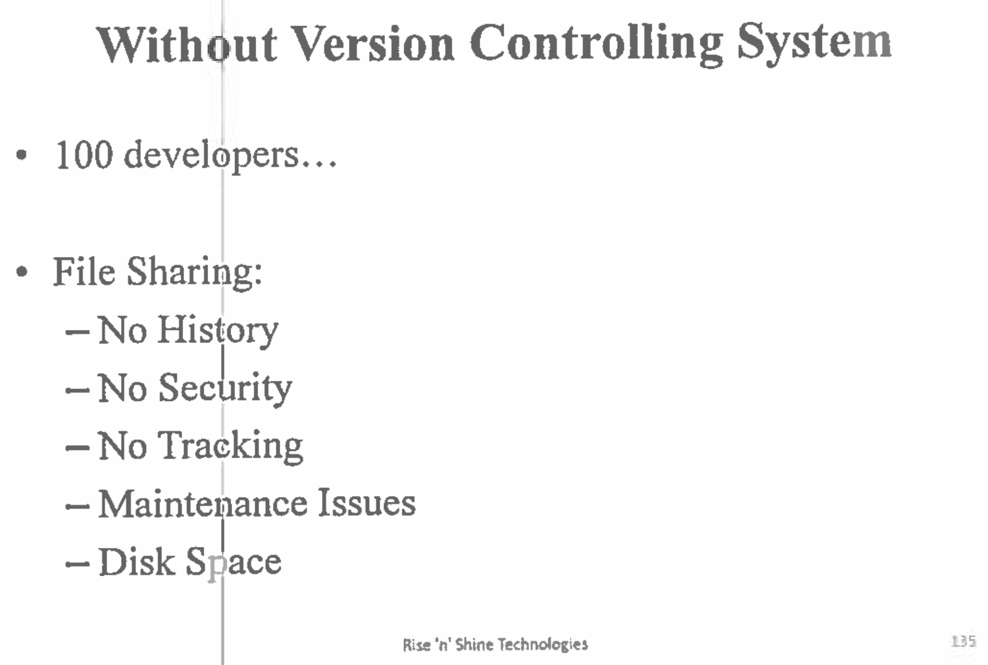
Devops - 5

Version Controlling Tool - Git

Version Control

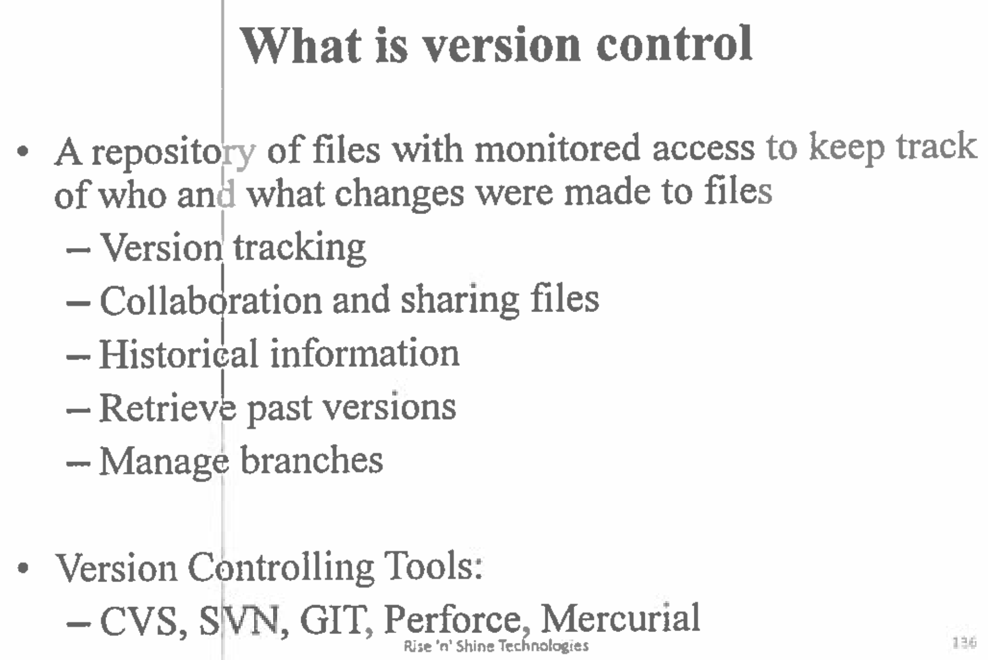
- Without Version Control
- What is version control
- Why use one



Without Version Controlling System

- 100 developers…
- File Sharing:
  - No History
  - No Security
  - No Tracking
  - Maintenance Issues
  - Disk Space

135



What is version control

- A repository of files with monitored access to keep track of who and what changes were made to files
  - Version tracking
  - Collaboration and sharing files
  - Historical information
  - Retrieve past versions
  - Manage branches
- Version Controlling Tools:
  - CVS, SVN, GIT, Perforce, Mercurial

136

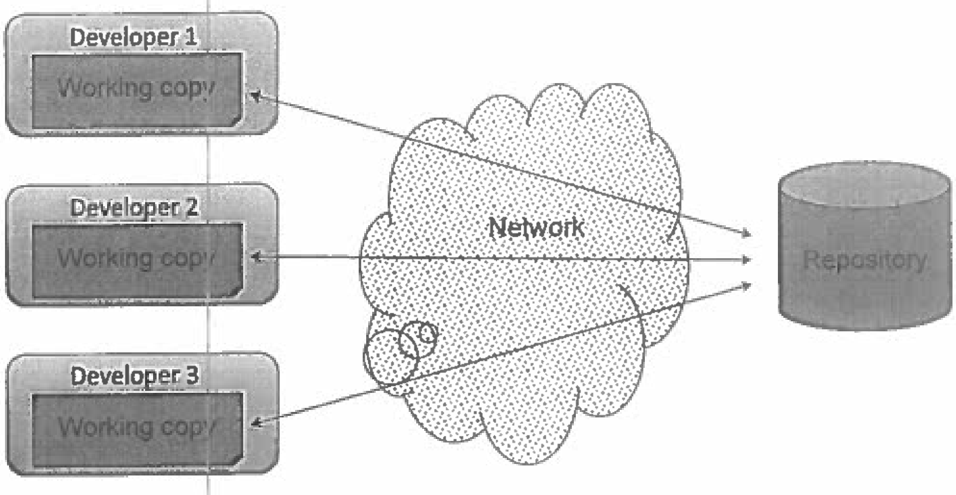
Why use one

- In code development, a version control system is, at this point, almost mandatory
  - Multiple developers working on more than one project
  - Must be able to roll back to any version
  - Must be able to tag releases, and milestones

Tool Types

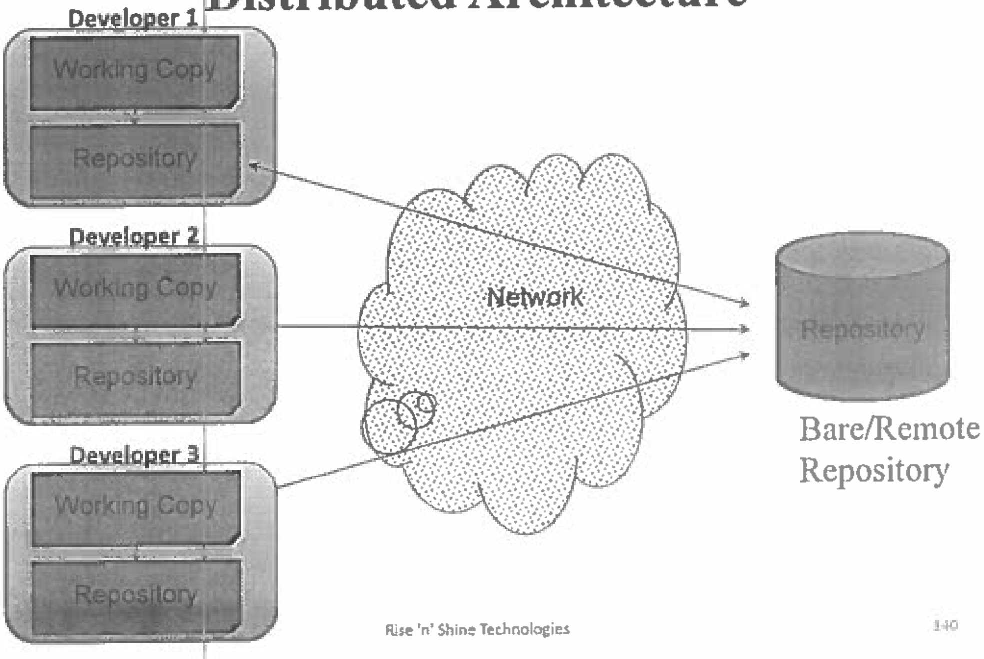
- Server Based Tools
  - CVS
  - SVN
  - Perforce
- Distributed VCS Tools
  - Git
  - Mercurial

Client-Server Architecture



139

Distributed Architecture



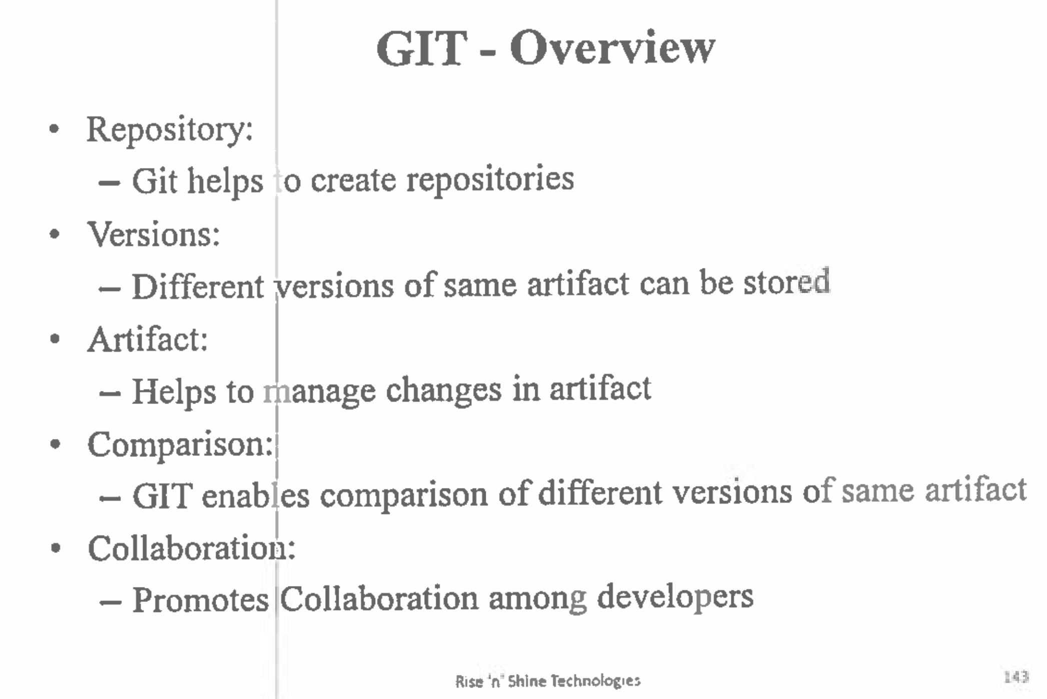
140

Key Concepts

- The **repository** is where files are stored under Git/SVN on the server.
- The contents of a project in the repository are usually copied into a local directory. This is known as the **working directory**, and is separate from the repository.
- The operation of copying the project to the local directory is called **"check out / Pull "** and the reverse is **"check in / Push"**.
- The connection to a Subversion/Git repository is usually given as a **URL**
- **Authentication** is how you prove who you are, such as through a username and password.
- **Authorization** deals with what you have access to and the permissions on what you can do. In this context, authorization is about your file read and write permissions.

Repository versus Working Copy

- Project code is stored in a server in a data store referred to as a "repository."
- Developers "check out / Pull" copies of the project code into their local environments. These copies are referred to as "working copies."
- After making changes to a working copy, the developer "commits" changes to the repository.
- Other developers get these changes by "updating" their working copies.

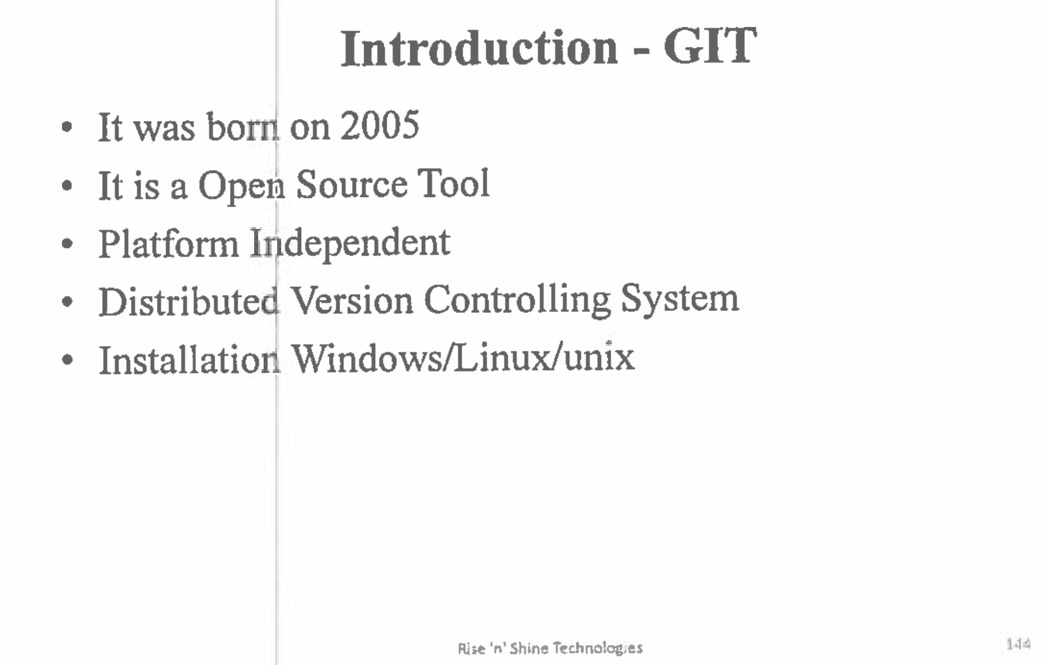


GIT - Overview

- Repository:
  - Git helps to create repositories
- Versions:
  - Different versions of same artifact can be stored
- Artifact:
  - Helps to manage changes in artifact
- Comparison:
  - GIT enables comparison of different versions of same artifact
- Collaboration:
  - Promotes Collaboration among developers

Rise 'n' Shine Technologies 143

143



Introduction - GIT

- It was born on 2005
- It is a Open Source Tool
- Platform Independent
- Distributed Version Controlling System
- Installation Windows/Linux/unix

Rise 'n' Shine Technologies 144

144

Git – Setup and Installation

- **Installing Git on Windows:**
  - Download git for Windows from git-scm.com and install it (Git Bash with default options)

- **Installing GIT on Unix:**
  - Go to git-scm.com/download and then select 'Linux' option then follow the instructions...

- **Verify Git Installation:**
  - which git
  - git version

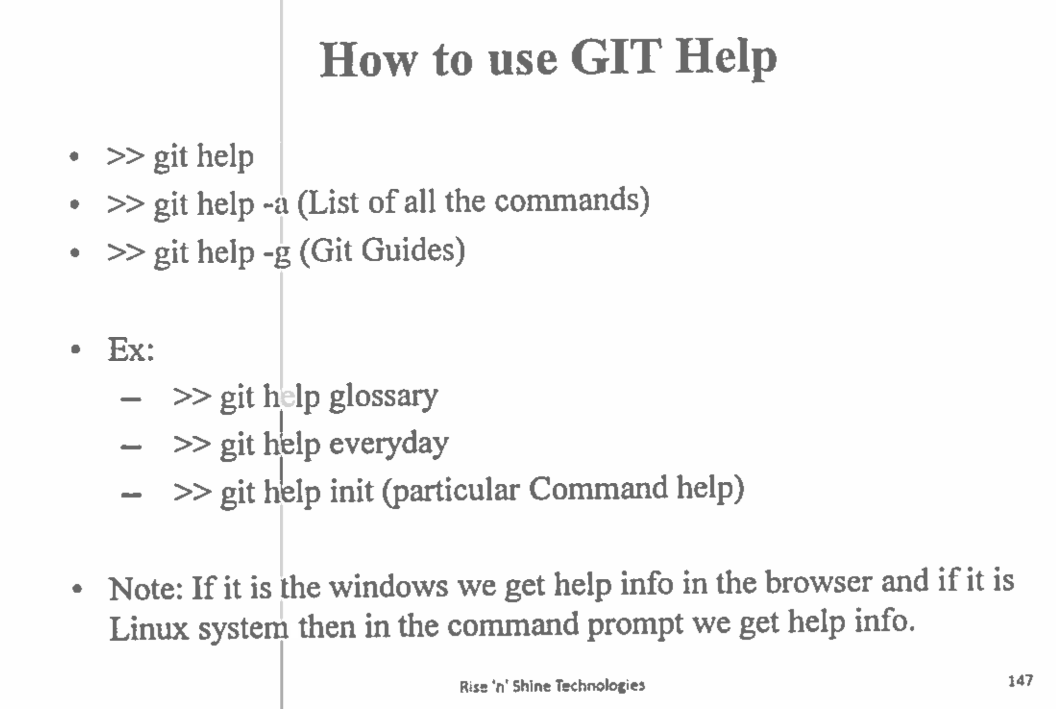
Git – Client Machine

- Client Machine (Developers Machine)

- -- ssh access to git repository from developers Machine

- -- Windows Client
  - Install the git-bash client software.

- -> Unix Client:
  - Install the git software which is installed in the Server Machine



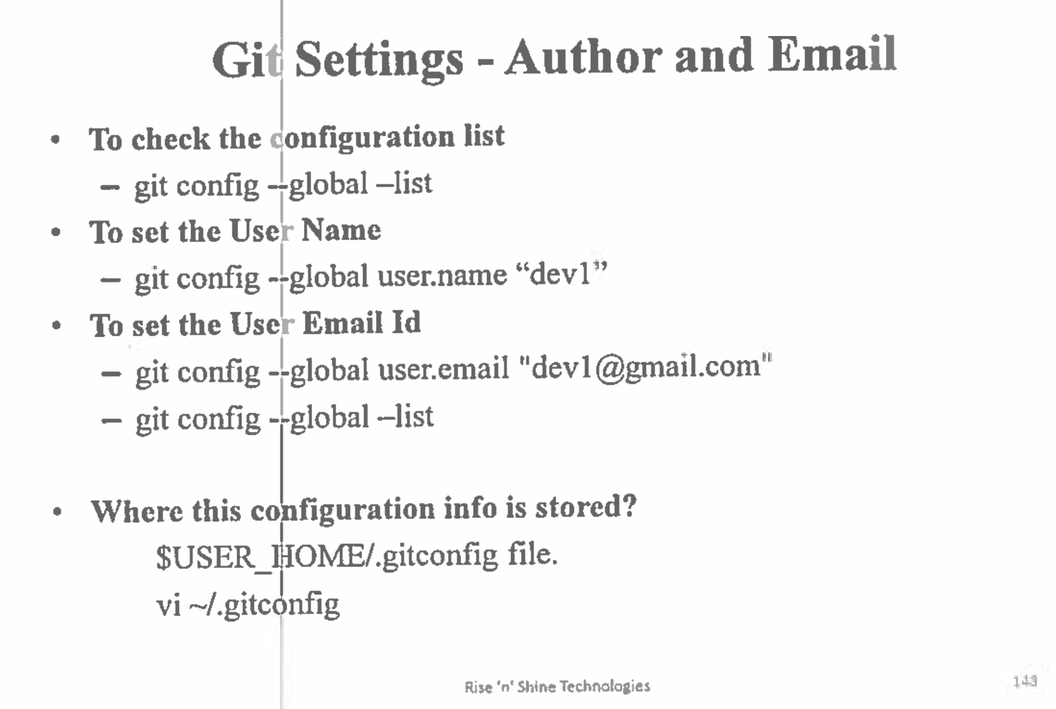
How to use GIT Help

- >> git help
- >> git help -a (List of all the commands)
- >> git help -g (Git Guides)

- Ex:
  - >> git help glossary
  - >> git help everyday
  - >> git help init (particular Command help)

- Note: If it is the windows we get help info in the browser and if it is Linux system then in the command prompt we get help info.

147



Git Settings - Author and Email

- **To check the configuration list**
  - git config --global –list
- **To set the User Name**
  - git config --global user.name “dev1”
- **To set the User Email Id**
  - git config --global user.email "dev1@gmail.com"
  - git config --global –list

- **Where this configuration info is stored?**

  $USER_HOME/.gitconfig file.

  vi ~/.gitconfig

148

Creating Repositories

- Local Repository
  - From Scratch
  - Existing Project
  - Cloning from Remote Repositories
- Bare/Remote Repository
  - GIT Server
  - Git Hub

Setting up Git repository

- 3 Ways of setting up Git Repository
- 1) From Scratch
  - create a repository from absolutely blank state
- 2) Existing Project
  - Convert an existing unversioned project to a Git Repo
- 3) By Copying from Remote Repository
  - Copy an existing Git repository from Git Server / Git HUB

1 – From Scratch

- **Initializing an Empty Repository**
  - – Go to directory where the repository should be created
  - – > mkdir Git_Repo_From_Scratch
  - – > cd Git_Repo_From_Scratch
  - – > git init

Note: The above command creates a set of files and metadata to store the users data as a repository

2 – Existing Project

- **Unversioned Project to Versioned:**
- → Create a project using mvn archetype

- mvn archetype:generate -DgroupId=com.rns.simpleweb -DartifactId=webapp -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false

--> Go to project directory which is 'unversioned'.

- --> Then run the 'git init' command which is going to create .git dir and this dir is called as versioned dir.

Creating Bare/Remote Repository on GitServer

- -- Login to Server
- – Go to directory where the repository should be created

- – > mkdir Git_Repo
- – > cd Git_Repo
- – > git init –bare

- Note: The above command creates a set of files and metadata to store the users data as a repository

3(a) - Cloning Repository from GitServer

- -> Cloning Repository in the client Machine whether that is Win/Unix same process

- --> Create a directory (mkdir git_practise)
- --> git clone root@ip_address:/u01/batch/gitrepo
- -- > cd gitrepo
- --> ls -la (which lists all the repo Meta data)

What is GitHub

- **Web Based**
  - It is a Git Repository with a web Interface
- **Origin**
  - GitHub was born on 2008
- **Subscription**
  - It is having the both free and Paid plans
- **User**
  - Need to create a User account
- **Other Features**
  - Documentation and Bug Tracking
- **Largest Host**
  - It is the largest hosted Git Repository

What is GitHub

- → Create a GitHub user account
- → Create a Repository in the GitHub

- ***What is fork and How to do it?***
  - Creating project from another existing project
  - It encourages outside contribution

  Exercise:

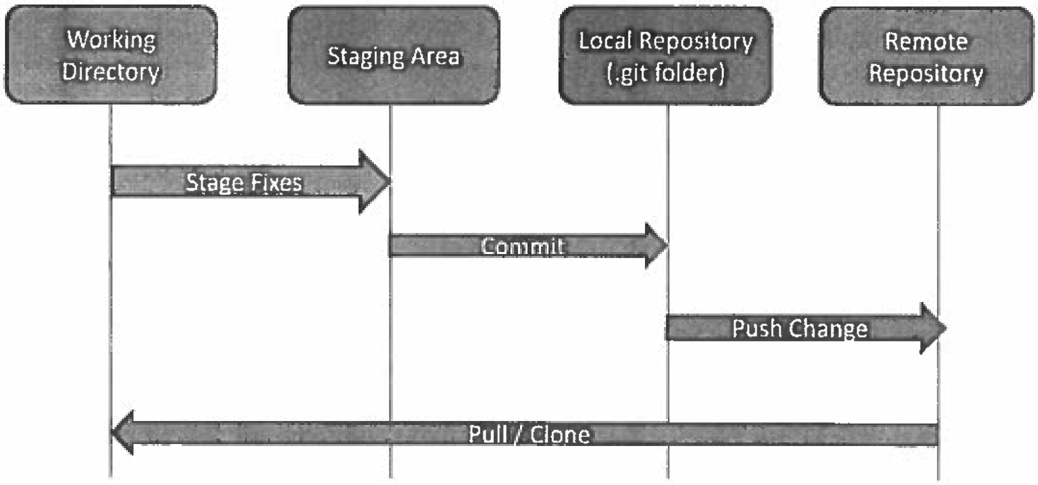
  --*fork the sample github project*

3(b) - Cloning Repository from GitHub

- -> Cloning Repository in the client Machine whether that is Win/Unix same process

- --> Create a directory (mkdir git_practise)
- --> git clone https://github.com/username/repo.git
- --> cd repo_dir
- --> ls -la (which lists all the repo Meta data)

How GIT Works



How GIT Works

- 3 main states of artifact/file
- Modified:
  - Here, the artifact/file goes through change made by the user
- Staged
  - Here, the user / developer adds the artifact/file to Git Index or staging area
- Committed
  - Here, the artifact/file gets safely stored in git database

Git - Adding the Files

- **Process to add the files to Remote Repository:**
- --> 1. Create the file
- --> 2. Stage the File
- --> 3. Commit the file
- --> 4. Push to remote Repository
- **Commands:**
- --> touch firstFile.txt
- --> git status (It gives working tree status)
- --> git add firstFile.txt (It stages the file)
- --> git status
- --> git commit firstFile.txt -m "This is my First commit"
- --> git push origin master
- --> git pull origin master

Git Commands

- Revert Back the Staged File:

```
> git rm --cached fileName
```

- Staging and commit:

```
> git commit -am “msg“
```

- Deleting the file:

```
> git rm file_name
```

- View which branch you are in:

```
> git branch
```

- Information about each line modification done by author

```
> git blame filename
```

- Fetch the changes from Remote Repository:

```
> git fetch origin master
```

- Note: The git pull command performs a git fetch and git merge and the git fetch does not perform merge operations and it just fetches

Git - Parallel Development

- **Dev1** User: (creation -> stage -> commit -> push)

```
git clone dev1@IP_Address:repo
cd repo
touch hello.txt
git status -> gives you the status of ur working tree -> added/new created
git add hello.txt
git commit Hello.txt -m "my first commit“
git log -> shows the log of commits performed as of now
```

- **Dev2:**

```
cloned the repo
added some text to Hello.txt
git add Hello.txt
git commit -m "second commit“
git push
```

Git - Parallel Development Strategy

- **Dev1** User:

```
>> vi hello.txt
>> git status -> gives you the status of ur working tree
>> git add hello.txt
>> git commit Hello.txt -m "my first commit"
>> git log --oneline
>> git push
```

- --> leads to conflicts to the files

Merging the conflicts

- git mergetool
- enter
- open ups merge window --> select respective changes then save and quit.

- git add .
- git commit -m “After merging”

- git branch (make sure you are in master branch)
- git push origin master --> push the merged changes

Git Status and Log Commands

- **Checking Repository Status:**
- --> git status
- --> git status --long
- --> git status –s
- **Checking Committed History:**
- --> git log
- --> git log --oneline
- --> git log filename

  --> git log –author author_name
- --> git shortlog
- --> git log –n 3 --oneline (Last 3 commits)

  --> git log <since>..<until>

  Ex: git log checksum1...Checksum2 –oneline

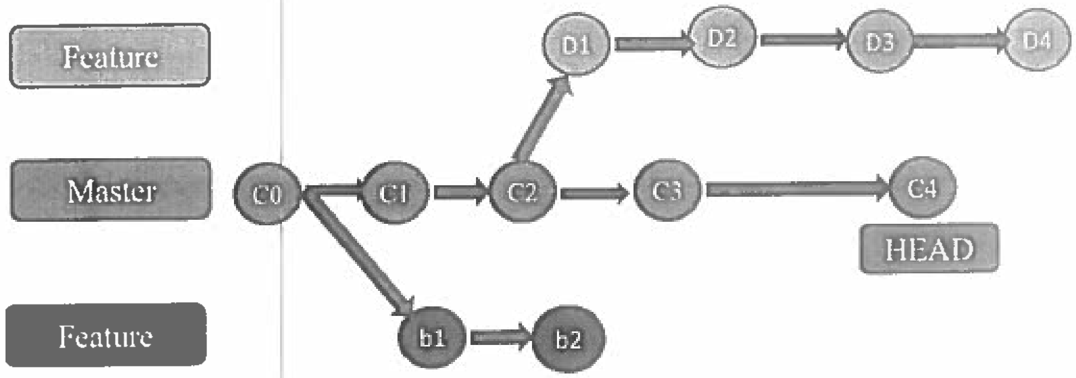
GIT - Directory Structure

- Master –> This is root directory and default one
- Branches –> It contains the Major releases
- tags --> It contains References of Bug fixes code and always read only
- Note:

  -- We create a branch when we want to develop a new feature or to do a bug fix

  -- Unstable code is never committed to main or master branch

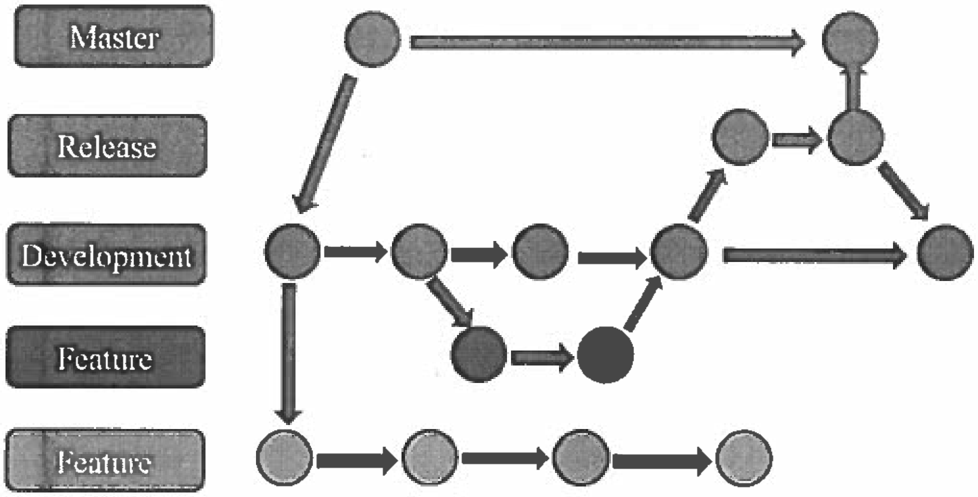
Regular Git Branching



Real World Branching Scenario

- **Master Branch:** Production ready copy
- **Development Branch:** It is a developers branch where continuous work will be done
- **Release Branch:** This branch is created from the Development Branch to make it ready for the Release and it is used for Bug tracking and documentation purpose.
- **Feature Branch:** Whenever the developers working on the new features, they use the feature branches and commit the code to that branch.

Real World Branching Scenario



Rise 'n' Shine Technologies 169

169

Git Branching

- -- Default branch is master
- -- How do we know branch name

  >> git branch

- -- Create a new branch from master

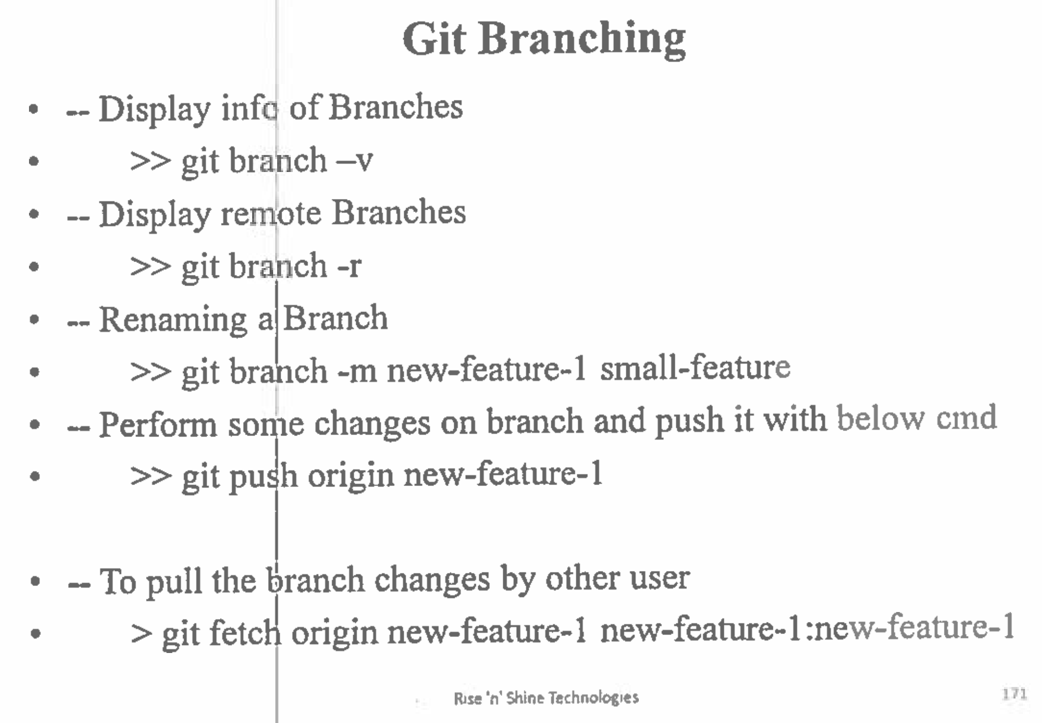
  >> git branch new-feature-1 (branch will be created from master)

  >> git checkout new-feature-1 (to switch to new-feature-1 Branch)

- -- Create a branch and switch to it immediately
- >> git checkout -b new-feature-2
- -- list all the branches including remote branches
- >> git branch -a

Rise 'n' Shine Technologies 170

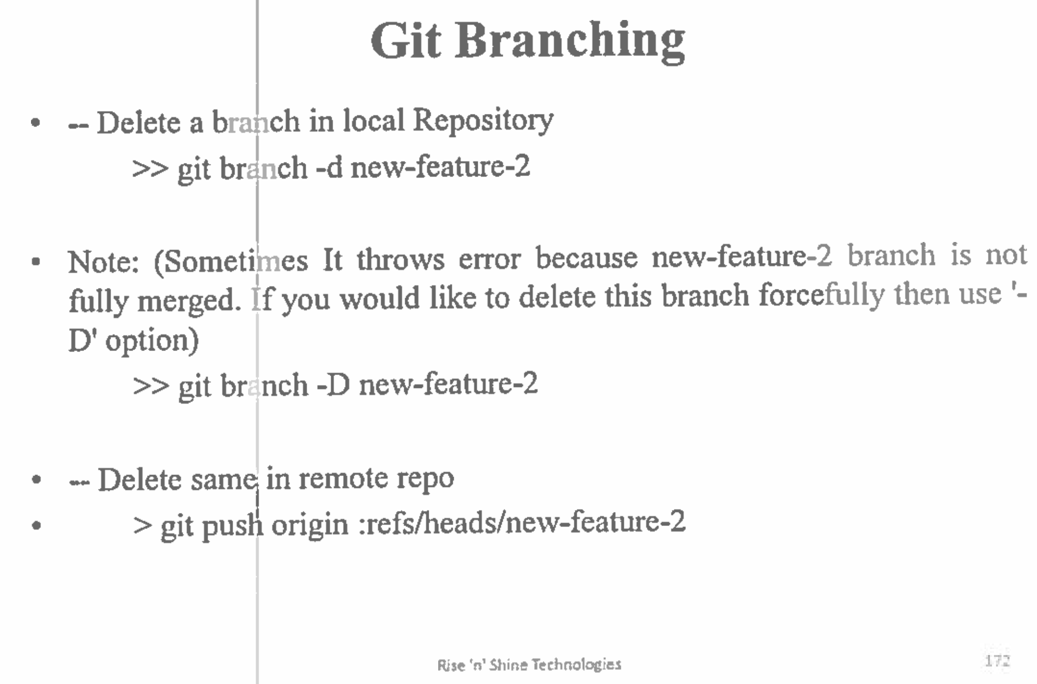
170



Git Branching

- -- Display info of Branches
- >> git branch –v
- -- Display remote Branches
- >> git branch -r
- -- Renaming a Branch
- >> git branch -m new-feature-1 small-feature
- -- Perform some changes on branch and push it with below cmd
- >> git push origin new-feature-1

- -- To pull the branch changes by other user
- > git fetch origin new-feature-1 new-feature-1:new-feature-1



Git Branching

- -- Delete a branch in local Repository
  >> git branch -d new-feature-2

- Note: (Sometimes It throws error because new-feature-2 branch is not fully merged. If you would like to delete this branch forcefully then use '-D' option)
  >> git branch -D new-feature-2

- -- Delete same in remote repo
- > git push origin :refs/heads/new-feature-2

Task on the Branches

→ Go to the master branch and perform some commits.

→ Then verify the all commits using git log command

→ Then create a new branch based on master branch

```
>> git checkout -b new-branch-1 master
>> git log --oneline
```

→ Then do some modifications in the new-branch-1 and then commit to this branch

→ Then checkout to the master branch and verify the branch history using git log command

→ Create a new branch 'new-branch-2' based on the 'new-branch-1'

```
>> git checkout -b new-branch-2 new-branch-1
```

→ Verify the logs

Git -- Tags

- -- In git tags are read only
- -- Snapshot of a particular revision number having a name and meant for read only

- -- Create a light weight tag
- `> git tag tag_1.0`
- -- Show git tag revision
- `> git show tag_1.0`
- -- List all tags
- `> git tag`

Git -- Tags

- -- Create an annotated tag with some message
- > git tag tag_1.1 -m "Release 1.0 completed"
- -- Checkout a tag
- > git checkout tag_1.0
- -- push the tag using below command as it will not be saved in server with default push
- > git push origin tag_1.1
- -- Delete a tag
- > git tag -d tag_1.0
-
- -- Delete same in remote repo
- > git push origin :refs/tags/tag_1.0

Directory & Branching Structure

- Releases:

|R1-------------| |R2--------------||R3------------| ...R20

- **Objectives:**
- 1. No data loss in production

- **Branching Strategies:**
- 1. Sequential Branching Strategy
- 2. Parallel Branching Strategy

Sequential Branching Strategy - Branches

- Releases:

  |R1-------------||R2--------------||R3------------| ...R20

- **master** --> Master Copy (Production Copy)
- **branches/** R1_release_branch
- branches/R2_release_branch
- branches/R3_release_branch

  ……

  ……

- branches/R20_release_branch

Sequential Branching Strategy - Tags

-- Total 6 QA Builds within Release 1

R1 **|----1B----2B----3B----4B----5B----6B|** R2

-- Dev needs 3 Build QA src from Release 1, then how?

For each build, master code will be copied to the Tags

R1_QA1_tag
R1_QA2_tag
R1_QA3_tag
…..
…..
R2_QA1_tag
R2_QA2_tag

2. Parallel Branching Strategy

Sequential Branching Disadvantages

```
Core1   R1 |----|---|---|---|---|---|---|  R2 |-------------------------|  R3 ------------------
           Jan                              Jun                          Dec                     Jun
```

Parallel Branching Advantages

```
        R1 |----|---|---|---|---|---|---|  R2
           Jan                              Jun
                  R2 |----|---|---|---|---|---|---|---|
                      Mar                            Sep
                                  R3 |----|---|---|---|---|---|---|---|
                                      May                              Dec
```

2. Parallel Branching Strategy

- **Objectives:**
  - 1. No data loss in production
  - 2. Branch is the dev code where check in and check out
- **Before Deploying/Releasing:**
  - No data Loss / Overwriting code in prod environment
- **After Deploying/Releasing:**
  - Make your master = Prod Code base (Release branch)

2. Parallel Branching Strategy

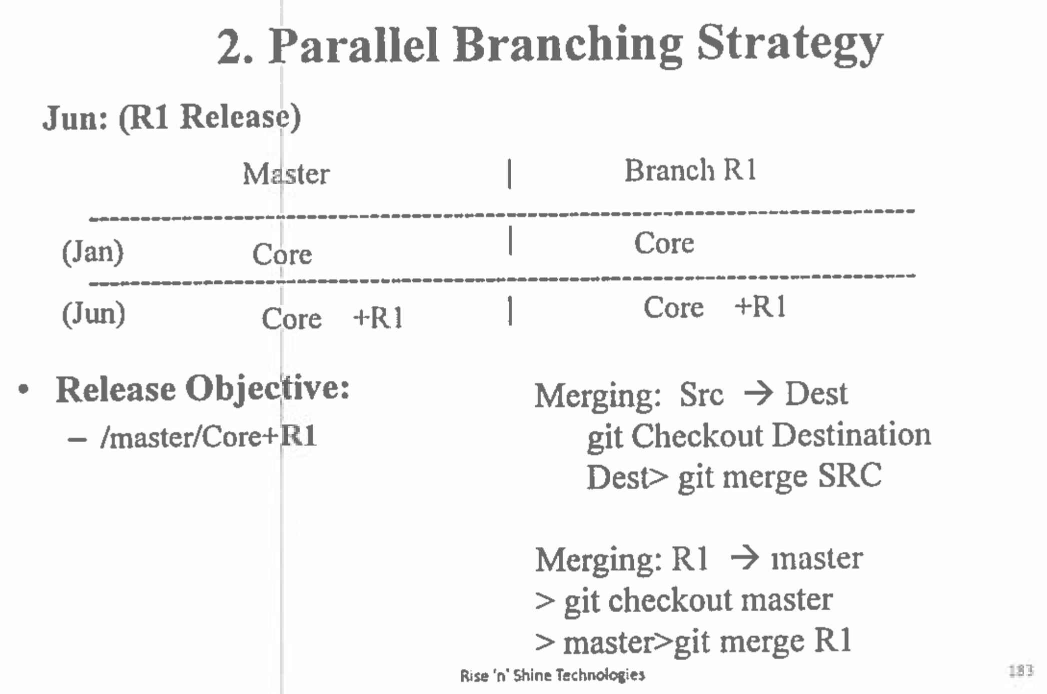
Objectives:

→No Data Loss in Production Env
→Master always should equivalent to Prod Env

	Master	Prod
(Jan)	Core	Core
(Jun)	Core +R1	Core +R1
(Sep)	Core +R1 +R2	Core +R1 +R2
(Dec)	Core +R1 +R2 +R3	Core +R1 +R2 +R3

2. Parallel Branching Strategy

- **Jan R1:** (Master and core should be same)
  - /master/Core
  - /branches/R1/Core
  - Master> git branch R1
- **Mar R2:** (master and core should be same)
  - /master/Core
  - /branches/R2/Core
  - Master > git branch R2
- **May R3:** (master and core should be same)
  - /master/Core
  - /branches/R3/Core
  - Master > git branch R3



2. Parallel Branching Strategy

Jun: (R1 Release)

	Master	Branch R1
(Jan)	Core	Core
(Jun)	Core +R1	Core +R1

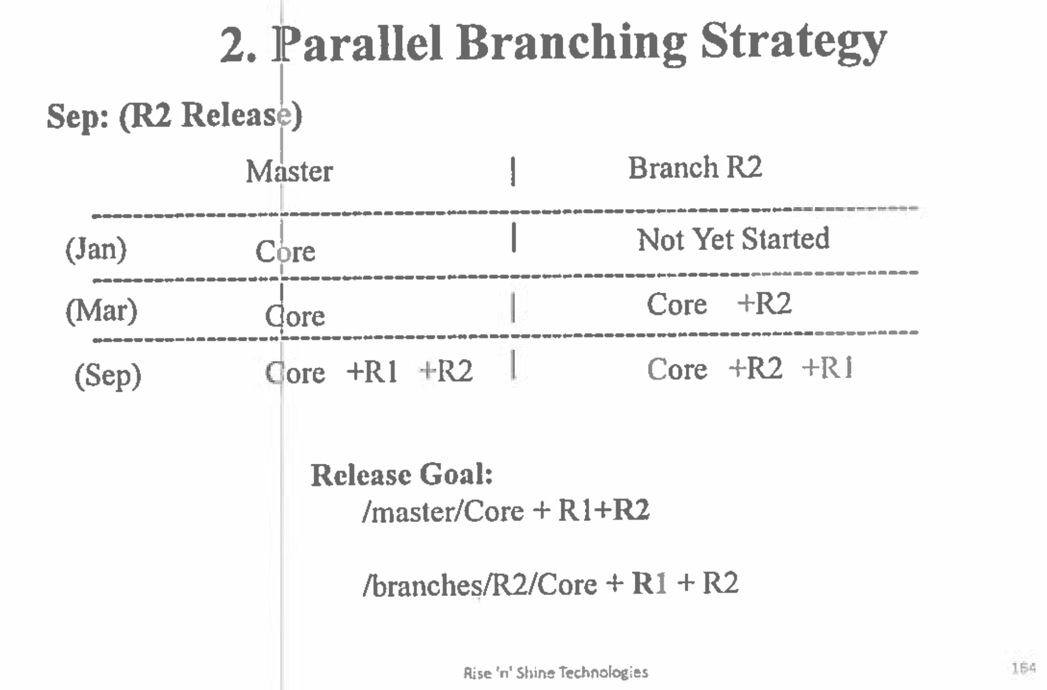
- **Release Objective:**
  - /master/Core+R1

Merging: Src → Dest
git Checkout Destination
Dest> git merge SRC

Merging: R1 → master
> git checkout master
> master>git merge R1

Rise 'n' Shine Technologies 183

183



2. Parallel Branching Strategy

Sep: (R2 Release)

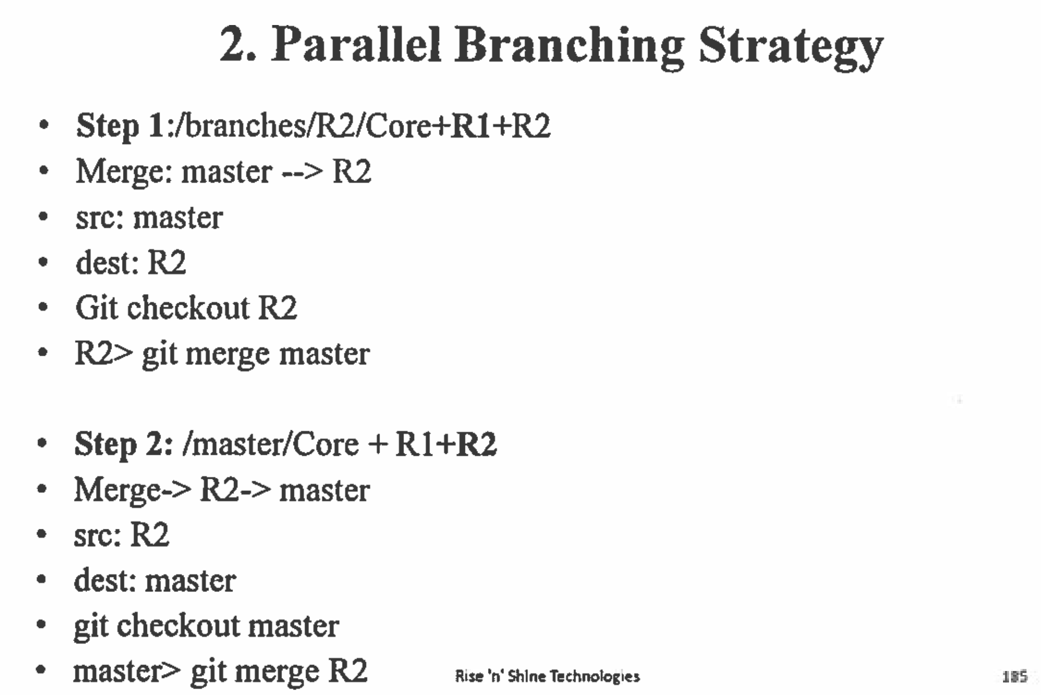
	Master	Branch R2
(Jan)	Core	Not Yet Started
(Mar)	Core	Core +R2
(Sep)	Core +R1 +R2	Core +R2 +R1

Release Goal:
/master/Core + R1+R2

/branches/R2/Core + R1 + R2

Rise 'n' Shine Technologies 184

184

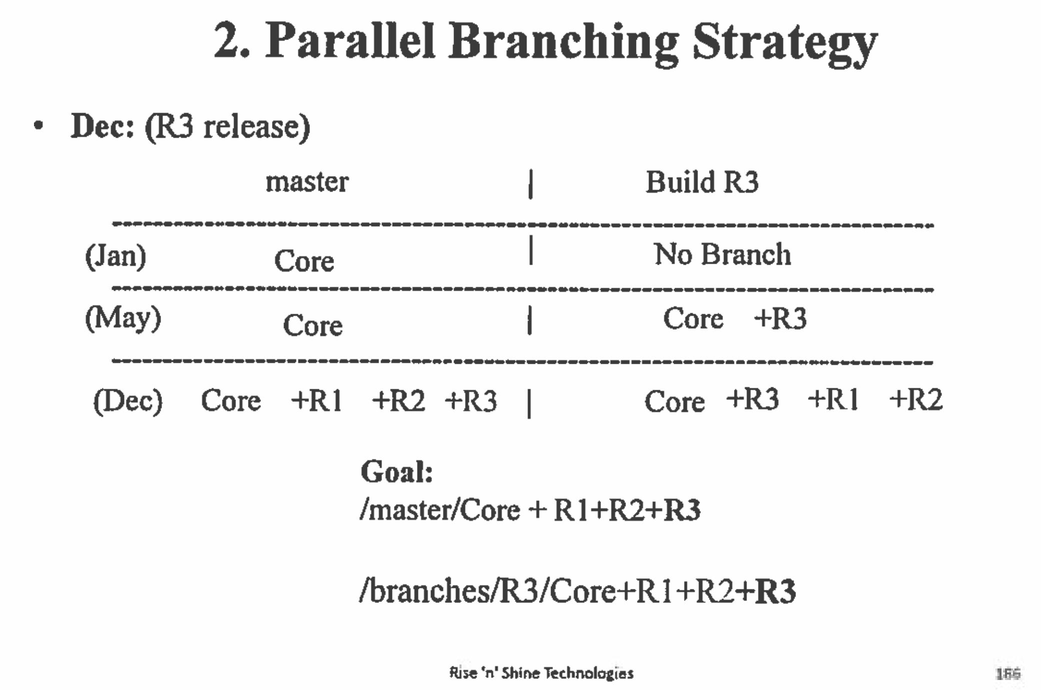


2. Parallel Branching Strategy

- **Step 1:**/branches/R2/Core+R1+R2
- Merge: master --> R2
- src: master
- dest: R2
- Git checkout R2
- R2> git merge master

- **Step 2:** /master/Core + R1+**R2**
- Merge-> R2-> master
- src: R2
- dest: master
- git checkout master
- master> git merge R2

185



2. Parallel Branching Strategy

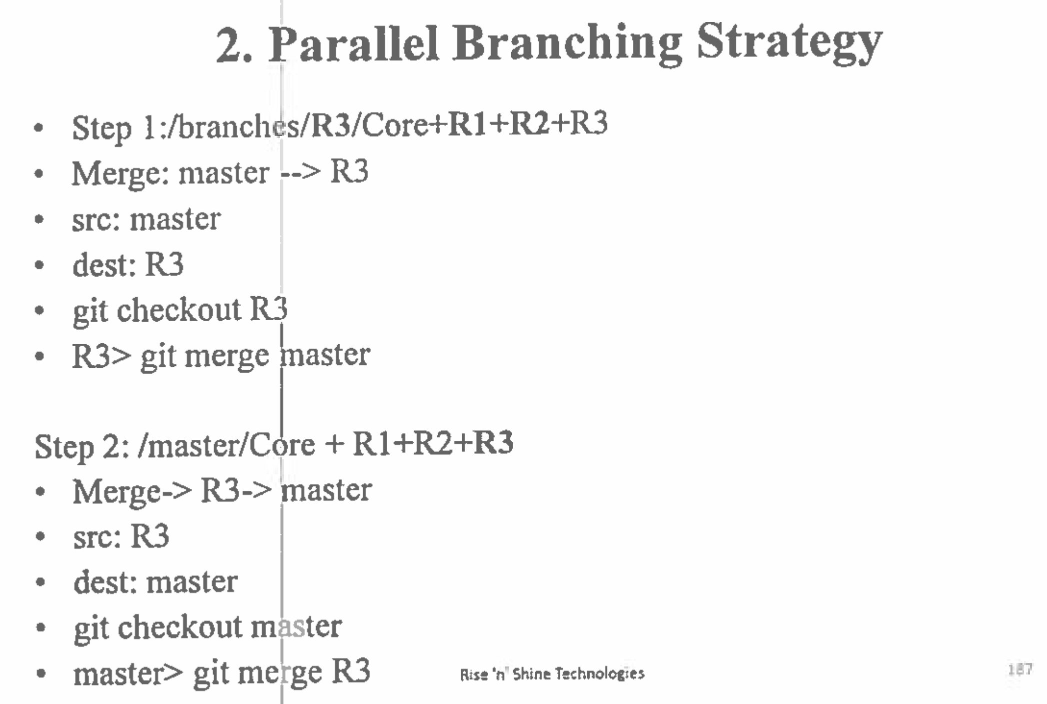
- **Dec:** (R3 release)

	master	Build R3
(Jan)	Core	No Branch
(May)	Core	Core +R3
(Dec)	Core +R1 +R2 +R3	Core +R3 +R1 +R2

Goal:
/master/Core + R1+R2+**R3**

/branches/R3/Core+R1+R2+**R3**

186



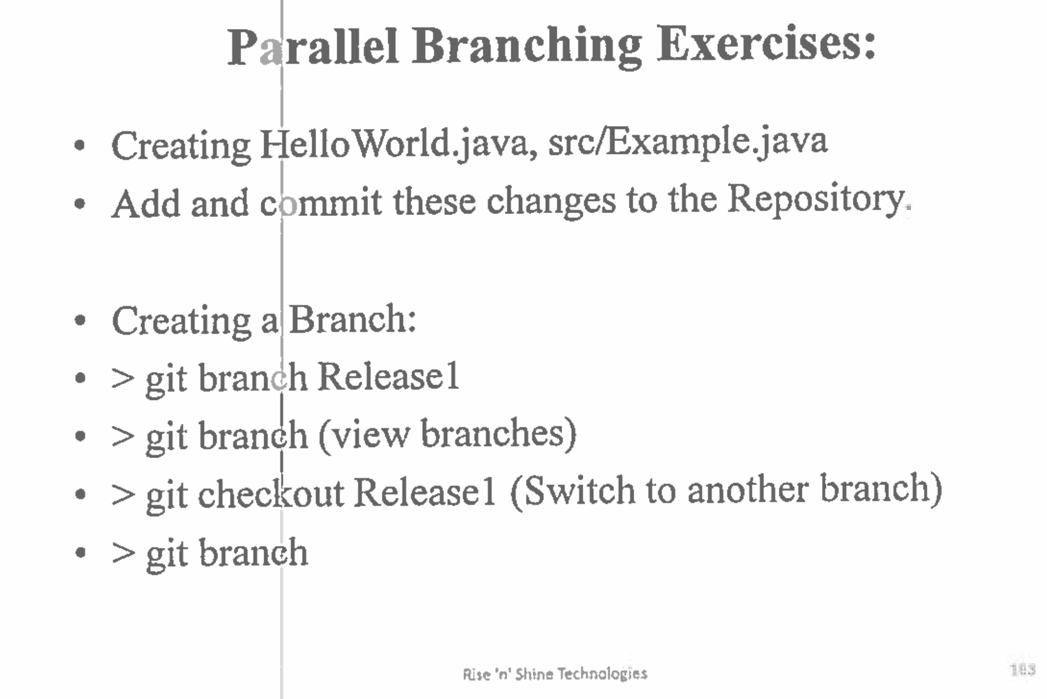
2. Parallel Branching Strategy

- Step 1:/branches/R3/Core+R1+R2+R3
- Merge: master --> R3
- src: master
- dest: R3
- git checkout R3
- R3> git merge master

Step 2: /master/Core + R1+R2+R3

- Merge-> R3-> master
- src: R3
- dest: master
- git checkout master
- master> git merge R3

187



Parallel Branching Exercises:

- Creating HelloWorld.java, src/Example.java
- Add and commit these changes to the Repository.

- Creating a Branch:
- > git branch Release1
- > git branch (view branches)
- > git checkout Release1 (Switch to another branch)
- > git branch

188

Parallel Branching Exercises:

- Step 2: Modify the Helloworld.java and commit to the Release 1.
- > execute the required commands
- > git status
- > ls
- > git branch (it should be Release1)
- > git checkout master (Change to master repository)
- > cat HelloWorld.java (No changes in master)

- Step3: Create a new branch Release2 from the master and update the code to HelloWorld.java and commit the changes to the Release 2.

- -- View the git log messages (which will be displayed only that particular Release log)

Git tags -- Exercise

- 1) Create a tag

- 2) After creating a tag modify some changes src/Example.java and commit the changes to the Repository.

- 3) Then change back the Created tag and we will be having previous changes but not latest changes.

Important Commands

- -- To check the diff between staging area and git
- > git diff
- -- Diff between branches
- > git diff Release1 Release2
- -- Show changes of patchset
- > git show charset
- -- Remove untracked files (before add operation)
- -- The below command shows what are all the files will be deleted
- > git clean -n
- > git clean -f (delete them actually)

191

Important Commands

- -- Revert the unstaged changes with below command
- > git checkout filename

192

Git Stashing

- --stash stores the staged(git add) changes into a package and stores in memory, when ever u want to pull remote changes and want to apply the unstaged changes you can do this.

- --> git stash
- -- list the stashes
- git stash list
- -- Apply the stash when you wanted with below command
- git stash pop
- or you can use the id as well as shown below [pop is always latest stash]
- git stash apply stash@{0}

Git Stashing

- -- This may lead to conflicts, resolve conflicts manually
- -- Once the conflicts are resolved commit the changes
- -- Delete the stash packet
- git stash drop stash@{0}

Git Cmds - Remote Repositories

- --Default will be origin but we can add multiple repositories as well
- git remote add reponame username@github.com/project/repo
- git remote show origin (which repository it has been pointed)
- git remote –v