Introduction to GIT version control system

About Version Control Tools

What is a version control system?

- Revision control is the management of changes to documents, computer programs, large web sites, and other collections of information.
- Changes are usually identified by a number or letter code, termed the ”revision number”. For example, an initial set of files is ”revision 1”. When the first change is made, the resulting set is ”revision 2”, and so on.
- Each revision is associated with a timestamp and the person making the change.
- Revisions can be compared, restored, and with some types of files, merged.

About Version Control Tools

Use case 1: keeping an history

- The life of your software/article is recorded from the beginning
  - At any moment you can revert to a previous revision (let's say you are not happy with your latest changes)
  - The history is browseable. You can inspect any revision (this is useful for understanding and fixing bugs)
    - when was it done ?
    - who wrote it ?
    - what was change ?
    - why ?
    - in which context ?
- All the deleted content remains accessible in the history

About Version Control Tools

Use case 2: working with others

- The VC tools help you to:
  - share a collection of files with your team
  - merge changes done by other users
  - ensure that nothing is accidentally overwritten
  - know who you must blame when something is broken

About Version Control Tools

Use case 3: branching

- You may have multiple variants of the same software, materialized as **branches**, for example:
  - a main branch
  - a maintenance branch (to provide bug fixes in older releases)
  - a development branch (to make disruptive changes)
  - a release branch (to freeze code before a new release)

- VC tools will help you to:
  - handle multiple branches concurrently
  - merge changes from a branch into another one

About Version Control Tools

Use case 4: working with external contributors

- VC tools help working with third-party contributors:
  - It gives them visibility of what is happening in the project
  - It helps them to submit changes (patches) and

    it helps you to integrate these patches
  - Forking the development of a software and merging it back into mainline

About Version Control Tools

Use case 5: scaling

- Some metrics about the Linux kernel (developed with Git):
  - about 10000 changesets in each new version (every 2 or 3 months)
  - 1000+ unique contributors

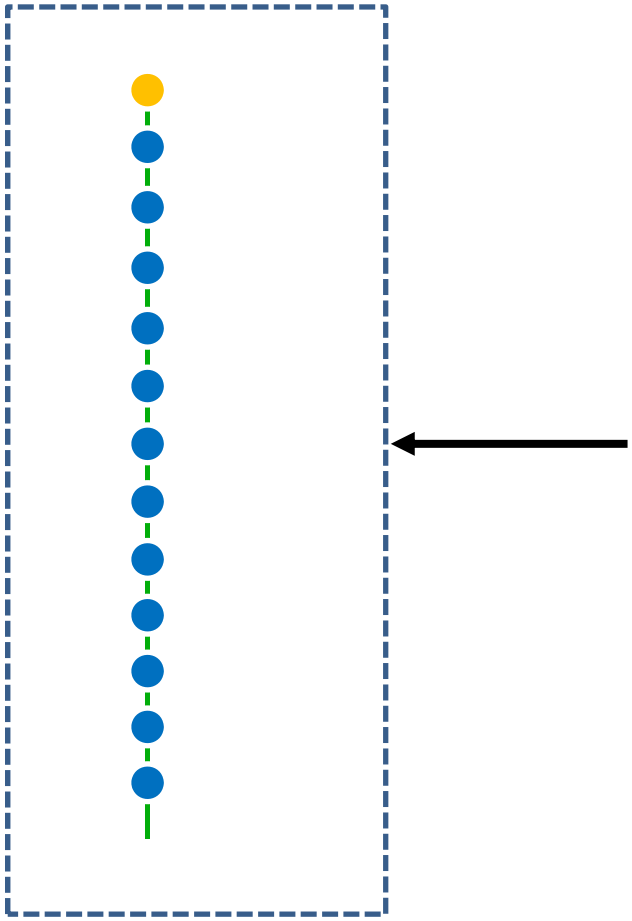
Version Control Tools

git

- You can imagine **git** as something that sits on top of your file system and manipulates files.
- This “something” is a **tree** structure where each **commit** creates a new node in that tree.
- Nearly all **git** commands actually serve to navigate on this tree and to manipulate it accordingly.
- The purpose of **git** is to manage a project, or a set of files, as they change over time. It stores this information in a data structure called a **repository**.

Version Control Tools

Some illustrations

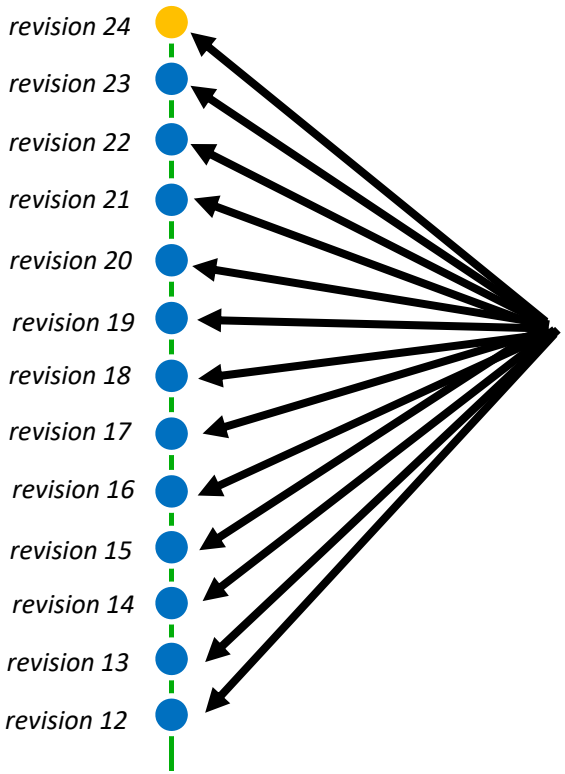


The repository

It contains the full history of your project (all revisions from the beginning)

Version Control Tools

Some illustrations



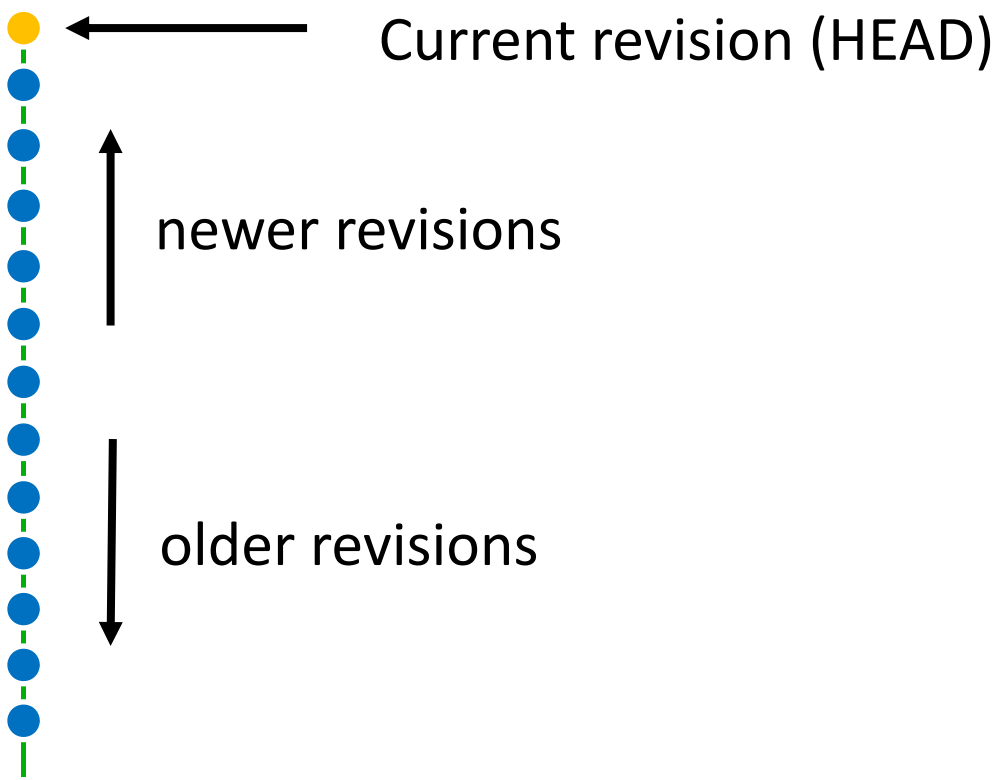
Revisions

Each revision:

- introduces changes from previous revision
- has an identified author
- contains a textual message describing the changes

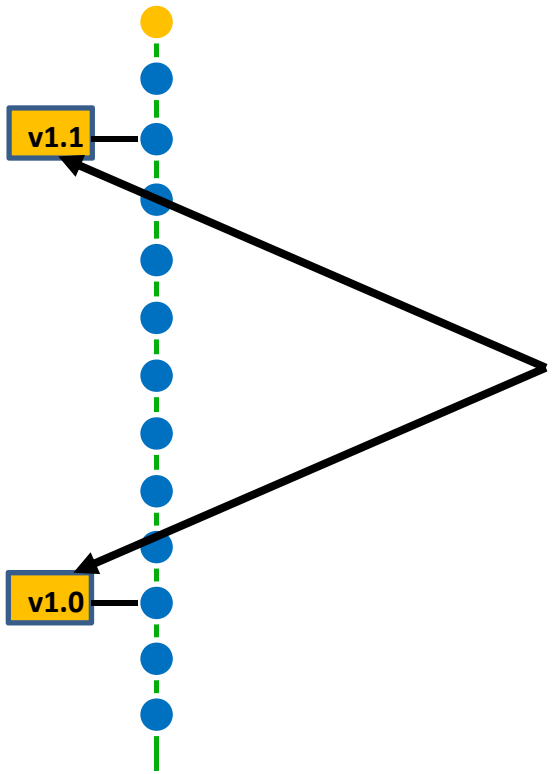
Version Control Tools

Some illustrations



Version Control Tools

Some illustrations

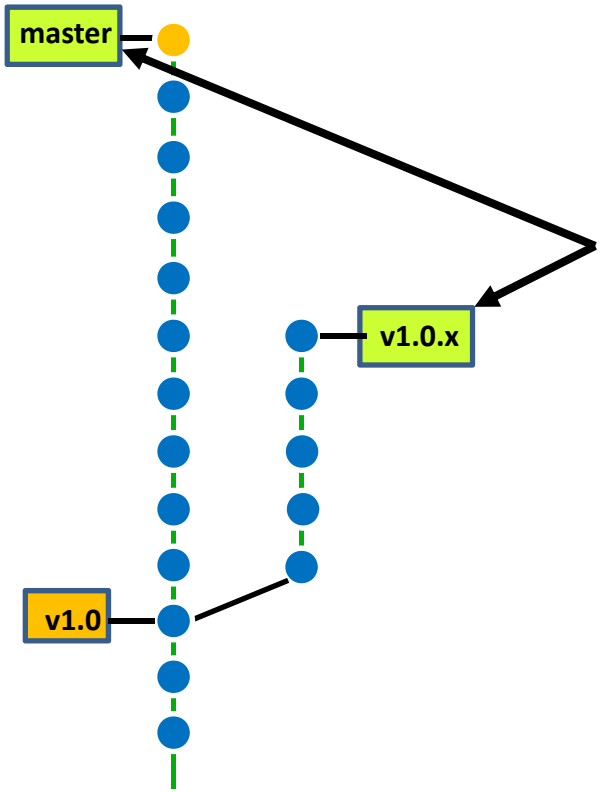


Tags

a tag identifies a particular revision
(typically each release of the software)

Version Control Tools

Some illustrations

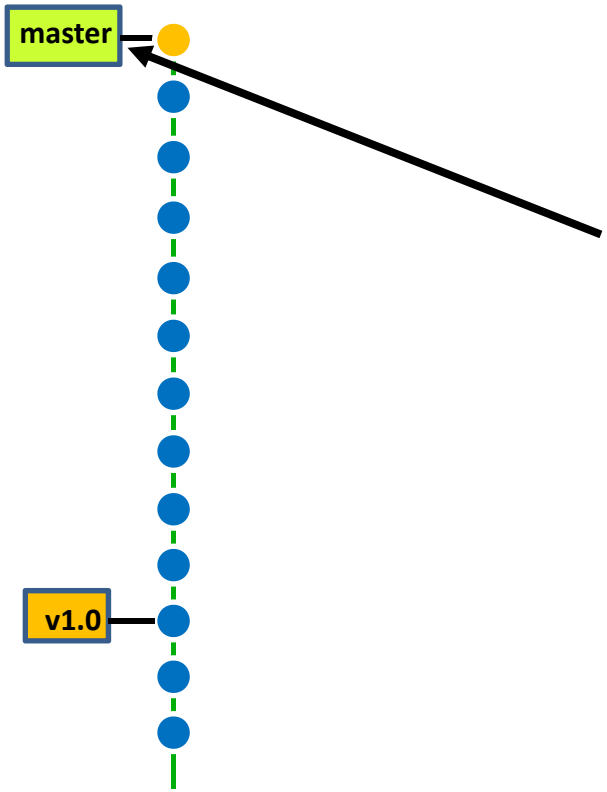


Branches

- branches are different variants of the same collection of files
- they evolve independently of each other

Version Control Tools

Some illustrations

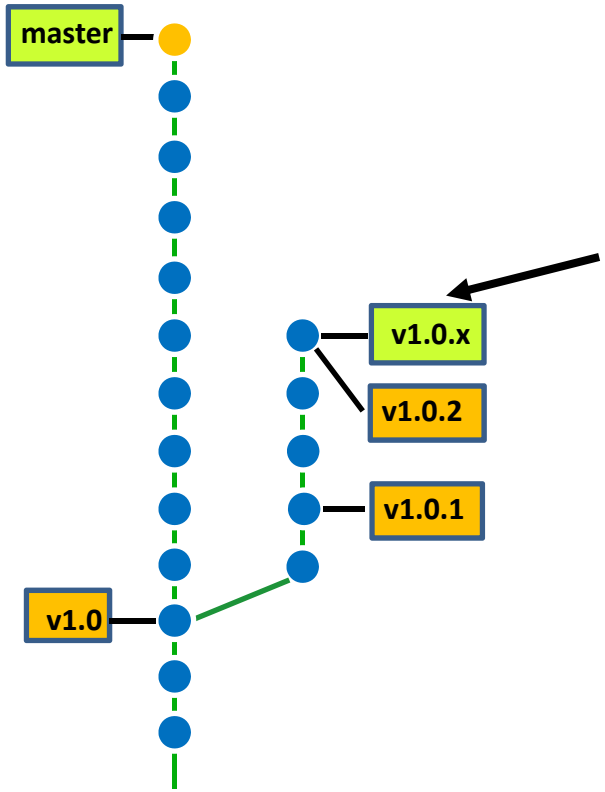


Main branch

where the everyday development happens

Version Control Tools

Some illustrations

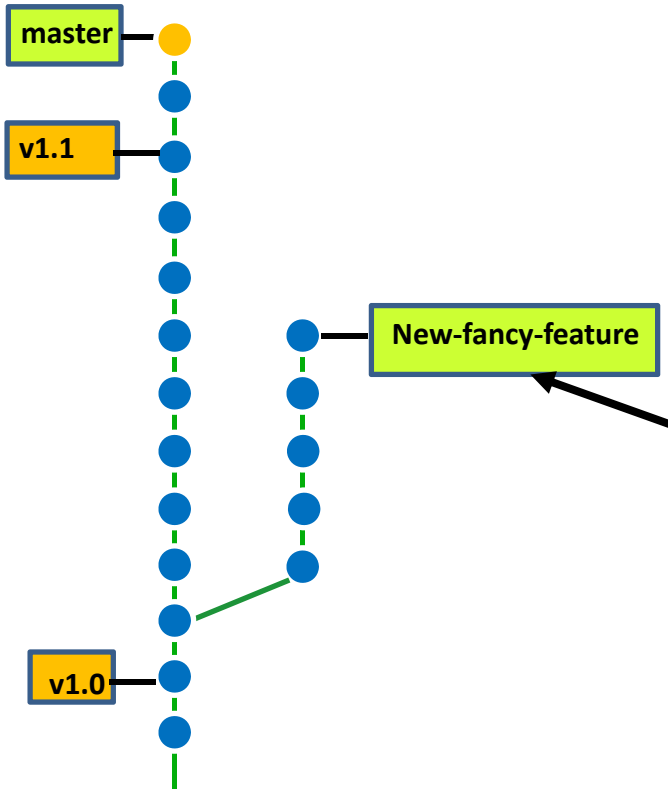


Maintenance branch

to issue bug fixes for older releases of the software

Version Control Tools

Some illustrations

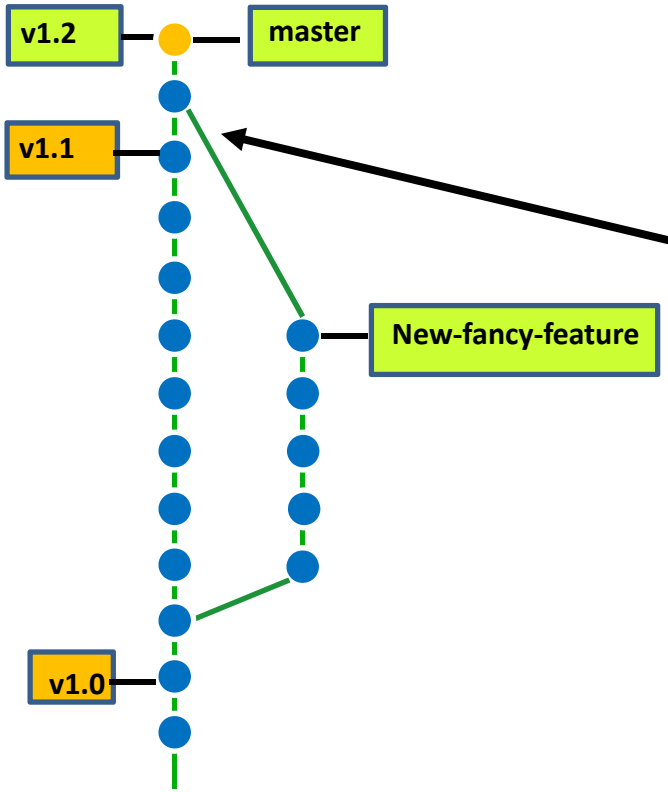


Feature branch

- for a new feature requiring intrusive changes in the code
- normal development continues to happen in the master branch (without disturbance)

Version Control Tools

Some illustrations



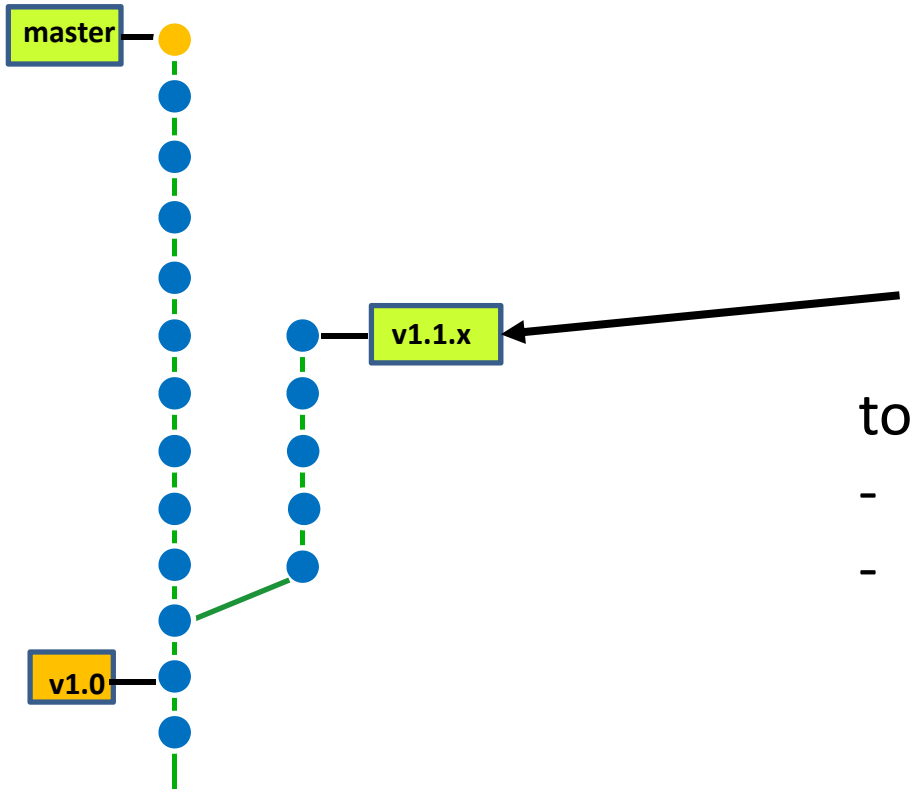
Merging

When the new feature is ready, it can merged back into the master branch

-> all changes done in the feature branch are imported

Version Control Tools

Some illustrations



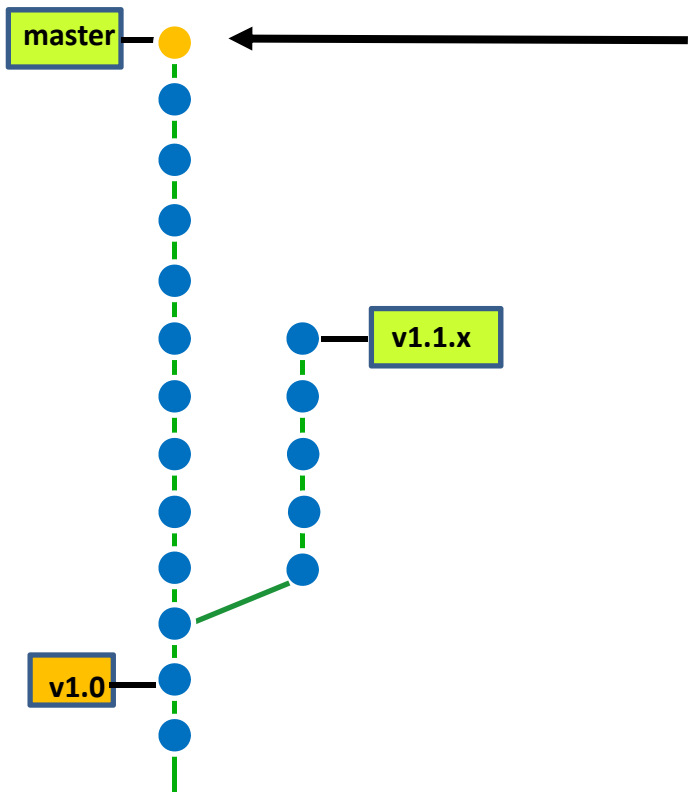
Release branch

to prepare the next release

- the code is frozen
- only bug fixes are accepted

Version Control Tools

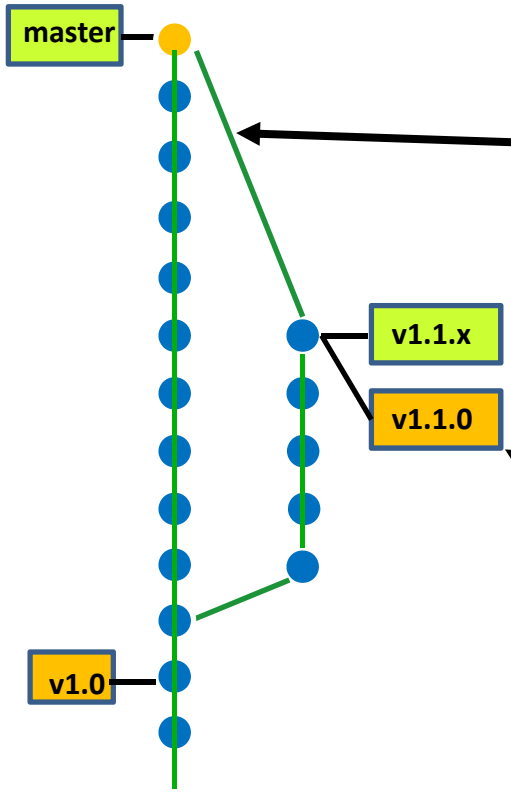
Some illustrations



meanwhile developments continue in the master branch

Version Control Tools

Some illustrations



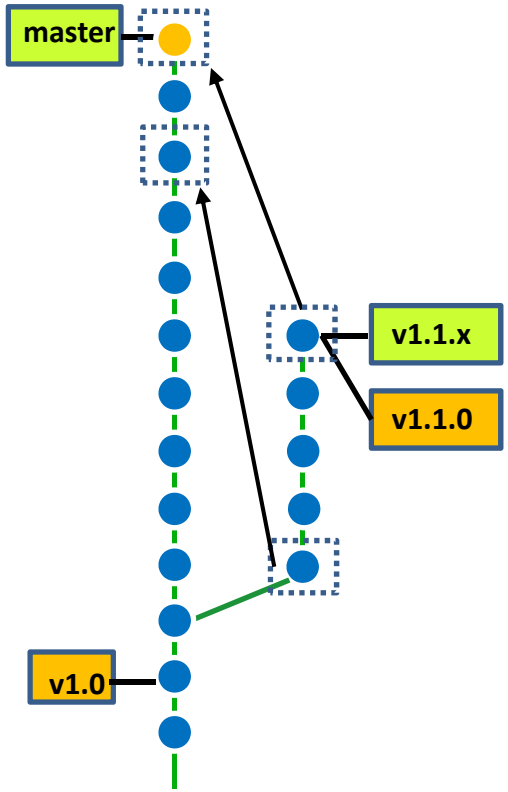
When the code is ready, the new version is released

- the release branch becomes a maintenance branch
- bug fixes can be merged back into the main branch

New release

Version Control Tools

Some illustrations



Cherry picking

It may not be desirable to merge all the commits into the other branch (e.g. a bug may need a different fix)

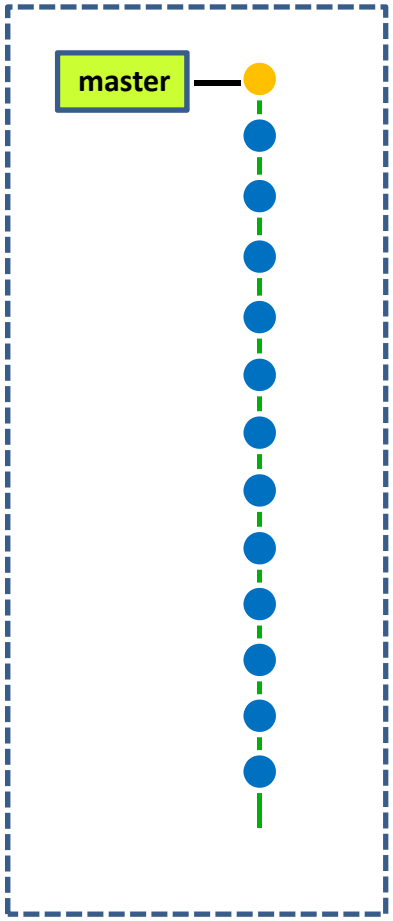
-> It is possible to apply each commit individually

Commit

- A commit object mainly contains three things:
  - A set of changes the commit introduces
  - Commit message describing the changes
  - A hash, a 40-character string that uniquely identifies the commit object

Version Control Tools

Creating new revisions

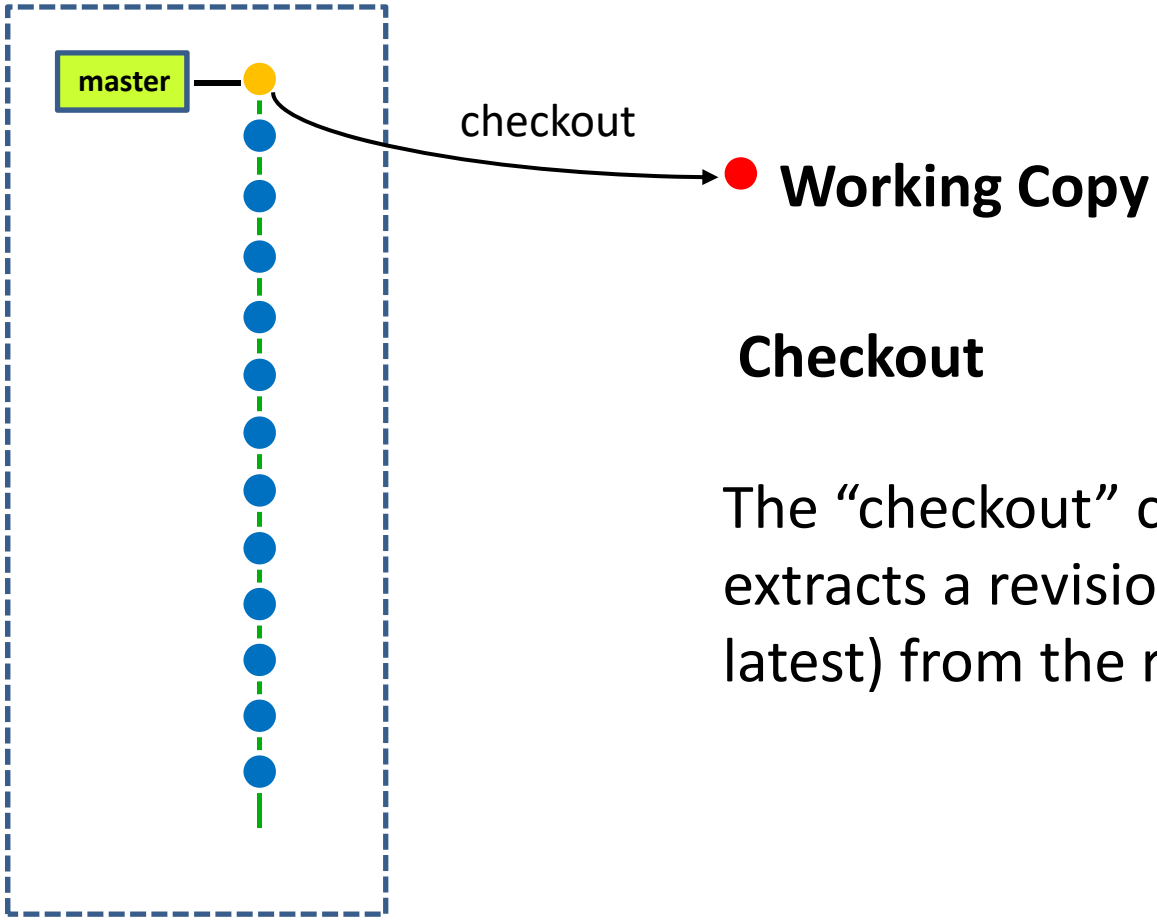


A repository is an opaque entity, it cannot be edited directly

We will first need to extract a local copy of the files

Version Control Tools

Creating new revisions

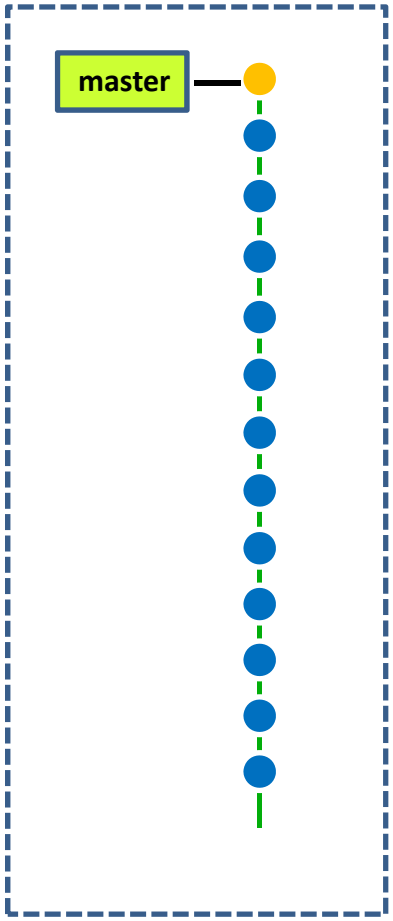


Checkout

The “checkout” command extracts a revision (usually the latest) from the repository

Version Control Tools

Creating new revisions



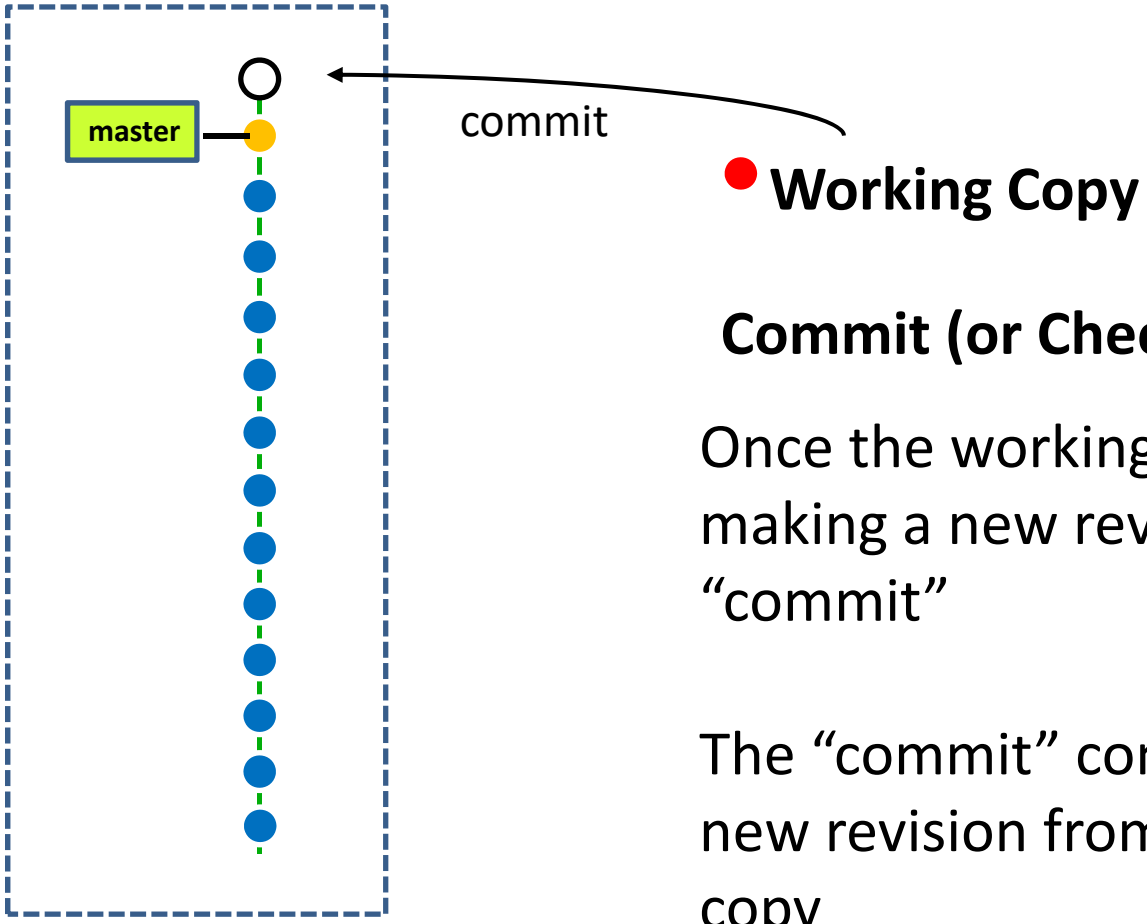
Edition

The working copy is hosted in the local filesystem

It can be edited with any editor,
It can be compiled,…

Version Control Tools

Creating new revisions



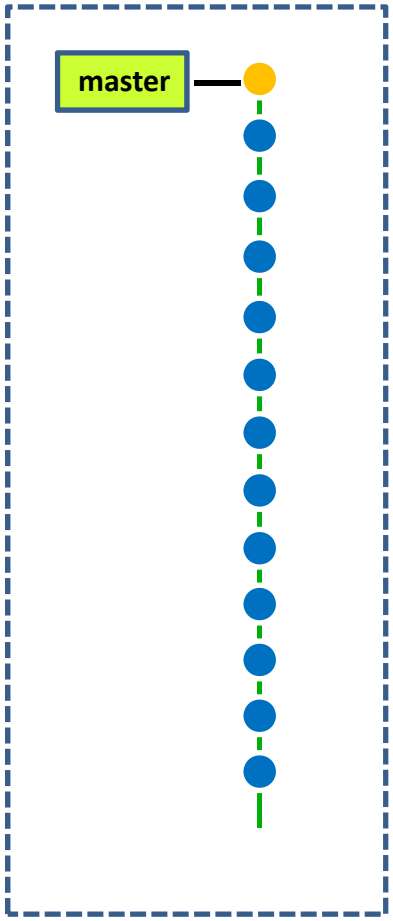
Commit (or Checkin)

Once the working copy is ready for making a new revision, we do a "commit"

The "commit" command creates a new revision from the current working copy

Version Control Tools

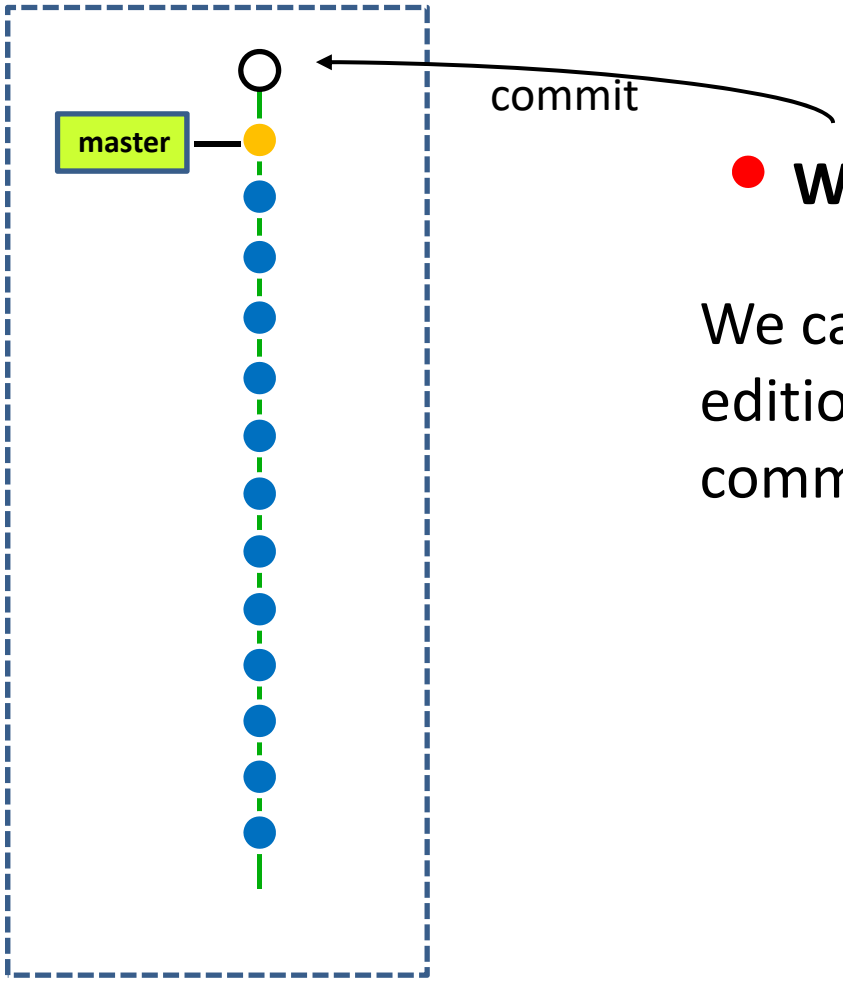
Creating new revisions



We can do further editions and further commits…

Version Control Tools

Create new revisions



- **Working Copy**

We can do further editions and further commits…

Version Control Tools

What shall be stored into the repository ?

- You should store all files that are not generated by a tool:
  - source files (.c .cpp .java .y .l .tex . . .)
  - build scripts / project files (Makefile configure.in Makefile.am CMakefile.txt wscript .sln)
  - documentation files (.txt README . . .)
  - resource files (images, audio, . . .)
- You should not shore generated files

  (*or you will experience many unnecessary conflicts*)
  - .o .a .so .dll .class .jar .exe .dvi .ps .pdf
  - source files / build scripts when generated by a tool (like autoconf, cmake, lex, yacc)

Version Control Tools

Guidelines for committing

- Commit often
- Commit independent changes in separate revisions
- In commit messages, describe the rationale behind of your changes (it is often more important than the change itself)

Working locally

Create a new repository

```
git init myrepository
```

This command creates the directory *myrepository*

- the repository is located in *myrepository/.git*
- the (initially empty) working copy is located in *myrepository/*

```
sepc92.se.cuhk.edu.hk - PuTTY
sepc92:/info_filter/wlam[49] > pwd
/info_filter/wlam
sepc92:/info_filter/wlam[50] > git init helloproj
Initialized empty Git repository in /info_filter/wlam/helloproj/.git/
sepc92:/info_filter/wlam[51] > ls -a helloproj/
.  ..  .git
sepc92:/info_filter/wlam[52] > ls helloproj/.git
branches  config  description  HEAD  hooks  info  objects  refs
sepc92:/info_filter/wlam[53] >
```

Note: The /.git/ directory contains your whole history, **do not delete it**

Working locally

Commit your first files

git add *file*
git commit [-m *message*]

sepc92.se.cuhk.edu.hk - PuTTY

```
sepc92:/info_filter/wlam[87] > cd helloproj/
sepc92:/info_filter/wlam/helloproj[88] > echo "code line 1" > hello.c
sepc92:/info_filter/wlam/helloproj[89] > git add hello.c
sepc92:/info_filter/wlam/helloproj[90] > git commit -m "Bob added program 'hello
.c'"
[master (root-commit) e82a15f] Bob added program 'hello.c'
 Committer: Wai Lam <wlam@sepc92.se.cuhk.edu.hk>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

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

    git commit --amend --reset-author

 1 file changed, 1 insertion(+)
 create mode 100644 hello.c
sepc92:/info_filter/wlam/helloproj[91] >
```

Note: "master" is the name of the default branch created by git init

Working locally

Commit your updated file

git commit -a [-m *message*]

```
sepc92:/sac/lec/wlam/helloproj[29] > nano hello.c
  GNU nano 2.3.1                File: hello.c

code line 1
code line 2
```

linux03 - default - SSH Secure Shell

File Edit View Window Help

```
sepc92:/sac/lec/wlam/helloproj[33] > git commit -a -m "v2.0"
[master b2c73c1] v2.0
 Committer: Wai Lam <wlam@sepc92.se.cuhk.edu.hk>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

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

    git commit --amend --reset-author

 1 file changed, 1 insertion(+)
sepc92:/sac/lec/wlam/helloproj[34] > git show
commit b2c73c1521f946d6fba0c9fd3208696c997eb7b9
Author: Wai Lam <wlam@sepc92.se.cuhk.edu.hk>
Date:   Wed Nov 24 18:08:29 2021 +0800

    v2.0

diff --git a/hello.c b/hello.c
index f62eeae..8349d42 100644
--- a/hello.c
+++ b/hello.c
@@ -1 +1,2 @@
 code line 1
+code line 2
```

Connected to linux03 SSH2 - aes128-cbc - hmac-sha1 - n 91x29

Working locally

branch - master

"master" is the name of the default branch created by git init

```
git  branch
```

This command can list the current branch(es)

```
git  show
```

This command can show various types of objects

sepc92.se.cuhk.edu.hk - PuTTY

```
sepc92:/info_filter/wlam/helloproj[29] > git branch
* master
sepc92:/info_filter/wlam/helloproj[30] > git show
commit 31631d43ec2225959719585799149e4c571a1fe2
Author: Wai Lam <wlam@sepc92.se.cuhk.edu.hk>
Date:   Sun Nov 15 14:09:16 2020 +0800

    Bob added program 'hello.c'

diff --git a/hello.c b/hello.c
new file mode 100644
index 0000000..f62eeae
--- /dev/null
+++ b/hello.c
@@ -0,0 +1 @@
+code line 1
sepc92:/info_filter/wlam/helloproj[31] >
```

Working locally

The staging area (aka the “index”)

- Usual version control systems provide two spaces
  - the **repository**

    (the whole history of your project)
  - the **working tree** (or local copy)

    (the files are editing and that will be in the next commit)

  Git introduces an intermediate space: the **staging area** (also called **index**)
- The index stores the files scheduled for the next commit:
  - git add *files* -> copy files into the index
  - git commit -> commits the content of the index

Git Branching

Branches in a Nutshell

- Git doesn't store data as a series of changesets or differences
- Git stores data as a series of **snapshots**
- When you make a commit, Git stores a commit object that contains:
  - A pointer to the snapshot of the content you staged
  - The author's name
  - The author's email address
  - The message that you typed
  - Pointer(s) to the commit(s) that directly came before this commit (its parent or parents)
    - Zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches

Git Branching

Branches in a Nutshell

- Let's assume that you have a directory containing three files: README, test.rb, LICENSE
- You stage them all and commit

```
git add README test.rb LICENSE
git commit –m 'Initial commit'
```

- Staging:
  - A checksum is computed for each file (SHA-1 hash)
  - That version of the file is stored in the Git repository (blob)
  - The checksum is added to the staging area

Git Branching

Branches in a Nutshell

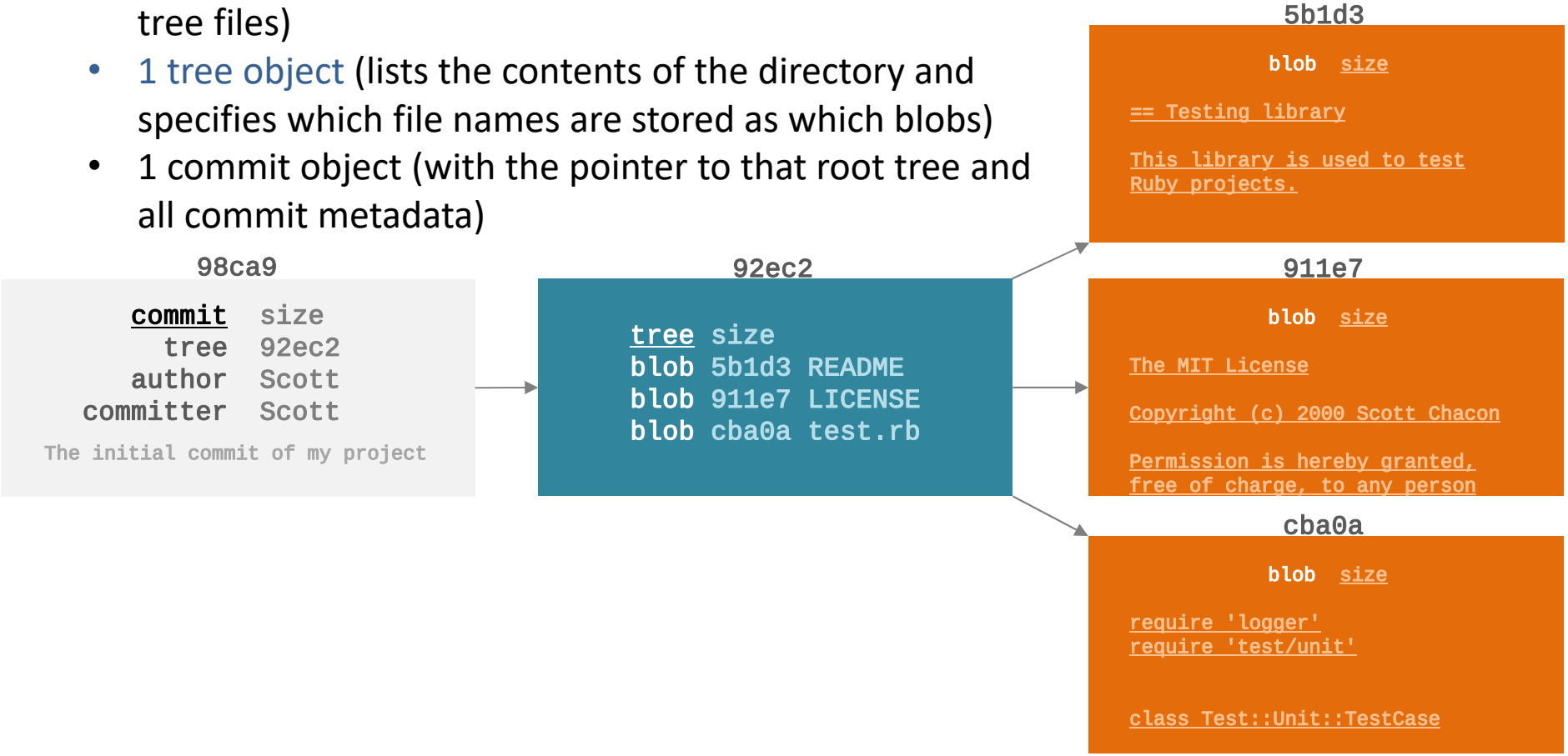
- Commit:
  - Git checksums each subdirectory (just the root directory in this case)
  - A tree object is created in the Git repository to store them
  - A commit object is created
    - Containing the metadata
    - A pointer to the root project tree (for re-creating that snapshot when needed)

Git Branching

Branches in a Nutshell

The Git repository now contains five object:

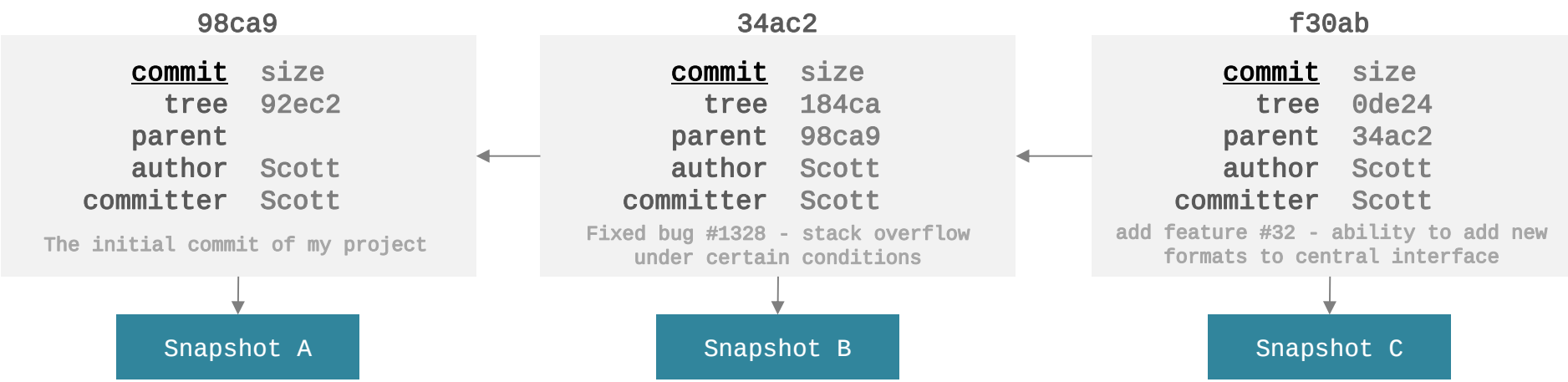
- 3 blobs (each representing the contents of one of the tree files)
- 1 tree object (lists the contents of the directory and specifies which file names are stored as which blobs)
- 1 commit object (with the pointer to that root tree and all commit metadata)



Git Branching

Branches in a Nutshell

- If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it
- A branch in Git is simply a lightweight movable pointer to one of these commits



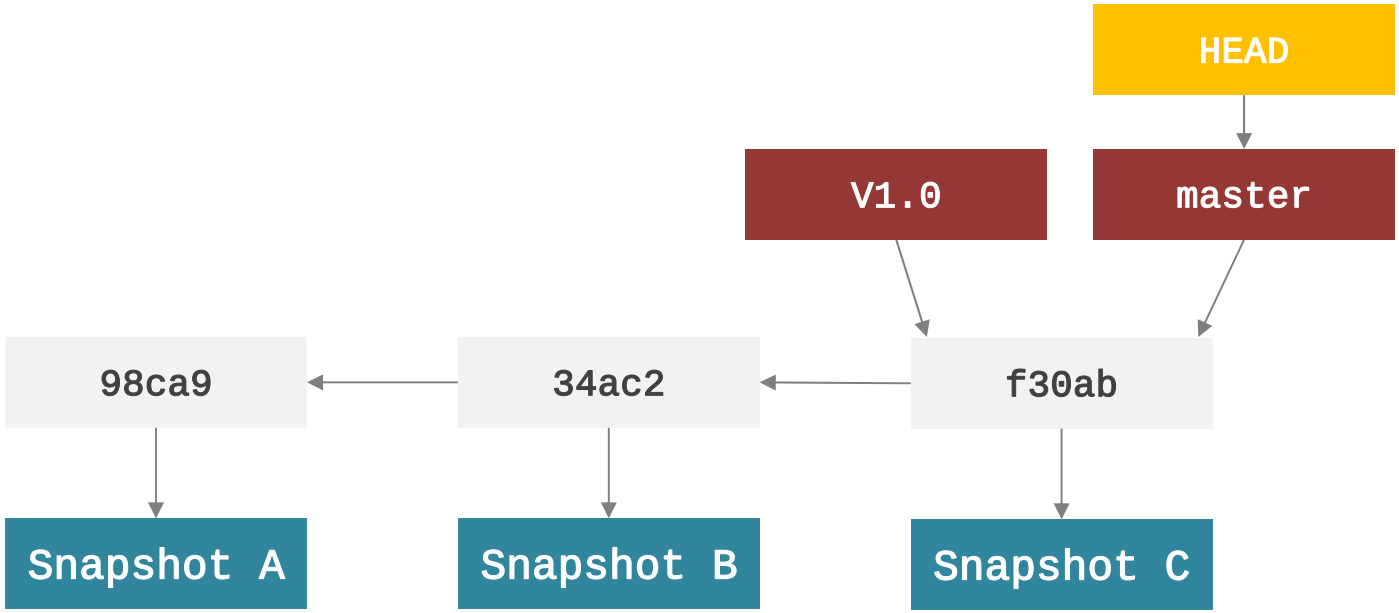
Git Branching

Branches in a Nutshell

- The default branch name in Git is master
- As you start making commits, you're given a master branch that points to the last commit you made
- Every time you commit, the master branch pointer moves forward automatically
- `git init` command creates the master branch by default

Git Branching

Branches in a Nutshell



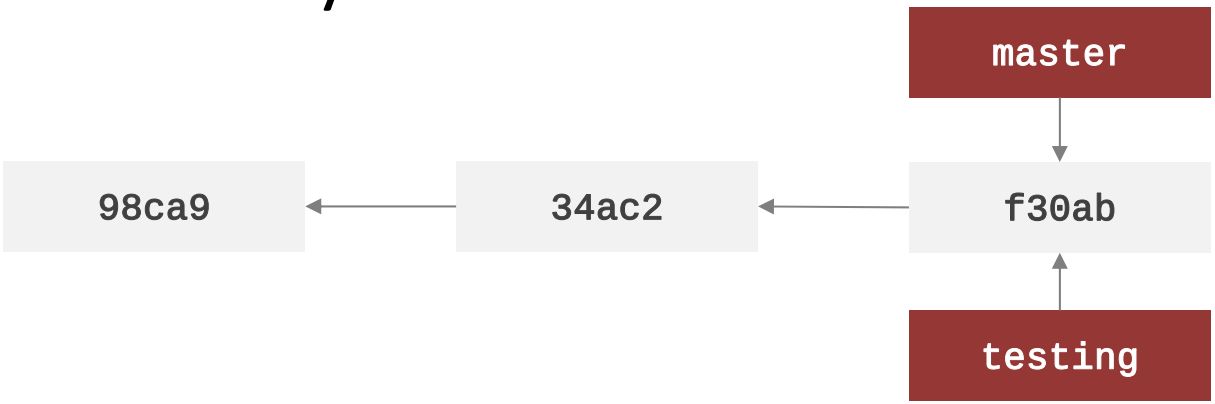
Git Branching

Creating a New Branch

- Creating a new branch generates a new pointer for you to move around
- Let's create a new branch called testing using the `git branch` command:

```
git branch testing
```

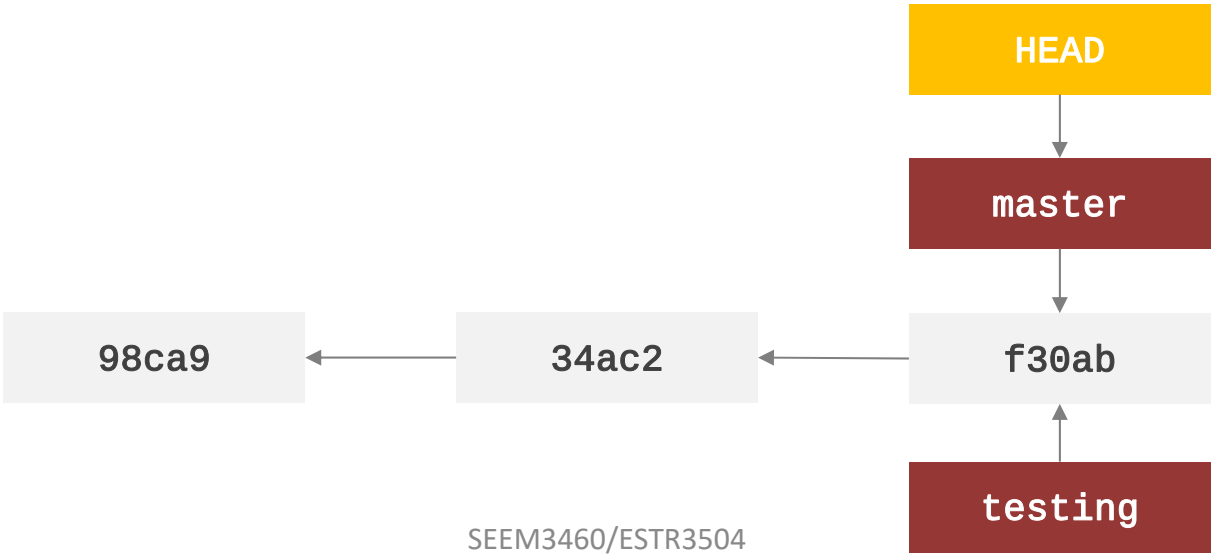
- This creates a new pointer to the same commit you're currently on.



Git Branching

Creating a New Branch

- How does Git know what branch you're currently on?
- It keeps a special pointer called HEAD
- This is a pointer to the local branch you're currently on.
- In this case, you're still on master.
- The `git branch` command only created a new branch — it didn't switch to that branch.



Git Branching

Creating a New Branch

- You can easily see this by running a simple `git log` command that shows you where the branch pointers are pointing
- This option is called `--decorate`

```
git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add...
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

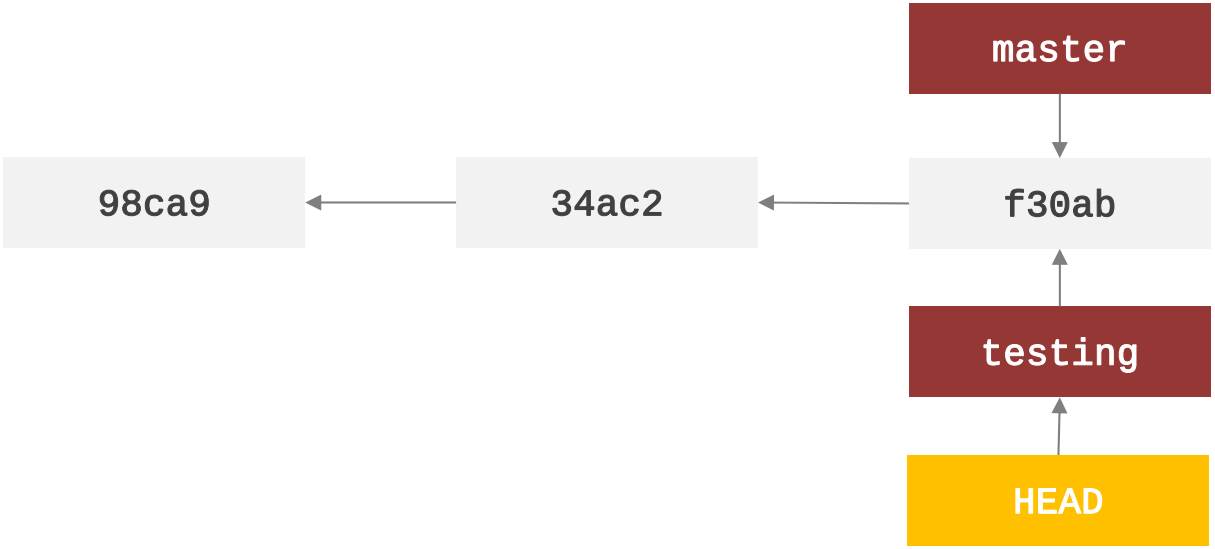
Git Branching

Switching Branches

- To switch to an existing branch, you run the `git checkout` command
- Let's switch to the new testing branch:

```
git checkout testing
```

- This moves HEAD to point to the testing branch

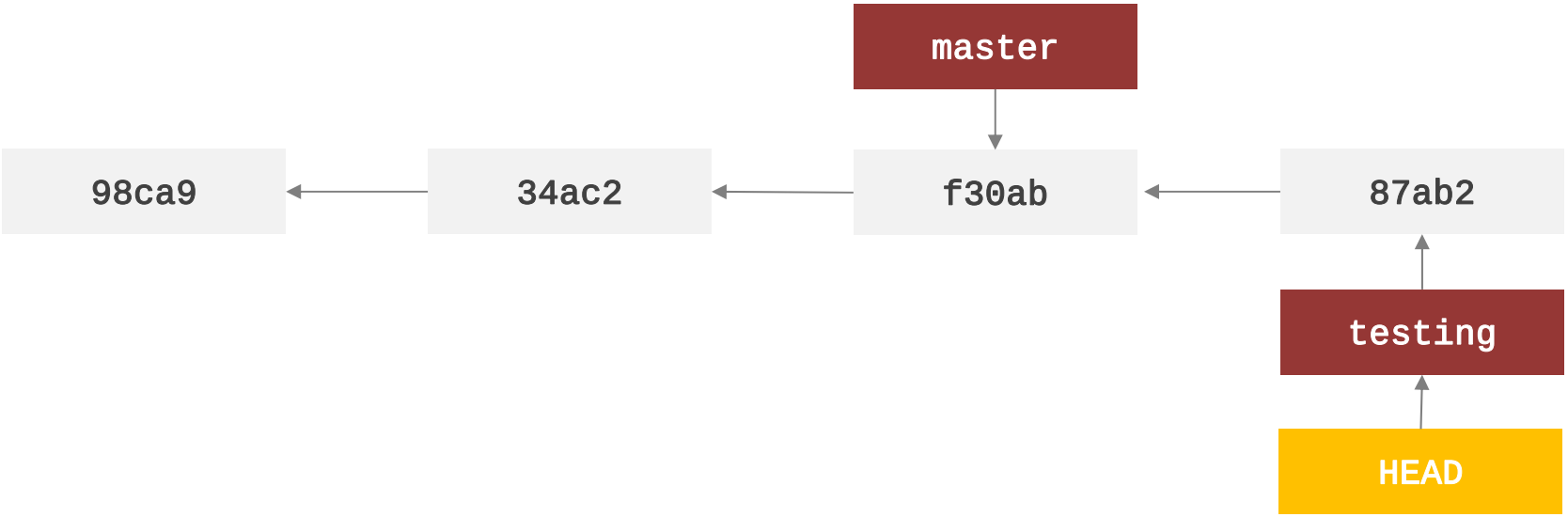


Git Branching

Switching Branches

- Let’s do another commit:

```
vim test.rb
git commit -a -m 'made a change'
```

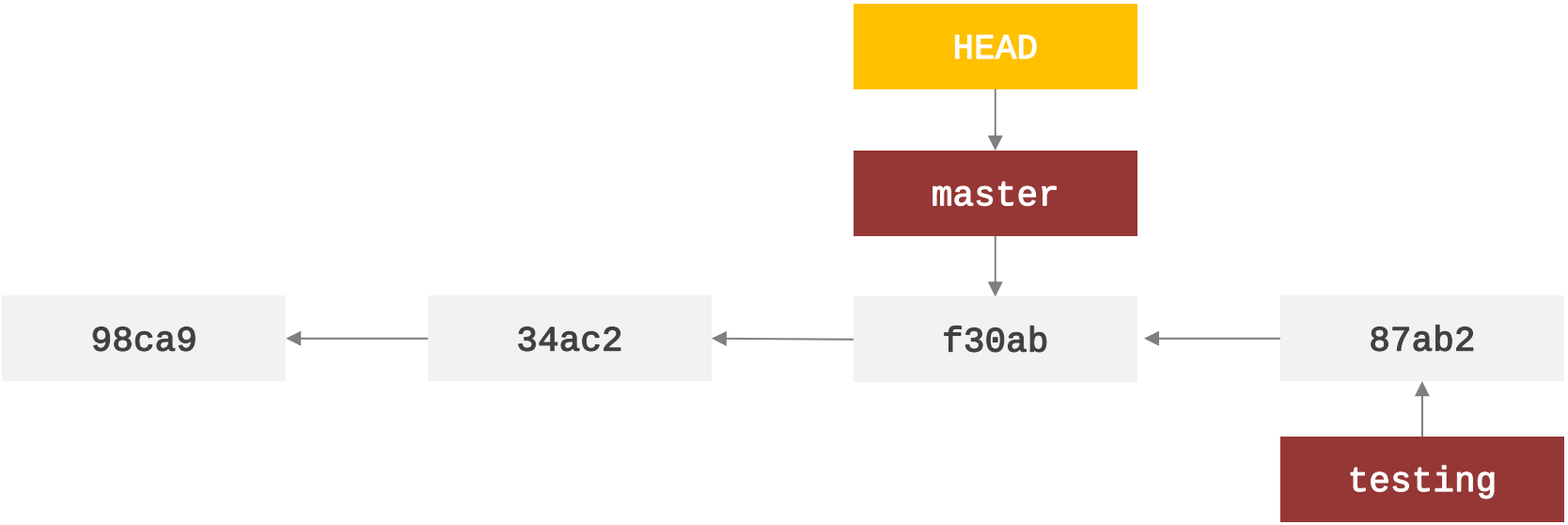


Git Branching

Switching Branches

- Now your testing branch has moved forward
- But your master branch still points to the commit you were on when you ran git checkout to switch branches
- Let's switch back to the master branch:

```
git checkout master
```



Git Branching

Switching Branches

- `git log` doesn't show all the branches all the time
- If you were to run `git log` right now, you might wonder where the "testing" branch you just created went, as it would not appear in the output
- Git just doesn't know that you're interested in that branch and it is trying to show you what it thinks you're interested in
- By default, `git log` will only show commit history below the branch you've checked out
- To show commit history for the desired branch you have to explicitly specify it:

```
git log testing
```

- To show all of the branches, add `--all` to your git log command.

Git Branching

Switching Branches

- `git checkout master` command did two things:
- It moved the HEAD pointer back to point to the master branch
- It reverted the files in your working directory back to the snapshot that master points to
- This also means the changes you make from this point forward will diverge from an older version of the project
- It essentially rewinds the work you've done in your testing branch so you can go in a different direction

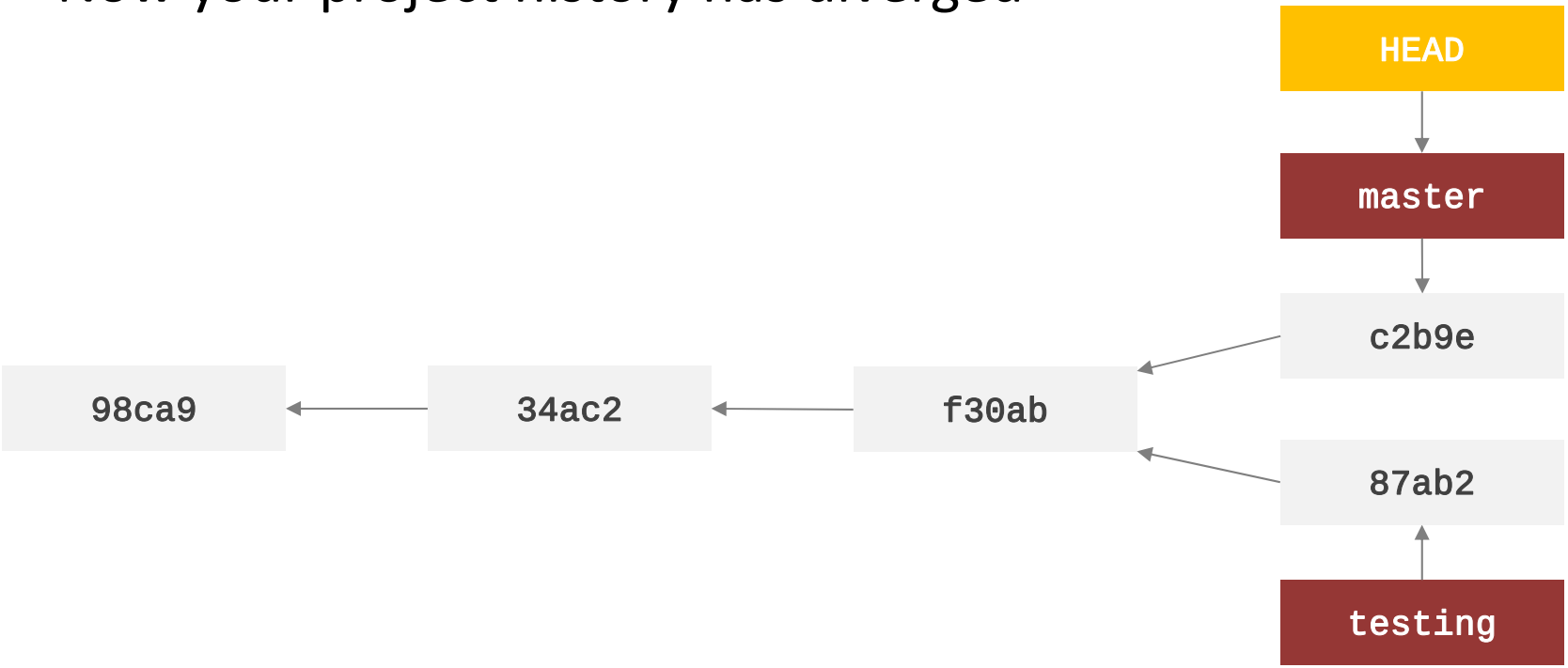
Git Branching

Switching Branches

- Let's make a few changes and commit again:

```
vim test.rb
git commit -a -m 'made other changes'
```

- Now your project history has diverged



Git Branching

Switching Branches

- You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work
- Both of those changes are isolated in separate branches
- You can switch back and forth between the branches and merge them together when you're ready
- And you did all that with simple `branch`, `checkout`, and `commit` commands.

Git Branching

Switching Branches

- You can also see this easily with the git log command

```
git log --oneline --decorate --graph --all
$ * c2b9e (HEAD, master) Made other changes
| * 87ab2 (testing) Made a change
|/
* f30ab Add feature #32 - ability to add new formats to the central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

- It will print out the history of your commits, showing where your branch pointers are and how your history has diverged

Git Branching

Switching Branches

- A branch in Git is a simple file that contains the 40 character SHA-1 checksum of the commit it points to
- Branches are cheap to create and destroy
- Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline)

Git Branching

Switching Branches

- Creating a new branch and switching to it at the same time

```
git checkout -b <newbranchname>
```

- From Git version 2.23 onwards you can use git switch instead of git checkout to:
  - Switch to an existing branch:

```
git switch <testing-branch>
```

  - Create a new branch and switch to it:

```
git switch -c <new-branch>
```

  - The `-c` flag stands for create, you can also use the full flag: `--create`
  - Return to your previously checked out branch:

```
git switch -
```

Git Branching

Basic Branching and Merging

- Let's go through a simple example of branching and merging with a workflow that you might use in the real world
  - Do some work on a website
  - Create a branch for a new user story you're working on
  - Do some work in that branch
- At this stage, you'll receive a call that another issue is critical and you need a hotfix
  - Switch to your production branch
  - Create a branch to add the hotfix
  - After it's tested, merge the hotfix branch, and push to production
  - Switch back to your original user story and continue working

Git Branching

Basic Branching

- First, let's say you're working on your project and have a couple of commits already on the master branch

Git Branching

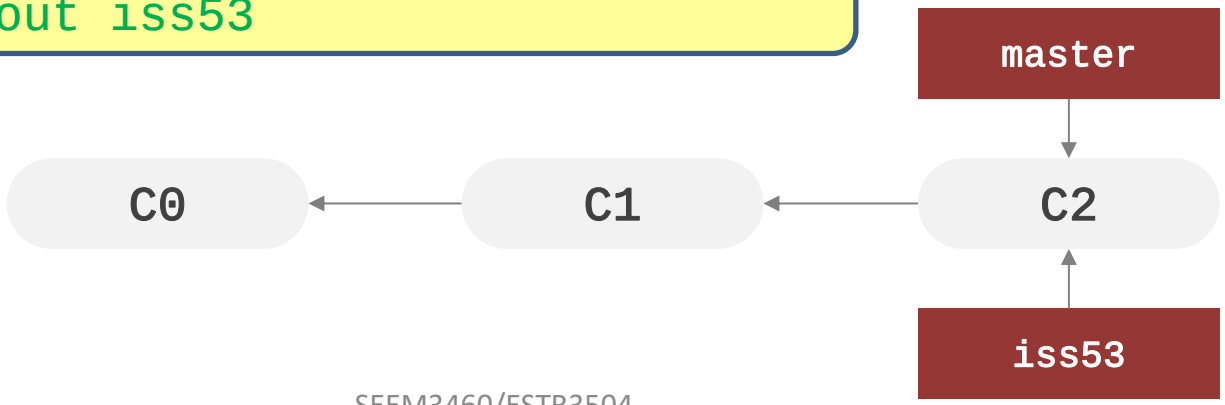
Basic Branching

- You've decided that you're going to work on issue #53 in the issue-tracking system
- To create a new branch and switch to it at the same time, you can run the `git checkout` command with the `-b` switch:

```
git checkout -b iss53
```

- This is shorthand for:

```
git branch iss53
git checkout iss53
```

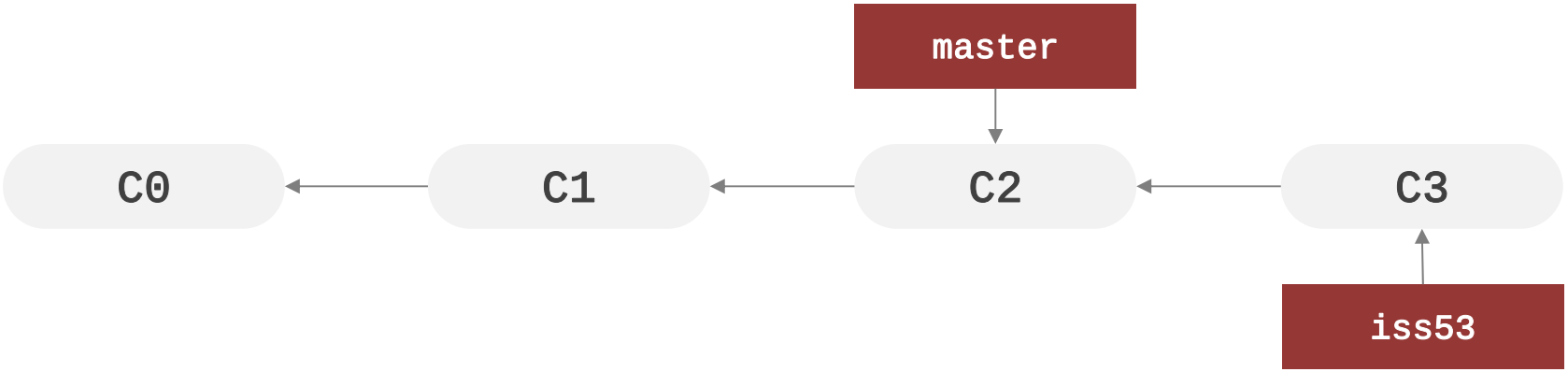


Git Branching

Basic Branching

- You work on your website and do some commits
- Doing so moves the iss53 branch forward, because you have it checked out (that is, your HEAD is pointing to it):

```
vim index.html
git commit -a -m 'Create new footer [issue 53]'
```



Git Branching

Basic Branching

- Now you get the call that there is an issue with the website, and you need to fix it immediately
- With Git
  - you don't have to deploy your fix along with the iss53 changes you've made
  - you don't have to put a lot of effort into reverting those changes before you can work on applying your fix to what is in production
- All you have to do is switch back to your master branch

Git Branching

Basic Branching

- Note that Git won't let you switch branches if your working directory or staging area has uncommitted changes that conflict with the branch you're checking out
- It's best to have a clean working state when you switch branches
- Let's assume you've committed all your changes, so you can switch back to your master branch:

```
git checkout master
```

Git Branching

Basic Branching

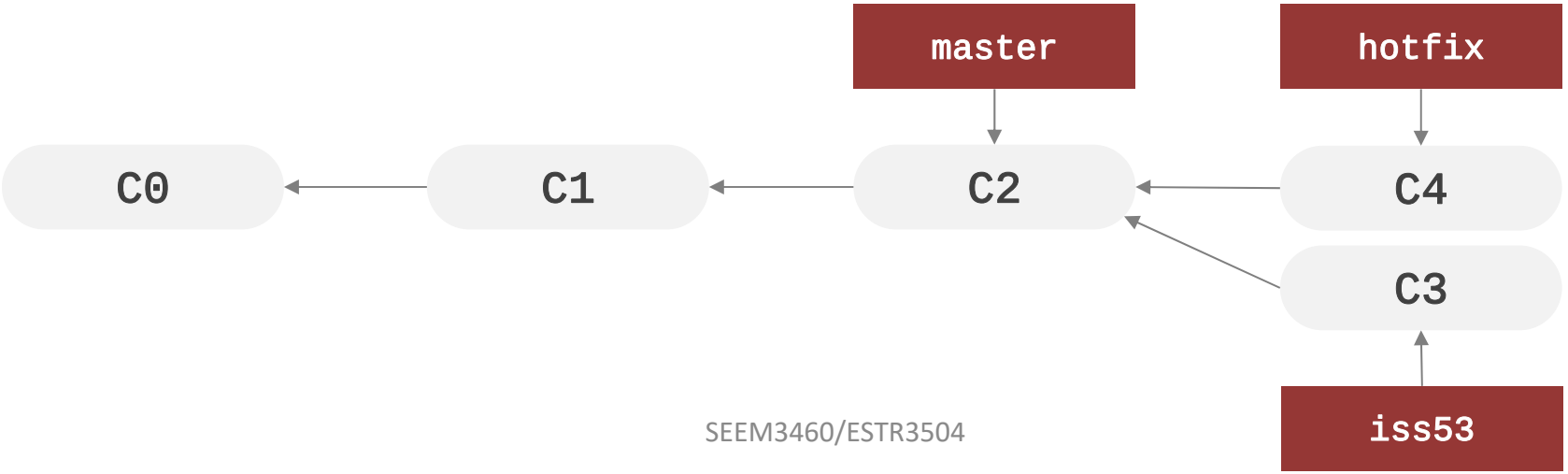
- At this point, your project working directory is exactly the way it was before you started working on issue #53
- You can concentrate on your hotfix
- When you switch branches :
  - Git resets your working directory to look like it did the last time you committed on that branch
  - It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it

Git Branching

Basic Branching

- Next, you have a hotfix to make
- Let's create a hotfix branch on which to work until it's completed:

```
git checkout -b hotfix
Switched to a new branch 'hotfix'
vim index.html
git commit -a -m 'Fix broken email address'
[hotfix 1fb7853] Fix broken email address
 1 file changed, 2 insertions(+)
```

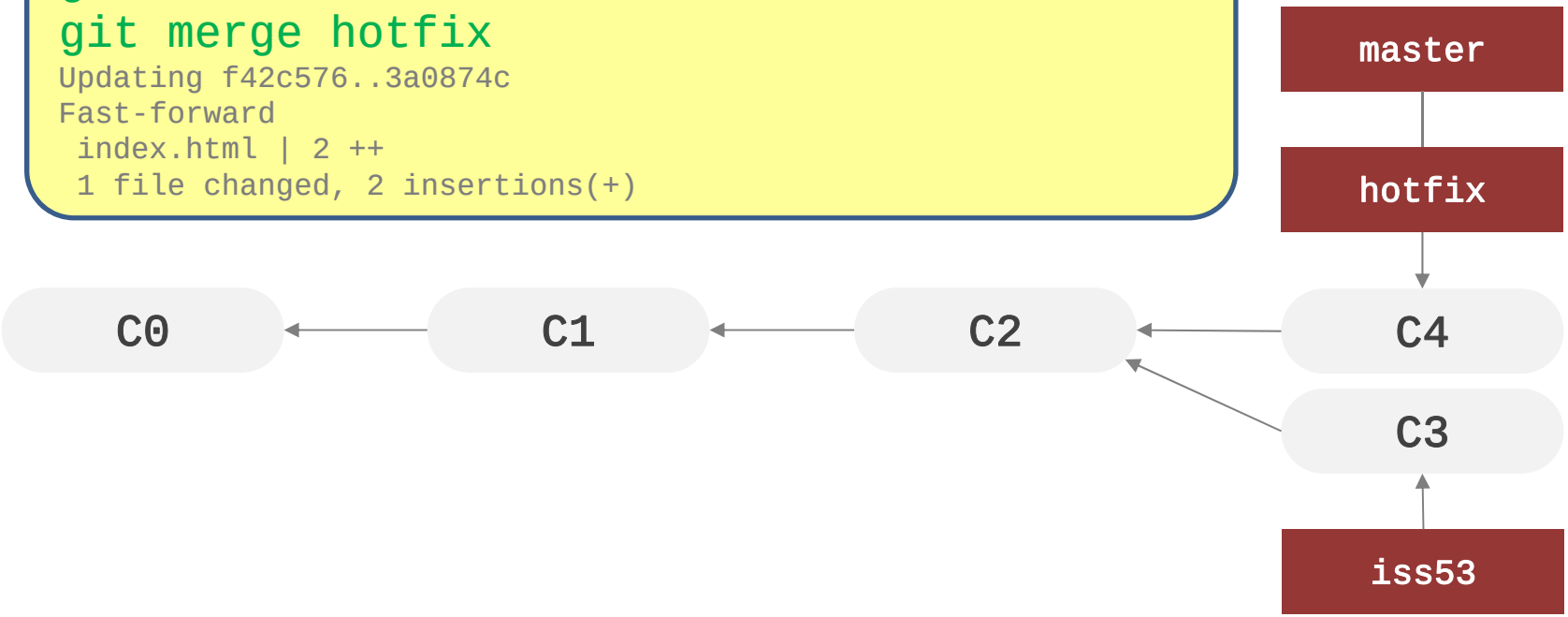


Git Branching

Basic Branching

- You can run your tests, make sure the hotfix is what you want, and finally merge the hotfix branch back into your master branch to deploy to production
- You do this with the `git merge` command:

```
git checkout master
git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```



Git Branching

Basic Branching

- You'll notice the phrase "fast-forward" in that merge
- Because the commit C4 pointed to by the branch hotfix you merged in was directly ahead of the commit C2 you're on, Git simply moves the pointer forward
- To phrase that another way, when you try to merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a "fast-forward"
- Your change is now in the snapshot of the commit pointed to by the master branch, and you can deploy the fix

Git Branching

Basic Branching

- After your super-important fix is deployed, you're ready to switch back to the work you were doing before you were interrupted
- However, first you'll delete the hotfix branch, because you no longer need it — the master branch points at the same place
- You can delete it with the `-d` option to git branch:

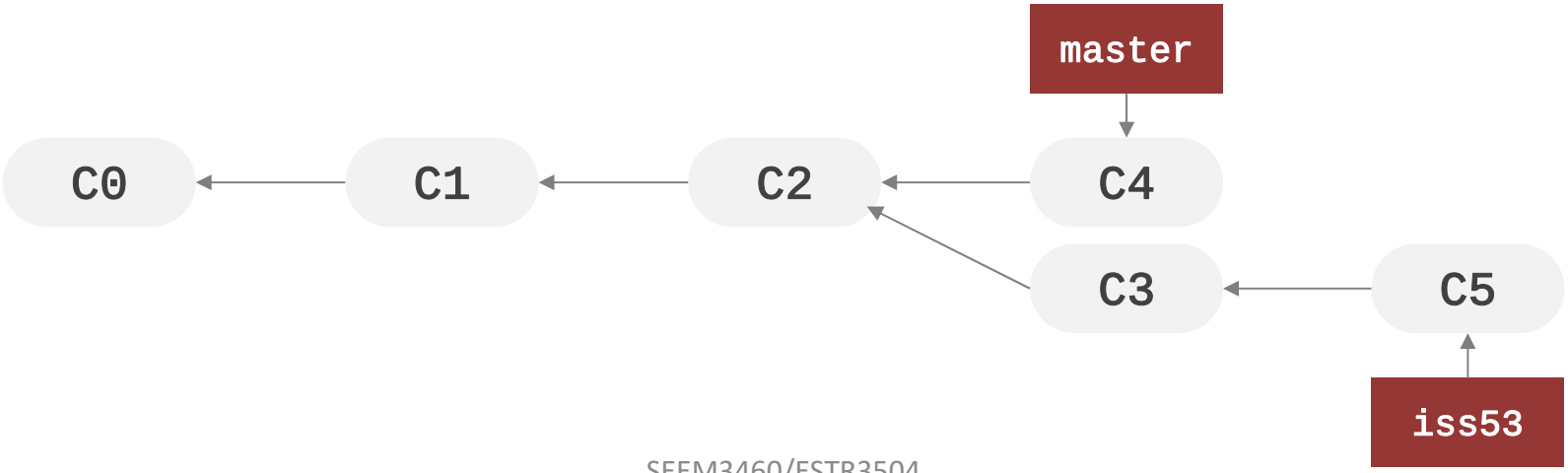
```
git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Git Branching

Basic Branching

- Now you can switch back to your work-in-progress branch on issue #53 and continue working on it

```
git checkout iss53
Switched to branch "iss53“
vim index.html
git commit -a -m 'Finish the new footer [issue 53]'
[iss53 ad82d7a] Finish the new footer [issue 53]
1 file changed, 1 insertion(+)
```



Git Branching

Basic Merging

- It's worth noting here that the work you did in your hotfix branch is not contained in the files in your iss53 branch
- If you need to pull it in, you can merge your master branch into your iss53 branch by running `git merge master`, or you can wait to integrate those changes until you decide to pull the iss53 branch back into master later.

Git Branching

Basic Merging

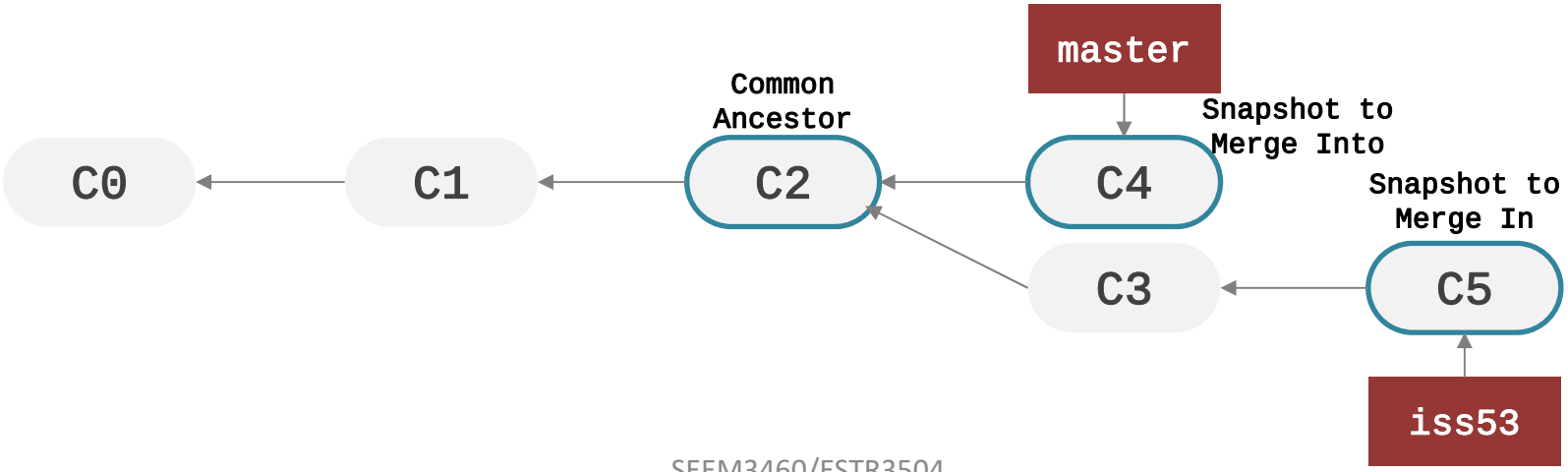
- Suppose you've decided that your issue #53 work is complete and ready to be merged into your master branch
- You'll merge your iss53 branch into master, much like you merged your hotfix branch earlier
- All you have to do is check out the branch you wish to merge into and then run the git merge command:

```
git checkout master
Switched to branch 'master'
git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

Git Branching

Basic Merging

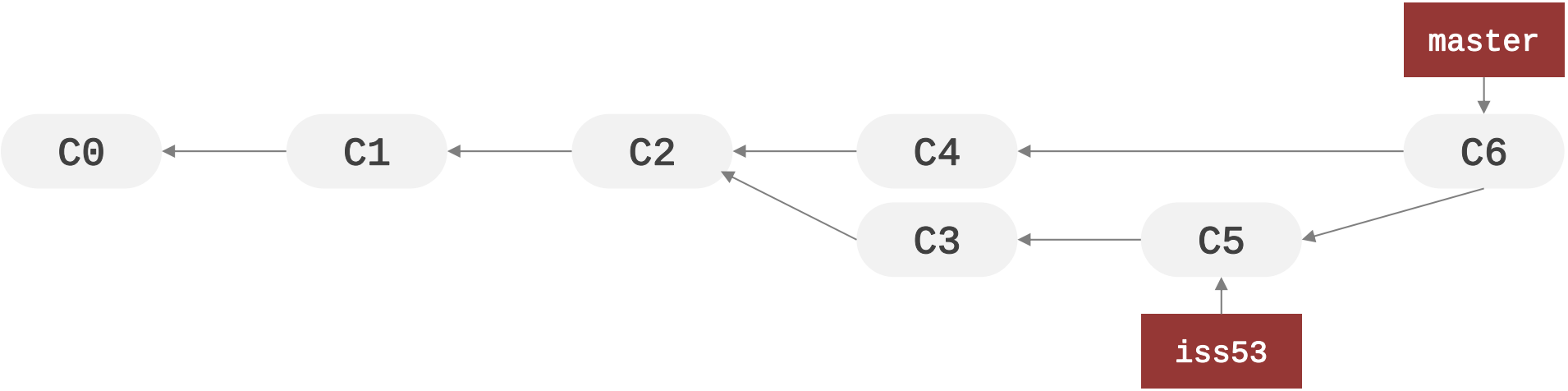
- Your development history has diverged from some older point
- The commit on the branch you're on isn't a direct ancestor of the branch you're merging in
- Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two



Git Branching

Basic Merging

- Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it
- This is referred to as a merge commit
- This is special in that it has more than one parent

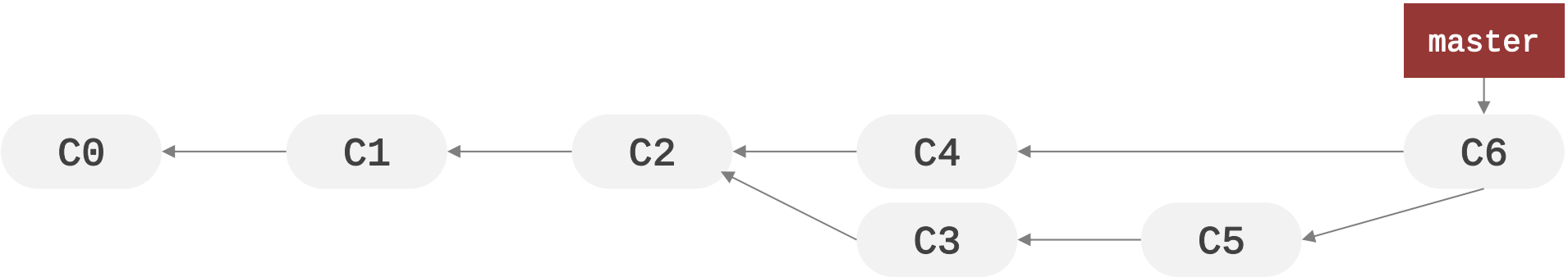


Git Branching

Basic Merging

- Now that your work is merged in, you have no further need for the iss53 branch
- You can close the issue in your issue-tracking system, and delete the branch:

```
git branch -d iss53
```



Git Branching

Basic Merge Conflicts

- Occasionally, this process doesn't go smoothly
- If you changed the same part of the same file differently in the two branches you're merging, Git won't be able to merge them cleanly
- If your fix for issue #53 modified the same part of a file as the hotfix branch, you'll get a merge conflict that looks something like this:

```
git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git Branching

Basic Merge Conflicts

- Git hasn't automatically created a new merge commit
- It has paused the process while you resolve the conflict
- If you want to see which files are unmerged at any point after a merge conflict, you can run `git status`:

```
git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

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

Git Branching

Basic Merge Conflicts

- Anything that has merge conflicts and hasn't been resolved is listed as unmerged
- Git adds standard conflict-resolution markers to the files that have conflicts
- You can open them manually and resolve those conflicts
- Your file contains a section that looks something like this:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Git Branching

Basic Merge Conflicts

- This means the version in HEAD (your master branch) is the top part of that block (everything above the =======)
- The version in your iss53 branch looks like everything in the bottom part
- In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself
- For instance, you might resolve this conflict by replacing the entire block with this:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

Git Branching

Basic Merge Conflicts

- This resolution has a little of each section, and the <<<<<<<, =======, and >>>>>>> lines have been completely removed
- After you've resolved each of these sections in each conflicted file, run `git add` on each file to mark it as resolved
- Staging the file marks it as resolved in Git

Git Branching

Basic Merge Conflicts

- If you want to use a graphical tool to resolve these issues, you can run `git mergetool`, which fires up an appropriate visual merge tool and walks you through the conflicts:

```
git mergetool
This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more
details.
'git mergetool' will now attempt to use one of the following
tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse
diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Git Branching

Basic Merge Conflicts

- If you want to use a merge tool other than the default (Git chose opendiff in this case), you can see all the supported tools listed at the top after “one of the following tools”
- Just type the name of the tool you’d rather use
- After you exit the merge tool, Git asks you if the merge was successful
- If you tell the script that it was, it stages the file to mark it as resolved for you
- You can run `git status` again to verify that all conflicts have been resolved:

```
git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

Git Branching

Basic Merge Conflicts

- If you're happy with that, and you verify that everything that had conflicts has been staged, you can type `git commit` to finalize the merge commit
- The commit message by default looks something like this:

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#       modified:   index.html
```

Git Branching

Basic Merge Conflicts

- You can modify this commit message with details about how you resolved the merge and explain why you did the changes you made if these are not obvious
- It would be helpful to others looking at this merge in the future