COMP 210: Object-Oriented Programming **Lecture Notes 1**

Java Program Structure and Eclipse

Robert Utterback

In these notes we talk about the basic structure of Java-based OOP programs and how to setup and build them with the Eclipse IDE. We'll more or less ignore Java syntax for now as the code that's written in this process is written by the IDE.

Overview

All code in Java *must* be in a class or something similar[1]. *A source file can only contain one public class and the file name must match the class name*. So, if programs are built from the interaction of many objects and we need multiple classes to define the types of those objects, then we're quickly going to be dealing with programs whose code is spread across many, many files. Classes are organized within *packages*. Just like we did with C++ namespaces, we'll always define our classes within packages. Not using packages is not good style. Package placement will also have an impact on the scope of the definition as well.

[1] You'll learn about some other class-like definitions

We'll continue to use unit testing in the design and implementation of our programs. The go-to framework in Java is JUnit[2]. You'll find it's very similar to GTest. Just like before we'll separate tests from code. Each class will get its own file of tests, which will also be a class of course. That class will typically get placed in the same package as the class its testing.

[2] `http://junit.org`

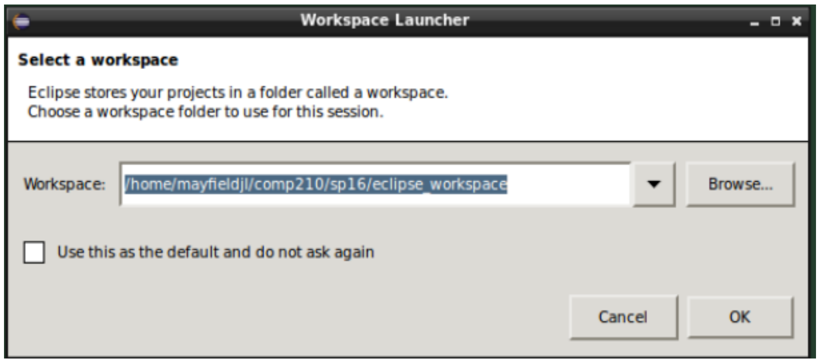
To start out we'll work with programs composed of several classes that are defined in separate files and done so relative to a package.

Eclipse Projects and Project Structure

Eclipse is a programming IDE. It's written in Java and very well suited for Java development. Through plug-ins you can use it for a wide variety of other languages as well. As a batteries included IDE it includes a robust text editor, integrates with a compiler, and allows you to run and debug programs from within the IDE itself.

Programs in Eclipse are represented as *projects*. Projects are organized within a *workspace*. The workspace corresponds to directory and the program to a sub-directory of the workspace directory. When you start Eclipse you'll be prompted to select a workspace.

I recommend you create a class specific workspace, either way, make a fresh, new directory to act as your Eclipse workspace. You can also check the box to make this the default. You can always switch workspace from the File menu in Eclipse.


Workspace Launcher

Select a workspace

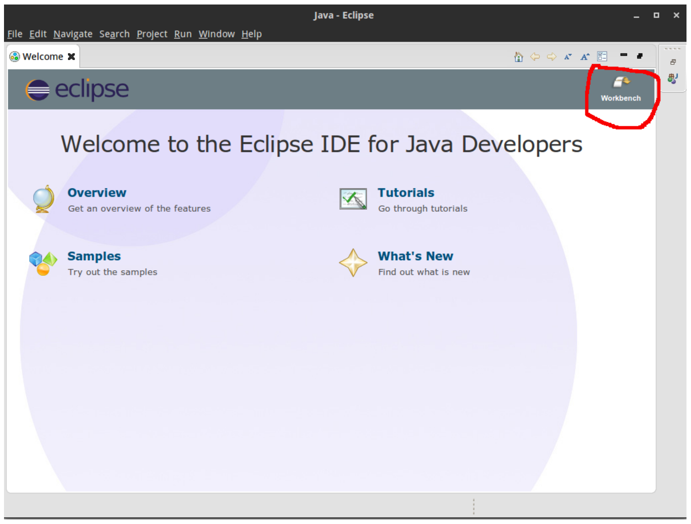
Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: /home/mayfieldjl/comp210/sp16/eclipse_workspace

☐ Use this as the default and do not ask again

Figure 1: The Eclipse workspace selection window

The first time you launch Eclipse you're greeted with a Welcome screen. Just click the icon to take you to the workbench, where we'll be working.



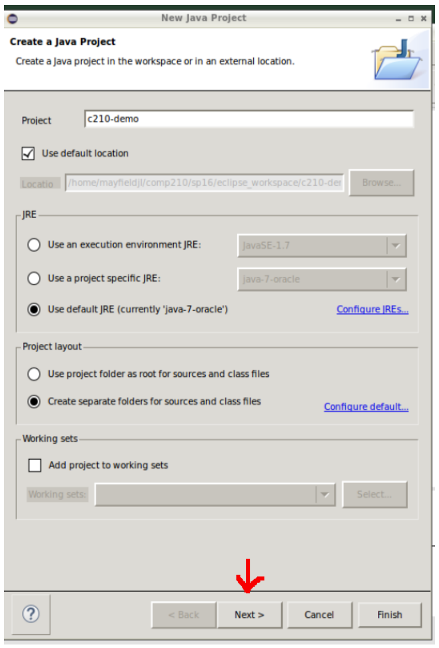
Welcome to the Eclipse IDE for Java Developers

Figure 2: The Eclipse Welcome Screen. Just go to the Workbench.

You're now ready to create a *Java Project* you can do this in the usual ways, from File > New or the New Quick button. Once there, choose a project name and ensure your settings look like those in figure 3. Then click Next.

Next we need to do some setup for using our unit-testing framework JUnit 5[3]. There are two things to do: create a folder to hold all out test code and add the JUnit library to the project. The folder is optional, but it's nice to keep our tests separate from our actual code. To create the folder you'll hit the new source folder link. In the dialog that pops up, name the new folder *tests* and click Finish. You can see this in figure 4.

[3] `http://junit.org/`



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project: c210-demo

[✓] Use default location

Location: /home/mayfieldjl/comp210/sp16/eclipse_workspace/c210-der

JRE

( ) Use an execution environment JRE: JavaSE-1.7

( ) Use a project specific JRE: java-7-oracle

(●) Use default JRE (currently 'java-7-oracle') Configure JREs...

Project layout

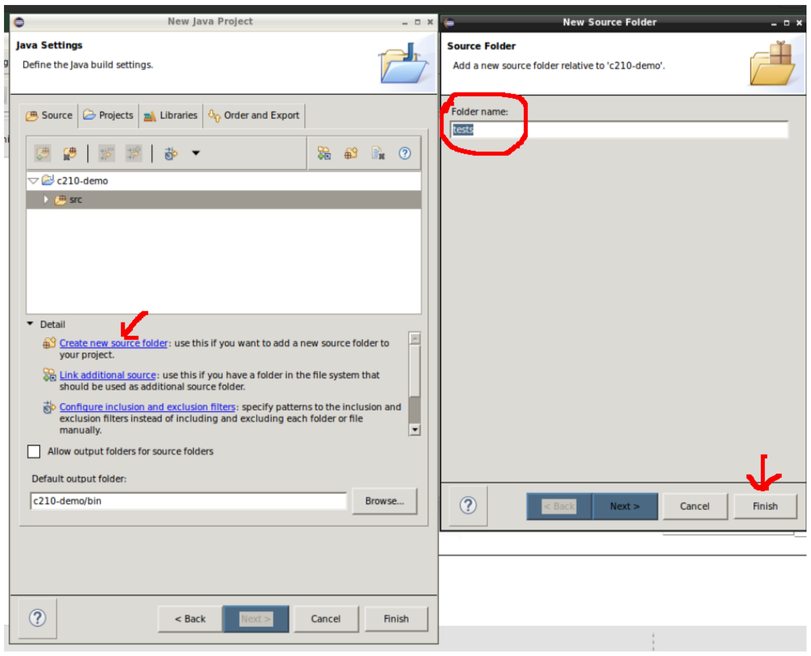
( ) Use project folder as root for sources and class files

(●) Create separate folders for sources and class files Configure default...

Working sets

[ ] Add project to working sets

Figure 3: First Screen in New Project Dialog



New Java Project

Java Settings

Define the Java build settings.

Source | Projects | Libraries | Order and Export

c210-demo
src

Detail

Create new source folder: use this if you want to add a new source folder to your project.

Link additional source: use this if you have a folder in the file system that should be used as additional source folder.

Configure inclusion and exclusion filters: specify patterns to the inclusion and exclusion filters instead of including and excluding each folder or file manually.

[ ] Allow output folders for source folders

Default output folder: c210-demo/bin

New Source Folder

Source Folder

Add a new source folder relative to 'c210-demo'.

Folder name: tests

Figure 4: First Screen in New Project Dialog

To add the JUnit library itself, you need to select the *Libraries* tab, then select "ClassPath", then "Add Library...", then select JUnit in the dialog that pops up. Then press the Next button. You can see this in figure 5. The last window ensures you have the right version of JUnit. It should look like figure 6.

Once you've setup the test folder and library, you're project is all setup.

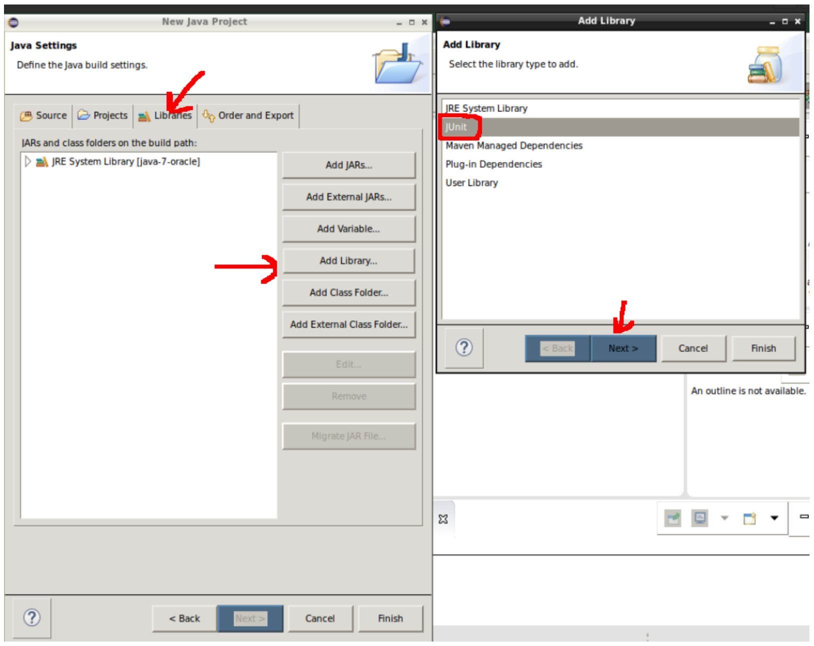
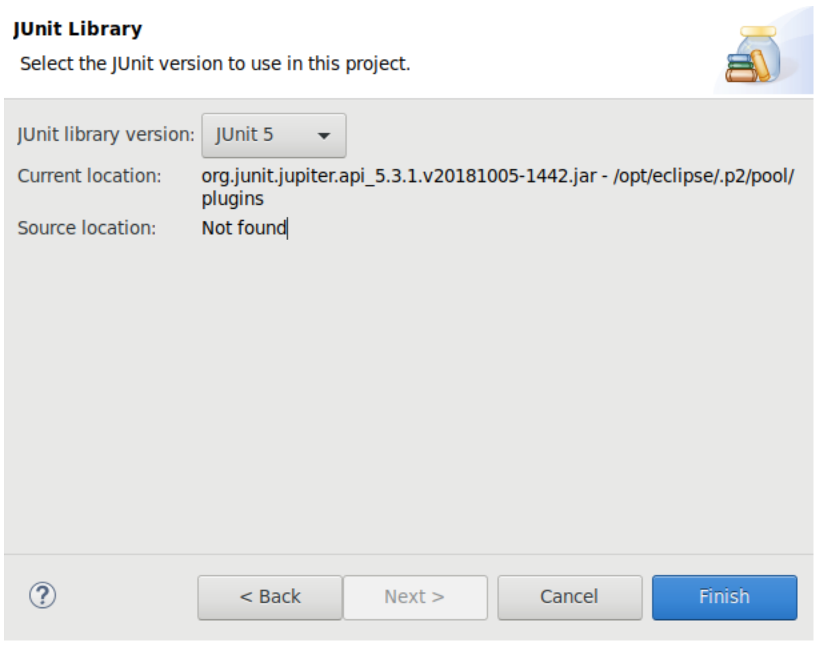


Figure 5: Dialogs for Adding JUnit to a Project


JUnit Library

Select the JUnit version to use in this project.

JUnit library version: JUnit 5

Current location: org.junit.jupiter.api_5.3.1.v20181005-1442.jar - /opt/eclipse/.p2/pool/plugins

Source location: Not found

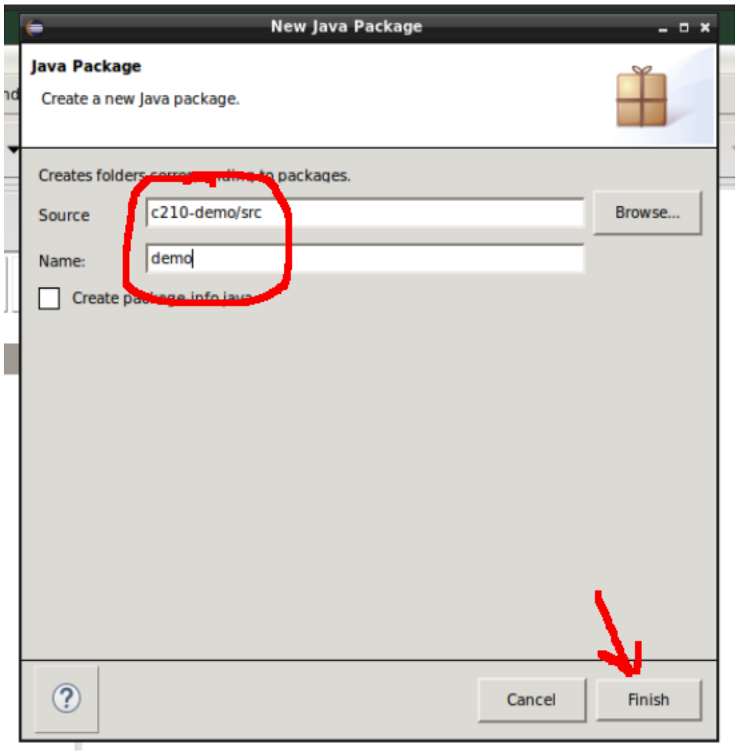
Figure 6: Dialog to Select version 5 of JUnit

Packages

Before we start creating classes we need to create a package or two for those classes. Packages are created within source folders and correspond to physical sub-directories within that folder. So, unlike C++ Namespaces, packages have a physical presence.

If our tests and classes are going in the same package then we need to create that package twice, once in the *src* folder and once in the test folder. There are at least three ways to create a new package: File > New, the New quick button, and right clicking the folder in which the package is to be placed. In either case you end up at a dialog like figure 7 that lets you specify the parent director and name of the new package. **The name of your package should start with a lower case letter**[4]. This is a convention and not a syntax error. You should consider it an error though as you'll lose points for violating the convention.

[4] Eclipse will warn you if you use Uppercase



Source: c210-demo/src

Name: demo

Figure 7: New Package Dialog

When you're done your project should show the packages within their respective source folders as seen in figure 8.

With packages in place, we're ready to start creating Classes, i.e. source files.

Classes and Files

To create a new Class you an use the usual options: File > New, the New quick button, or right click New on the package in which you want the Class placed. We'll always start by stubbing out the class. We can then use Eclipse to stub out the tests. You'll find that Eclipse can auto-generate a lot of boiler place code for you. This is good because it'll save time. It's bad because it's all too easy for you to ignore that code and not know how to write it yourself. This will undoubtedly lead to you losing points on quizzes and exams or botching an interview for an internship or job. So, *just because the IDE*

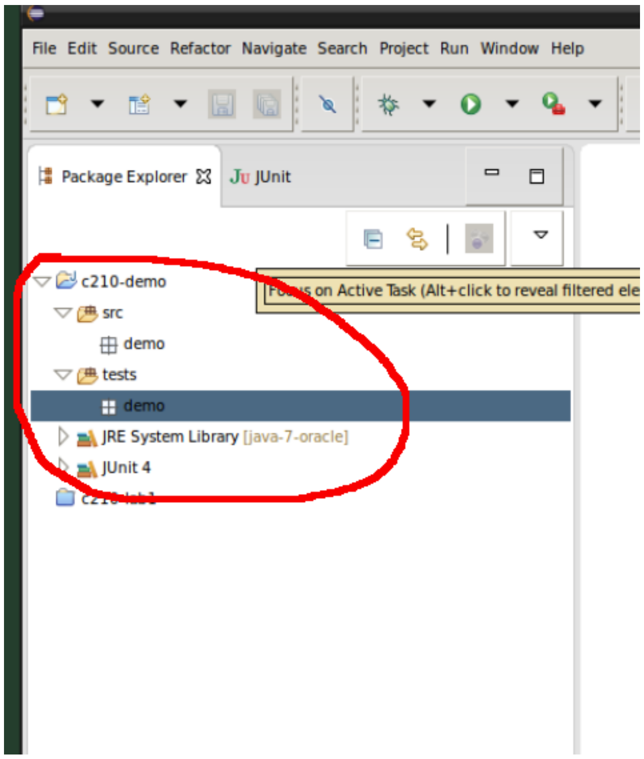

File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
JUnit
Focus on Active Task (Alt+click to reveal filtered ele
c210-demo
src
demo
tests
demo
JRE System Library [java-7-oracle]
JUnit 4
c210-lab1

Figure 8: A Project with src and tests folders and a shared package

writes the code for you doesn't mean you don't need to know how to do it yourself!

The new Class dialog has lots of options. We'll deal with them on an as needed basis. The core options are specifying the location, package, and name of the class. **The class name should start with an uppercase letter**. This is, again, a convention that we'll follow as a rule. The class name becomes the file name. Java files have the unsurprising extension of *java*. Figure 9 points the core options and one extra option, stubbing out a *main*, that's discussed a bit below.

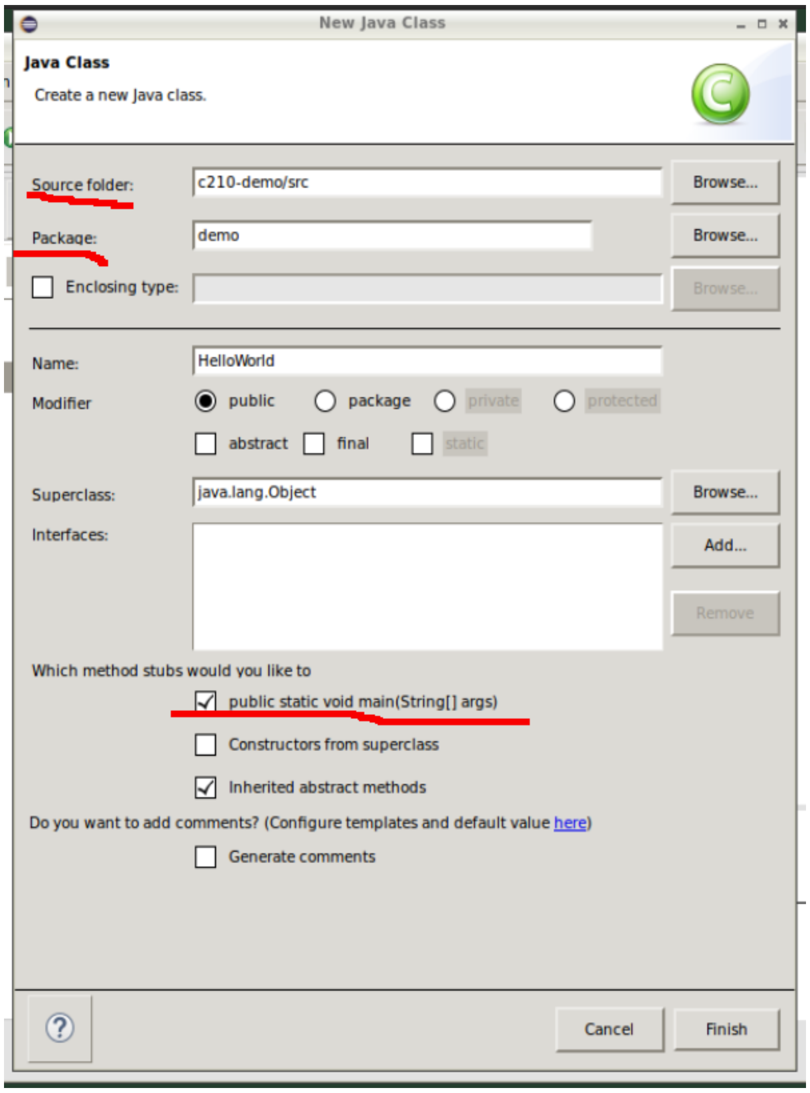
The main procedure

The other New Class dialog option I'll point out now is that this dialog give you the option to automatically generate a stub for *main*. Java, like C++, operates by executing a *main* procedure. It's the one procedure we'll write with any kind of regularity.

Compile and Run

In figure 10 you'll see a Hello World program in Java. We'll leave the details of the code itself for the next set of notes. For now you should notice a few things highlighted in the figure. The green play button compiles and runs the program. Console output[5] happens down at the bottom. In code, the inclusion of a file/class in a package

[5] and input



Java Class

Create a new Java class.

Source folder: c210-demo/src

Package: demo

☐ Enclosing type:

Name: HelloWorld

Modifier: [✓ public] package private protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to

✓ public static void main(String[] args)

☐ Constructors from superclass

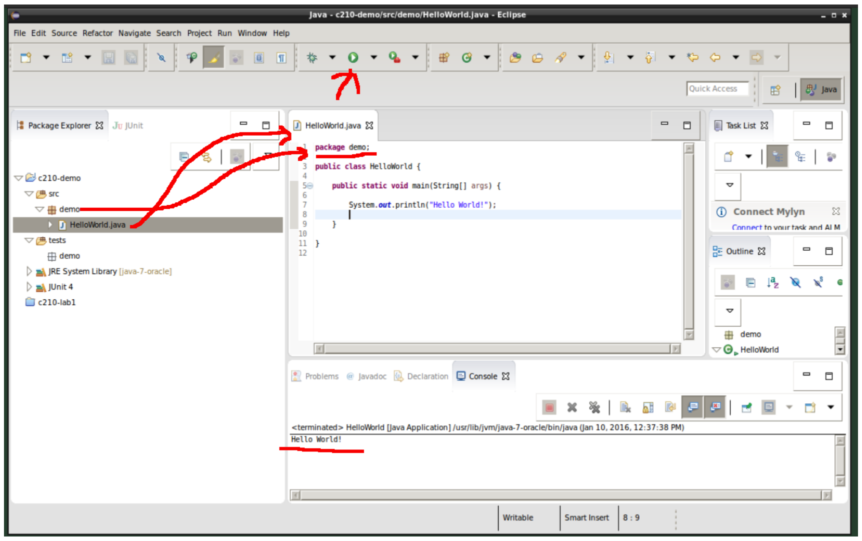
✓ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Figure 9: New Class Dialog with core options highlighted.

takes place with a simple package declaration. Files are opened and edited in the center of the window. IF you've been following along, then you're encouraged to play around a bit. Go find some other Java tutorials and make a more interesting Hello World. In the next set of notes we'll design, document, implement, and test some procedures in order to get a better feel for the nuts and bolts of Java.



```java
package demo;

public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello World!");

    }

}
```

```
<terminated> HelloWorld [Java Application] /usr/lib/jvm/java-7-oracle/bin/java (Jan 10, 2016, 12:37:38 PM)
Hello World!
```

Figure 10: Hello World in Java