About

Lab 0 - Git + TypeScript

Sept. 25, 2025

Point of Contact: Prof. Coblenz (mcoblenz@ucsd.edu)

Learning goals for this lab:

1. **Be able to create a Git repository and manipulate it in a team context.**
2. **Become familiar with TypeScript.**

Lab participation rules:

1. **Be in the lab. You will need a partner for part of this lab, and being in the lab will enable you to find a partner and easily get help if you are stuck.**
2. **Try to get your lab checked off by a TA or tutor by the end of lab. If you can't, complete it at home by the end of day Oct. 1, 2025.**

Notes

- **Do not skip ahead. Read each section closely before going on to the exercises.**
    - If you rush through this lab, you may be missing important concepts, which will hurt you later. Don't just cut and paste your way through this lab. Try to understand what you're doing. Ask if you don't understand.
- **Remember to refresh this document every so often.**
    - When many (50+) people are viewing a Google Doc, changes will not show up automatically! Refresh every so often, in case we need to make just-in-time changes to fix mistakes in the lab or add clarification.
- Using AI is permitted on labs and projects (but not on homework assignments). If you choose to use AI systems, use them judiciously. Understand the code that the AI generates. Question whether it's right. Pay special attention to design: the code may build, but is this the structure you wanted? Imagine using an AI system to design a car. If you let it do

whatever it wants, you might end up with the seats on the roof, sixteen wheels, and the steering wheel right in the center. Each part sort of makes sense (there's a nice view from the roof) but it's not the design you wanted. You are the designer, not the AI system.

Help

- For help, submit an Autograder ticket (https://autograder.ucsd.edu).

Completion

- When you're finished with the tasks, submit an Autograder ticket (https://autograder.ucsd.edu).

About Labs

Welcome to the CSE 110 Labs!

Over the course of this quarter, you will be completing about five labs, which will guide you through a variety of material related to your course project.

Labs are here to help you. Labs teach you the concepts, skills, and best practices that are essential to your success on your course project, and hopefully beyond. Think of labs as a sequence of guided tutorials plus exercises and questions tailored to help you to succeed on your project.

Working on Labs

To the extent possible, you should try to work on your labs while physically co-located partner as much as possible. You and your lab partner will turn-in/check-off together. In general, you should both complete your own copy of the lab, but if you want to work on one copy we highly recommend pair programming. You can take turns being the driver or navigator. Plus, thinking and

talking things through together is much more fun than both staring at your own screens. The TypeScript task can be done entirely together.

On the flip side, **do <u>not</u> attempt any "clever" work-splitting arrangements**, be that splitting work on one lab up into parts, or alternating who does each lab. We can safely say **they don't work, and will only come back to bite you later on**. The former simply doesn't work with how labs are structured, while the latter will result in skill gaps that will make working on your project unpleasant for you and your team.

Getting Help with Labs

If you need help with labs, you have several options:

During the Thursday lab, you can request help from an on-duty staff member via Autograder. Be sure to include any relevant details about your location (physical or virtual). If your ticket has not been handled and you have to leave, please help your classmates out and delete your ticket. Also please note that queues will be emptied at the end of each lab session.

The rest of the week, you can request help in any open office hours. Support is first come first serve. At the discretion of the TA, this may also be via Autograder. Depending on how many checkoffs remain, TAs may choose to prioritize them later in the week to ensure everyone gets checked off.

And any time at all, you can ask on Piazza! If possible, post your lab-related questions publicly! This helps to reduce duplication, and allows your classmates to help answer questions they've already figured out. On the flip side, be sure to check that your question hasn't already been asked elsewhere before posting.

Turning in Labs

During any lab session or office hours, once you have finished your lab, you can get checked off. In general, the process is:

1. Finish your lab.
2. Create a ticket of type "check-off" on Autograder.

3. When your ticket is taken, the staff member will then:
    a. Review your question answers with you.
    b. Go through your exercise work to verify that it works properly.
    c. Record that you have been successfully checked off.

Please do your part to keep this process as smooth as possible, for everyone. It's a big class, we are a small staff, and towards the deadline there can be a lot of checkoffs. If we take your ticket and you aren't ready, you will end up back in the queue so we can get to other people waiting.

Some Tips for Labs

Here are some tips for labs based on the course staff's experience which will make things a lot easier for you:

1. **Proceed step by step**. **Do not just skim through to the next screenshot or exercise**.
    - This is a recipe for missing important information.
    - You'll end up spending more time going back to figure out what you missed than if you'd just gone through it carefully to start with.
2. **If you see links in the text, you should check them out.**
    - They're usually a slightly fuller or more detailed explanation of a concept.
    - If you find you're not really sure you understand something, and there's a link, click it.
3. **Avoid copy-pasting code from the internet. Type it out.**
    - Take the time to go through it line by line. Ask yourself if you understand what each line is doing.
4. **Make use of version control as you work through the lab.**
    - When you finish a section or a hard exercise, commit so you have a checkpoint to go back to.
5. **Use the outline view of Google Docs.**

- To open, click the little menu/list icon floating in the top left of the screen, below the toolbar.

Git

Learning goals for Git:

- Be able to create a fresh git repository and push it to GitHub.
- Be able to use *command-line tools* to commit, push, pull, and resolve conflicts.
- Be able to clone an existing repository on a Git server.

You have some experience with Git from CSE 29, but in this course you will take that knowledge several steps further. We'll focus on the basics today, and then we'll do a deeper dive in two weeks.

This part of the lab uses "Pro Git", a free online book: https://git-scm.com/book/en/v2. Rather than duplicating content, we will ask you today to read sections of the book.

Regarding the Command Line

There are lots of fancy GUIs for git. You might even decide to use them. However, everything you can do in them somehow maps to things that you could do on the command line. To make sure you understand the fundamentals, today's work will be *command-line git only.*

If you are on Windows, consider using the Windows Subsystem for Linux (https://learn.microsoft.com/en-us/windows/wsl/install) so you can use a more UNIX-like command line environment. If you are on Mac or Linux, you can use your regular command line.

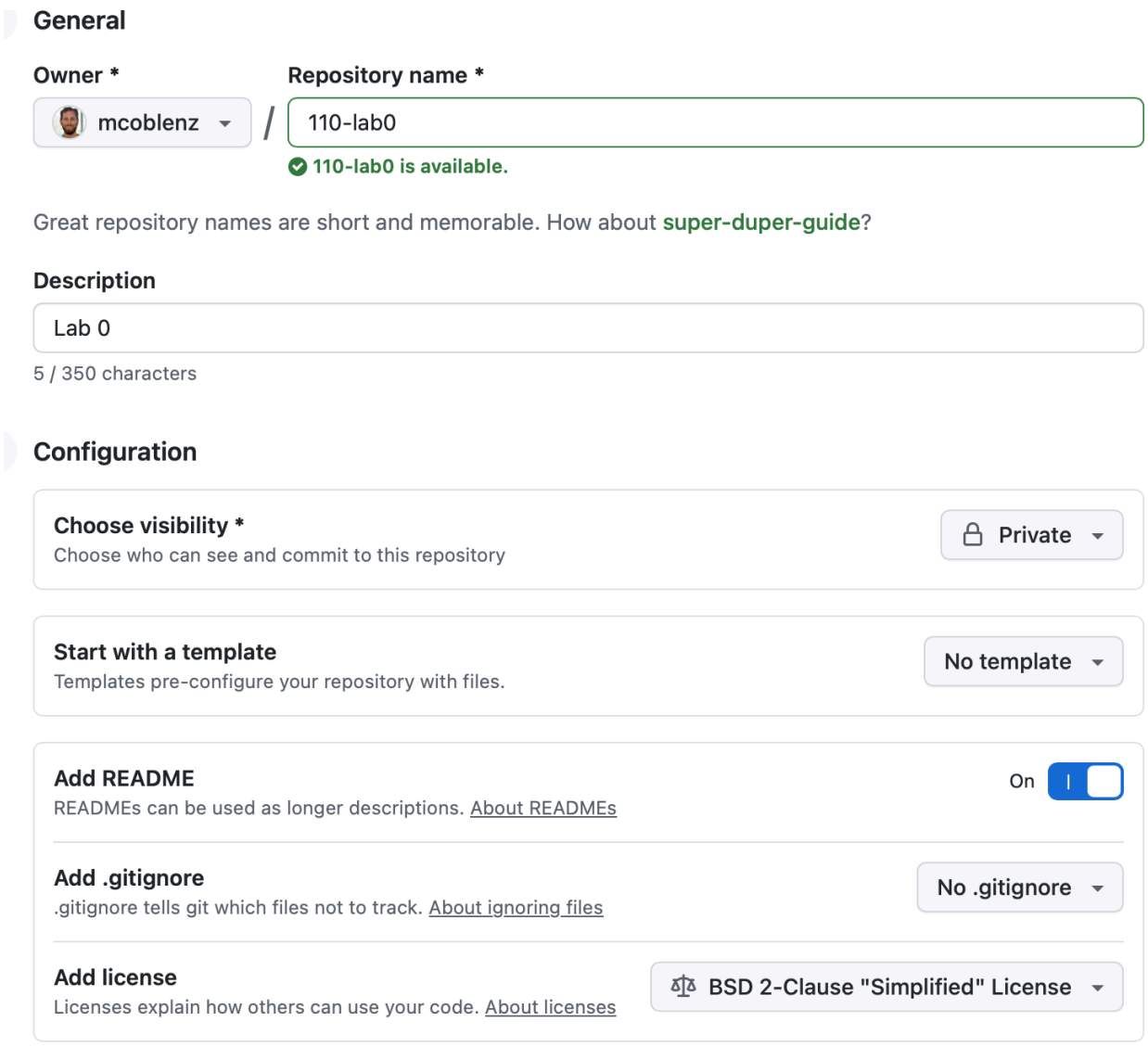
Task 1: Install Git

You will need a *local* installation of git on your computer. Use these instructions, according to your operating system: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git. You do not need to install from source. If you are on macOS, installing the XCode command line tools will suffice; you don't need to install git separately.

Task 2: Create a repository

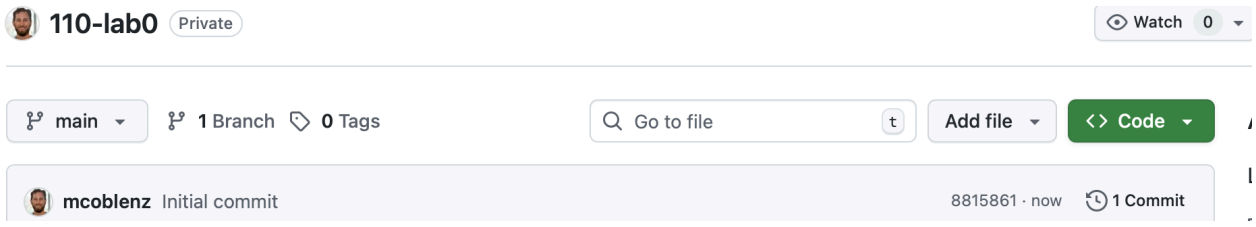
You can create a repository locally and then push it, but GitHub provides some convenient defaults, so let's do that.

Go to github.com and log in. Create a new repository (`110-lab0`). Make it Private and pick an appropriate license (this your code, so you can do whatever you like here). Say that you want a README. It should look like this:



Now, you need to clone your repository so you can work with it locally.

Click the green “Code” button to get a link:



Copy the resulting URL. Now go back to the command line. Change to a good place for the code and run:

```
$ git clone <paste>
```

Since we have made the repository private, we need to authenticate with GitHub to clone it. You may use any authentication method. You may find the GitHub guide useful (generating a SSH key, Adding a new SSH key to your GitHub account). The gh command can also automatically manage credentials..

Now you should have a directory with the checked out code.

```
$ cd 110-lab0
$ ls
LICENSE          README.md
```

Task 3: Edit the code

Open this directory with VSCode (if you don't have it installed, install it (https://code.visualstudio.com/). On the command line:
`$ code .`

Now, you're going to need a place to record your favorite snacks. Create a new file, snacks.txt. Write down your favorite three kinds of snack food in alphabetical order, with one item per line.

Task 4: Commit the code.

Back to the command line, let's find out what's up.

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
     snacks.txt
```

Important conceptual information: your local state includes an "index," which is a temporary place for tracking changes to the repository that you will make soon. The workflow is: add changes to the index, *commit* those changes to the local repository, and then *push* them to the remote repository.

Important: the upcoming steps are intended to show you how git works. As you do them, build a conceptual model of git in your head. Do not just follow these steps like a tutorial.

We're going to want to `add` that new file to add it to the index.

```
$ git add snacks.txt
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
     new file:   snacks.txt
```

Let's see what happens if we make some changes before committing. Add a fourth snack to your file.

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
     new file:   snacks.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
     modified:   snacks.txt
```

This means that the index has recorded the "add" from before but not the new changes you made since. When you commit, you'll add a new commit to the graph that includes **only the changes that are in the index**.

```
$ git commit
```

At this point, git will open an editor in which you can type a commit message. Type "Added snacks file." Save and quit the editor. You'll see:

```
[main 0935a3d] Added snacks file.
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 snacks.txt
```

If the editor wasn't your favorite, change your $EDITOR environment variable. You can ask the Internet how (it will depend on which shell you're using).

Note that you only committed the changes you previously added to the index:

```
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
      modified:   snacks.txt
```

Task 5: Commit another version

As a shortcut, you can add a file to the index and commit at the same time:

```
$ git commit snacks.txt
```

Go ahead and write an appropriate commit message. It should be descriptive and specific. For example, do not write “updated snacks list.” What was the update? Why did you make the change?

Task 6: Editing history

Now, let’s suppose you had a typo in your previous commit message. Read how to fix it here: https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things

Change your previous commit message (don’t just add another commit saying “oops”).

Task 7: Undoing local changes.

Add another snack to your snacks.txt file. But suppose you change your mind and you want to throw that change away. Use `git checkout` to restore the old version of the file.

Task 8: Pushing

Read https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes.

Then, push to the remote repository.

Be prepared to explain to your TA at checkoff time what the difference is between `git reset`, `git commit --amend`, and `git checkout`.

Task 9: Pulling

Find a partner. Clone their repository. Ask them to make another change. Then, use `pull --rebase` to get their changes. We will discuss in another lab why you should use `--rebase`, but for now, trust us. If you're feeling impatient and are ready to think about merging, read this: https://levelup.gitconnected.com/git-pull-vs-git-pull-rebase-a-complete-guide-for-developers-e209ce0df18c?gi=ee9865431666.

TypeScript

Getting started with TypeScript

TypeScript is a *superset* of JavaScript: it extends JavaScript with a type system that enables the compiler to check various properties of your code. The goal is to help you write more reliable code and to help you reason about code by showing you what can be done with each value. For example, if you want to call a function that takes an argument `x`, types tell you what kind of `x` you can pass.

We're interested in writing code that runs in web browsers, but web browsers do not support TypeScript directly. This means that we will need to compile our TypeScript code to JavaScript code, so we'll need to learn how to use the TypeScript compiler. We'll also need to package our code so it can run in a browser.

Our first steps will follow https://tsx.is/typescript.

Task 0

TypeScript runs in the *node* ecosystem and uses the Node package manager to manage dependencies. If you haven't already, install Node.js.

Make sure `node` and `npm` commands are installed.

Task 1

In your directory from before, run:

```
$ npm init -y
```

This will emit a package.json file that describes your project and how to build it.

Edit the package.json to say that this is a TypeScript project:

```
$ npm install --save-dev typescript @types/node
```

Note that this command resulted in the following text in package.json:

```
"devDependencies": {
    "@types/node": "^24.5.2",
    "typescript": "^5.9.2"
```

```
  }
```

In other words, it says that when building the project, the build system will need to use version 5.9.2 (or newer) of the TypeScript package, and likewise for @types/node. There are no *runtime* dependencies; once you've compiled the project, it doesn't matter that you wrote it with TypeScript. That's why this is in "devDependencies" and not "dependencies."

Now, we want git to know which files to ignore (to not add them to the repository). Download this file and put it in the directory:

https://github.com/github/gitignore/blob/main/Node.gitignore

Let's configure the TypeScript compiler. Create a tsconfig.json file and put the following in it:

```
{
      "compilerOptions": {

            // Treat files as modules even if it doesn't use
import/export
            "moduleDetection": "force",

            // Ignore module structure
            "module": "Preserve",

            // Allow JSON modules to be imported
            "resolveJsonModule": true,

            // Allow JS files to be imported from TS and vice versa
            "allowJs": true,

            // Use correct ESM import behavior
            "esModuleInterop": true,

            // Disallow features that require cross-file awareness
            "isolatedModules": true,
      },
}
```

Create a `src` directory. In a file `src/main.ts`, write:

```
console.log("Hello, World!");
```

We're going to use a tool called tsx to invoke the compiler. Install it:

```
npm install -g tsx
```

Now, run the code:

```
$ tsx src/main.ts
Hello, World!
```

You're not done until you commit!

```
$ git add .gitignore package.json src/* tsconfig.json
$ git commit
(enter a suitable message)
$ git push
```

Note that we didn't commit node_modules because that's *derived* content, not your source code. In particular, it contains downloaded packages. Those who clone the repository should fetch those separately.

Task 2

Read this documentation to get you started:
https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes-oop.html

Then you will likely want to read a few pages of documentation, starting here:
https://www.typescriptlang.org/docs/handbook/2/basic-types.html
https://www.typescriptlang.org/docs/handbook/2/objects.html

There's a fine balance in learning a new programming language. On one hand, you want to read some documentation to explain key concepts. On the other hand, you want to get programming relatively quickly to internalize what you've learned. Those few pages might be enough to get you started.You'll want to refer back to the documentation frequently.

This task is inspired by an old game, shown here (you do not need to watch the video):
https://www.youtube.com/watch?v=ryArL9lU3EI

We encourage you to work in *pairs* for this task. You can be checked off together.

Because you have limited time in this lab, do not try to replicate all aspects of the game. For example, we do not expect you to provide the user with graphics or sound of any kind. Instead, focus on

Using good object-oriented design principles, implement a lemonade stand simulation. You should have a class to represent the state of the lemonade stand. The stand should maintain an inventory, which is updated every time a cup is sold according to a recipe that is maintained over time.

The simulation should proceed one day at a time. Each day, the player of the game will be told what the weather is (more lemonade will be sold on hotter days) and the current prices for supplies (cups, ice, lemons, and sugar). The player should input how much supplies to buy at the current prices. Then, the simulation should tell the player how many cups were sold, how many supplies are left, and what the current cash balance is.

You should do some searching to find out how to handle console input in Node.js. This is an important skill (beyond just using AI and hoping it uses the API correctly). Read the documentation and make sure you're doing it right.

Be sure to commit your code each time you add a feature. We expect to see a few different commits with appropriate comments as you make progress. Do not just undo and then commit at the end — we will see you've done this when we look at your commit log!

Before you ask to be checked off, prepare a log of a typical game session. Then, prepare to explain key design decisions you made in your implementation and what you learned about TypeScript. How does TypeScript compare to Java?

When you have finished, submit an autograder ticket to be checked off by a TA or tutor.