Lab 3: DIY HJ-lib Programming, Futures, HJ-Viz

Instructors: Vivek Sarkar, Shams Imam

Course wiki: `http://comp322.rice.edu`

Staff Email: comp322-staff@rice.edu

Goals for this lab

- Learn how to use HJ-lib on your own (without maven), and how to optionally generate abstract metrics
- Experiment with functional programming and futures, including the `future` API
- Learn how to use HJ-Viz to visualize Computation Graphs (CGs) for small inputs

Downloads

Download the .zip file for this lab located in the Lab 3 entry towards the botton of the course web site, `http://comp322.rice.edu`. It contains the following:

- Three `.java` source files, `AsyncFinishTest.java`, `BinaryTreeTest.java`, and `BinaryTrees.java`.
- A directory named `Jars` containing three `.jar` files that you will use today.
- A directory named `hjviz` for the HJ-Viz part of the lab.

1 Setting up an HJ-lib project from scratch

1.1 Creating Project Structure and adding External Dependencies

For this exercise our first task is to structure our project. This was done automatically for you in Lab 1 and Lab 2 by using maven, but now you can learn how to do it yourself manually.

Create an IntelliJ project using the three java files in `lab_3.zip`. You should see a project structure similar to Figure 1

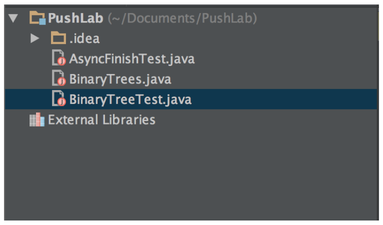
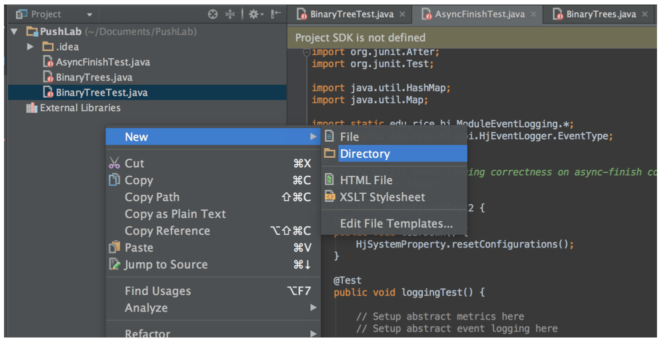


Figure 1: Initial Directory Layout of the project

- Create two folders one for source files and the other for test files. Add folders (as shown in Figure 2) and re-organize the files as shown in Figure 3. Note, we have added the source files in main (you may choose any name) and the test files in test (you may choose any name).



Project SDK is not defined

```
import org.junit.After;
import org.junit.Test;

import java.util.HashMap;
import java.util.Map;

import static edu.rice.hj.ModuleEventLogging.*;
                            i.HjEventLogger.EventType;

                            ing correctness on async-finish c

                            2 {

        HjSystemProperty.resetConfigurations();
    }

    @Test
    public void loggingTest() {

        // Setup abstract metrics here
        // Setup abstract event logging here
```

Figure 2: Creating folders

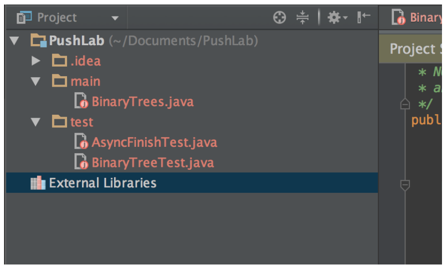


Figure 3: A screen grab of the new folder structure

- Now we need to add our dependencies for the Java project. Open module settings, as shown in Figure 4.

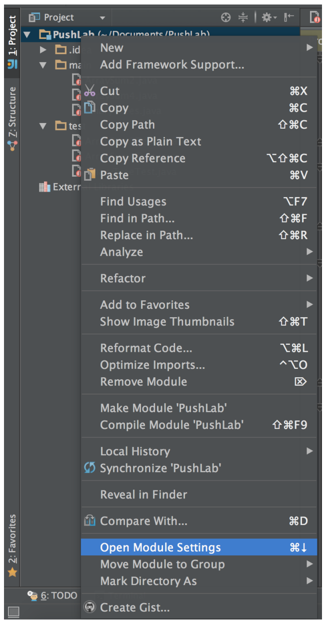
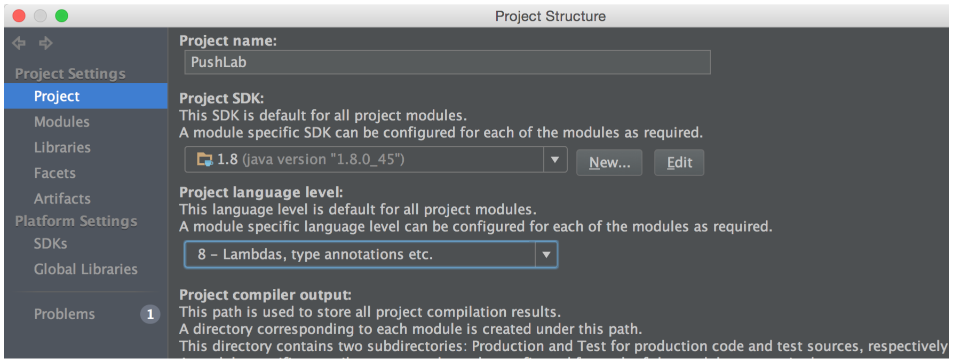


Figure 4: Opening module settings

- Under the 'Projects' tab, set Project SDK as 1.8 and Project Language Level as 8, as shown in Figure 5.



Project name:
PushLab

Project SDK:
This SDK is default for all project modules.
A module specific SDK can be configured for each of the modules as required.
1.8 (java version "1.8.0_45")

Project language level:
This language level is default for all project modules.
A module specific language level can be configured for each of the modules as required.
8 - Lambdas, type annotations etc.

Project compiler output:
This path is used to store all project compilation results.
A directory corresponding to each module is created under this path.
This directory contains two subdirectories: Production and Test for production code and test sources, respectively

Figure 5: Configuring the Project SDK and Language Level

- Under the 'Libraries' tab, click on add(+) > New Project Library from *Java*. Locate each .jar file under the Jars directory on your local machine and add each one through the dialog shown in Figure 6.

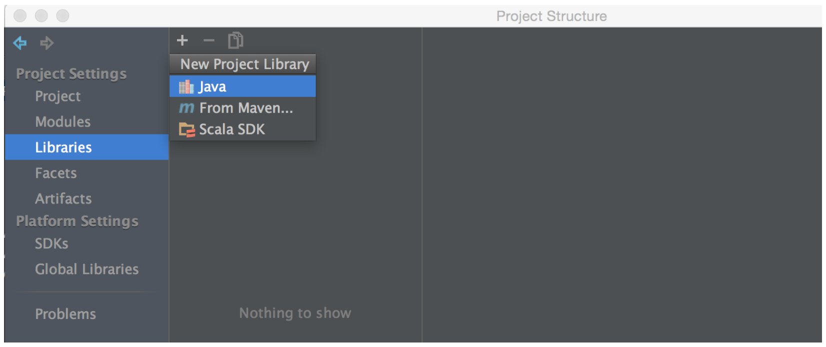
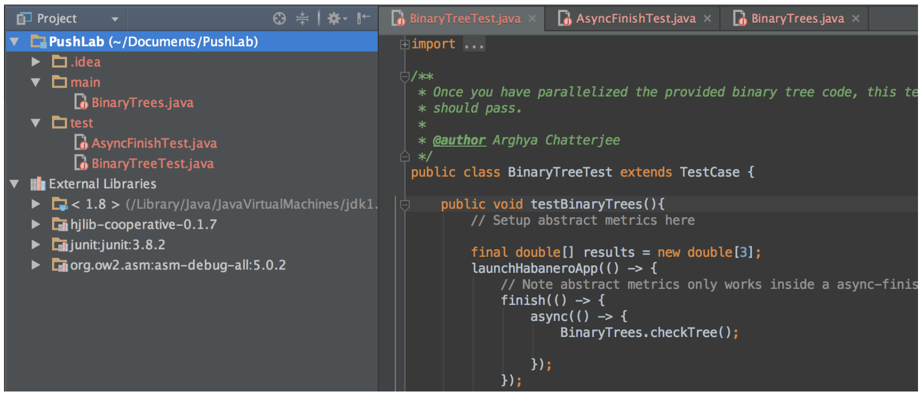


Figure 6: Adding the HJlib dependency

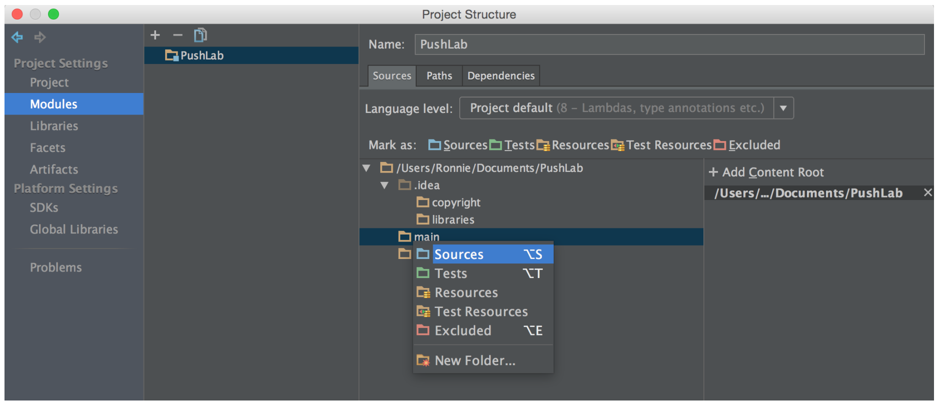
- After adding the External Dependencies your project structure should look like this :



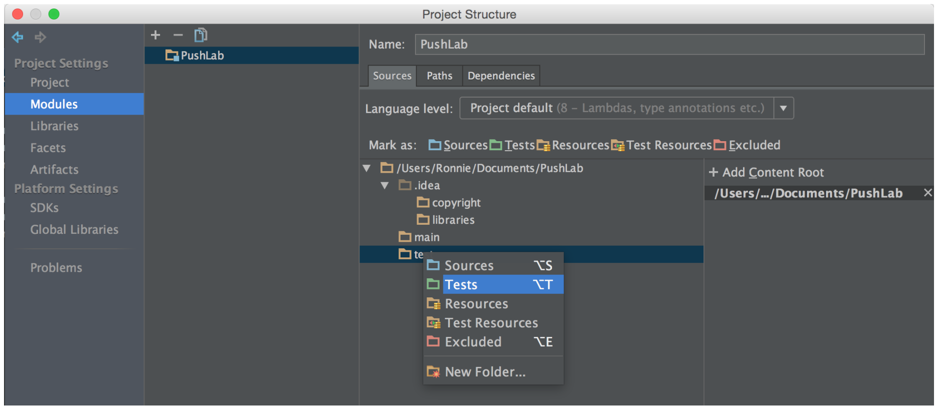
```
import ...

/**
 * Once you have parallelized the provided binary tree code, this te
 * should pass.
 *
 * @author Arghya Chatterjee
 */
public class BinaryTreeTest extends TestCase {

    public void testBinaryTrees(){
        // Setup abstract metrics here

        final double[] results = new double[3];
        launchHabaneroApp(() -> {
            // Note abstract metrics only works inside a async-finis
            finish(() -> {
                async(() -> {
                    BinaryTrees.checkTree();

                });
            });
```

1.2 Add sources for our project files and add VM options

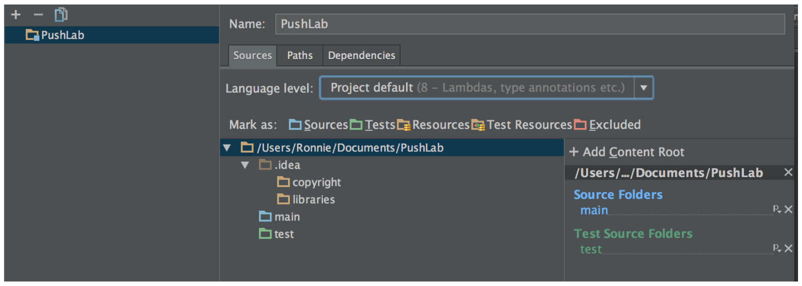
- Open module settings, and under the 'Modules' tab > Sources, mark source folder (*'main' in my case*) as Sources :



- Under the same 'Modules' tab > Sources, mark tests folder (*'test' in my case*) as Tests :



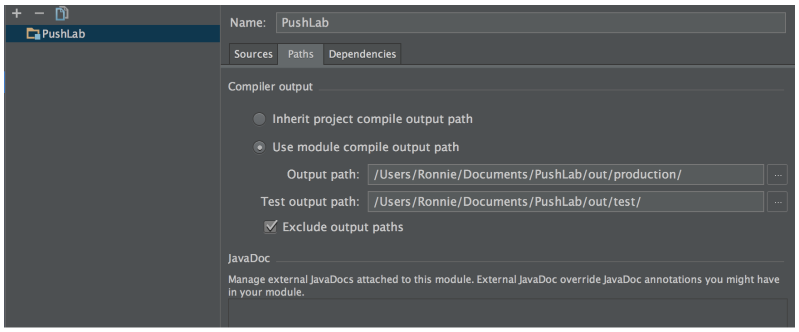
- After marking your sources and tests your content root should look like :


PushLab
Name: PushLab
Sources
Paths
Dependencies
Language level: Project default (8 – Lambdas, type annotations etc.)
Mark as: Sources Tests Resources Test Resources Excluded
/Users/Ronnie/Documents/PushLab
.idea
copyright
libraries
main
test
+ Add Content Root
/Users/.../Documents/PushLab
Source Folders
main
Test Source Folders
test

- We need to add the *Output folder* for our project, under the same 'Modules' tab > Paths, click on *Use module compile output path* –
  for Output Path : add < `your folder path` > `/out/production/` [1].
  for Test Output Path : add < `your folder path` > `/out/test/` [1]
  On my machine it looks like :

- Now we need to add the VM options, go to Run/Debug configurations and under 'Defaults' tab >

[1]Please do not add the angle brackets (only folder path, check figure for reference)



PushLab

Name: PushLab

Sources | Paths | Dependencies

Compiler output

○ Inherit project compile output path

● Use module compile output path

Output path: /Users/Ronnie/Documents/PushLab/out/production/

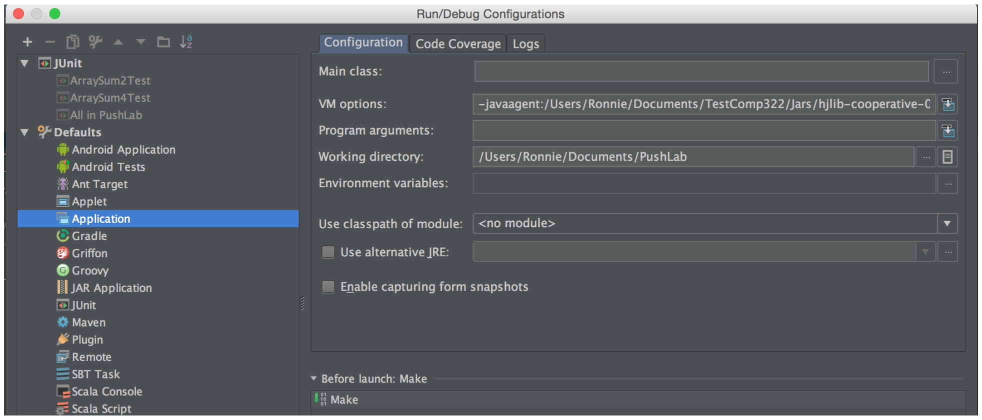
Test output path: /Users/Ronnie/Documents/PushLab/out/test/

☑ Exclude output paths

JavaDoc

Manage external JavaDocs attached to this module. External JavaDoc override JavaDoc annotations you might have in your module.

Application add the *-javaagent:* $<$ *Path/to/the/HJ-lib/jar/ on/your/local/machine* $>$ [2]. for the VM options :



Run/Debug Configurations

Configuration | Code Coverage | Logs

- JUnit
  - ArraySum2Test
  - ArraySum4Test
  - All in PushLab
- Defaults
  - Android Application
  - Android Tests
  - Ant Target
  - Applet
  - Application
  - Gradle
  - Griffon
  - Groovy
  - JAR Application
  - JUnit
  - Maven
  - Plugin
  - Remote
  - SBT Task
  - Scala Console
  - Scala Script

Main class:

VM options: -javaagent:/Users/Ronnie/Documents/TestComp322/Jars/hjlib-cooperative-0

Program arguments:

Working directory: /Users/Ronnie/Documents/PushLab

Environment variables:

Use classpath of module: <no module>

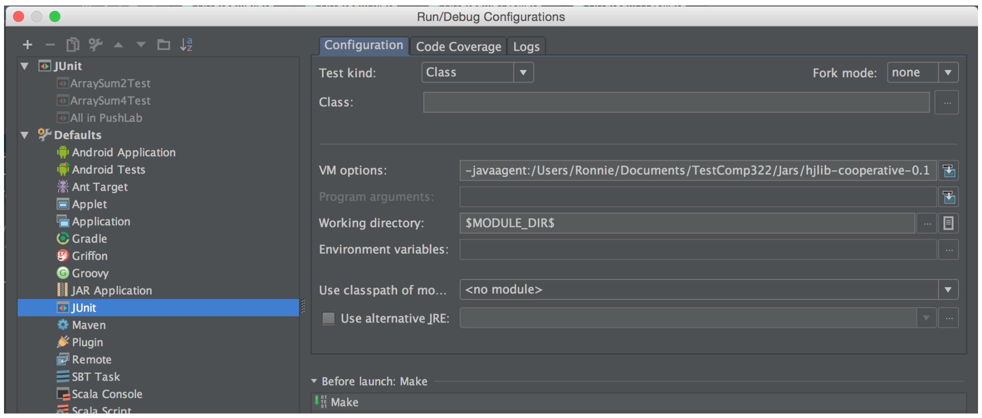
☐ Use alternative JRE:

☐ Enable capturing form snapshots

Before launch: Make

Make

- Similarly add the VM options for the jUnit under the 'Defaults' tab (same *-javaagent:../../*):



Run/Debug Configurations

Configuration | Code Coverage | Logs

- JUnit
  - ArraySum2Test
  - ArraySum4Test
  - All in PushLab
- Defaults
  - Android Application
  - Android Tests
  - Ant Target
  - Applet
  - Application
  - Gradle
  - Griffon
  - Groovy
  - JAR Application
  - JUnit
  - Maven
  - Plugin
  - Remote
  - SBT Task
  - Scala Console
  - Scala Script

Test kind: Class

Fork mode: none

Class:

VM options: -javaagent:/Users/Ronnie/Documents/TestComp322/Jars/hjlib-cooperative-0.1

Program arguments:

Working directory: $MODULE_DIR$

Environment variables:

Use classpath of mo... <no module>

☐ Use alternative JRE:

Before launch: Make

Make

[2]Please do not add the angle brackets (only folder path, check figure for reference)

2 Manually turning on HJ-lib Abstract Metrics

Abstract metrics were turned on in Lab 2 but were off by default in Lab 1. In this exercise, you will learn how to use the HJ-lib API to optionally turn on abstract metrics. First, you will need to add two import statements:

- `import edu.rice.hj.runtime.config.HjSystemProperty`
- `import static edu.rice.hj.Module0.abstractMetrics`

Now, to turn '*on*' abstract metrics you need to add " `HjSystemProperty.abstractMetrics.setProperty(true)`" inside each test file, just before the call to "`launchHabaneroApp()`".

The purpose of the call to `launchHabaneroApp()` is to launch the specified code expression as a lambda to be executed in parallel by the HJ-lib runtime, while all code before and after the call to `launchHabaneroApp()` is executed as standard Java code. For the current version of HJ-lib, it is good practice to include a top-level `finish` in the body of `launchHabaneroApp()` *In particular, the current implementation of abstract metrics may not print correct results if a top-level* `finish` *is omitted in* `launchHabaneroApp()`.

3 Getting Familiar with Futures

In this exercise, you are given sequential code for a functional program that constructs a binary tree, and then traverses it using calls to `checkTree()` and `itemCheck()`. Your task is to convert it to a correctly executing parallel HJ-lib program with futures. Once correctly implemented, your code should pass the provided `BinaryTreeTest`. Think back on your previous experience with functional programming, and your knowledge of futures and how they relate to your task.

1. Compile and run the original `BinaryTreeTest` program and note that it fails.
2. Now modify the program by replacing standard object references by future object references, *e.g.,* by replacing `TreeNode` by `HjFuture<TreeNode>`.
3. After you get your modified program to work, think about the new WORK and CPL values.

4 Visualize Parallelism using HJ-Viz

4.1 Prerequisites

- Python 2.7 (http://python.org/)
- GraphViz (http://graphviz.org/)

If you're on Linux, we suggest using your package manager. Homebrew and macports on Mac have the software as well. For GraphViz:

- Windows: `http://graphviz.org/pub/graphviz/stable/windows/graphviz-2.38.msi`
- Mac: `http://www.graphviz.org/pub/graphviz/stable/macos/mountainlion/graphviz-2.36.0.pkg`

4.2 Assignment

HJ-Viz generates interactive Computation Graphs (CGs) of parallel programs by analyzing event logs. Programmers can use the visualization of the CG by HJ-Viz to pinpoint potential sources of bugs and points of improvement for parallel performance by highlighting the critical path.

The goal of tonight's lab is to write a program whose output graph drawn using HJ-Viz, matches the provided figure below and to verify its correctness using the tool.

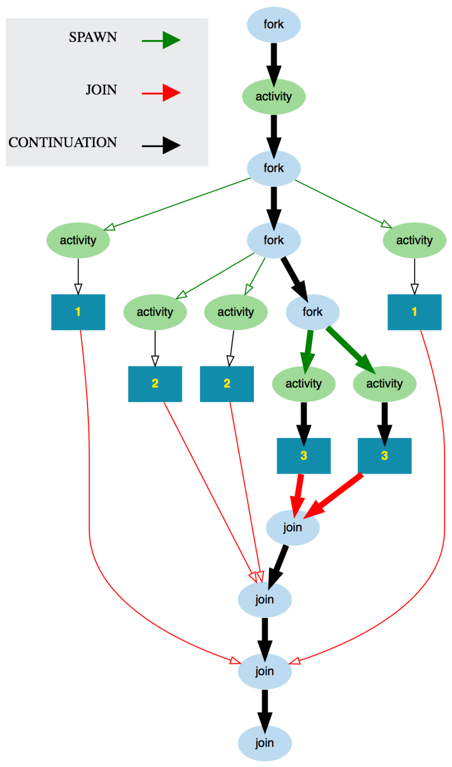


Figure 7: `fork` nodes indicate the start of a finish scope. `join` nodes correspond to the end of a finish scope. `activity` nodes refer to the start of an async task. Square nodes labeled by a number x shows that the task did x units of abstract work. Green edges direct from the enclosing scope to the spawned async task. Red edges direct from the ending task to the closing scope. Black edges indicate sequential dependence within the same task. Bold edges are on the critical path.

We have provided a skeleton file, AsyncFinishTest.java in your turnin repository as well.

- Follow section 2 to add the Abstract Metrics for this file as well.
- Add `HjSystemProperty.eventLogging.setProperty(true);` to the provided file, *before* the call to launchHabaneroApp(). This option activates event logging in the runtime
- Add `ModuleEventLogging.dumpEventLog(<file path>)`[3]; *within* the call to launchHabaneroApp, after your async-finish constructs, replacing < file path> [3] with the path to a file to write the output log (*e.g.,* "output.log").
- Use only finish and async constructs to write a program whose output through HJ-Viz matches the given figure.

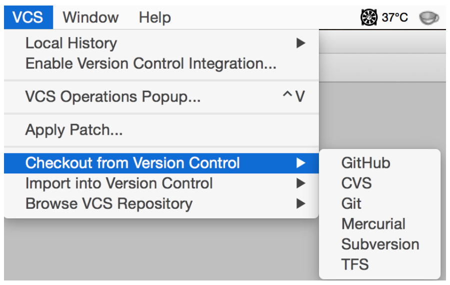
[3] Please do not add the angle brackets (only folder path)

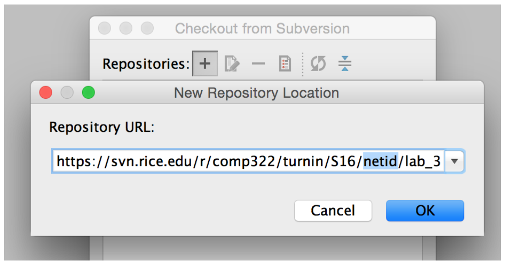
- On running the AsyncFinishTest.java file the code should generate a .log file.
- We have provided a HJ-Viz folder with all the required files to run HJ-Viz
  1. HJ-Viz is a Python program. Execute 'python main.py <path/to/the/log/file >' [3]., where logfile is the path to the .log file produced in the previous step.
  2. On Windows, you should be able to drag and drop the log file onto main.py in the file explorer.
  3. The graph output resides in output.html.
  4. See README for details.
- Show a teaching assistant that the output is correct.

Turning in your lab work

For each lab, you will need to turn in your work before leaving, as follows.

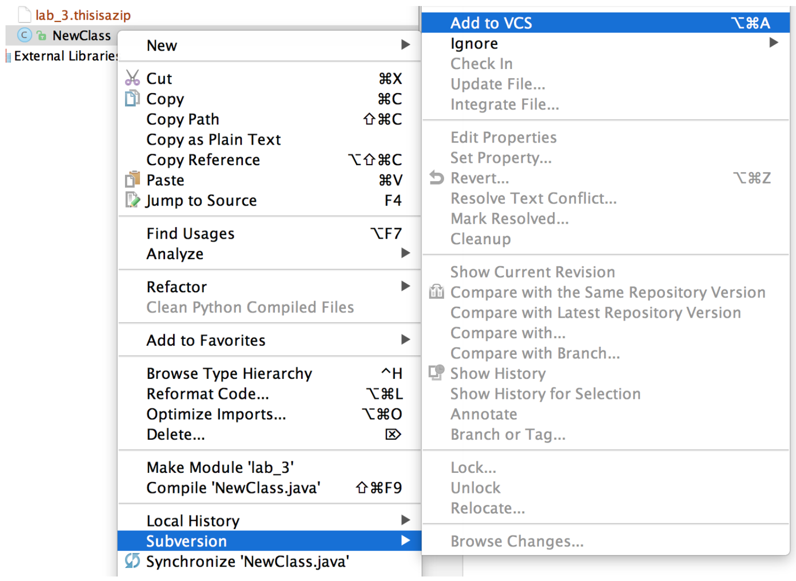
1. Show your work to an instructor or TA to get credit for this lab. Be prepared to explain the lab at a high level, as well as answer the following question:
   - What was your strategy in rewriting the provided sequential binary tree as a parallel one? How did functional programming aid you in this pursuit?
2. Turn in your code to SVN.
   - Your `lab_3` repository is currently empty. Import it as subversion from IntelliJ by checking out into where you extracted the project zip previously downloaded. `https://svn.rice.edu/r/comp322/turnin/S16/netid/lab_3`


VCS
Window
Help
37°C
Local History
Enable Version Control Integration...
VCS Operations Popup... ^V
Apply Patch...
Checkout from Version Control
Import into Version Control
Browse VCS Repository
GitHub
CVS
Git
Mercurial
Subversion
TFS



Repository URL: https://svn.rice.edu/r/comp322/turnin/S16/netid/lab_3

   - Use IntelliJ to add the Java files to Subversion, and commit your solutions.


lab_3.thisisazip
NewClass
External Libraries
New
Cut ⌘X
Copy ⌘C
Copy Path ⇧⌘C
Copy as Plain Text
Copy Reference ⌥⇧⌘C
Paste ⌘V
Jump to Source F4
Find Usages ⌥F7
Analyze
Refactor
Clean Python Compiled Files
Add to Favorites
Browse Type Hierarchy ^H
Reformat Code... ⌥⌘L
Optimize Imports... ⌥⌘O
Delete... ⌦
Make Module 'lab_3'
Compile 'NewClass.java' ⇧⌘F9
Local History
Subversion
Synchronize 'NewClass.java'
Add to VCS ⌥⌘A
Ignore
Check In
Update File...
Integrate File...
Edit Properties
Set Property...
Revert... ⌥⌘Z
Resolve Text Conflict...
Mark Resolved...
Cleanup
Show Current Revision
Compare with the Same Repository Version
Compare with Latest Repository Version
Compare with...
Compare with Branch...
Show History
Show History for Selection
Annotate
Branch or Tag...
Lock...
Unlock
Relocate...
Browse Changes...