EPAM- Centre of Excellence

"Version Control with Git & GitHub"

Git Install & Usage

C. Ravi Kishore Reddy

Assistant Professor

Department of CSE

Contents

- Version Control Concept
- Version Control Types
- Why Git
- GitHub and its Usage
- Download, Install and configure Git
- Using Git
- Git graphical tools
- Git Internals
- Branching and Merging
- Tags, Stash, Remotes, Branching Strategies

Installing and configuring Git

- Git - Downloading Package (git-scm.com)
- Download the installer and install git in the machine.
- Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates.

Installing and configuring Git

- The first thing you should do when you install Git is to set your user name and email address.
- This is important because every Git commit uses this information

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

- You need to do this only once if you pass the --global option, because then Git will always use that information for anything you do on that system.
- If you want to override this with a different name or email address for specific projects, you can run the command without the --global option when you're in that project

Installing and configuring Git

- **Your Editor**
  - You can configure the default text editor that will be used when Git needs you to type in a message.
  - $ git config --global core.editor emacs
  - $ git config --global core.editor '"C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin

Installing and configuring Git

- **Your default branch name**
  - By default Git will create a branch called master when you create a new repository with git init.
  - From Git version 2.28 onwards, you can set a different name for the initial branch.
  - To set main as the default branch name do
  - $ git config --global init.defaultBranch main

Installing and configuring Git

Checking Your Settings

- If you want to check your configuration settings, you can use:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Git Basics

- **Getting a Git Repository**
- You typically obtain a Git repository in one of two ways:
  - *You can take a local directory that is currently not under version control, and turn it into a Git repository*
  - *You can clone an existing Git repository from elsewhere.*

Git Basics

- **Initializing a Repository in an Existing Directory**
- First need to go to that project's directory
- $ git init
- This creates a new subdirectory named .git that contains all of your necessary repository files — a Git repository skeleton. **Git Internals.**
- At this point, nothing in your project is tracked yet.

Git Basics

- **Initializing a Repository in an Existing Directory**
- If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit.

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'Initial project version'
```

Git Basics

- **Cloning an Existing Repository**
- If you want to get a copy of an existing Git repository — for example, a project you'd like to contribute to — the command you need is git clone.
- Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run git clone.

Git Basics

- **Cloning an Existing Repository**
- $ git clone https://github.com/libgit2/libgit2 mylibgit
- That command does the same thing as the previous one, but the target directory is called mylibgit.
- Git has a number of different transfer protocols you can use.
- The previous example uses the https:// protocol, but you may also see git:// or user@server:path/to/repo.git, which uses the SSH transfer protocol.

Git Basics

- **Recording Changes to the Repository**
- At this point, you should have a Git repository on your local machine, and a working copy of all of its files in front of you.
- Typically, you'll want to start making changes and committing snapshots of those changes into your repository each time the project reaches a state you want to record.

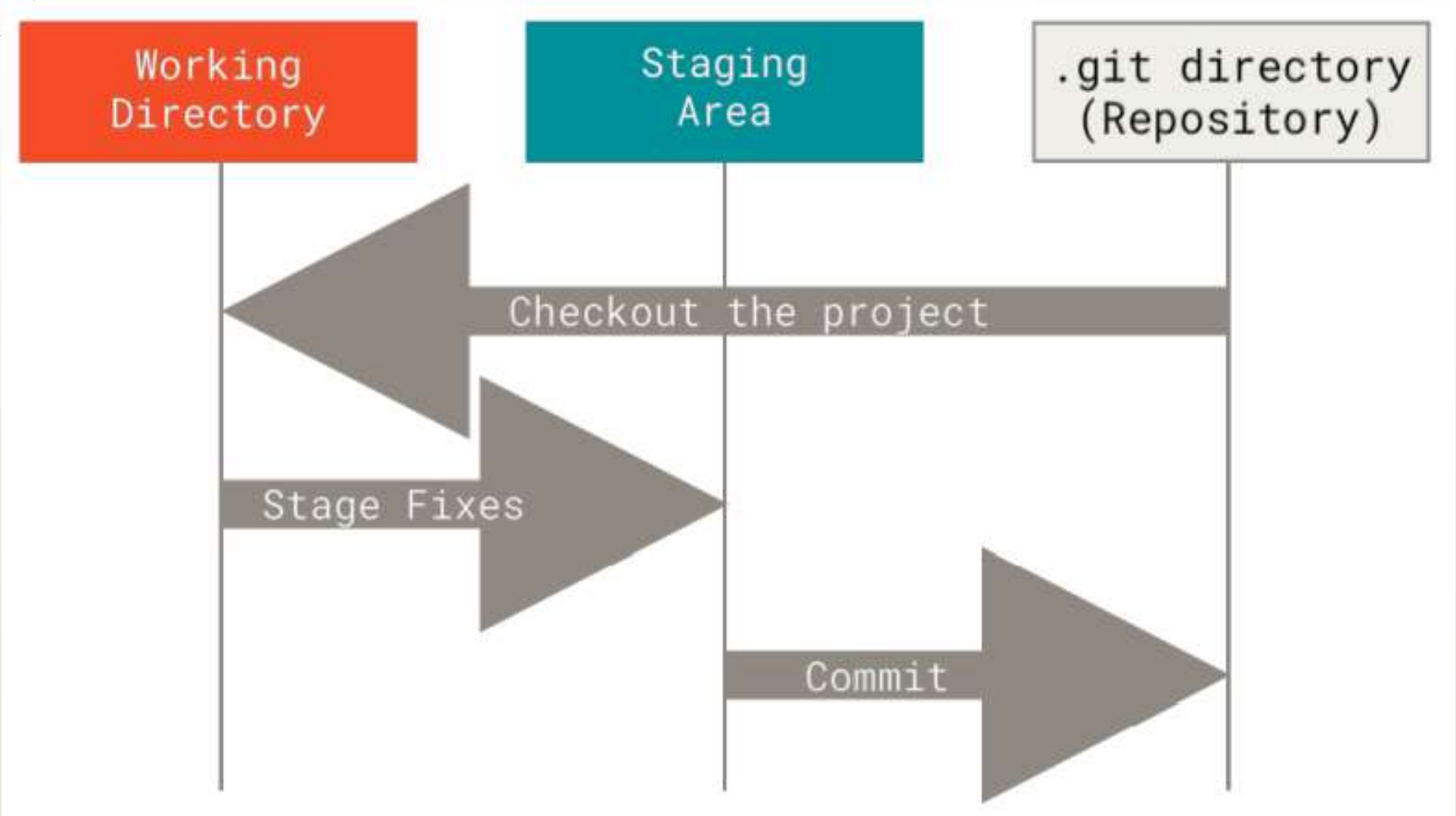
The Three States

- Git has three main states that your files can reside in:
  - Modified
  - Staged
  - Committed

The Three States

- Git has three main states that your files can reside in:
  - **Modified-** Modified means that you have changed the file but have not committed it to your database yet.
  - **Staged-** Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
  - **Committed-** Committed means that the data is safely stored in your local database

The Three States



The Three States

- **The basic workflow is as follows:**
  - You modify files in your working tree.
  - You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area.
  - You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Git Basics

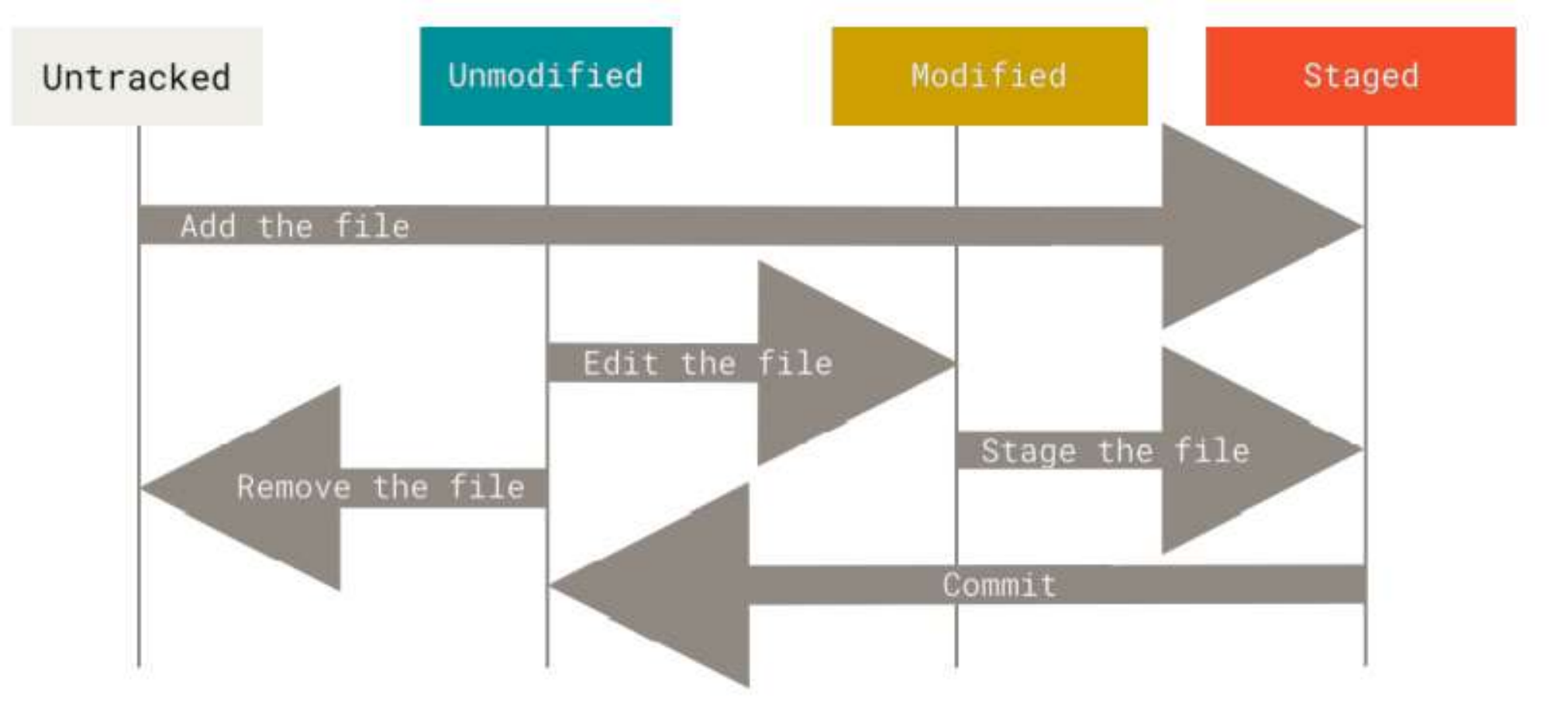
- **Recording Changes to the Repository**
- Remember that each file in your working directory can be in one of two states: tracked or untracked.
- Tracked files are files that were in the last snapshot, as well as any newly staged files.
- They can be unmodified, modified, or staged.
- In short, tracked files are files that Git knows about.

Git Basics

- **Recording Changes to the Repository**
- Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area.
- When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.

Git Basics

Recording Changes to the Repository



Git Basics

- **Checking the Status of Your Files**
- The main tool you use to determine which files are in which state is the ***git status*** command.
- If you run this command directly after a clone

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Git Basics

Checking the Status of Your Files

Let's say you add a new file to your project, a simple README file. If the file didn't exist before, and you run git status, you see your untracked file like so:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

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

Git Basics

- **Tracking New Files**
- In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:
- $ git add README

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

    new file:   README
```

Git Basics

- **Staging Modified Files**
- Let's change a file that was already tracked. If you change a previously tracked file called CONTRIBUTING.md

```
$ 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)

    new file:   README

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:   CONTRIBUTING.md
```

Git Basics

Staging Modified Files

```
$ git add CONTRIBUTING.md
$ 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)

    new file:   README
    modified:   CONTRIBUTING.md
```

Git Basics

- **Staging Modified Files**
- At this point, suppose you remember one little change that you want to make in CONTRIBUTING.md before you commit it. You open it again and make that change, and you're ready to commit. However, let's run git status one more time

Git Basics

Staging Modified Files

```
$ vim CONTRIBUTING.md
$ 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)

    new file:   README
    modified:   CONTRIBUTING.md

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:   CONTRIBUTING.md
```

Git Basics

- **Staging Modified Files**
- Now CONTRIBUTING.md is listed as both staged and unstaged. How is that possible?
- It turns out that Git stages a file exactly as it is when you run the git add command.
- If you modify a file after you run git add, you have to run git add again to stage the latest version of the file

Git Basics

- **Staging Modified Files**
- Short Status
- $ git status –s
- Ignoring Files
- $ cat .gitignore *.[oa]
- line tells Git to ignore any files ending in “.o” or “.a”

Git Basics

- **Staging Modified Files**
- If you want to know exactly what you changed, not just which files were changed — you can use the *git diff command*
- git status answers very generally by listing the file names
- git diff shows you the exact lines added and removed — the patch, as it were.

Git Basics

Staging Modified Files

```
$ 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:   README

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:   CONTRIBUTING.md
```

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Git Basics

- **Committing Your Changes**
- Now that your staging area is set up the way you want it, you can commit your changes.
- Remember any files you have created or modified that you haven't run git add on since you edited them — won't go into this commit
- $ git commit

Git Basics

- **Committing Your Changes**
- Alternatively, you can type your commit message inline with the commit command by specifying it after a -m flag

```
$ git commit -m "Story 182: fix benchmarks for speed"
[master 463dc4f] Story 182: fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Thank You