Setting Up Your CSE 332 Environment

This document guides you through setting up IntelliJ for CSE 332 in various parts. If you run into any problems or questions when setting up or using any of these plug-ins, feel free to ask on Piazza or during office hours!

- Part 1: Downloading Tools
- Part 2: Setting Up Gitlab
- Part 3: Running Tests
- Part 4: Submitting Projects and using Git
- Part 5: Using FlooBits for Peer Programing (optional)
- Part 6: Visualizer to Debug (Optional)

Part 1: Downloading Tools

The first thing you should do is download the tools needed to work locally.
Note: Skip any steps if you already have the tool downloaded

(1) Download Java 11 from here. Don't download the JRE (Java Runtime Environment), which lets you run Java programs, but does not provide the tools for Java development. We recommend downloading Java 11 but Java 12 and up will possibly work too if you have those versions already downloaded.

(2) Download Git

(3) Download IntelliJ

- As a student, you can get a free education account and can download either version of IntelliJ IDEA for free

Part 2: Setting Up GitLab

We will be using `gitlab.cs.washington.edu` to submit homeworks and give feedback. `gitlab` is a web-based `git` hosting service which is similar to `github` but hosted locally by the CSE Department. You will need to learn basic `git` to be able to work on and submit your homework.

Generating a SSH key

The first time you clone from `gitlab`, you will need to create an *SSH key*. To do this, follow these steps:

(1) Open a Terminal on Linux or macOS, or Git Bash / WSL on Windows.

(2) Run the following command with your UW netID.

- `ssh-keygen -o -trsa -b4096 -C "yourUWnetID"`

(3) Next, you will be prompted to input a file path to save your SSH key pair to. If you don't already have an SSH key pair and aren't generating a deploy key, use the suggested path by pressing Enter. Using the suggested path will normally allow your SSH client to automatically use the SSH key pair with no additional configuration.

(4) Once the path is decided, you will be prompted to input a password to secure your new SSH key pair. It's a best practice to use a password, but it's not required and you can skip creating it by pressing **Enter** twice.

Adding SSH to your GitLab account

The first time you clone from `gitlab`, you will need to create an *SSH key*. To do this, follow these steps:

(1) Open a terminal on Linux or macOS, or Git Bash / WSL on Windows.

- macOS: `pbcopy < ~/.ssh/id_rsa.pub`
- Linux: `xclip -selclip < ~/.ssh/id_rsa.pub`
- Bash on Window: `cat ~/.ssh/id_rsa.pub`

(2) Add your public SSH key to your GitLab account by clicking your avatar in the upper right corner and selecting Settings. From there on, navigate to SSH Keys and paste your public key in the **Key** section. Your UW netID should appear under **Title**. If not, give your key an identifiable title like Work Laptop or Home Workstation. Make sure your key does not end in a blank line and click Add key.

(3) Run `ssh -T git@gitlab.cs.washington.edu` to ensure that your key is correctly setup. You should receive a welcome message and should not be prompted for a password.

Creating The IntelliJ IDEA Project

Each project in the course will have its own project in IntelliJ. We will create the project in IntelliJ by cloning a `git` repository on `gitlab` which contains the starter code. To do this, follow these steps:

(1) If no project is currently opened, choose **Checkout from Version Control** | **Git** on the Welcome screen. Otherwise, from the main menu, choose **VCS** | **Checkout from Version Control**.

(2) In the Clone Repository dialog, specify the URL of the remote repository you want to clone.

- `git@gitlab.cs.washington.edu:cse332-`**20su/p1-animal**`.git`

A **bold** phrase in the repository path means you should substitute it with your equivalent. Do *not forget* the **.git** at the end!

(3) In the **Directory** field, specify the path where the folder for your local Git repository will be created into which the remote repository will be cloned.

(4) Click **Clone** and click **Yes** in the confirmation dialog.

- A pop-up may appear asking you to open the project after check-out, click **Yes** to open the project.
- Another pop-up may appear notifying you that the SDK is not set-up or configured, click **Yes** and the SDK will be set up later.

(5) Open any java class in **p1-animal** | **src**

(6) If the java class has a lot of red underlining the code, you have to set up the SDK. A pop-up may appear notifying you that "Module SDK is not defined". Click **Setup SDK**, choose whatever version of Java you are using as long as it is Java version 11 or above, and then click **Yes**.

Part 3: Running Tests in IntelliJ

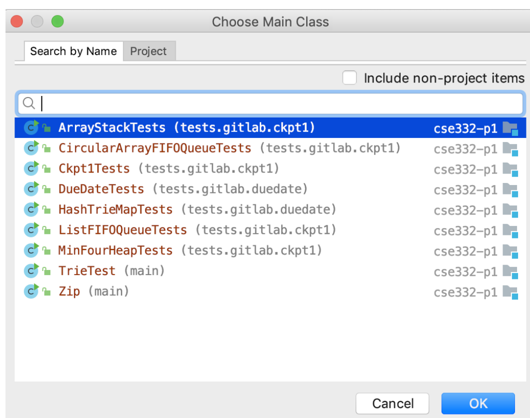
Here are a few ways to run your tests in IntelliJ.

```
4   import cse332.interfaces.worklists.WorkList;
5   import datastructures.worklists.CircularArrayFIFOQueue;
6   import org.junit.Before;
7   import org.junit.Test;
8
9   import static org.junit.Assert.*;
10
11  public class CircularArrayFIFOQueueTests extends WorklistGradingTests {
12      public static final int DEFAULT_CAPACITY = 1000;
13
14      @Before
15      public void init() {
16          STUDENT_STR = new CircularArrayFIFOQueue<>(DEFAULT_CAPACITY);
17          STUDENT_DOUBLE = new CircularArrayFIFOQueue<>(DEFAULT_CAPACITY);
18          STUDENT_INT = new CircularArrayFIFOQueue<>( capacity: 100000);
19      }
20
21      @Test(timeout = 3000)
22      public void testClear() {
23          FixedSizeFIFOWorkList<String> queue = new CircularArrayFIFOQueue<>( capacity: 5);
24          addAll(queue, new String[] {"Beware", "the", "Jabberwock", "my", "son!"});
25          assertTrue(queue.hasWork());
26          assertTrue( condition: queue.size() == 5);
27          assertTrue(queue.isFull());
28          assertTrue( condition: queue.capacity() == 5);
29
```

- Click the **green arrow** near the line numbers. To run all the tests in a file, click on the green arrow next to the class name (ej. line number 11 in the picture above). To run an individual test in a test file, click on the green arrow next to the individual test name (ej. line number 22 in the image above. Then select "Run TestFile".

- Right Click the Test File name and select "Run TestFile"

- Click **Add Configuration** on the upper right side corner next to the hammer icon. In the "Run Configuration" Pop-up, click the **+** button and select **Application**.

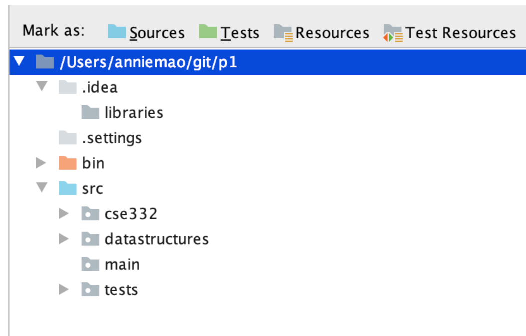


Specify the main class as the test file you want to run. You can click **...** on the right side to select from all the test file options.

Common IntelliJ Problems

- If you are unable to run tests (No green arrows in test files and/or no test file options...

  Go to **Project Structure** | **Modules** | **Sources** to mark **src** as a Source Folder. The end result should look like the image below.



- If IntelliJ cannot find certain files when you run tests...

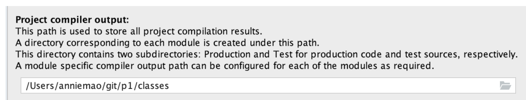
  Go to **Project Structure** | **Libraries**. Then go to the + | **Library** | **Java**, select the **hamcrest-core-1.3.jar**, and press **OK**. Repeat with all other jar files in the repo (ej. junit-4-12.jar, SuffixTrie.jar for p1, Ab.jar/chatter.jar for p2).

- If you have problems with the Compiler Output...

  Go to **Project Structure** | **Project**. You should find the location of your repo, you can do this by right clicking the project folder in IntelliJ (ej. p2-blahblah) and then clicking **Copy Path**. The location may be in a folder called git or IdeaProjects. This is my personal location "/Users/anniemao/git/p2-bundtcake/".

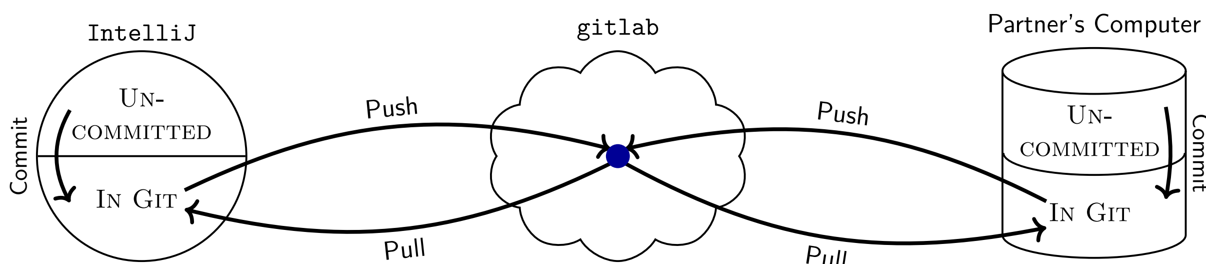
  Paste the location of your repo into the **Project Compiler Output** and then add "/classes" (with no quotes) to the end. Press **Apply** | **OK**.

  The end result should look something like the image below...



Part 4: Submitting Projects and using Git

git is the *version control system* (VCS) underlying gitlab. Most real projects are kept in a VCS to avoid losing data, and for the ability to easily revert code back to older versions. Another major reason VCS's are important is that they allow you to effectively work together with other people. They allow you to combine ("merge") several different versions of your codebase together.



As shown in the diagram, there are several major actions you can do with respect to your git repository:

- **Commit:** A “commit” is a set of changes that go together. By “committing” a file, you are asking `git` to “mark” that it has changed. `git` requires that you give a message summarizing the effect of your changes. An example commit message might be “Add error handling for the empty queue case in ListFIFOQueue”.
- **Push:** A “push” sends your commits to another version of the repository (in our case, this will almost always be `gitlab`). If you do not push your commits, nobody else can see them!
- **Pull:** A “pull” gets non-local commits and updates your version of the repository with them. If you and someone else both edited a file, `git` will ask you to explain how to merge the changes together (this process is called “resolving a merge conflict”).

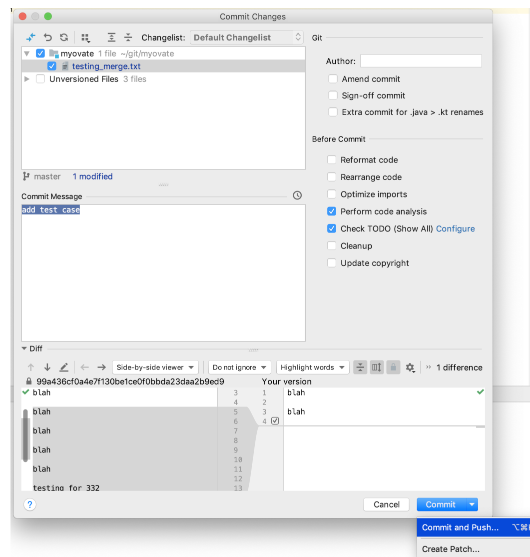
Using Git in IntelliJ

IntelliJ provides a GUI for all of the `git` operations. We now explain how to handle a `git` workflow in IntelliJ.

Committing and Pushing

After making changes to, adding, or removing files, you must commit and "push" your changes to git. This step will cause git to record your changes to the repository, so that your changes are backed-up and available to other people working on the repository, or to you when working on a different computer system.

(1) Click the green checkpoint in the upper right corner or in the main menu **VCS** | **Commit**



(2) Select all the files to commit (all the files you changed should be selected by default) and add an appropriate commit message.

(3) Press the **down** arrow next to the "Commit" button, and select **Commit and Push**

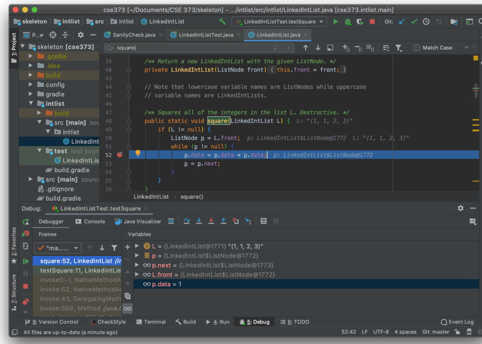
(4) Note if you accidentally press the "Commit" button, push manually by going to **VCS** | **Git** | **Push**

Do NOT commit the .idea folder. This will cause issues for your partner down the line.

Pulling

There are two reasons to pull: (1) you want to get the changes your partner made, or (2) you want to push your changes, but they were rejected because of a conflict.

(1) To pull in IntelliJ, from the main menu, choose **VCS** | **Update Project** or click on the **blue arrow** on the upper right side corner.



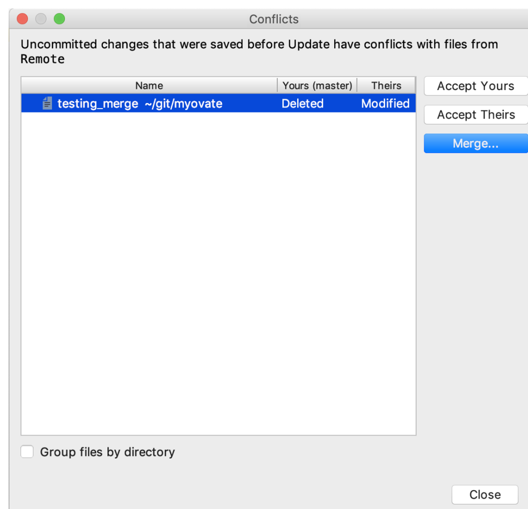
(2) The following window should appear. The default selections should be **Merge** and **Using Stash**, if not, change "Update Type" and "Clean working tree before update" to them respectively.

The result will either indicate that you pulled cleanly or that there is a merge conflict. If you have a merge conflict, then a conflict pop-up will appear, which means you have uncommitted local changes which conflict with changes on `gitlab`. Refer to the next section, Merging a Conflict.

Merging a Conflict

If you have any items that have a conflict and you select the **Merge...** option, there are two ways to resolve the conflict: (1) Use the *merge tool* that appears (2) Exit out of the *merge tool* and resolve the conflicts manually

(1) Using the merge tool



- If you want to use your version of the file, select **Accept Yours**.
- If you want to use your partner's version on the file, select **Accept Theirs**.
- If you want to merge your version and your partner's version of the file, select **Merge...**. The *merge tool* should appear. Refer to the IntelliJ Help to resolve the conflict using the *merge tool*.

(2) Resolving the conflict manually

When git detects a file conflict, it changes the file to include both versions of any conflicting potions in this format:

```
<<<<<<< filename
YOUR VERSION
=======
REPOSITORY'S VERSION
>>>>>>> 4e2b407... -- repository version's revision number
```

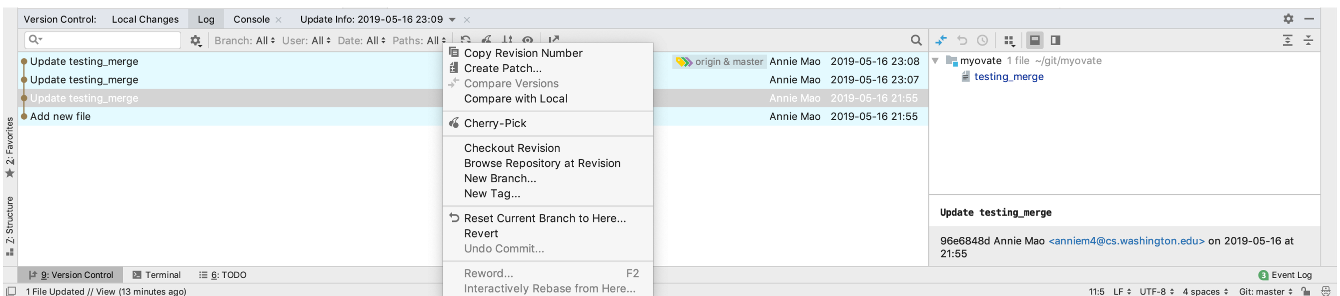
Each conflicting file will be colored in Red. For each conflicting file, edit it to choose one of the versions (or to merge them by hand). Be sure to remove the <<<<<<<, =======, and >>>>>>> lines. (Searching for <<< until you've resolved all the conflicts is generally a good idea.)

Don't forget to commit and push these changes as you normally would once you have made these edits, so that you can tell git that you have resolved the conflicts. Note that if there are still conflicting parts of the file (<<<<<<<, =======, and >>>>>>> lines still remaining), the commit will fail.

Revert a commit

(1) Open the **Version Control** tool window at the bottom left corner

(2) Select the **Log** tab



(3) Locate the commit you want to revert, right click, and select **Revert**

(4) Select the files you want to revert, and deselect the files you do not want to revert

(5) Press the down arrow next to the "Commit" button, and select **Commit and Push**

Reset a Branch to a Specific Commit

(1) Open the **Version Control** tool window at the bottom left corner

(2) Select the **Log** tab

(3) Locate the commit you want to revert, right click, and select **Reset Current Branch to Here**

(4) In the Git Reset dialog that opens, select **Mixed** and the index to be updated and click **Reset**.

Submitting Your Final Version

In some courses, you are asked to "tag" your final commit. We will not ask you to do this. The last commit you make within the deadline will be the one we grade.

Part 5: Peer Programming using FlooBits (Optional)

FlooBits is a real time collaborative editing plug-in for IntelliJ, so that multiple users can simultaneously edit the same project remotely. Installing FlooBits is **OPTIONAL**. If you and your partner prefer to meet in person for peer programming or an alternative method for collaboration, you do not have to install FlooBits. This guide will help you get started with FlooBits and using it to peer program with your CSE 332 project partner.

Installing Floobits

(1) On Windows, go to **Settings** | **Plug-ins**, on Mac, go to **Preferences** | **Plug-ins**

(2) Search for FlooBits and install the Plug-in. Researt IntelliJ when finished.

(3) After IntelliJ restarts, a pop-up should appear and select **Create a FlooBits Account**

(4) Use your UW email to create a FlooBits account. Make sure to select the "Free" Plan.

(5) Verify your UW email to receive 3 private workspaces for free. One for each project!

Only ONE partner should upload the project.

Uploading your Project

(1) From the main menu, choose **Tools** | **Floobits** | **Share Project Privately**

(2) Select **Upload Entire Project**

(3) Share the FlooBits URL of your project to your partner- it should be https://floobits.com/**username**/**projectname**
Note the text in bold should be different

(4) Log into FlooBits in your browser, go to your project, and then **Permissions**

(5) Add your partner to permissions using their FlooBits username

Joining your Project

(1) From the main menu, choose **Tools** | **Floobits** | **Join Workspace by URL**

(2) Use the FlooBits URL that your partner shared

Tips for working in FlooBits

(1) You can follow your partners changes by clicking on the eyeball in the bottom left

(2) Both users can make code changes but only the owner of the workspace can use Git to commit and push

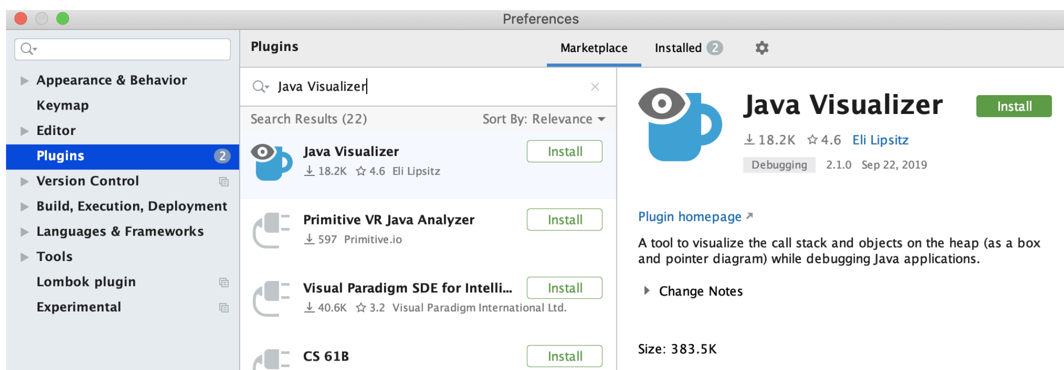
Part 6: Visualizer to Debug (optional)

Java Visualizer is a way to visualize objects using box and point diagrams, similar to the debugger used in JGRASP in the 14X series. Installing the Java Visualizer is **OPTIONAL**. It can be helpful to debug projects if you prefer visualizing the data structure. This guide will help you get started with the Java Visualizer.

Installing Java Visualizer

(1) On Windows, go to **Settings** | **Plug-ins**, on Mac, go to **Preferences** | **Plug-ins**

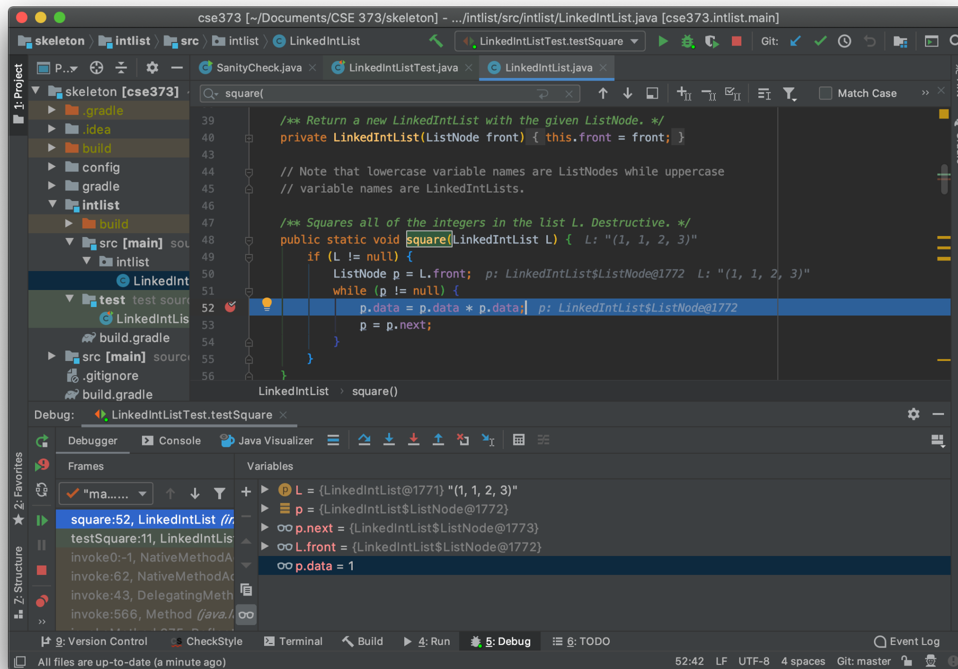
(2) Search for and install **Java Visualizer**.



(3) Restart IntelliJ

Using Java Visualizer

(1) Add a breakpoint by clicking near the line number of the line you want to pause execution on. A red stop sign icon should appear next to the line number.



(2) Repeat steps to run tests in Part 3, but select **Debug TestFile** (It should have a green bug icon with it).

(3) Click on the **Java Visualizer** buttom at the bottom of the Debug tool window. A box-and-pointer diagram representing the paused program state should appear. Step though the execution of the program, or use the green continue icon on the left side of the Debug tool window.