

---

# CSE 331

# Software Design & Implementation

Winter 2023

Section 2 – Development Tools

# Administrivia

---

- HW1 due this past Tuesday
- HW2 due Tuesday, 1/17, at **11PM**
- HW3 due Thursday, 1/19 at **11PM**
  - Our first programming assignment!
- Come to office hours!

# Course resources

---

- We can't cover everything in an hour
- Read documentation: [cs.uw.edu/331](https://cs.uw.edu/331) > “Resources” tab
  - [“Project Software Setup”](#)
  - [“Editing, Compiling, Running, and Testing Java Programs”](#)
  - [“Version Control \(Git\) Reference”](#)
  - [“Assignment Submission”](#)
- The resources page is a treasure trove of helpful information!

# Software You Need

---

- Java 17
  - [adoptium.net/temurin/releases/](https://adoptium.net/temurin/releases/)
  - Install **jdk-17.0.5+8** with the **JDK installer** for your OS
  - Windows: Select "Add to PATH" and "Fix Registry" during install
    - Uninstall old Java versions
  - MacOS: use the right installer if you have M1/M2
- IntelliJ
  - [jetbrains.com/idea](https://jetbrains.com/idea)
  - Recommended: Ultimate version
    - Free for students, see course website for link to license
  - **Install the latest version**
- Git
  - [git-scm.com](https://git-scm.com)
  - (Might be slightly newer version than the XCode command line tools on macOS if you have those installed, but command line version is all you need)
  - Comes with Git Bash on Windows – important!

# Warning: You must use JDK 17

---

- Must use JDK version 17
  - Be sure that's what you have installed!
  - Download links in Resources webpage
  - Use the Adoptium installers (only)
- An out-of-date JDK can lead to very confusing bugs
  - No fun for either of us!
- Don't use more recent versions either!



# IntelliJ

---

- The officially supported IDE/editor for this course
  - Full setup instructions in “Project Software Setup” handout
- A modern IDE, commonly used in industry
  - Get the “Ultimate” version – free license for education use
- IDE = “Integrated Development Environment”
  - Auto completion
  - Version-control (git) integration
  - Debugger integration
  - ...and an assortment of other fun features
- Necessary functionality covered in course documentation
  - “Editing, Compiling, Running, and Testing Java Programs”

# Version control

---

- Also called source control, revision control
- System to track changes in a project codebase
  - Unit of change ~ lines inserted/deleted across some files
- Essential for managing software projects
  - Maintain a history of code changes
  - Revert to an older project state
  - Merge changes from multiple sources
- We'll use **git** and **GitLab** in this course, but alternatives exist
  - Subversion, Mercurial, CVS
  - Email, dropbox, thumbdrives (don't even think of doing this!)



# Version control concepts

---

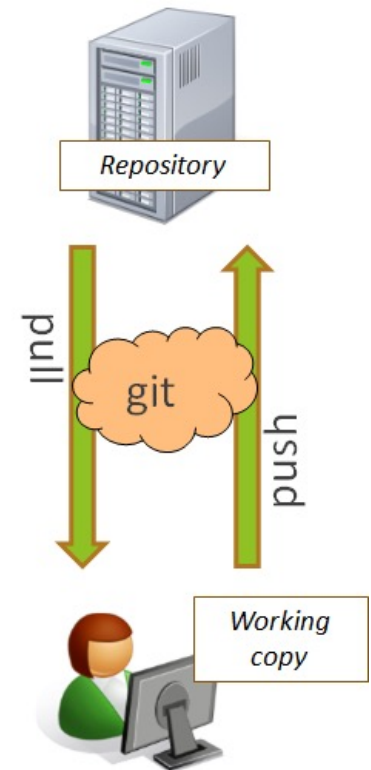
- A **repository** (“repo”) stores a project’s entire codebase
  - Stored in multiple places and synchronized over the internet
  - Tracks the files themselves and changes to them over time
- Each developer **clones** her own **working copy** of the repo
  - Makes a local copy of the codebase, on her laptop/computer
  - She modifies these files directly, with her IDE or text editor
- Each developer **commits** changes to her working copy
  - Saves “a commit” to version control history
  - Affects only the local working copy
  - Must synchronize with remote repo to share commits each way



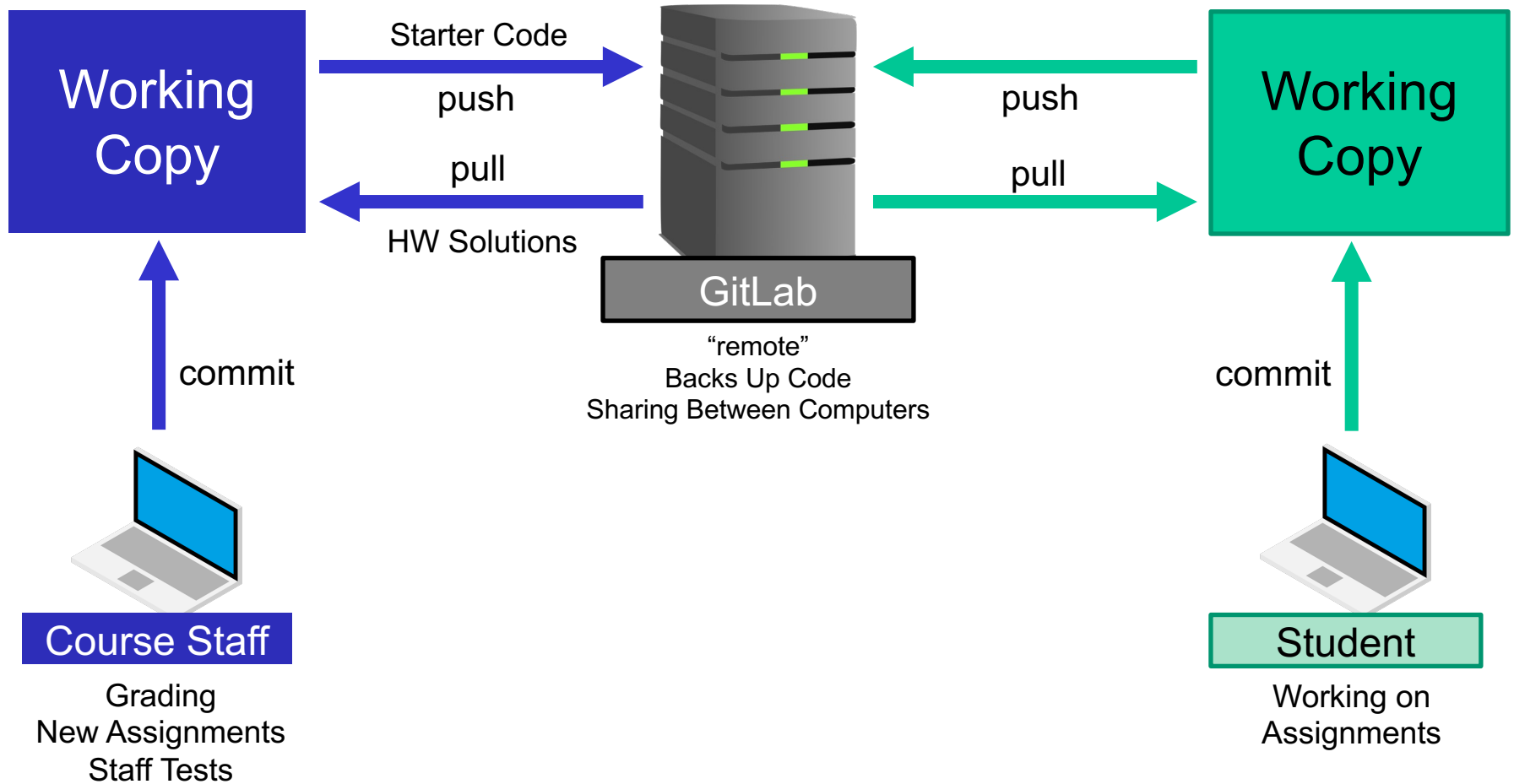
# Essential git concepts

---

- **commit**
  - Saves (a subset of) the changes to the local repository
  - Has a brief message summarizing changes
- **push**
  - Sends local commits to the repository (on GitLab)
  - Allows other computers to then “pull” those commits/changes, see below.
- **pull**
  - Synchronizes working copy to match the remote repository
  - **clone** = the first pull, also sets up the repository for the first time



# Diagram of git usage



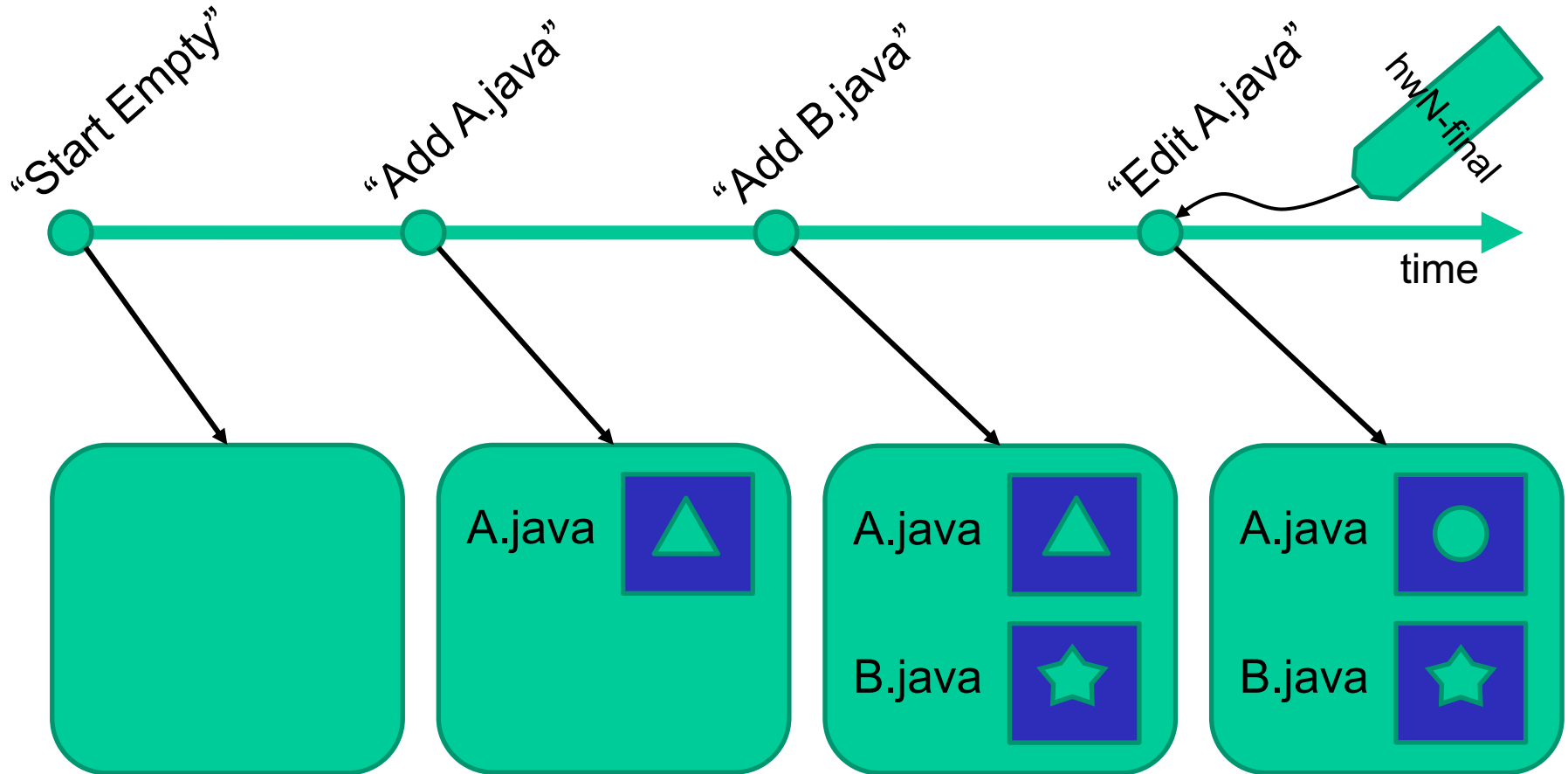
# Your GitLab repository

---

- We'll push starter code to your repo for each homework
  - After HW3, you'll get it by pulling
- Commit and push your code as you do the assignment
  - Recommended process: edit, test, pull, commit, push
- Submit homework  $n$  by creating a tag "**hwN-final**"
  - *Check that you've committed and pushed all your work **before you tag!***
  - Do **not** attach a message with the tag
  - Example: "**hw3-final**" for HW3
- Without the right tag, your homework might not be graded!

# Example commit history

---



# Best practices when using git

---

- Pull/Commit/Push your code *early* and *often*!!
  - You really, really don't want to deal with merge conflicts
  - Best to pull before you commit (in 331, industry is more complex)
  - Keep your repo up-to-date as much as possible
  - Copies on the server are a backup if you need to find older files
- Do not rename files and folders that we gave you
  - That will mess up our grading process
  - It would be a silly reason to lose points!
- Use this repo just for homework

# Best practices when using git

---

- Pull/Commit/Push your code *early and often!!*
  - You really, really don't want to deal with merge conflicts
  - Best to pull often (even if it's a bit complex)
  - Keep your local copy up to date
  - Copies of files are not the same as the files
- Do not rebase
  - That will mess up your history
  - It would be a silly reason to lose points!
- Use this repo just for homework

**Do not branch,  
rebase, or fork!**

# Gradle: what is it

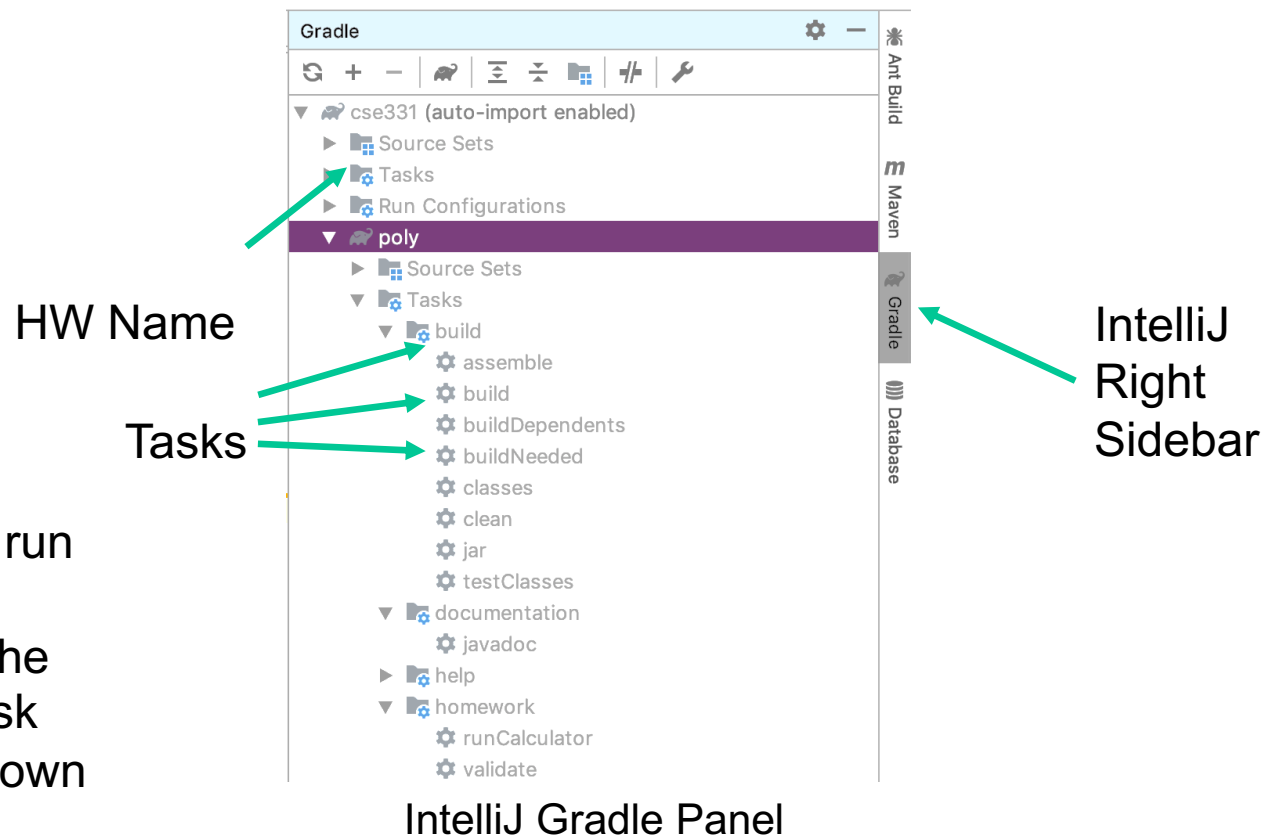
---



- Gradle is a tool for build automation
  - Simplifies compiling, running, and testing a software project
  - No need to install: included in the starter code!
- Configured by the file `build.gradle` (and others) in your repo
  - You shouldn't modify this (can interfere with grading)!
  - Ask the course staff for help if it got messed up accidentally.
- IntelliJ has built-in support to work with Gradle
- Gradle is how you run/validate your code on attu

# Gradle: how to use it

- You can use Gradle at the command line or in IntelliJ (recommended)
  - Every homework assignment has a “name” – HW3 is “hw-setup”



- Double-click tasks to run them.
- Make sure you're in the right assignment's task list, each one has its own tasks.



# Let's Try It!

---

Get your computers out and start up  
Terminal (macOS) or Git Bash (Windows)

# Getting Connected to GitLab

---

- Generate an RSA key pair:

```
ssh-keygen -o -t rsa -b 4096 -C "your@email.com"
```

- The (-C) comment can be any string, make it something you'll recognize.
  - Press enter when asked for a file name (use default)
  - No passphrase
  - You'll be told: "Your public key has been saved in (...)"
- Copy the generated public key (use the file name of the public key from above, if different)

```
cat ~/.ssh/id_rsa.pub | clip          (Windows)  
cat ~/.ssh/id_rsa.pub                (macOS/linux)
```
  - macOS/linux: Select and copy the output of running the `cat` command

# Getting Connected to GitLab (2)

---

- Paste that into your GitLab account, under “Preferences” > “SSH Key”
  - Sign in at: `gitlab.cs.washington.edu`
- In Terminal/Git Bash, type the following to check that you’re set up:  

```
ssh -T git@gitlab.cs.washington.edu
```
- Getting “The authenticity of host (...) can’t be established”?
  - Type **yes** – only a one-time thing, the GitLab server is just unfamiliar to your computer.
- Should get a welcome message back!

# Cloning Your Repo

---

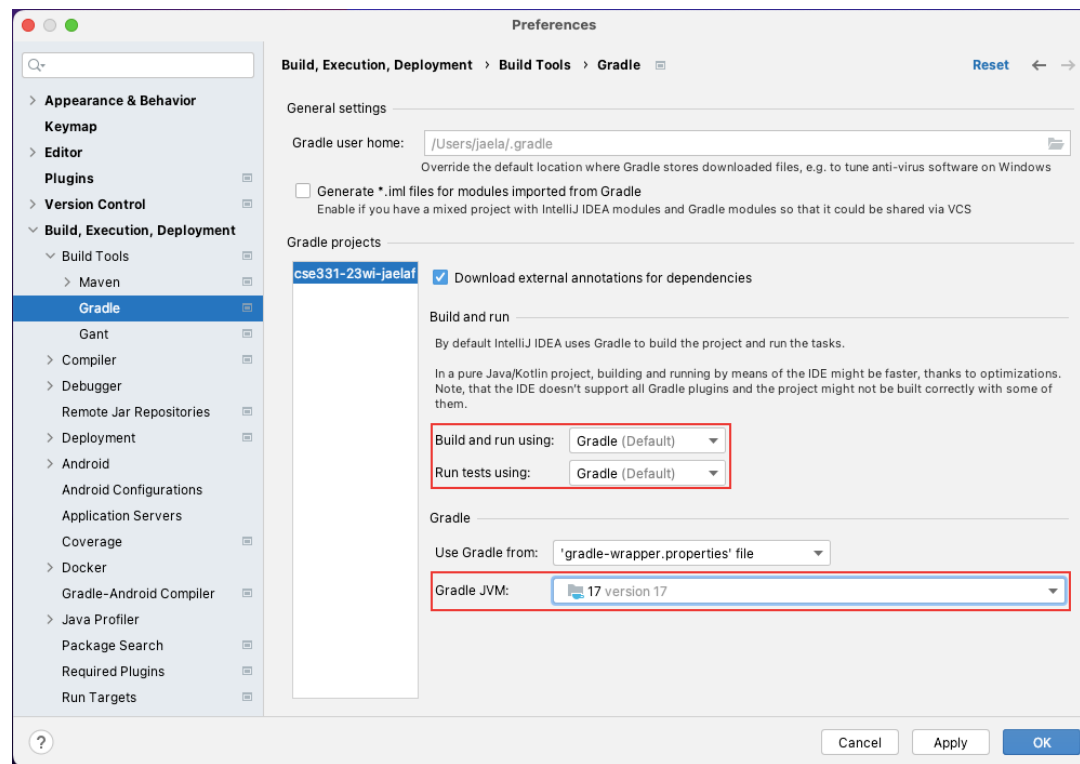
- In GitLab, open your project page and get the SSH clone URL

**gitlab.cs.washington.edu/cse331-23wi-students/cse331-23wi-*NETID***

- Blue “Clone” button in top right: copy the “Clone with SSH” URL
- Open IntelliJ
  - You don’t need any plugins or launcher scripts, skip those steps
- Choose “**Get from VCS**”
- Choose 'Git', paste the clone link from earlier in 'URL', and choose a place on your computer in 'Directory' where you want to keep your 331 work. This 'Directory' folder should not be located in a OneDrive, iCloud, GoogleDrive, etc.. folder
- Click **Clone**

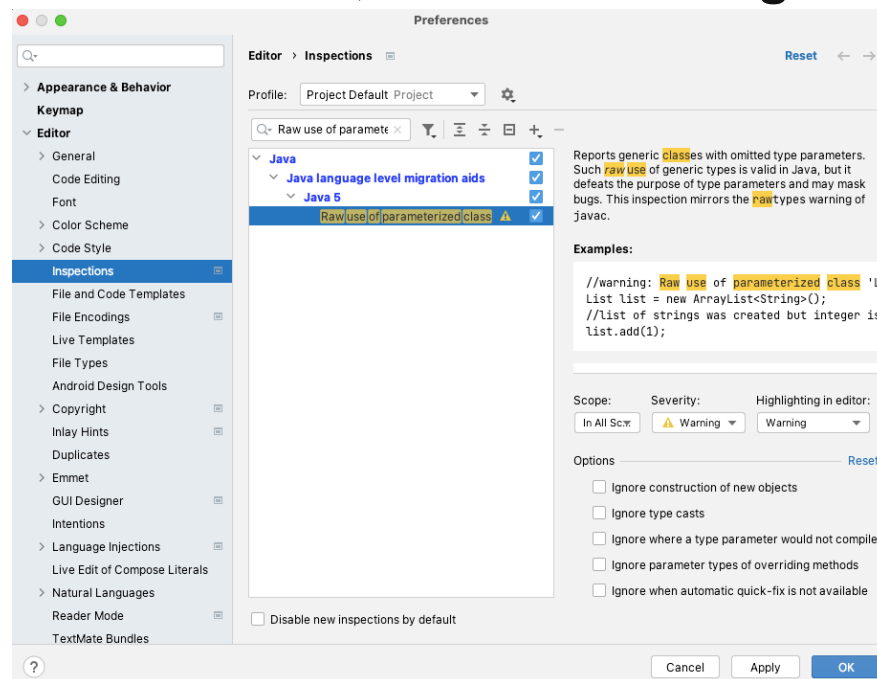
# Importing Into IntelliJ

- Need to check some Gradle settings
- IntelliJ IDEA > Preferences (macOS), File > Settings (Windows/linux)
- **Build, Execution Deployment > Build Tools > Gradle**



# Importing Into IntelliJ (2)

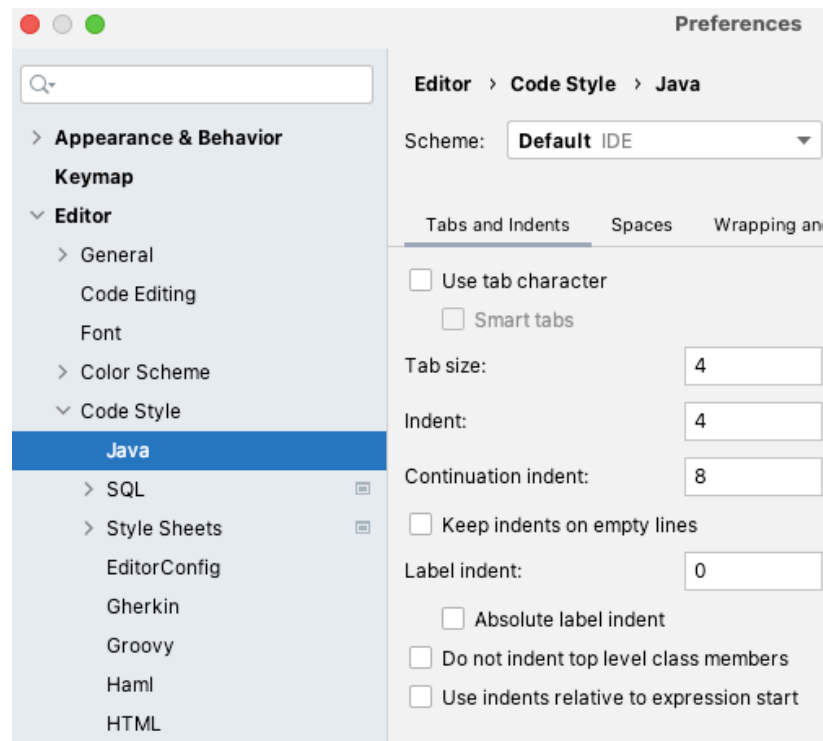
- Also, need to update some settings for generic errors
- IntelliJ IDEA > Preferences (macOS), File > Settings (Windows/linux)
- **Editor > Inspections**, search for “Raw use of parameterized class” and enable checkbox
- With that inspection selected, *disable* all the "Ignore..." checkboxes



# Importing Into IntelliJ (3)

---

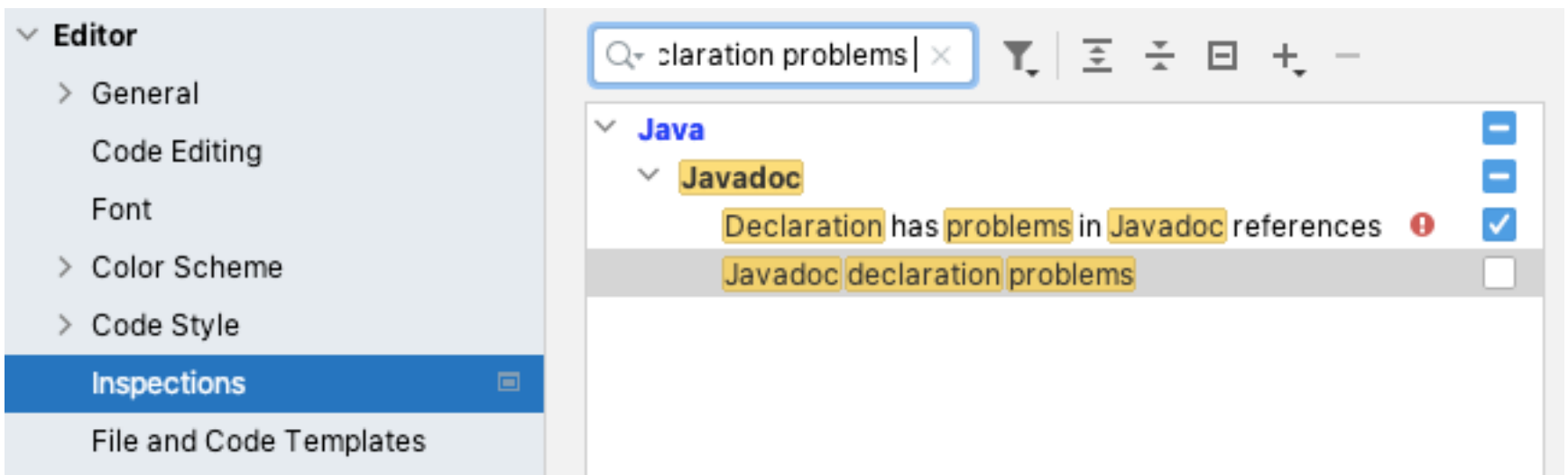
- Also, need to correctly convert tabs to spaces
- IntelliJ IDEA > Preferences (macOS), File > Settings (Windows/linux)
- **Editor > Code Style > Java**, make sure the “Use tab character” option is unselected.



# Importing Into IntelliJ (4)

---

- Also, need to update Javadoc options
- IntelliJ IDEA > Preferences (macOS), File > Settings (Windows/linux)
- **Editor > Inspections > Java > Javadoc**, uncheck the box next to “Javadoc declaration problems”





# Development Workflow Demo

---

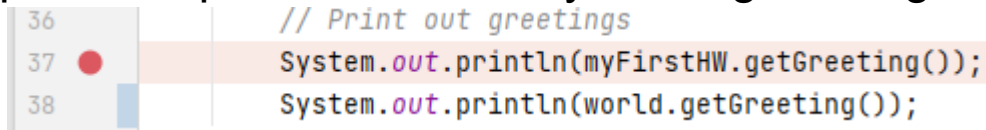
1. Open the first part of the hw3 starter code:
  - **hw-setup/src/main/java/setup/HolaWorld.java**
2. Fix the two bugs in this code: Lines 38 & 48
3. Run the code using Gradle:
  - Open the Gradle panel on the right edge of IntelliJ
  - Provided a runHolaWorld Gradle task under the “homework” group
  - **cse331 > hw-setup > Tasks > homework > runHolaWorld**
4. Double-click to run the task: see the output at the bottom!
  - Gradle automatically compiles your code and then runs it.

# Development Workflow Demo (2)

---

Let's try using the IntelliJ Debugger to examine our code.

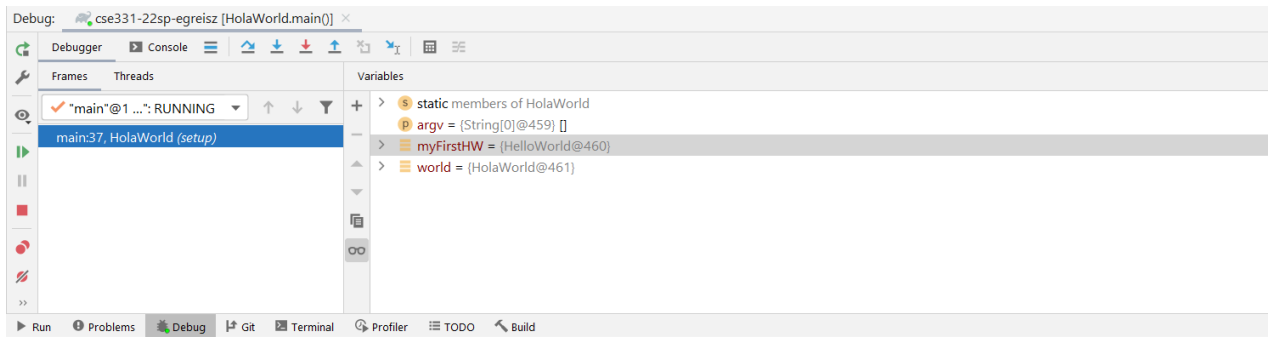
1. Set up a breakpoint on line 37 by clicking in margin.






```
36 // Print out greetings
37 System.out.println(myFirstHW.getGreeting());
38 System.out.println(world.getGreeting());
```

A screenshot of the IntelliJ code editor. Line 37 is highlighted with a red dot in the left margin, indicating a breakpoint. The code on line 37 is `System.out.println(myFirstHW.getGreeting());`. Line 38 is `System.out.println(world.getGreeting());`. Line 36 is a comment: `// Print out greetings`.

2. Run the code with “Debug”. Look at the program state:



3. **Step Over**  to move to line 38. This executes the current line.
4. **Step Into**  the `getGreeting()` method. This enters a method called on the current line to debug. We are now in `getGreeting()`.
5. **Resume Program** . This will run to the next breakpoint (or in our case, until the program finishes).

# Development Workflow Demo (3)

---

We've finished part 4 of the assignment (!) – let's commit this code to save it.

1. "pull" to make sure we have any updates that happened while we were editing:
  - **Git > Pull** (use the default options)
2. "commit" the changes to save them to our local copy of the repository:
  - **Git > Commit**
  - **Check the boxes for the file changes you want to include in the commit (usually all files)**
  - Uncheck everything under "Before Commit" (just extra IntelliJ warnings, you can keep them but it adds extra steps to the commit). You may need to click the settings wheel in the bottom right of the commit window to find the "Before Commit" section
  - Enter a short (< 25 words), **helpful** description of the changes in "Commit Message"
3. "push" the changes to tell GitLab about the new commit:
  - **Git > Push**

# Development Workflow Demo (4)

---

In general, only do this at the end of an assignment, but let's see how it works with a practice tag.

1. Create the tag with the correct name. For now, use **section-demo**. See assignment specs for the tags to use for each assignment.
  - **Git > New Tag**
  - Enter a tag name. (**Tags are case-sensitive.**)
  - **DO NOT include a message.** (This can make the tags difficult to move later, if you need to.)
  - Tags are automatically attached to the current commit on the remote repository (so **you need to create tags after creating and pushing the commit you want to tag in a separate transaction**).
2. “push” the changes to tell GitLab about the tag (so the staff can see it!)
  - **Git > Push**
  - Make sure “Push Tags” (bottom left) is **checked**. (Choose “All”)

# Development Workflow Demo (5)

---

## GitLab Runners:

- Triggered when you push the tag
  - Don't see a runner? Make sure you have the right tag name! (Tags are case-sensitive)
- Runs some sanity checks (build, javadoc, and your tests) to look for common errors.
- If your runner fails, you should **definitely** fix it, then move the tag and check the runner again.
- Open your GitLab project online, go to CI/CD → Pipelines (found in left hand options bar)
- For **section-demo**, you'll see a message and the pipeline should pass.
- For actual assignments, you'll see it run checks on your assignment, then it'll either pass or fail and print an error message on failure.

# Development Workflow Demo (6)

Verifying your tag is on the correct commit:

- GitLab Repository: **Left Sidebar > Repository > Graph**

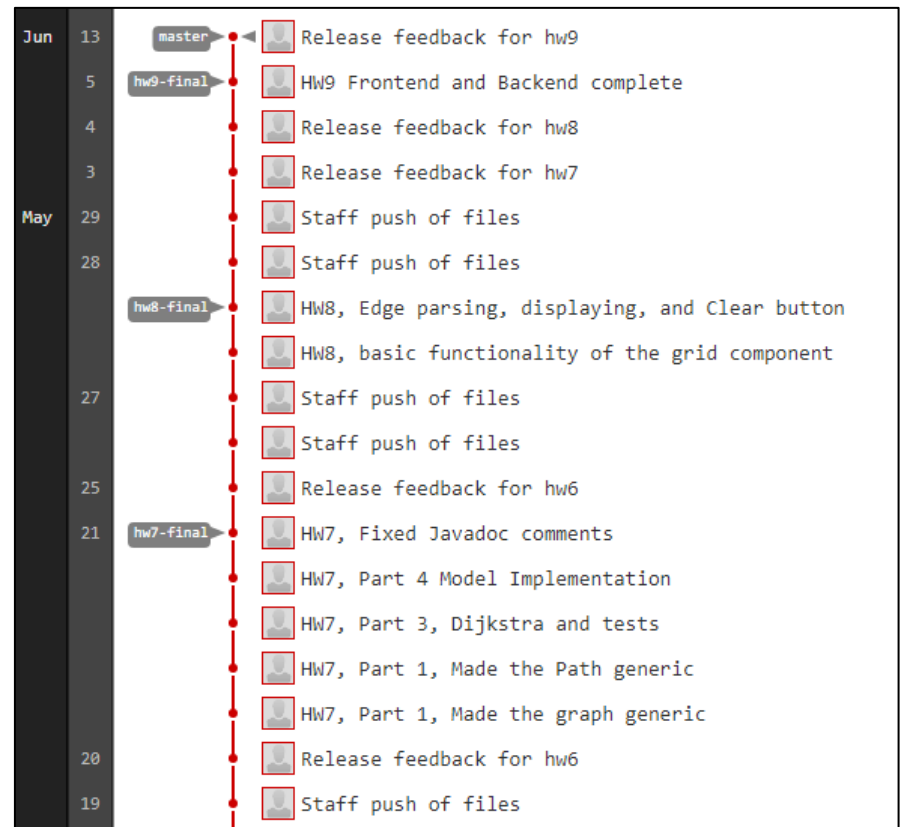
This page provides a good visual for which commit your tags are attached to!

Also can check out **Repository > Tags** (browse the files and check that the SHA matches the one found in **Repository > Commits**)

hw9-final  
dc75b40e · HW9 Frontend and Backend complete

hw8-final  
0c79b6ff · HW8, Edge parsing, displaying, and

hw7-final  
7f59acc5 · HW7, Fixed Javadoc comments



# Development Workflow Demo (7)

---

For an additional sanity check on your code, run it on attu:

- Run your code in the same place we'll be grading it!
- Sign into attu: `ssh NETID@attu.cs.washington.edu`
  - Clone your repo, checkout tag, and run the gradle task
  - See Assignment Submission handout for instructions.
- Since you're testing on a new clone, it'll only have the files that are in git.
  - Makes sure you didn't miss any files when making commits. (Common error in 331, can make assignments impossible to run/evaluate.)

# Gradescope + Late Days

---

- Once we are done grading, you will find your feedback on Gradescope.
- You **do not** need to upload the coding portion to Gradescope—we will do that automatically
  - **Ignore** the submission time on Gradescope
    - We will send you a confirmation email including the # of late days this assignment used and the # of late days you have remaining
  - But you will need to upload the written portion
- To use a late day, just submit your assignment late (tag a commit within **24** hours after it's due). No need to email us or let us know.
  - Realized you made a mistake? You can re-tag before the deadline! Instructions here: [Assignment Submission](#)
  - If you want to spend a late day to fix a mistake in your submission, feel free! (Even after you receive the confirmation email.) But you must turn it in before the late deadline.



# Important Handouts

---

<https://cs.uw.edu/331/resources.html>

- Project Software Setup handout
  - Important settings for IntelliJ (**you need to set these**)
  - Running your code on attu, in a virtual machine, or on remote desktop.
- Running/Compiling/Testing/Editing
  - How to use Gradle to run automated tests and see test results in IntelliJ.
  - [Optional] SpotBugs: A useful tool for finding bugs in your code
- Version Control handout
  - Git best practices, instructions, and more advanced usage
- Assignment Submission handout
  - Creating and moving tags, using late days, GitLab validation