Unit Test Walkthrough

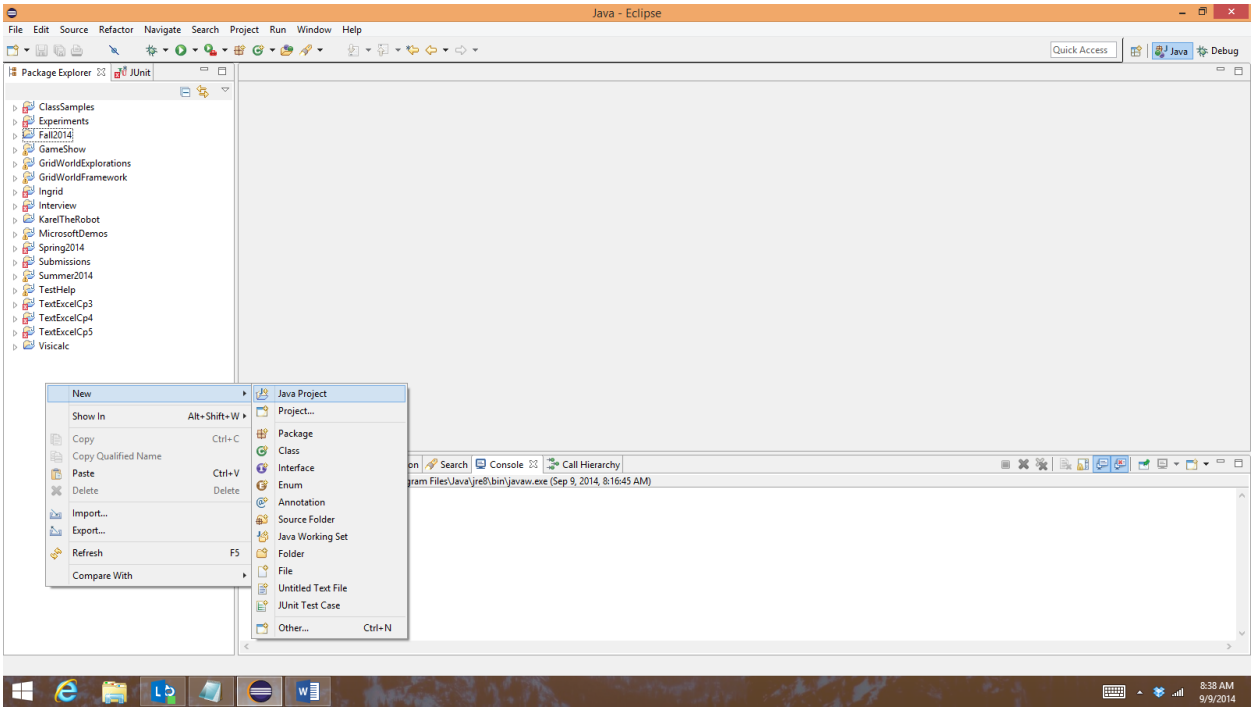
At various points in AP CS, we will give you a set of automated tests, also known as unit tests, that can prove that your code works the way it is supposed to. You can (and should) run these tests yourself before you turn in your code, since they will tell you whether you've done things correctly.

This walkthrough will show you all the steps to set up a new project in Eclipse with code like what you would write and some tests like what we would normally provide you.

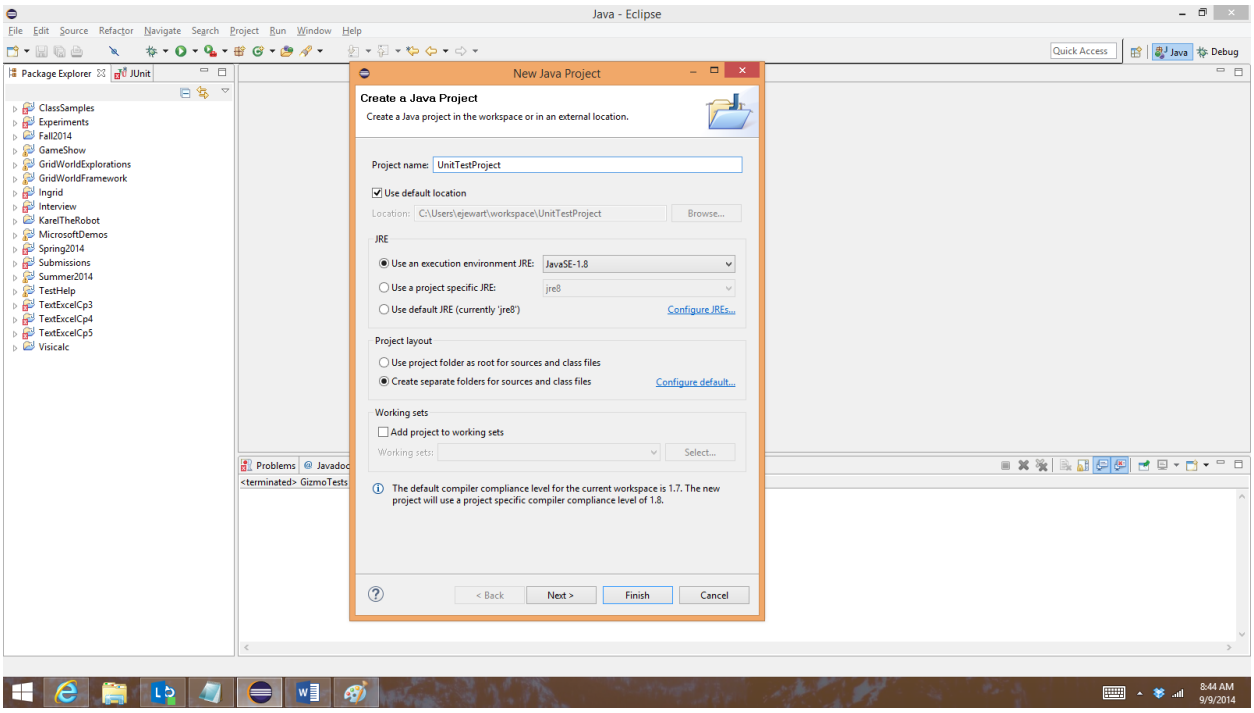
The code we'll work with is a class called Gizmo. Gizmo has one method, greet(), which reads whatever the user types and responds with a greeting. Gizmo provides a slightly different greeting depending on what the user types. In future assignments, you will probably be writing some code instead of using Gizmo, but since we want to focus on automated tests in this lab, we'll give you the Gizmo code.

Let's get started! Follow the steps carefully—if you miss one, something will not work and you'll have a hard time figuring out what…

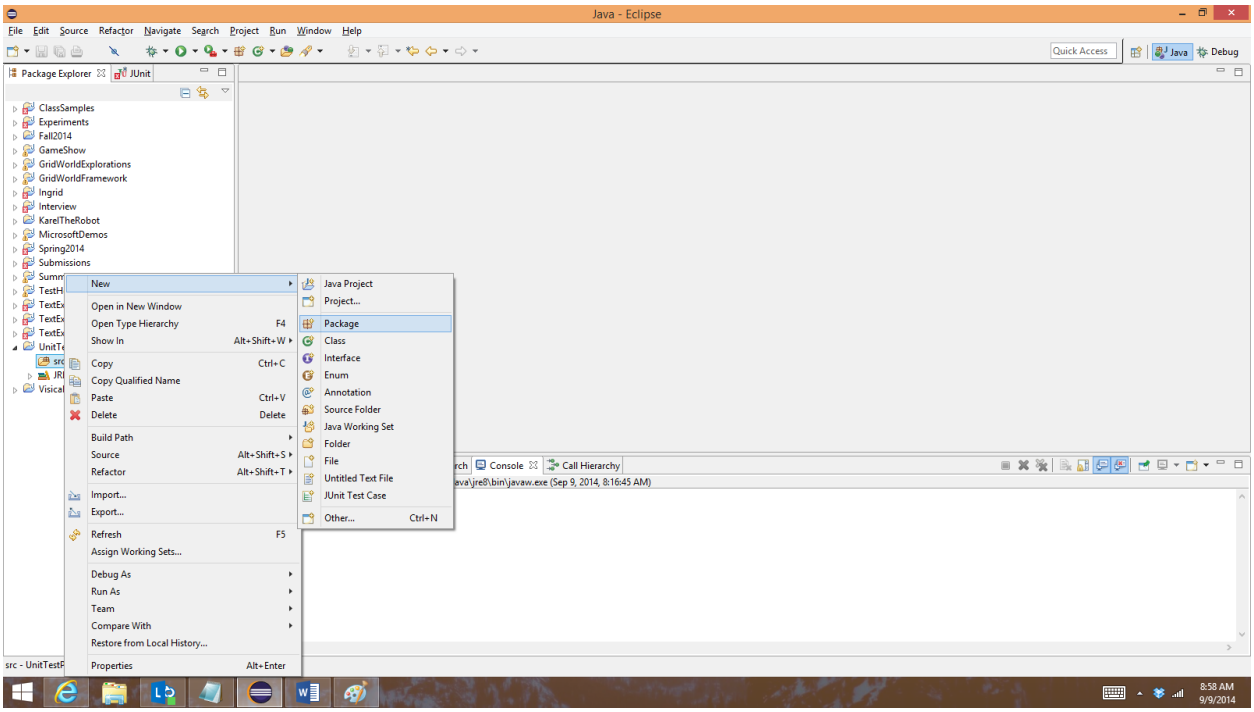
1. Launch Eclipse.
2. Find the Package Explorer window (normally at the top left) and right-click in the white space somewhere in that window. Choose New, then Java Project from the popup menu.



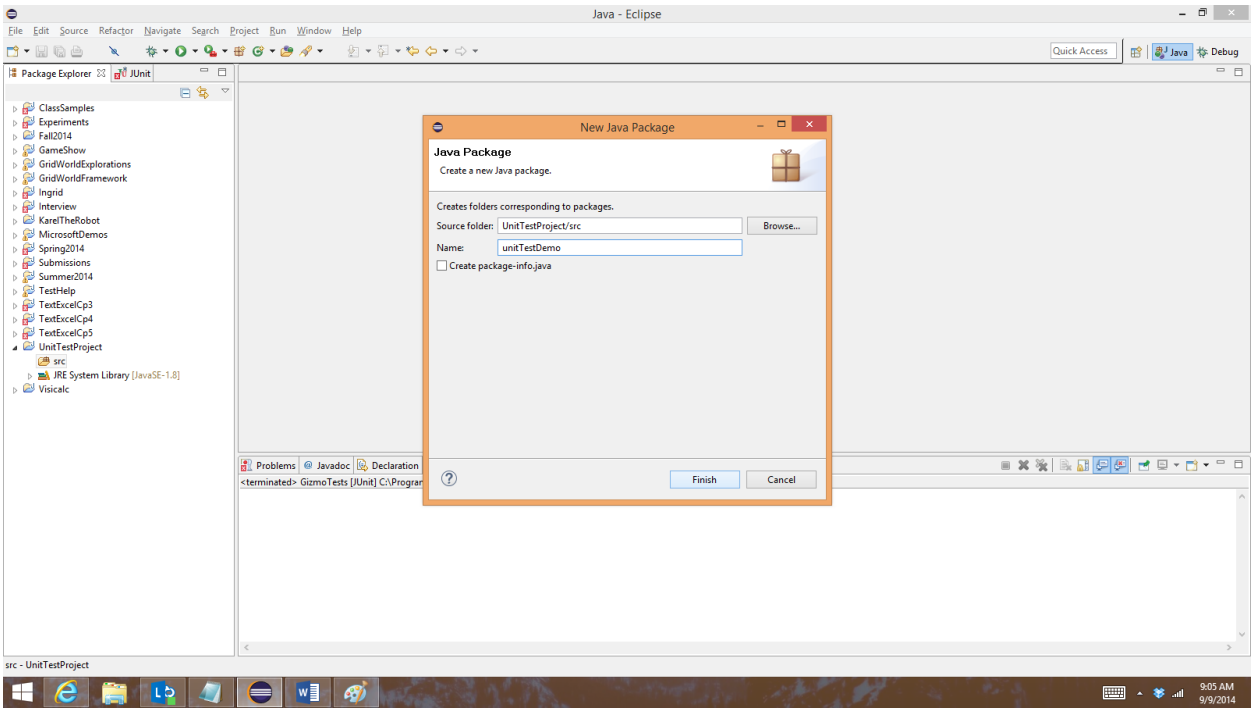
3. In the New Java Project window that pops up, name your project UnitTestProject, then click Finish.



4. Double-click on the new UnitTestProject folder in PackageExplorer to expand it, so you can see the src folder underneath it.
5. Right-click on src and choose New, then Package.

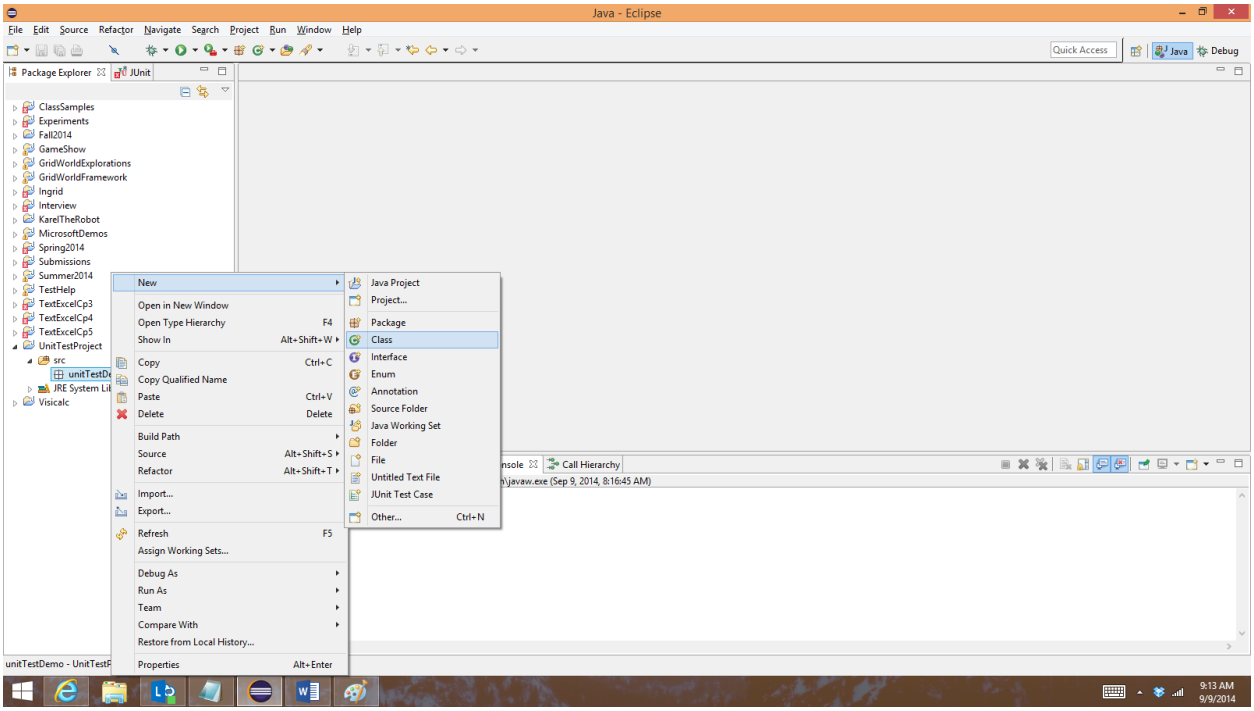


6. Name your package “unitTestDemo” (exactly like that, including the capitalization) and click Finish.

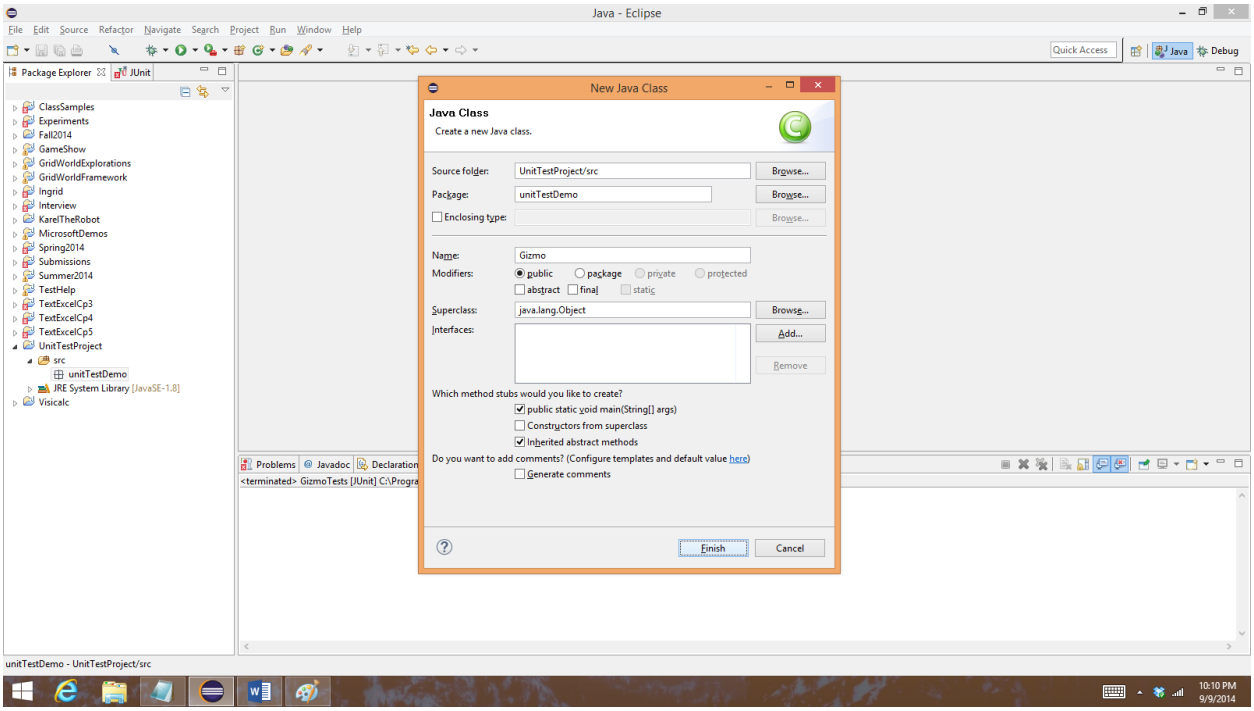


Now you have a project called UnitTestProject with a package called unitTestDemo. Setting up a project and package are things you may have to do other times this semester (for example, you may want a separate project for Fractional Calculator when we start that project. We’ll continue by adding the Gizmo class.

7. Add a new class by right-clicking on the unitTestDemo folder and choosing New, then Class.



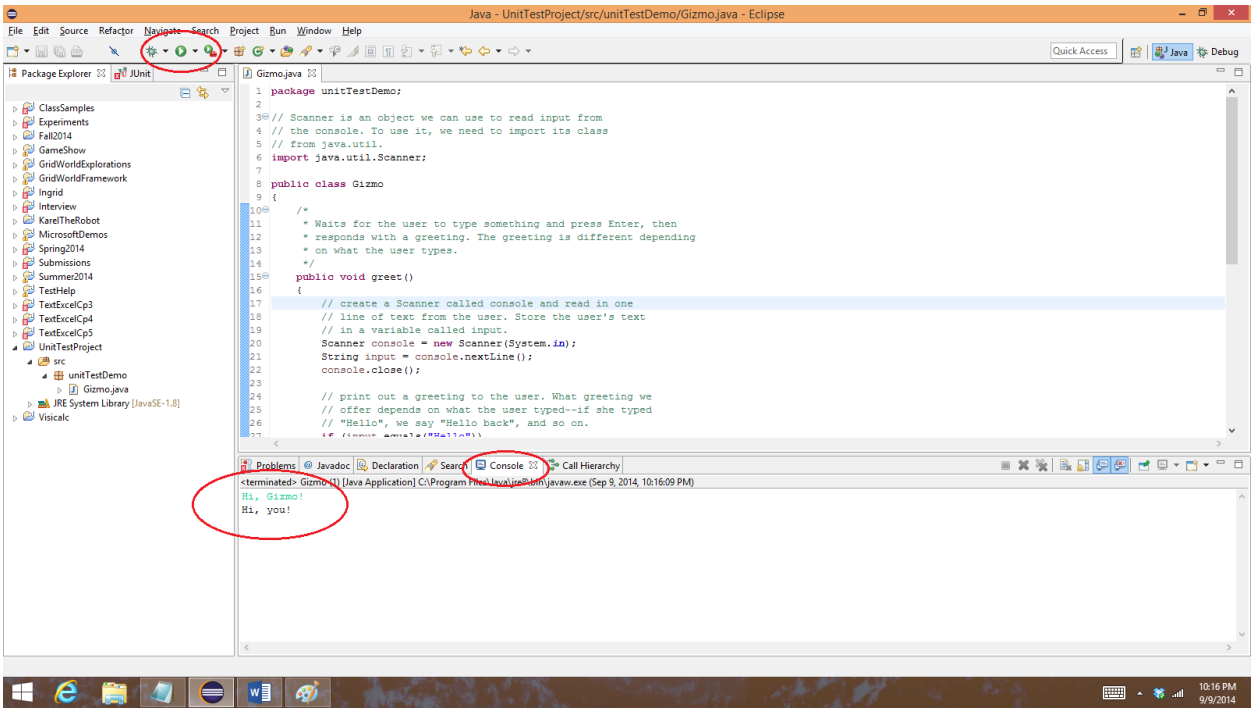
8. Name your class Gizmo. Click Finish. Make sure you spell Gizmo exactly the same way, with a capital G.



9. Copy and paste the following code into your new Gizmo.java file, replacing whatever is already there.

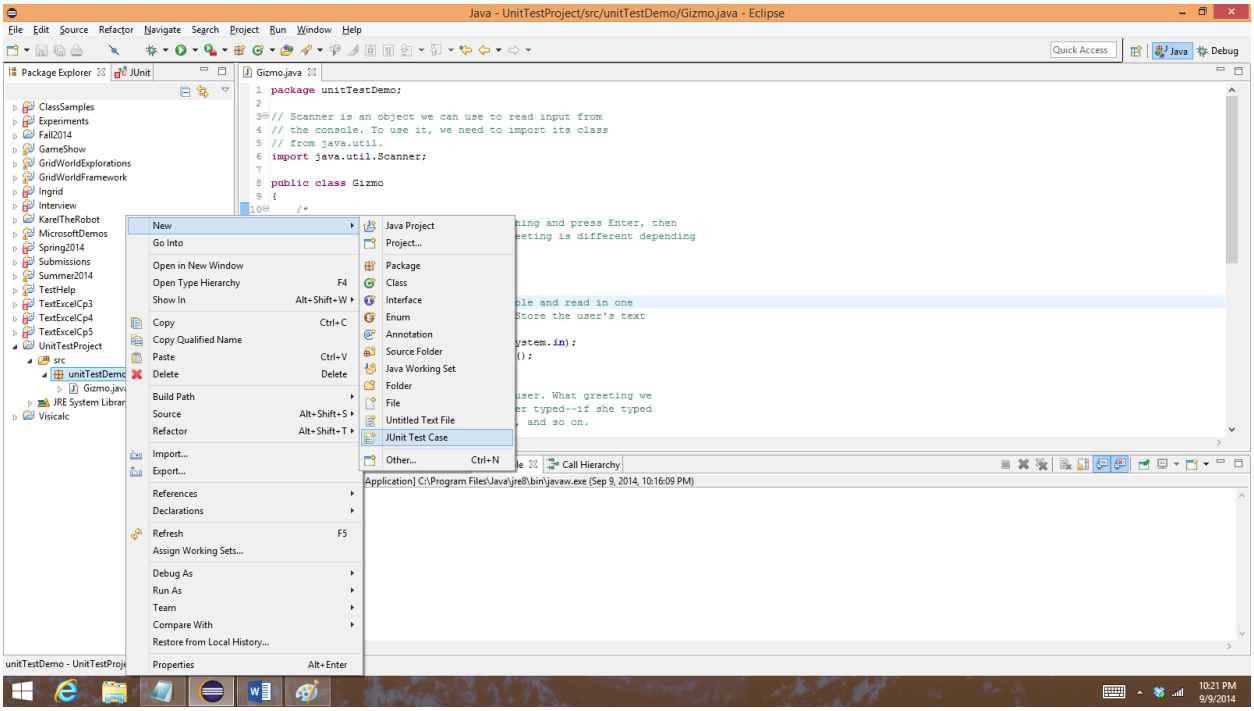
```
package unitTestDemo;

// Scanner is an object we can use to read input from
// the console. To use it, we need to import its class
// from java.util.
import java.util.Scanner;

public class Gizmo
{
	/*
	 * Waits for the user to type something and press Enter, then
	 * responds with a greeting. The greeting is different depending
	 * on what the user types.
	 */
	public void greet()
	{
		// create a Scanner called console and read in one
		// line of text from the user. Store the user's text
		// in a variable called input.
		Scanner console = new Scanner(System.in);
		String input = console.nextLine();
		console.close();

		// print out a greeting to the user. What greeting we
		// offer depends on what the user typed--if she typed
		// "Hello", we say "Hello back", and so on.
		if (input.equals("Hello"))
		{
			System.out.println("Hello back");
		}
		else if (input.equals("Hi, Gizmo!"))
		{
			System.out.println("Hi, you!");
		}
		else
		{
			System.out.println("Hey");
		}
	}

	/*
	 * All Java programs start with a main method. In this one, we
	 * create a new Gizmo object called 'g' and tell it to greet
	 * the user.
	 */
	public static void main(String[] args)
	{
		Gizmo g = new Gizmo();
		g.greet();
	}
}
```

10. Run Gizmo by pressing the arrow in the green circle in the toolbar. At first, it will look like nothing has happened, but if you type a greeting in the Console window, Gizmo will respond.

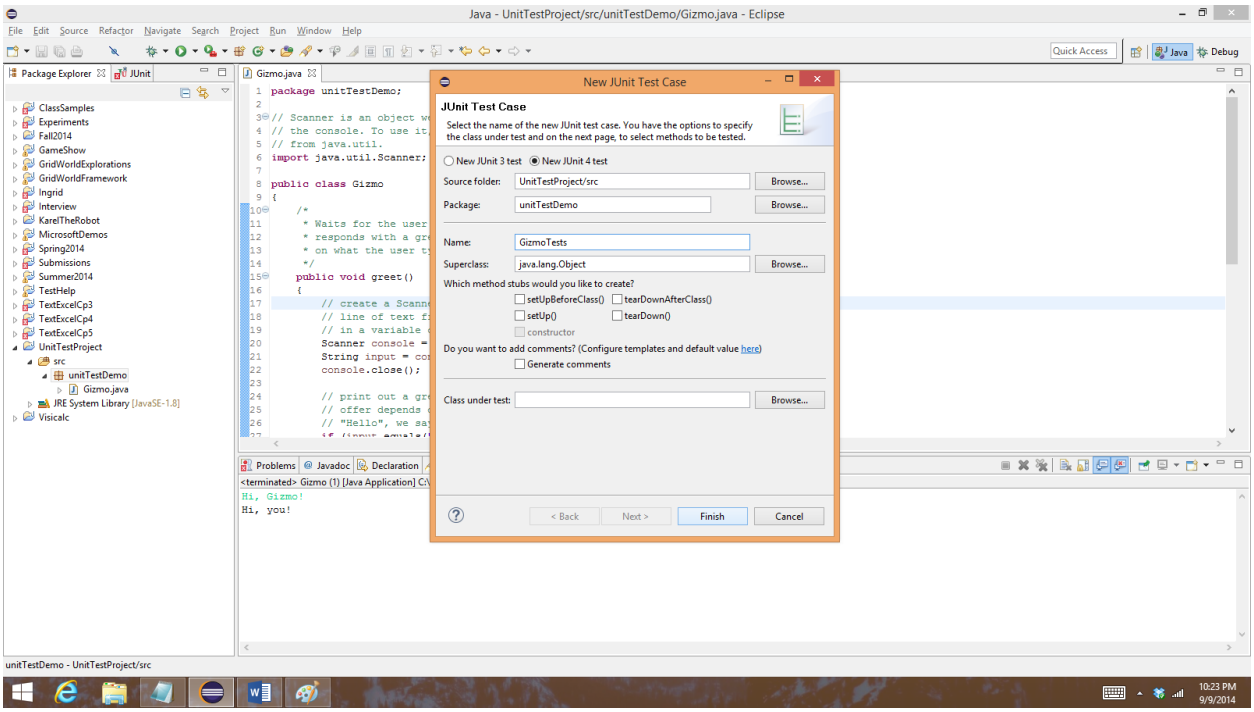


At this point, you have a working program. In most labs and projects, you will have to write code instead of simply copying and pasting it from the assignment (though you may start with some code to copy), but knowing the steps to create a new class and add some code to it are going to be important this semester. Next, we'll learn how to add some unit tests that show whether the Gizmo program is doing what it's supposed to.

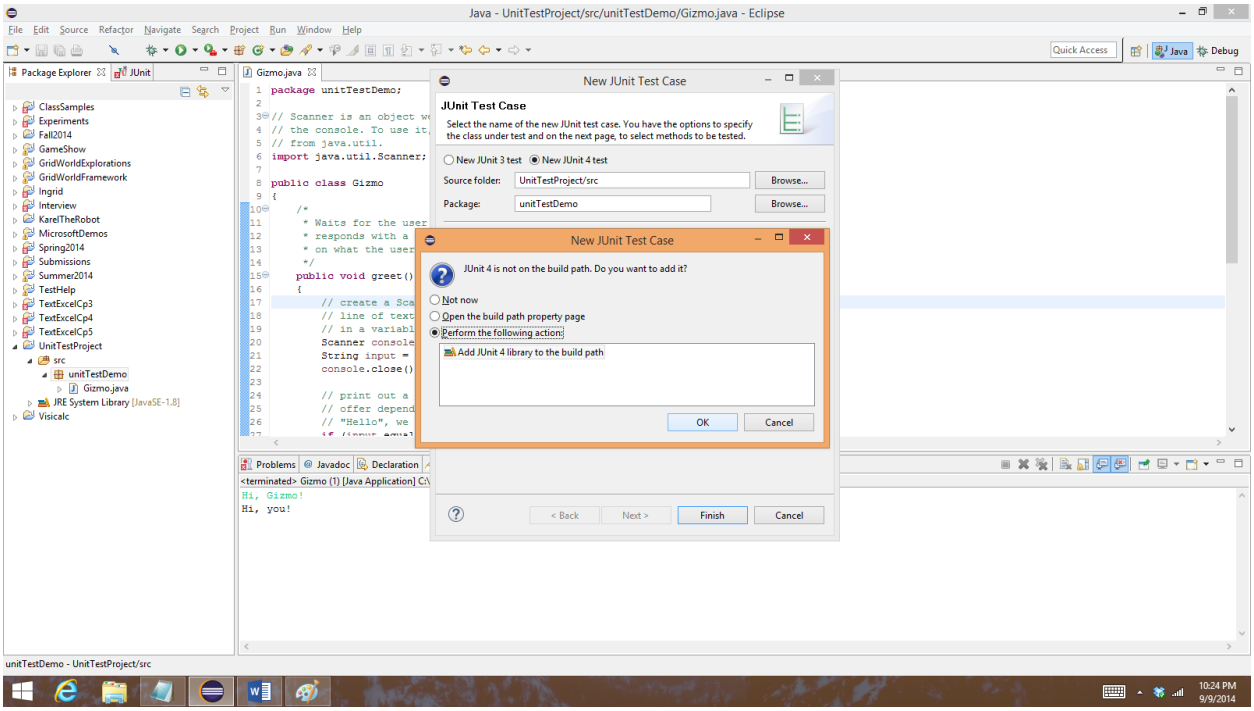
11. Right click on the unitTestDemo package again, and this time choose New, then JUnit Test Case.



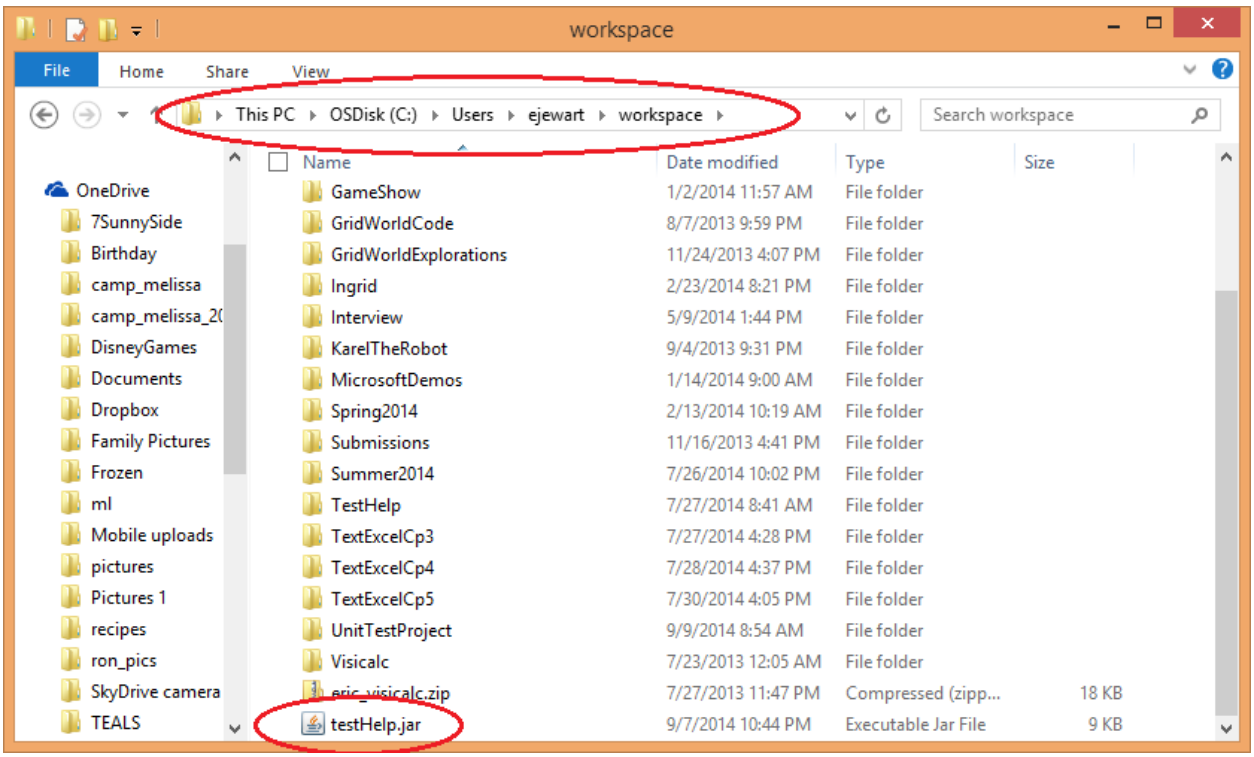
12. In the New JUnit Test Case window that pops up, name your test GizmoTests, then click Finish.



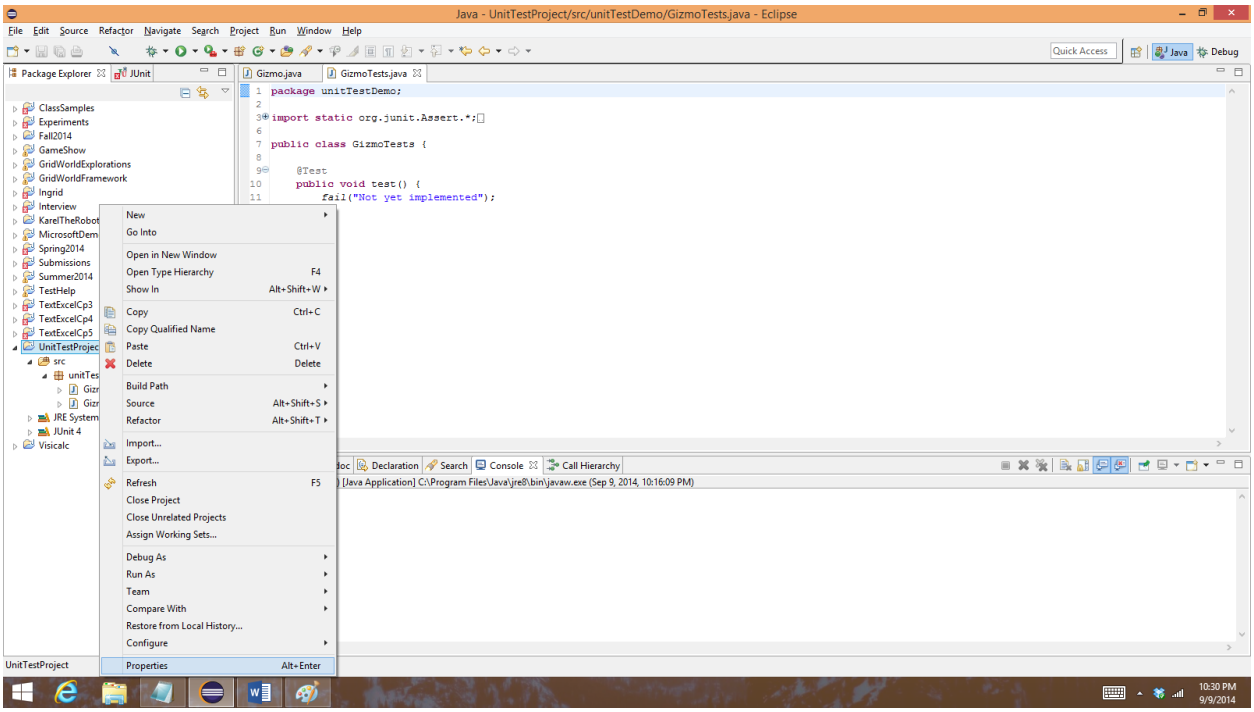
13. Eclipse should pop up another dialog box saying that JUnit is not in the build path and asking if you want to add it. Recall that JUnit is a library of code that helps you write tests. You need it for this to work, so let Eclipse add it to your project by clicking OK.



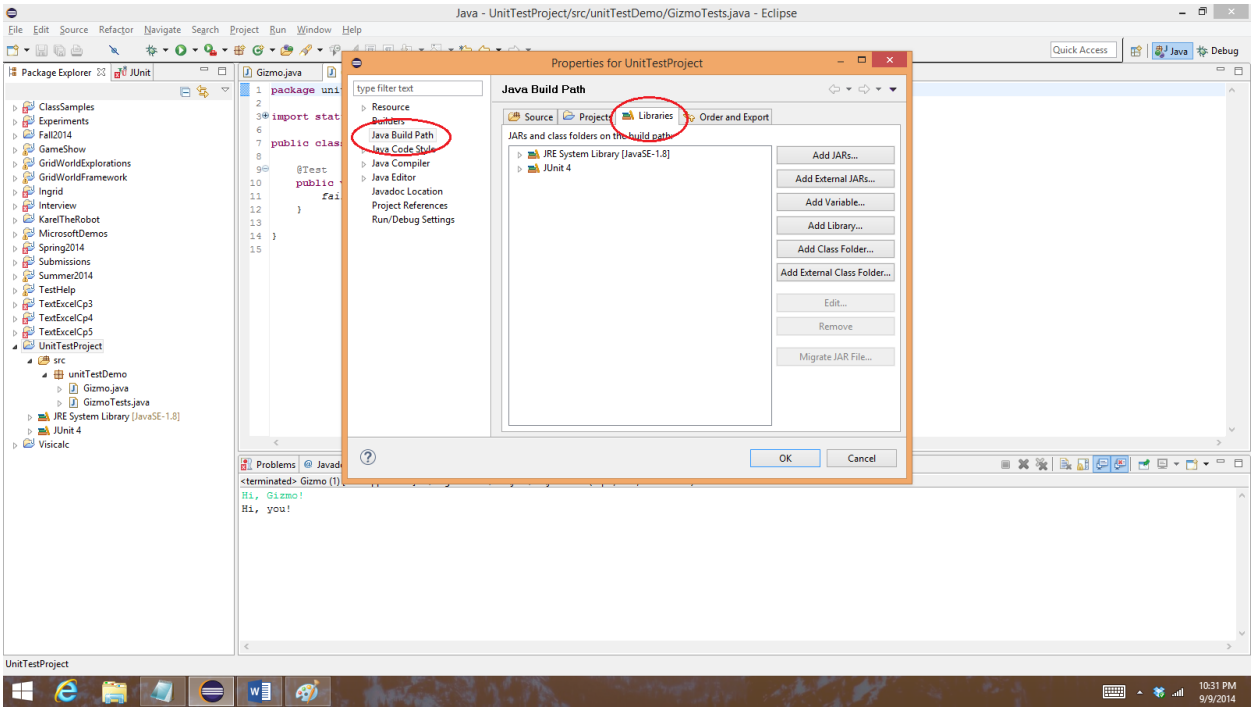
14. There's another library you'll need to add for the unit tests to work. [Download testHelp.jar from moodle] and save it to your workspace. You can just put it in the root of your workspace. Your path might look a bit different—it's probaby under your GoogleDrive folder.



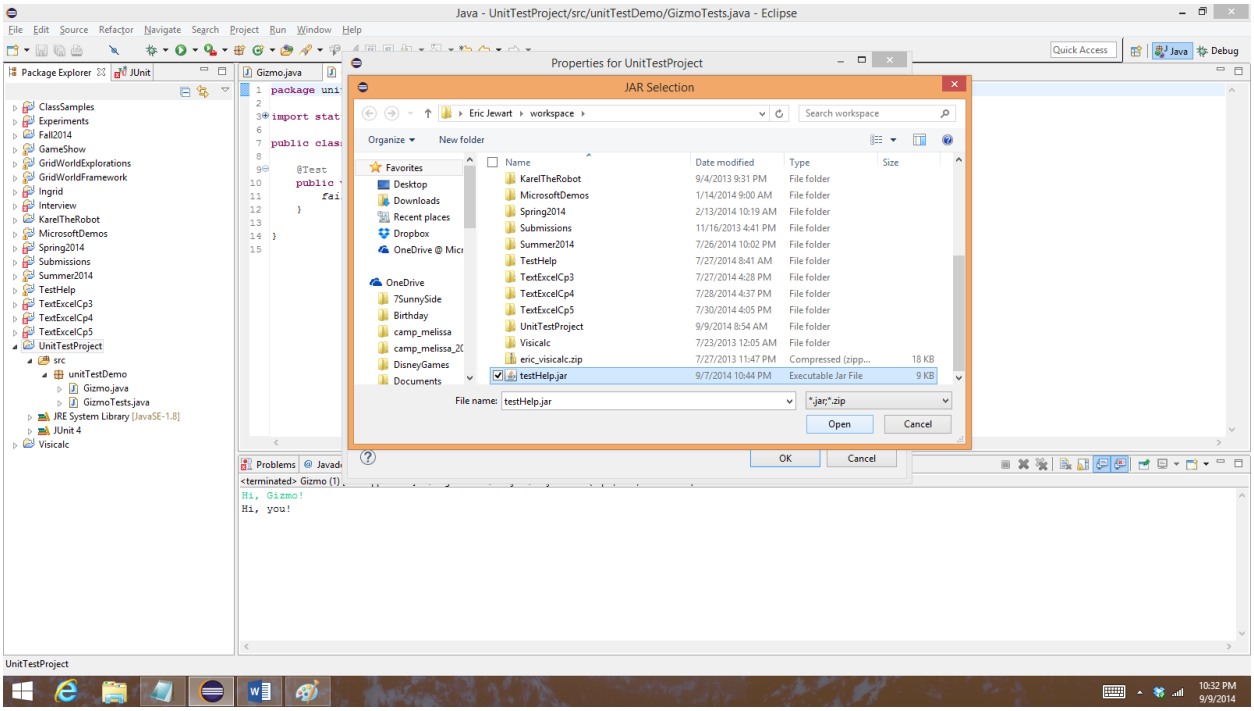
15. Now we need to tell Eclipse to use this library in your UnitTestProject. In Eclipse, right click on UnitTestProject and choose Properties.



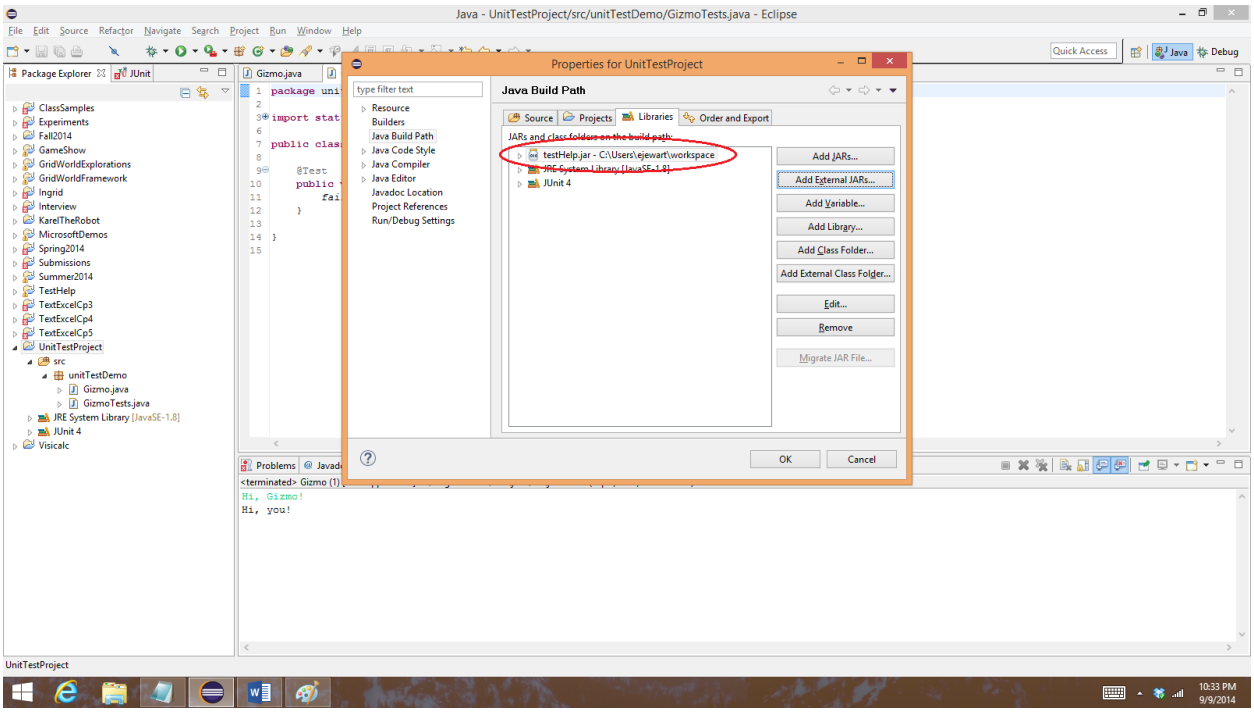
16. In the window that pops up, select Build Path on the left side, then Libraries in the center.



17. Click the Add External Jars button on the right side. In the file dialog that comes up, browse to your workspace folder, select testHelp.jar, and click Open.



18. This should bring you back to the Properties dialog, where you should now see testHelp.jar listed with the other libraries. Click OK in this dialog.

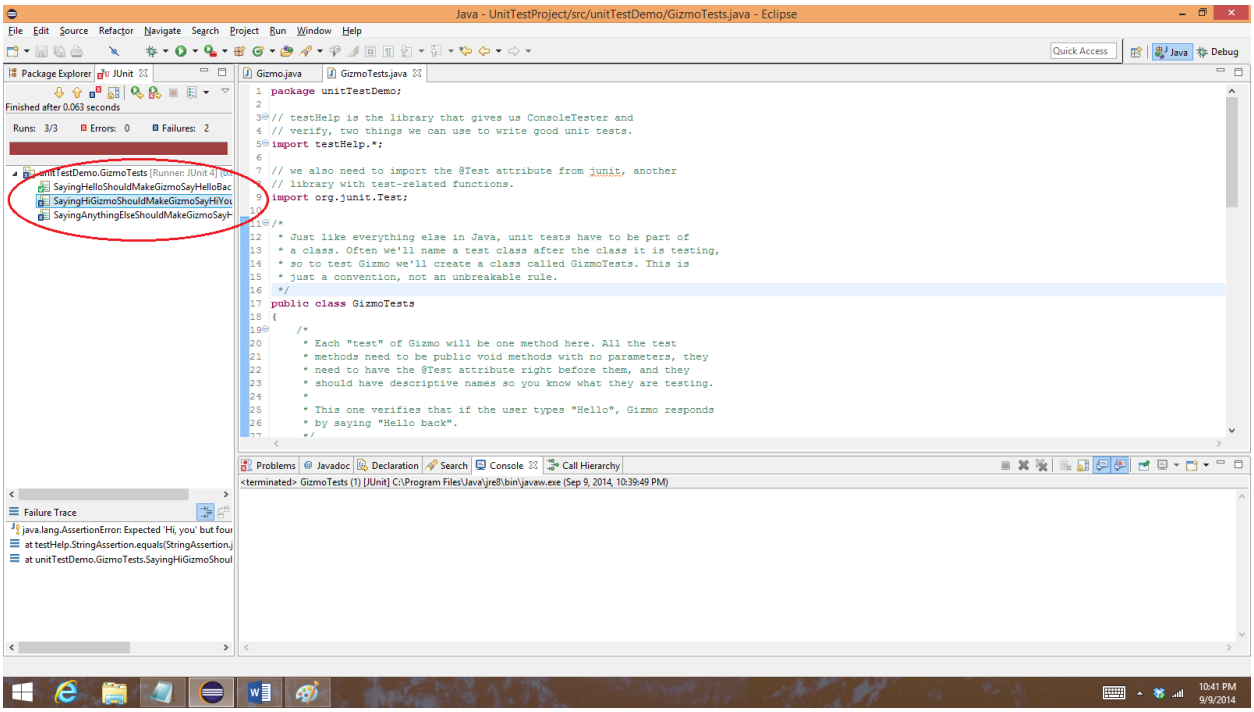


19. For this lab, I've already written some unit tests for you. Copy the following code into the GizmoTests window, replacing everything that is there.

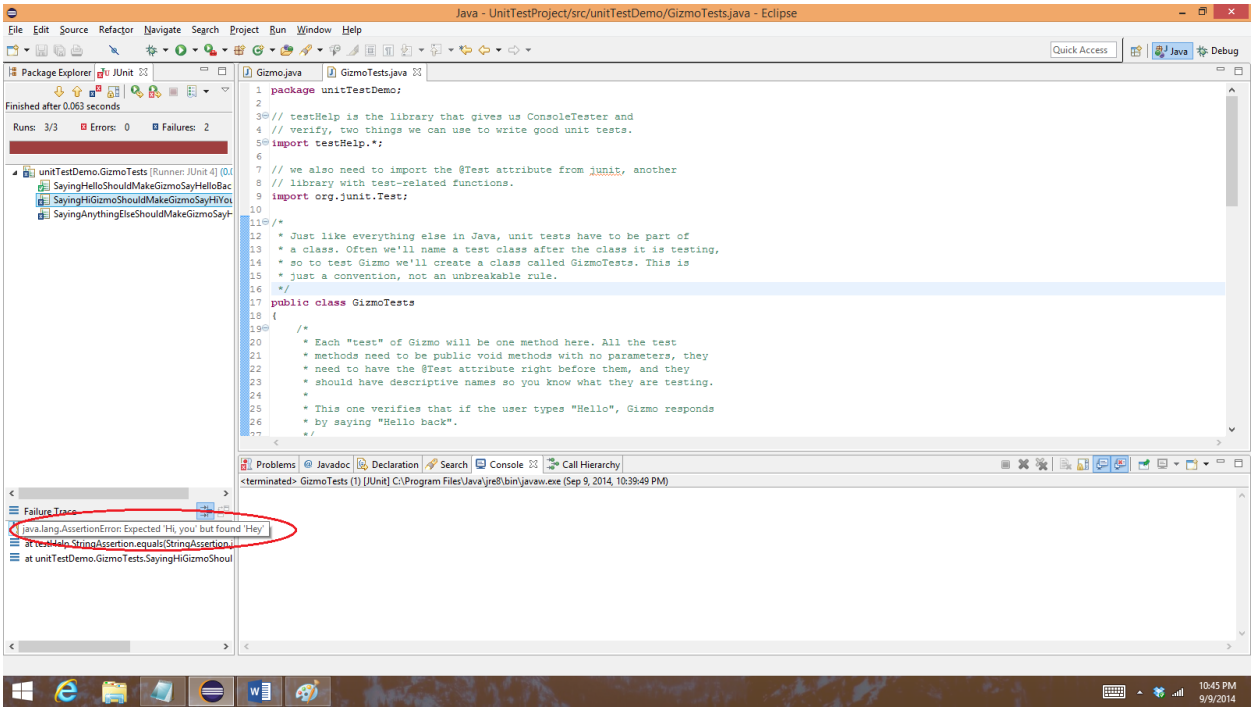
```
package unitTestDemo;

// testHelp is the library that gives us ConsoleTester and
// verify, two things we can use to write good unit tests.
import testHelp.*;

// we also need to import the @Test attribute from junit, another
// library with test-related functions.
import org.junit.Test;

/*
 * Just like everything else in Java, unit tests have to be part of
 * a class. Often we'll name a test class after the class it is testing,
 * so to test Gizmo we'll create a class called GizmoTests. This is
 * just a convention, not an unbreakable rule.
 */
public class GizmoTests
{
	/*
	 * Each "test" of Gizmo will be one method here. All the test
	 * methods need to be public void methods with no parameters, they
	 * need to have the @Test attribute right before them, and they
	 * should have descriptive names so you know what they are testing.
	 *
	 * This one verifies that if the user types "Hello", Gizmo responds
	 * by saying "Hello back".
	 */
	@Test
	public void SayingHelloShouldMakeGizmoSayHelloBack()
	{
		// ConsoleTester.getOuput is a function from the testHelp library
		// that will run the main method of the class you specify, pass
		// in the input you specify as if you had typed it at the
		// console, and return whatever output the program prints. So
		// here, we're running the main method from Gizmo (note that
		// because Gizmo is in a package, we need the fully qualified
		// name, unitTestDemo.Gizmo) and passing in "Hello" as if we had
		// typed it. The output is stored in the variable 'response'.
		String response = ConsoleTester.getOutput("unitTestDemo.Gizmo", "Hello");

		// This test is trying to confirm that if we say 'Hello', Gizmo
		// says 'Hello back'. So, verify that 'response' is "Hello back"
		// by using the syntax below.
		verify.that(response).isEqualTo("Hello back");
	}

	/*
	 * This is a second test method. You can tell because it is a public void
	 * method with no parameters, and (most importantly) because it has @Test
	 * in front of it.
	 *
	 * This one verifies that if you say "Hi, Gizmo!", Gizmo will respond with
	 * "Hi, you!" As provided, it fails. Why? Look for the assertion error in
	 * the "Failure trace" window, usually near the bottom left side of Ecliipse.
```

```
         *
         * Can you fix the failing test? Hint: there are two problems, and both are
         * here in the test code.
         */
        @Test
        public void SayingHiGizmoShouldMakeGizmoSayHiYou()
        {
              String response = ConsoleTester.getOutput("unitTestDemo.Gizmo", "Hi, Gizmo");
              verify.that(response).isEqualTo("Hi, you");
        }

        /*
         * This method verifies that saying something other than Hello or Hi, Gizmo!
         * results in Gizmo's default response, which should be "Hi". It is failing.
         * Why? The problem is in Gizmo.java, NOT HERE, so fix it in that code and
         * re-run these tests to make sure they all pass now.
         */
        @Test
        public void SayingAnythingElseShouldMakeGizmoSayHi()
        {
              String response = ConsoleTester.getOutput("unitTestDemo.Gizmo", "Whatever");
              verify.that(response).isEqualTo("Hi");
        }
}
```

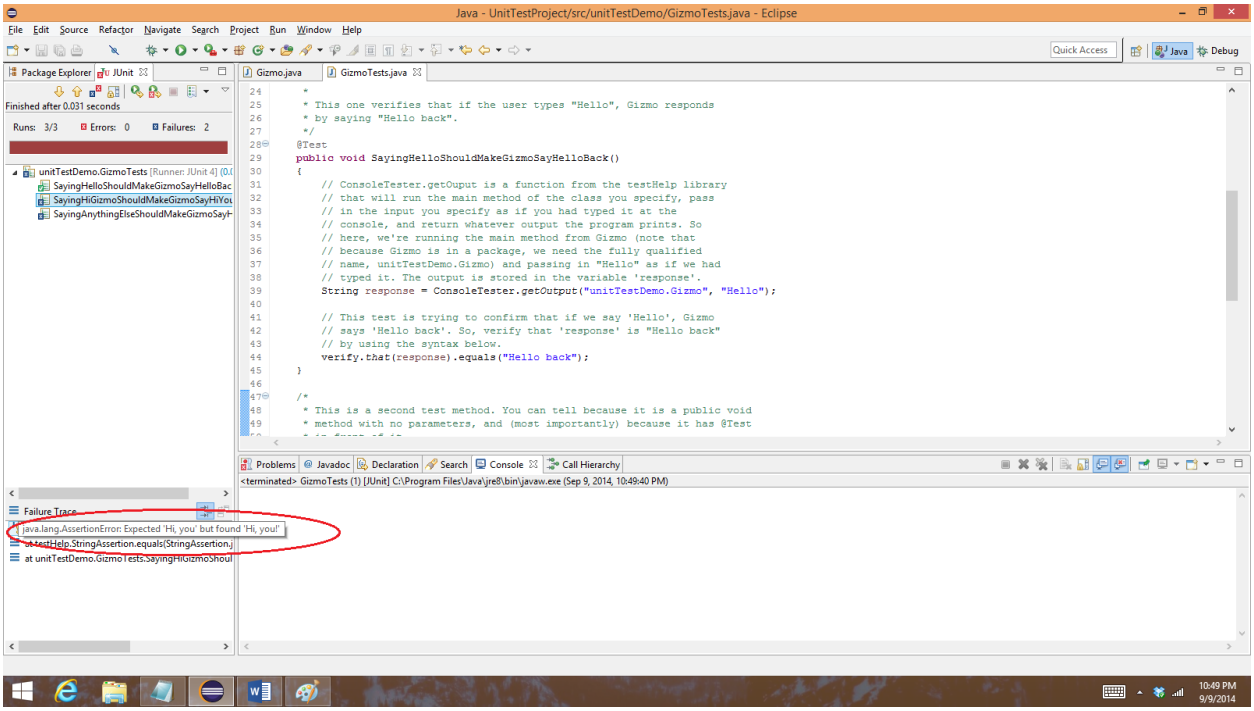
20. Click on the arrow in the green circle again. Since you have the GizmoTests window open, Eclipse will run the three tests from this file rather than running your Gizmo program. You should see that one of the tests passes (it has a tiny green checkmark next to it) and the other two fail (blue x's).



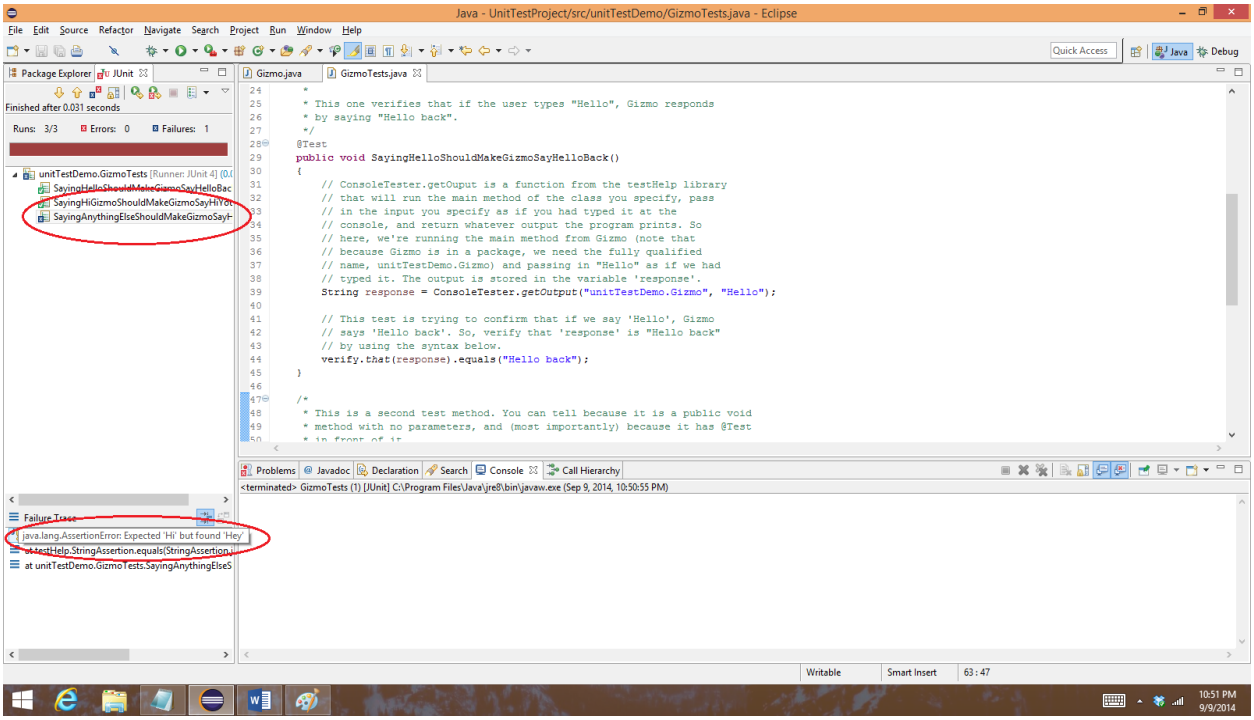
21. Click on the first failing test, the one called SayingHiGizmoShouldMakeGizmoSayHiYou. Now if you look at the Failure Trace window at the bottom left, you can see what went wrong. Hover over the assertion error if you can't see the whole thing.



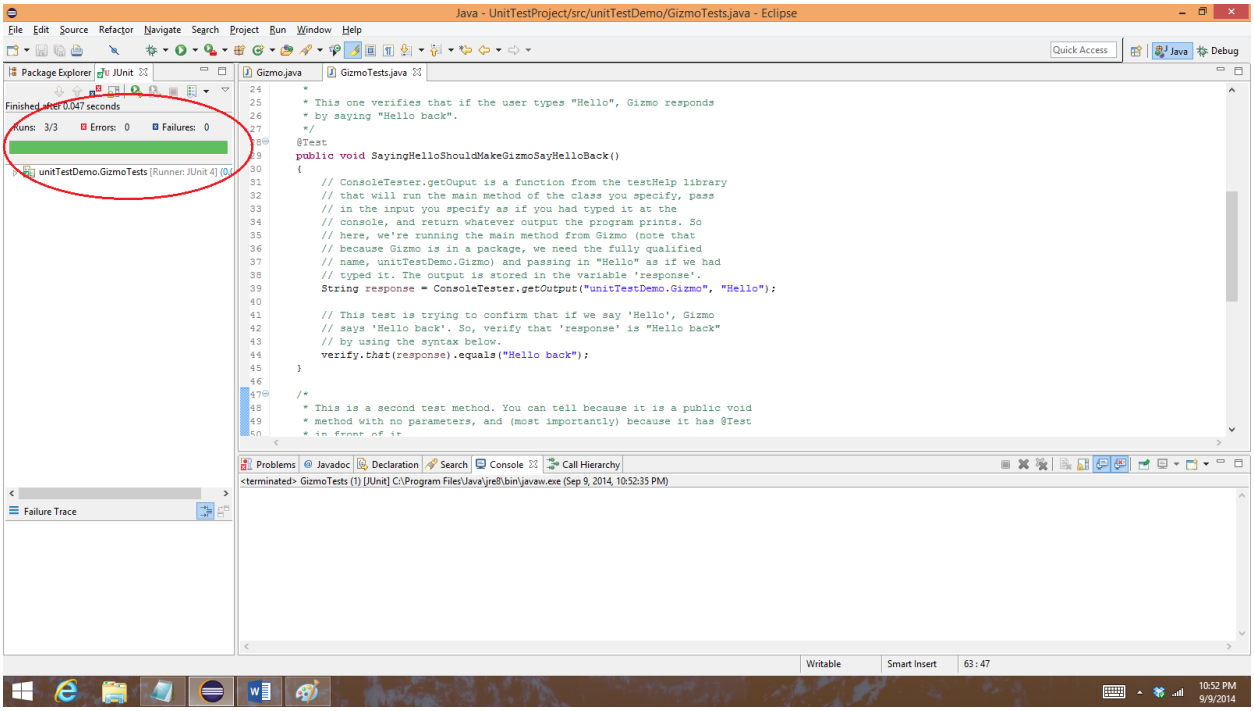
22. Based on the descriptive name of the test and on the message you got, you should be able to figure out what is wrong with this test and fix it. There are also some hints in the comments above the test. Read the comments carefully—they say exactly what the behavior should be. After you fix the first problem, run the tests again to see the next problem.



23. Once you fix both problems in that test, run the tests again. Now you should just see one failing test.



24. This time, the test code is correct and the problem is in Gizmo.java. Switch back to that file and fix the problem the test is pointing out. When you have fixed it, you should see all the tests pass if you run them again.



Now you've set up a new project, written (or at least copied and pasted) some code, set up some unit tests we supplied, and made sure they all pass when they run on your code. We'll provide unit tests like this for some of your future assignments, especially projects, so you can use them to make sure your code is correct before you check in.

For now, you don't have to know how to write your own unit tests, but you can easily add tests to what we supply if you are interested. Many professional developers find that having a good set of automated tests makes it immensely easier to write code.