Q1: What is Git fork? What is difference between fork, branch and clone?

- A **fork** is a remote, server-side copy of a repository, distinct from the original. A fork isn't a Git concept really, it's more a political/social idea.
- A **clone** is not a fork; a clone is a local copy of some remote repository. When you clone, you are actually copying the entire source repository, including all the history and branches.
- A **branch** is a mechanism to handle the changes within a single repository in order to eventually merge them with the rest of code. A branch is something that is within a repository. Conceptually, it represents a thread of development.

Q2: What's the difference between a "pull request" and a "branch"?

-

- A **branch** is just a separate version of the code.
- A **pull request** is when someone take the repository, makes their own branch, does some changes, then tries to merge that branch in (put their changes in the other person's code repository).

Q3: What is the difference between "git pull" and "git fetch"?

In the simplest terms, `git pull` does a `git fetch` followed by a `git merge`.

- When you use `pull`, Git tries to automatically do your work for you. **It is context sensitive**, so Git will merge any pulled commits into the branch you are currently working in. `pull` **automatically merges the commits without letting you review them first**. If you don't closely manage your branches, you may run into frequent conflicts.
- When you `fetch`, Git gathers any commits from the target branch that do not exist in your current branch and **stores them in your local repository**. However, **it does not merge them with your current branch**. This is particularly useful if you need to keep your repository up to date, but are working on something that might break if you update your files. To integrate the commits into your master branch, you use `merge`.

Q4: How to revert previous commit in git?

Say you have this, where C is your HEAD and (F) is the state of your files.

```
   (F)
A-B-C
    ↑
  master
```

- To nuke changes in the commit:

```
git reset --hard HEAD~1
```

Now B is the HEAD. Because you used --hard, your files are reset to their state at commit B.

- To undo the commit but keep your changes:

```
git reset HEAD~1
```

Now we tell Git to move the HEAD pointer back one commit (B) and leave the files as they are and `git status` shows the changes you had checked into C.

- To undo your commit but leave your files and your index

```
git reset --soft HEAD~1
```

When you do `git status`, you'll see that the same files are in the index as before.

Q5: What is "git cherry-pick"?

The command git *cherry-pick* is typically used to introduce particular commits from one branch within a repository onto a

different branch. A common use is to forward- or back-port commits from a maintenance branch to a development branch.

This is in contrast with other ways such as merge and rebase which normally apply many commits onto another branch.

Consider:

```
git cherry-pick <commit-hash>
```

Q6: Explain the advantages of Forking Workflow

The **Forking Workflow** is fundamentally different than other popular Git workflows. Instead of using a single server-side repository to act as the "central" codebase, it gives every developer their own server-side repository. The Forking Workflow is most often seen in public open source projects.

The *main advantage* of the Forking Workflow is that contributions can be integrated without the need for everybody to push to a single central repository that leads to a clean project history. Developers push to their own server-side repositories, and only the project maintainer can push to the official repository.

When developers are ready to publish a local commit, they push the commit to their own public repository—not the official one. Then, they file a pull request with the main repository, which

lets the project maintainer know that an update is ready to be integrated.

Q7: Tell me the difference between HEAD, working tree and index, in Git?

- The **working tree/working directory/workspace** is the directory tree of (source) files that you see and edit.
- The **index/staging area** is a single, large, binary file in /.git/index, which lists all files in the current branch, their sha1 checksums, time stamps and the file name - it is not another directory with a copy of files in it.
- **HEAD** is a reference to the last commit in the currently checked-out branch.

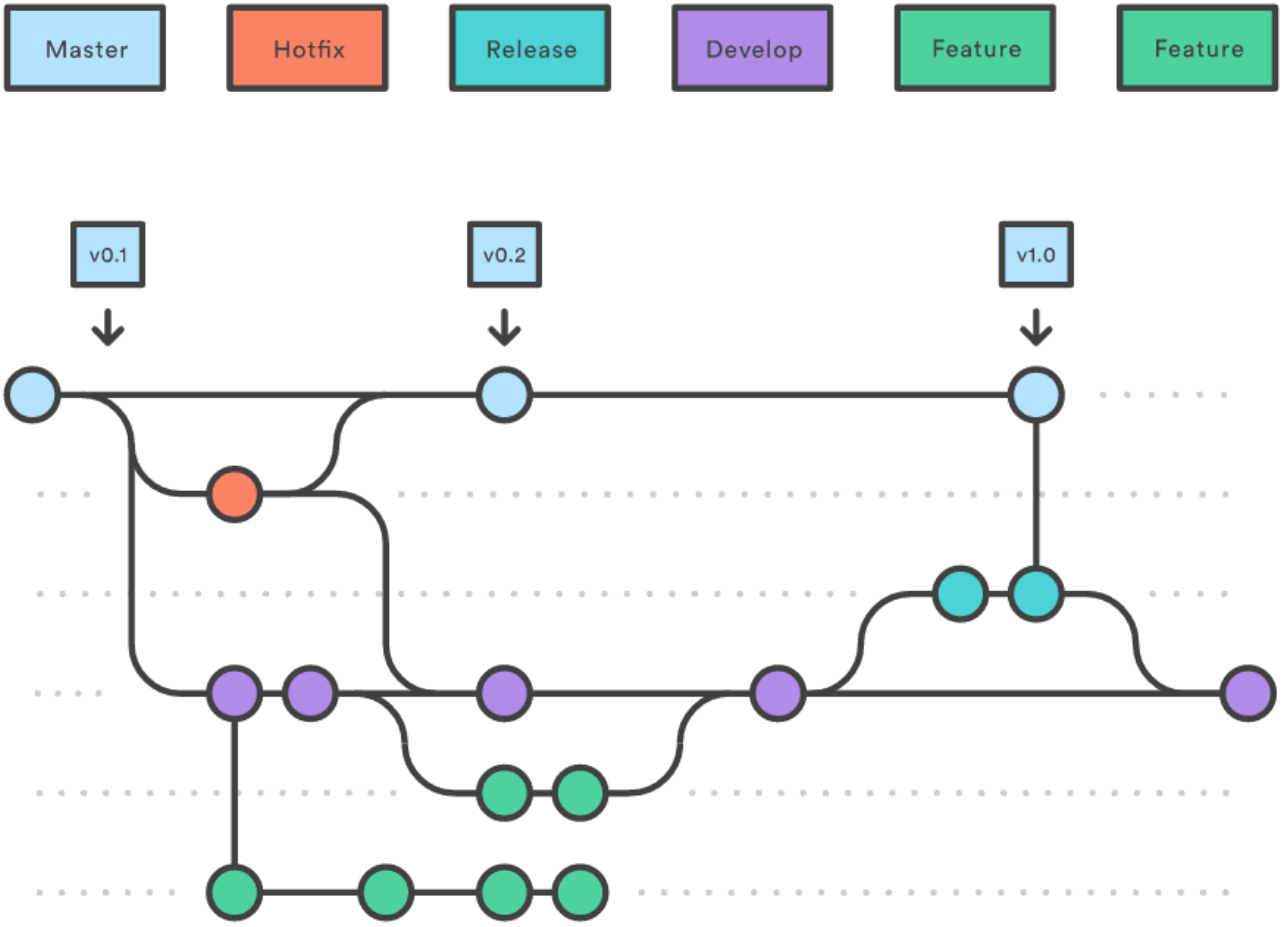
Q8: Could you explain the Gitflow workflow?

Gitflow workflow employs two parallel *long-running* branches to record the history of the project, `master` and `develop`:

- **Master** - is always ready to be released on LIVE, with everything fully tested and approved (production-ready).
    - **Hotfix** - Maintenance or "hotfix" branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on `master` instead of `develop`.
- **Develop** - is the branch to which all feature branches are merged and where all tests are performed. Only when

everything's been thoroughly checked and fixed it can be merged to the `master`.

- **Feature** - Each new feature should reside in its own branch, which can be pushed to the `develop` branch as their parent one.



Q9: When should I use "git stash"?

The `git stash` command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy.

Consider:

```
$ git status
On branch master
Changes to be committed:
new file: style.css
Changes not staged for commit:
modified: index.html
$ git stash
Saved working directory and index state WIP on master: 5002d47 our new homepag
HEAD is now at 5002d47 our new homepage
$ git status
On branch master
nothing to commit, working tree clean
```

The one place we could use stashing is if we discover we forgot something in our last commit and have already started working on the next one in the same branch:

```
# Assume the latest commit was already done
# start working on the next patch, and discovered I was missing something

# stash away the current mess I made
$ git stash save

# some changes in the working dir

# and now add them to the last commit:
$ git add -u
$ git commit --ammend

# back to work!
$ git stash pop
```

Q10: How to remove a file from git without removing it from your file system?

If you are not careful during a `git add`, you may end up adding files that you didn't want to commit. However, `git rm` will remove it from both your staging area (index), as well as your file system (working tree), which may not be what you want.

Instead use `git reset`:

```
git reset filename          # or
echo filename >> .gitingore # add it to .gitignore to avoid re-adding it
```

This means that `git reset <paths>` is the opposite of `git add <paths>`.

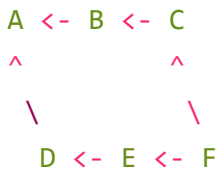
Q11: When do you use "git rebase" instead of "git merge"?

Both of these commands are designed to integrate changes from one branch into another branch - they just do it in very different ways.

Consider before merge/rebase:

```
A <- B <- C    [master]
^
 \
  D <- E       [branch]
```

after `git merge master`:



after `git rebase master`:

With rebase you say to use another branch as the new base for your work.

When to use:

1. If you have any doubt, use merge.
2. The choice for rebase or merge based on what you want your history to look like.

More factors to consider:

1. **Is the branch you are getting changes from shared with other developers outside your team (e.g. open source, public)?** If so, don't rebase. Rebase destroys the branch

and those developers will have broken/inconsistent repositories unless they use `git pull --rebase`.

2. **How skilled is your development team?** Rebase is a destructive operation. That means, if you do not apply it correctly, you could lose committed work and/or break the consistency of other developer's repositories.
3. **Does the branch itself represent useful information?** Some teams use the *branch-per-feature* model where each branch represents a feature (or bugfix, or sub-feature, etc.) In this model the branch helps identify sets of related commits. In case of *branch-per-developer* model the branch itself doesn't convey any additional information (the commit already has the author). There would be no harm in rebasing.
4. **Might you want to revert the merge for any reason?** Reverting (as in undoing) a rebase is considerably difficult and/or impossible (if the rebase had conflicts) compared to reverting a merge. If you think there is a chance you will