Getting Started

Follow this guide to set up the Eclipse IDE and the Git version control system on your computer and learn how to use them in 6.005.

Learning Java

6.005 requires you to get up to speed quickly with the basics of Java. If you are not familiar with Java:

- review **materials for learning Java (java.html)**
- review **Reading 2 (../classes/02-basic-java/)**

Step 1: Install software

You need to install the following software on your laptop for 6.005:

- **JDK 8 (//www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html)** (for Windows, Linux, or OS X): From this page, download *Java SE Development Kit 8u71* (you don't need the demos and samples).

  The latest version of Java is required (either 8u71 or 8u72).

- **Eclipse Mars.1 (//www.eclipse.org/downloads/packages/eclipse-ide-java-developers/mars1)** : Download *Eclipse IDE for Java Developers* (download links are on the right side of the page). For **Windows** users, see determining whether you're running a 32- or 64-bit OS (//support.microsoft.com/kb/827218) . Unpack the downloaded .zip or .tar.gz file and run Eclipse.

  The latest version of Eclipse is required (Mars.1, a.k.a. 4.5.1).

- **Git (//www.git-scm.com/)** : We will be using the command-line interface to Git.

  If you already have Git installed, you do not need to install the latest version. **Windows** users should look for Git Bash. **OS X & Linux** users should try running `git` in a terminal.

  The Git site should prompt you to download the appropriate version. Follow the instructions in the README file in the downloaded .zip or .dmg file.

  Windows : choose "use Git from the windows command prompt", "checkout windows-style, commit unix-style line endings", and select to add a shortcut to Desktop during installation (this creates a Git Bash shortcut which you will use for all Git commands).

  OS X : if you receive an "unidentified developer" warning, right-click the .pkg file, select *Open* , then click *Open* .

Step 2: Configure Eclipse

The Eclipse integrated development environment (IDE) is a powerful, flexible, complicated, and occasionally frustrating set of tools for developing and debugging programs, especially in Java.

When you run Eclipse, you will be prompted for a "workspace" directory, where Eclipse will store its configuration and metadata. The default location is a directory called `workspace` in your home directory. You should not run more than one copy of Eclipse at the same time with the same workspace.

The first time you run Eclipse, it will show you a welcome screen. Click the "Workbench" button and you're ready to begin.

You can store your code in the workspace directory, but this is not recommended. On the left side of your Eclipse window is the Package Explorer, which shows you all the projects in your workspace. **The Package Explorer looks like a file browser, but it is not.** It rearranges files and folders, includes things that are not files or folders, and can include projects stored **anywhere on disk** that have been added (but not copied into) to the workspace.

1. Open Eclipse preferences.

   Windows & Linux : go to *Window* → *Preferences* .

   OS X : go to *Eclipse* → *Preferences* .

2. Make sure Eclipse is configured to use **Java 8** .

   1. In preferences, go to *Java* → *Installed JREs* . Ensure that "Java SE 8" or "1.8.0_71" is the only one checked. If it's not listed, click Search.

   2. Go to *Java* → *Compiler* and set "Compiler compliance level" to 1.8. Click OK and Yes on any prompts.

3. Make sure **assertions are always on** . Assertions are a great tool for keeping your code safe from bugs, but Java has them off by default.

   In preferences, go to *Java → Installed JREs* . Click “Java SE 8”, click “Edit…”, and in the “Default VM arguments” box enter: `-ea` (which stands for *enable assertions*).

4. **Tab policy** . Configure your editor to use spaces instead of tabs, so your code looks the same in all editors regardless of how that editor displays tab characters.

   In preferences, go to *Java → Code Style → Formatter* . Click the “Edit…” button next to the active profile. In the new window, change the Tab policy to “Spaces only.” Keep the Indentation size and Tab size at 4. To save your changes, enter a new “Profile name” at the top of the window and click OK.

The **6.005 Eclipse FAQ (eclipse.html)** has some tips and tricks to help you make the most of Eclipse.

Step 3: Open the command line

One thing that makes learning Git harder for many students is that it’s a command-line program. If you’re not familiar with the command-line, this can be confusing.

A command-line is just an interface to your computer, totally analogous to the Finder or Windows Explorer, except that it’s text-based. As the name implies, you interact with it through “commands” — each line of input begins with a command and might have zero or more arguments, separated by spaces. The command-line keeps track of what directory (folder) you’re in, which is important to many of the commands you might be running.

On **OS X and Linux** , open the **Terminal** application.

On **Windows** , Git for Windows (//git-scm.com/) includes **Git Bash** . Run Git Bash to open a terminal where you can run all the commands below, in addition to Git commands.

Common commands

- **`cd`** (stands for “change directory”)

  Changes the current directory. In you’re in a directory that has a subdirectory called `hello` , then `cd hello` moves into that subdirectory.

  Use `cd ..` to move to the parent directory of your current directory.

- **`pwd`** (“print working directory”)

  Prints out the current directory, if you’re not sure where you are.

  On a well-configured system, your current directory is displayed as part of the *prompt* that the system shows when it’s ready to receive a command. If that’s not the case on your system, post on Piazza to get help configuring your prompt.

- **`ls`** (“list”)

  Lists the files in the current directory.

  Use `ls -l` for extra information (a “long” listing) about the files. Use `ls -a` (stands for “all”) to show hidden files, which are files and subdirectories whose names begin with a period.

- **`mkdir`** (“make directory”)

  Creates a new directory in the current directory. To create a directory called `goodbye` , use `mkdir goodbye` .

- ***up arrow* and *down arrow***

  Use *up arrow* to put the command you just ran back on the command line. You can now edit that command to fix a typo, or just press *enter* to run it again.

  Use the up and down arrow keys to navigate through your history of commands, so you never have to re-type a long command line.

Step 4: Configure Git

Before using Git, we’ll do some required setup and make it behave a little nicer.

1. **Who are you?**

   Every Git commit includes the author’s name and e-mail. Make sure Git knows your name and email by running these two commands:

   ```
   git config --global user.name "Your Name"
   git config --global user.email username@mit.edu
   ```

2. **Editing commit messages.**

   Every Git commit has a descriptive message, called the commit message. Pick one of the two options below to set up your commit message editor.

Option 1: set up an easy-to-use editor

OS X and Linux: use nano

nano (//www.nano-editor.org) is a simple text editor. It does not come with the Windows version of Git, so Windows users should choose a different option.

To see if you have nano, try running:

```
nano
```

in the terminal. The result should be a simple editor with instructions at the bottom of the screen; quit with `ctrl-X` . If that worked:

```
git config --global core.editor nano
```

will configure Git to use the nano editor. The commands to use the text editor (like copy, paste, quit, etc.) will be shown on the bottom of the screen. The `^` symbol represents the `ctrl` key. For example, you can press `ctrl-O` to save (nano calls it "write out") and then `ctrl-X` to quit.

Windows: use Notepad

You can change the default editor to Notepad with:

```
git config --global core.editor notepad
```

If you prefer to edit your commit messages in the terminal, choose Option 2 instead.

Option 2: use Vim

On OS X and Windows, your default editor will be Vim (//www.vim.org) .

On Linux, the default editor depends on your distribution.

Vim is a popular text editor, but it's tricky to use.

Before making your first commit, try running:

```
vim
```

in the terminal.

You start in a mode called "normal mode". You can't immediately type anything into the file!

In order to start typing, press `i` (stands for "insert"). This will bring you to "insert mode", so named because in this mode you can type text into the file.

When you are done typing, press `esc` . This will bring you back to "normal mode".

Once you're back in normal mode, you can type commands that start with `:` .

To save your work, type `:w` (stands for "write") and press return.

To exit (quit) Vim, type `:q` and press return.

To save and quit in one command, combine them: type `:wq` and press return.

3. **Add some color.**

   Out of the box, it can be hard to see and understand all the output that git prints out at you. One way to make it a little easier is to add some color. Run the following commands to make your git output colorful:

   ```
   git config --global color.branch auto
   git config --global color.diff auto
   git config --global color.interactive auto
   git config --global color.status auto
   git config --global color.grep auto
   ```

4. **LOL.**

   As we'll see in the next step of this guide, `git log` is a command for looking at the history of your repository.

   To create a special version of `git log` that summarizes the history of your repo, let's create a `git lol` *alias* using the command **(all on one line)** :

   ```
   git config --global alias.lol "log --graph --oneline --decorate --color --all"
   ```

   Now, in any repository you can use:

   ```
   git lol
   ```

   to see an ASCII-art graph of the commit history.

Step 5: Learn the Git workflow

What is Git?

Git is a version control system (VCS). The *Pro Git* book (//git-scm.com/book) describes what Git is used for:

> What is version control, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. […] It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover.

Some of the most important Git concepts:

- **repository:** A folder containing all the files associated with a project (e.g. a 6.005 problem set or team project), as well as the entire history of *commits* to those files.

- **commit (or "revision"):** A snapshot of the files in a repository at a given point in time.
- **add (or "stage"):** Before changes to a file can be *committed* to a repository, the files in question must be *added* or *staged* (before each commit). This lets you commit changes to only certain files of your choosing at a time, but can also be a bit of a pain if you accidentally forget to add all the files you wanted to commit before committing.
- **clone:** Since git is a "distributed" version control system, there is no concept of a centralized git "server" that holds the latest official version of your code. Instead, developers "clone" remote repositories that contain files they want access to, and then commit to their local clones. Only when they *push* their local commits to the original remote repository are other developers able to see their changes.
- **push:** The act of sending your local commits to a remote repository. Again, until you add, commit, *and* push your changes, no one else can see them.
- **pull:** The act of retrieving commits made to a remote repository and writing them into your local repository. This is how you are able to see commits made by others after the time at which you made an initial clone.

Cloning

You start working with Git repos in 6.005 by cloning a remote repository into a local repository on your computer.

To do this, open the terminal (use Git Bash on Windows) and use the `cd` command to change to the directory where you would like to store your code. Then run:

```
git clone URI-of-remote-repo
```

or

```
git clone URI-of-remote-repo project-name
```

Replace `URI-of-remote-repo` with the location of the remote repository, and replace `project-name` with the appropriate project name, like `ps0`. The result will be a new directory `project-name` with the contents of the repository. This is your local repository.

Cloning problem sets : for each problem set in 6.005, you will have a Git repository.

Initially this remote repository only contains some template code.

To start working on the problem set, you will *clone* that repository onto your machine.

As you complete each part of the problem set, you will *commit* your changes to the local repository and then *push* them to the remote repository.

When the time comes for grading your assignment, we will clone the remote repository and look at the last commit you made *and pushed there* before the deadline.

Getting the history of the repository

After you have cloned a repository, you should navigate into the repository on your command prompt using `cd`. This lets you run `git` commands on the repository.

For example, you can see the last commit on the repository using `git show`. This will show you the commit message as well as all the modifications.

You can see the list of all the commits in the repository (with their commit messages) using `git log`.

Long output : if `git show` or `git log` generate more output than fits on one page, you will see a colon (`:`) symbol at the bottom of the screen. You will not be able to type another command! Use the arrow keys to scroll up and down, and quit the output viewer by pressing `q`.

Commit IDs : every Git commit has a unique ID, the hexadecimal numbers you see in `git log` or `git show`. The commit ID is a unique cryptographic hash of the contents of that commit. Every commit, not just within your repository but within the *universe* of all Git repositories, has a unique ID (with extremely high probability).

You can reference a commit by its ID (usually just by the first few characters). This is useful with a command like `git show`, where you can look at a particular commit rather than only the most recent one.

Creating a commit

The basic building block of data in Git is called a "commit". A commit represents some change to one or more files (or the creation of one or more files).

When you change a file or create a new file, that change is not part of the repository. Adding it takes two steps. First, run:

`git add file.txt` (where `file.txt` is the file you want to add)

You'll either need to run that command from the same directory as the file, or include directory names in the file path.

This *stages* the file. Second, once you've staged all your changes, run:

```
git commit
```

This will pop up the editor for your *commit message*. When you save and close the editor, the commit will be created.

Getting the status of your repository

Git has some nice commands for seeing the status of your repository.

The most important of these is `git status`. Run it any time to see which files Git sees have been modified and are still unstaged and which files have been modified and staged (so that if you `git commit` those changes will be included in the commit). Note that the same file might have both staged and unstaged changes, if you changed the file more after running `git add`.

When you have unstaged changes, you can see what the changes were (relative to the last commit) by running `git diff`. Note that this will *not* include changes that were staged (but not committed). You can see those by running `git diff --staged`.

Pushing

After you've made some commits, you'll want to push them to a remote repository. In 6.005, you should have only one remote repository to push to, called `origin`. To push to it, you run the command:

```
git push origin master
```

The `origin` in the command specifies that you're pushing to the `origin` remote. By convention, that's the remote repository you cloned from.

The `master` refers to the `master` branch. We won't use branches in 6.005. `master` is Git's default branch name, so all our commits will be on `master`, and that's the branch we want to push.

Once you run this, you will be prompted for your password and hopefully everything will push. You'll get a line like this:

```
a67cc45..b4db9b0  master -> master
```

Merges

Sometimes, when you try to push, things will go wrong. You might get an output like this:

```
! [rejected]      master -> master (non-fast-forward)
```

What's going on here is that Git won't let you push to a repository unless all your commits come after all the ones already in your remote repository. If you get an error message like that, it means that there is a commit in your remote repository that you don't have in your local one (on a project, probably because a teammate pushed before you did). If you find yourself in this situation, you have to pull first and then push.

Pulling

To perform a pull, you should run `git pull`. When you run this, Git actually does two things:

1. It downloads the changes and stores them in its internal state. At this point, the repository doesn't look any different, but it knows what the state of the remote repository is and what the state of your local repository is.
2. It incorporates the changes from the remote repository into the local repository by *merging*, described below.

Merging

If you made some changes to your repository and you're trying to incorporate the changes from another repository, you need to merge them together somehow. In terms of commits, what actually needs to happen is that you have to create a special *merge commit* that combines both changes. How this process actually happens depends on the changes.

If you're lucky, then the changes you made and the changes that you downloaded from the remote repository don't conflict. For example, maybe you changed one file and your project partner changed another. In this case, it's safe to just include both changes. Similarly, maybe you changed different parts of the same file. In these cases, Git can do the merge automatically. When you run `git pull`, it will pop up an editor as if you were making a commit: this is the commit message of the merge commit that Git automatically generated. Once you save and close this editor, the merge commit will be made and you will have incorporated both changes. At this point, you can try to `git push` again.

Merge conflicts

Sometimes, you're not so lucky. If the changes you made and the changes you pulled edit the same part of the same file, Git won't know how to resolve it. This is called a *merge conflict*. In this case, you will get an output that says `CONFLICT` in big letters. If you run `git status`, it will show the conflicting files with the label `Both modified`. You now have to edit these files and resolve them by hand.

First, open the files in Eclipse. The parts that are conflicted will be really obviously marked with obnoxious `<<<<<<<<<<<<<<<<<<`, `==================`, and `>>>>>>>>>>>>>>>>>>` lines. Everything between the `<<<<` and the `====` lines are the changes you made. Everything between the `====` and the `>>>>` lines are the

changes you pulled in. It's your job to figure out how to combine these. The answer will of course depend on the situation. Maybe one change logically supercedes the other, or maybe they can be merged somehow. You should edit the file to your satisfaction and remove the `<<<<` / `====` / `>>>>` markers when you're done.

Once you have resolved all the conflicts (note that there can be several conflicting files, and also several conflicts per file), `git add` all the affected files and then `git commit` . You will have an opportunity to write the merge commit message (where you should describe how you did the merge). Now you should be able to push.

Avoid merges and merge conflicts:

Pull before you start working

Before you start working, **always** `git pull` . That way, you'll be working from the latest version of your code, and you'll be less likely to have to perform a merge later.

Reverting to previous versions

If you'd like to practice using the version history to undo a change:

We will revisit Git and learn more about version control in future classes. If you've installed the software, set up Eclipse, and completed the GitStream exercises above, you're ready to move on to Problem Set 0 (../psets/ps0/) .

Have fun in 6.005!

MIT EECS