Part I

Problem 0: Install and set up

Read and complete the [**Getting Started guide**](http://web.mit.edu/6.005/www/fa16/getting-started/). The guide will step through:

* installing the JDK, Eclipse
* configuring Eclipse

You need to complete all the steps in the guide before you start working on this problem set.

Problem 1: Clone and import

Create Java project, and import the codes to your project.

Problem 2: Warm up with may­Use­Code­In­Assignment

1. Look at the source code contained in RulesOf6005.java in package rules. Your warm-up task is to implement:
2. mayUseCodeInAssignment(
3. boolean writtenByYourself, boolean availableToOthers,
4. boolean writtenAsCourseWork, boolean citingYourSource,

boolean implementationRequired)

You can find the policy under [General Information](http://web.mit.edu/6.005/www/fa16/general/) on the [course home page](http://web.mit.edu/6.005/www/fa16/).

1. Once you’ve implemented this method, run the main method in RulesOf6005.java.

public static void main(String[] args) is the entry point for Java programs. In this case, the main method calls the mayUseCodeInAssignment method with input parameters. To run main in RulesOf6005, right click on the file RulesOf6005.java in either your Package Explorer, Project View, or Navigator View, go to the *Run As* option, and click on *Java Application*.

Unit testing

Right now, we can use the main method plus some visual inspection to verify that our implementation is correct. More generally, programs will have many dozens of methods that need to be tested; visually inspecting output for each one is fragile, time-consuming, and inherently non-scalable.

Instead, we will use *automated unit testing*, which runs a suite of tests to automatically test whether the implementations are correct. For this problem set, we will write unit tests for methods that do not draw graphics on the screen; unit-testing GUIs is a more complex problem.

Automated unit testing with JUnit

[JUnit](http://www.junit.org/) is a widely-adopted Java unit testing library, and we will use it heavily in 6.005. A major component of the 6.005 design philosophy is to decompose problems into minimal, orthogonal units, which can be assembled into the larger modules that form the finished program. One benefit of this approach is that each unit can be tested thoroughly, independently of others, so that faults can be quickly isolated and corrected as code is rewritten and modules are configured. Unit testing is the technique of writing tests for the smallest testable pieces of functionality, to allow for the flexible and organic evolution of complex, correct systems.

By writing thoughtful unit tests, it is possible to verify the correctness of one’s code, and to be confident that the resulting programs behave as expected. In 6.005, we will use JUnit version 4.

Anatomy of JUnit

JUnit unit tests are written method by method. There is nothing special a class has to do to be used by JUnit; it only need contain methods that JUnit knows to call, which we call *test methods*. Test methods are specified using *annotations*, which may be thought of as keywords (more specifically, they are a type of metadata), that can be attached to individual methods and classes. Though they do not themselves change the meaning of a Java program, at compile- or run-time other code can detect the annotations and make decisions accordingly. Though we will not deeply explore annotations in 6.005, you will see a few important uses of them.

Look closely at RulesOf6005Test.java, and note the @Test that precedes method definitions. This is an example of an annotation. The JUnit library uses this annotation to determine which methods to call when running unit tests. The @Test annotation denotes a test method; there can be any number in a single class. Even if one test method fails, the others will be run.

Unit test methods can contain calls to assertEquals, which is an assertion that compares two objects against each other and fails if they are not equal, assertTrue, which checks if the condition is true, and assertFalse, which checks if the condition is false. [Here is a list of all the assertions supported by JUnit](http://junit.org/junit4/javadoc/latest/org/junit/Assert.html). If an assertion in a test method fails, that test method returns immediately, and JUnit records a failure for that test.

1. Run the unit tests.

To run the tests in RulesOf6005Test, right click on the RulesOf6005Test.java file in either your Package Explorer, Project View, or Navigator View, and go to the *Run As* option. Click on *JUnit Test*, and you should see the JUnit view appear.

If your implementation of mayUseCodeInAssignment is correct, you should see a green bar, indicating that all the tests (there’s only 1 test, containing 2 assertions) passed.

1. Try *breaking* your implementation and running RulesOf6005Test again.

You should see a red bar in the JUnit view, and if you click on test­May­Use­Code­In­Assignment, you will see a *stack trace* in the bottom box, which provides a brief explanation of what went wrong. Double-clicking on a line in the failure stack trace will bring up the code for that frame in the trace. This is most useful for lines that correspond to your code; this stack trace will also contain lines for Java libraries or JUnit itself.

1. Enough breaking: fix your implementation so it’s correct again. Make sure the tests pass.

Passing the JUnit tests we provide does **not** necessarily mean that your code is perfect. You need to review the function specifications carefully, and **always write your own JUnit tests** to verify your code.

Problem 3: Commit and push（忽略）

Github内容暂时不做要求

Part II

Turtle graphics and the Logo language

[Logo](http://en.wikipedia.org/wiki/Logo_%28programming_language%29) is a programming language created at MIT that originally was used to move a robot around in space. Turtle graphics, added to the Logo language, allows programmers to issue a series of commands to an on-screen “turtle” that moves, drawing a line as it goes. Turtle graphics have also been added to many different programming languages, [including Python](http://docs.python.org/2/library/turtle.html), where it is part of the standard library.

In the rest of problem set 0, we will be playing with a simple version of turtle graphics for Java that contains a restricted subset of the Logo language:

* forward(units)   
  Moves the turtle in the current direction by *units* pixels, where units is an integer. Following the original Logo convention, the turtle starts out facing up.
* turn(degrees)   
  Rotates the turtle by angle *degrees* to the right (clockwise), where degrees is a double precision floating point number.

You can see the definitions of these commands in Turtle.java.

**Do NOT use any turtle commands other than forward and turn in your code for the following methods.**

Problem 4: drawSquare

Look at the source code contained in TurtleSoup.java in package turtle.

Your task is to implement drawSquare(Turtle turtle, int sideLength), using the two methods introduced above: forward and turn.

Once you’ve implemented the method, run the main method in TurtleSoup.java. The main method in this case simply creates a new turtle, calls your drawSquare method, and instructs the turtle to draw. Run the method by going to *Run → Run As… → Java Application*. A window will pop up, and, once you click the “Run!” button, you should see a square drawn on the canvas.

Problems 5—10: Polygons and headings

**For detailed requirements, read the specifications of each function to be implemented above its declaration in TurtleSoup.java. Be careful when dealing with mixed integer and floating point calculations.**

You should not change any of the *method declarations* ([what’s a declaration?](http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html)) below. If you do so, you risk receiving **zero points** on the problem set.

Drawing polygons

* Implement calculateRegularPolygonAngle   
  There’s a simple formula for what the inside angles of a regular polygon should be; try to derive it before googling/binging/duckduckgoing.
* Run the JUnit tests in TurtleSoupTest   
  The method that tests calculateRegularPolygonAngle should now pass and show green instead of red.

If testAssertionsEnabled fails, you did not follow the instructions in the Getting Started guide. [Getting Started step 2](http://web.mit.edu/6.005/www/fa16/getting-started/#config-eclipse) has setup you must perform before using Eclipse.

* Implement calculatePolygonSidesFromAngle   
  This does the inverse of the last function; again, use the simple formula. However, **make sure you correctly round** to the nearest integer. Instead of implementing your own rounding, look at Java’s [java.lang.Math](http://docs.oracle.com/javase/8/docs/api/?java/lang/Math.html) class for the proper function to use.
* Implement drawRegularPolygon   
  Use your implementation of calculateRegularPolygonAngle. To test this, change the main method to call drawRegularPolygon and verify that you see what you expect.

Calculating headings

* Implement calculateHeadingToPoint   
  This function calculates the parameter to turn required to get from a current point to a target point, with the current direction as an additional parameter. For example, if the turtle is at (0,1) facing 30 degrees, and must get to (0,0), it must turn an additional 150 degrees, so calculateHeadingToPoint(30, 0, 1, 0, 0) would return 150.0.
* Implement calculateHeadings   
  Make sure to use your calculateHeadingToPoint implementation here. For information on how to use Java’s List interface and classes implementing it, look up [java.util.List](http://docs.oracle.com/javase/8/docs/api/?java/util/List.html) in the Java library documentation. Note that for a list of *n* points, you will return *n-1* heading adjustments; this list of adjustments could be used to guide the turtle to each point in the list. For example, if the input lists consisted of xCoords=[0,0,1,1] and yCoords=[1,0,0,1] (representing points (0,1), (0,0), (1,0), and (1,1)), the returned list would consist of [180.0, 270.0, 270.0].

Problem 11: Personal art

* Implement drawPersonalArt   
  In this function, you have the freedom to draw any piece of art you wish. Your work will be judged both on aesthetics and on the code used to draw it. Your art doesn’t need to be complex, but it should be more than a few lines. Use helper methods, loops, etc. rather than simply listing forward and turn commands.

**For drawPersonalArt only, you may also use the color method of Turtle to change the pen color.** You may only use the provided colors.

[Here are some examples](https://www.google.com/search?q=python+turtle+example+images) of the kinds of images you can generate procedurally with turtle graphics, though note that many of them use more commands than what we’ve provided here. Modify the main method to see the results of your function.

可参考原网页：http://web.mit.edu/6.005/www/fa16/psets/ps0/