**Basic Branching and Merging:**

* 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. You’ll do the following:

* 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.

**Basic Branching:**

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

You’ve decided that you’re going to work on issue #53 in whatever issue-tracking system your company uses. 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

Switched to a new branch "iss53"

This is shorthand for:

* git branch iss53

$ git checkout iss53

1. 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 'added a new footer [issue 53]'

The iss53 branch has moved forward with your work

1. 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, and 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.

However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you’re checking out, Git won’t let you switch branches. It’s best to have a clean working state when you switch branches. There are ways to get around this (namely, stashing and commit amending) that we’ll cover later on, in [Stashing and Cleaning](https://git-scm.com/book/en/v2/ch00/_git_stashing). For now, let’s assume you’ve committed all your changes, so you can switch back to your master branch:

* git checkout master

Switched to branch 'master'

At this point, your project working directory is exactly the way it was before you started working on issue #53, and you can concentrate on your hotfix. This is an important point to remember: 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.

1. 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 'fixed the broken email address'

[hotfix 1fb7853] fixed the broken email address

1 file changed, 2 insertions(+)

Hotfix branch based on master

1. 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(+)

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.

master is fast-forwarded to hotfix

1. 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).

1. 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 'finished the new footer [issue 53]'

[iss53 ad82d7a] finished the new footer [issue 53]

1 file changed, 1 insertion(+)

Work continues on iss53

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.

**Basic Merging**

1. Suppose you’ve decided that your issue #53 work is complete and ready to be merged into your master branch. In order to do that, 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(+)

This looks a bit different than the hotfix merge you did earlier. In this case, your development history has diverged from some older point. Because the commit on the branch you’re on isn’t a direct ancestor of the branch you’re merging in, Git has to do some work. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.

Three snapshots used in a typical merge

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, and is special in that it has more than one parent.

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

* git branch -d iss53