

# Lab 2 - Git, Tests & Continuous Integration

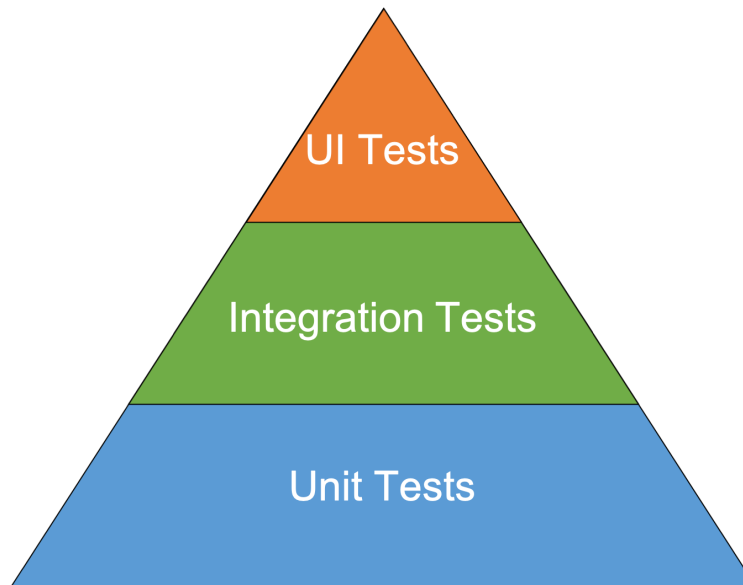
## Important Link

- [Check-off Form](#)
  - Please submit this **before** meeting with a TA/Tutor to check off.
  - If your form is not submitted there is no guarantee that you will keep your place in the Autograder queue!
- [Autograder](#)
  - This is where you can submit a question ticket to get help or a check-off ticket to get your lab checked off.

## Important Notes

- **If you are not ready to check off, do not mark your ticket in AutoGrader as a check-off.**
  - We try to split duties to handle technical issues for people who are stuck from those just waiting for a check-off. If you mark your ticket as check-off when you have technical issues, you slow the queue down for **everyone**.
- Don't skip the reading in this lab! There might be important details...

## Part 1: The Test Pyramid



This “test pyramid” is used to picture the relationship between different types of software tests. The big three are UI, integration and unit tests.

The size of each portion signifies the number of tests in that respective portion. In other words **your application should have many unit tests, a good number of integration tests and a small number of UI tests**.

People generally recommend you have somewhere around 70% unit tests, 20% integration tests and 10% UI tests. Doing this makes sure your tests run fast, reliably, and helps to make it easy to locate errors.

Let’s first clarify the terms: what are the differences between unit tests, integration tests, and UI tests?

### Unit Tests

Unit tests test a single **unit** of behavior. What exactly constitutes a unit depends on the codebase, but ideally a unit corresponds to a single method or single small class *with a single responsibility*.

**Unit tests should nearly always be short and test one thing and one thing only**, though they may do so by testing multiple possible input and output values.

**A good suite of unit tests often looks a lot like a set of examples of how to use a certain function or class properly.** In fact, reading unit tests is often a great way to learn how to use a library when the documentation is lacking.

**Being able to write good unit tests requires that your code is well structured.** If you are struggling with writing unit tests, it is a sign your code is not well factored: you likely have huge methods that do too many things, have not written enough small methods, etc. We will cover this more in the last section of the lab with a worked example and exercises!

**Whenever you add a new feature to an application, you should write unit tests for it as you develop it, and include them with it when you merge the changes into your shared repository.** These tests serve as an easy way to check your work in progress against some example inputs. The TDD (Test Driven Development) and BDD (Behavior Driven Development) styles take this to an extreme and advocate writing your tests *first*, then "going from red to green" by filling out your code so that they all pass. This is more common in scripting languages like Python and Ruby than in Java.

**Unit tests usually also serve as regression tests.** A regression test is not strictly a particular kind of test in the pyramid. Rather, it is a test intended to ensure that future changes don't break expected behavior (cause a "regression" in behavior).

Most (if not all) dependencies of the tested class are faked using test doubles in a unit test. Why? Because otherwise we wouldn't be testing just one unit, we'd be testing the integration of a unit with everything it depends on! Ideally, your code is written such that you need as few mocks as possible. (Optional reading: [Mocking is a Code Smell](#))

You already learned about test doubles in the previous lab and they will become even more important in future labs when we start to use other services. Remember that people use the terms "faked", "mocked" and "stubbed" interchangeably though they refer to different things in the [test doubles](#) definitions. In this lab we will use "mocking" as a general term for "creating/using a test double".

**Advantages:**

- Run very fast.
- Easy to find what is broken if a test fails.
- Easy to write (if your code is structured properly, more on this later).
- Can be used to guide development in "Test Driven Development" styles.
- Can be used to provide examples of how to use your code.

**Disadvantages:**

- You're not testing your entire application, only isolated parts.
  - See the GIF in the Integration Test section below.
- All dependencies are mocked, and might not give a realistic picture of things such as performance.

## Integration Tests

Integration testing means you test your code in a more realistic scenario where different parts (units) come together. It is a type of testing that focuses on testing how different parts of a software application work together. Unlike unit testing, which tests individual parts of the codebase in isolation, integration testing looks at how multiple parts interact with one another. Integration tests can be run on various levels, such as within a single module or across multiple modules, to ensure that everything is functioning as expected. Integration testing can be more complex and time-consuming than unit testing, but it is crucial for finding bugs and issues that might only arise when different components are combined. By testing the integration of multiple units, integration testing can help ensure that the entire application works together seamlessly.

Note that with an integration test, you have **not** yet tested your application **as a user**.

**Advantages:**

- Tests the *integration* of multiple units instead of testing them in isolation.

**Disadvantages:**

- Often slower and more complex than unit tests.
- Harder to find what is broken if a test fails.
  - Solution: write more unit tests, to isolate the integration itself as the point of failure!



*"The lock and door units both passed tests and work, but the integration..."*

## UI Tests

UI tests ensure that users do not encounter unexpected results or have a poor experience when interacting with your app. These tests are much slower than unit tests and most integration tests.

They also are prone to fail for weird reasons that are not related to your code being wrong. For instance if a certain animation takes longer than expected the test may fail because it wanted to click a button that had not yet appeared.

UI tests are an example of what is called an **end-to-end (E2E) test**. That is, they test a full scenario of user interaction from beginning to end.

### Advantages:

- Tests actual user interaction with your application's UI.
- Test application features instead of class methods, can translate user stories to test cases really easily.

### Disadvantages:

- Even slower than integration tests.
- Are prone to being "flaky" (not working consistently).
- Harder to find what is broken if a test fails.

## Other Kinds of Testing

There are also some other kinds of tests that we won't refer back to in the future but that you should be aware of as you will see them in the software industry. These are:

- [Smoke Testing](#)
  - Key idea: simply test "is there smoke?" (e.g. app doesn't run), then look for fire.
- [Performance Testing](#)
  - Key idea: does the app run well enough under expected workloads?

## Part 2: CI with GitHub Actions

### Project Setup

This lab comes with starter code. You will be working with a simple Queue class.

First, we will **import** the following repository to **your own GitHub account** (make one if you don't have one) and then to your computer.

**Do not clone this link directly, you cannot push changes to it!**

<https://github.com/ucsd-cse110-fa23/Lab-2-Test>

To **import** this repo, click "Import Repository" on GitHub.

## Import your project to GitHub

Import all the files, including revision history, from another version control system.

Required fields are marked with an asterisk (\*).


Support for importing Mercurial, Subversion and Team Foundation Version Control (TFVC) repositories will end on October 17, 2023. For more details, see the [changelog](#).

Your old repository's clone URL \*

Learn more about the types of [supported VCS](#).


### Your new repository details


Owner \*

 anmoltdtu

Repository name \*

✔ CSE110-Lab2 is available.

☐  **Public**  
Anyone on the internet can see this repository. You choose who can commit.

☒  **Private**  
You choose who can see and commit to this repository.

 You are creating a private repository in your personal account.

[Cancel](#)

[Begin import](#)

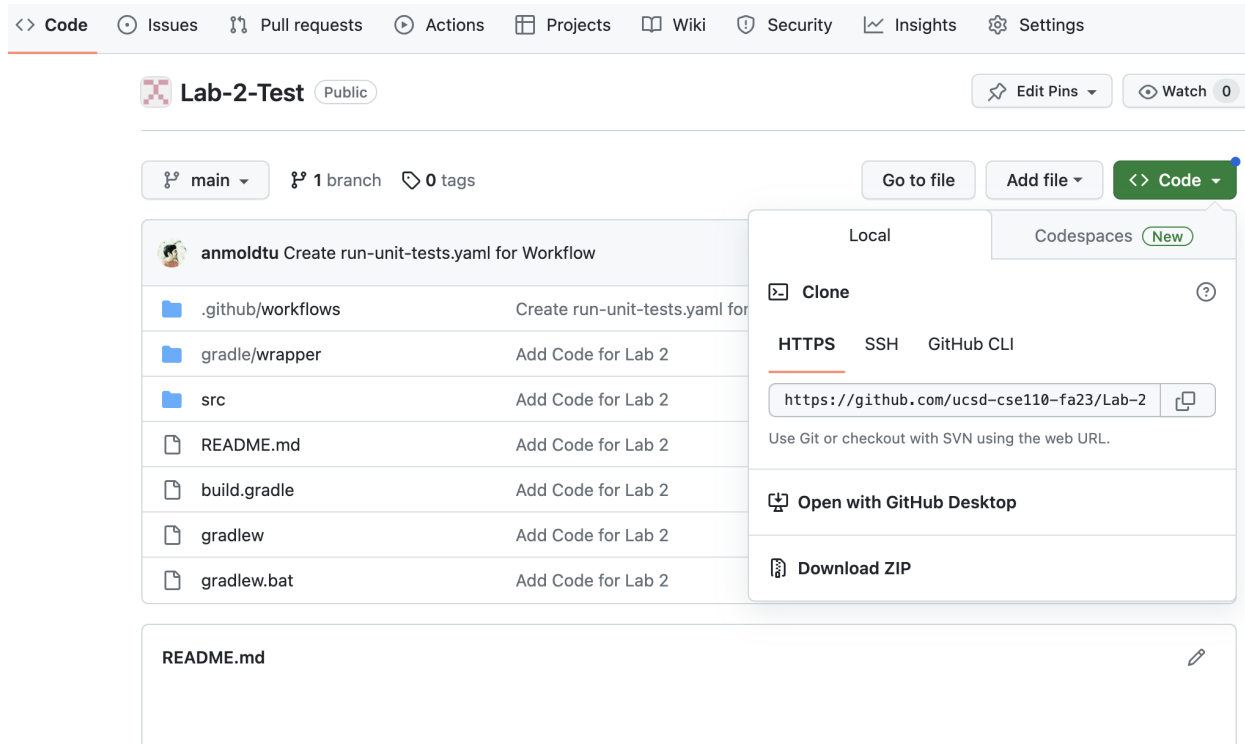
When you import this repository, make your repository **private**

Paste the URL for the lab starter code mentioned above into the “old repository” field like so. You can name your repository whatever you like.

Click *Begin Import* to proceed. Once your repository has been created, click the green *Code* button and copy the **HTTPS link** for your repository.

**Note:** later on and especially in your own projects, you will want to use the SSH link but this requires a bit of setup. We will cover this in the next lab when we thoroughly cover how to use Git. **If you already know how to do this and have your SSH keys set up, feel free to just use the SSH link.** If you want to jump ahead a little bit, you can follow the [instructions provided by GitHub to set up Git over SSH](#).

Remember, this should be the URL for **your imported copy** of the starter code.

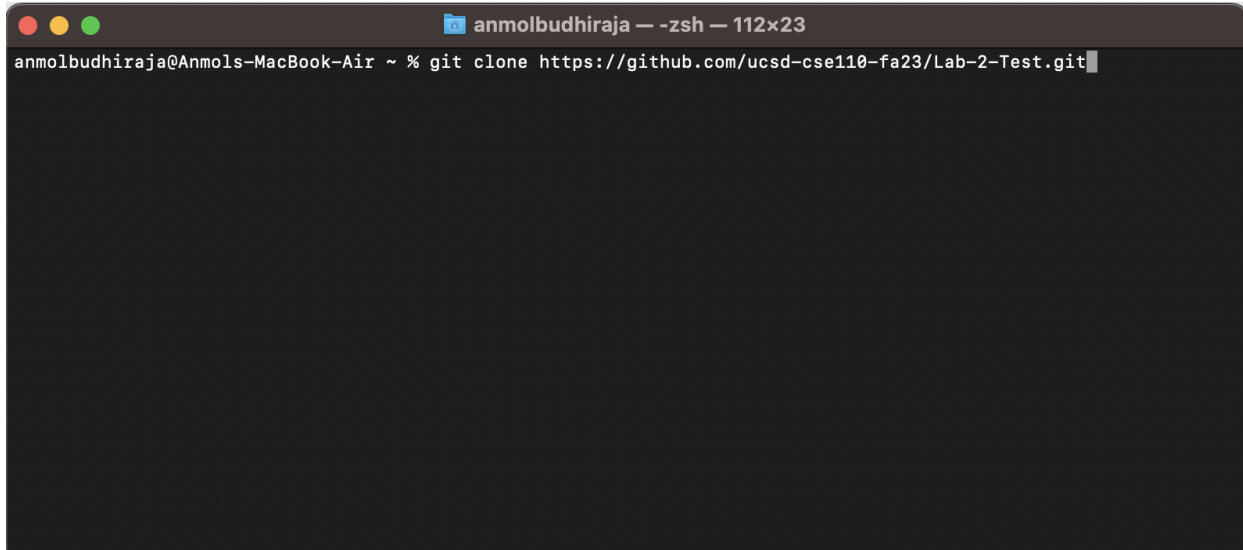


The screenshot shows the GitHub interface for a repository named "Lab-2-Test" (Public). The top navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. Below the repository name, there are buttons for "Edit Pins" and "Watch" (0). The main content area shows the repository structure with a file tree on the left and a list of files on the right. The file tree includes a "main" branch, 1 branch, and 0 tags. The file list shows a commit by "anmoltdtu" titled "Create run-unit-tests.yaml for Workflow". The files listed are: ".github/workflows" (Create run-unit-tests.yaml for Workflow), "gradle/wrapper" (Add Code for Lab 2), "src" (Add Code for Lab 2), "README.md" (Add Code for Lab 2), "build.gradle" (Add Code for Lab 2), "gradlew" (Add Code for Lab 2), and "gradlew.bat" (Add Code for Lab 2). The "Code" dropdown menu is open, showing options for "Local" and "Codespaces" (New). Under "Local", there is a "Clone" button with a question mark. Below "Clone", there are three tabs: "HTTPS", "SSH", and "GitHub CLI". The "HTTPS" tab is selected, showing the URL "https://github.com/ucsd-cse110-fa23/Lab-2" with a copy icon. Below the URL, it says "Use Git or checkout with SVN using the web URL." Under "Codespaces", there is an "Open with GitHub Desktop" button. At the bottom of the dropdown menu, there is a "Download ZIP" button.

Now you will use the command prompt / terminal to clone your repository so that you can make changes to the code in the repository locally. If you are using Windows, make sure you have installed Git for Windows. You can clone your repository by running the command `git clone [url to your repo]`

**Your URL should NOT MATCH the one shown below. You should be cloning YOUR OWN repo.**



A terminal window with a dark background. The title bar at the top shows three colored window control buttons (red, yellow, green) on the left, followed by the text "anmolbudhiraja — zsh — 112x23". The terminal content shows a command prompt "anmolbudhiraja@Anmols-MacBook-Air ~ %" followed by the command "git clone https://github.com/ucsd-cse110-fa23/Lab-2-Test.git" which has been executed, as indicated by a cursor at the end of the line.

```
anmolbudhiraja@Anmols-MacBook-Air ~ % git clone https://github.com/ucsd-cse110-fa23/Lab-2-Test.git
```

Now that you have cloned your repo and have a copy of the code in your local computer, open it up in VSCode or Eclipse (or any other IDE/editor you use for Java projects).

**Note:** Make sure that you have installed JUnit for your IDE as mentioned in the [Setup Documentation](#) in the Lab 1. JUnit is necessary for performing Unit Testing in this Lab.

## Github Actions

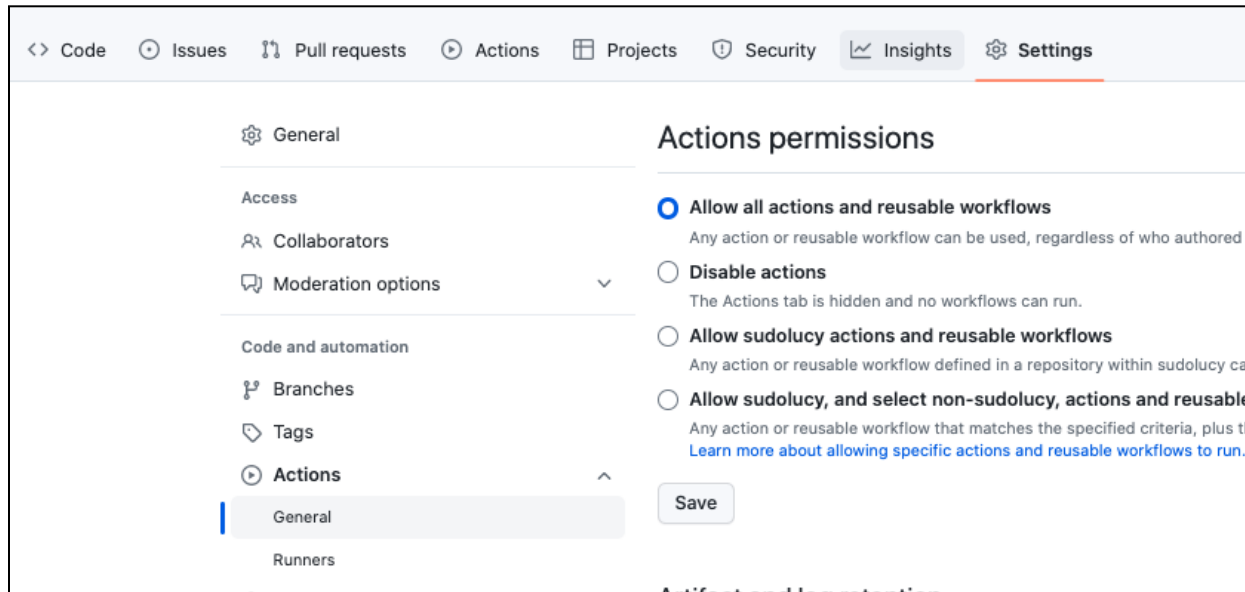
Github Actions are a system for automating things like testing on Microsoft provided servers. For example, maybe I want “every time I ask to merge in new code, it should be tested and the result included in my request”.

This is an example of what we call **continuous integration**. Continuous integration is **not** just about merging everyone’s work every day as you may find online (such as on Wikipedia), but it is also about ensuring that **everyone can get code that builds, runs and passes tests** (if they pull from main).

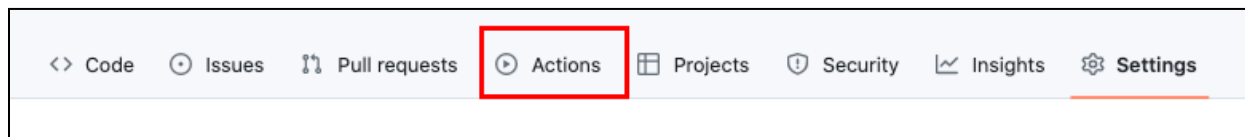
**We never merge broken code into main! We always have a running main branch!**

**Note:** you may need to first enable Actions as follows.

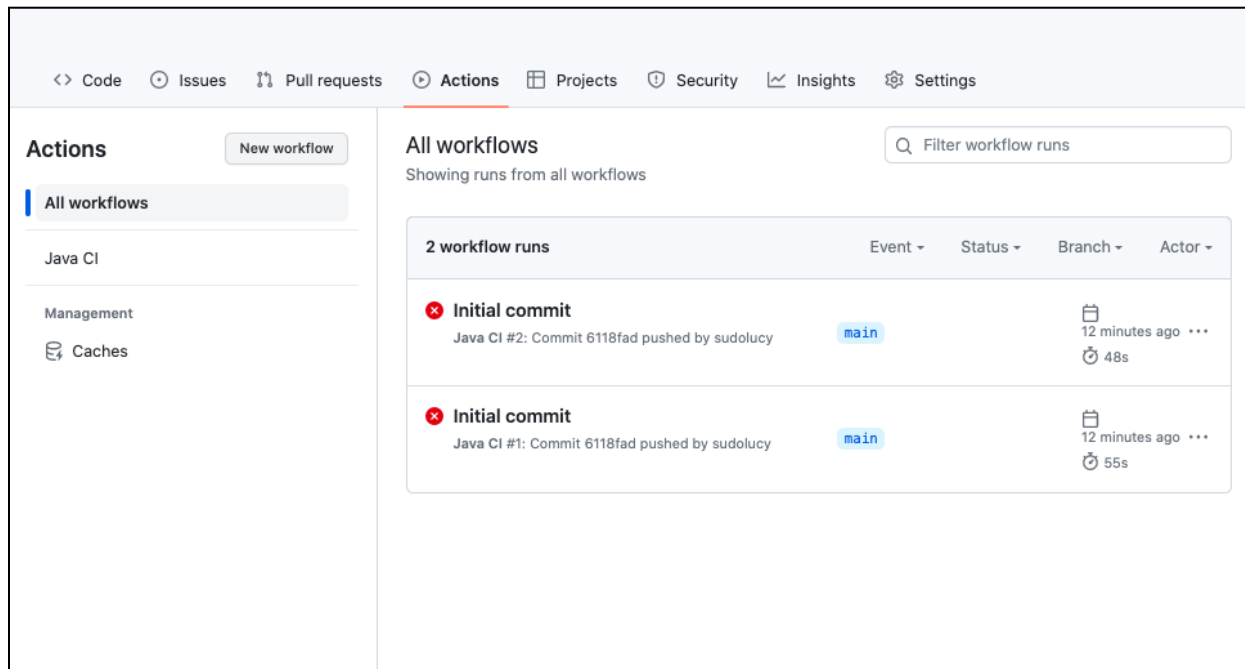
Go to the *Settings* tab on your repository’s page. Click on *Actions* and then click on *General*. Enable “*Allow all actions and reusable workflows*” if it is not enabled. Finally, click on *Save*.



You should now see **Actions** in the main GitHub menu bar for your repository:



Click it and let's take a look. You should see something like this (don't worry if you don't see the "2 workflow runs" section yet).



**Note: Make sure you have a `.github/workflows/run-unit-tests.yaml` file in your repository, otherwise you won't see this screen. This is present in the source project, if by any chance it is not present while importing, copy the file to your repository.**

The starter repository already comes with a workflow called "Java CI". It's set up to run all of the **unit tests** (and only unit tests) each time you push your code. Unfortunately, running UI tests on GitHub actions is prohibitively expensive, so we're going to avoid doing it in this course. This is because by default, GitHub only provides a few thousand minutes of runtime per month, and running UI tests takes a lot of time, and requires using a special machine that bills at 5x!

## Special file for Github Actions

How is this set up? By a special file in the repository:  
`.github/workflows/run-unit-tests.yaml`.

! run-unit-tests.yaml ×

.github > workflows > ! run-unit-tests.yaml > {} jobs > {} build

GitHub Workflow - YAML schema for GitHub Workflow (github-workflow.json)

```
1  name: Java CI
2
3  on: [push]
4
5  permissions:
6    contents: read
7    checks: write
8    id-token: write
9
10 jobs:
11   build:
12     runs-on: ubuntu-latest
13
14     steps:
15       - uses: actions/checkout@v3
16
17       - name: Set up JDK
18         uses: actions/setup-java@v3
19         with:
20           distribution: 'temurin'
21           java-version: '11'
22
23       - name: Validate Gradle wrapper
24         uses: gradle/wrapper-validation-action@v1
25
26       - name: Setup Gradle
27         uses: gradle/gradle-build-action@v2
28
29       # build everything (includes running unit tests)
30       - name: Execute Gradle build
31         run: ./gradlew build
```

```

32
33     # report the results
34     - name: Publish Test Report
35       uses: mikepenz/action-junit-report@v3
36       if: success() || failure() # always run even if the previous step fails
37       with:
38         report_paths: '**/build/test-results/test/TEST-*.xml'
39         detailed_summary: true # display detailed summary of the report
40         include_passed: true # include passed tests in the results table

```

This file specifies when to run the workflow (on push), [the necessary permissions](#) and then the steps for the test job:

1. Check out a copy of the code onto the machine running the workflow.
2. Set up a copy of JDK 11.
3. Validate and sets up the build tool
4. Build and run the unit tests.
5. Report the results.

You can read more about **building and testing Java project with Gradle** here (Also talks about the yaml file):

<https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-gradle>

Now, let's try to get it to run. To do so, we need to make some changes to the project. Since there is nothing on the README.md file, let's write something in it:

```

i README.md M X
i README.md > # CSE110 FA23 Lab 2!
1 # CSE110 FA23 Lab 2!
2 |

```

When you save the README file and run `git status` on the command prompt (in your local repository), you will see that it is now modified:

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

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:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Now, stage the change by running -

```
git add README.md
```

Then, commit the changes by running -

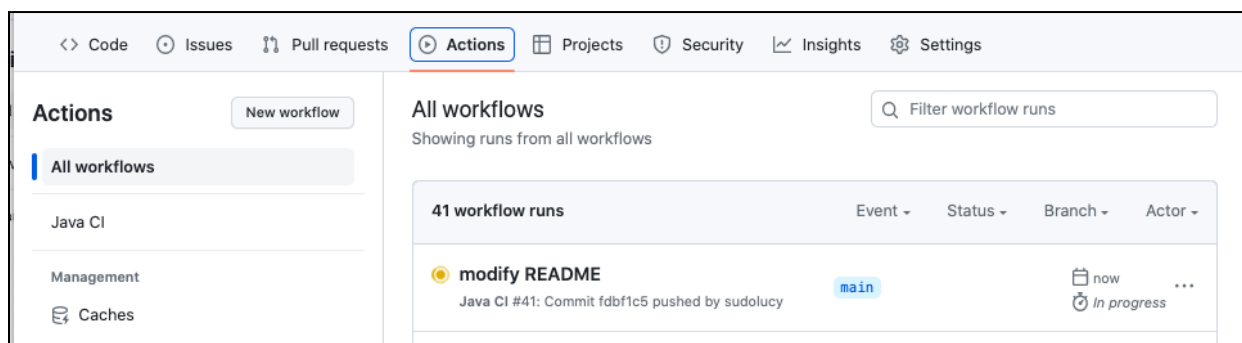
```
git commit -m "modify README"
```

Finally, push the change to the remote repository by running -

```
git push
```

**Note:** If it asks for your password, you will need to paste in a [Personal Access Token](#). If you already have [your own SSH key set up and used the SSH link before](#), you won't have to bother with this.

Go back to the *Actions* panel in GitHub. You should see that the workflow has started to run, as shown in the screenshot above (reproduced below again).




And now... **you wait. This will take about 40 seconds.** You can follow along by clicking on the workflow run.




## Resolving errors:

**Note: If you see the report page like the following instead of the test summary (as shown below), click on 'build' to see more details of the error.**

**Annotations**  
1 error

 **build**  
Process completed with exit code 126.

**build summary** ...

	Tests	Passed 	Skipped 	Failed 
JUnit Test Report	0 ran	0 passed	0 skipped	0 failed

	Test	Result
-	No test annotations available	-

► Caching for gradle-build-action was enabled - expand for details

**If the error is `./gradlew: Permission denied`, then use the following commands to fix the error:**

```
git update-index --chmod=+x gradlew
git add .
git commit -m "Changing permission of gradlew"
git push
```

**If this doesn't work, please try the following:**

**Go to `.github/workflows/run-unit-tests.yaml` file, and add the following lines:**

```
23 - name: Validate Gradle wrapper
24 | uses: gradle/wrapper-validation-action@v1
25
26 - name: Setup Gradle
27 | uses: gradle/gradle-build-action@v2
28
29 - name: Change wrapper permissions
30 | run: chmod +x ./gradlew
31
32 # build everything (includes running unit tests)
33 - name: Execute Gradle build
34 | run: ./gradlew build
35
```

This happens when certain files lack the execution permissions after the repository gets imported and cloned. To resolve this error, we are simply granting execution permissions to the gradlew file.

### Viewing the test report:

When it's done running, you can view the test report by clicking on the *Summary* tab on the left pane of your test run. You should see that there are some tests **failing**. You can view the details by scrolling down to the generated report:



✖ modify README #41

Re-run jobs ⌵ ⋮

Summary

Jobs

Run details

✖ build

✖ JUnit Test Report

Usage

Workflow file

Triggered via push 4 minutes ago

sudolucy pushed → fdbf1c5 main

Status

Failure

Total duration

41s

Billable time

1m

Artifacts

—

run-unit-tests.yaml

on: push

✖ build

32s

Annotations

5 errors

✖ MyQueueTest.testEnqueueWhenFull(): src/test/java/edu/ucsd/cse110/lab2/MyQueueTest.java#L70

java.lang.IllegalStateException: Queue is full

✖ MyQueueTest.testPeek(): src/test/java/edu/ucsd/cse110/lab2/MyQueueTest.java#L42

org.opentest4j.AssertionFailedError: expected: <1> but was: <3>

✖ MyQueueTest.testDequeue(): src/test/java/edu/ucsd/cse110/lab2/MyQueueTest.java#L34

org.opentest4j.AssertionFailedError: expected: <1> but was: <5>

✖ MyQueueTest.testIsFull(): src/test/java/edu/ucsd/cse110/lab2/MyQueueTest.java#L60

java.lang.IllegalStateException: Queue is full

Summary

Jobs

Run details

✖ build

✖ JUnit Test Report

Usage

Workflow file

✖ build

Process completed with exit code 1.

build summary

⋮

	Tests	Passed	Skipped	Failed
JUnit Test Report	7 run	3 passed	0 skipped	4 failed

	Test	Result
JUnit Test Report	MyQueueTest.testEnqueue()	pass
JUnit Test Report	MyQueueTest.testEnqueueWhenFull()	failure
JUnit Test Report	MyQueueTest.testDequeueWhenEmpty()	pass
JUnit Test Report	MyQueueTest.testPeek()	failure
JUnit Test Report	MyQueueTest.testDequeue()	failure
JUnit Test Report	MyQueueTest.testIsFull()	failure
JUnit Test Report	MyQueueTest.testIsEmpty()	pass

Gradle Builds

Root Project	Requested Tasks	Gradle Version	Build Outcome	Build Scan™
CSE110-SP23-Lab2-draft	build	7.4.2		Build Scan™ NOT PUBLISHED

The above test report is generated because we specified the step to generate it in the `.github/workflows/run-unit-tests.yaml` file:

```
32
33     # report the results
34     - name: Publish Test Report
35       uses: mikepenz/action-junit-report@v3
36       if: success() || failure() # always run even if the previous step fails
37       with:
38         report_paths: '**/build/test-results/test/TEST-*.xml'
39         detailed_summary: true # display detailed summary of the report
40         include_passed: true # include passed tests in the results table
```

To publish this test report we use: <https://github.com/marketplace/actions/junit-report-action>  
(Not required to read, but feel free to check out if interested)

## Exercise and Questions

### Exercise 1

Fix the failing unit tests and ensure it passes **both locally and on GitHub Action**. The tests are actually written perfectly fine. The problem is with the `MyQueue` class itself. Navigate and fix the problems in `MyQueue` class, and check by running the unit tests.

**You should work and test locally first**, then once you have it working push and have CI run your tests. Once it's working locally, push your fixed code to GitHub and **ensure that the GitHub action works and produces the same output as what you see locally**.

### Question 1

What is, do you think, one big benefit of continuous integration to a software team?

### Question 2

In your own words, what is the difference between a unit test and an integration test? Which kind(s) of tests could serve as regression tests?

### Question 3

In your own words, what is a regression test and why is it important?

## Question 4

What does it mean if you are struggling to write unit tests?

## Question 5

Why is it important to fake dependencies in unit tests?

## Question 6

Why is it important to write UI tests? What role do they serve that unit/integration tests alone do not cover?

## Question 7

How do you set up continuous integration using Github Actions (What kind of file do you need to use and where should you store it?)

**Note: do not just paste in what is written above. Read carefully and formulate your own response.**


## Check-Off Instructions




**First**, submit your [check-off form](#) with your answers to the questions above.

**Next**, submit a ticket on Autograder. You should be ready to demonstrate the following:








1. Exercise 1: The provided example unit test passes and runs both locally and on Github Actions  
Should look like this on Github Actions:

**run-unit-tests.yaml**  
on: push

 build
 24s

build summary				
	Tests	Passed 	Skipped 	Failed 
JUnit Test Report	7 run	7 passed	0 skipped	0 failed

	Test	Result
JUnit Test Report	MyQueueTest.testEnqueue()	 pass
JUnit Test Report	MyQueueTest.testEnqueueWhenFull()	 pass
JUnit Test Report	MyQueueTest.testDequeueWhenEmpty()	 pass
JUnit Test Report	MyQueueTest.testPeek()	 pass
JUnit Test Report	MyQueueTest.testDequeue()	 pass
JUnit Test Report	MyQueueTest.testIsFull()	 pass
JUnit Test Report	MyQueueTest.testIsEmpty()	 pass

- Be able to explain the answers you wrote for the questions above

**Do not submit a “checkoff” ticket to Autograder if you are not ready to go.** We work on support and check-off tickets separately, so submitting a ticket marked as check-off when you still need support can really slow down check-offs for your classmates! Thanks!