

1 Introduction

Eclipse is an Open Source IDE (integrated development environment).¹ It can significantly increase productivity with build and test automation, code management and visualization, profiling, and debugging. In this lab you will learn the basics of Eclipse by:

1. Creating an Eclipse Java project and a package for your code
2. Importing existing code
3. Setting up program run configurations
4. Adding JUnit test cases
5. Debugging the code

Since this lab focuses on an application, the directions can become involved. **Be sure to read the instructions carefully! If you make a mistake, it can be difficult to figure out what happened.** The code that you'll be working with is a graphical version of tic-tac-toe, a game that you're hopefully all familiar with.

2 Getting Started

To run Eclipse, simply type `eclipse &` into your shell. When you first start Eclipse, it may take some time, and then you will be prompted for a workspace directory. Simply choosing `/home/username/workspace` is probably best, but be forewarned that the Eclipse metadata can end up taking up a lot of space. If you don't check the box at the prompt, Eclipse will ask for the workspace directory every time. You can switch workspaces by selecting `File` → `Switch Workspace`.

Once you've created your workspace, you'll be welcomed by Eclipse. Now close the welcome window (click "Go to workbench" if necessary), so we can create a new project.

3 Creating a New Project and Package

First, switch into the Java perspective by going to `Window` → `Open Perspective` → `Java Browsing`. Notice how the screen rearranges itself.

1. Go to `File` → `New` → `Project...`
2. Select `Java Project` from the list and click `Next`
3. Give it a suitable name, like "Eclipse Lab" or "cs032_lab01."
4. You should create it in the existing workspace.
5. Click 'Finish' (you could go on to configure dependencies etc., but there's no need to now).

To keep your files organized, you should create a java package for them.

1. Right click on your newly-made project and select `New` → `Package`

¹Sun...Eclipse...get it?

2. Name it `edu.brown.cs.cs032.tictactoe` and click **Finish**.

NOTE: any files inside this package should have the proper package declaration at the top (e.g. `package tictactoe;`). These are already in the code we give you.

4 Importing the Lab Code

Having an empty project and package isn't particularly useful, so let's import some code.

1. Right click on your newly-created package in the left window, and then choose **Import...**, select **General** → **File System**, and click **Next**.
2. Browse for the directory that contains your `.java` files (in this case `/course/cs032/pub/labs/01_eclipse/`).
3. Check the box next to the folder icon and the text "01_eclipse" on the left side of the dialog box.
4. You should see checkboxes appear next to all of the `.java` files listed on the right, which means that they will be imported. You could also click on them individually.
5. Click **Finish**.

You should see the four java files in your package now (click the arrow next to your package to expand its hierarchy if you don't see them).

You have reached **CHECKPOINT 1**.

5 Running the Code

The Run button – the green circle with a white triangle ('play' icon) in the top toolbar – will run your code. Click the black down-arrow on the side of the Run button and choose **Run...** from the drop-down menu. In the management window's list pane on the left, right-click on **Java Application** and select **New**. It should automatically create a run task named **TicTacToe** under the **Java Application** section (if it does not appear, open up `TicTacToe.java` and try again). When you're ready, click **Run** all the way at the bottom of the window.² Play around with the game a bit and note that it's not quite working correctly.

6 Refactoring

In addition to being able to write new code, Eclipse comes with tools for efficiently and *correctly* enacting sweeping changes to the code you already have. For example, a variable with a name like `stuff` not terribly useful, and it should probably be renamed to reflect its usage.

Normally, when you try to rename a variable, class, interface, etc., you have to go digging around through all of your old code, changing references and types everywhere (often manually to avoid find/replace errors). Thankfully, Eclipse provides you with a set of clever refactoring tools to help you enact sweeping code changes without the tedium. Refactoring is an operation on code that does not change the meaning of the code, just ambiguous details such as variable names. Eclipse refactors on a package-wide level, so you don't have to worry about finding all of the appropriate references.

Take a look at the variables declared at the top of `TicTacToe.java` and compare them with those in `GameBoard.java` and `GameSquare.java`. See the difference? These files were written by two different TAs (we're not making this up), one of whom created classwide variables preceded by `'m_'` and the other used the convention of an internal `'.'`. Though it may not seem like a big deal, in extremely large projects among

²Note that you can enter command-line arguments and arguments to the JVM (Java Virtual Machine) in this window as well. Any subsequent clicks on the Run button will run your program with the configuration that you last set up in the Run window, so you don't need to re-enter complex arguments.

multiple collaborators, maintaining readable code of similar style greatly aids efficiency. It's time to rename the variables in `GameBoard.java` and `GameSquare.java` to use the internal '_' convention without an initial 'm.'

To rename a variable:

1. Left-click on it to highlight it,³ and then right-click on it.
2. Select **Refactor** → **Rename...** and type the new variable name in the text field. Make sure that **Update references** is checked, then click **OK**.
3. Do this for all four classwide variables in `GameBoard.java` and three in `GameSquare.java`.

You can often also preview your changes before clicking **Preview**, but for changes like renaming a variable, it shouldn't be necessary. Further, you can undo refactoring just as you would typing.

7 Importing Libraries

It is often necessary, especially in cs032, to make use of both external and internal libraries. As you've experienced with Solar, it is often necessary to go through a few steps to get libraries working with the rest of your code, namely by adding them to your build path. We'll begin by adding the JUnit library to your build path; no project should be done without unit testing and JUnit is a great way to go about it.

To add the JUnit jar file:

1. Right-click on your project, and select **Properties**.
2. Select **Java Build Path**.
3. Click the **Libraries** tab.
4. Click on the **Add External JARs...** button.
5. Navigate to `/course/cs032/lib/junit.jar`, and click **OK**.

Though you will not be using it in this lab, note the **Add JARs...** button. This is for importing JARs that are inside of your project or in other projects in your workspace, rather than being located elsewhere. This is especially useful when you are working on your final projects in teams, as it creates a relative path within the project to access the JAR, whereas **Add External JARs...** creates an absolute path.

8 Testing the Lab Code

Don't worry, it's the TAs' fault – they forgot to test their code! Fortunately, Eclipse is well integrated with JUnit. To begin, we will be setting up a JUnit test. Eclipse has a tool for automating the process of creating a JUnit test.

1. Select your project folder.
2. Four buttons to the right of the **Run** button there is a button with a picture of a green circle with a 'C' in it. Click the black down-arrow on that button, and choose **JUnit Test Case**.
3. Choose a name for the new test class (something simple like "TicTacToeTest" is good).
4. Check the box marked **setUp()** and click the **Finish** button. This allows you to automatically generate method stubs for methods that you want to override.

³You can click on any instance of it in your code – it doesn't have to be where you declared it. You should wait until you see a gray background behind the variable name telling you that Eclipse has selected it.

You should now have a stencil JUnit test definition. Add the following to the class like you did for the JUnit lab, where `<Class Name>` is the name you chose for your JUnit test class:

```
public static Test suite() {
    return new TestSuite(<Class Name>.class);
}
```

At this point, `Test` and `TestSuite` should be underlined in red. Eclipse is informing you that there is an error in the code. Nifty, huh? When you hover your mouse over `Test`, Eclipse will tell you what the error is (“Test cannot be resolved to a type”). This error means that the class `Test` has not been imported. What import statement do you need to get `Test`? I don’t know, but Eclipse does! Here’s how to fix it:

1. Left-click on `Test` and then right-click on it.
2. Go down to the `Source` menu, and select `Add Import`. Voila! Eclipse has added the correct import.
3. Let’s try to do the same thing another way. Right click on the little lightbulb with a red x on the left margin across from `TestCase`, select `Quick Fix`, and then select the first item, `Import ‘TestSuite’ (junit.Framework)`.

As you may have noticed, Eclipse offers many ways to do the same thing. Explore and find the ones you like.

Now, you need to add some test functions. Remember to name your test something starting with `test` and to make it `public`. The TAs have written one to test whether the game correctly ends when a player ‘X’ has filled the diagonal.

```
...
private TicTacToe m_game;

protected void setUp() throws Exception {
    super.setUp();
    m_game = new TicTacToe();
}

public void testDiagonal()
{
    m_game.parseInput(0, 0); // X goes
    m_game.parseInput(1, 0); // O goes
    m_game.parseInput(1, 1); // X goes
    m_game.parseInput(2, 0); // O goes
    m_game.parseInput(2, 2); // X goes

    char[][] board = m_game.getBoard();

    assertTrue(board[0][0] == 'X');
    assertTrue(board[1][1] == 'X');
    assertTrue(board[2][2] == 'X');
    assertTrue(m_game.isGameOver());
}

...
```

To run the test, right click on the test in the Package Explorer, select `Run As...`, and then choose `JUnit Test`. The TA test should pass. Now write one to test whether the game correctly ends when a player ‘X’ has filled the top row. This test should fail.

Whenever you detect and fix a bug, you should write a test to catch similar bugs in the future.⁴ But this won't help you when you don't know where the bug is! You need to use a debugger. You have reached **CHECKPOINT 2**.

9 Debugging the Lab Code

Now it is time to find why the TAs' code is not working properly. **NOTE: only look into `TicTacToe.java`; the GUI should work just fine.** Also note that our code keeps two representations of the board: one in the GUI and one in the game logic in `TicTacToe.java`. This is redundant for the purposes of this lab.

Eclipse has an excellent debugger. We can begin by setting a *breakpoint* in `parseInput()`, as this is the main function that executes a player's move. When you run your program in Eclipse's debugger, breakpoints pause execution, allowing you to inspect variables and step through the code one line at a time. You can add a breakpoint by double-clicking on the vertical blueish bar at the left edge of the code editor window. Place a breakpoint at line 50 next to the first `if` statement so we can begin stepping through the method to figure out where everything goes wrong.

To the left of the run button is a green bug icon; this is the debug button. It works just like the run button, only Eclipse will stop at breakpoints. Click its arrow and select `TicTacToe`. Eclipse should switch automatically to the debug perspective (it might ask you if you'd like to do so, and you should say yes).⁵

The GUI should show up as usual. Click on a the upper-left square (square at 0,0) and you should enter debug mode, where the breakpoint line will be highlighted.

The upper-right of the screen shows what the local variables are. If you expand `this`, you will see all the instance variables. If you expand `_board` and highlight all the rows (hold Ctrl and click on each one), you will see a representation of the board array as it is represented in the game logic. Keep this highlighted to see what's going on when the player makes a move. Click the 'resume' button (yellow and green) to continue the program and click on the square to the right of the one you already clicked on (square at 0,1). The program will hit another breakpoint.

Some of the variables listed in the upper right of the screen are now highlighted yellow; Eclipse will automatically highlight values that have changed in this manner. This time, instead of resuming the execution, step through the code by clicking on the 'step over' button until you reach line 62.

The 'O' was added to (1,0) instead of (0,1)! This is odd, because the player clicked on (0,1). This implies that the bug is probably in the code around line 62 that inserts a player's move in the proper place in the array. Can you find where the bug is? You may find it helpful to do few more resumes and step-throughs to get a feel for what's going on.

Once you believe that you have found the bug, you will have reached **CHECKPOINT 3**.

Now that you have been introduced to the debugging tools available to you in Eclipse, you are ready to let go of `System.out.println` (though it's still useful occasionally) and join the debugging elite. We only touched on some of the capabilities of Eclipse's debugger, and we encourage you to read up on it and explore all the features it has to offer.

⁴If you fix a bug, the code will work for a day. If you write a test to detect a bug, the code will work for the rest of your life. That's how the saying goes, right?

⁵If Eclipse doesn't automatically switch to the Debug Perspective, go to **Window** on the top menu bar and select **Open Perspective** → **Debug**.