

6.005 — Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |  
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>)  
Spring 2016

General Information & FAQ

Objectives

Classes, readings, and nanoquizzes

Problem sets, beta & final, and slack days

Projects and quizzes

Collaboration and public sharing

Grading

# General Information & FAQ

## Objectives

See the 6.005 Curricular Goals Map (<https://gmap.csail.mit.edu/gmap/public.html?focus=6.005>) giving a dynamic graphical display connecting the class outcomes with the outcomes of other subjects in the Course 6 curriculum.

## Classes, readings, and nanoquizzes

**Class meetings.** There are two 90-minute class meetings each week on Monday and Wednesday and one 1-hour class meeting each week on Friday. You are expected to attend all class meetings and to participate actively in exercises and discussions.

**Laptops required.** Classes will include multiple-choice questions and programming exercises that require a laptop.

**Readings.** Most classes will have a reading that you must read before coming to class. There is no course textbook.

**Nanoquizzes.** Every class meeting will begin with a short quiz on the required reading for the class, plus recent class meetings. Nanoquizzes are closed-book and closed-notes, with a 3-minute time limit. There will be approximately 25 nanoquizzes. Your lowest 5 nanoquiz grades will be automatically dropped, and you can make up missed nanoquizzes or low grades. More info: **details of nanoquiz grading and makeup (nanoquiz-grading.html)** .

### # Why is 6.005 structured the way it is?

*Practice and feedback* are key to learning, and 6.005 is structured to provide as many opportunities for practice and feedback as possible. That means we don't want to spend class time on lectures, we want to spend it on exercises to practice the concepts and skills of building software.

Citations: Wieman et al., *Course Transformation Guide*

([http://www.cwsei.ubc.ca/resources/files/CourseTransformationGuide\\_CWSEI\\_CU-SEI.pdf](http://www.cwsei.ubc.ca/resources/files/CourseTransformationGuide_CWSEI_CU-SEI.pdf)) ·

Deslauriers et al., *Improved Learning in a Large-Enrollment Physics Class*

([//www.sciencemag.org/content/332/6031/862.full](http://www.sciencemag.org/content/332/6031/862.full))

## # Why is attendance in class required?

Class meetings are all about *practice and feedback*. The individual, pair, and small group questions, exercises, and coding problems we work on in class, and the discussions and feedback led by instructors, are a required component of 6.005.

Class is like swim/judo/math team practice: you don't get good unless you show up, and practicing with others is a necessary complement to practicing alone.

## # Why are we sometimes asked to close laptops during class?

Your laptop is a necessary tool for in-class programming, but it also presents a huge opportunity for distraction. The price of that distraction is paid not only by you, but by all those around you who can see your screen. In one study linked below:

- For note-takers with laptops, multi-tasking led to a 11% drop in comprehension test scores.
- For note-takers without laptops, merely having a laptop multi-tasker in their field of view led to a 17% drop in comprehension test scores!

If you want to use a smartphone in your lap, so that the screen is not visible and not distracting to others, we have no objection, but you'll still be hurting yourself.

Citations: Sana et al., *Laptop multitasking hinders classroom learning for both users and nearby peers* ([//www.sciencedirect.com/science/article/pii/S0360131512002254](http://www.sciencedirect.com/science/article/pii/S0360131512002254)) · Mueller and Oppenheimer, *The Pen Is Mightier Than the Keyboard* ([//pss.sagepub.com/content/25/6/1159.full](http://pss.sagepub.com/content/25/6/1159.full))

## # How is participation in class graded?

In general, participation in class is graded based on whether you attempted the questions, exercises, coding problems, etc., not on correctness.

Programming exercises are graded based on whether you've attempted the exercise and made some progress, even if you don't complete it. You are not expected to complete unfinished exercises after class, but TAs and LAs will be happy to help you review or finish them.

## # Why do I need to read the readings and complete the reading exercises the night before coming to class?

Reading the material before class prepares you to spend class time *practicing* the concepts and skills you're learning. Reading in advance gives you time to think and ask questions, and repeated exposure to material spaced out over time improves learning.

See: Spacing effect ([https://en.wikipedia.org/wiki/Spacing\\_effect](https://en.wikipedia.org/wiki/Spacing_effect))

## # Why does the class have nanoquizzes on topics before we practice them in class?

Nanoquizzes assess whether you did the reading and practiced with the reading exercises before coming to class, and they provide feedback to you on your comprehension. Nanoquizzes are themselves part of the practice we do in class: recalling information from the readings benefits learning more than just re-reading or re-hearing it.

See: Testing effect ([https://en.wikipedia.org/wiki/Testing\\_effect](https://en.wikipedia.org/wiki/Testing_effect))

# Problem sets, beta & final, and slack days

**Problem sets.** To consolidate your understanding of the ideas from class, you will do five problem sets, PS0 to PS4, involving both design and implementation work. Problem sets will be done individually. More info: [collaboration on problem sets \(collaboration.html#problem\\_sets\)](#) .

**Code review.** As part of each problem set, there will be a 2-day code-reviewing period when other students and staff will give you feedback about the code you submitted, using a web-based system. You will be expected to participate in this process by reviewing some of your classmates' code. More details about objectives and guidelines for the code reviewing process: [code reviewing \(code-review.html\)](#) .

**Beta and final submission.** Each problem set will have beta and final submission. The beta submission will be graded by an automated tester, and will be subject to code review. The final submission is due a week later. You will be expected to fix any failed test cases and revise your code based on code review feedback.

The final submission must address all the code reviews received by the beta submission – all the human comments plus all the automated checkstyle comments that are marked #important. You can address a code review either by changing the code to reflect the review or by including a source code comment in your code that explains why the code wasn't changed. If code reviews are unclear, you can discuss them with the reviewers, but you still must edit your own code in reaction to the review. A grader will check the submission and deduct points if it hasn't addressed the code reviews.

Since the final submission inevitably happens after code review for the problem set, it's understood that you've looked at other students' written solutions, and been inspired by other ways to solve the problems. You must be exceedingly scrupulous, therefore, in not using those written solutions during your revision. Both your original code and your revised code must be your own. Looking at other students' answers to the problem set while you are revising your solution will be considered a violation of the collaboration policy ([collaboration.html#problem\\_sets](#)) .

## # How is the overall grade for a problem set determined?

The overall grade for a problem set will **typically** be computed as follows:

$$\text{overall grade} = 40\% \times \text{beta-autograde} + 45\% \times \text{final-autograde} + 15\% \times \text{manual-grade}$$

where beta-autograde and final-autograde are determined by automated tests, and manual-grade is determined by graders reading the code. One part of manual-grade is fixing code in response to code reviews.

The breakdown may vary by 5-10% from problem set to problem set, depending on how much of the problem set can be autograded and how much requires human eyes. Each problem set's breakdown will be announced when final grades are released.

# Projects and quizzes

**Project.** You will complete a group software development project at the end of the semester. The project will be done in teams of three students. Each team member is required to participate roughly equally in every activity (design, implementation, test, documentation), and we may ask for an accounting of what each team member did. A single grade will be assigned to all members of the team.

**Team meetings.** During the project, you and your project team will meet with your TA to discuss the work. Your TA will assign a grade based in part on this meeting. Team meetings will usually be scheduled during class times that will be reserved for this purpose.

**Quizzes.** There will be two quizzes, on dates specified on the course calendar. Each quiz will be comprehensive, drawing on any topics covered up to that point in the course, so e.g. Quiz 2 may include topics that were already covered on Quiz 1.

# **Are quizzes closed-book?**

Yes, but you may bring a single 8.5×11” double-sided page of notes, readable without magnification, that was created by you. Since the process of creating a crib sheet is most of its benefit, you may not share these notes or use someone else’s.

# **Are past quizzes available?**

Yes, in the archive of past quizzes (../quizzes/archive/).

## Collaboration and public sharing

This topic has its own page: **collaboration and public sharing** (collaboration.html) .

## Grading

The relative contributions of the various elements to your grade are:

- **Quizzes: 30%.** Nanoquizzes are worth 10% total, and each comprehensive quiz is worth 10%.
- **Problem sets: 50%.** PS0 is worth 6%, and PS1-PS4 are each worth 11%.
- **Project: 10%.**
- **Code review: 5%.** Judged by regular participation in code review and providing substantive, useful comments.
- **Participation: 5%.** Reflects your continuous participation in the course. Determined by completion of reading exercises, participation in class and in the online Piazza forum.

Letter grades are determined at the end of the semester. The default cutoffs are: a final average of 90 and above is an A, 80 and above is a B, 70 and above is a C. These boundaries may be adjusted downwards if necessary because of the difficulty of the assignments or quizzes, but the boundaries will never be adjusted upwards, so a final average of 90 is guaranteed to be an A. The boundary adjustment is done heuristically, and there are no grade quotas, no grade targets, and no centering of the class on a particular grade boundary.

Every student is considered individually in the final grading meeting, judging from their entire performance in the course. A single bad mark in an otherwise consistent record will often be discounted.

MIT EECS

6.005 — Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |  
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>)  
Spring 2016

## Collaboration and Sharing

### Collaboration

#### Public sharing of work

# Collaboration and Sharing

In line with MIT's policy on Academic Integrity (<https://integrity.mit.edu/handbook/writing-code>) , here are our expectations regarding collaboration and sharing of work.

## Collaboration

We encourage you to help each other with work in this class, but there are limits to what you can do, to ensure that everybody has a good individual learning experience. This section describes those limits.

### Problem sets

Problem sets are intended to be primarily individual efforts. You are encouraged to discuss approaches with other students but your code and your write-up must be your own.

**You may not use materials produced as course work by other students, whether in this term or previous terms, nor may you provide work for other students to use.**

During code review, you will see classmates' solutions to a problem set. While it is fine to take inspiration from their approach, do not copy their work.

It's fine to use material from external sources like StackOverflow, but only with proper attribution (<https://integrity.mit.edu/handbook/writing-code>) , and only if the assignment allows it. In particular, if the assignment says "implement X ," then you must create your own X , not reuse one from an external source.

It's also fine to use any code provided by this semester's 6.005 staff (in class, readings, or problem sets), without need for attribution. Staff-provided code may not be publicly shared without permission, however, as discussed later in this document.

### Examples

1. Alyssa and Ben sit next to each other with their laptops while working on a problem set. They talk in general terms about different approaches to doing the problem set. They draw diagrams on the whiteboard. When Alyssa discovers a useful class in the Java library, she mentions it to Ben. When Ben finds a StackOverflow answer that helps, he sends the URL to Alyssa. **OK.**
  - As they type lines of code, they speak the code aloud to the other person, to make sure both people have the right code. **INAPPROPRIATE.**
  - In a tricky part of the problem set, Alyssa and Ben look at each other's screens and compare them so that they can get their code right. **INAPPROPRIATE.**
2. Alyssa and Ben haven't been working together on the problem set so far, but Ben is now struggling with a nasty bug. Alyssa sits next to him, looks at his code, and helps him debug. **OK.**

- Alyssa opens her own laptop, finds her solution to the problem set, and refers to it while she's helping Ben correct his code. **INAPPROPRIATE.**
- 3. Louis had three problem sets and two quizzes this week, was away from campus for several days for a track meet, and then got sick. He's already taken two slack days on the deadline and has made almost no progress on the problem set. Ben feels sorry for Louis and wants to help, so he sits down with Louis and talks with him about how to do the problem set while Louis is working on it. Ben already handed in his own solution, but he doesn't open his own laptop to look at it while he's helping Louis. **OK.**
  - Ben opens his laptop and reads his own code while he's helping Louis. **INAPPROPRIATE.**
  - Ben has by now spent a couple hours with Louis, and Louis still needs help, but Ben really needs to get back to his own work. He puts his code in a Dropbox and shares it with Louis, after Louis promises only to look at it when he really has to. **INAPPROPRIATE.**
- 4. John and Ellen both worked on their problem sets separately. They exchange their test cases with each other to check their work. **INAPPROPRIATE.** Test cases are part of the material for the problem set, and part of the learning experience of the course. You are copying if you use somebody else's test cases, even if temporarily.

Note that in the examples marked inappropriate above, *both* people are held responsible for the violation in academic honesty. Copying work, or knowingly making work available for copying, in contravention of this policy is a serious offense that may incur reduced grades, failing the course and disciplinary action.

## Group projects

You should collaborate with your partners on all aspects of group project work, and each of you is expected to contribute a roughly equal share to design and implementation.

You may reuse designs, ideas and code from your own work earlier in the semester (even if it was done in a group project with a different partner). You may also use any code provided by this semester's 6.005 staff.

You may also use material from external sources, so long as: (1) the material is available to all students in the class; (2) you give proper attribution (<https://integrity.mit.edu/handbook/writing-code>) ; and (3) the assignment itself allows it. In particular, if the assignment says "implement X," then you must create your own X, not reuse someone else's. Finally, your group may not reuse designs, ideas, or code created by another group, in this semester or previous semesters.

## Public sharing of work

People often want to share their code publicly, e.g., on Github, in order to show off a portfolio of code they've written to potential employers. Building a portfolio is a great idea, but 6.005 is *not* a good class to use for it, because the problem sets and projects are fixed by the course staff, not chosen by you. Personal projects, hackathons, and IAP contests are much better ways to build up your portfolio.

The policy for public sharing of 6.005 code is described below.

## Problem sets

Copyright for the starter problem set code is held by the 6.005 course staff, and does not allow redistribution of derived works without prior permission. Your solutions are a derived work, so you may not distribute your problem set solutions publicly. This means you cannot post them on Github, in a public Dropbox folder, or on a public server accessible to others.

Keep in mind that when work on an individual problem set is copied, both the provider and the consumer of copied materials are violating academic honesty standards, as described above.

## Group projects

For group projects, before you can post any code for public distribution, all **authors** of the code must **agree** to public distribution, and their authorship must be **acknowledged**.

- **Authors** means anyone who worked on the project at any time, including previous phases of the project. Suppose Alice wants to post her project code. Alice worked on project 1 with Bob and Carol, and then she uses code from project 1 during project 2 with Yolanda and Zach. Yolanda and Zach also use code from their own project 1, when they worked with Morris, Nancy, Ollie, and Patricia. Alice, Bob, Carol, Morris, Nancy, Ollie, Patricia, Yolanda, and Zach are *all* authors of Alice's project 2.

Even if Alice only wants to post one file from the project, *even if she wrote that code herself*, she must ask all of her teammates from all phases of the project, and all of their teammates from previous phases of the project. Projects are a team effort, and code ownership is shared.

- **Agree** means all authors must be asked whether public distribution is OK, and they must answer yes. In our example, if Alice wants to post any part of project 2, then Bob, Carol, Morris, Nancy, Ollie, Patricia, Yolanda, and Zach must *all* be asked and answer yes.
- **Acknowledged** means the names of all authors appear either in a comment at the top of each source file, or in a `README.md` (or other prominent README file) at the root of the project. If you're posting on a site like GitHub, keep the commit history – and for projects that use code from a previous group, import that code with its commit history. This history demonstrates each person's individual contribution, and shows off your software development process.

These rules are derived from the standard practice for open-source code projects, in which it's necessary to clarify the origin of all the code and obtain Contributor License Agreements

([//en.wikipedia.org/wiki/Contributor\\_License\\_Agreement](https://en.wikipedia.org/wiki/Contributor_License_Agreement)) from all identified contributors. You should follow this standard ethical practice of the software development community even for your own projects. If you want to publish code from a hackathon where you worked with other people, then make sure all the authors agree and are appropriately acknowledged.

MIT EECS

6.005 — Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |  
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>)  
Spring 2016

Nanoquiz Grading and Makeup

Nanoquiz Grading

Nanoquiz Makeup

# Nanoquiz Grading and Makeup

## Nanoquiz Grading

Nanoquizzes consist of multiple-choice questions and/or short answer questions.

For multiple choice questions, each option is worth one point. If you correctly choose the answer when it's right, then you get one point. If you correctly don't choose the answer when it's wrong, then you get one point. This scheme applies uniformly to choose-all-good-answers and choose-one-best-answer questions.

For example, consider a choose-all-good-answers question with choices A,B,C,D, of which only C and D are correct. If you answer C,D then you get 4 points. If you answer A,C then you get 2 points (since you were wrong about A, right about B, right about C, and wrong about D). If you answer A,B then you get 0 points.

For example, consider a choose-one-best-answer question with choices A,B,C of which only C is correct. If you answer C, then you get 3 points. If you answer A, then you get 1 point (since you were wrong about A, wrong about C, but right about B).

The point value for short answer questions is determined by the staff. It is up to staff discretion.

Your total score for a quiz will be scaled to be out of 10 points.

## Nanoquiz Makeup

We automatically drop the lowest 5 nanoquiz grades. If you're dissatisfied with any of your remaining nanoquiz grades, however, you can earn up to half the lost points for that quiz back by writing a good nanoquiz question for the same class meeting.

- Your question must be multiple-choice, either "choose one answer" or "choose all good answers."
- Your submission must include what you think the right answer(s) should be.
- Good questions should be relevant to the reading for that class, straightforward to answer by someone who did the reading, and hard to get full credit on otherwise.
- Good questions will generally come from the Comprehension or Application levels of Bloom's Taxonomy (<http://www.nwlink.com/~donclark/hrd/bloom.html>) . Higher levels are better. See Multiple Choice Questions based on Bloom's Taxonomy (<http://learningsciences.utexas.edu/content/blooms-taxonomy-approach>) for examples of questions at different levels.

Be sure you do a makeup for the right class. Nanoquizzes are numbered by the class meetings when they were held. If you don't like your grade for Nanoquiz 2, then you should be offering a makeup related to Class 2.



Note that these alternatives are worth only half of the lost points. If you took the nanoquiz and got 8/10, then this makeup can bring your grade up to 9/10. If you missed a nanoquiz entirely and got a 0/10, then this makeup can raise it to 5/10.

The deadline for making up a nanoquiz is one week ( $7 * 24$  hours) after that nanoquiz's grades are posted. Makeups may not be revised or resubmitted. Only one submission per nanoquiz will be considered.

**MIT EECS**

6.005 — Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |  
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>)  
Spring 2016

## Getting Started

Step 1: Install software

Step 2: Configure Eclipse

Step 3: Install more software

Step 4: Open the command line

Step 5: Configure Git

Step 6: Learn the Git workflow

# 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** (<http://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** (<http://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 (<http://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** (<http://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/](https://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](http://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 n
ano
```

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 n
otepad
```

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](http://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](https://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)
```



## Pulling

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.

## Merge conflicts

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

6.005 — Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |  
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>)  
Spring 2016

Learning Java

Materials

# 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 the materials below
- review **Reading 2** ([../classes/02-basic-java/](#))

## Materials

The lecture slides and short programming assignments below are from IAP course 6.092 in 2010 (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/index.htm>) . They were authored by Evan Jones, Adam Marcus, and Eugene Wu.

You can also look at **materials from 6.S092 in 2015**

([//stellar.mit.edu/S/course/6/ia15/6.S092/materials.html](http://stellar.mit.edu/S/course/6/ia15/6.S092/materials.html)) or the most recent **materials from 6.178 in 2016** (<https://learning-modules.mit.edu/materials/index.html?uuid=/course/6/ia16/6.178>) .

## Slides

- Lecture 1 - Types, Variables, Operators ([https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/lecture-notes/MIT6\\_092IAP10\\_lec01.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/lecture-notes/MIT6_092IAP10_lec01.pdf))
- Lecture 2 - More Types, Methods, Conditionals ([https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/lecture-notes/MIT6\\_092IAP10\\_lec02.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/lecture-notes/MIT6_092IAP10_lec02.pdf))
- Lecture 3 - Loops, Arrays ([https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/lecture-notes/MIT6\\_092IAP10\\_lec03.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/lecture-notes/MIT6_092IAP10_lec03.pdf))
- Lecture 4 - Classes and Objects ([https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/lecture-notes/MIT6\\_092IAP10\\_lec04.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/lecture-notes/MIT6_092IAP10_lec04.pdf))

## Short Programming Assignments

You can use these assignments (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/assignments/>) to practice Java before diving into PS0 ([../psets/ps0/](#)) , Git ([//git-scm.com](http://git-scm.com)) , and JUnit ([//junit.org](http://junit.org)) .

**The assignments below are not required for 6.005.** You will not turn them in.

To work on the assignments, create a new Eclipse Java project ( *File* → *New* → *Java Project* , name it `tutorial` , click *Finish* ).

Then create a new class file ( *File* → *New* → *Class* , enter the CamelCase name, click *Finish* ).

Then copy-and-paste in the provided starting code.

To run a class with a `main` method, right-click and choose *Run As* → *Java Application* .

- Assignment 1 - GravityCalculator ([https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/assignments/MIT6\\_092IAP10\\_assn01.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/assignments/MIT6_092IAP10_assn01.pdf))
- Assignment 2 - FooCorporation ([https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/assignments/MIT6\\_092IAP10\\_assn02.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/assignments/MIT6_092IAP10_assn02.pdf))
- Assignment 3 - Marathon ([https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/assignments/MIT6\\_092IAP10\\_assn03.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/assignments/MIT6_092IAP10_assn03.pdf))

---

## Have fun in 6.005!

MIT EECS

6.005 — Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |  
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>)  
Spring 2016

Eclipse FAQ

Resources

FAQ

Tips & Tricks

# Eclipse FAQ

## Resources

**Java Debugging with Eclipse** (<http://www.vogella.com/tutorials/EclipseDebugging/article.html>) introduces Eclipse's debug mode, a recommended tool for tracking down bugs using breakpoints. This step-by-step tutorial explains the *Debug perspective* and provides screenshots to decode the mysterious debugger icons in Eclipse.

## FAQ

### I messed up my Eclipse and it doesn't show all the things it used to show

First, make sure you are in the *Java perspective* : in the top right corner of the window, click the **Open Perspective** toolbar button and select **Java** .



If your Java perspective doesn't look right, go to **Window** → **Reset Perspective** to return to the Eclipse defaults.

## Tips & Tricks

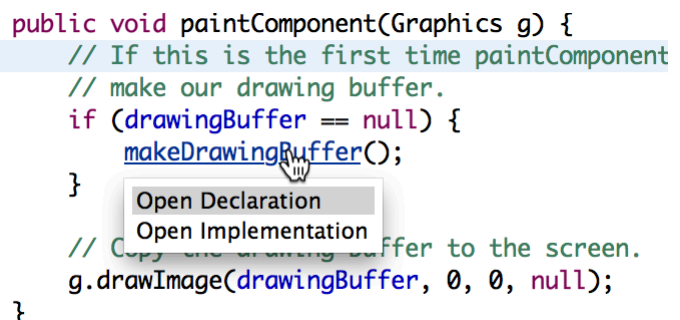
### Navigating

Command-click (on Mac) or control-click to **jump straight to the definition** of a variable, field, method, class, etc.

Return to the previous file you were working on with **Navigate** → **Back** , which might have keyboard shortcut Alt-left-arrow or Command-[, Eclipse's *Navigate Back* and *Forward* are analogous to the back and forward buttons in your web browser.

Command- (or control-) -shift-R and command-shift-T bring up a dialog for quickly searching for and **navigating to files** (a.k.a. resources) and **types** (classes), respectively.

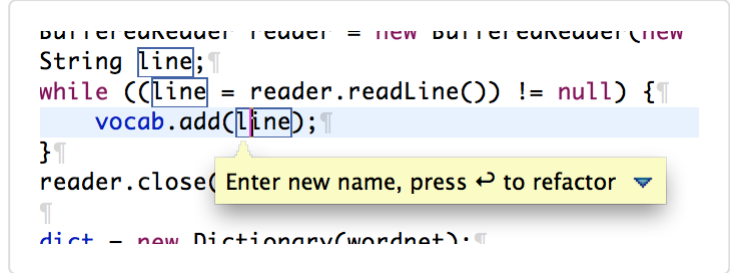
```
public void paintComponent(Graphics g) {  
    // If this is the first time paintComponent  
    // make our drawing buffer.  
    if (drawingBuffer == null) {  
        makeDrawingBuffer();  
    }  
    // Copy the drawing buffer to the screen.  
    g.drawImage(drawingBuffer, 0, 0, null);  
}
```



## Editing

Select an element and use **Refactor** → **Rename** (or its keyboard shortcut) to rename that variable, field, method, class, etc. wherever it appears.

Take advantage of **code completion** to help you use a new API: if you have an object squiggle, typing the name and then a period will bring up code completion options.



Use **Source** → **Organize Imports** or command- (or control-) -shift-O to automatically add and organize Java import statements. Just start using a new class, *Organize Imports*, and you're good to go.

Don't like formatting and indenting your code manually? Use **Source** → **Format** to clean up your act.

Use **Source** → **Toggle Comment** or keyboard shortcut Command-/ to quickly toggle on or off some lines of code.

Use **block selection** (<http://blog.jooq.org/2013/10/12/eclipses-awesome-block-selection-mode/>) to edit across multiple lines at once.

---

## Have fun in 6.005!

MIT EECS