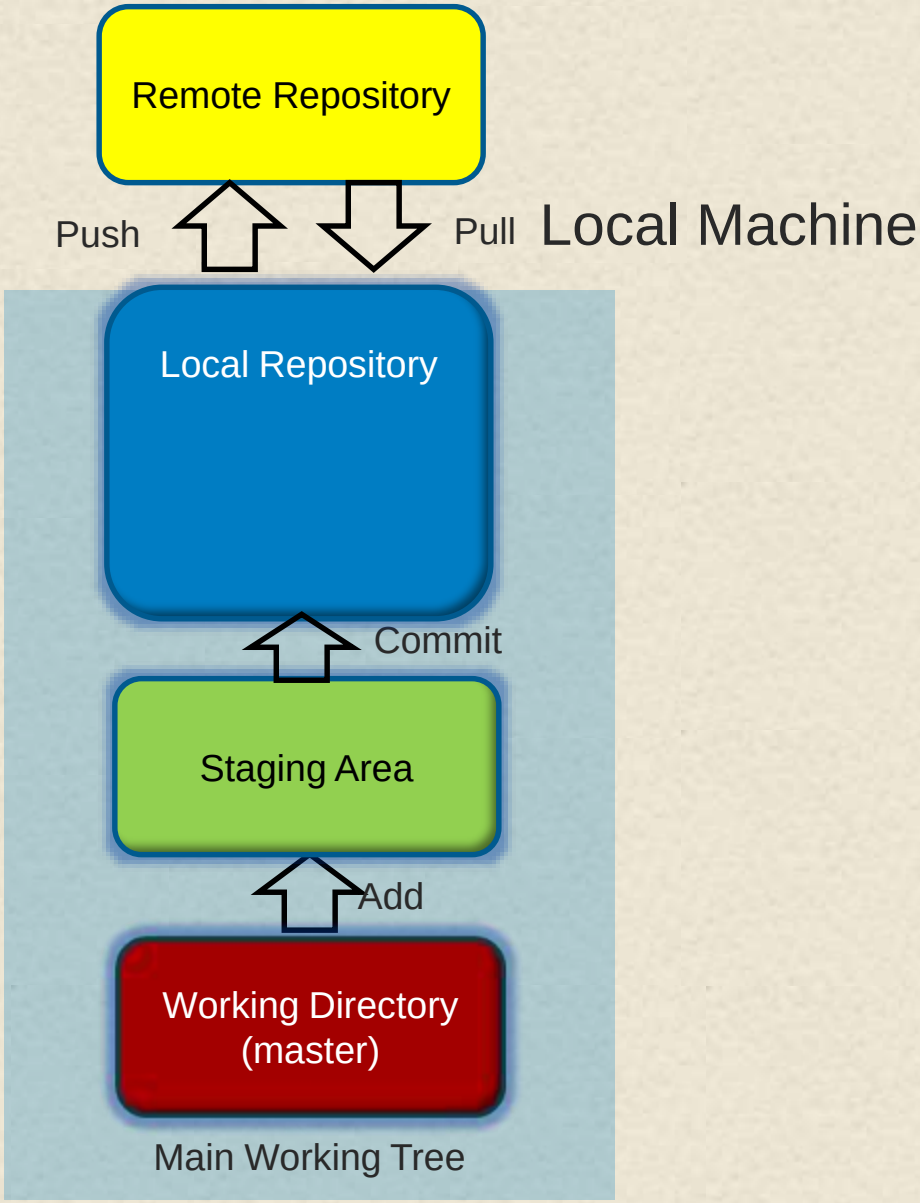
- Purpose - Allows multiple, separate Working Areas attached to one Local Repository
- Use case - Simultaneous development in multiple branche
- Syntax
  - *git worktree add* [-f] [--detach] [-b <new-branch>] <path> [<branch>]
  - *git worktree list* [--porcelain]
  - *git worktree prune* [-n] [-v] [--expire <expire>]
- Notes
  - "Traditional" working directory is called the *main working tree;* Any new trees you create with this command are called *linked working trees*
  - Information about working trees is stored in the .git area (assuming .git default GIT_DIR is used)
  - Working tree information is stored in .git/worktrees/*<name of worktree>*.

Worktrees



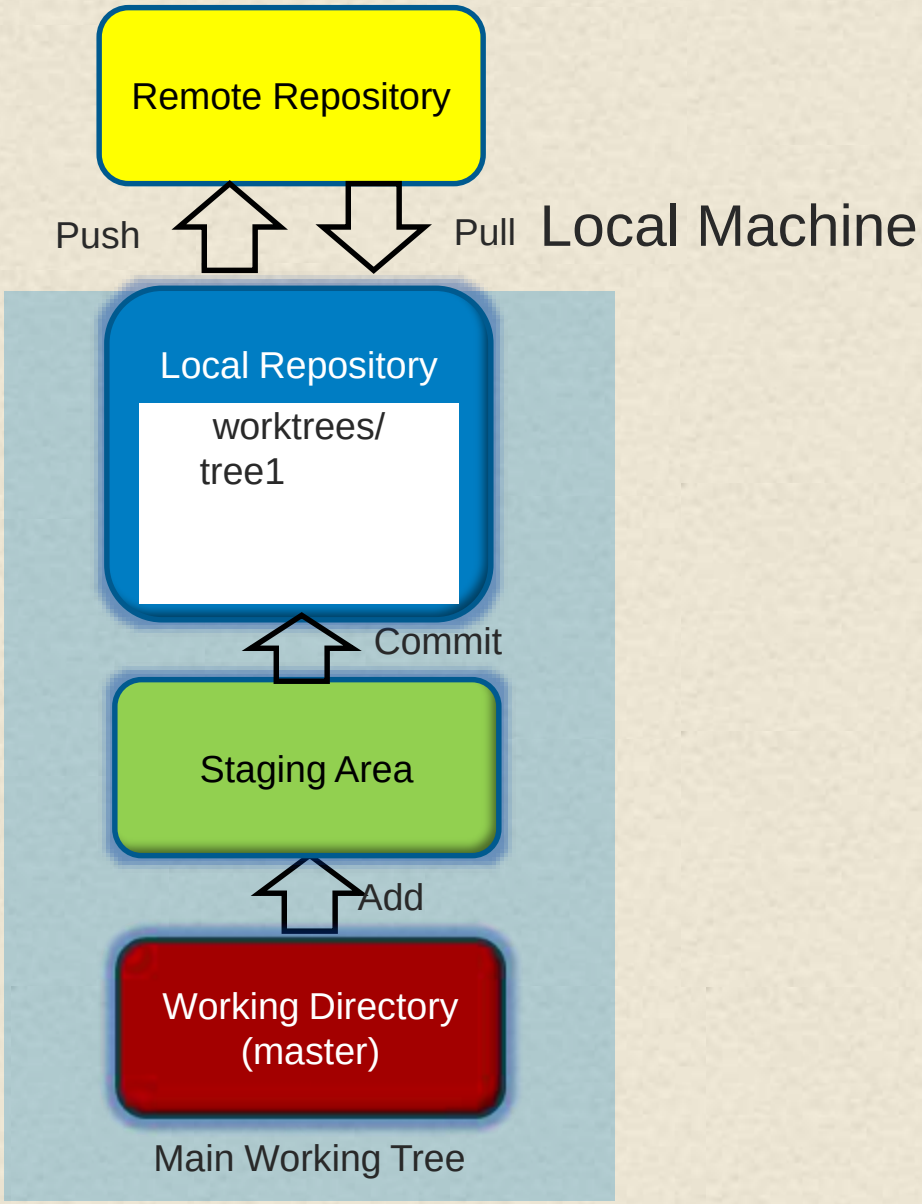
Worktrees

- git worktree add -b exp tree1



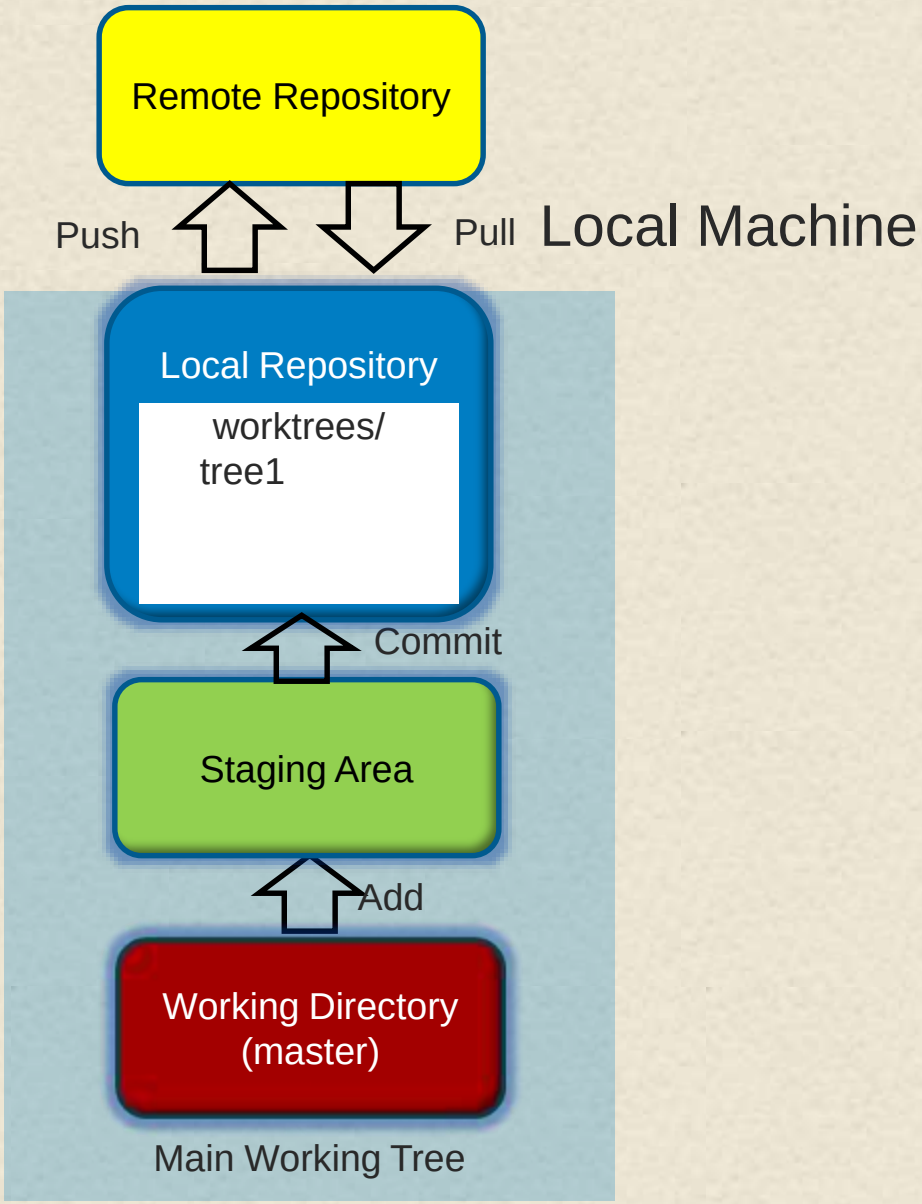
Worktrees

- git worktree add -b exp tree1



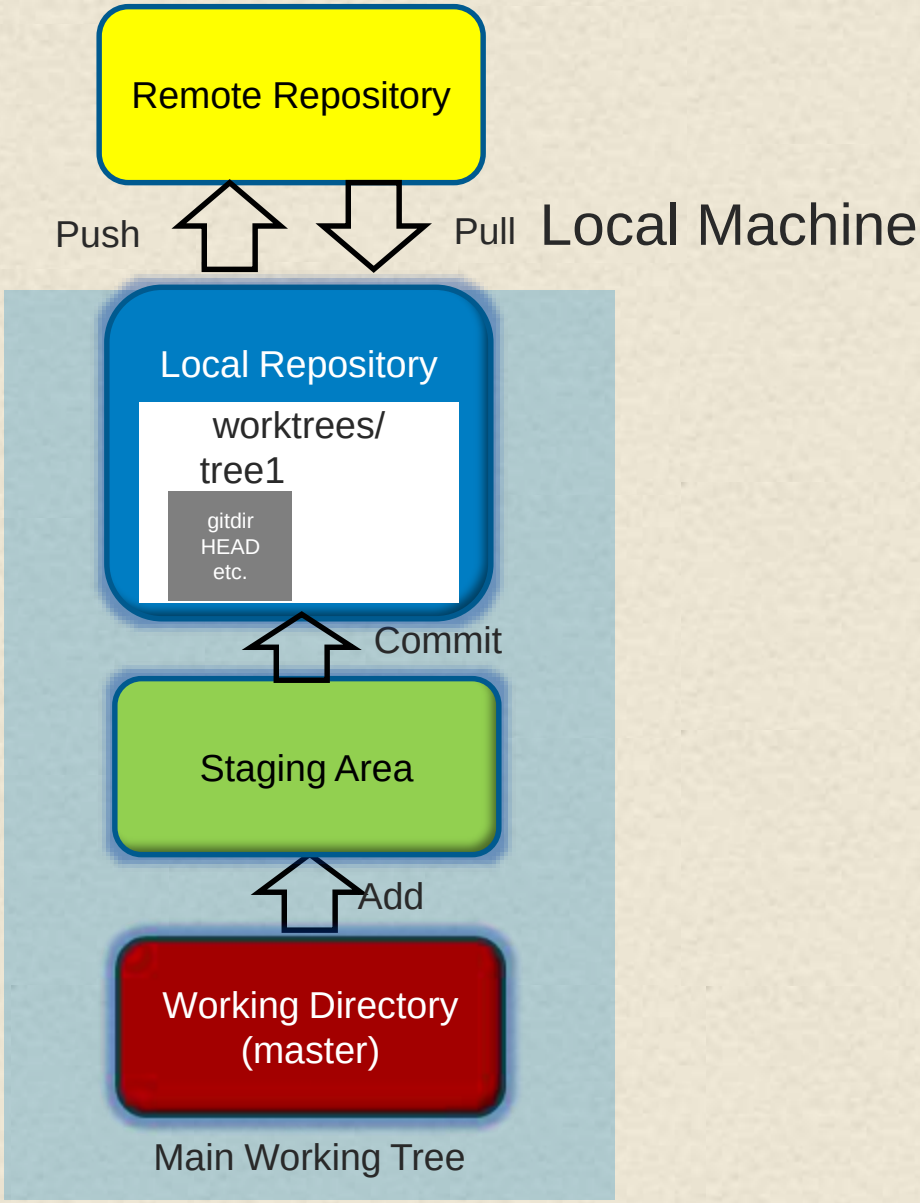
Worktrees

- git worktree add -b exp tree1



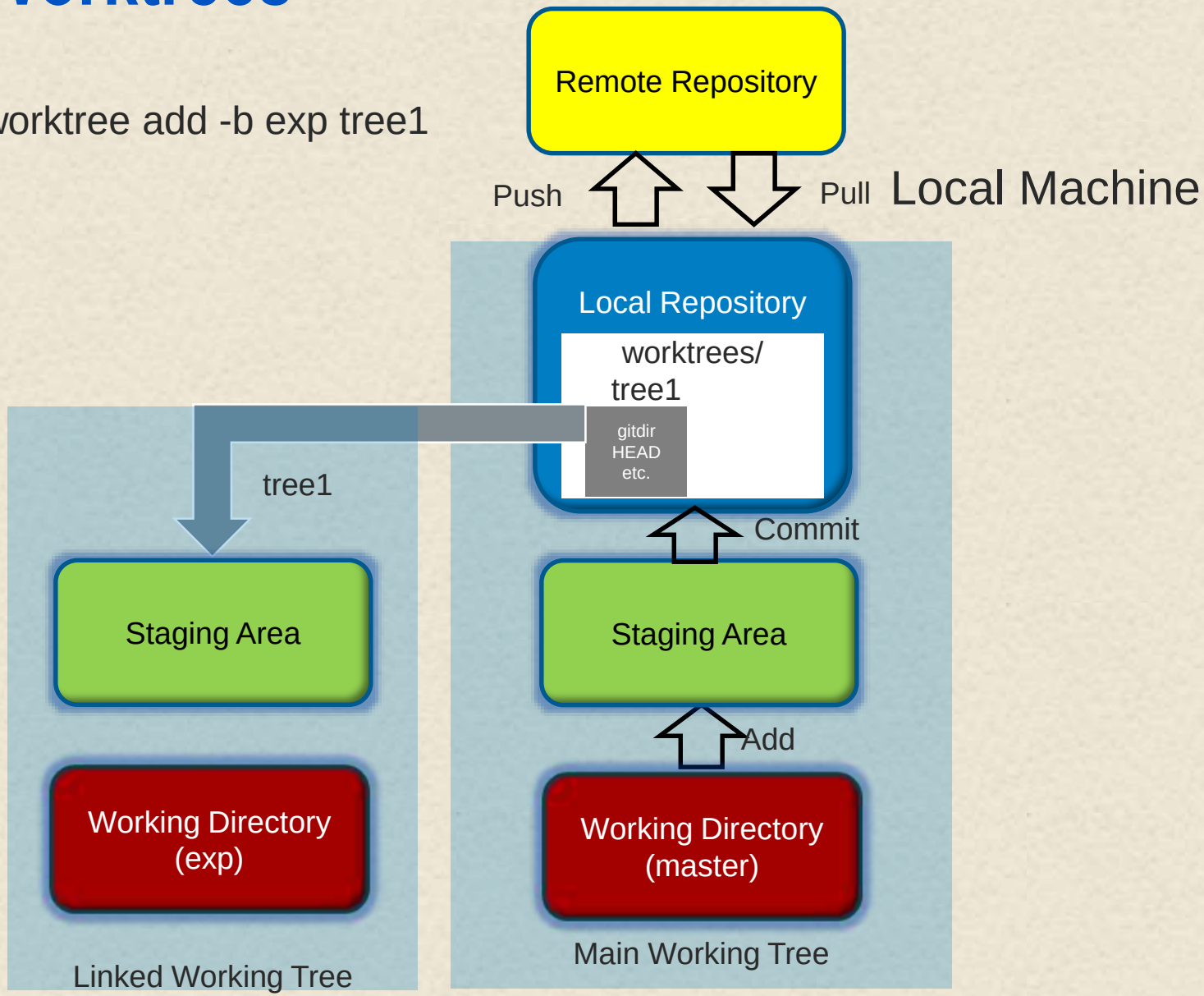
Worktrees

- git worktree add -b exp tree1



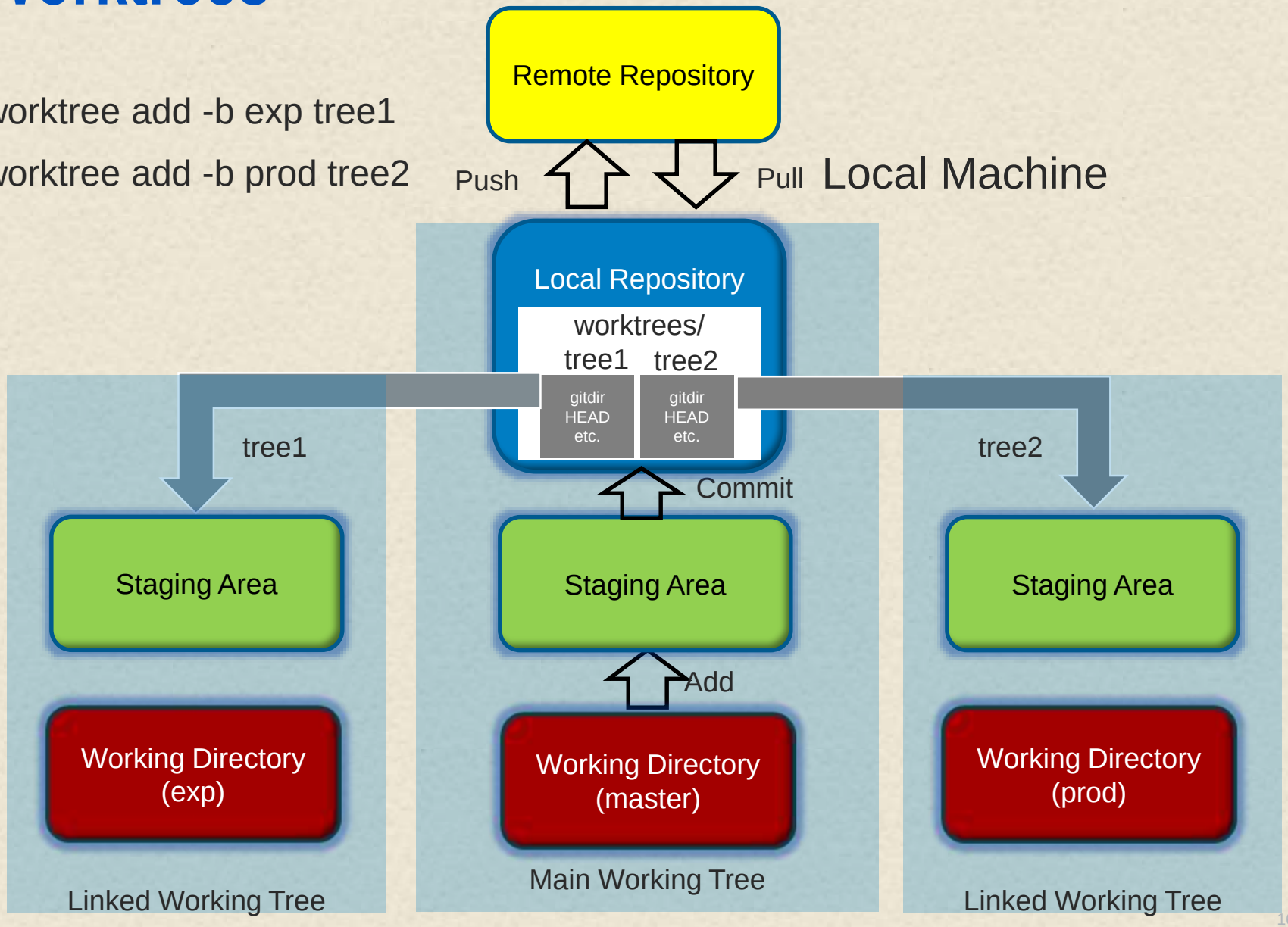
Worktrees

- git worktree add -b exp tree1



Worktrees

- git worktree add -b exp tree1
- git worktree add -b prod tree2



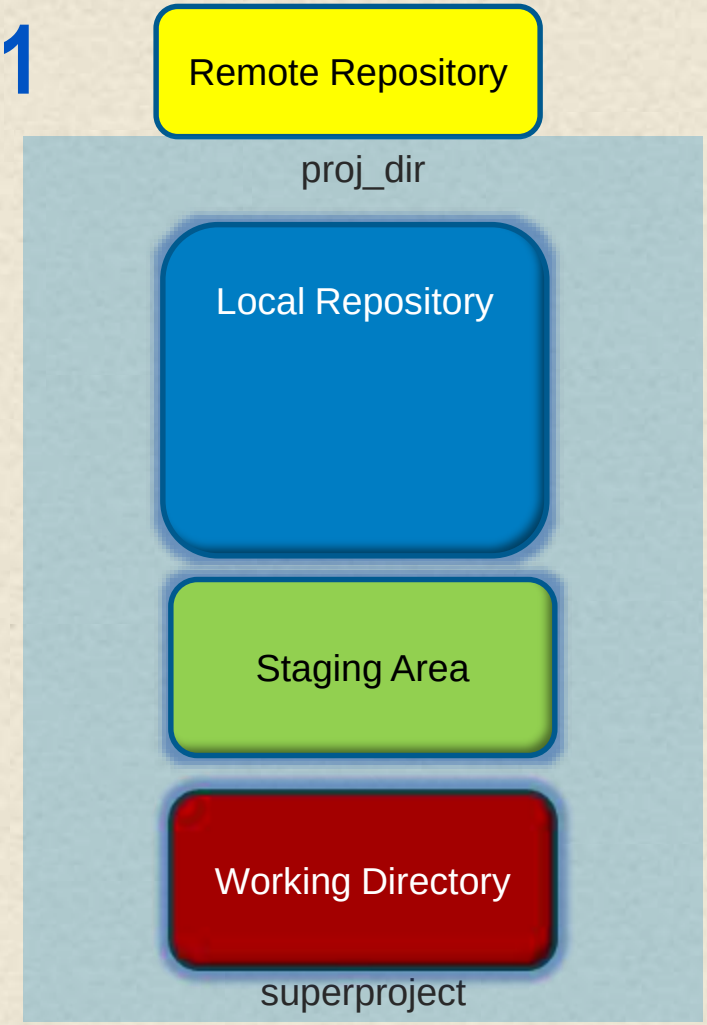
Command: Submodules

- Purpose - Allows including a separate repository with your current repository
- Use case - include the Git repository for one or more dependencies along with the original repository for a project
- Syntax

```
git submodule [--quiet] add [-b <branch>] [-f|--force] [--name <name>]
        [--reference <repository>] [--depth <depth>] [--] <repository> [<path>]
git submodule [--quiet] status [--cached] [--recursive] [--] [<path>…]
git submodule [--quiet] init [--] [<path>…]
git submodule [--quiet] deinit [-f|--force] [--] <path>…
git submodule [--quiet] update [--init] [--remote] [-N|--no-fetch]
            [-f|--force] [--rebase|--merge] [--reference <repository>]
           [--depth <depth>] [--recursive] [--] [<path>…]
git submodule [--quiet] summary [--cached|--files] [(-n|--summary-limit) <n>]
         [commit] [--] [<path>…]
git submodule [--quiet] foreach [--recursive] <command>
git submodule [--quiet] sync [--recursive] [--] [<path>…]
```

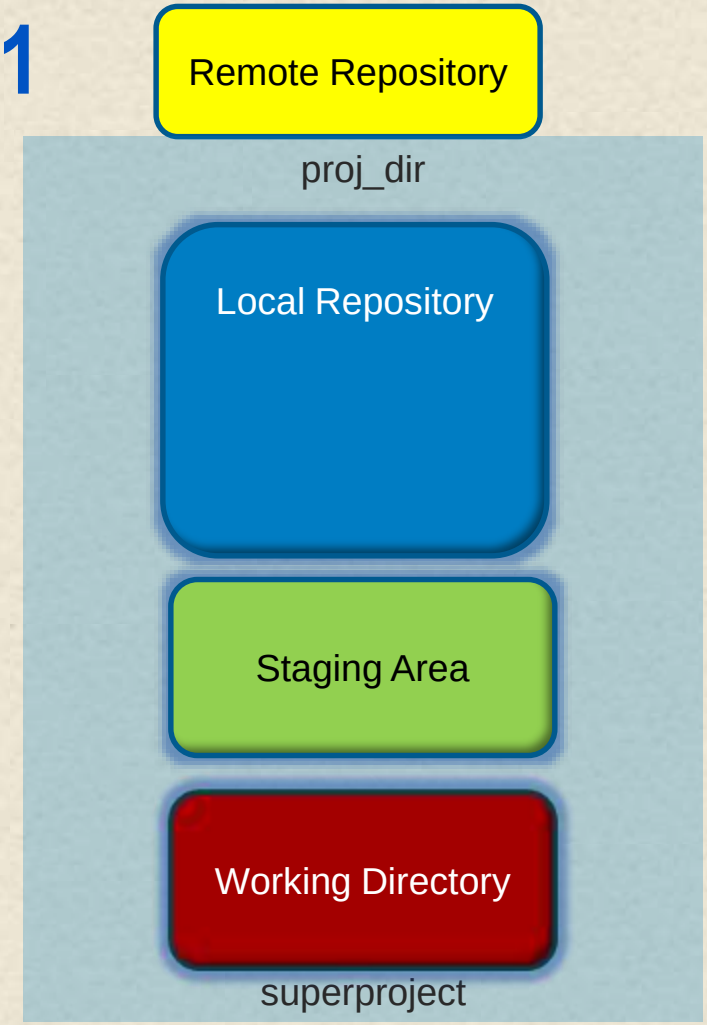
- Notes
  - Creates a subdirectory off of your original repository that contains a clone of another Git repository
  - Original repository is typically called *superproject*
  - Metadata stored in *.gitmodules* file

Submodules 1



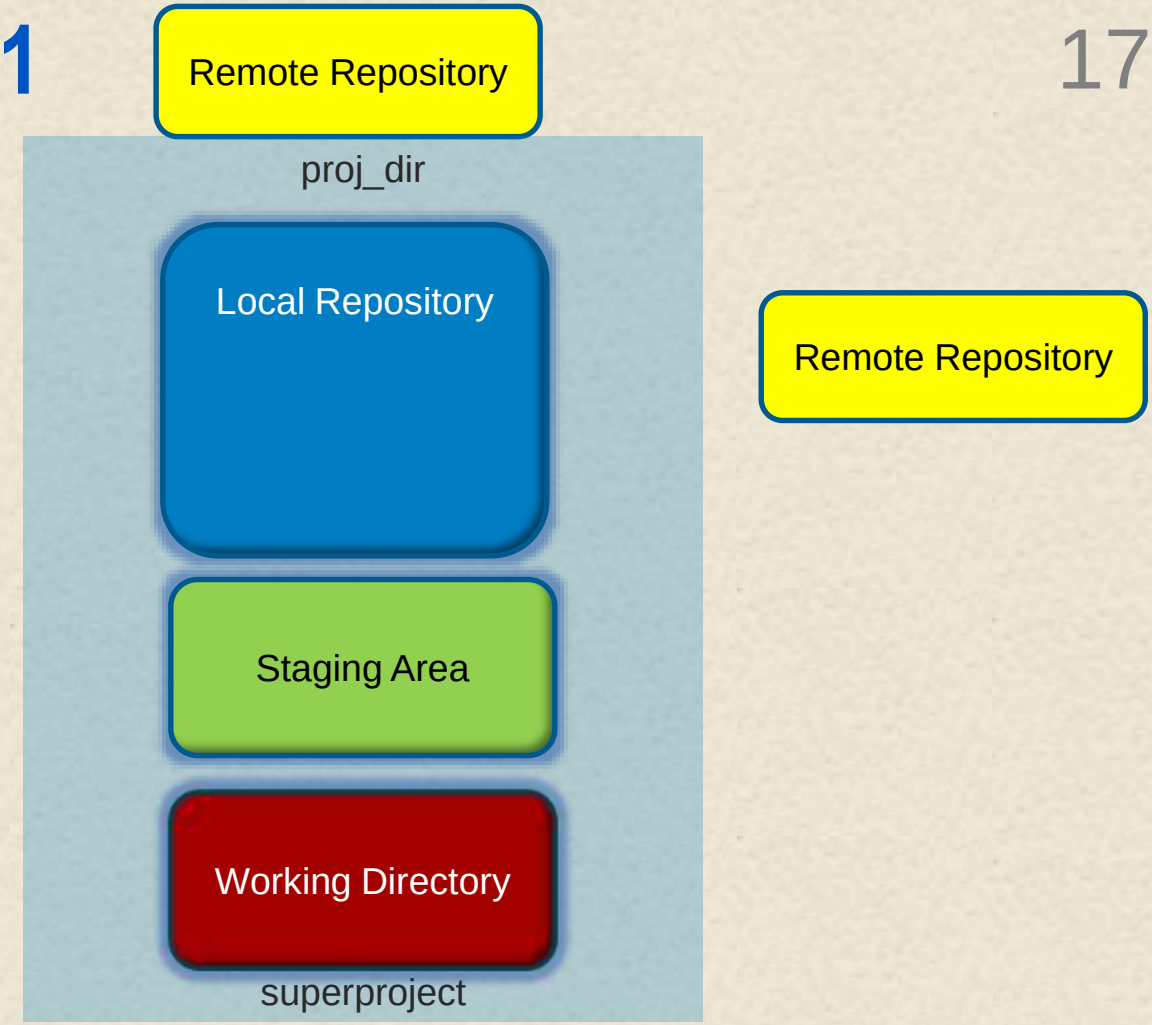
Submodules 1

What happens when you add a submodule?



Submodules 1

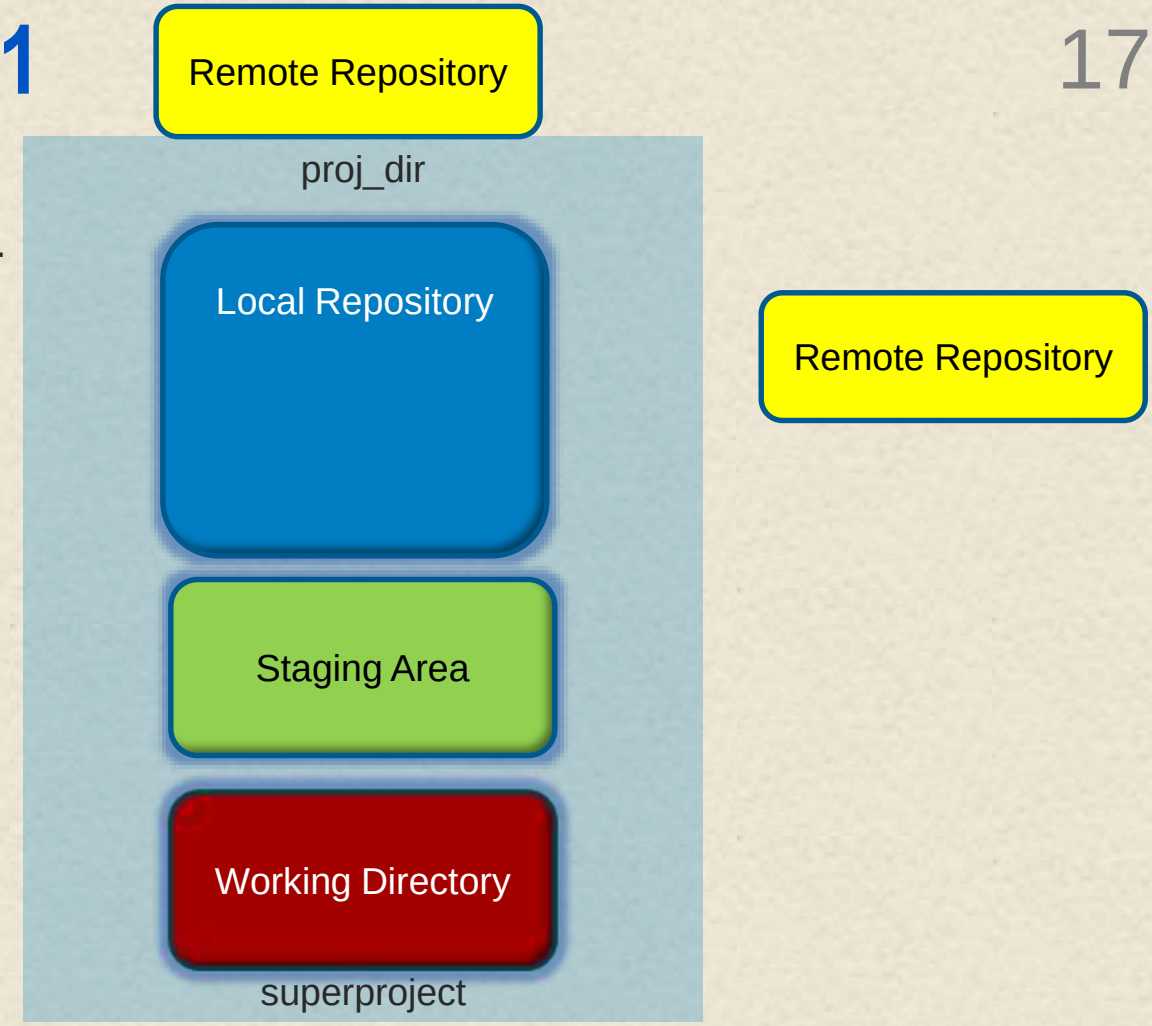
What happens when you add a submodule?



Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

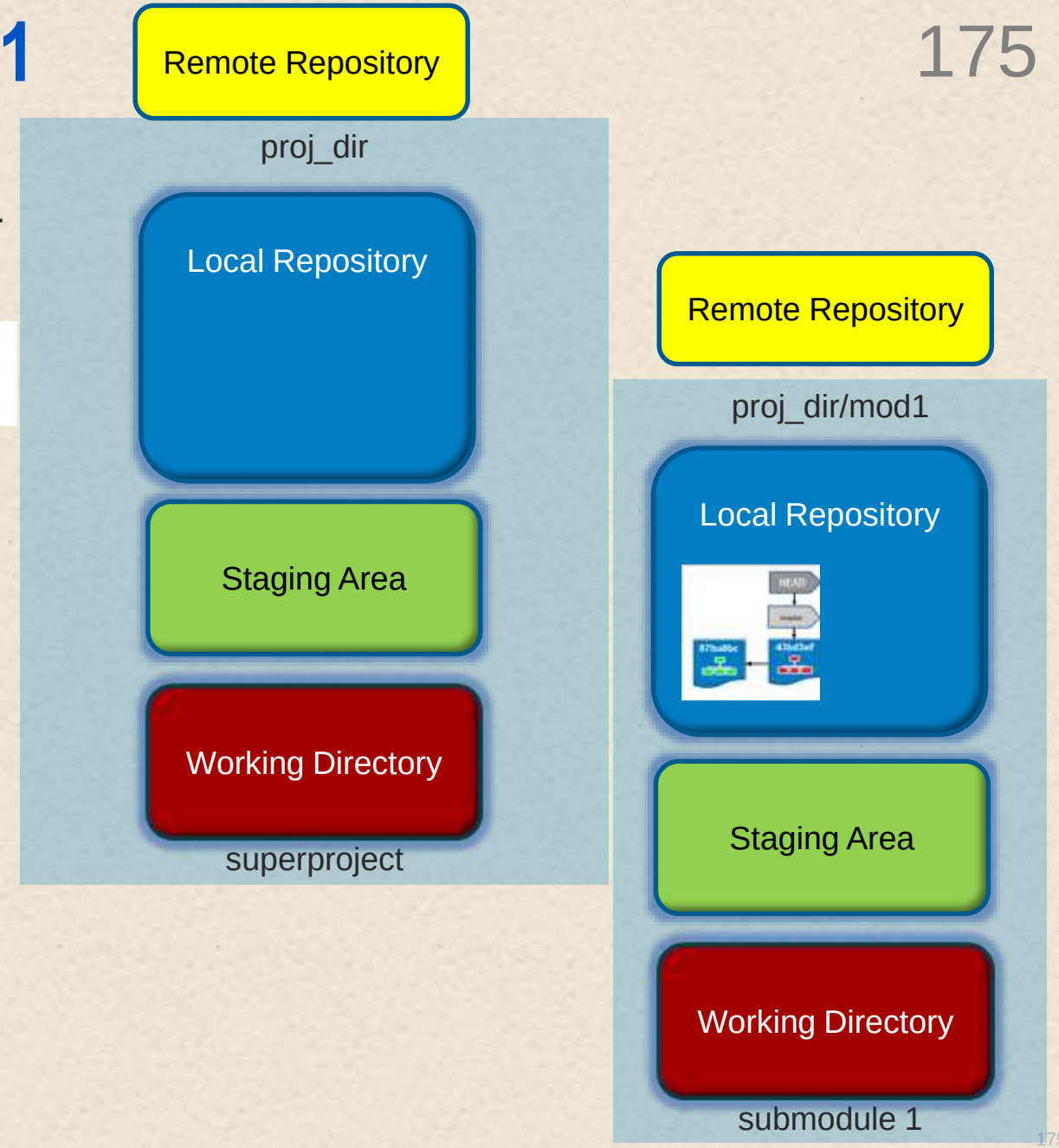


Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```



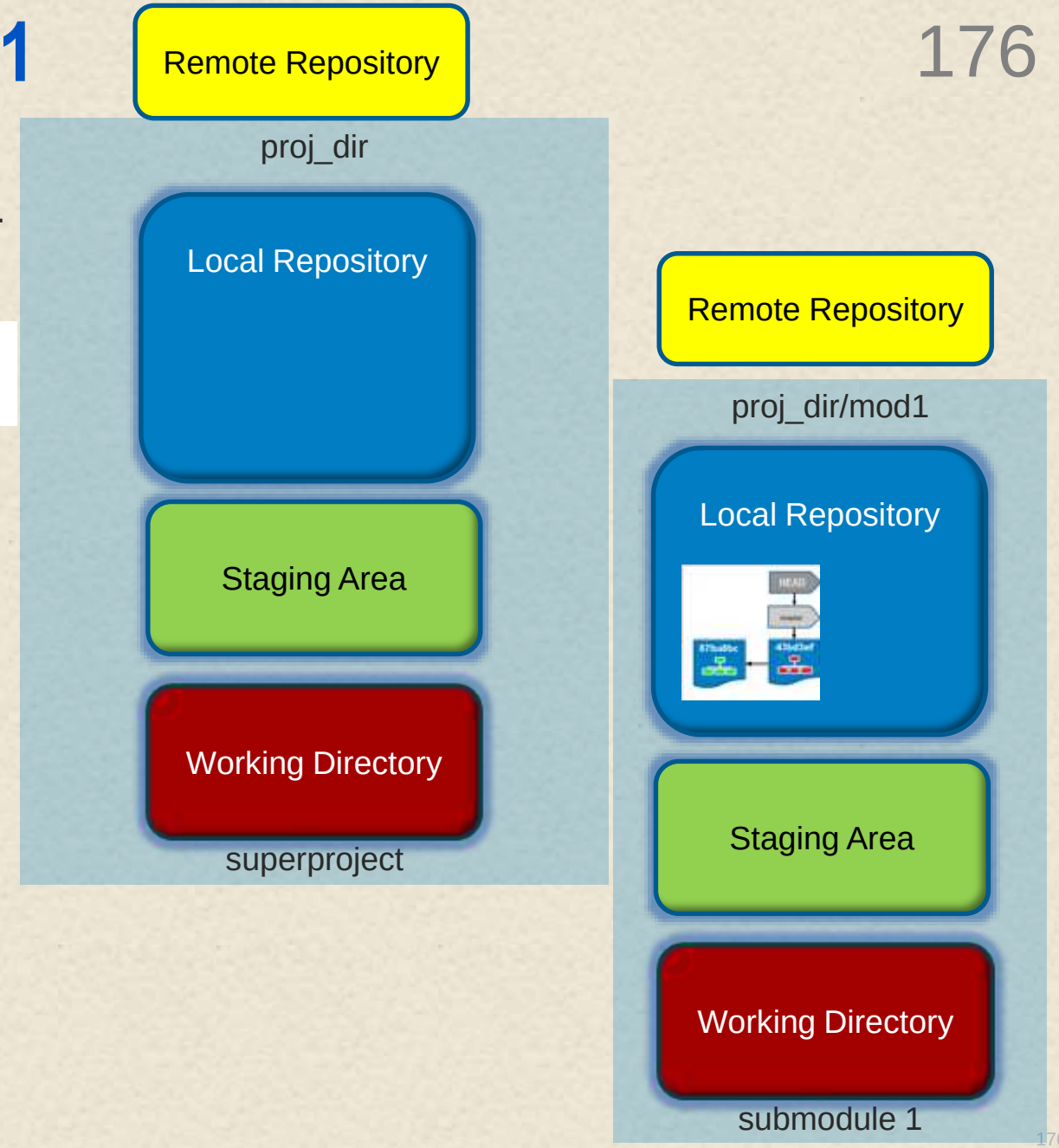
Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.



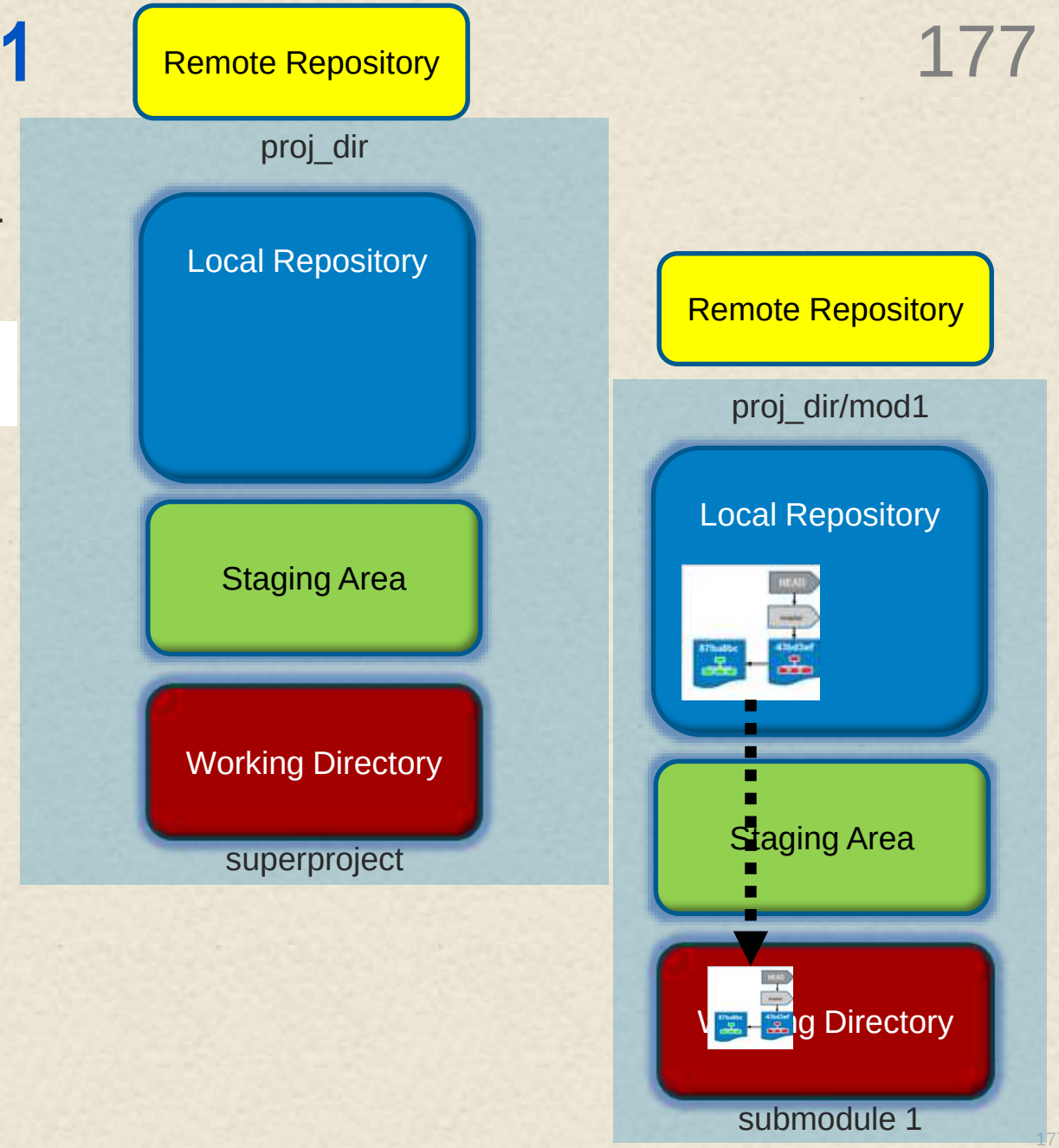
Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.



Submodules 1

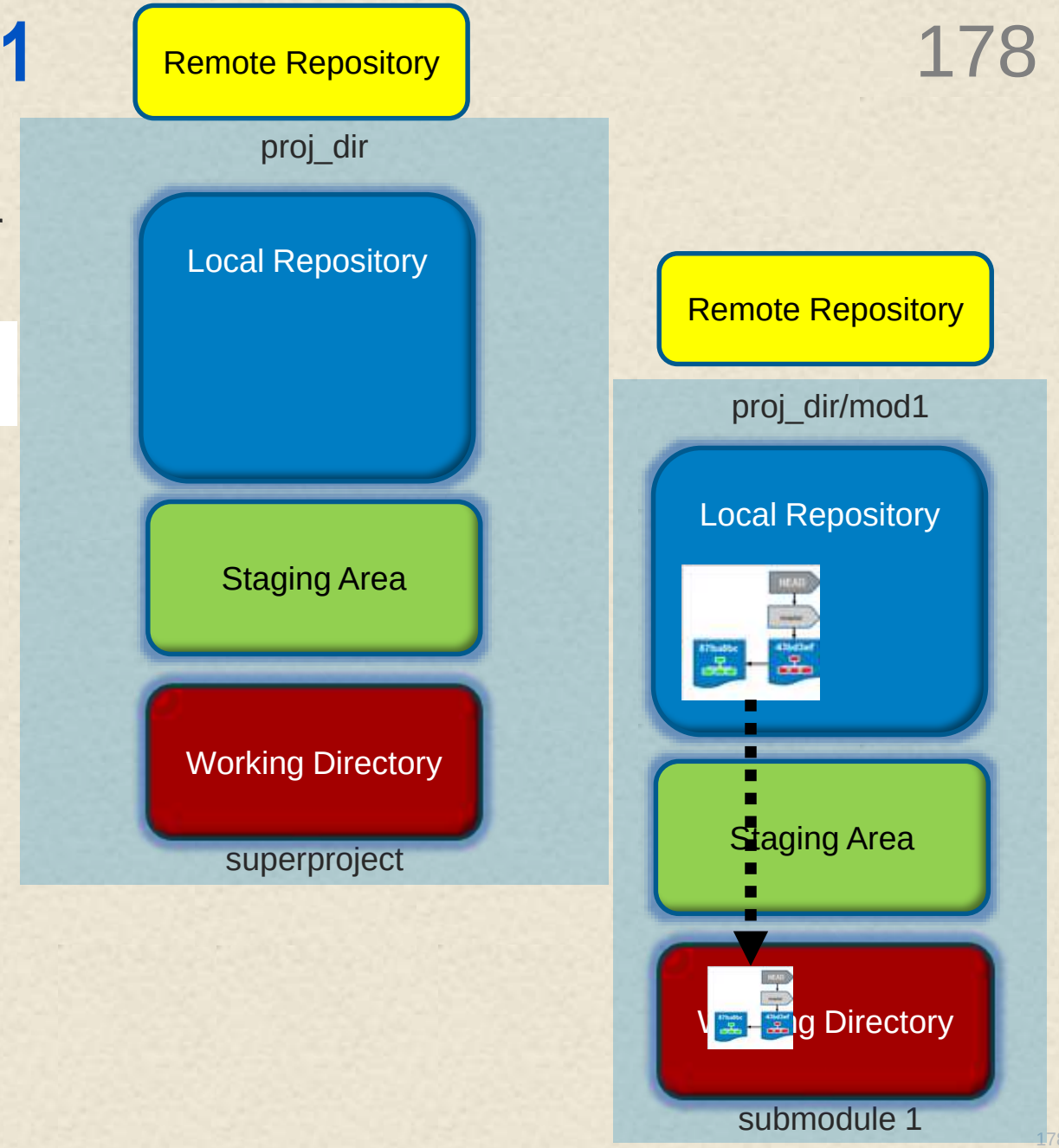
What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

3. Git adds the submodule's path for cloning to the .gitmodules file.



Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

3. Git adds the submodule's path for cloning to the .gitmodules file.

Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

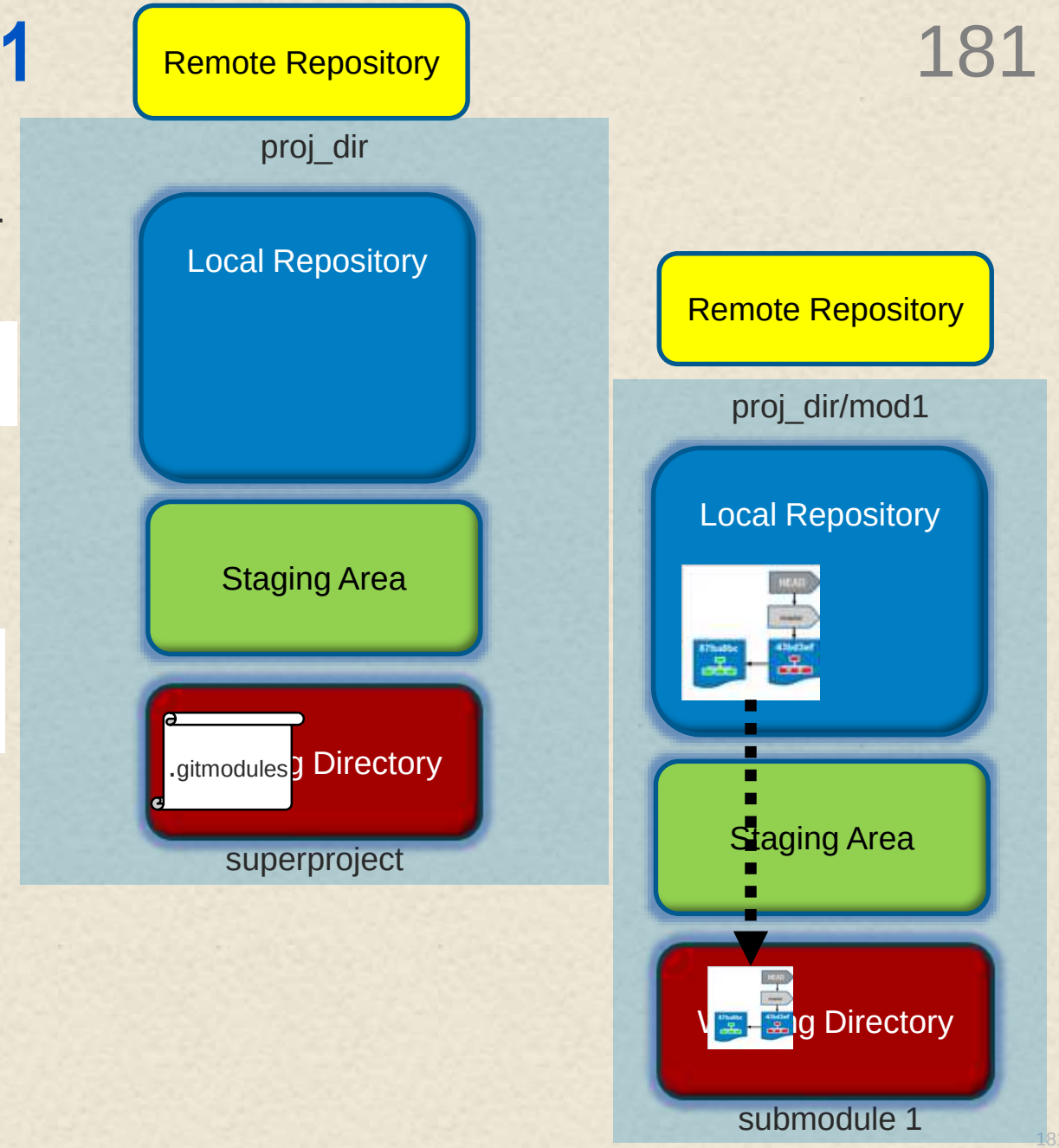
```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.



Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

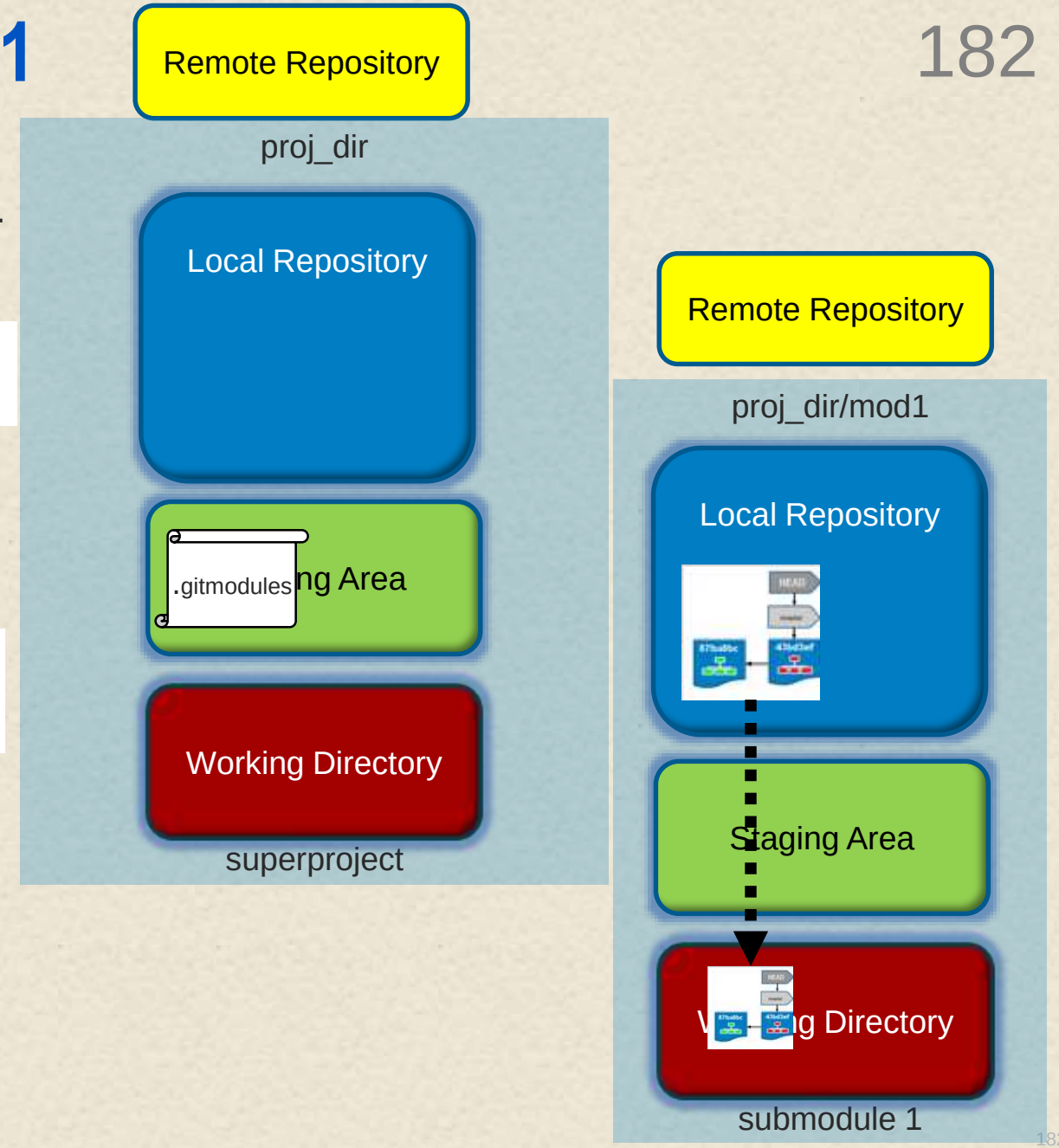
```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.



Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

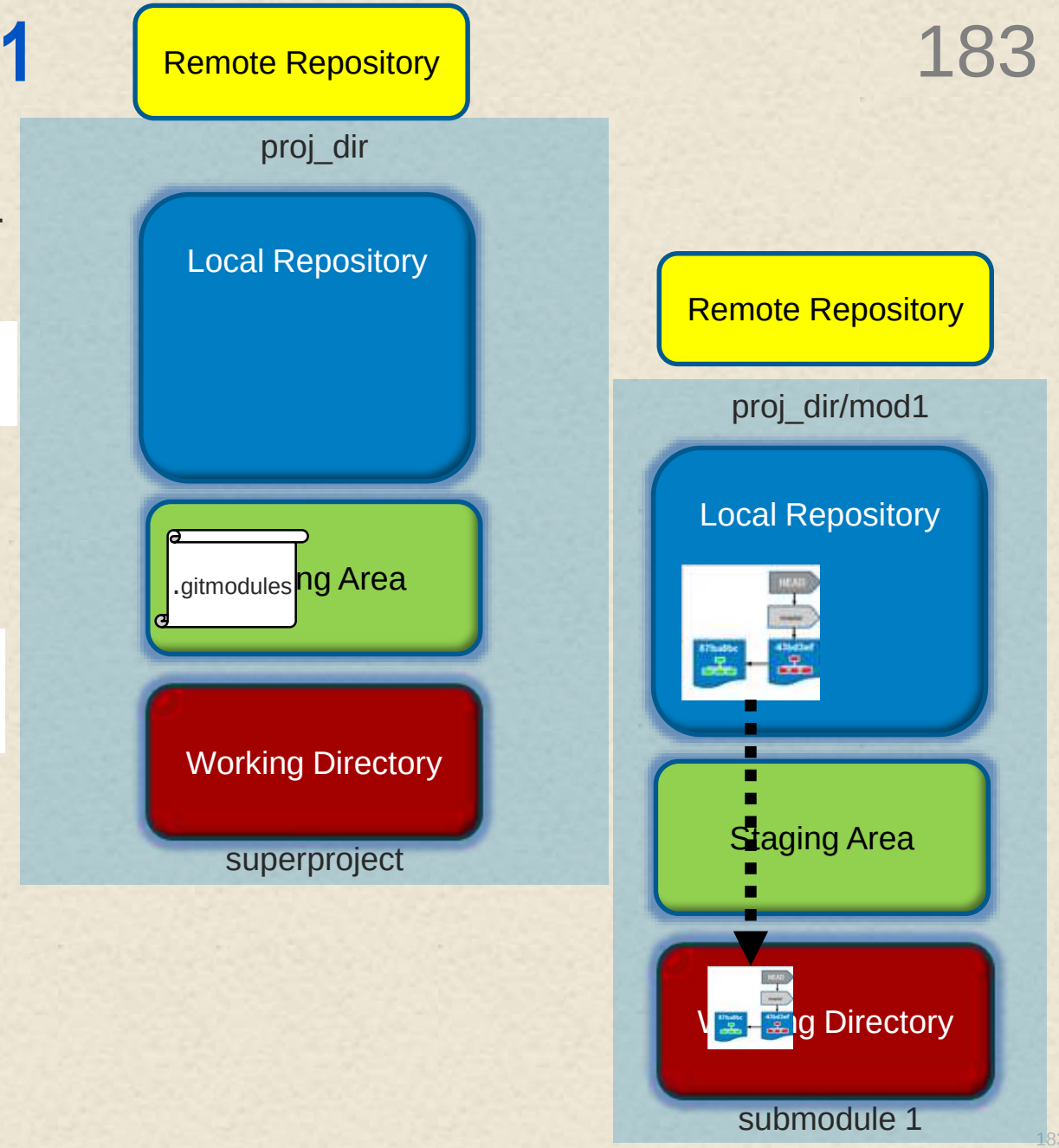
2. By default, Git checks out the master branch.

3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.

5. Git adds the current commit ID of the submodule to the index, ready to be committed.



Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

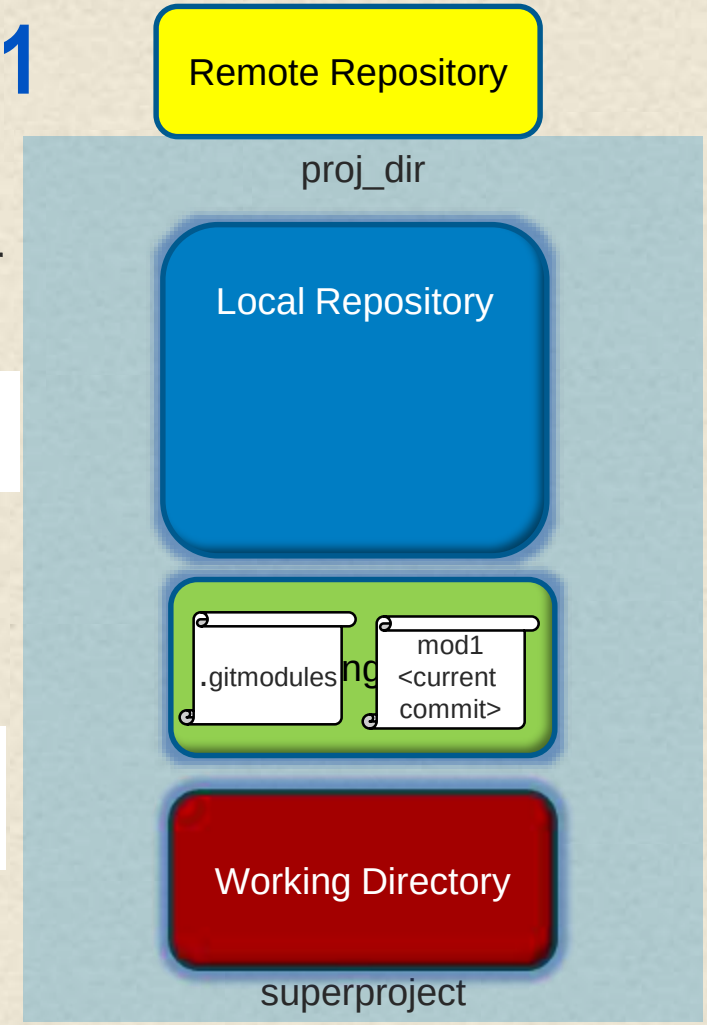
2. By default, Git checks out the master branch.

3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.

5. Git adds the current commit ID of the submodule to the index, ready to be committed.



Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

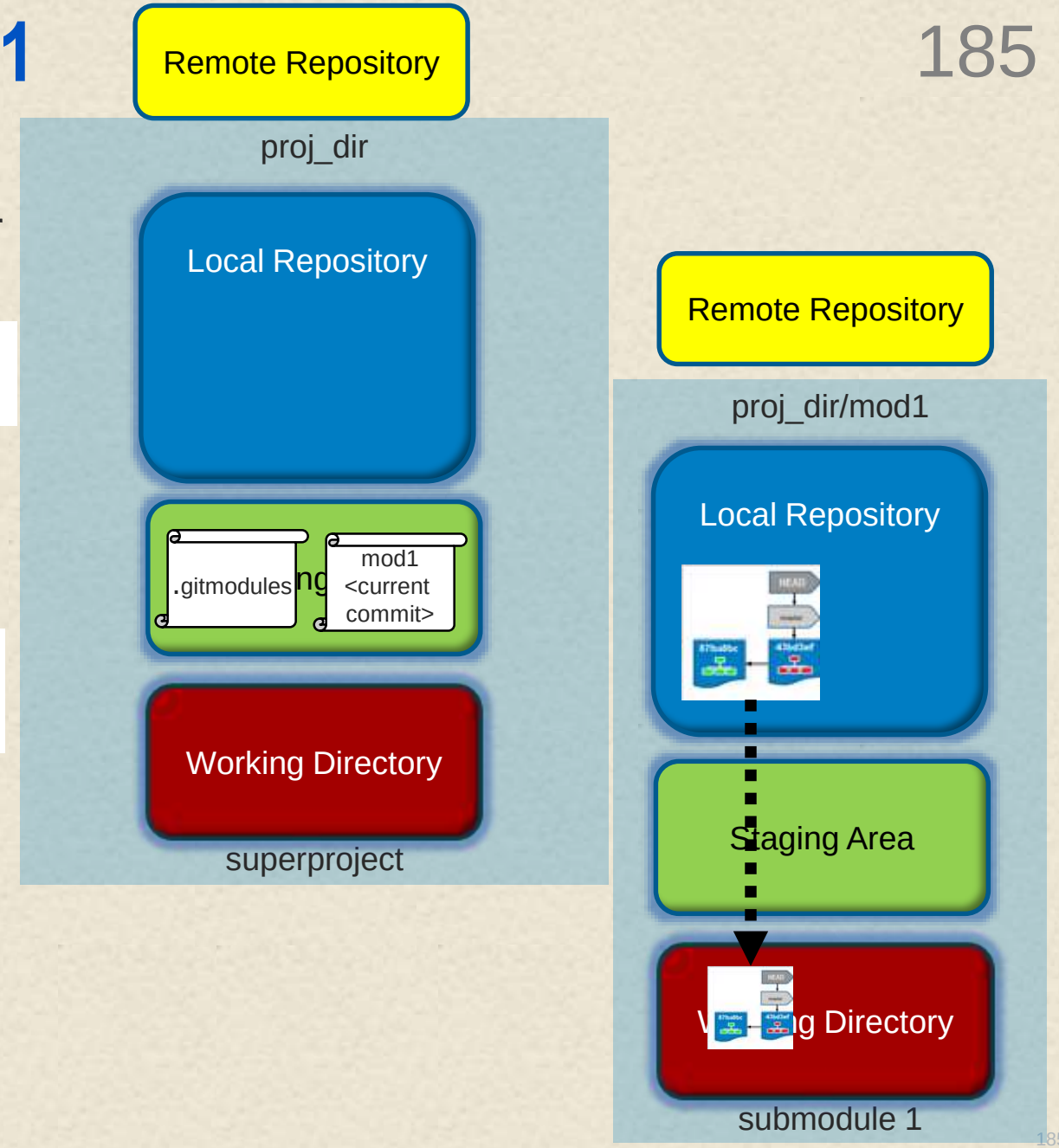
3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.

5. Git adds the current commit ID of the submodule to the index, ready to be committed.

```
$ git status
On branch master
  …
        new file:   .gitmodules
        new file:   mod1
```



Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

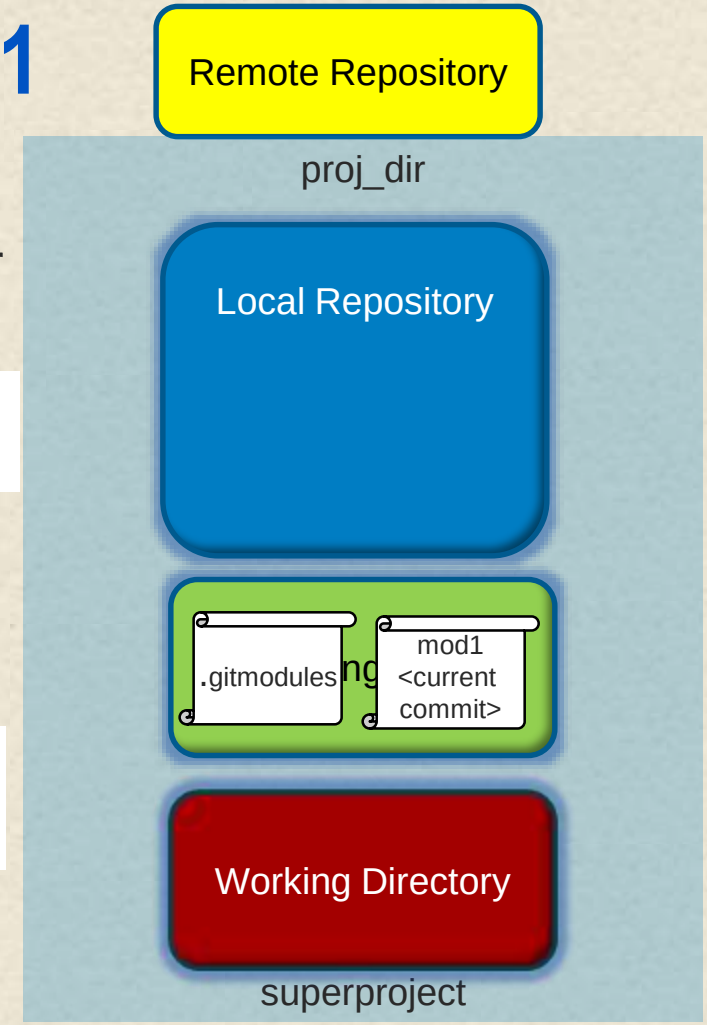
3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.

5. Git adds the current commit ID of the submodule to the index, ready to be committed.

```
$ git status
On branch master
  …
        new file:   .gitmodules
        new file:   mod1
```



6. To complete the add process, you need to complete the Git workflow for the staged changes. From the superproject's directory:

Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

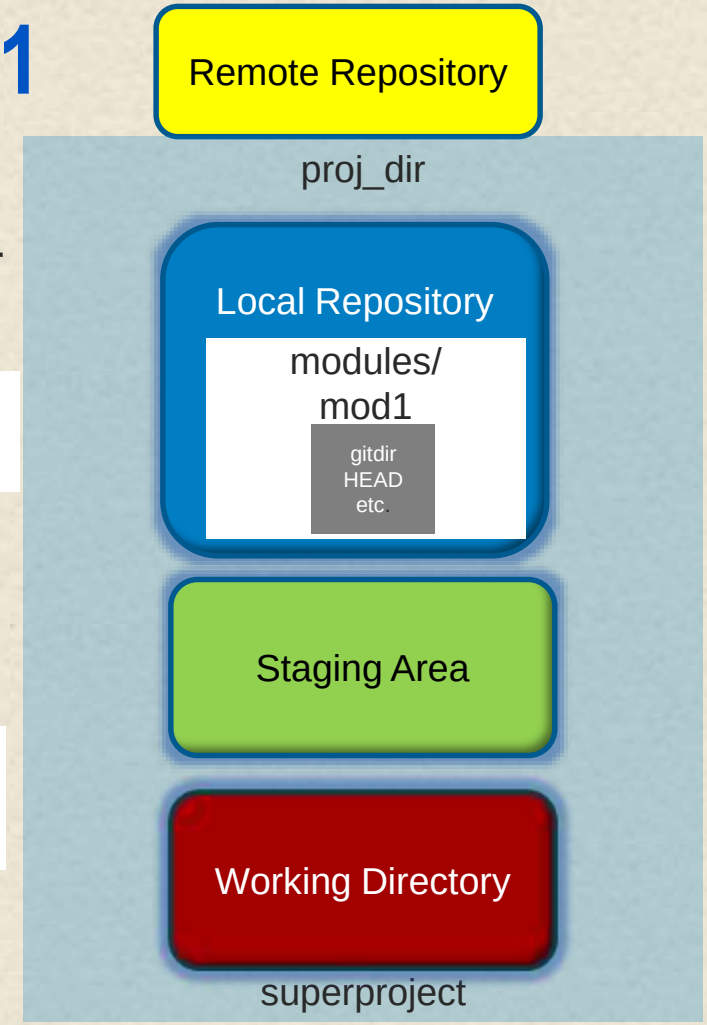
3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.

5. Git adds the current commit ID of the submodule to the index, ready to be committed.

```
$ git status
On branch master
  …
        new file:   .gitmodules
        new file:   mod1
```



6. To complete the add process, you need to complete the Git workflow for the staged changes. From the superproject's directory:

7. Commit

Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

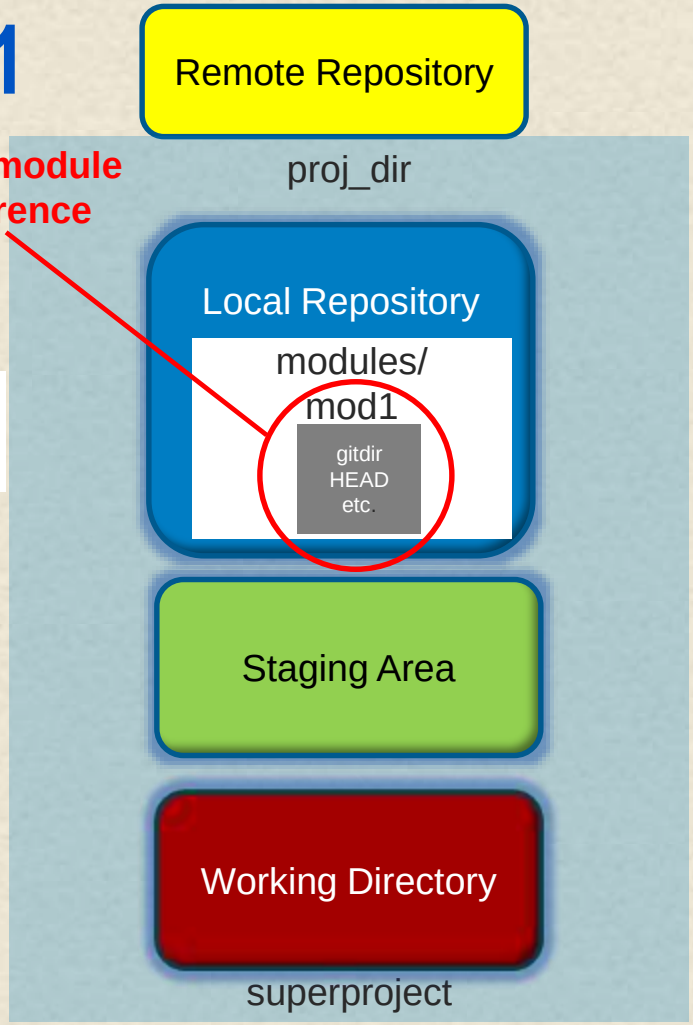
3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.

5. Git adds the current commit ID of the submodule to the index, ready to be committed.

```
$ git status
On branch master
  …
        new file:   .gitmodules
        new file:   mod1
```



6. To complete the add process, you need to complete the Git workflow for the staged changes. From the superproject's directory:

7. Commit

Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

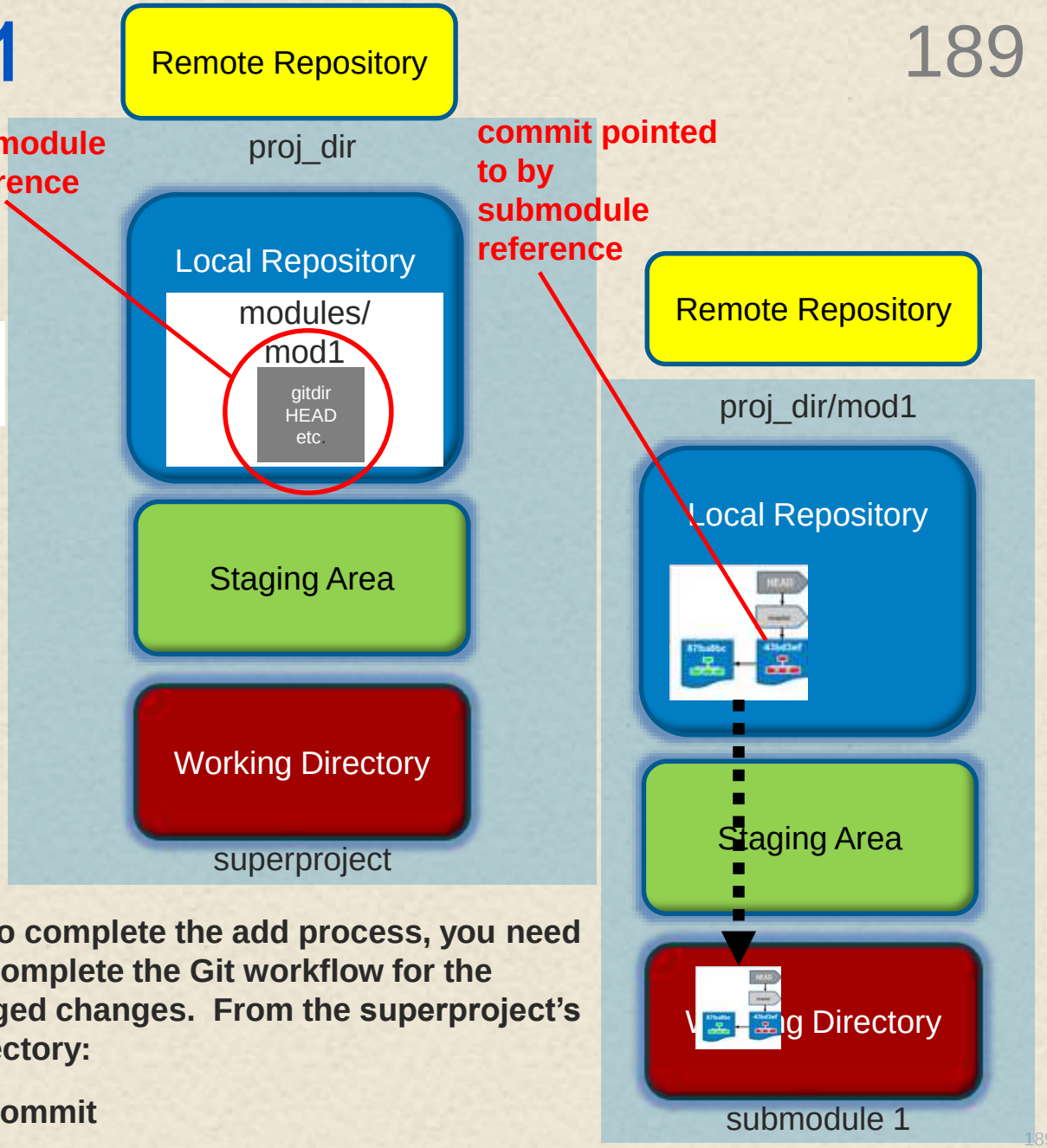
3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.

5. Git adds the current commit ID of the submodule to the index, ready to be committed.

```
$ git status
On branch master
  …
        new file:   .gitmodules
        new file:   mod1
```



6. To complete the add process, you need to complete the Git workflow for the staged changes. From the superproject's directory:

7. Commit

Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

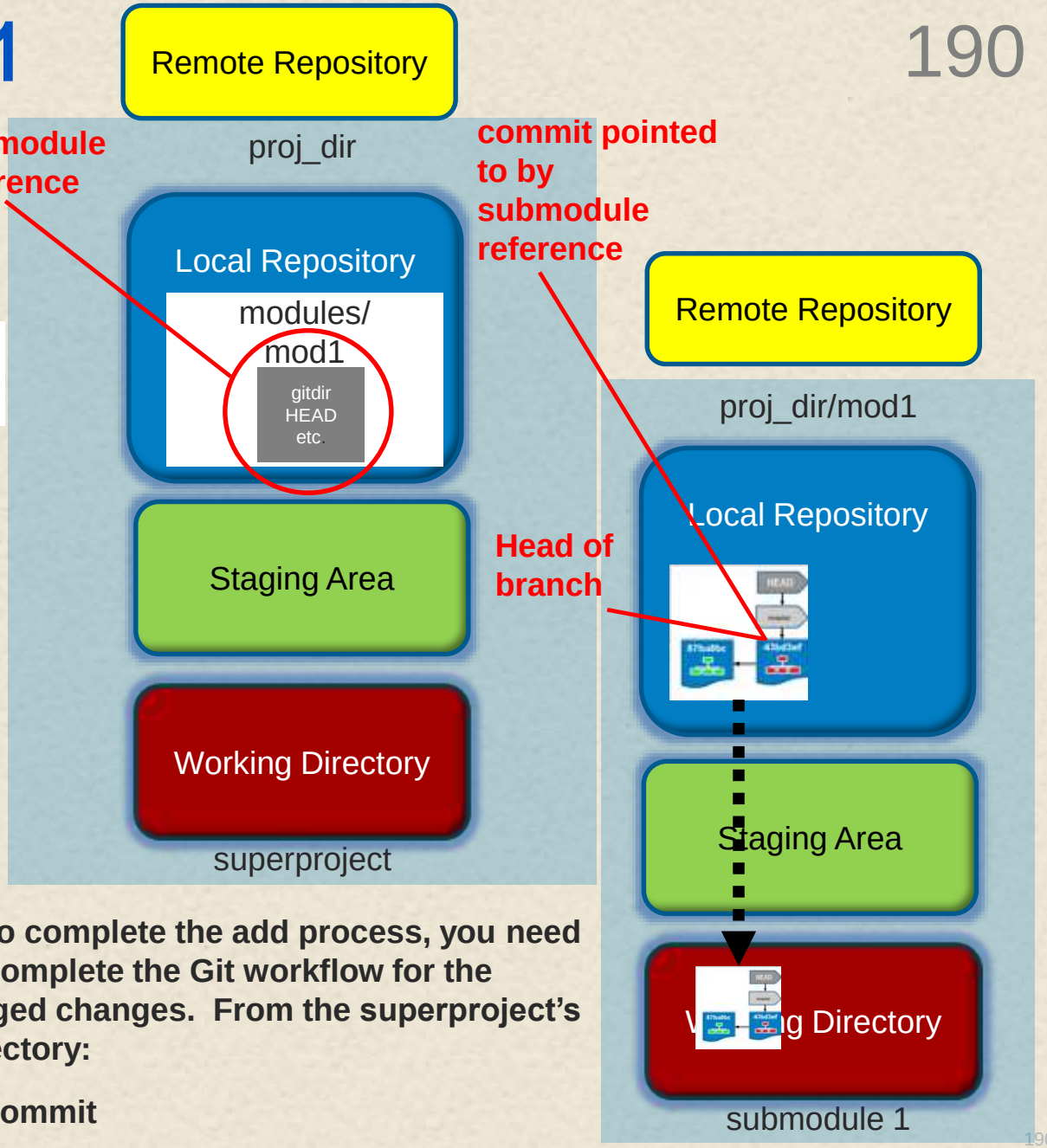
3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.

5. Git adds the current commit ID of the submodule to the index, ready to be committed.

```
$ git status
On branch master
  …
        new file:   .gitmodules
        new file:   mod1
```



6. To complete the add process, you need to complete the Git workflow for the staged changes. From the superproject's directory:

7. Commit

Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

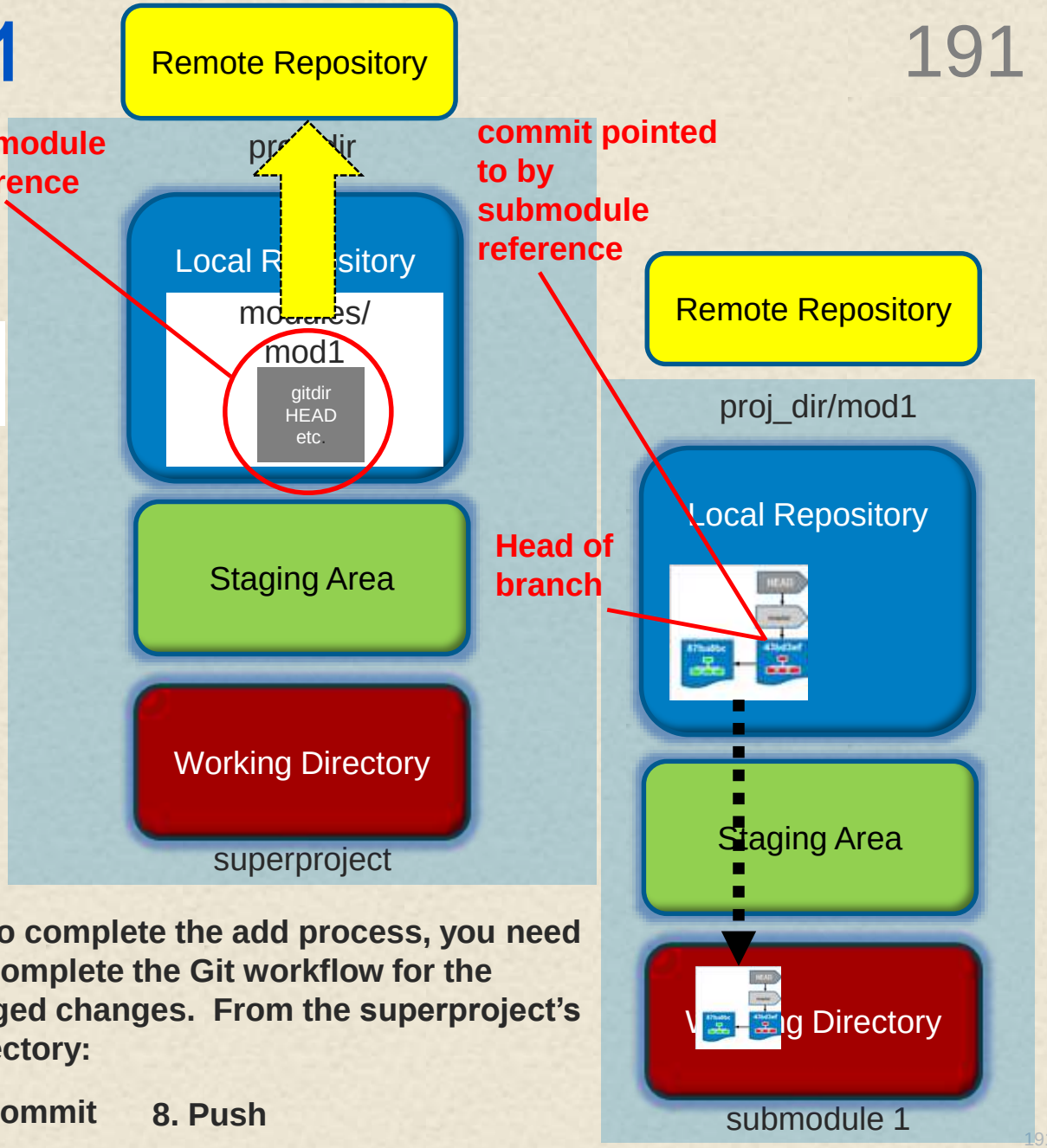
3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.

5. Git adds the current commit ID of the submodule to the index, ready to be committed.

```
$ git status
On branch master
  …
        new file:   .gitmodules
        new file:   mod1
```



6. To complete the add process, you need to complete the Git workflow for the staged changes. From the superproject's directory:

7. Commit **8. Push**

Submodules 1

What happens when you add a submodule?

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.
```

2. By default, Git checks out the master branch.

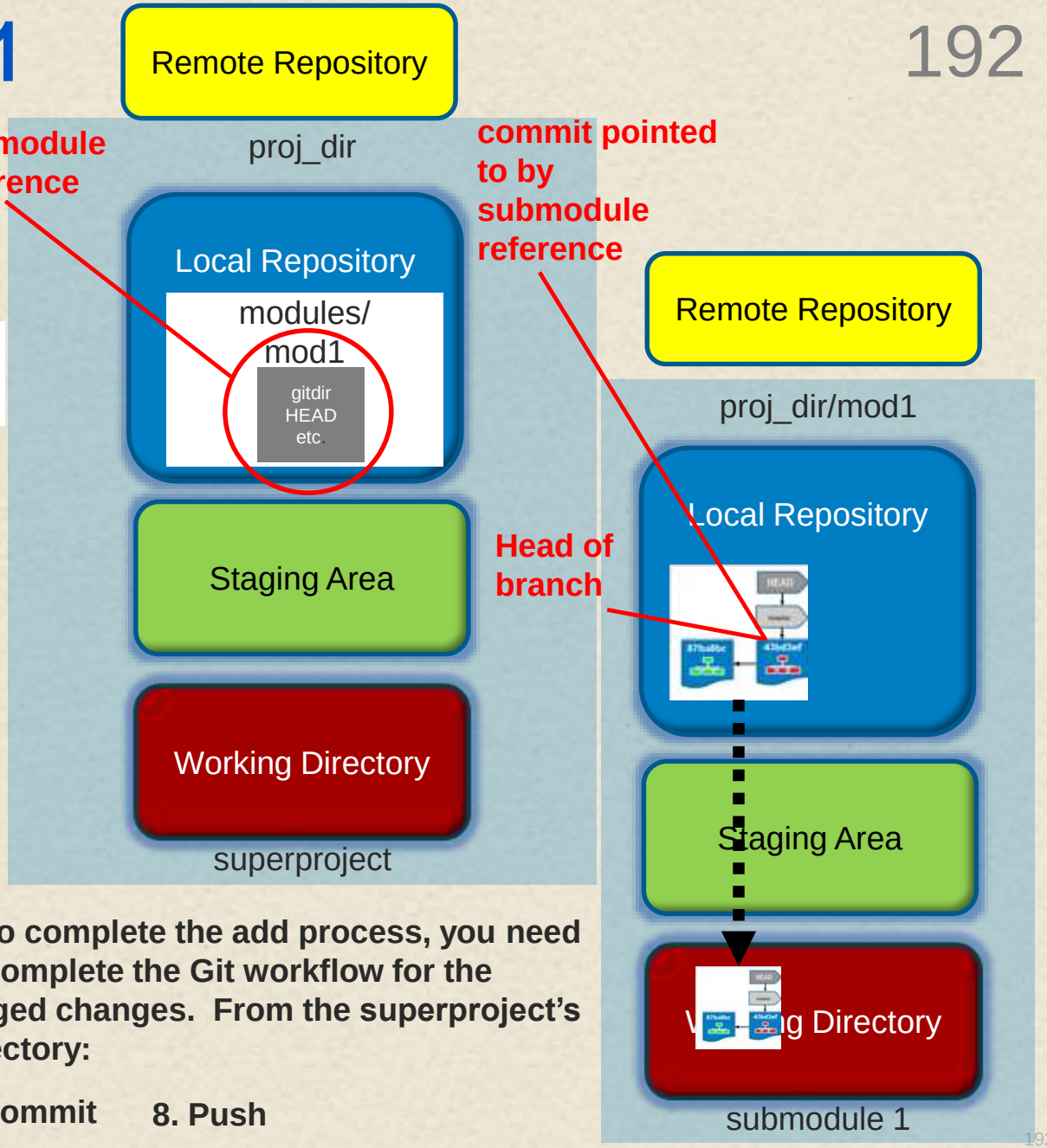
3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
```

4. Git adds the .gitmodules file to the index, ready to be committed.

5. Git adds the current commit ID of the submodule to the index, ready to be committed.

```
$ git status
On branch master
  …
        new file:   .gitmodules
        new file:   mod1
```



6. To complete the add process, you need to complete the Git workflow for the staged changes. From the superproject's directory:

7. Commit

8. Push

Submodules 2

Remote Repository

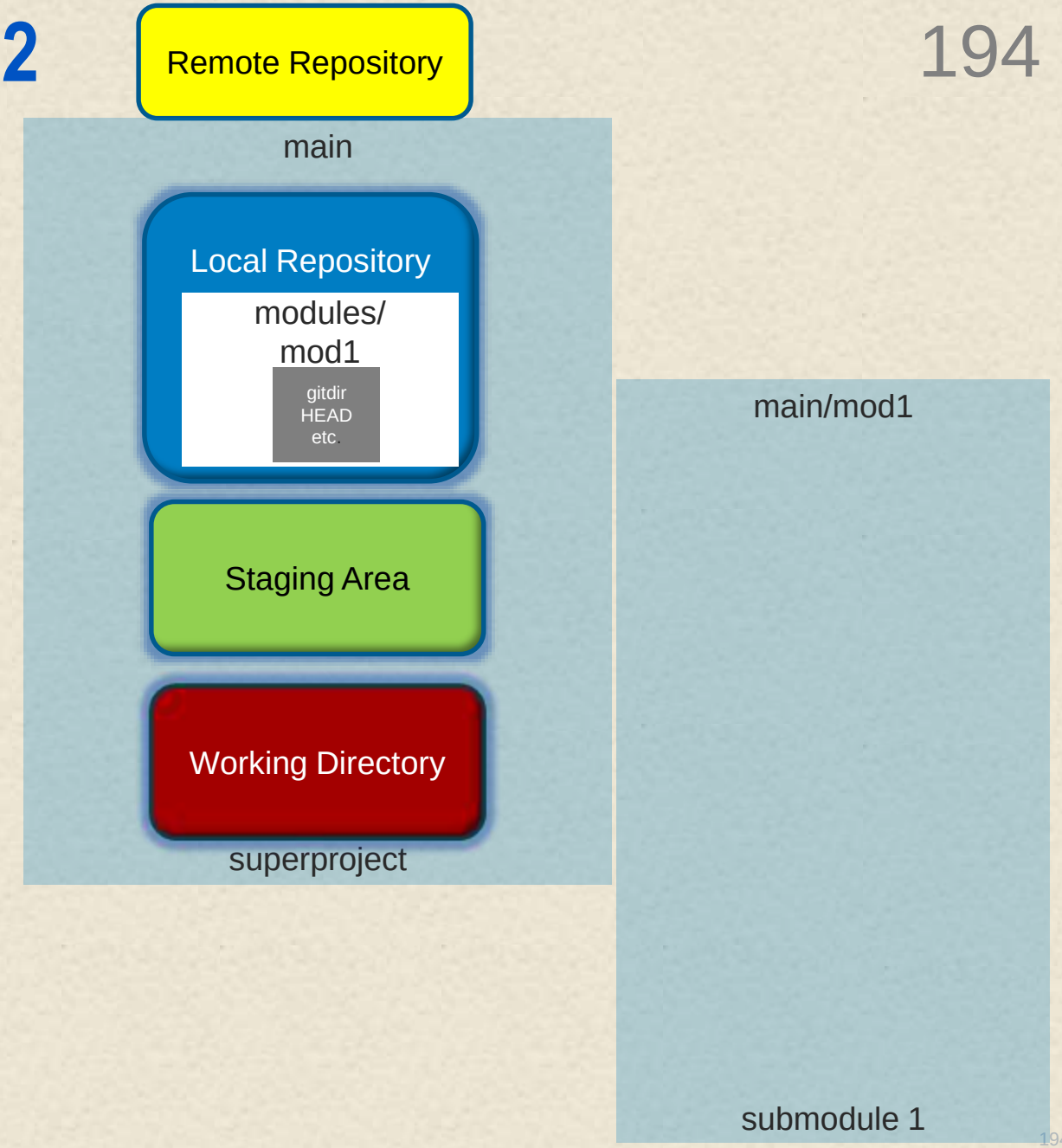
How do we clone a repository with submodules?

Submodules 2

How do we clone a repository with submodules?

1. git clone - puts in structure - but submodules areas are empty

```
$ git clone <remote path>/main.git
Cloning into 'main'...
done.

$ cd main

$ ls -a
./  ../  .git/  .gitmodules  file1.txt  mod1/
```



Submodules 2

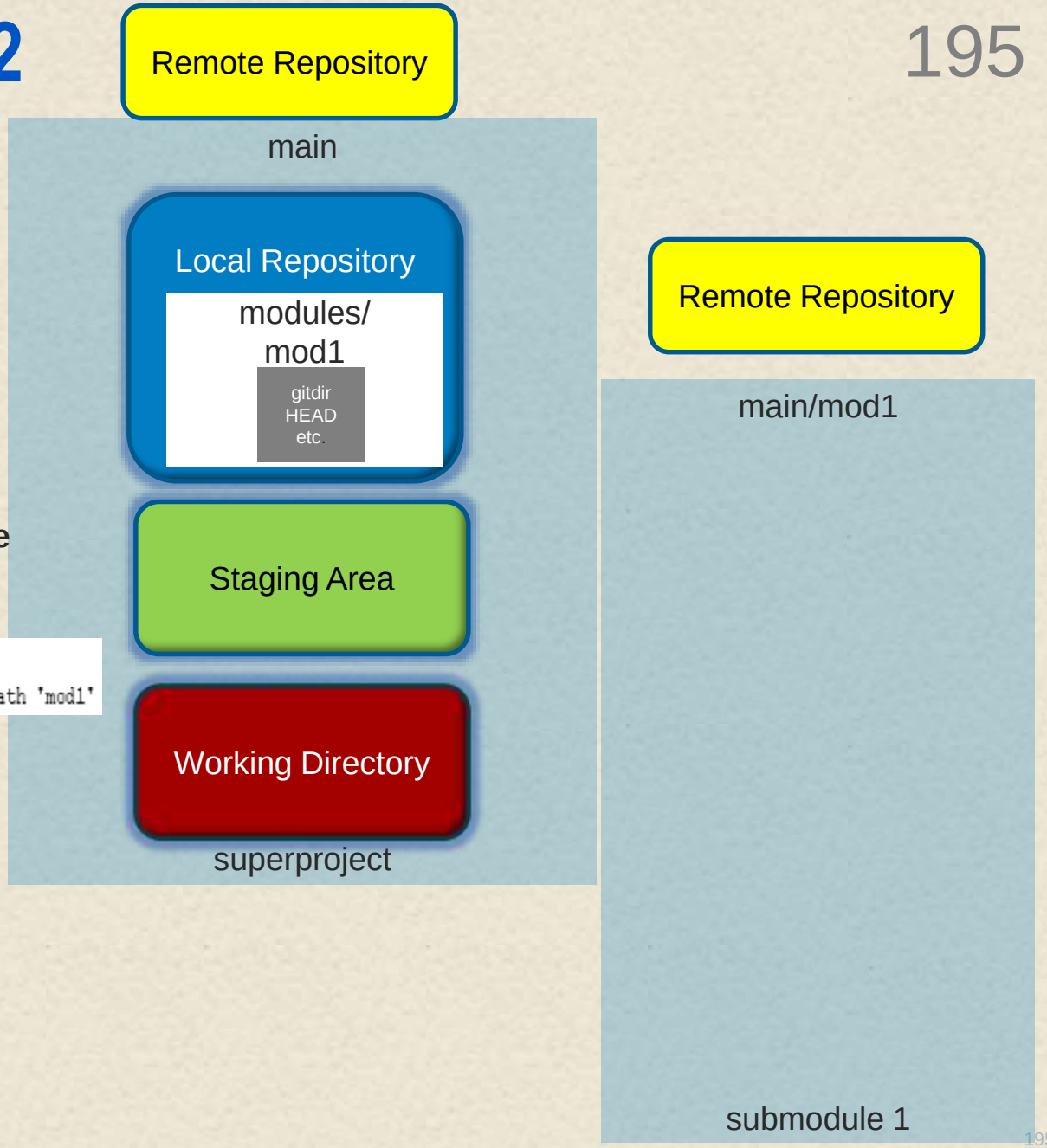
How do we clone a repository with submodules?

1. git clone - puts in structure - but submodules areas are empty

```
$ git clone <remote path>/main.git
Cloning into 'main'...
done.

$ cd main

$ ls -a
./  ../  .git/  .gitmodules  file1.txt  mod1/
```

2. git submodule init - puts submodule location information in superproject's config file

```
$ git submodule init
Submodule 'mod1' (<remote path>/mod1.git) registered for path 'mod1'
```



Submodules 2

How do we clone a repository with submodules?

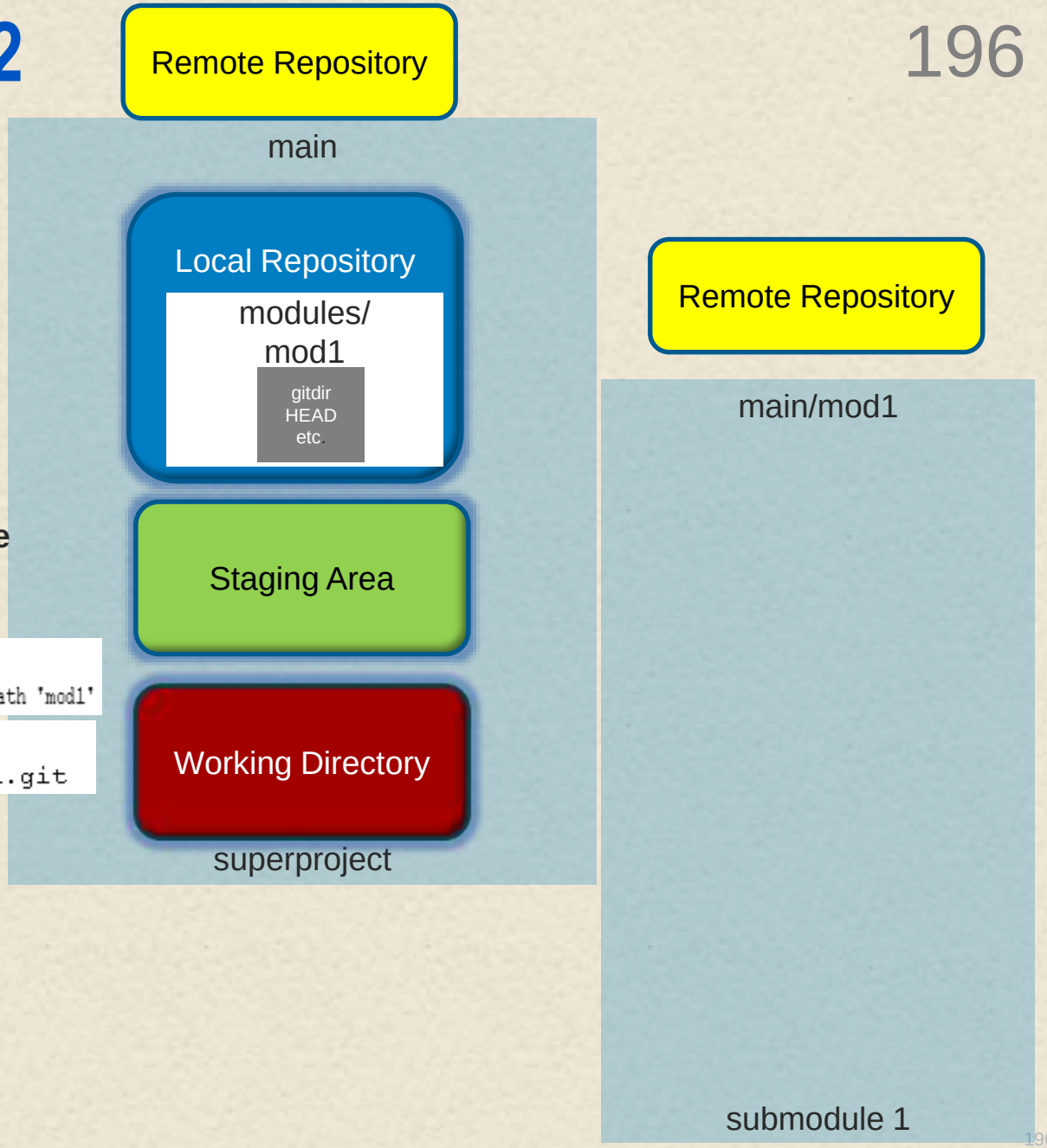
1. git clone - puts in structure - but submodules areas are empty

```
$ git clone <remote path>/main.git
Cloning into 'main'...
done.

$ cd main

$ ls -a
./  ../  .git/  .gitmodules  file1.txt  mod1/
```

2. git submodule init - puts submodule location information in superproject's config file

```
$ git submodule init
Submodule 'mod1' (<remote path>/mod1.git) registered for path 'mod1'
```

```
$ git config -l | grep submodule
submodule.mod1.url=<remote path>/mod1.git
```



Submodules 2

How do we clone a repository with submodules?

1. git clone - puts in structure - but submodules areas are empty

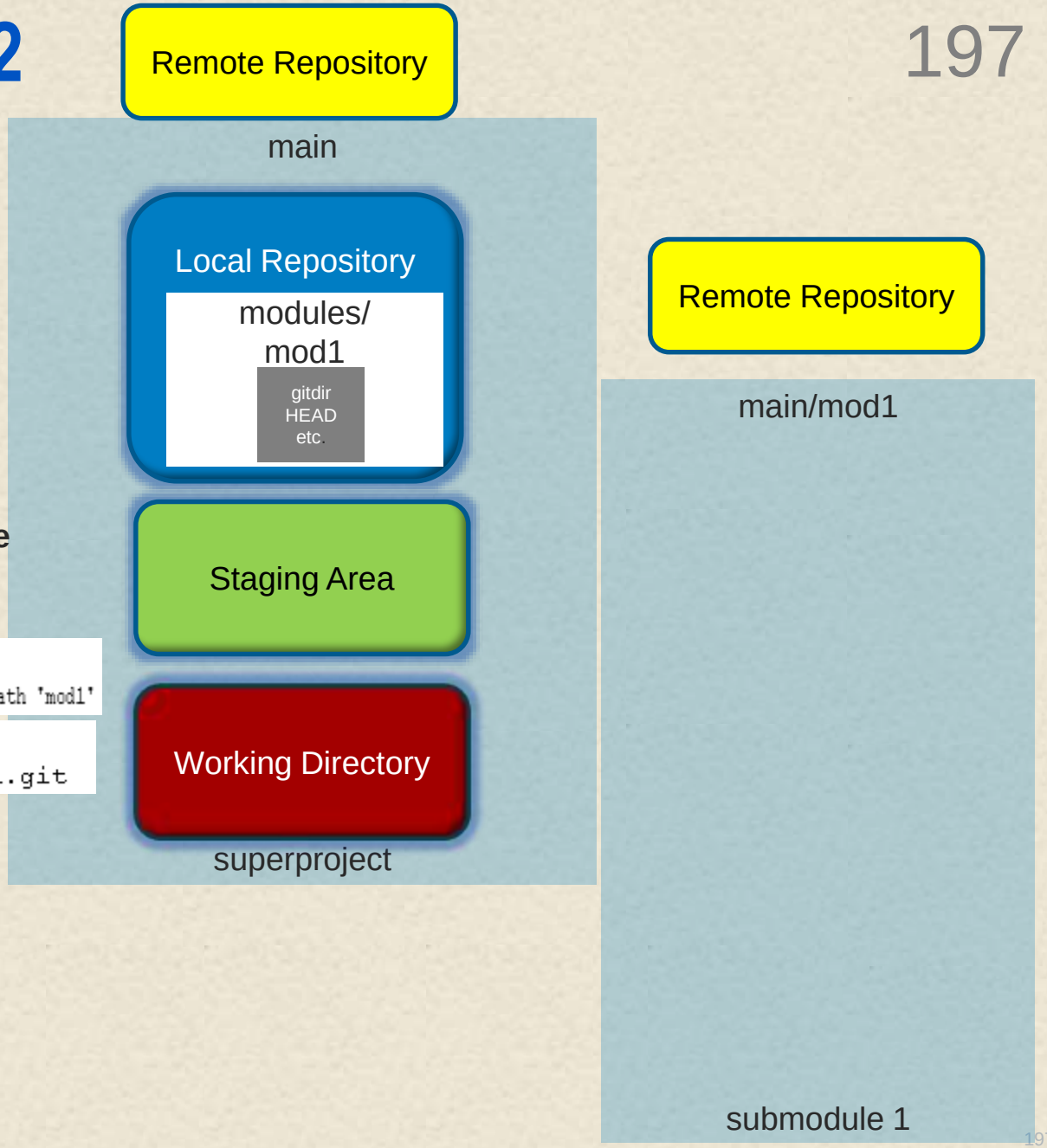
```
$ git clone <remote path>/main.git
Cloning into 'main'...
done.

$ cd main

$ ls -a
./  ../  .git/  .gitmodules  file1.txt  mod1/
```

2. git submodule init - puts submodule location information in superproject's config file

```
$ git submodule init
Submodule 'mod1' (<remote path>/mod1.git) registered for path 'mod1'
```

```
$ git config -l | grep submodule
submodule.mod1.url=<remote path>/mod1.git
```

3. git submodule update - actually clones repositories for submodules into the corresponding subdirectories and checks out the indicated commits for the containing project



Submodules 2

How do we clone a repository with submodules?

1. git clone - puts in structure - but submodules areas are empty

```
$ git clone <remote path>/main.git
Cloning into 'main'...
done.

$ cd main

$ ls -a
./  ../  .git/  .gitmodules  file1.txt  mod1/
```

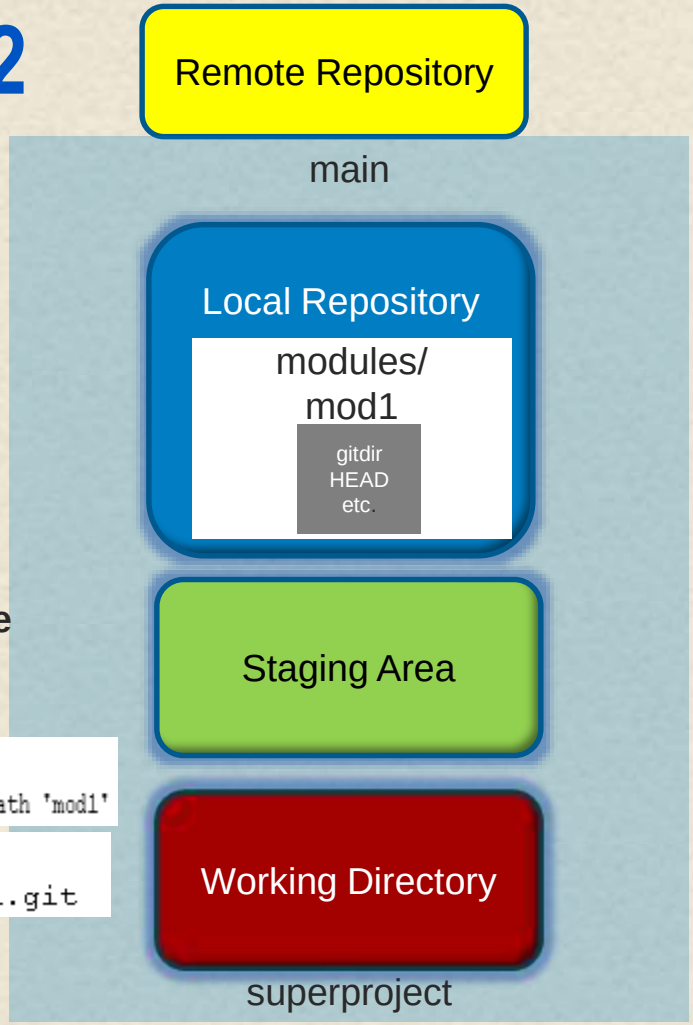
2. git submodule init - puts submodule location information in superproject's config file

```
$ git submodule init
Submodule 'mod1' (<remote path>/mod1.git) registered for path 'mod1'
```

```
$ git config -l | grep submodule
submodule.mod1.url=<remote path>/mod1.git
```

3. git submodule update - actually clones repositories for submodules into the corresponding subdirectories and checks out the indicated commits for the containing project

```
$ git submodule update
Cloning into 'mod1'...
done.
Submodule path 'mod1': checked out '8add7dab652c856b65770bca867db2bbb39c0d00'
```



Submodules 2

How do we clone a repository with submodules?

1. git clone - puts in structure - but submodules areas are empty

```
$ git clone <remote path>/main.git
Cloning into 'main'...
done.

$ cd main

$ ls -a
./  ../  .git/  .gitmodules  file1.txt  mod1/
```

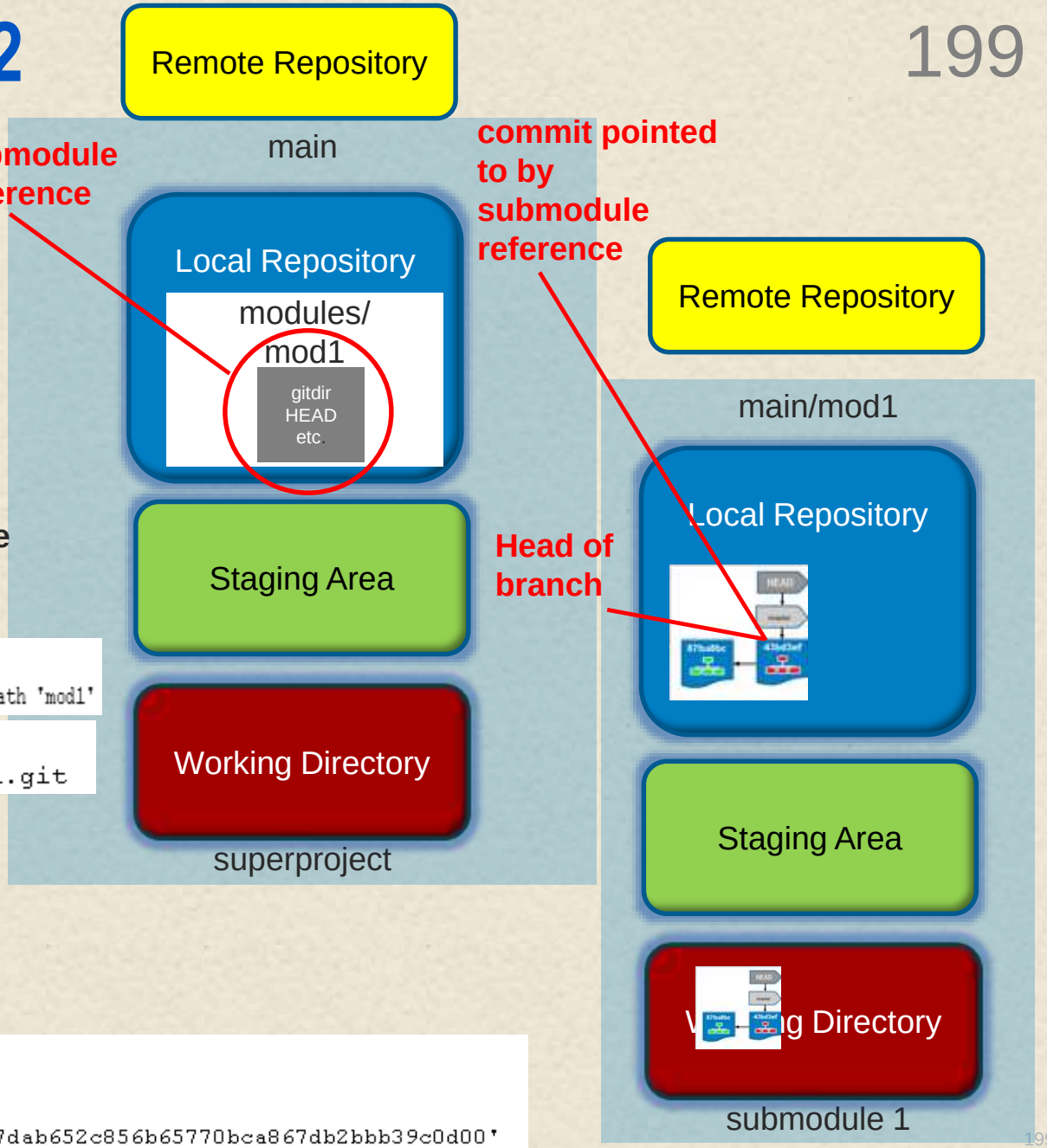
2. git submodule init - puts submodule location information in superproject's config file

```
$ git submodule init
Submodule 'mod1' (<remote path>/mod1.git) registered for path 'mod1'
```

```
$ git config -l | grep submodule
submodule.mod1.url=<remote path>/mod1.git
```

3. git submodule update - actually clones repositories for submodules into the corresponding subdirectories and checks out the indicated commits for the containing project

```
$ git submodule update
Cloning into 'mod1'...
done.
Submodule path 'mod1': checked out '8add7dab652c856b65770bca867db2bbb39c0d00'
```



Submodules 2

How do we clone a repository with submodules?

1. git clone - puts in structure - but submodules areas are empty

```
$ git clone <remote path>/main.git
Cloning into 'main'...
done.

$ cd main

$ ls -a
./  ../  .git/  .gitmodules  file1.txt  mod1/
```

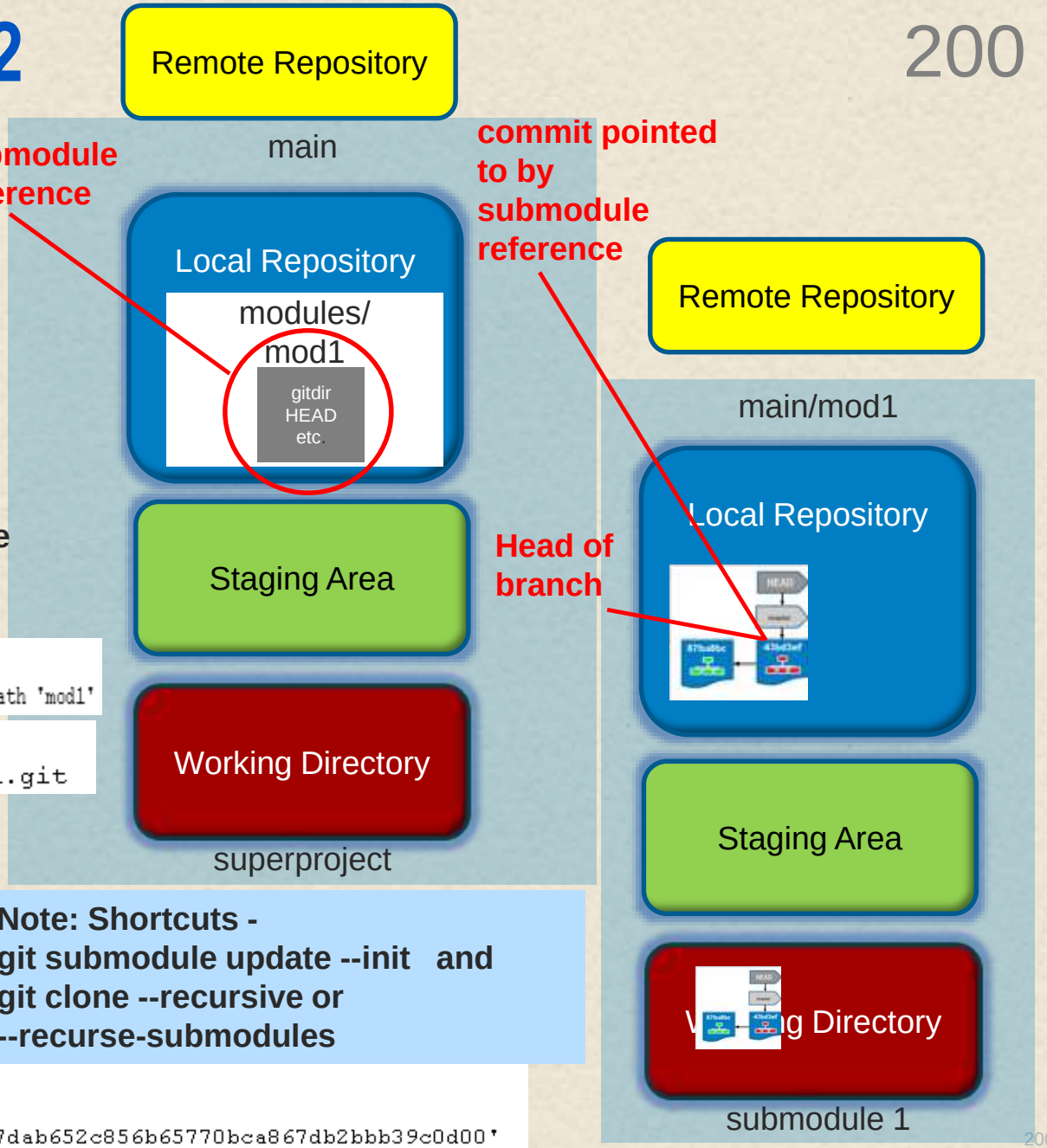
2. git submodule init - puts submodule location information in superproject's config file

```
$ git submodule init
Submodule 'mod1' (<remote path>/mod1.git) registered for path 'mod1'
```

```
$ git config -l | grep submodule
submodule.mod1.url=<remote path>/mod1.git
```

3. git submodule update - actually clones repositories for submodules into the corresponding subdirectories and checks out the indicated commits for the containing project

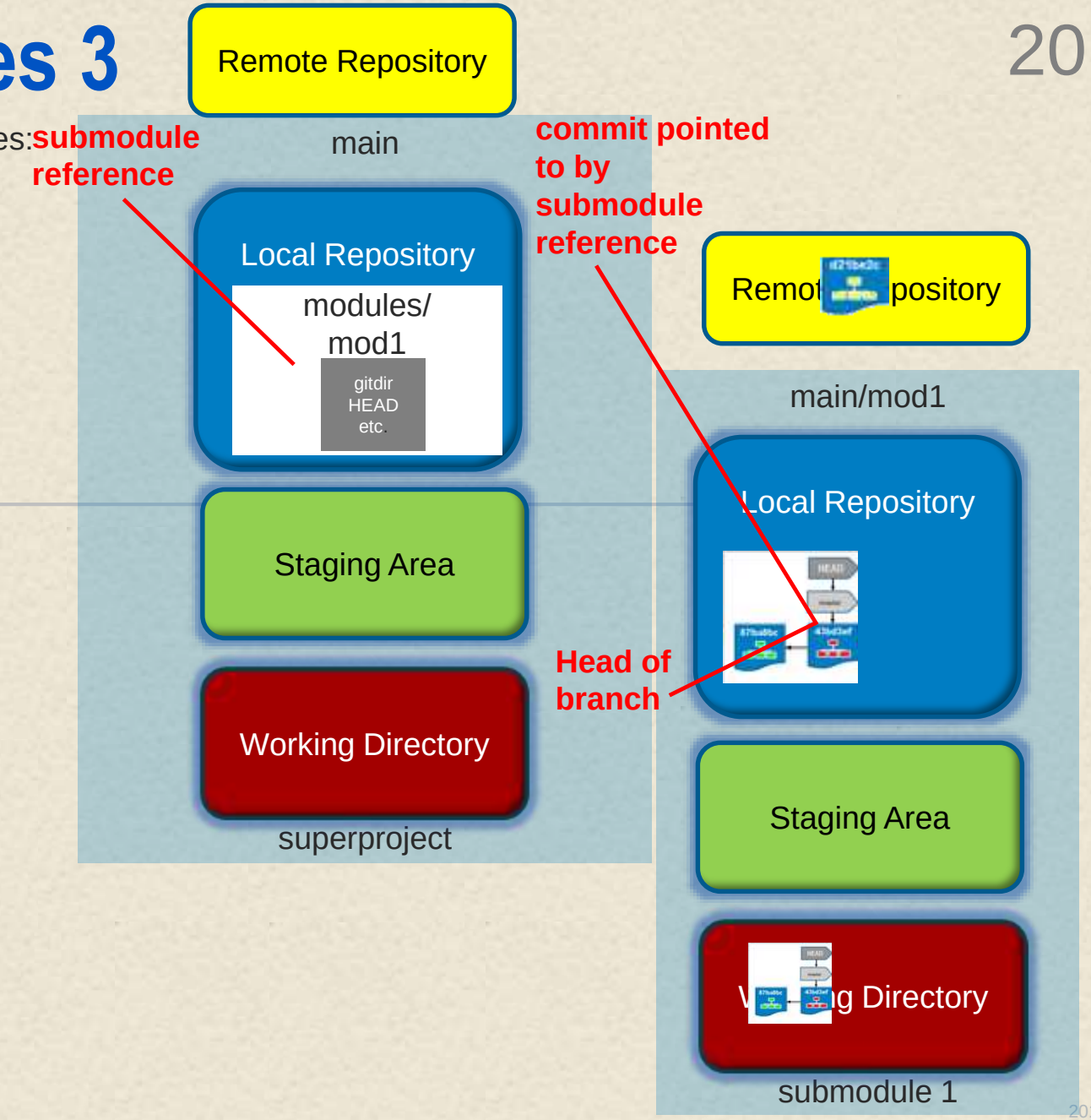
```
$ git submodule update
Cloning into 'mod1'...
done.
Submodule path 'mod1': checked out '8add7dab652c856b65770bca867db2bbb39c0d00'
```



Note: Shortcuts -
git submodule update --init and
git clone --recursive or
--recurse-submodules

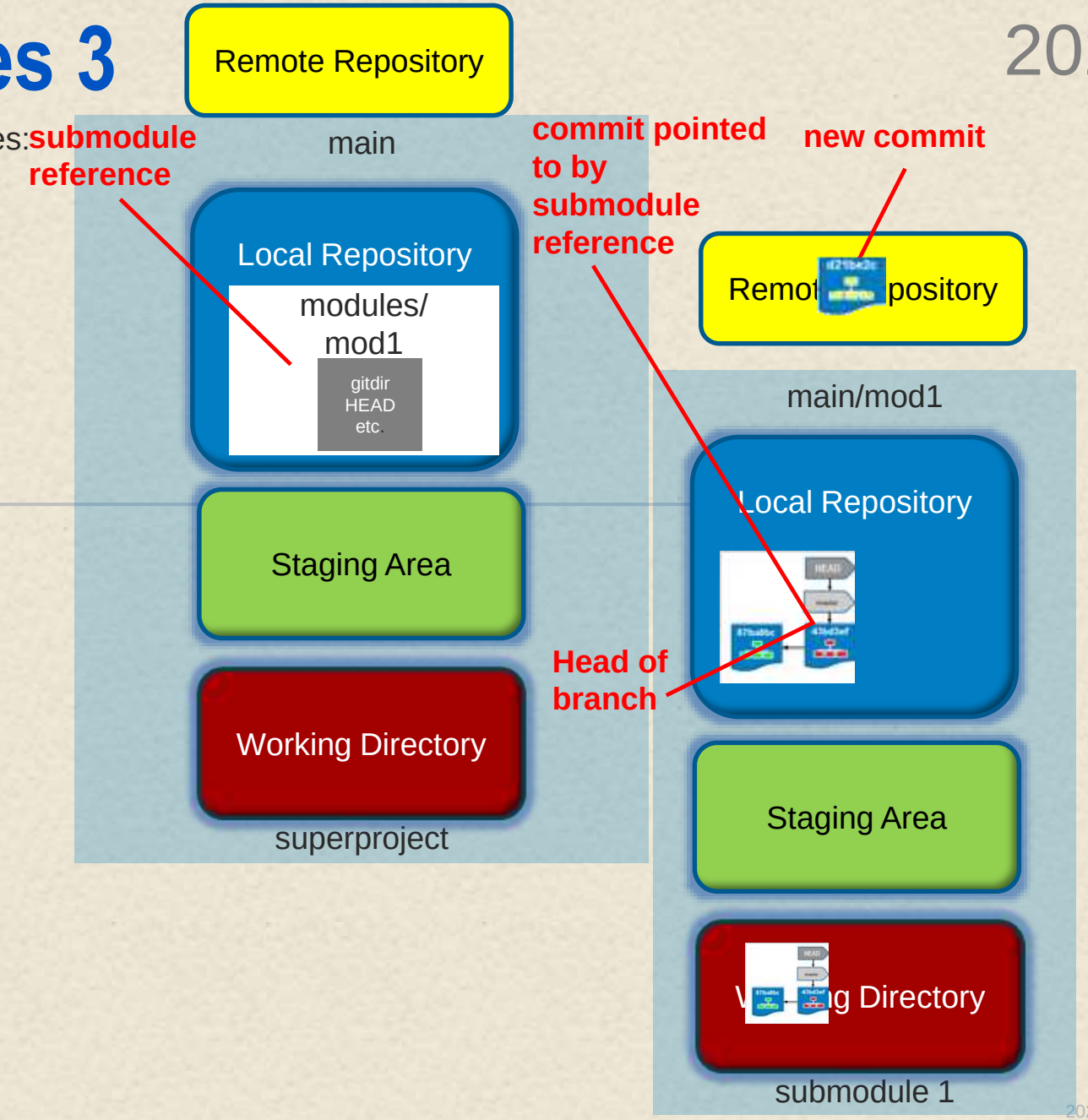
Submodules 3

Incorporating updates to submodules:



Submodules 3

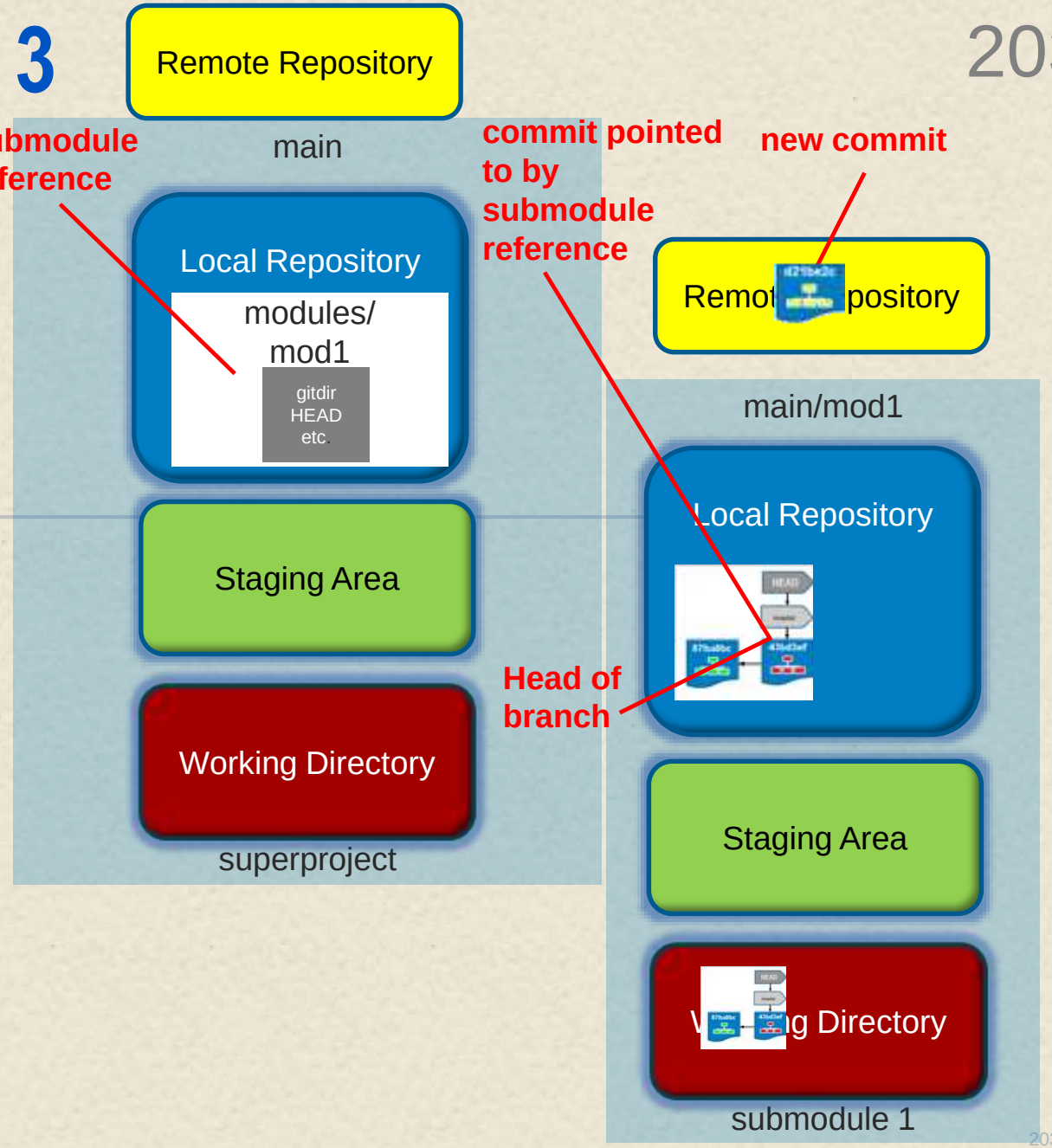
Incorporating updates to submodules:



Submodules 3

Incorporating updates to submodules:

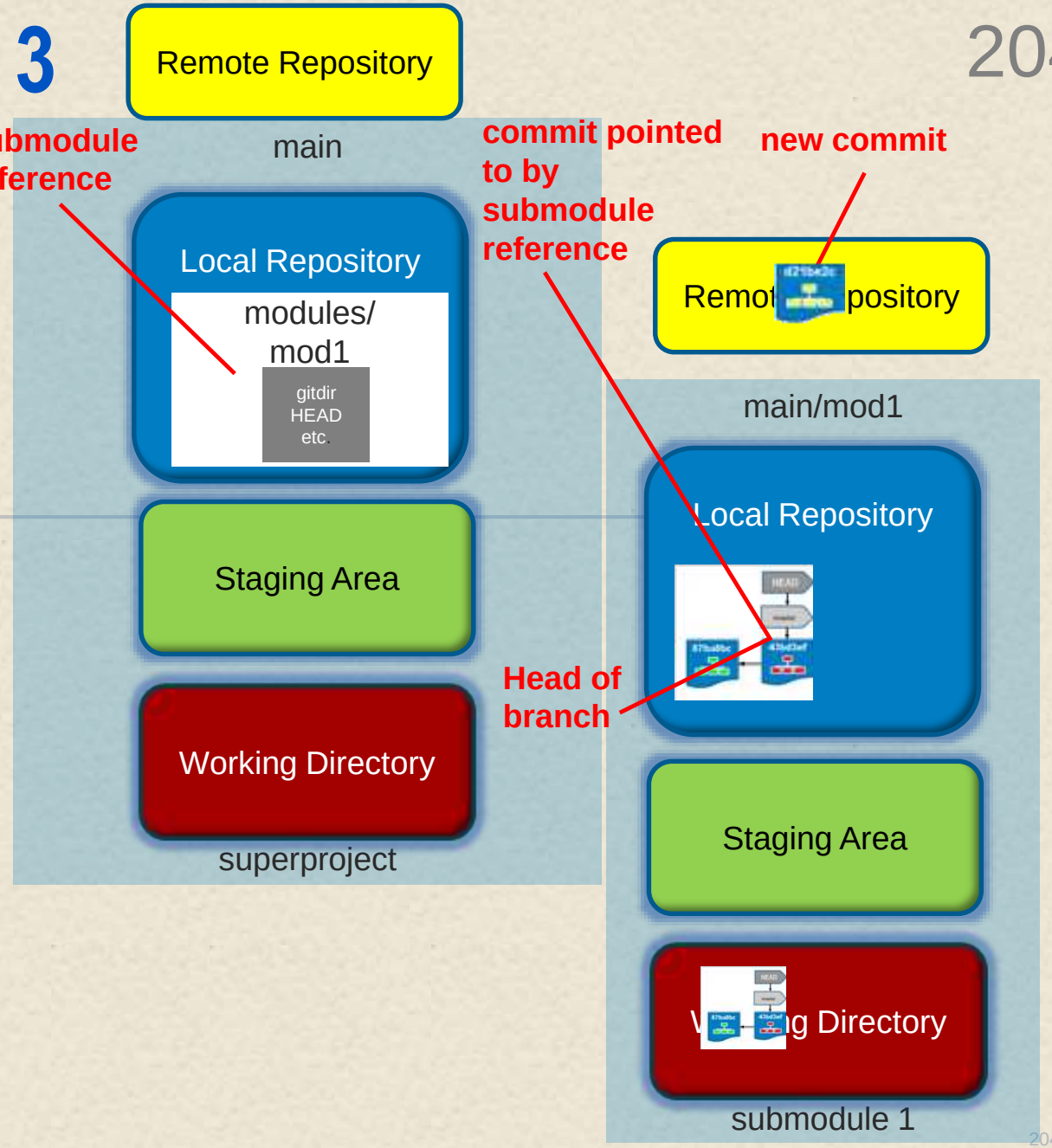
1. You can



Submodules 3

Incorporating updates to submodules:

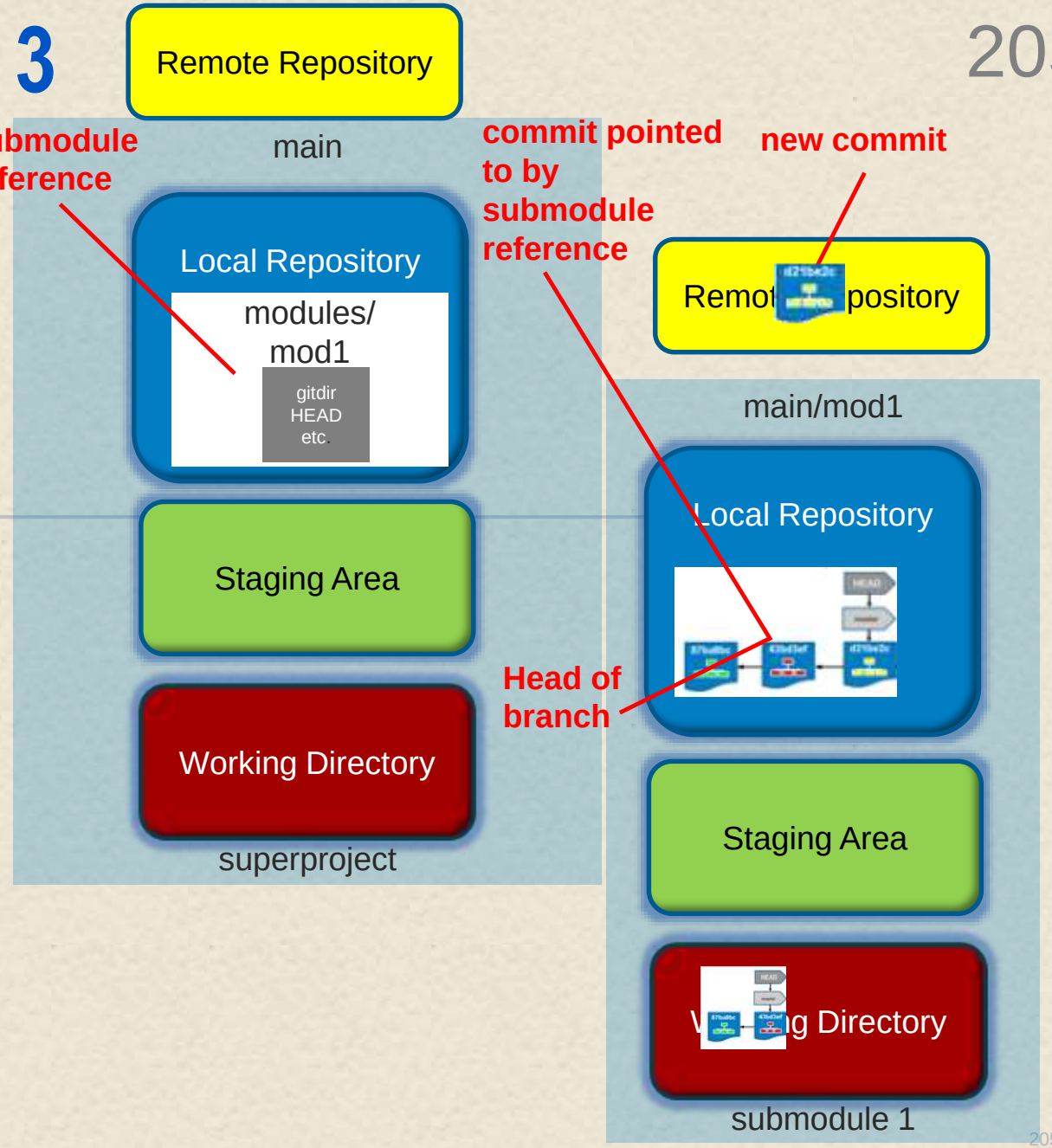
1. You can
   $ cd mod1; git checkout <branch> ; git pull



Submodules 3

Incorporating updates to submodules:

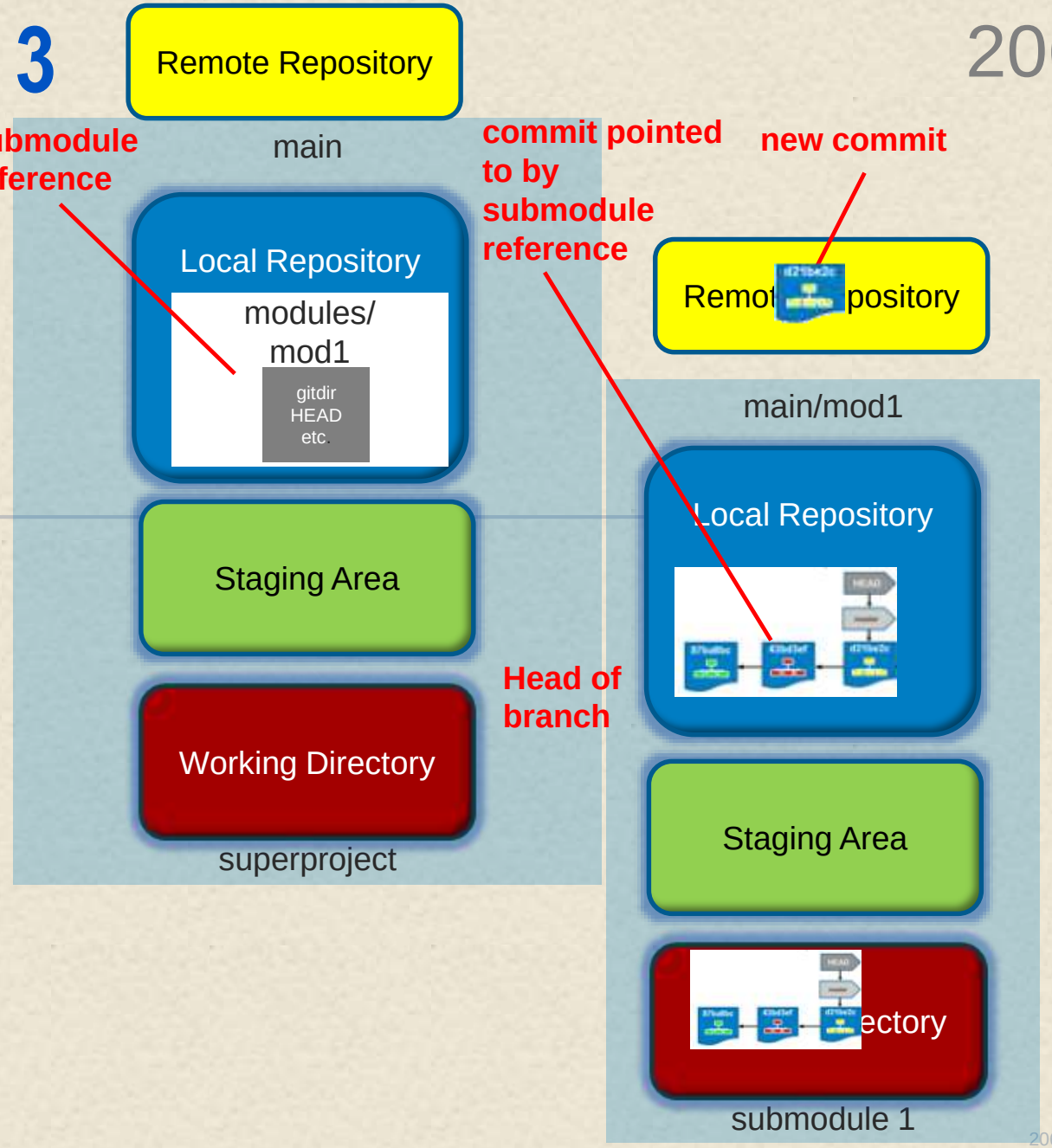
1. You can
   $ cd mod1; git checkout <branch> ; git pull



Submodules 3

Incorporating updates to submodules:

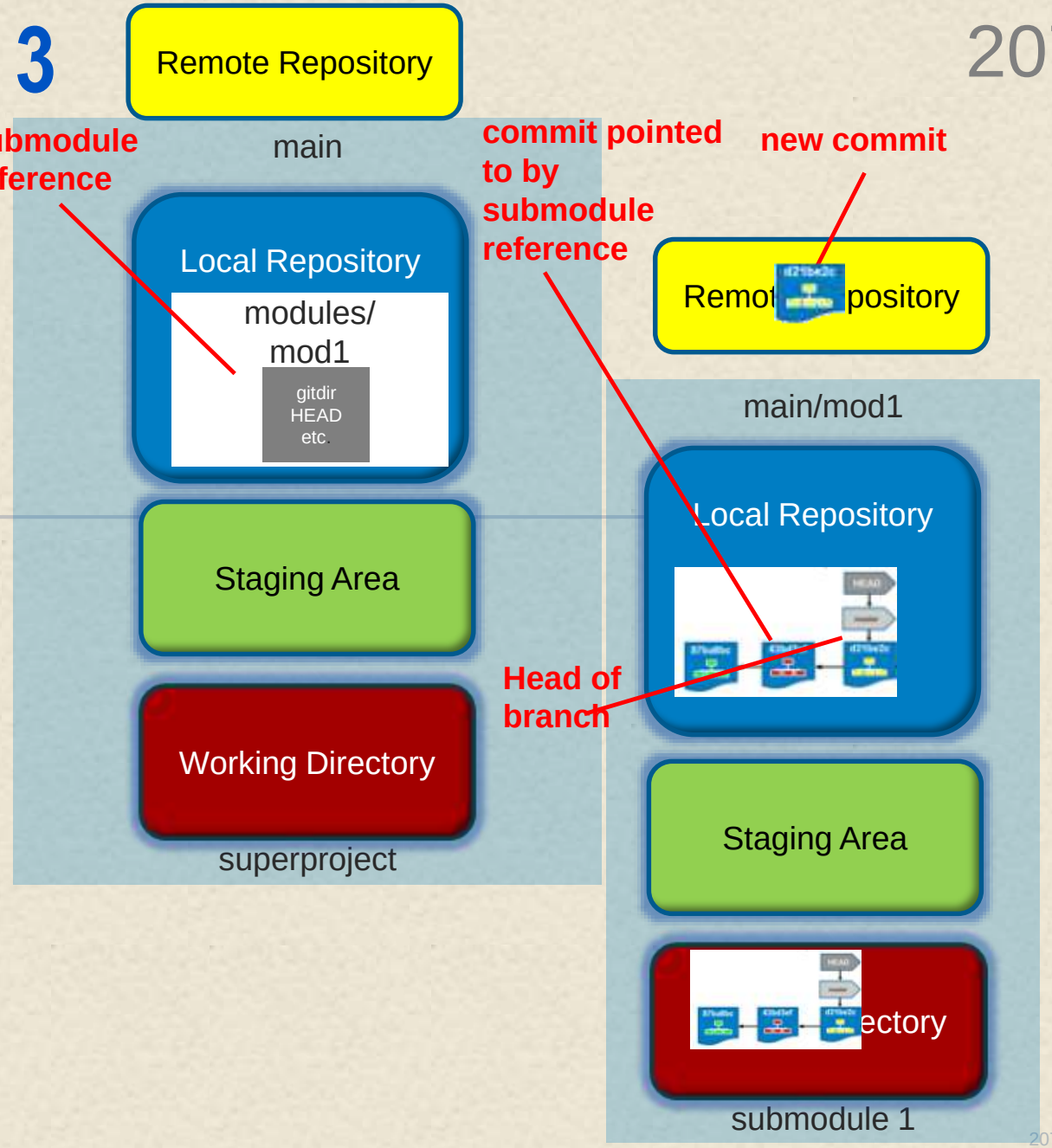
1. You can
   $ cd mod1; git checkout <branch> ; git pull



Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

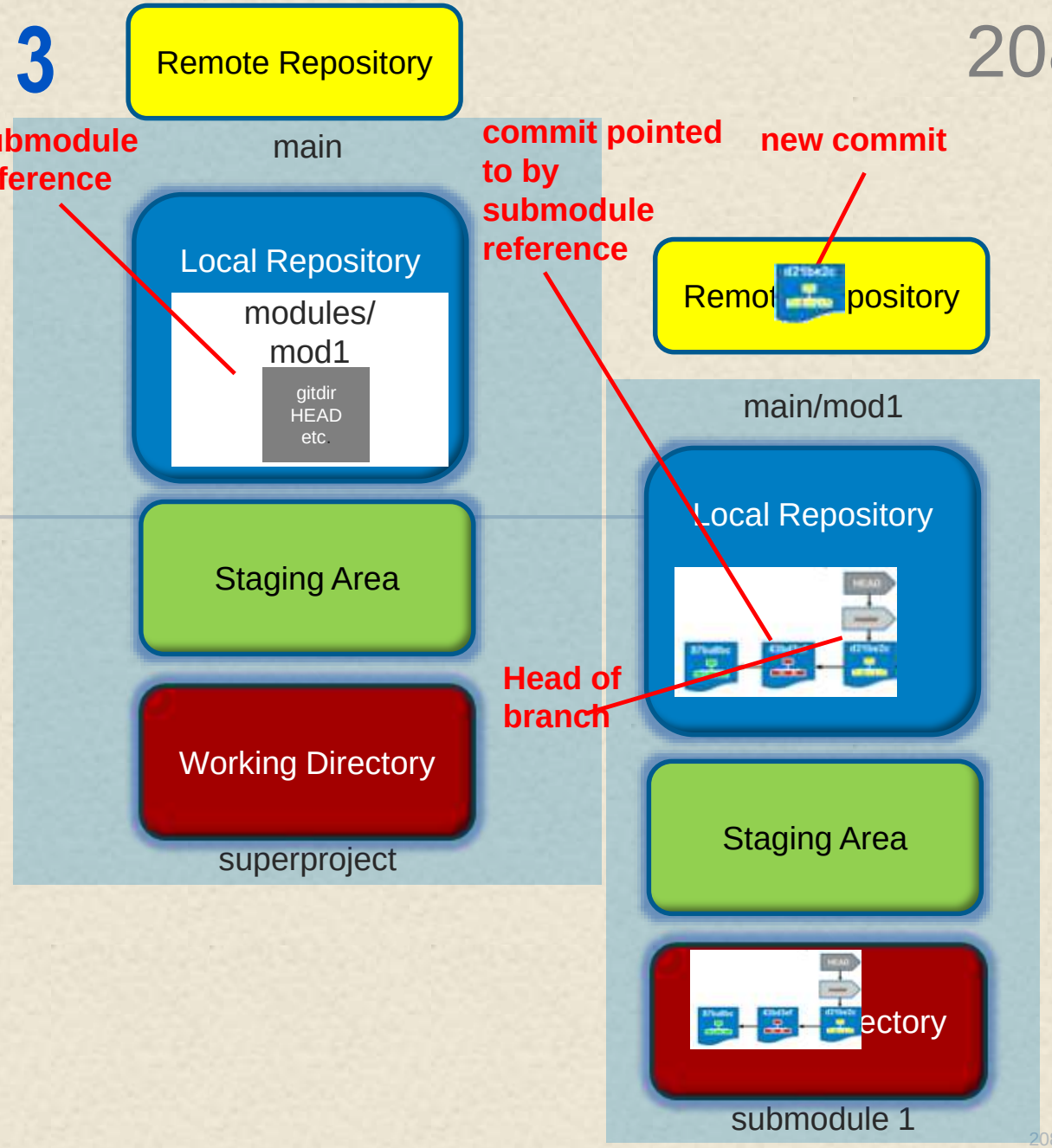


Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master



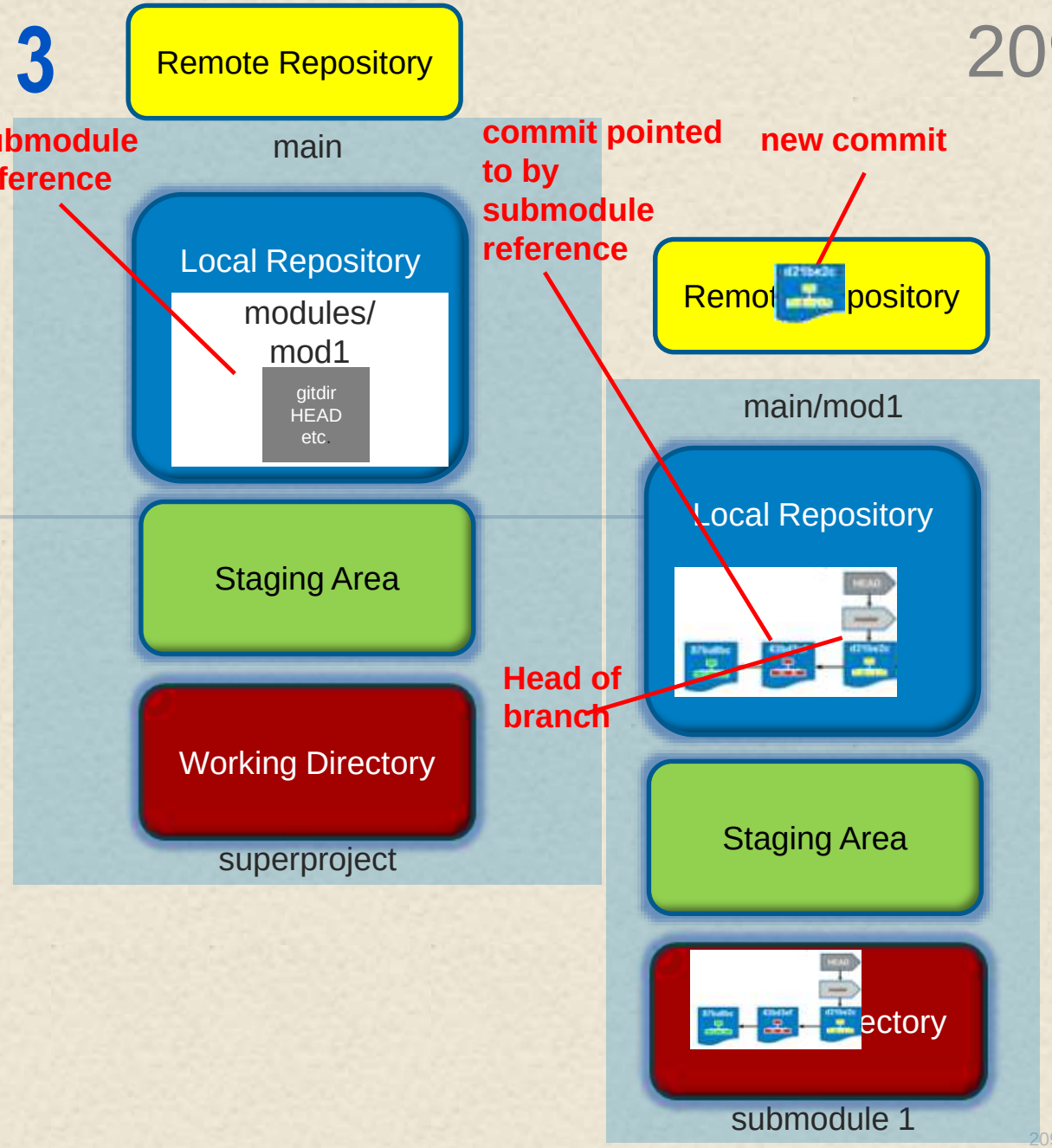
Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

   OR



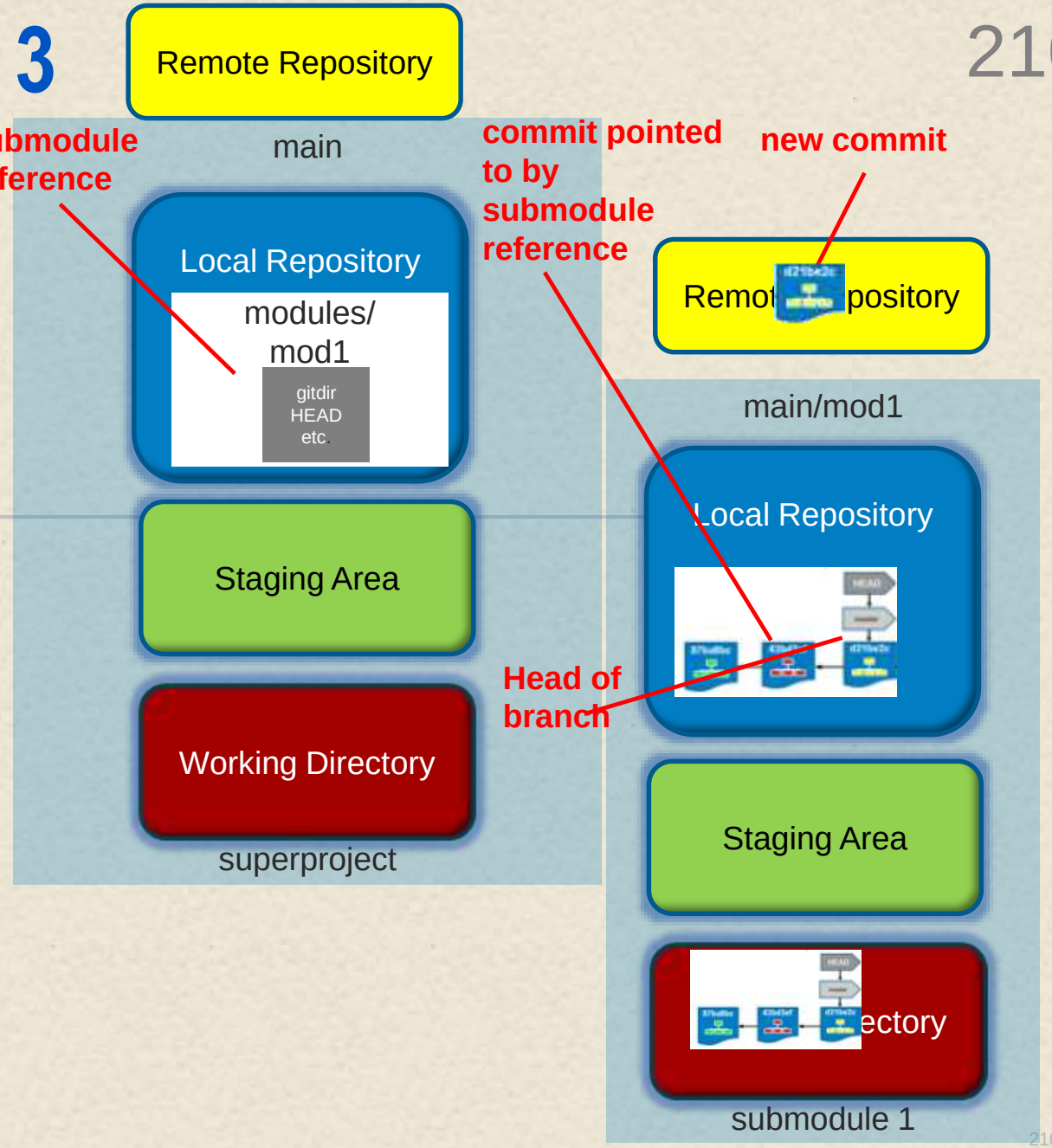
Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

   OR
   $ git submodule update --remote



Submodules 3

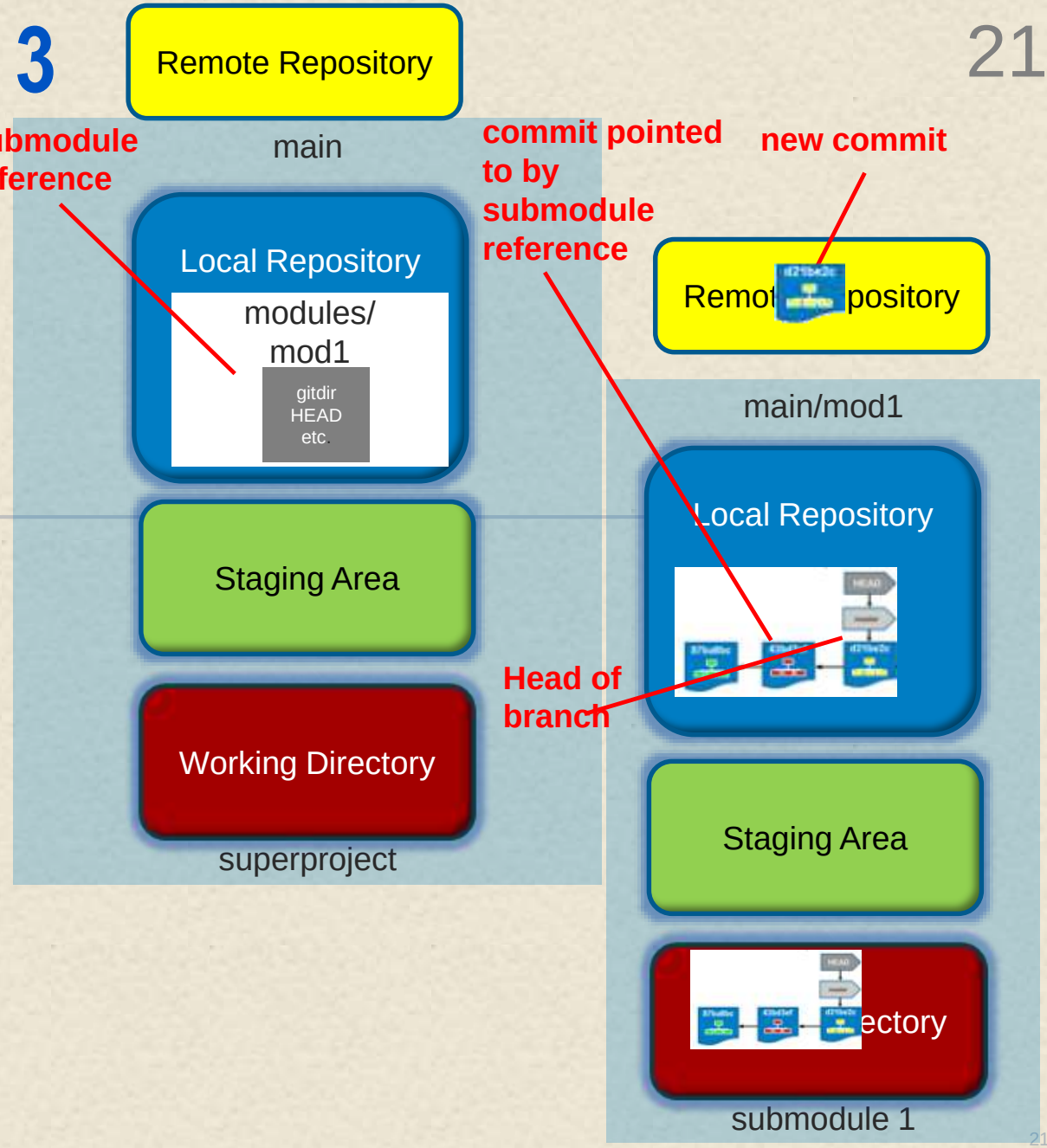
Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

   OR
   $ git submodule update --remote

2. Although module has been updated, references that the superproject has to it haven't been updated



Submodules 3

Incorporating updates to submodules:

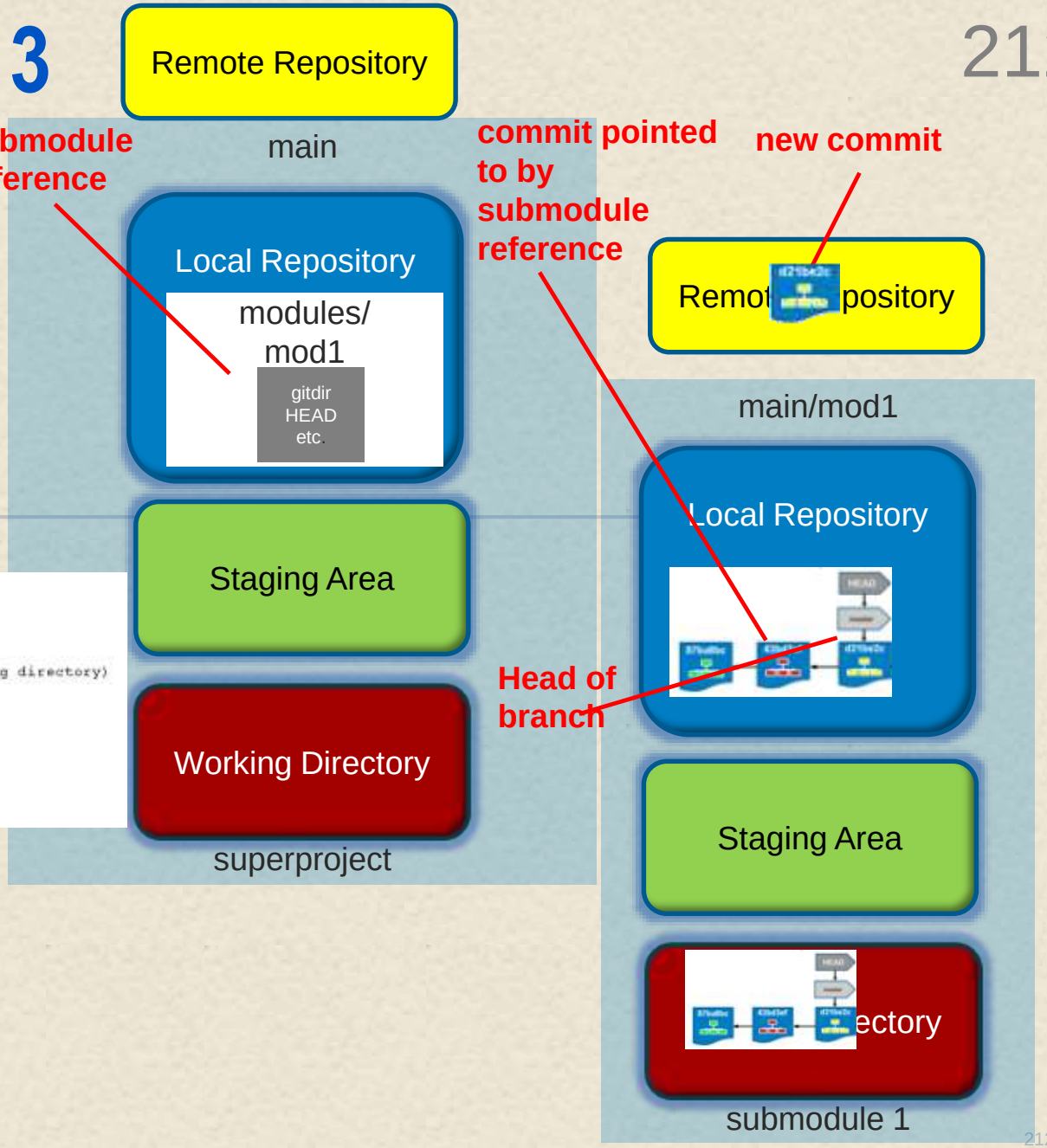
1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

   OR
   $ git submodule update --remote

2. Although module has been updated, references that the superproject has to it haven't been updated

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mod1 (new commits)

Submodules changed but not updated:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```



Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

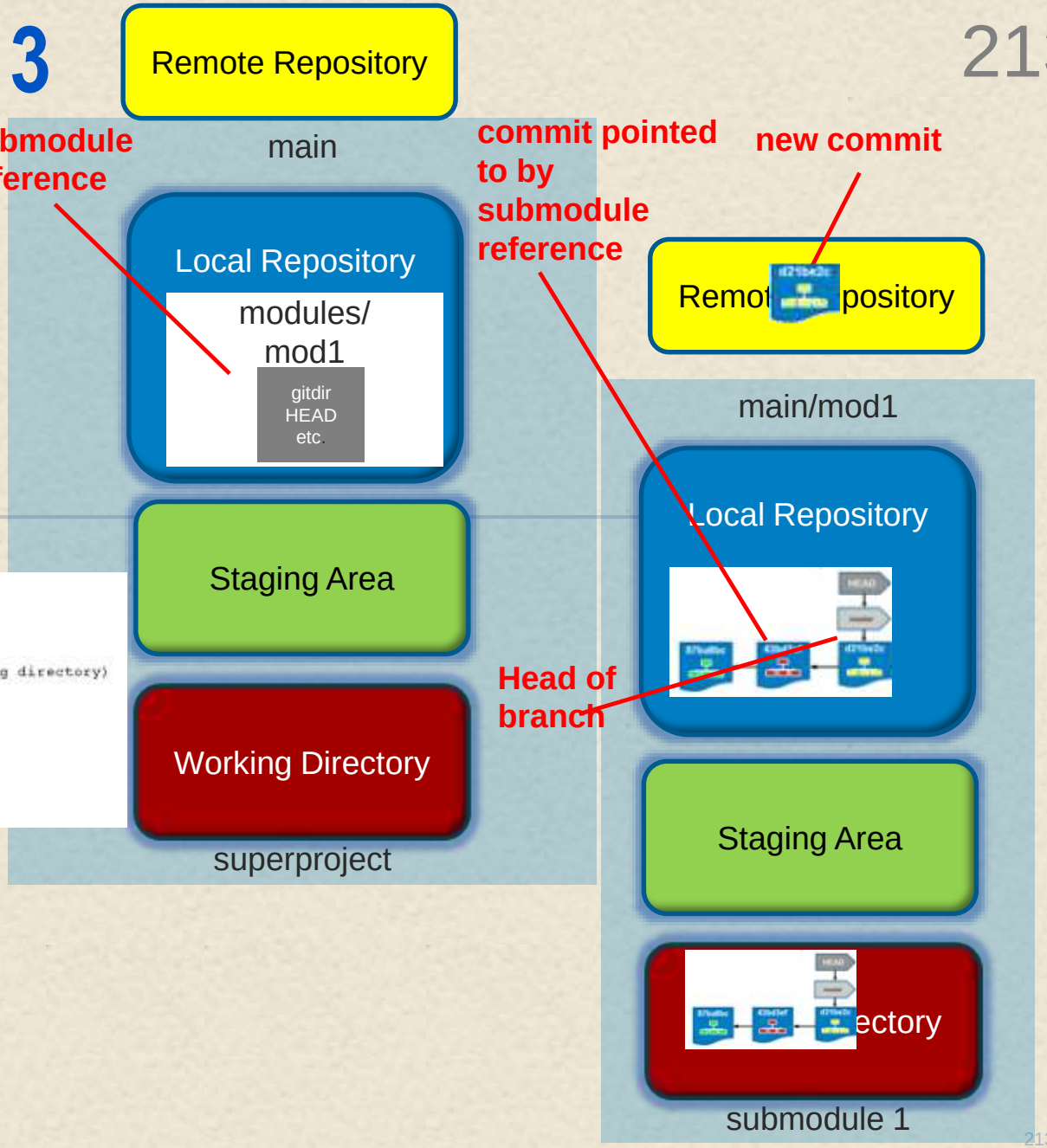
   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

   OR
   $ git submodule update --remote

2. Although module has been updated, references that the superproject has to it haven't been updated

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mod1 (new commits)

Submodules changed but not updated:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

3. To update: add, commit, and push reference changes.



Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

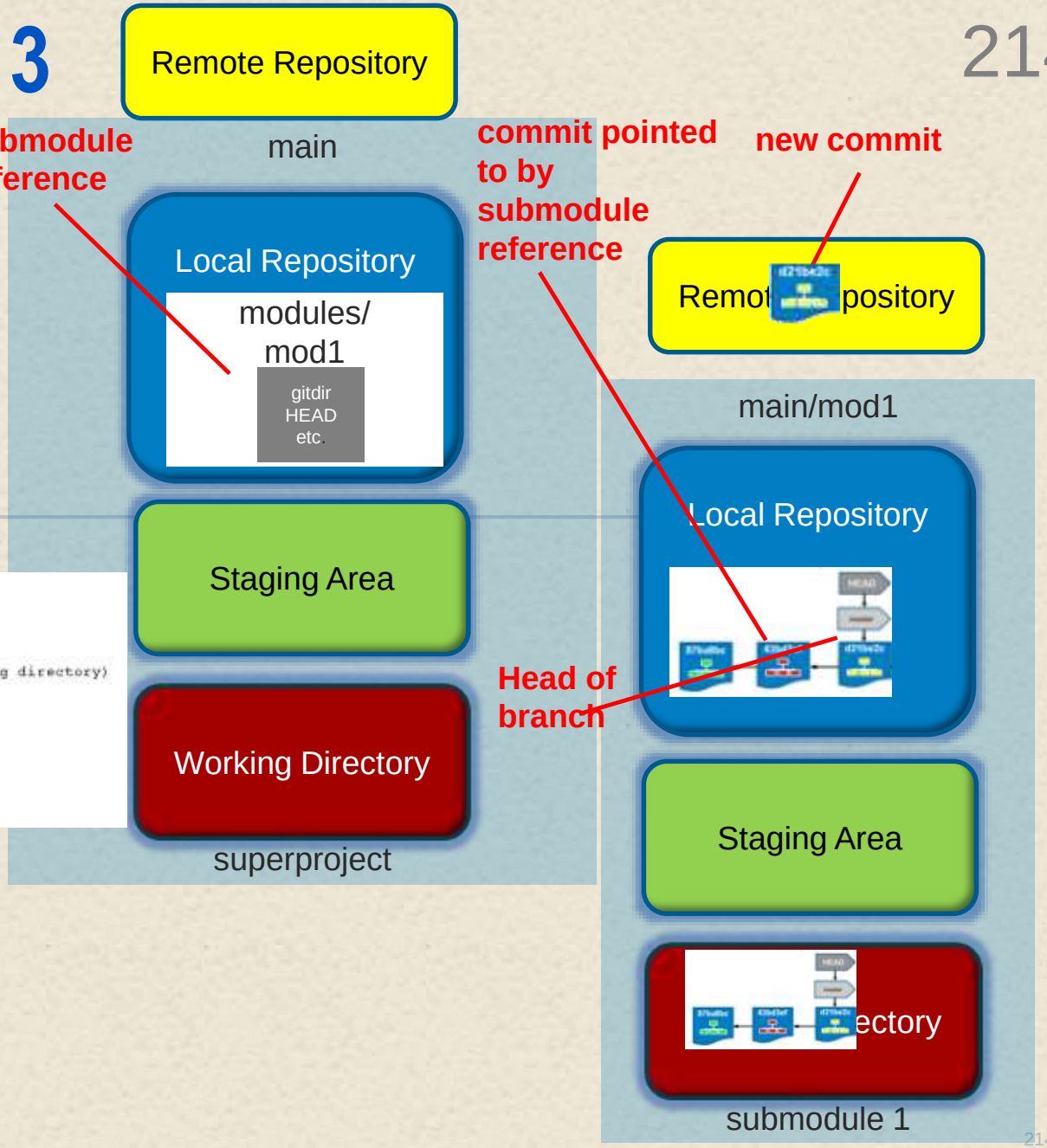
   OR
   $ git submodule update --remote

2. Although module has been updated, references that the superproject has to it haven't been updated

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mod1 (new commits)

Submodules changed but not updated:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

3. To update: add, commit, and push reference changes.

```
$ git add .

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   mod1

Submodule changes to be committed:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```



Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

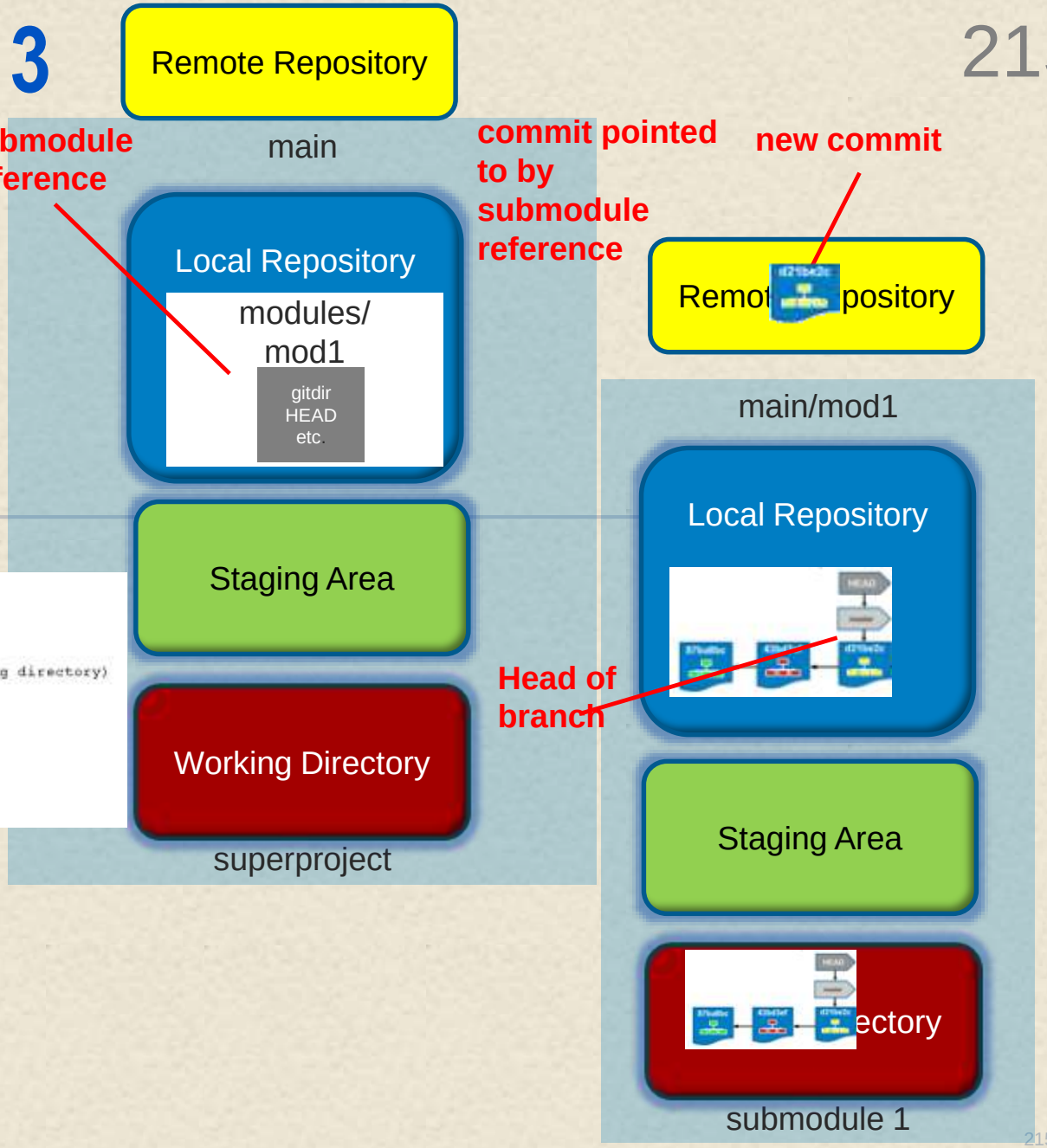
   OR
   $ git submodule update --remote

2. Although module has been updated, references that the superproject has to it haven't been updated

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mod1 (new commits)

Submodules changed but not updated:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

3. To update: add, commit, and push reference changes.

```
$ git add .

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   mod1

Submodule changes to be committed:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```



Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

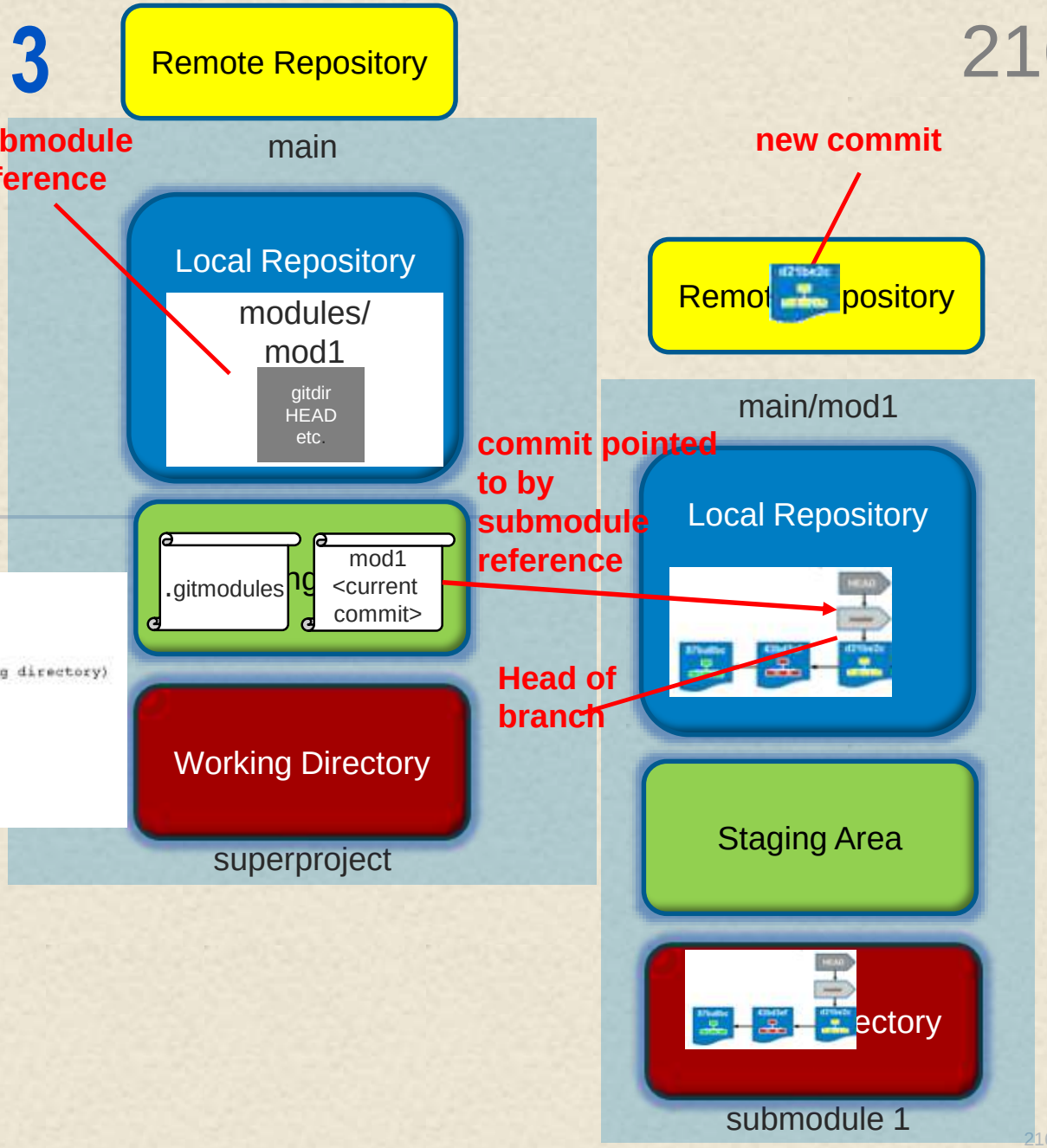
   OR
   $ git submodule update --remote

2. Although module has been updated, references that the superproject has to it haven't been updated

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mod1 (new commits)

Submodules changed but not updated:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

3. To update: add, commit, and push reference changes.

```
$ git add .

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   mod1

Submodule changes to be committed:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```



Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

   OR
   $ git submodule update --remote

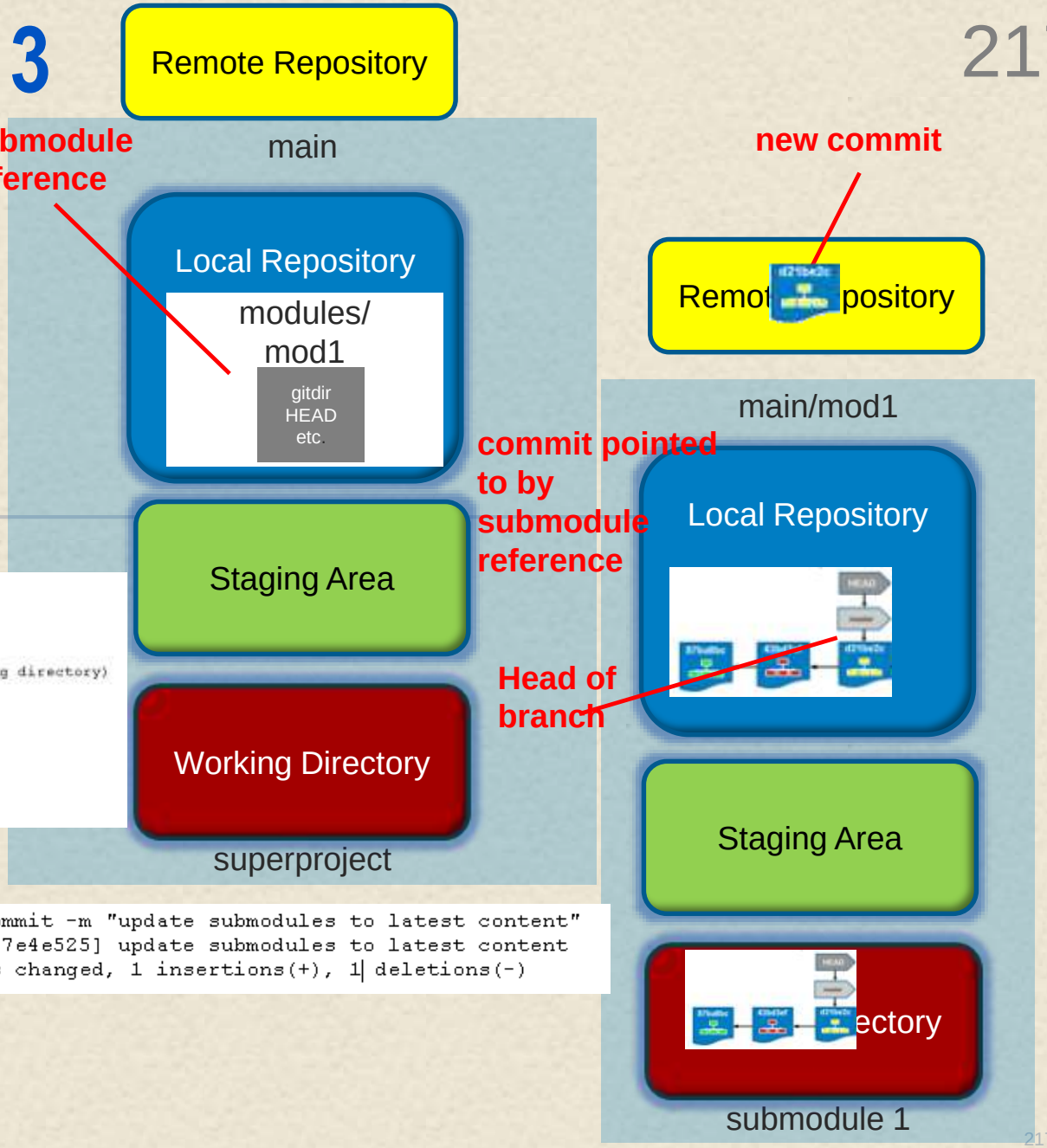
2. Although module has been updated, references that the superproject has to it haven't been updated

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mod1 (new commits)

Submodules changed but not updated:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

3. To update: add, commit, and push reference changes.

```
$ git add .

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   mod1

Submodule changes to be committed:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

```
$ git commit -m "update submodules to latest content"
[master 7e4e525] update submodules to latest content
 1 files changed, 1 insertions(+), 1 deletions(-)
```



Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

   OR
   $ git submodule update --remote

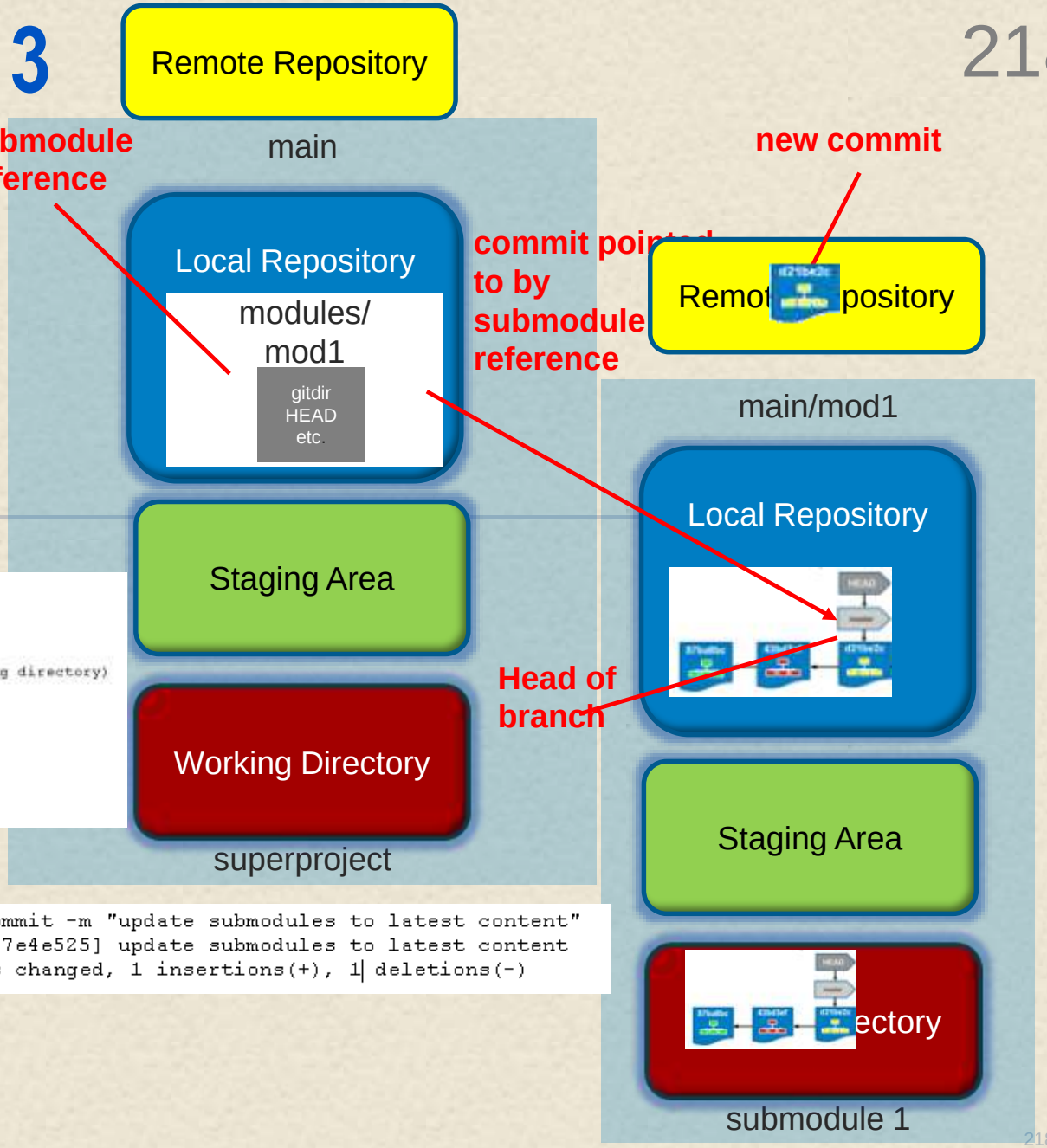
2. Although module has been updated, references that the superproject has to it haven't been updated

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mod1 (new commits)

Submodules changed but not updated:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

3. To update: add, commit, and push reference changes.

```
$ git add .

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   mod1

Submodule changes to be committed:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

```
$ git commit -m "update submodules to latest content"
[master 7e4e525] update submodules to latest content
 1 files changed, 1 insertions(+), 1 deletions(-)
```



Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

   OR
   $ git submodule update --remote

2. Although module has been updated, references that the superproject has to it haven't been updated

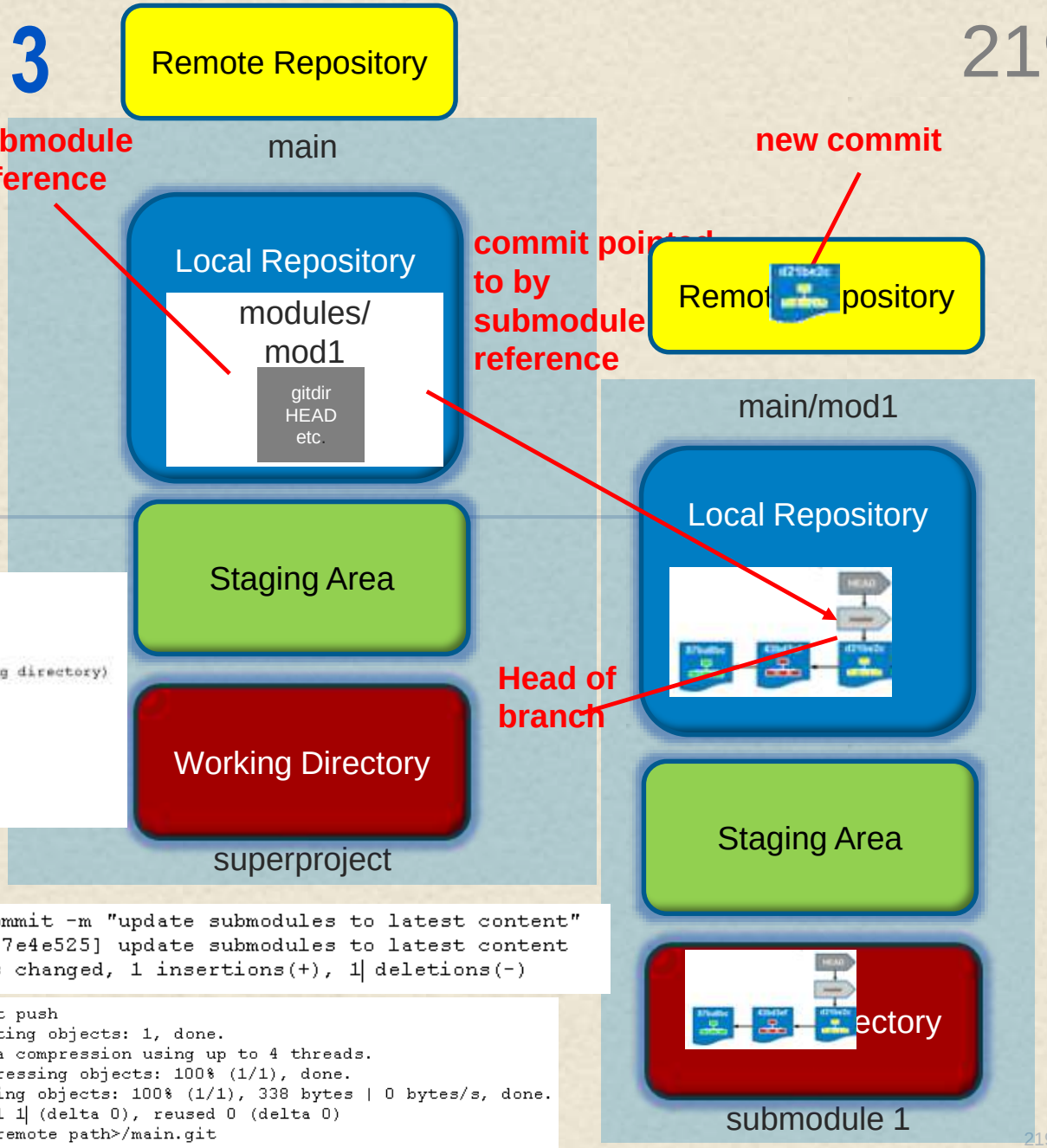
```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mod1 (new commits)

Submodules changed but not updated:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

3. To update: add, commit, and push reference changes.

```
$ git add .

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   mod1

Submodule changes to be committed:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

```
$ git commit -m "update submodules to latest content"
[master 7e4e525] update submodules to latest content
 1 files changed, 1 insertions(+), 1 deletions(-)
```

```
$ git push
Counting objects: 1, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (1/1), done.
Writing objects: 100% (1/1), 338 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To <remote path>/main.git
   2745a27..7e4e525  master -> master
```



Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

   OR
   $ git submodule update --remote

2. Although module has been updated, references that the superproject has to it haven't been updated

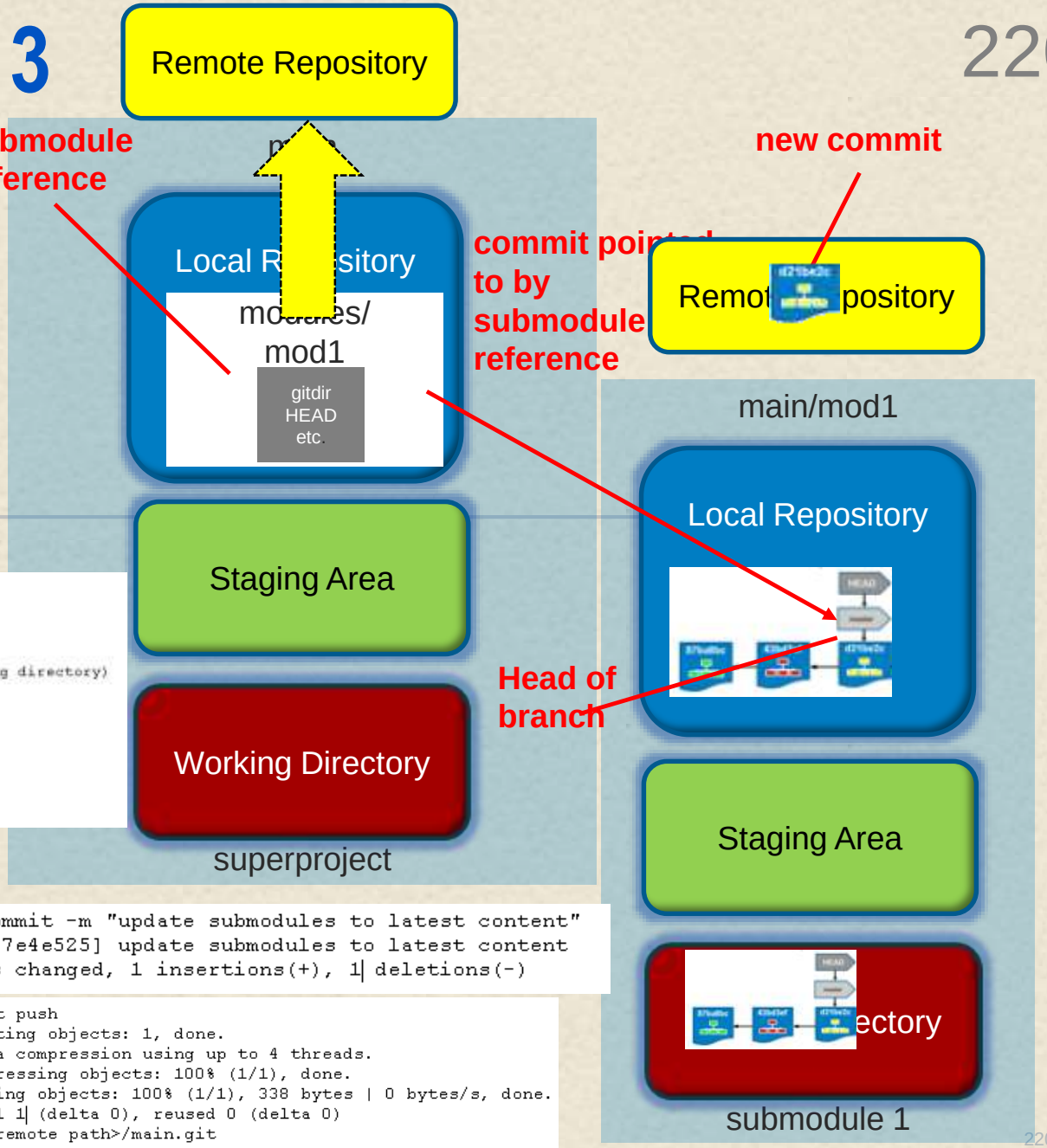
```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mod1 (new commits)

Submodules changed but not updated:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

3. To update: add, commit, and push reference changes.

```
$ git add .

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   mod1

Submodule changes to be committed:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

```
$ git commit -m "update submodules to latest content"
[master 7e4e525] update submodules to latest content
 1 files changed, 1 insertions(+), 1 deletions(-)
```

```
$ git push
Counting objects: 1, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (1/1), done.
Writing objects: 100% (1/1), 338 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To <remote path>/main.git
   2745a27..7e4e525  master -> master
```



Submodules 3

Incorporating updates to submodules:

1. You can
   $ cd mod1; git checkout <branch> ; git pull

   OR
   $ git pull --recurse-submodules; cd <module dir>; git merge origin/master

   OR
   $ git submodule update --remote

2. Although module has been updated, references that the superproject has to it haven't been updated

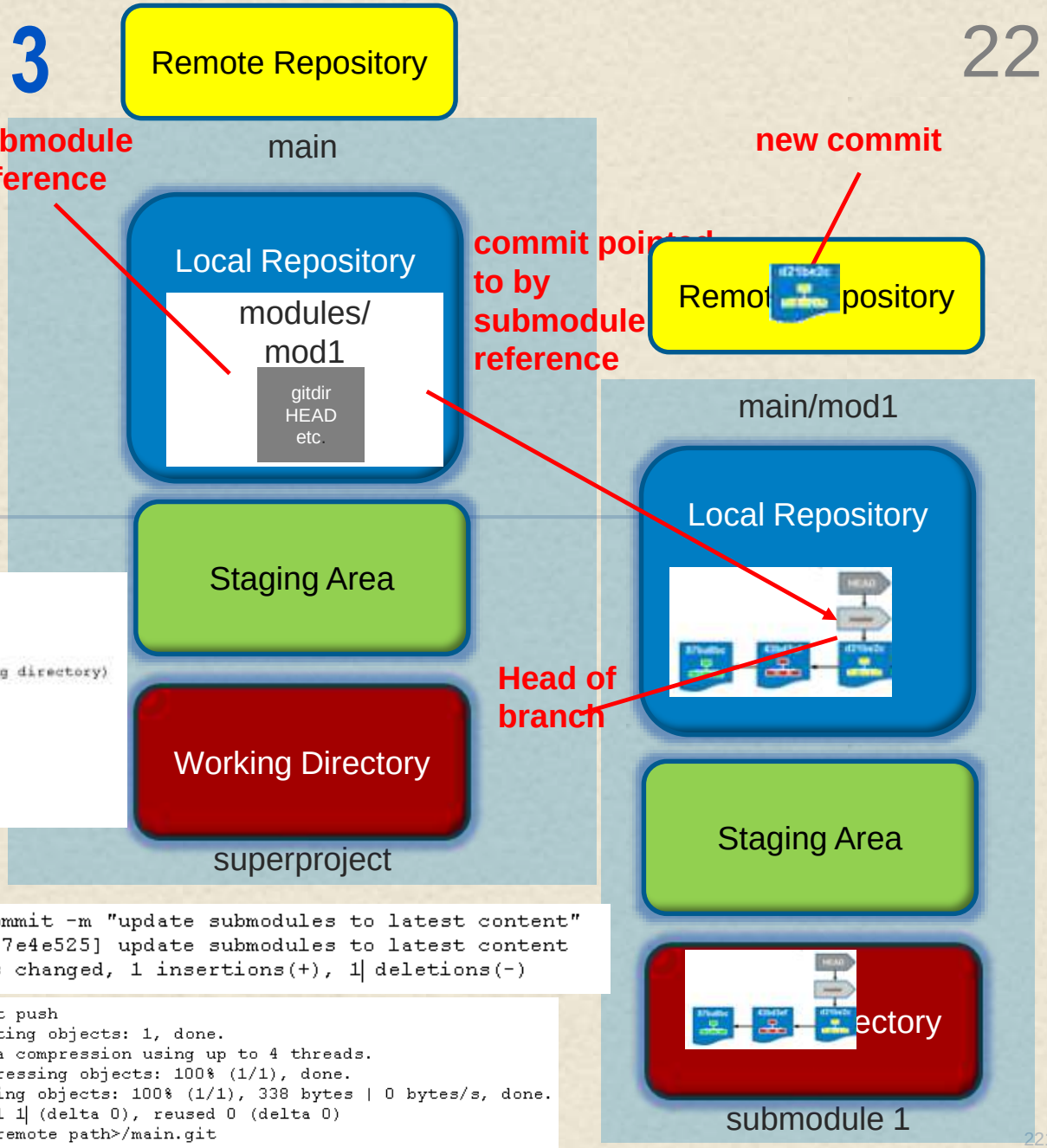
```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mod1 (new commits)

Submodules changed but not updated:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

3. To update: add, commit, and push reference changes.

```
$ git add .

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   mod1

Submodule changes to be committed:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file
```

```
$ git commit -m "update submodules to latest content"
[master 7e4e525] update submodules to latest content
 1 files changed, 1 insertions(+), 1 deletions(-)
```

```
$ git push
Counting objects: 1, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (1/1), done.
Writing objects: 100% (1/1), 338 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To <remote path>/main.git
   2745a27..7e4e525  master -> master
```



Submodule - Challenges

- Challenges around using submodules nearly always involve keeping submodule content (and “current” commit) in sync with submodule references in superproject
- If references are wrong, operations like “git submodule update” will backlevel submodule content to commits in reference
- If these references are out of sync and that inconsistency is pushed to the remote for the superproject, then other users that pull that version of the superproject can end up back-leveling their submodules, even if they’ve updated their superproject before.

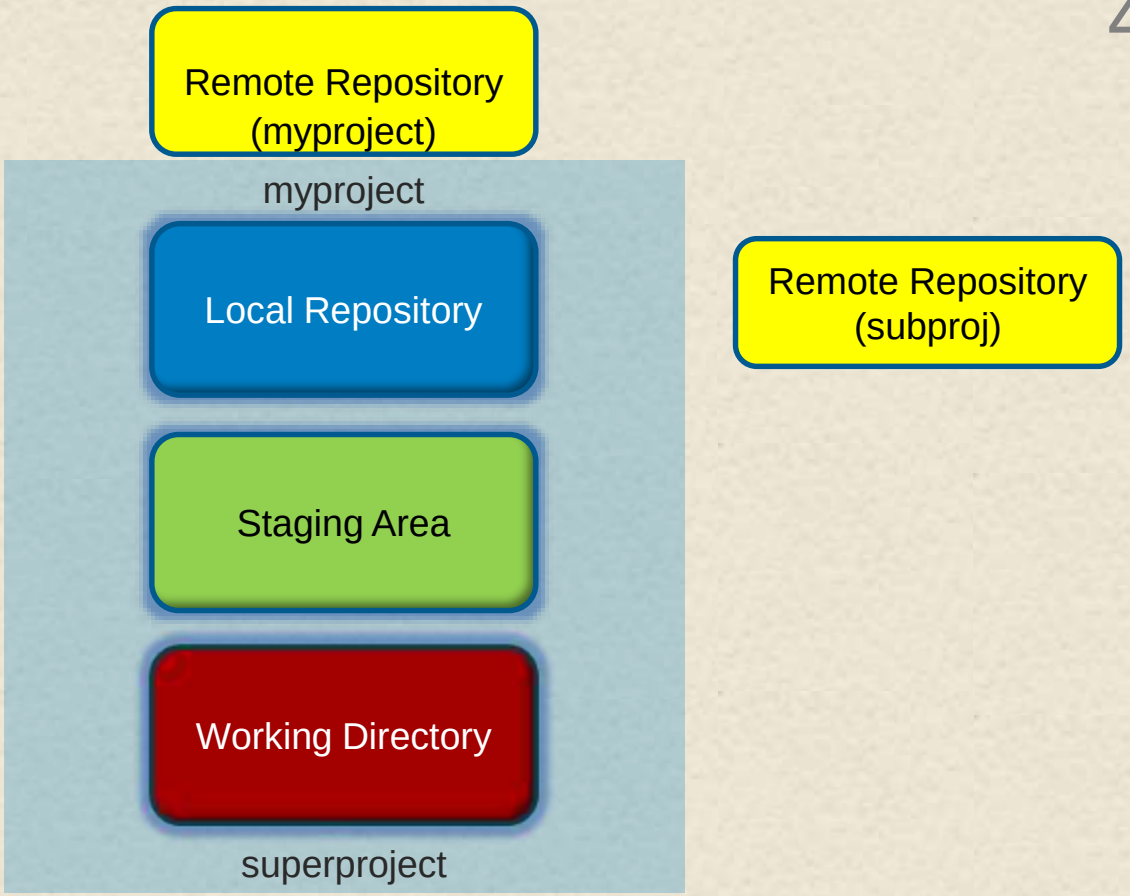
Command: Subtrees

- Purpose - Allows including **a copy** of a separate repository with your current repository
- Use case - include **a copy** of a Git repository for one or more dependencies along with the original repository for a project
- Syntax

```
git subtree add   -P <prefix> <commit>
git subtree add   -P <prefix> <repository> <ref>
git subtree pull  -P <prefix> <repository> <ref>
git subtree push  -P <prefix> <repository> <ref>
git subtree merge -P <prefix> <commit>
git subtree split -P <prefix> [OPTIONS] [<commit>]
```

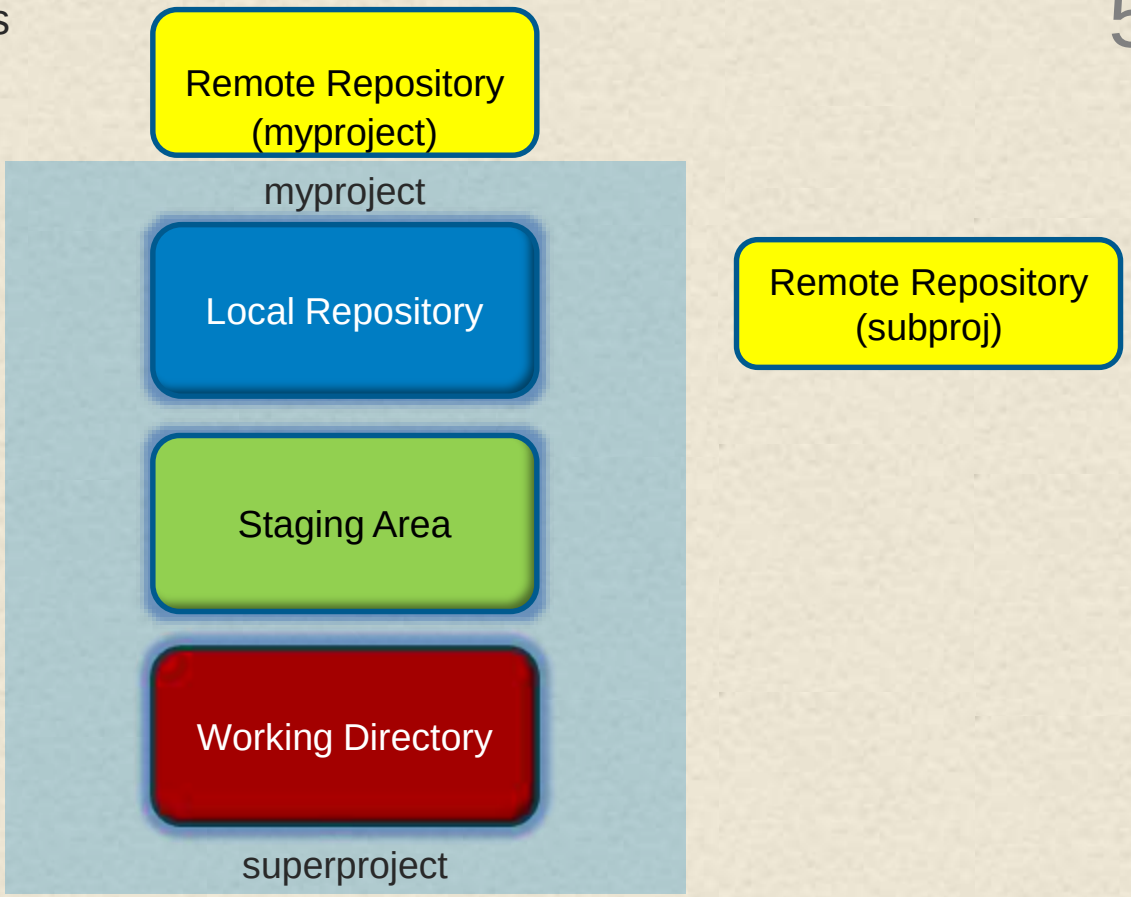
- Notes
  - No links like a submodule - just a copy in a subdirectory
  - Advantage - no links to maintain
  - Disadvantage - extra content to carry around with your project

Subtrees



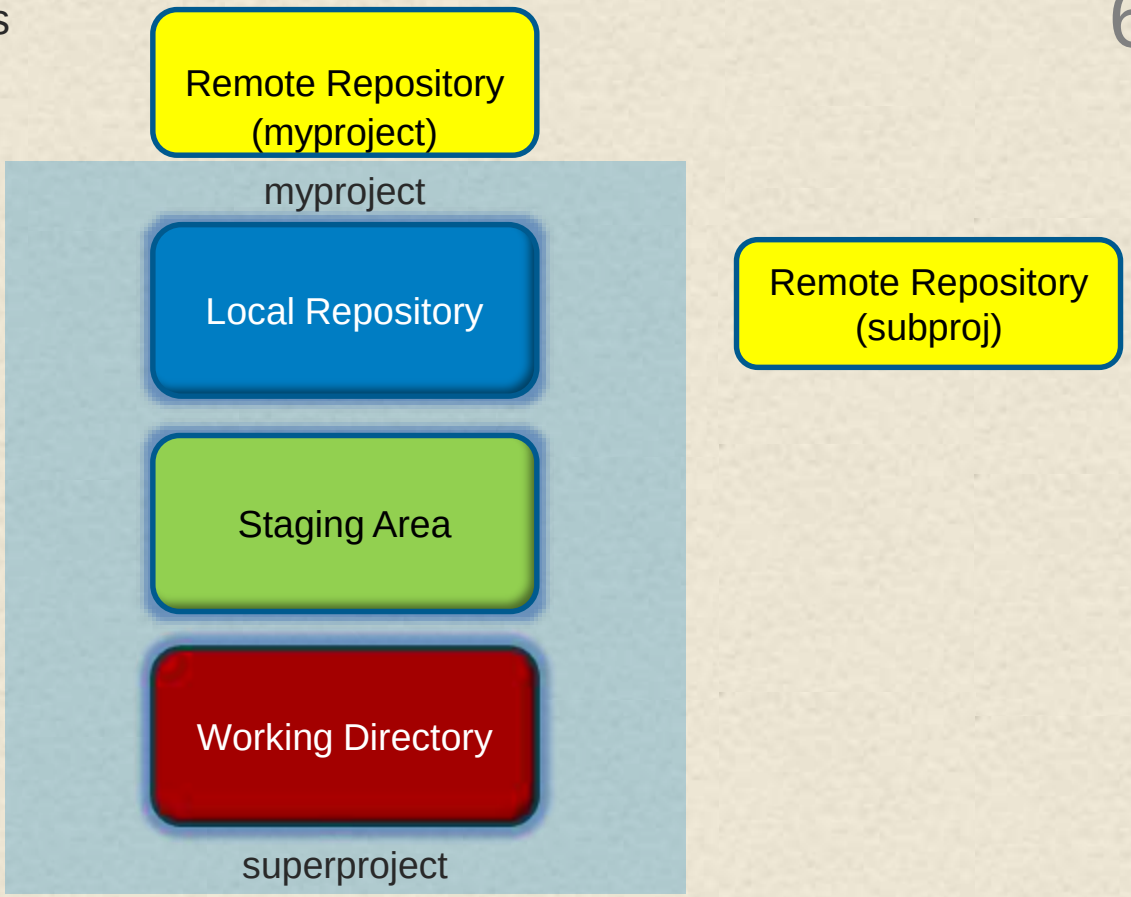
Subtrees

- Adding a copy from a remote as a subtree



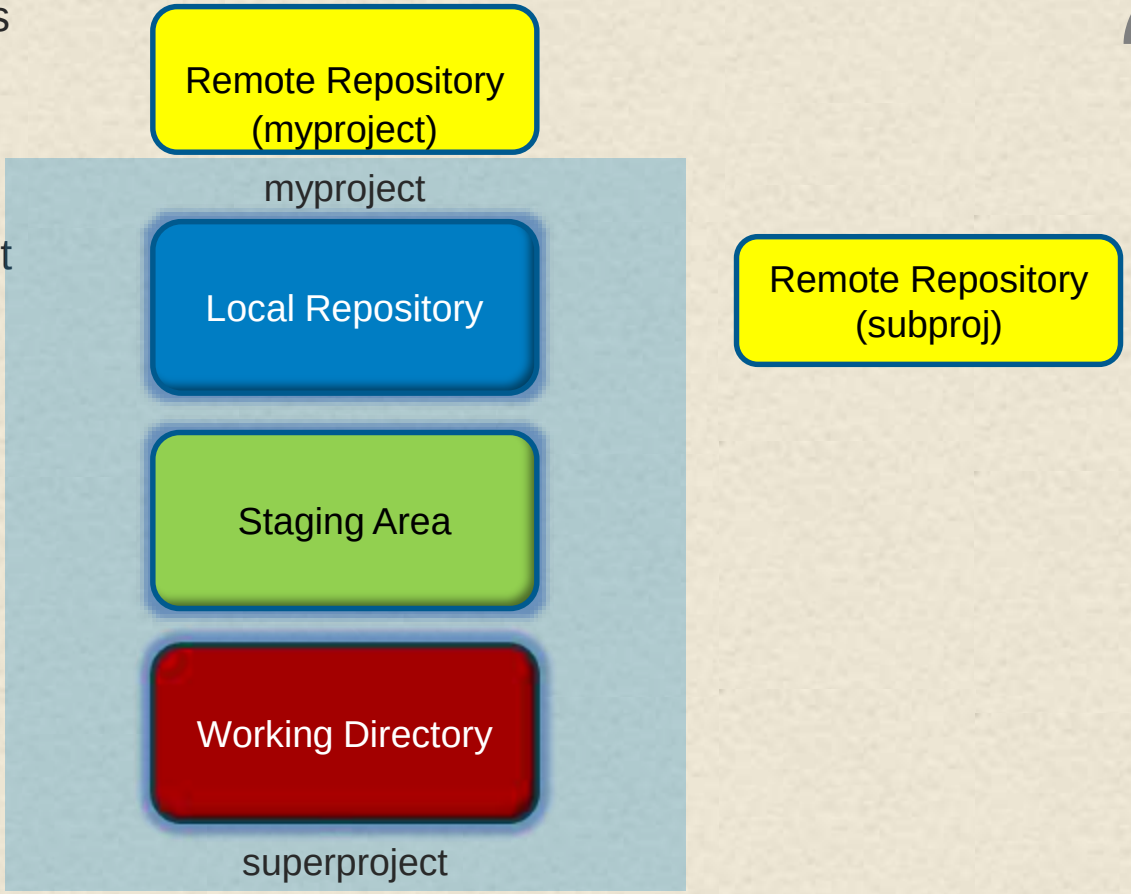
Subtrees

- Adding a copy from a remote as a subtree
- cd myproject



Subtrees

- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
-



Subtrees

- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
-

Subtrees

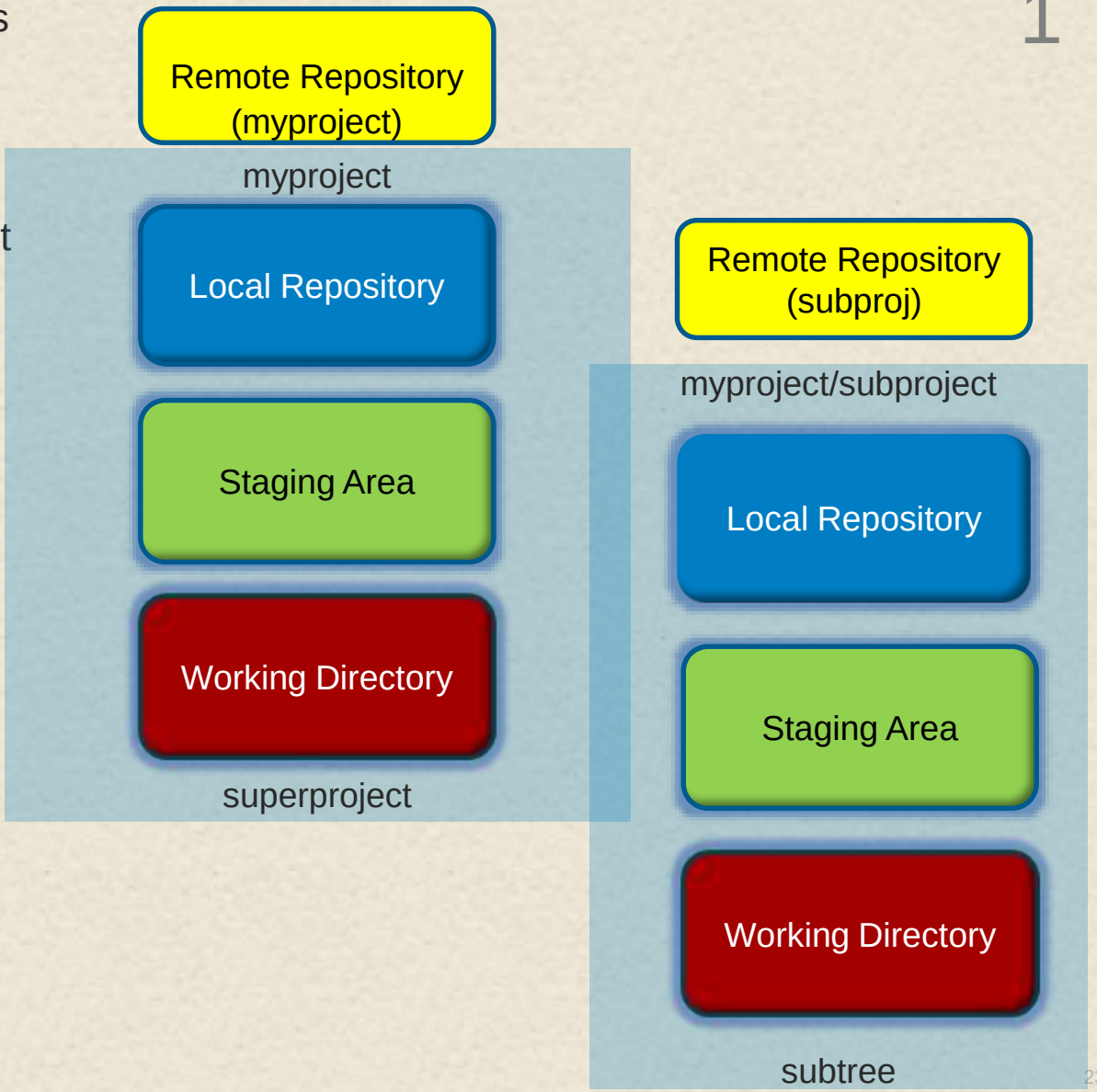
- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
  - Use “prefix” option to specify path for subproject

Subtrees

- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
  - Use “prefix” option to specify path for subproject
  - Use “squash” option to compress history from remote before adding it

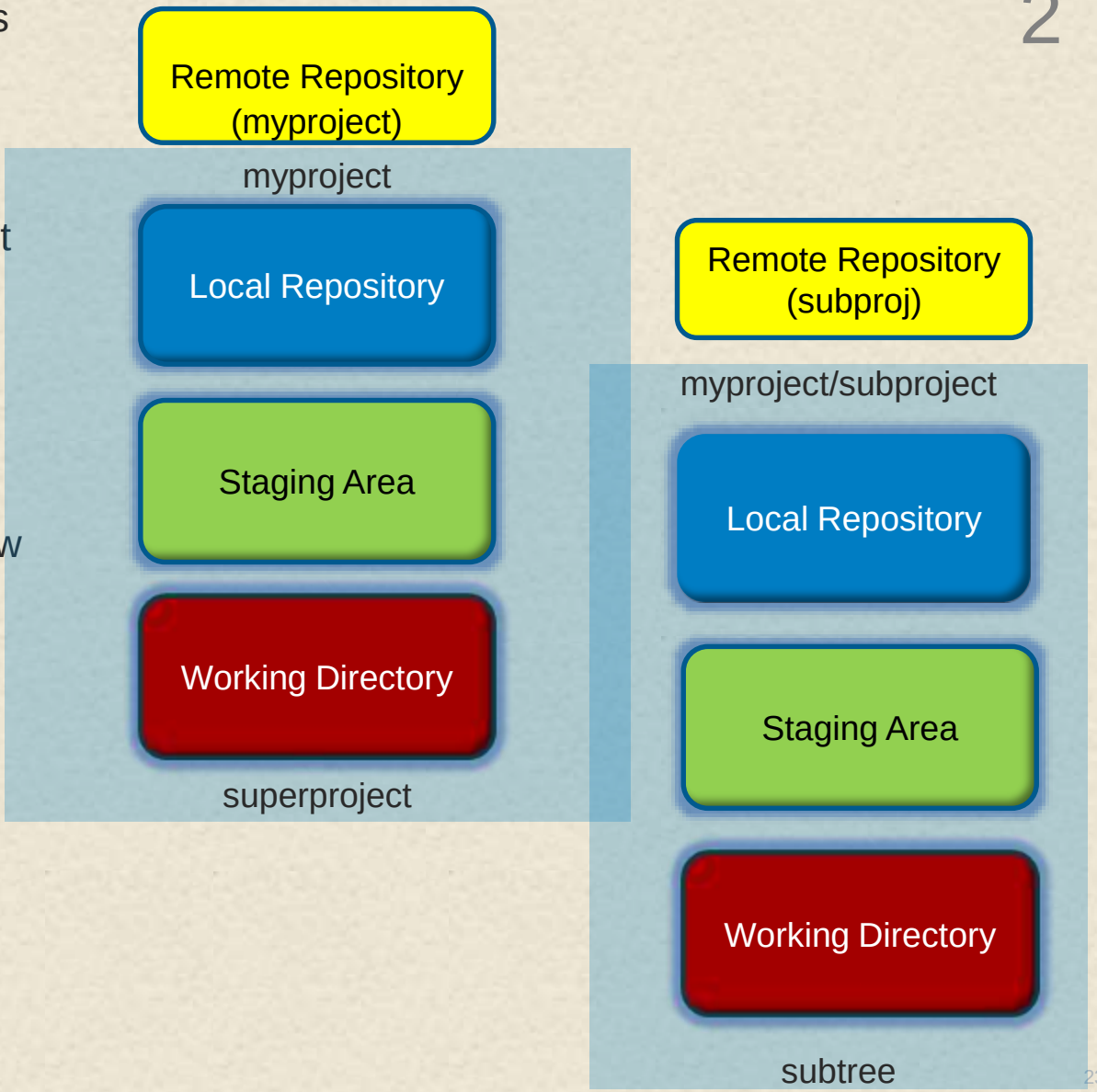
Subtrees

- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
  - Use “prefix” option to specify path for subproject
  - Use “squash” option to compress history from remote before adding it
  - branch (master) is optional



Subtrees

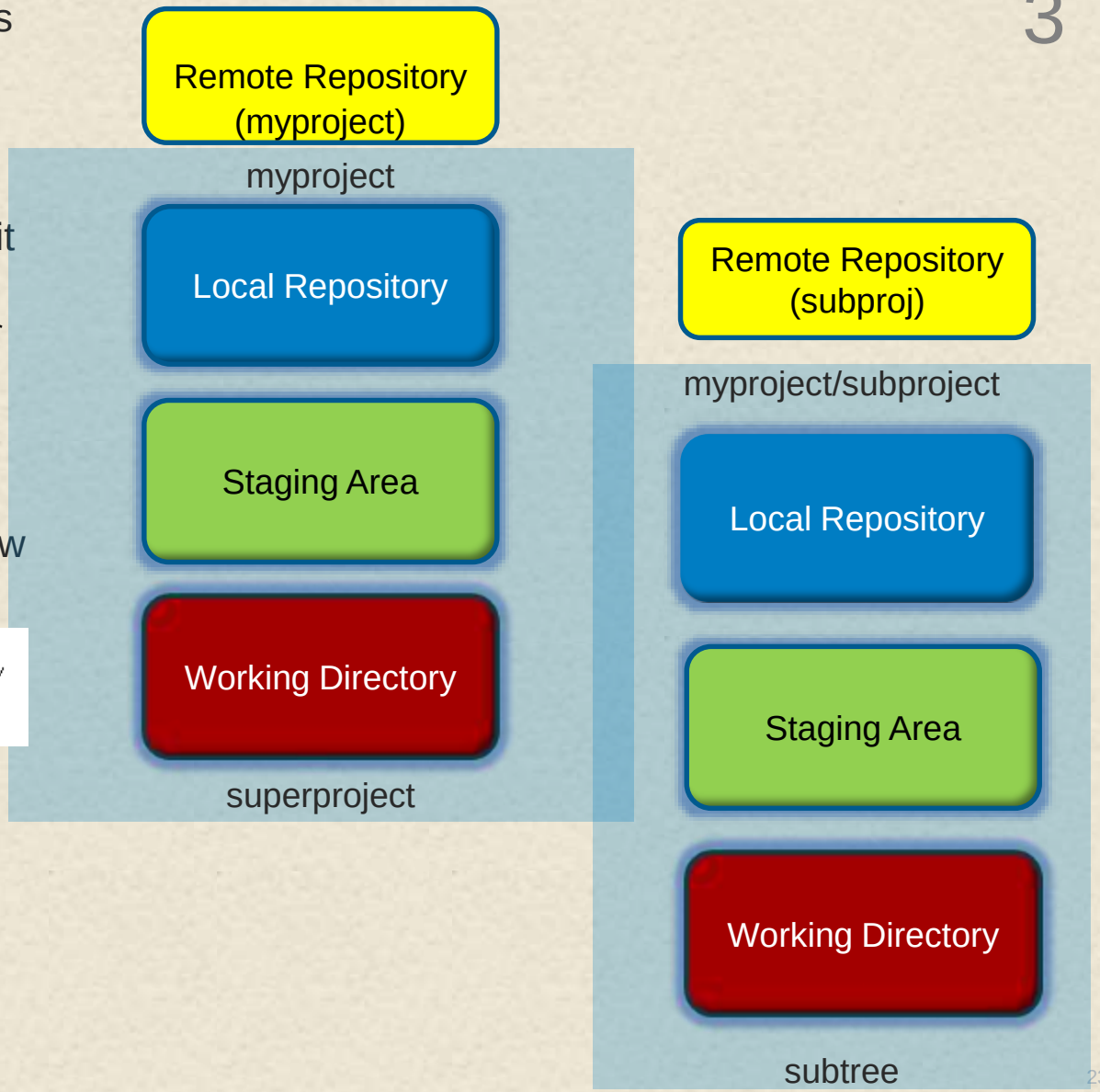
- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
  - Use “prefix” option to specify path for subproject
  - Use “squash” option to compress history from remote before adding it
  - branch (master) is optional
- directory listing of myproject now shows subproj as subdirectory



Subtrees

- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
  - Use “prefix” option to specify path for subproject
  - Use “squash” option to compress history from remote before adding it
  - branch (master) is optional
- directory listing of myproject now shows subproj as subdirectory

```
~/subtrees/local/myproject$ ls
file1.txt   file2.txt   file3.txt   subproject/
~/subtrees/local/myproject$ ls subproject
subfile1.txt  subfile2.txt
```

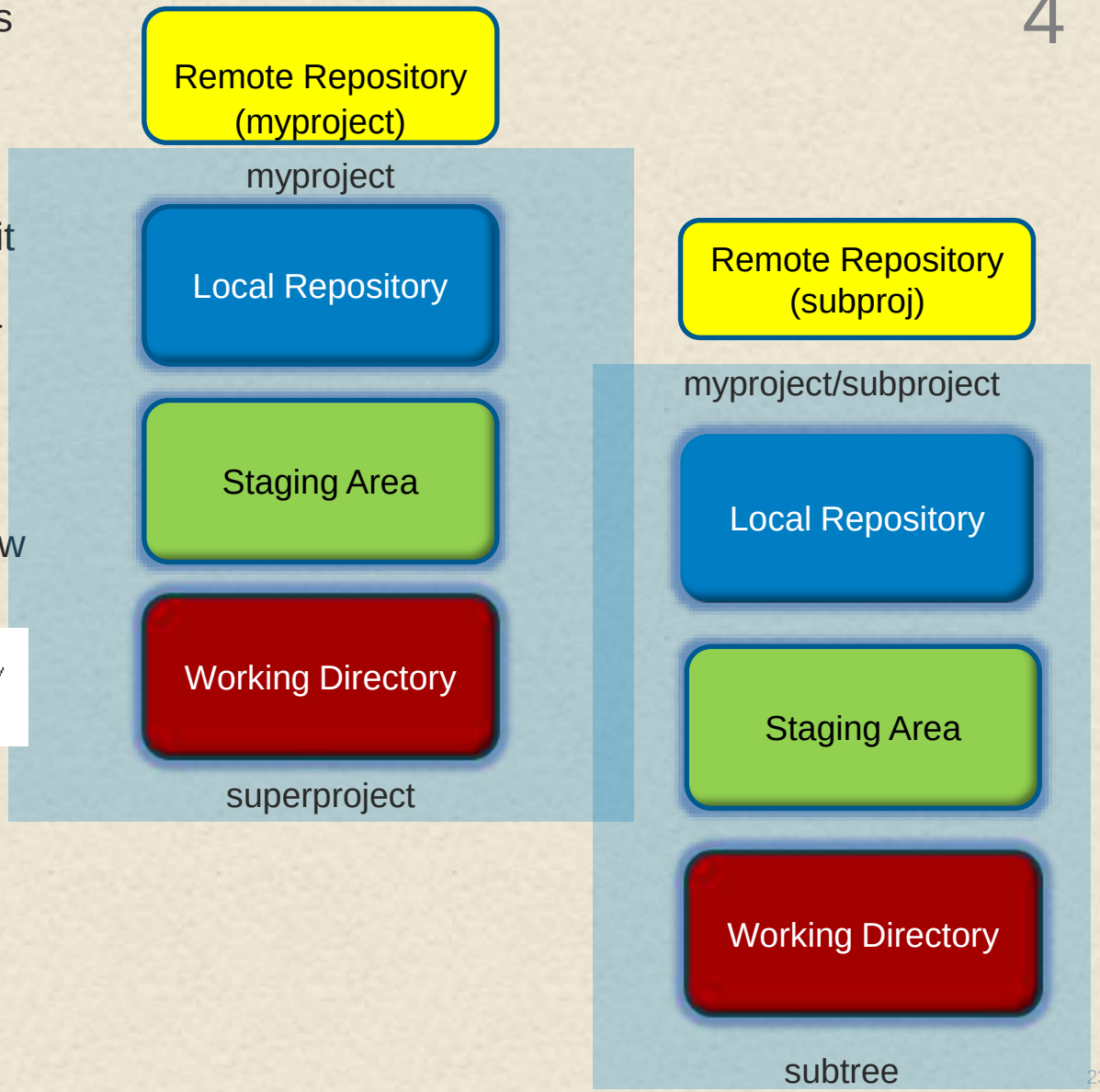


Subtrees

- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
  - Use “prefix” option to specify path for subproject
  - Use “squash” option to compress history from remote before adding it
  - branch (master) is optional
- directory listing of myproject now shows subproj as subdirectory

```
~/subtrees/local/myproject$ ls
file1.txt   file2.txt   file3.txt   subproject/
~/subtrees/local/myproject$ ls subproject
subfile1.txt  subfile2.txt
```

- Looking in the logs of the subproject will show the squashed history

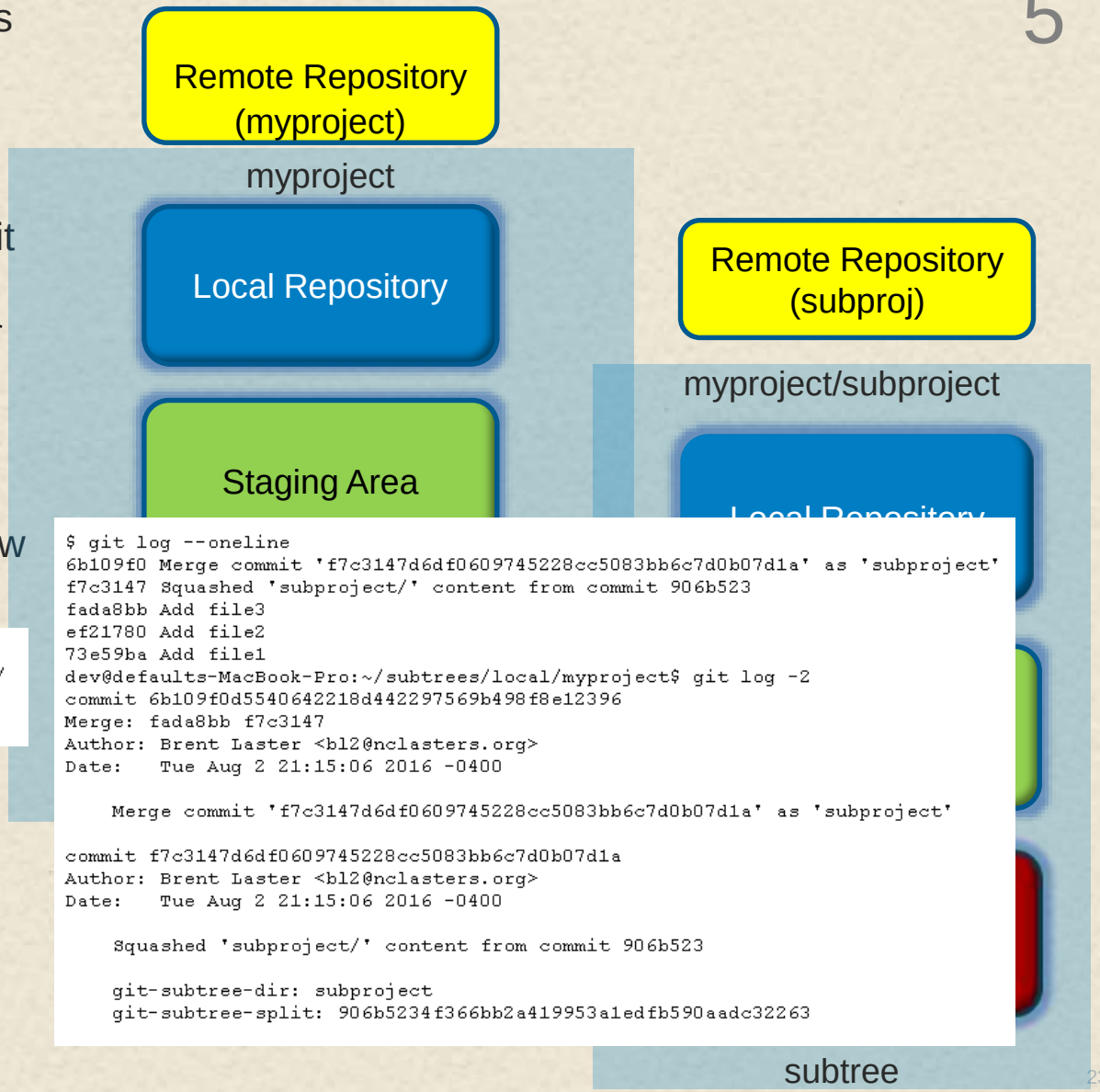


Subtrees

- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
  - Use “prefix” option to specify path for subproject
  - Use “squash” option to compress history from remote before adding it
  - branch (master) is optional
- directory listing of myproject now shows subproj as subdirectory

```
~/subtrees/local/myproject$ ls
file1.txt   file2.txt   file3.txt   subproject/
~/subtrees/local/myproject$ ls subproject
subfile1.txt  subfile2.txt
```

- Looking in the logs of the subproject will show the squashed history



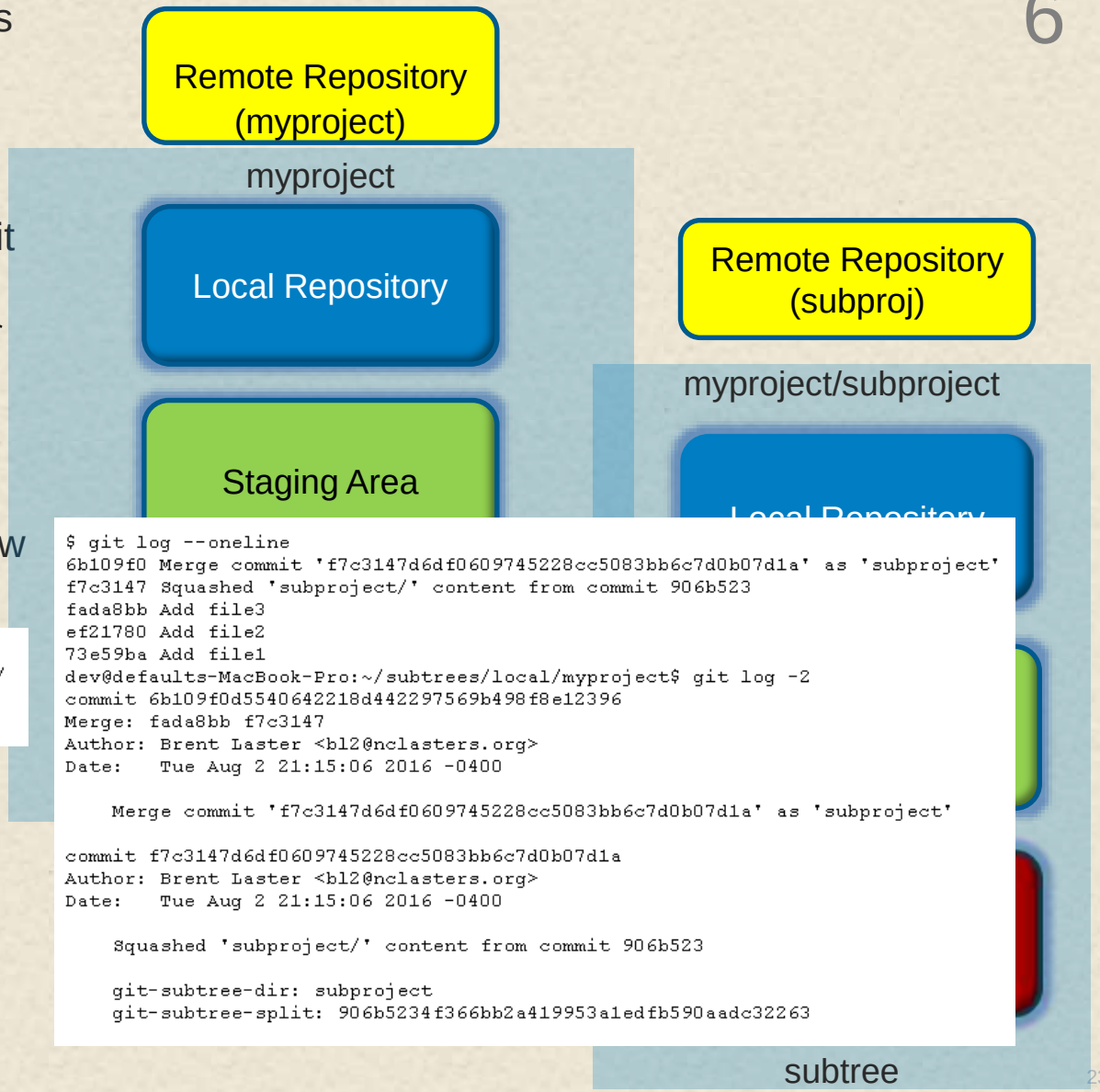

```
$ git log --oneline
6b109f0 Merge commit 'f7c3147d6df0609745228cc5083bb6c7d0b07d1a' as 'subproject'
f7c3147 Squashed 'subproject/' content from commit 906b523
fada8bb Add file3
ef21780 Add file2
73e59ba Add file1
dev@defaults-MacBook-Pro:~/subtrees/local/myproject$ git log -2
commit 6b109f0d5540642218d442297569b498f8e12396
Merge: fada8bb f7c3147
Author: Brent Laster <bl2@nclasters.org>
Date:   Tue Aug 2 21:15:06 2016 -0400

    Merge commit 'f7c3147d6df0609745228cc5083bb6c7d0b07d1a' as 'subproject'

commit f7c3147d6df0609745228cc5083bb6c7d0b07d1a
Author: Brent Laster <bl2@nclasters.org>
Date:   Tue Aug 2 21:15:06 2016 -0400

    Squashed 'subproject/' content from commit 906b523

    git-subtree-dir: subproject
    git-subtree-split: 906b5234f366bb2a419953a1edfb590aadc32263
```

Subtrees

- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
  - Use “prefix” option to specify path for subproject
  - Use “squash” option to compress history from remote before adding it
  - branch (master) is optional
- directory listing of myproject now shows subproj as subdirectory

```
~/subtrees/local/myproject$ ls
file1.txt   file2.txt   file3.txt   subproject/
~/subtrees/local/myproject$ ls subproject
subfile1.txt  subfile2.txt
```

- Looking in the logs of the subproject will show the squashed history
- To get the latest, use pull



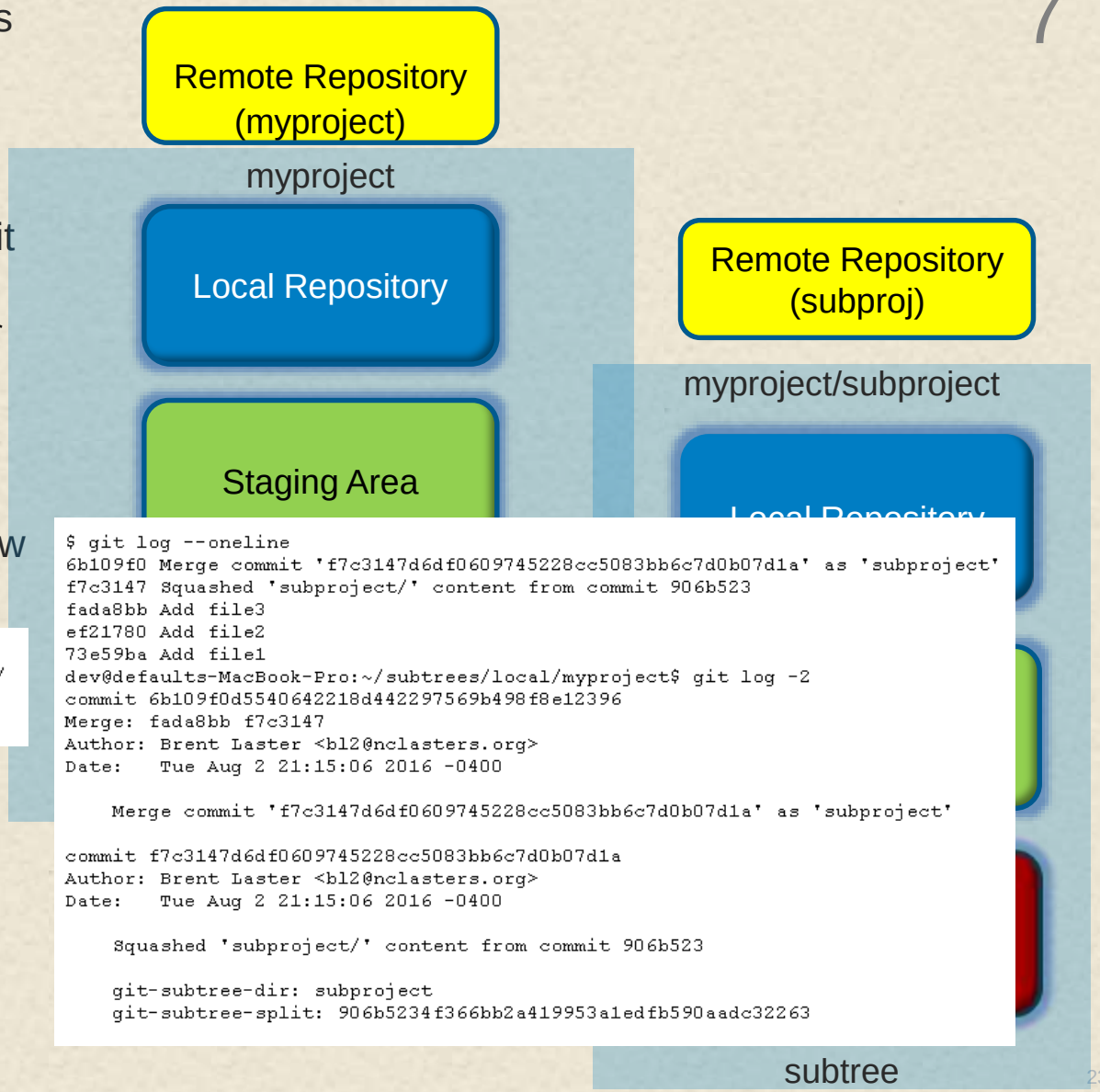

```
$ git log --oneline
6b109f0 Merge commit 'f7c3147d6df0609745228cc5083bb6c7d0b07d1a' as 'subproject'
f7c3147 Squashed 'subproject/' content from commit 906b523
fada8bb Add file3
ef21780 Add file2
73e59ba Add file1
dev@defaults-MacBook-Pro:~/subtrees/local/myproject$ git log -2
commit 6b109f0d5540642218d442297569b498f8e12396
Merge: fada8bb f7c3147
Author: Brent Laster <bl2@nclasters.org>
Date:   Tue Aug 2 21:15:06 2016 -0400

    Merge commit 'f7c3147d6df0609745228cc5083bb6c7d0b07d1a' as 'subproject'

commit f7c3147d6df0609745228cc5083bb6c7d0b07d1a
Author: Brent Laster <bl2@nclasters.org>
Date:   Tue Aug 2 21:15:06 2016 -0400

    Squashed 'subproject/' content from commit 906b523

    git-subtree-dir: subproject
    git-subtree-split: 906b5234f366bb2a419953a1edfb590aadc32263
```

Subtrees

- Adding a copy from a remote as a subtree
- cd myproject
- git subtree add --prefix subproject --squash subproj.git master
  - Use “prefix” option to specify path for subproject
  - Use “squash” option to compress history from remote before adding it
  - branch (master) is optional
- directory listing of myproject now shows subproj as subdirectory

```
~/subtrees/local/myproject$ ls
file1.txt   file2.txt   file3.txt   subproject/
~/subtrees/local/myproject$ ls subproject
subfile1.txt  subfile2.txt
```

- Looking in the logs of the subproject will show the squashed history
- To get the latest, use pull
- git subtree pull --prefix subproject --squash subproj.git master




```
$ git log --oneline
6b109f0 Merge commit 'f7c3147d6df0609745228cc5083bb6c7d0b07d1a' as 'subproject'
f7c3147 Squashed 'subproject/' content from commit 906b523
fada8bb Add file3
ef21780 Add file2
73e59ba Add file1
dev@defaults-MacBook-Pro:~/subtrees/local/myproject$ git log -2
commit 6b109f0d5540642218d442297569b498f8e12396
Merge: fada8bb f7c3147
Author: Brent Laster <bl2@nclasters.org>
Date:   Tue Aug 2 21:15:06 2016 -0400

    Merge commit 'f7c3147d6df0609745228cc5083bb6c7d0b07d1a' as 'subproject'

commit f7c3147d6df0609745228cc5083bb6c7d0b07d1a
Author: Brent Laster <bl2@nclasters.org>
Date:   Tue Aug 2 21:15:06 2016 -0400

    Squashed 'subproject/' content from commit 906b523

    git-subtree-dir: subproject
    git-subtree-split: 906b5234f366bb2a419953a1edfb590aadc32263
```

Subtrees - split

- split subcommand can be used to extract a subproject's content into a separate branch
- extracts the content and history related to <prefix> and puts the resulting content at the root of the new branch instead of in a subdirectory

```
~/subtrees/local/myproject$ git subtree split --prefix=subproject \
--branch=split_branch
Created branch 'split_branch'
906b5234f366bb2a419953a1edfb590aadc32263
```

- As output, Git prints out the SHA1 value for the HEAD of the newly created tree
  - Provides a reference to work with for that HEAD if needed
  - New branch shows only the set of content from the subproject that was split out (as opposed to content from the superproject).

```
~/subtrees/local/myproject$ git checkout split_branch
Switched to branch 'split_branch'
~/subtrees/local/myproject$ ls
subfile1.txt  subfile2.txt
~/subtrees/local/myproject$ git log --oneline
906b523 Add subfile2
5f7a7db Add subfile1
```

Subtree - create new project from split content

- Since can split out content from a subtree, may want to transfer that split content into another project
- Very simple with Git
  - create new, empty project

```
~/subtrees/local/myproject$ cd ~/
~$ mkdir newproj
~$ cd newproj
~/newproj$ git init
Initialized empty Git repository in /Users/dev/newproj/.git/
```

  - pull contents of new branch into the new project (repository)

```
~/newproj$ git pull ~/subtrees/local/myproject split_branch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From /Users/dev/subtrees/local/myproject
 * branch            split_branch -> FETCH_HEAD
```

Subtree - push

- subtree command also supports a push subcommand
- This command does a split followed by an attempt to push the split content over to the remote
- Example: the following command splits out the subproject directory and then pushes it to the sub_origin remote reference and into a new branch named *new branch*:

```
~/subtrees/local/myproject$ git subtree push --prefix=subproject sub_origin new_branch
git push using:  sub_origin new_branch
Total 0 (delta 0), reused 0 (delta 0)
To /Users/dev/subtrees/remotes/subproj.git
 * [new branch]      906b5234f366bb2a419953a1edfb590aadc32263 -> new_branch
~/subtrees/local/myproject$
```

Command: (Interactive) Rebase

- Purpose - allows you to modify commits in the git history
- Use case - you need to make some kind of modification to one or more commits in the repository (rewrite history)
- Syntax

```
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
    [<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
    --root [<branch>]
```

- Notes - creates an interactive script/batch file to modify commits
- Cautions - don’t use this on anything already pushed

Interactive Rebase

Interactive Rebase

Git Bash Shell for Windows: /irebase_demo

```
$ git log
commit a1eb3156334f91aa81737ab4c30d6a7eecdf8ae2
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 15:03:56 2017 -0500

    change 4

commit b20b227d08ed1cba144c2c073c3edd97a2724a84
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:51:04 2017 -0500

    change 3

commit d5d8790bc88a15830612282acf8ffaf80b8b1b51
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:50:50 2017 -0500

    change 2

commit dc6eaef93b386c6e05904165a78bfe07f451dd8f
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:50:33 2017 -0500

    change 1
```

Interactive Rebase

- Choose set of commits

Git Bash Shell for Windows: /irebase_demo

```
$ git log
commit a1eb3156334f91aa81737ab4c30d6a7eecdf8ae2
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 15:03:56 2017 -0500

    change 4

commit b20b227d08ed1cba144c2c073c3edd97a2724a84
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:51:04 2017 -0500

    change 3

commit d5d8790bc88a15830612282acf8ffaf80b8b1b51
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:50:50 2017 -0500

    change 2

commit dc6eaef93b386c6e05904165a78bfe07f451dd8f
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:50:33 2017 -0500

    change 1
```

Interactive Rebase

- Choose set of commits

Git Bash Shell for Windows: /irebase_demo

```
bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master)
$ git log --oneline
a1eb315 change 4
b20b227 change 3
d5d8790 change 2
dc6eaef change 1

bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master)
$ git rebase -i HEAD~3
```

Interactive Rebase

- Choose set of commits
- Initiate interactive rebase

Git Bash Shell for Windows: /irebase_demo

```
bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master)
$ git log --oneline
a1eb315 change 4
b20b227 change 3
d5d8790 change 2
dc6eaef change 1

bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master)
$ git rebase -i HEAD~3
```

Interactive Rebase

- Choose set of commits
- Initiate interactive rebase

C:\Program Files\Git\irebase_demo\.git\rebase-merge\git-rebase-todo - Notepad++...

File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

```
pick d5d8790 change 2
pick b20b227 change 3
pick a1eb315 change 4

# Rebase dc6eaef..a1eb315 onto dc6eaef (3 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

length Ln : 1 Col : 1 Sel : 0 | 0 Unix (LF) UTF-8 INS

Interactive Rebase

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script

C:\Program Files\Git\irebase_demo\.git\rebase-merge\git-rebase-todo - Notepad++...

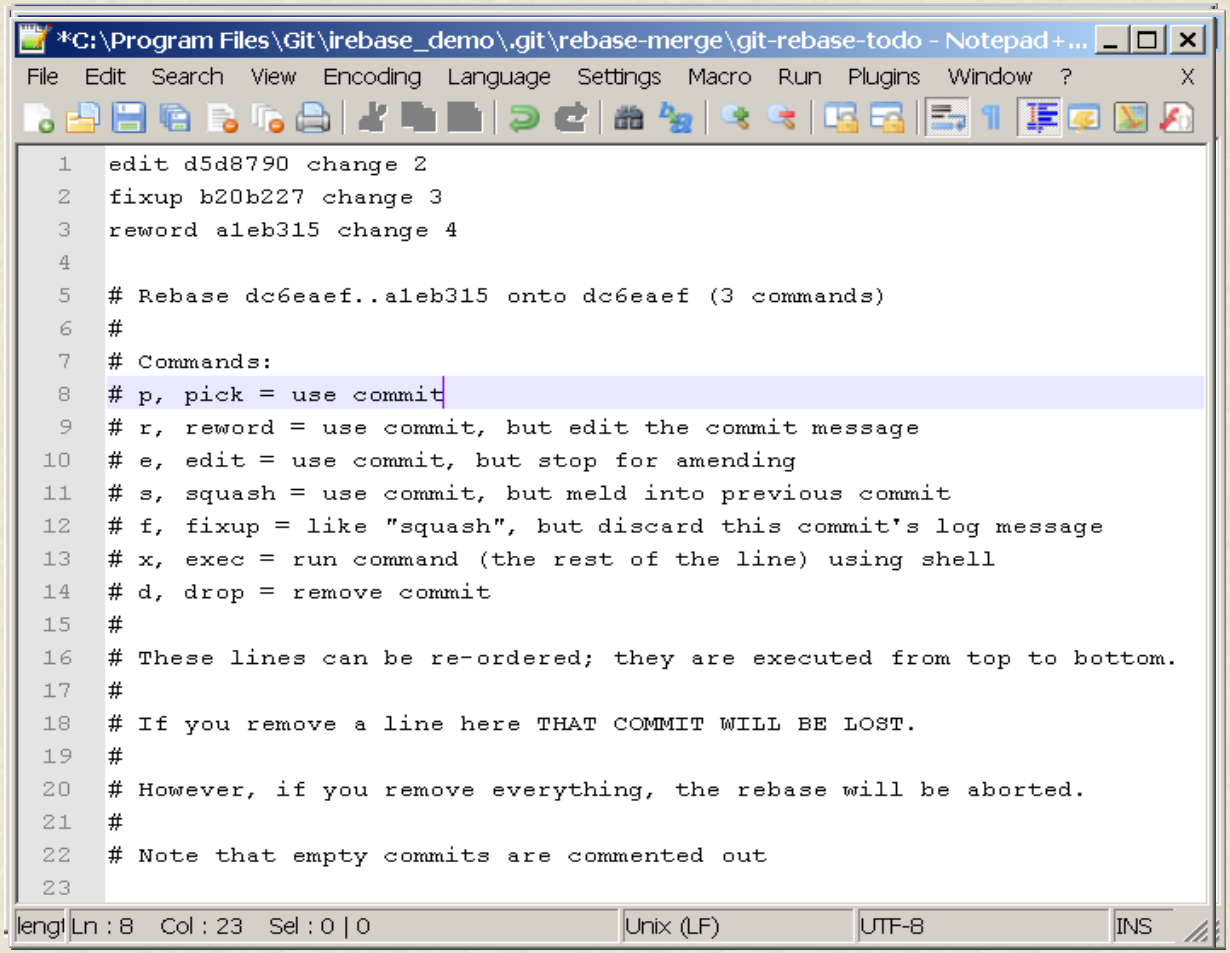
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

```
pick d5d8790 change 2
pick b20b227 change 3
pick a1eb315 change 4

# Rebase dc6eaef..a1eb315 onto dc6eaef (3 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

length Ln : 1 Col : 1 Sel : 0 | 0 Unix (LF) UTF-8 INS

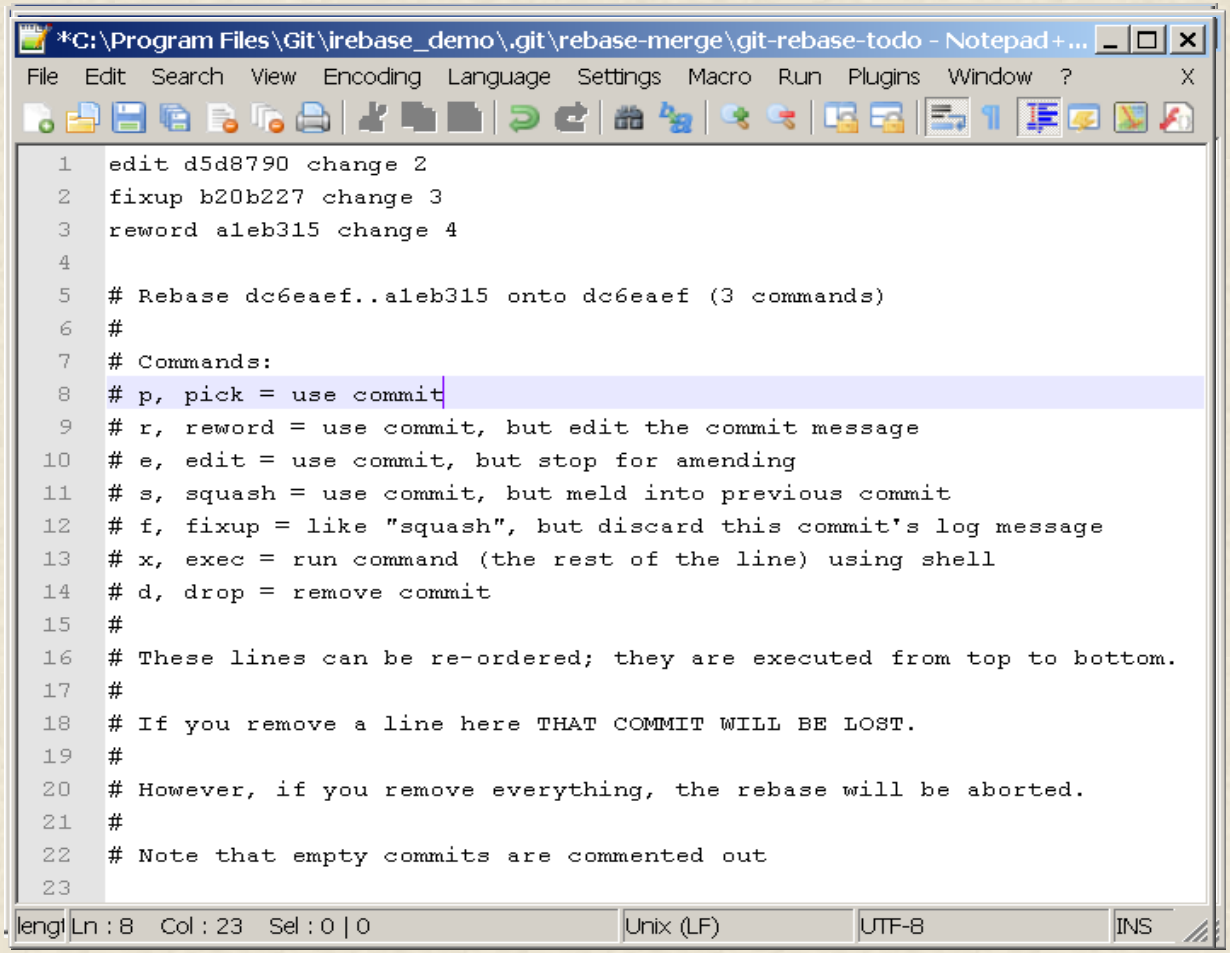
Interactive Rebase

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script



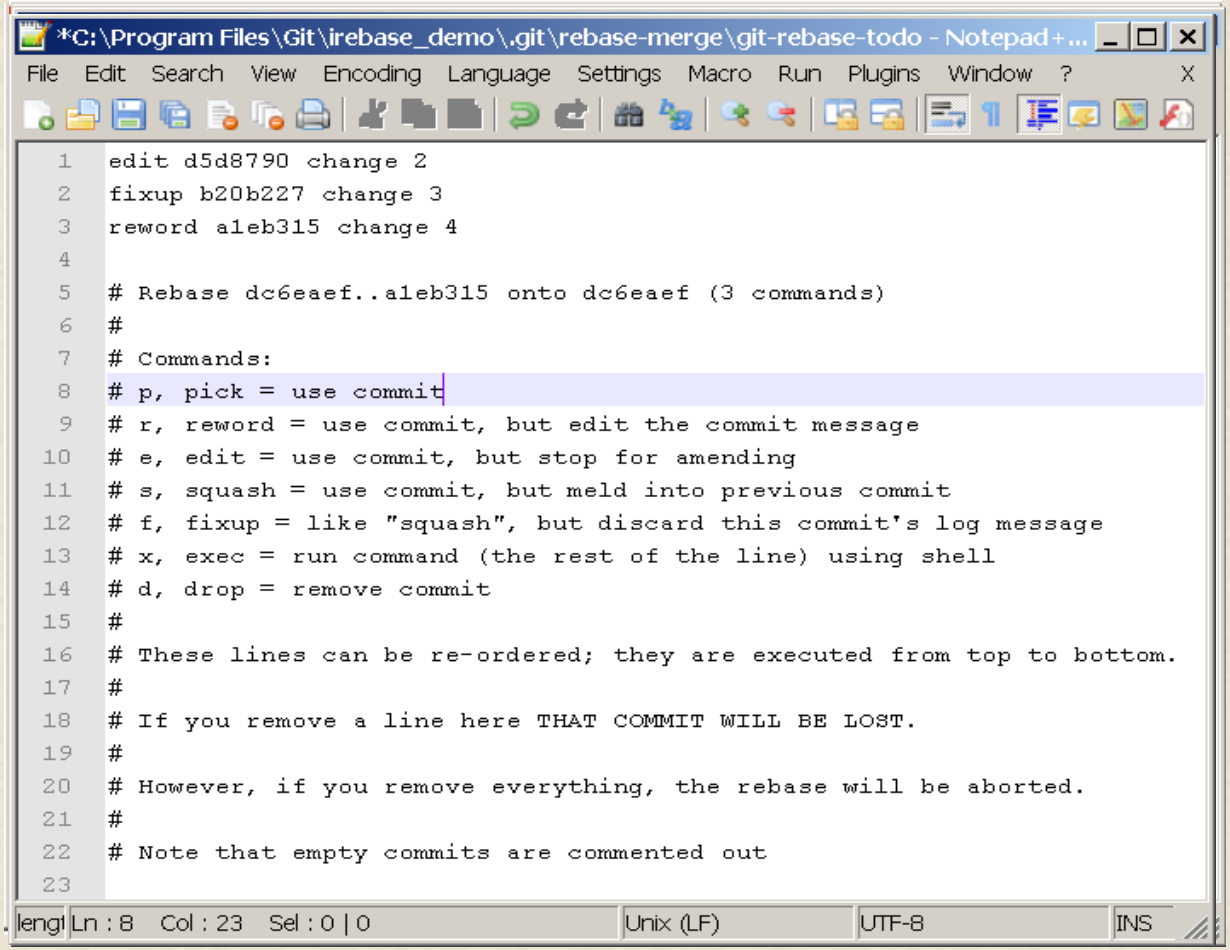
Interactive Rebase

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script
- Modify commands (save and exit)



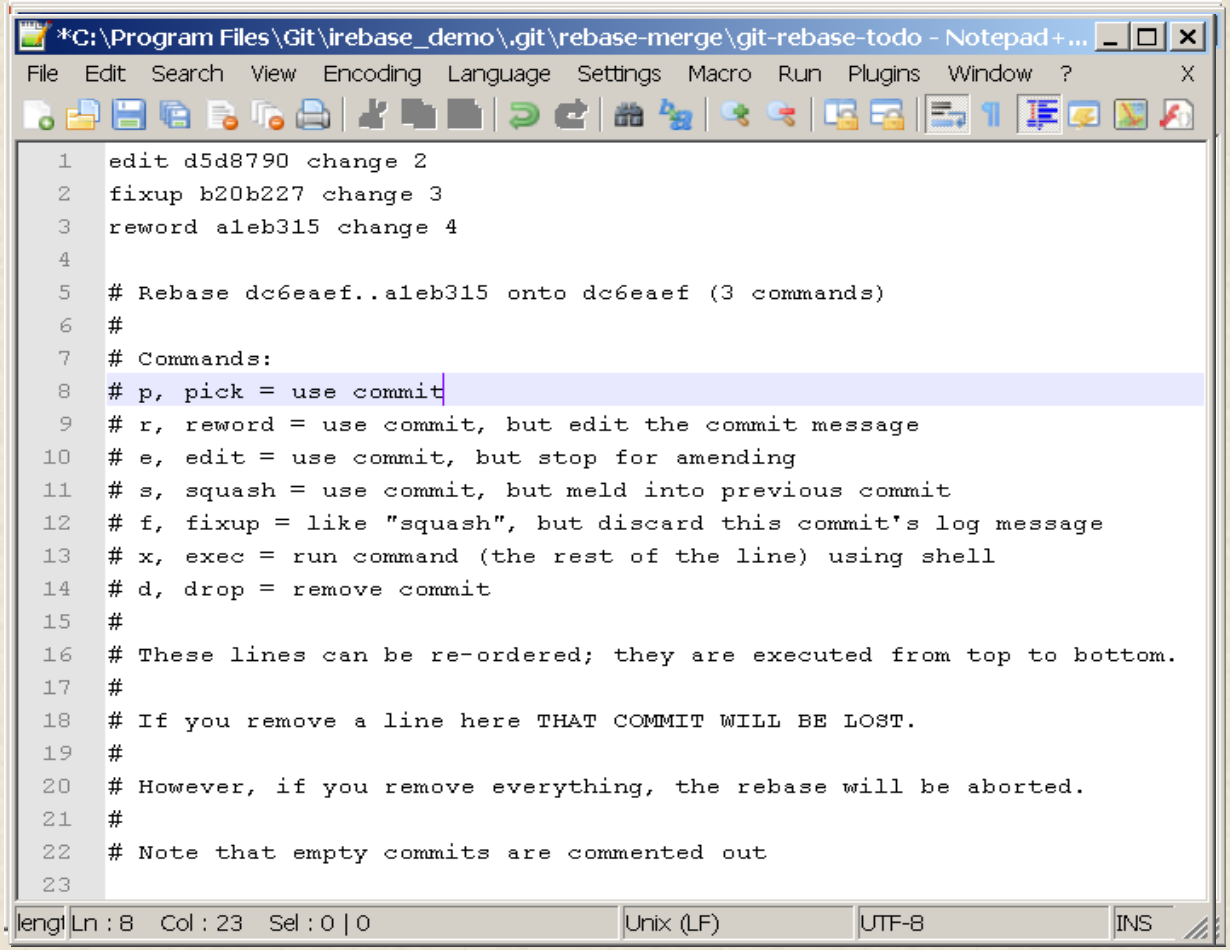
Interactive Rebase

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script
- Modify commands (save and exit)



Interactive Rebase

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script
- Modify commands (save and exit)
- Act on prompts



Interactive Rebase

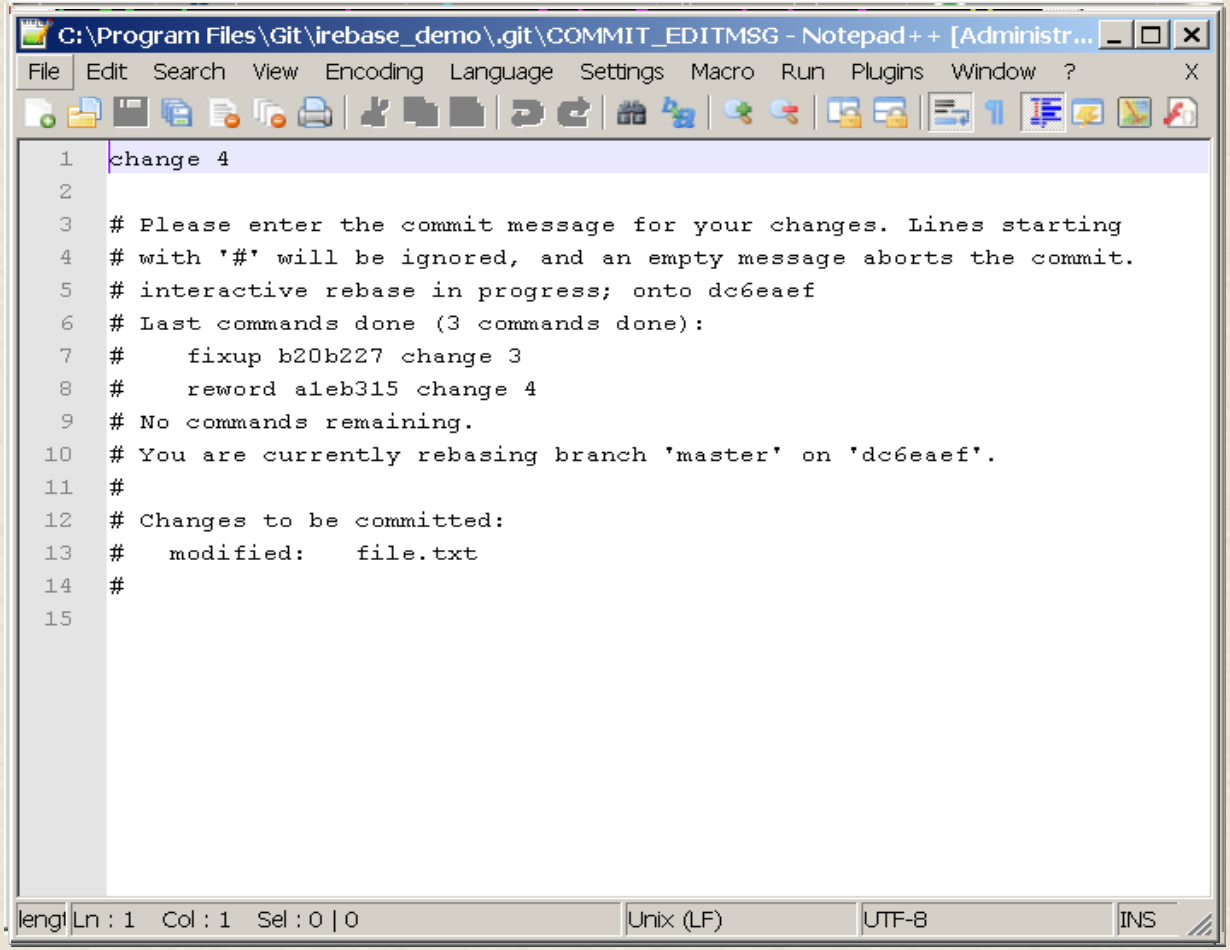
- Choose set of commits
- Initiate interactive rebase
- Git presents initial script
- Modify commands (save and exit)
- Act on prompts

Git Bash Shell for Windows: /irebase_demo

```
bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master|REBASE-i 1/3)
$ vi newfile.txt

bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master|REBASE-i 1/3)
$ git add newfile.txt

bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master|REBASE-i 1/3)
$ git commit --amend -m "adding file to change 2
"
[detached HEAD 7a3426c] adding file to change 2
 Date: Tue Jan 31 14:50:50 2017 -0500
 2 files changed, 2 insertions(+)
 create mode 100644 newfile.txt

bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master|REBASE-i 1/3)
$ git rebase --continue
```

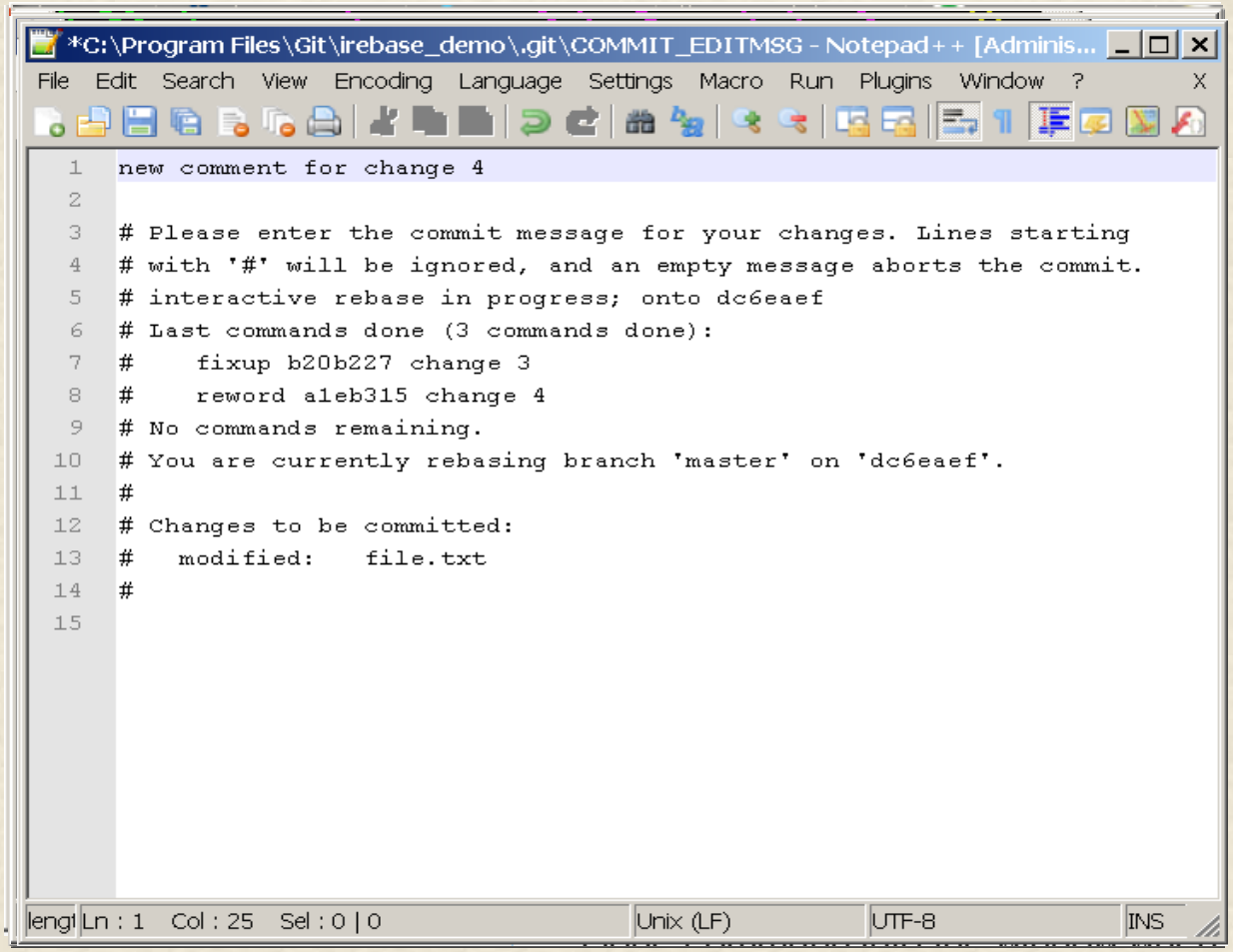
Interactive Rebase

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script
- Modify commands (save and exit)
- Act on prompts



Interactive Rebase

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script
- Modify commands (save and exit)
- Act on prompts



Interactive Rebase

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script
- Modify commands (save and exit)
- Act on prompts

Git Bash Shell for Windows: /irebase_demo

```
Successfully rebased and updated refs/heads/mast
er.

bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master)
$ git log
commit d624dcaa354ba3e74c371bcbe48b20fadd577d58
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 15:03:56 2017 -0500

    new comment for change 4

commit 2bb53263378fe60cb50e45cd33aac239b28bba4b
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:50:50 2017 -0500

    adding file to change 2

commit dc6eaef93b386c6e05904165a78bfe07f451dd8f
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:50:33 2017 -0500

    change 1
```

Interactive Rebase

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script
- Modify commands (save and exit)
- Act on prompts
- Commits are updated

Git Bash Shell for Windows: /irebase_demo

```
Successfully rebased and updated refs/heads/mast
er.

bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master)
$ git log
commit d624dcaa354ba3e74c371bcbe48b20fadd577d58
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 15:03:56 2017 -0500

    new comment for change 4

commit 2bb53263378fe60cb50e45cd33aac239b28bba4b
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:50:50 2017 -0500

    adding file to change 2

commit dc6eaef93b386c6e05904165a78bfe07f451dd8f
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:50:33 2017 -0500

    change 1
```

Interactive Rebase

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script
- Modify commands (save and exit)
- Act on prompts
- Commits are updated

Git Bash Shell for Windows: /irebase_demo

```
Date:   Tue Jan 31 14:50:50 2017 -0500

    adding file to change 2

commit dc6eaef93b386c6e05904165a78bfe07f451dd8f
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jan 31 14:50:33 2017 -0500

    change 1

bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master)
$

bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master)
$ git log --oneline
d624dca new comment for change 4
2bb5326 adding file to change 2
dc6eaef change 1

bcl@myhost Bash Prompt (Git for Windows) => /ire
base_demo (master)
$
```

Command: Notes

- Purpose - Add additional information to objects in the Git repository or look at such information
- Use case - At some point after making a commit, you may decide that there are additional comments or other non-code information that you'd like to add with the commit - *without changing the commit itself*.
- Syntax

git notes **[list [<object>]]**

git notes **add [-f] [--allow-empty] [-F <file> | -m <msg> | (-c | -C) <object>] [<object>]**

git notes **copy [-f] (--stdin | <from-object> <to-object>)**

git notes **append [--allow-empty] [-F <file> | -m <msg> | (-c | -C) <object>] [<object>]**

git notes **edit [--allow-empty] [<object>]**

git notes **show [<object>]**

git notes **merge [-v | -q] [-s <strategy>] <notes-ref>**

git notes **merge --commit [-v | -q]**

git notes **merge --abort [-v | -q]**

git notes **remove [--ignore-missing] [--stdin] [<object>…]**

git notes **prune [-n | -v]**

git notes **get-ref**

Notes

- Create a note

```
$ git notes add -m "This is an example of a note" 2f2ea1e
```

- Create a note in a custom namespace (add --ref)

```
$ git notes --ref=review add -m "Looks ok to me" f3b05f9
```

- View a note (for a specific revision)

```
$ git notes show 2f2ea1e
This is an example of a note
```

- List notes in log

```
$ git log --show-notes=*
commit 80e224b24e834aaa8915e3113ec4fc635b060771
Author: Brent Laster <bcl@nclasters.org>
Date:   Fri Jul 1 13:01:58 2016 -0400

commit ef15dca5c6577d077e38a05b80670024e1d92c0a
Author: unknown <bcl@nclasters.org>
Date:   Fri Apr 24 12:32:50 2015 -0400

    Removing test subdir on master

commit f3b05f9c807e197496ed5d7cd25bb6f3003e8d35
Author: Brent Laster <bcl@nclasters.org>
Date:   Sat Apr 11 19:56:39 2015 -0400

    update test case

Notes (review):
    Looks ok to me

commit 2f2ea1e30fe4630629477338a0ab8618569f0f5e
Author: Brent Laster <bcl@nclasters.org>
Date:   Sat Apr 11 17:34:57 2015 -0400

    Add in testing example files

Notes:
    This is an example of a note
```

Command: grep

- Purpose - provides a convenient (and probably familiar) way to search for regular expressions in your local Git environment.
- Use case - self-explanatory
- Syntax

```
git grep [-a | --text] [-I] [--textconv] [-i | --ignore-case] [-w | --word-regexp]
        [-v | --invert-match] [-h|-H] [--full-name]
        [-E | --extended-regexp] [-G | --basic-regexp]
        [-P | --perl-regexp]
        [-F | --fixed-strings] [-n | --line-number]
        [-l | --files-with-matches] [-L | --files-without-match]
        [(-O | --open-files-in-pager) [<pager>]]
        [-z | --null]
        [-c | --count] [--all-match] [-q | --quiet]
        [--max-depth <depth>]          [--color[=<when>] | --no-color]
        [--break] [--heading] [-p | --show-function]
        [-A <post-context>] [-B <pre-context>] [-C <context>]
        [-W | --function-context]
        [--threads <num>]
        [-f <file>] [-e] <pattern>
        [--and|--or|--not|(|)|-e <pattern>…]
        [ [--[no-]exclude-standard] [--cached | --no-index | --untracked] | <tree>…]
        [--] [<pathspec>…]
```

- Notes
  - Several options are similar to OS grep options

grep

- Default behavior - search for all instances of an expression across all tracked files in working directory
- Search for all instances off expression “database” across all java files (note use of --)

```
$ git grep database -- *.java
api/src/main/java/com/demo/pipeline/status/status.java:      @Path("/database")
dataaccess/src/main/java/com/demo/dao/MyDataSource.java:
logger.log(Level.SEVERE, "Could not access database via connect string
jdbc:mysql://"+strMySQLHost+":"+strMySQLPort+"/"+strMySQLDatabase,e);
```

- -p option tells Git to try and show header of method or function where search target was found
- --break - make output easier to read
- --heading - prints filename above output

```
$ git grep -p --break --heading database -- *.java

api/src/main/java/com/demo/pipeline/status/status.java
13=public class V1_status {
31:      @Path("/database")

dataaccess/src/main/java/com/demo/dao/MyDataSource.java
18=public class MyDataSource {
64:                 logger.log(Level.SEVERE, "Could not access database via
connect string
jdbc:mysql://"+strMySQLHost+":"+strMySQLPort+"/"+strMySQLDatabase,e);
```

- boolean operators

```
$ git grep -e 'database' --and -e 'access' -- *.java
```

- search in staging area

```
$ git grep -e 'config' --cached -- '*.txt'
```

- search in specific commit(s)

```
$ git grep -e 'database' HEAD  -- *.java
$ git grep -e 'database' b2e575a  -- *.java
```

That’s all - thanks!