CS5004, Spring 2022

Lab1: Setting Up and Getting Started

Divya Chaudhary
Modified originally from lab by
Brian Cross

1. Summary

In today's lab, we will:

- Configure our machines to use Java and IntelliJ
- Configure our development environment to use a proper code style
- Configure our development environment to use Gradle software management tool
- Walk through a simple example to create a class in Java
- Practice designing simple classes
- Practice writing and running unit tests for classes and methods that we write
- Practice writing documentation, and generating Javadoc

Note : Labs are intended to help you get started and give you some practice while the course staff is present and able to assist. Labs will still need to be turned in via your Khoury Github repo. For this lab, you won't turn it in until next week's lab, but it is to your advantage to complete this lab before the next lecture so that you will be set up.

2. Setup

2.1. Java Tools

Usually we download the JDK separately from the Oracle site. IntelliJ now supports the installation of the OpenJDK automatically. We will use that method as it integrates into the IDE nicely.

In the event that IntelliJ does not offer the automatic download/install, you can always download and install from here: https://docs.oracle.com/en/java/javase/17/install/
This is only recommended if IntelliJ's install doesn't work.

Note : You are welcome to use a different version of the JDK, but if you do, you do so at your own risk. We will not give you any extra time, or amend your grade due to any possible issues that you might experience with that version of the JDK. We will run all of your code against the version 17 JDK.

2.2. IntelliJ

The instructors will be using the latest version of IntelliJ (example: version 2021.3.1 for Windows, versions may be slightly different for other OSes) in class and labs.

- IntelliJ (https://www.jetbrains.com/idea).

    1. As a student you, get the full version of the IDE for free (https://www.jetbrains.com/student/). Apply for the student discount first.

    2. Download IntelliJ Ultimate Edition (https://www.jetbrains.com/idea/download)

    3. Install IntelliJ on your machine using the process that you typically use for your operating system to install new software

Note : If you decide to use a different IDE than IntelliJ, we will not give you extra time or amend your grade due to issue that you might face with your IDE's configuration. Your code must still be able to build and run within IntelliJ, using the class build.gradle

2.3. Testing Your Setup

1. Open IntelliJ

2. Either from IntelliJ's main menu select `New → Project` , or from the "Welcome" dialog, you can choose `New Project` .
   The New Project Dialog window should appear with two tabs
   - a list of project types on the left
   - a dropdown with the title **"Project SDK"** on the right
   - a list of additional libraries also on the right
3. Select Gradle from the list of project types on the left

4. Using the dropdown on the right select Download JDK…
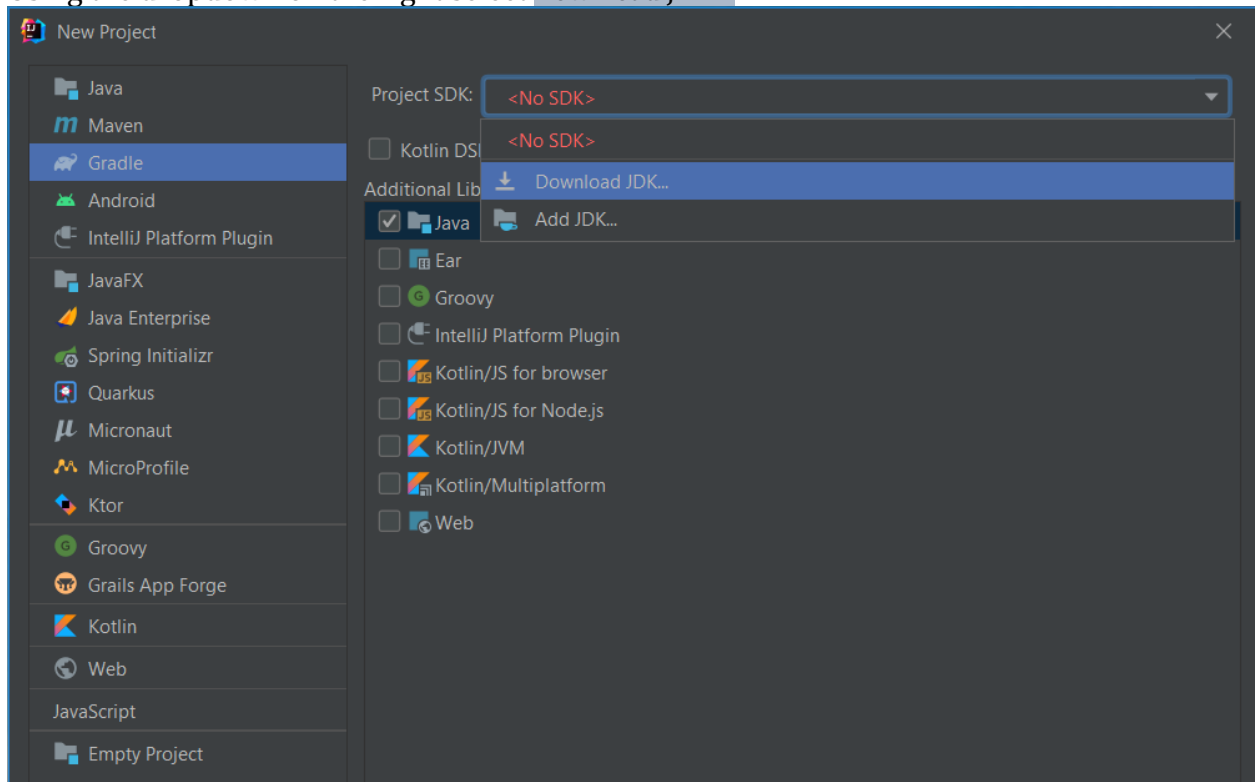
5. You should see a dialog allowing you to select the version of Java to download:
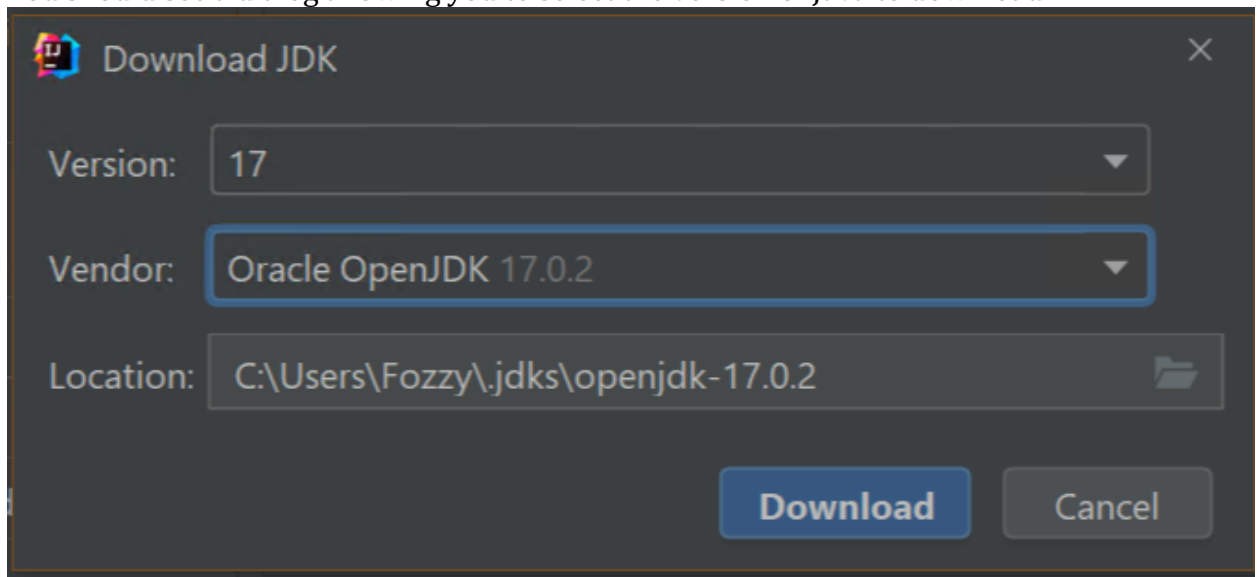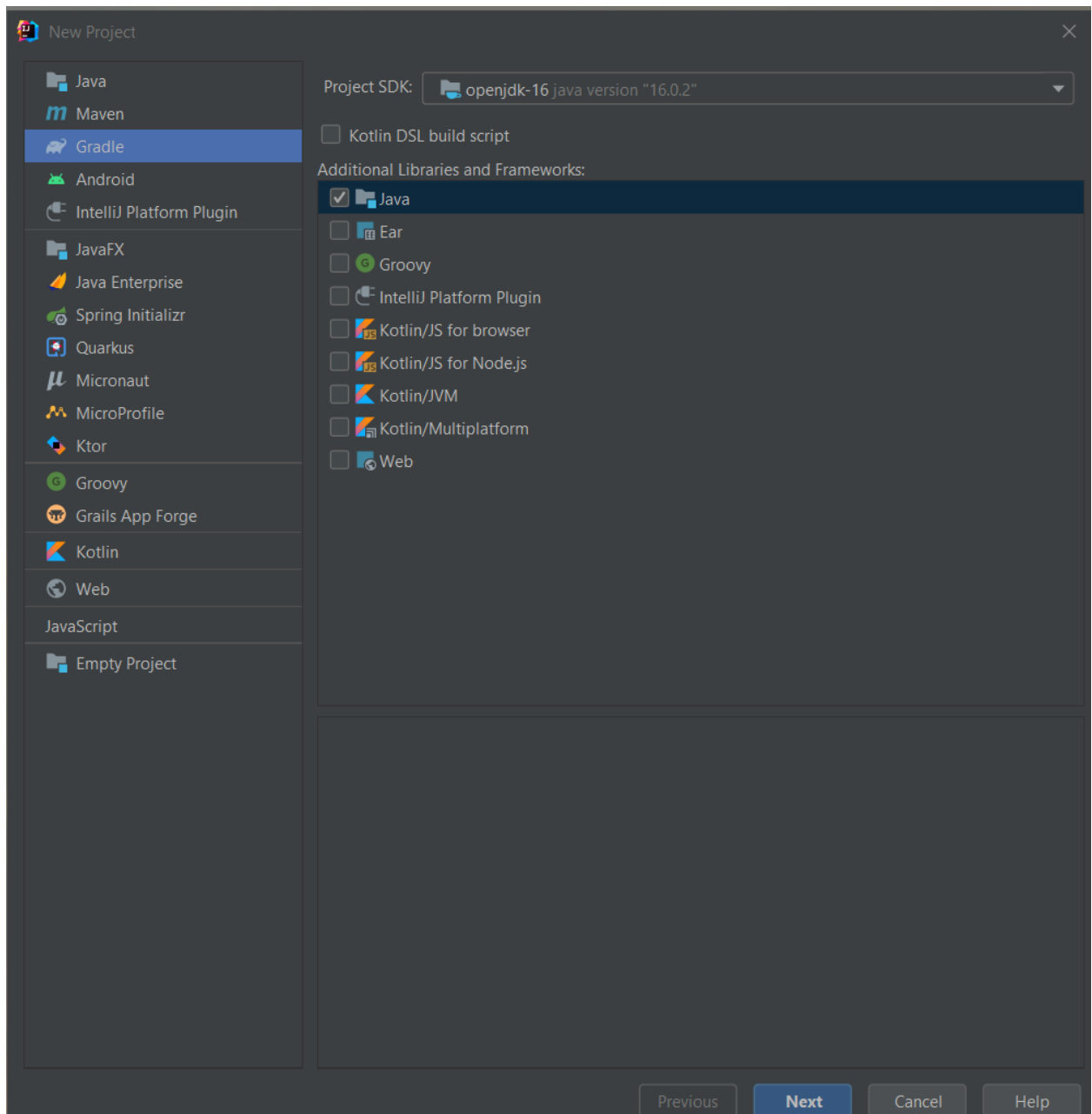

Choose the following:
Version: 17
Vendor: Oracle OpenJDK 17.0.2 (or any 17.x)
Location: use the default location
Click Download

6. After the JDK downloads and installs, you'll be back to the New Project dialog.

o   Make sure that "Java" is checked under Additional Libraries and Frameworks

7. Click Next . The contents of the window will change
8. You should see two fields

  o   **"Name"**, provide a name for your project. This name can be anything you like, but today, you probably want it to be Lab1.

  o   **"Location"**, this is the location on your computer that IntelliJ is going to use to create and store all your code. There should be a button to the far right of this field whose title is three dots, i.e., ... Click this button and navigate to the folder created due to the git clone operation in step 1. For the first lab it should be lab1.

9. Click Finish

10. IntelliJ will now open in a new window that will contain two tabs

    o   A left tab that is your Project Explorer. This is similar to your file explorer

    o   A right tab, the Editor, that will be used to show the contents of files that you select in the Project Explorer.

11. In the Project Explorer you should see a folder with the name you gave to your project. Double click on the folder's name to expand it.

12. You should see a sub-folder named src\main\java (I've noticed IntelliJ may take a few moments to generate all of the folders).

13. Right click on the folder named java to open the context menu. From the context menu select New → Java Class to create a new Java Class.

14. A small window will pop-up asking you to provide a name for your class. Input the name **Author** and click OK or press enter.

15. IntelliJ may detect that you are inside a git repository and will ask you if you would like to add the file to your repo. Please click No or Cancel

16. The right tab of your IntelliJ window should now contain a minimal Java class with

    o   an empty Java class definition

17. Notice that in the Project Explorer tab there is now a new file named Author under the folder named src

18. To test that your setup is working as expected, replace all the contents of the file Author.java with the following code

Author.java

```
/**
 * Represents an Author with their details--name, email and physical address
 *
 * @author therapon
 *
 */

public class Author {

  private String name;
  private String email;
  private String address;

  /**
   * Creates a new author given the author's name, email and address as strings.
   *
   * @param name the author's name
   * @param email the author's email address
   * @param address the authors physical address
   */
  public Author(String name, String email, String address) {
```

```java
    this.name = name;
    this.email = email;
    this.address = address;
  }

  /**
   * @return the name
   */
  public String getName() {
    return this.name;
  }

  /**
   * @return the email
   */
  public String getEmail() {
    return this.email;
  }

  /**
   * @return the address
   */
  public String getAddress() {
    return this.address;
  }
}
```

19. IntelliJ tries to assist you while you code by popping up an image of a small light bulb inside your editor (the right tab). Move your cursor to be inside the name of class, i.e., the word Author in the class header public class Author. Wait for a couple of seconds and the yellow light bulb image should appear at the start of that line. You can force the IntelliJ assistant to open using the following keystrokes

   o   Windows/Linux : Alt + Enter
   o   Mac : Option + Enter

20. Click on the yellow light bulb icon and a menu should appear. You'll take advantage of this menu often for automating many common tasks, like creating tests.
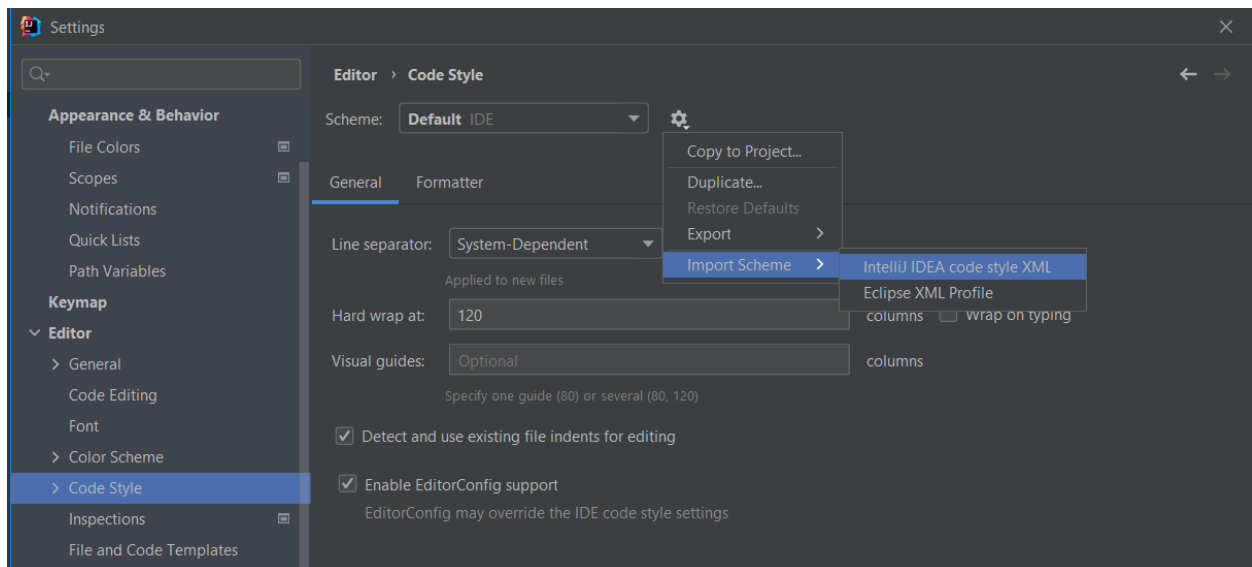
3. Code Style

We will be using the Google Codestyle guide (https://google.github.io/styleguide/javaguide.html). From now on, all code that you write for this class **must** follow this Google Java Style guide.

3.1. Configuring your IDE

Thankfully, there are saved IDE configurations that enforce **most,** but not **all** coding rules specified in the ~~Twitter~~ Google Java Style guide.

1. Download the saved configuration file for IntelliJ intellij-java-google-style.xml (https://github.com/google/styleguide/blob/gh-pages/intellij-java-google-style.xml) (You can right-click the "raw" link of the file on github and choose to `Save link as`)

2. Open `Preferences` .
    a. On Windows window to `File → Settings` on the main menu.
    b. On Mac go to `IntelliJ → Preferences` on the main menu.
3. Navigate to `Editor → Code Style` . Select `Code Style` but **do not expand the item**. In the right-hand side pane at the top you will see a button with the title `Manage` . Click it to open the `Code Style Schemes` option window.
    a. Instead of a `Manage` button, there may be a settings gear.

4. In the `Code Style Schemes` option window click the `Import Scheme`
5. Select `IntelliJIDEA code style XML` and a file explorer window will pop up for you to select the file you wish to import.

6. Select the file you just downloaded named `intellij-java-google-style.xml`. This is the file from step 1.

IntelliJ will **try** and format your code while you type. However, when editing a file at different locations in the file IntelliJ might not indent properly.

To force IntelliJ to re-indent all your code select `Code → Reformat Code` from the main IntelliJ menu. Read the IntelliJ documentation on code formatting for more options (https://www.jetbrains.com/help/idea/2016.3/reformatting-source-code.html).

3.2. IntelliJ Tips - Indentation

You can configure IntelliJ to use 2 spaces instead of 4 for each indentation level.

Go to `Preferences` . Navigate to `Editor → Code Style → Java` . Make sure the following items are set to these numbers

- **Tab size** : 2
- **Indent** : 2
- **Continuation Indent** : 4

4. Unit Testing with Java

In section 2.3, Testing Your Setup, we created class `Author`.
In this section, we want to test this class by writing unit tests for all of the methods in that class. Here are some steps to help you get started with that.

1. IntelliJ tries to assist you while you code by popping up an image of a small light bulb inside your editor (the right tab). Move your cursor to be inside the name of class, i.e., the word Author in the class header public class Author . Wait for a couple of seconds and the yellow light bulb image should appear at the start of that line. You can force the IntelliJ assistant to open using the following keystrokes
    a. Windows/Linux : `Alt` + `Enter`
    b. Mac : `Option` + `Enter`
2. Click on the yellow light bulb icon and a menu should appear.
3. From the menu select `Create Test` . This action will cause a new pop-up window to appear.
4. In the new pop-up window, we will configure and create a test for our Author class. Starting from the top of the window going down
5. For **"Testing library"** select JUnit 5
6. Leave **"Superclass"** empty.
7. Leave **"Destination Package"** empty
8. Select the check mark with the title **"setUp/@Before"** only.
9. At the bottom of this pop-up window you should see the list of methods that are available in class Author for testing. Select **all** of them.
10. Click `OK`
    This will create a new Java class (and a new file) called AuthorTest. If IntelliJ asks you to add this new file to your repo click `No` .
11. The Package Explorer should now have a new file with the name AuthorTest under the folder src\test\java
12. The editor should now display:

AuthorTest.java

```java
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class AuthorTest {

  @BeforeEach
  void setUp() {
  }

  @Test
  void getName() {
  }

  @Test
  void getEmail() {
  }

  @Test
  void getAddress() {
  }
}
```

3.1. Adding a test for `getName()`

Let's add a test for the method `getName()`.

1. We first create an example of an author. Since we are probably going to use this example in more than one test, we are going to create

   1. a field in the class `AuthorTest` to store the example

   2. create an instance of author and set it to our field inside our `setUp()` method

2. We then use our example inside a test method in order to call methods defined in the `Author` class on our instance of `Author` and verify using JUnit's `Assert.assertEquals` that we get the expected values back. Here is the resulting code:

AuthorTest.java
```java
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class AuthorTest {
  private Author jane;
  @BeforeEach
  void setUp() {
    this.jane = new Author("Jane Doe", "j@a.com",
```

```
        "222 Main St, Seattle, Wa, 98980");
  }

  @Test
  void getName() {
    Assertions.assertEquals("Jane Doe", this.jane.getName());
  }

  @Test
  void getEmail() {
    Assertions.fail("Not yet implemented");
  }

  @Test
  void getAddress() {
    Assertions.fail("Not yet implemented");
  }
}
```

Building your code using Gradle

Now that you've added your code and your tests, the gradle script will allow you to build and test your code (along with a host of other tools) all in one step.

1   In the Project Explorer, you should see a file named "build.gradle". You'll want to replace the contents of that file with the build.gradle for the class

build.gradle

```
plugins {
// Build Java: https://docs.gradle.org/current/userguide/building_java_projects.html
   id 'java'

// https://docs.gradle.org/current/userguide/pmd_plugin.html#header
   id 'pmd'

// https://docs.gradle.org/current/userguide/jacoco_plugin.html
   id 'jacoco'
}

group 'cs5004S2022EvenSEA'
version '1.0'

repositories {
   mavenCentral()
}

dependencies {
   testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.2'
   testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.2'
```

```
}

pmd {
    ignoreFailures=true
    pmdTest.enabled=false
    ruleSets = [
        "category/java/bestpractices.xml",
        "category/java/errorprone.xml",
        "category/java/codestyle.xml"
        //"java-basic",
        //"java-braces",
        //"java-strings",
    ]
}

javadoc {
    doLast {
        String fixedBuildDir = buildDir.toString().replace('\\', '/')
        println "file:///$fixedBuildDir/docs/javadoc/index.html"
    }
}

jacoco {
    toolVersion = "0.8.7"
}

jacocoTestReport {
    reports {
        html.destination file("${buildDir}/jacocoHtml")
    }
    doLast {
        String fixedBuildDir = buildDir.toString().replace('\\', '/')
        println "file:///$fixedBuildDir/jacocoHtml/index.html"
    }
}

jacocoTestCoverageVerification {
    violationRules {
        rule {
            limit {
                // minimum percentage of code coverage
                minimum = 0.7
            }
        }
        rule {
            enabled = false
            element = 'CLASS'
            includes = ['org.gradle.*']
            limit {
                counter = 'LINE'
```

```
            value = 'TOTALCOUNT'
            maximum = 0.3
         }
      }
   }
}

// Fail the build if code coverage isn't high enough
check.dependsOn jacocoTestCoverageVerification

// Run code coverage after tests run
jacocoTestReport.mustRunAfter test


task doAll {
   // Runs a java build. Imported from the java plugin
   dependsOn build
   // Runs all the tests. Imported from the java plugin.
   dependsOn test
   // Runs pmd. Imported by pmd plugin
   dependsOn check
   // Runs java doc. Imported from the java plugin
   dependsOn javadoc
   println 'all done!'
}

test {
   useJUnitPlatform()
   finalizedBy jacocoTestReport
}
```
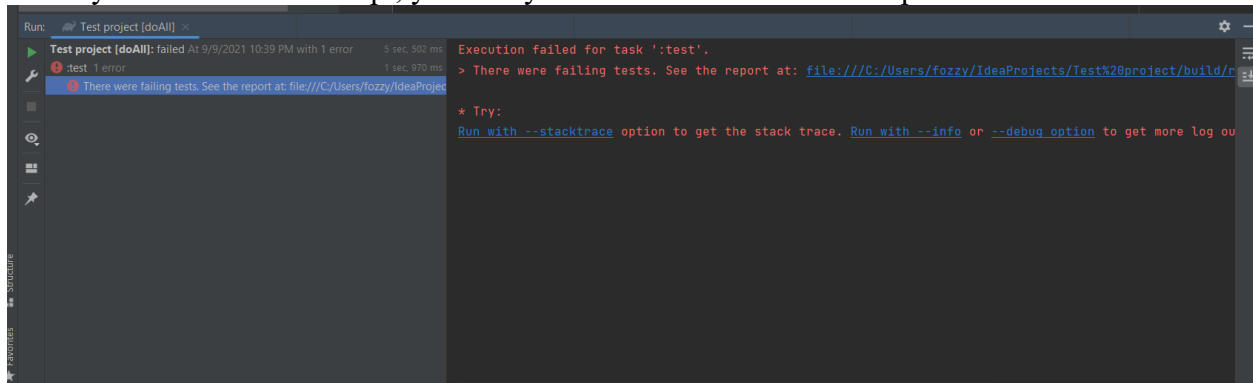
2   After you do that, you should see green triangles next to the gradle script functions. Take
    note of the script named "doAll"

```
83
84  ▶   ⊟task doAll {
85            // Runs a java build.  Imported from the java plugin
86            dependsOn build
87
88            // Runs all the tests.  Imported from the java plugin.
89            dependsOn test
90
91            // Runs pmd. Imported by pmd plugin
92            dependsOn check
93
94            // Runs java doc. Imported from the java plugin
95            dependsOn javadoc
96
97            println 'all done!'
98     ⊟}
```

3. Clicking the triangle will bring up a menu that will give you the option to run the script. The doAll script will build, run tests, check various code policies and then run the Javadoc tool to create documentation.
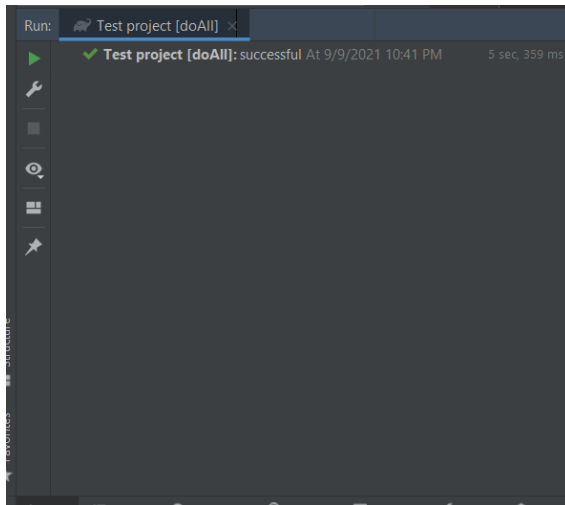
Practice Exercise
Testing

When you ran the doAll script, you likely saw some failed tests in output like this:

Fix the tests in `AuthorTest` so that they pass and rerun the doAll script. The last gradle script that you ran can be reran by clicking the green triangle on the left side (see image above).

Once you have fixed the tests you should see the following:

By default, this view may only show tasks that contained errors. It is useful to see all of the tasks that ran successfully so that you can click them and see the results. You can right click within the window to bring up a menu and choose `Show Successful Steps`
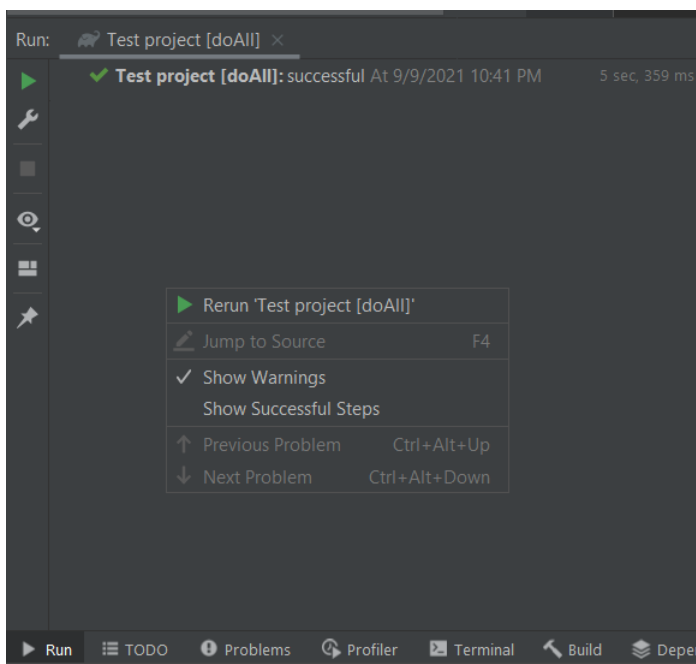
Once you select that you should see all of the tasks that ran:
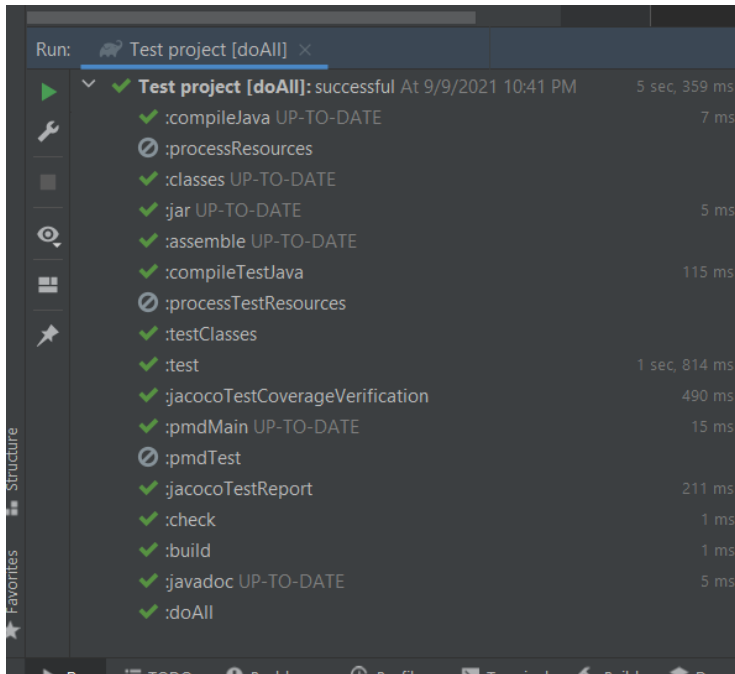
Clicking the root node of the task tree will show you all of the output from those tasks in the right text pane. Within the results, you'll find links to the following reports:

a. Jacoco – This is a tool that shows you the amount of code coverage your tests had. We will generally target having a code coverage of at a minimum of 70% (both in instructions and branches)

b. Pmd ("check" task)– A source code analyzer that helps to find common flaws in your code. It can help point out things like unused variables, empty catch blocks, unnecessary object creation and so on. You should strive to write clean code whenever you can.

c. Javadoc – Documentation generation from your annotations within your code.

d. Your tests – Your tests will run whenever you build and the report will be accessible from the logs.

UML

Draw the UML for Author.
(Examples of UML http://pages.cs.wisc.edu/~hasti/cs302/examples/UMLdiagram.html)
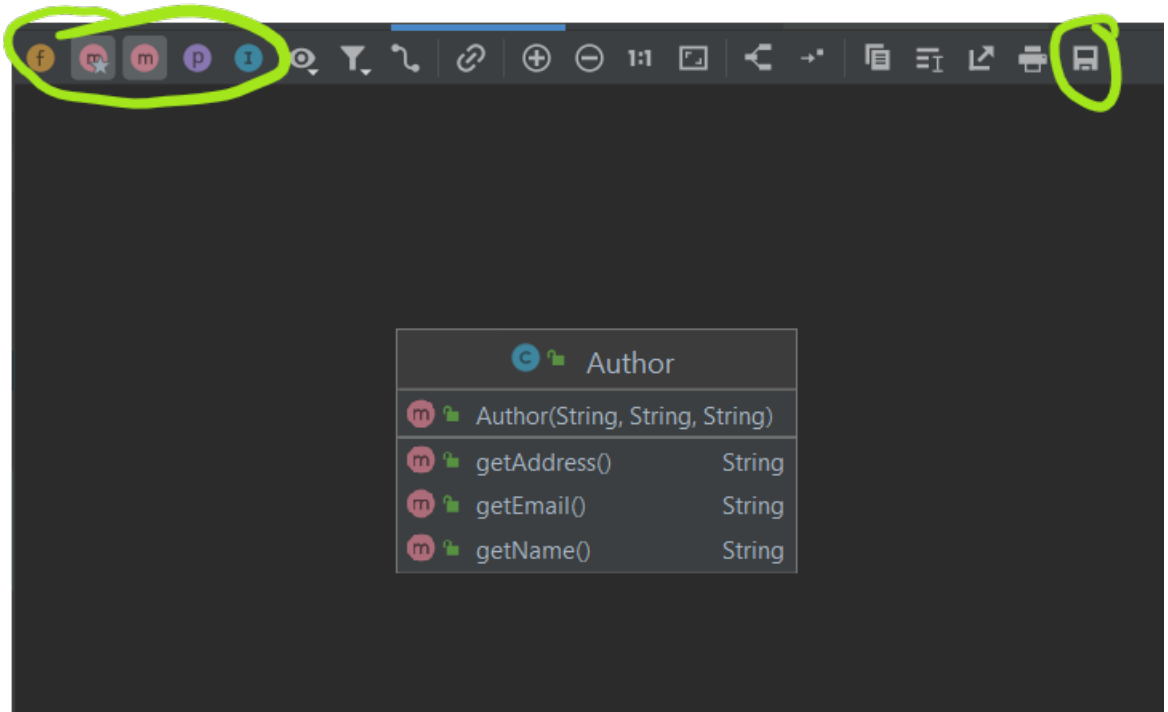
There are many ways you can create UML diagrams:

a. Drawing it by hand on paper/tablet

b. Using a diagram tool like Plant UML

c. Using the IntelliJ built-in Diagram tool. In IntelliJ, right-click the class that you want a diagram on, and select Diagrams/Show Diagram. See screenshot below:
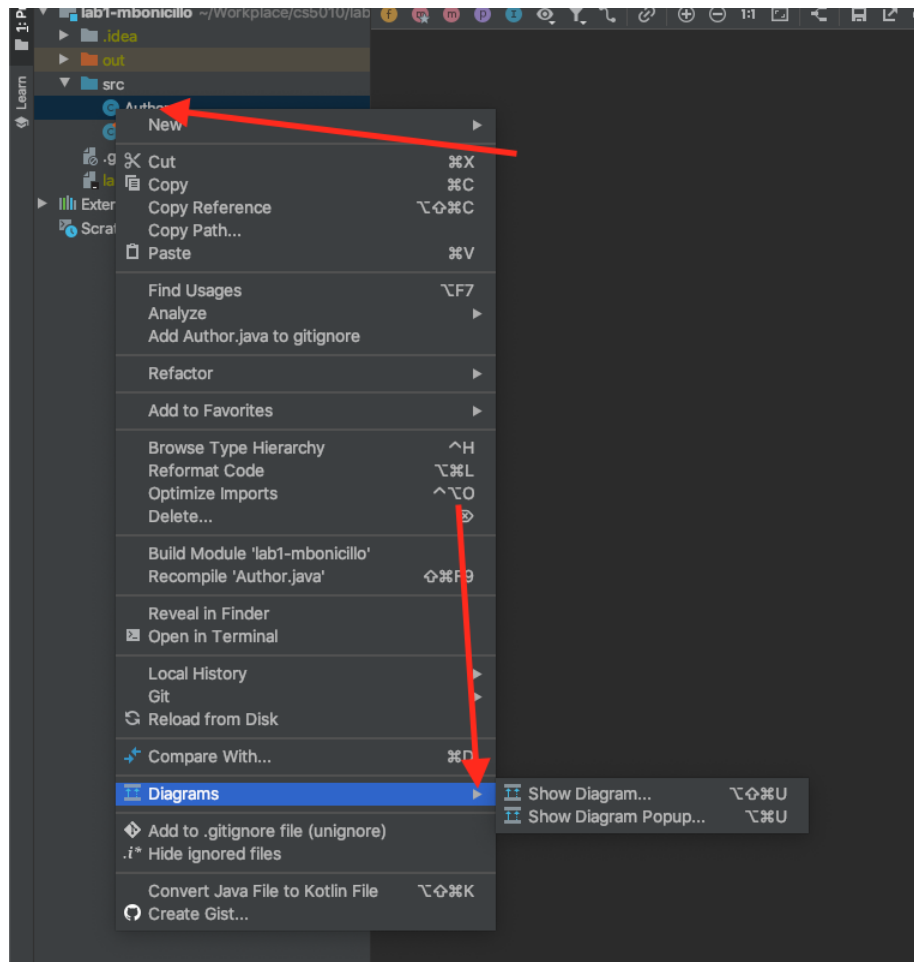
You'll have a UML automatically generated for you.

You can toggle the viewing of fields, methods, etc within your classes by selecting options on the top left (circled) and then you can save your diagram via the save icon (circled) on the right.

Once you have more than one class, you'll want to show the dependencies each type has (button circled in red)

4. Javadoc

For all the code that you have written in the lab up to this point, you should have also written documentation (https://www.oracle.com/technetwork/articles/java/index-137868.html).

If you have not, go back and add documentation now!

Let's manually generate the HTML version of your code's documentation.

1. Select your project in the Package Explorer tab.

2. On IntelliJ's main menu select `Tools → Generate Javadoc`. A new pop-up window will appear

3. In the new pop-up window

   a. Select **"Whole Project"**

b.   Unselect **"Include test sources"**

c.   Find the line that starts with the text **"Output Directory"**. At the end of that line click the ⬚ or folder icon and specify the output folder that you would like IntelliJ to place all the HTML files that will be generated.

d.   Click ⬚OK⬚. IntelliJ will run Javadoc and show its progress along with any warnings or errors in the bottom tab of the main IntelliJ window. If the Javadoc generation had no errors then IntelliJ will open your browser and point it to the newly generated documentation.

Our build.gradle file, however will build a Javadoc of your code automatically and output the path where you can view it in your build logs. You'll want to ensure there are no errors in the Javadoc task (or any other task) when running "doAll" in the classes build.gradle file.

5. What to turn in?

By the end this lab, you should be able to:

- Create a java project
- Use the class build.gradle file for your project
- Run tests
- See your code coverage
- Have documentation automatically generated for you
- Have your code analyzed for common flaws

Please hold on to your project. You'll use it to turn in during Lab 2.

6. Resources

- [Overview of the user interface | IntelliJ IDEA (jetbrains.com)](#)

- [Editor basics | IntelliJ IDEA (jetbrains.com)](#)

- [Prepare for testing | IntelliJ IDEA (jetbrains.com)](#)

- [JUnit 5 User Guide](#)