You can find more resources for this course at: <http://www.dukelearntoprogram.com/course2/index.php>

This website of programming resources will be essential to your success in this course. The link above for this course is where you will go to:

* Download our custom version of the BlueJ environment;
* Find project resources, such as example code from the lecture videos;
* Download images and data files for the programming exercises; and
* See documentation for the edu.duke library, and other Javadoc.

**Course Programming Resources**

The **BlueJ Environment** download, documentation, and programming resources are available at:

<http://www.dukelearntoprogram.com/course2/index.php>

(Note that if you have already downloaded a version of BlueJ, you will need to uninstall and download this one to get the edu.duke library.)

BlueJ has provided a [playlist of videos](https://www.youtube.com/watch?v=CPUaTT0Xoo4&list=PLYPWr4ErjcnzWB95MVvlKArO6PIfv1fHd) explaining some key Java terminology and how to use BlueJ.

**Download BlueJ**

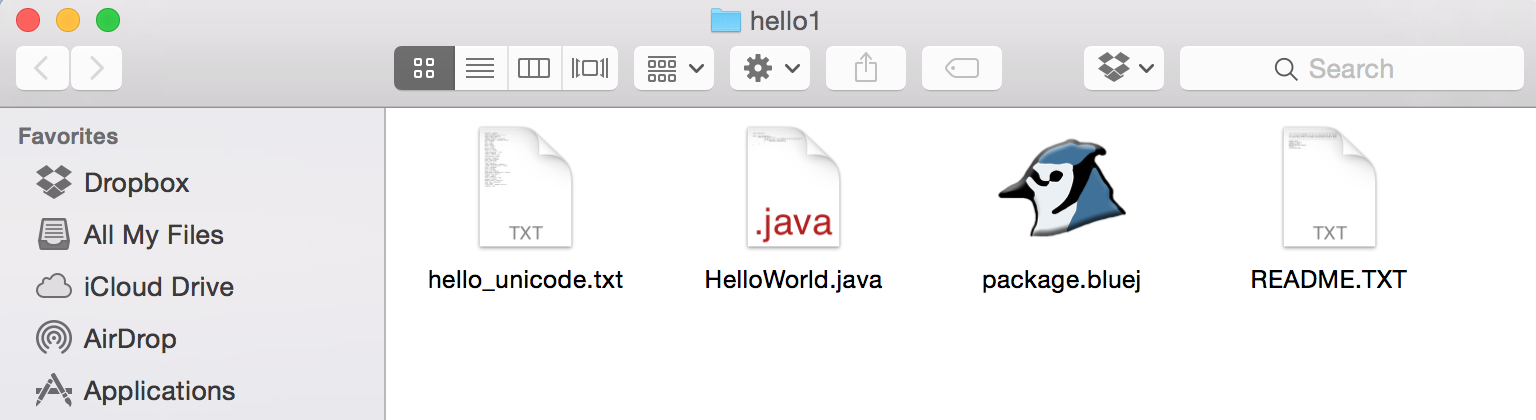
From the above linked DukeLearnToProgram page, click on "Download BlueJ Environment" (<http://www.dukelearntoprogram.com/downloads/bluej.php>). Follow the relevant instructions for your device's operating system (Windows, Mac, or Linux). Note that on a Mac, you may want to move the BlueJ application to your "Applications" folder.

Also, for Mac users, if you get an alert when you try to open BlueJ for the first time that says, "BlueJ” can’t be opened because the identity of the developer cannot be confirmed," simply two-finger or right click on the BlueJ application icon and click "Open". You will still receive the same alert message, but now there will be a button in the alert that will allow you to open BlueJ. BlueJ should open normally without the alert after you open the application this first time.

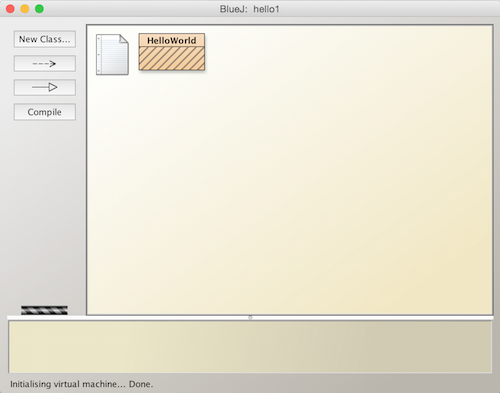
If you intend to follow along with the *"Hello!" around the World* example showcased in the next video, download the project containing the example code from the DukeLearnToProgram Project Resources (<http://www.dukelearntoprogram.com/course2/files.php>, download **"BlueJ Project: HelloWorld class using FileResource**").

**Opening BlueJ Project in Mac**

The HelloWorld project contains these files in a zip folder:

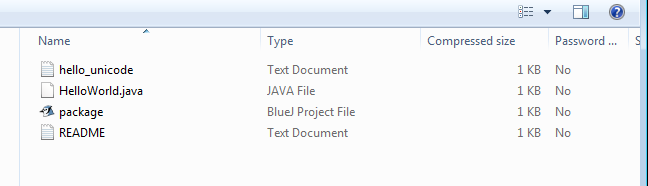


After downloading those files, open the BlueJ application, and from the menu bar, select Project -> Open Project. Select the project folder you have just downloaded, which should now have a BlueJ logo on the folder icon. Click "Open," and you are ready to compile and run the example program along with the next videos!

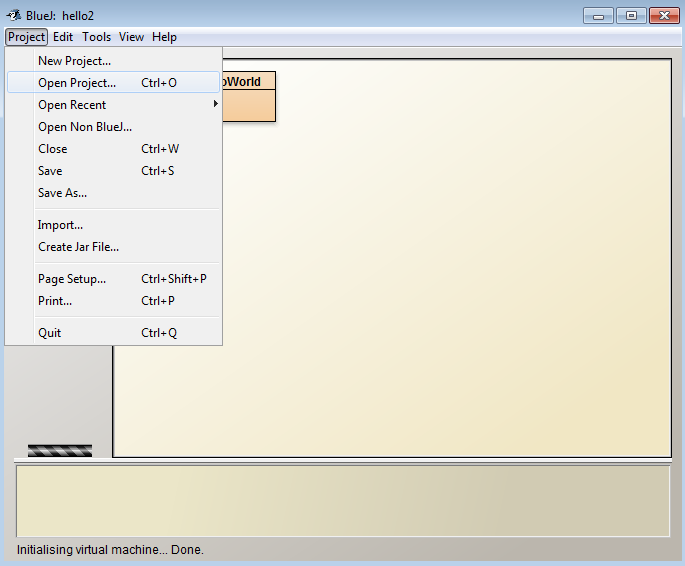


**Opening BlueJ Project in Windows**

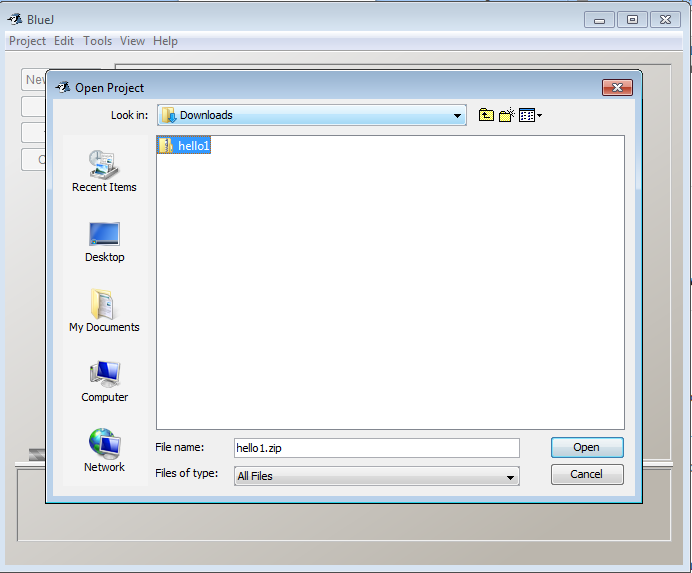
The HelloWorld project contains these files in a zip folder:



After downloading the zip folder, open the BlueJ application, and from the menu bar, select Project -> Open Project.



Select the zip folder you have just downloaded. Click "Open," and you are ready to compile and run the example program along with the next videos!



# **Perimeter Assignment Introduction/Code Review**

[**Practice with Lab Sandbox**](https://www.coursera.org/learn/java-programming/lab-sandbox)

In the next few readings, you're going to use what you've learned in the past few lessons to create a working Java program that can calculate the perimeter of a shape, from some given points. Before we get started with that, however, we thought it would be best to go through the sample code that you'll be using as the foundation for your Java Program!

# Sample Code Explanation and First Steps

**Note: You must have downloaded BlueJ on or after September 7, 2017, from** [**http://www.dukelearntoprogram.com/**](http://www.dukelearntoprogram.com/) **in order to access the Shape and Point Java classes used in the following programming assignments.**

In these exercises, you will use the Shape and Point classes to answer questions about a Shape that is made up of a collection of points from the x-y plane, as shown in this lesson. The shape is defined by drawing a line between two adjacent points, for every pair of adjacent points, and also a line between the first and last point. Be sure to consult the documentation on DukeLearnToProgram to understand how the Shape, Point, DirectoryResource and FileResource classes work:<http://www.dukelearntoprogram.com/course2/doc/javadoc/index.html?course=2>[.](http://www.dukelearntoprogram.com/course2/doc/)

## Calculating information about shapes

In this assignment, you will complete the PerimeterAssignmentRunner class to calculate lots of interesting facts about shapes. This class has been started for you in the BlueJ project called PerimeterAssignmentRunner (go to:<http://www.dukelearntoprogram.com/course2/files.php> and download the **Quiz - Calculating the Perimeter of a Shape** BlueJ project). This project also contains several data files. In addition, you will need to look at the documentation for the Shape class and the Point class.

The PerimeterAssignmentRunner class already includes the following complete methods:

1. The getPerimeter method has one parameter s of type Shape. Given a shape, this method returns the perimeter of the shape.
2. The testPerimeter method has no return value, hence its return type is void. This method is used to select a data file by using the FileResource class, create a shape based on the points from that data file, and then calculate the perimeter of the shape and output its value.
3. The triangle method has no return value and creates a triangle that you can use to test the methods you will create in this assignment.
4. The printFileNames method, which we will discuss in a future lesson.
5. The main method.

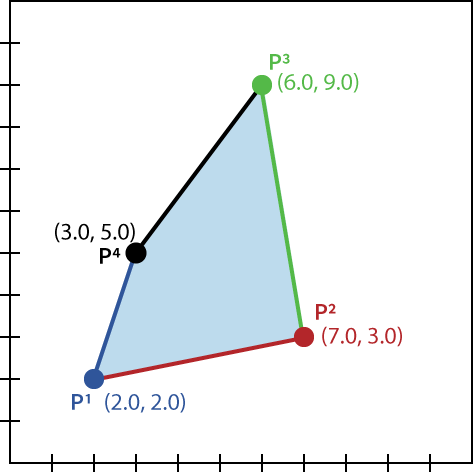
For this assignment, you will add or modify several methods in the PerimeterAssignmentRunner class.

## A few ground rules for Shapes

For the sake of familiarizing ourselves with the abstract concept of a Shape, we're going to take a look at an example. We can think of a shape as a collection of points that are connected together by lines, so for our example, we're going to use a shape with four random points. Here is an example Shape that has 4 points in this order: (2.0, 2.0) (7.0, 3.0) (6.0, 9.0) (3.0, 5.0)

By defining the shape with the points in this order, that means the shape has these lines: a line from (2.0, 2.0) to (7.0, 3.0), a line from (7.0, 3.0) to (6.0, 9.0), a line from (6.0, 9.0) to (3.0, 5.0), and finally, there is a line from the last point (3.0, 5.0) to the first point (2.0, 2.0).

Here's what our shape would look like:



## An Explanation of the complete getPerimeter method:

Here is the idea of how the getPerimeter method works with our example. Remember the Shape is defined as these four points: (2.0, 2.0) (7.0, 3.0) (6.0, 9.0) (3.0, 5.0) To calculate the perimeter, we would loop over all the points in Shape, and for each point calculate the length of the line formed with the previous point. We're just using the distance formula for these calculations, but we'll actually be able to get our program to calculate that for us in the code without having to remember the distance formula itself. Starting with the first point (2.0, 2.0) we would need the last point in the Shape with it, (3.0, 5.0) and calculate its length as 3.16 . For point (7.0, 3.0), calculate its length with (2.0, 2.0) which is 5.09. For point (6.0, 9.0), calculate its length with (7.0, 3.0) which is 6.08. For point (3.0, 5.0), calculate its length with (6.0, 9.0) which is 5.

We return the sum of these, which is 3.16 + 5.09 + 6.08 + 5, which is 19.33.

Now let's see how the code for getPerimeter implements this idea. The getPerimeter method reads as follows:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

public double getPerimeter (Shape s) {

        // Start with totalPerim = 0

        double totalPerim = 0.0;

        // Start wth prevPt = the last point

        Point prevPt = s.getLastPoint();

        // For each point currPt in the shape,

        for (Point currPt : s.getPoints()) {

            // Find distance from prevPt point to currPt

            double currDist = prevPt.distance(currPt);

            // Update totalPerim by currDist

            totalPerim = totalPerim + currDist;

            // Update prevPt to be currPt

            prevPt = currPt;

        }

        // totalPerim is the answer

        return totalPerim;

}





So immediately, we see some other classes and methods being called here. The “Point” class is something that is being used here and it was loaded in the duke **package** at the top of the project, with the line import edu.duke\*; . To find out more about this class, you can reference [the documentation on the Duke site](https://www.dukelearntoprogram.com/course2/doc/javadoc/index.html?course=2). Within that page, you’ll have to select the edu.duke package, and then select the “Point” class. This documentation shows that this class can make use of the “distance” method, which outputs a **double**-type value. You will also see the “distance” method being called in the above “getPerimeter” method. To run through the code sample above, we proceed as follows:

1. This method “getPerimeter” first has a parameter, **s**, which is of the type Shape.
2. We then create a double-type variable named totalPerim and initialize it as 0. This variable needs to be double-type, as the distance between two points is not always an integer, and subsequently the perimeter will not always be an integer.
3. Next, we create a new Point object named prevPt. This object is initialized as the output of the getLastPoint method from the object **s**. As it is listed in the documentation for the Shape Class, getLastPoint is a Point-type method that returns to the last point added to a selected object/shape (checking the documentation on all these methods is essential).
4. We next enter a “for each” loop, which initializes the Point-type variable currPt for each entry in the **iterable** getPoints. In the documentation for the Shape class, we see that the iterable getPoints allows access to a shape one point at a time.
5. Next, we create a double-type variable currDist, and initialize it by referencing another method, distance. This variable will be initialized as the distance from prevPt to currPt.
6. We then update the variable totalPerim (which was previously initialized at 0.0) to be equal to totalPerim + currDist. We do this, so that we can keep a running total. Each time this loop runs (which, as an iterable of the getPoints method, will be as many times as there are points in the Shape Object, **s**) totalPerim will be with the currDist between the next two points.
7. When the loop iterates through every point in **s**, we exit, and return the double-type variable totalPerim as the output of the “getPerimeter” Method

## An Explanation of the complete testPerimeter and main Methods:

1

2

3

4

5

6

public void testPerimeter () {

        FileResource fr = new FileResource();

        Shape s = new Shape(fr);

        double length = getPerimeter(s);

        System.out.println("perimeter = " + length);

}





First of all, we see that the testPerimeter method has no type, and so it gets the designation void. Proceeding from there, the code proceeds as follows

1. We initialize a FileResource object named fr, using the **new** keyword and the FileResource class (this was imported with the edu.duke package).
2. From the duke documentation on the FileResource class, we are shown that when called without a parameter, the FileResource class will open a dialogue box and allow us to select a file. This file will be stored as the variable fr.
3. Next, we initialize a Shape-type object named **s** (this is the object the code above referenced so many times), as a new object and call the Shape class, using the variable fr (the FileResource Object we just created) as the parameter for this shape object. This means that we will use the contents of fr (the file we selected in a dialogue box) to create a shape object.
4. We create a double-type variable named length, and initialize it as the output of the getPerimeter method, with the parameter of s (the shape object we just created). This will pass the Shape **s** to the getPerimeter method (detailed above), and initialize the variable length as the output of that method. If you recall, getPerimeter accepts a shape-type object as a parameter, and outputs a double-type variable.
5. We then issue a system  output command to print the string “perimeter = “ and the variable “length”.

And for the final bit of this sample file code that we’re examining in this exercise, the main method. This is the area that our program will execute by default after we compile and run it. For that reason, it’s important to take a look at anything in the main method.

1

2

3

4

public static void main (String[] args) {

      PerimeterAssignmentRunner pr = new PerimeterAssignmentRunner();

      pr.testPerimeter();

    }





Reading through this code we see that it simply creates a **new** PerimeterAssignmentRunner-type object, pr, and then calls the method testPerimeter using the object pr, which we outlined above.

## Running through the program as it is currently written

So as a brief summary, if we were to compile and run this program, it would progress as follows:

1. Execution starts in the main method, creates a new PerimeterAssignmentRunner-type object
2. Using the new PerimeterAssignmentRunner-type object, call the method testPerimeter
3. The testPerimeter method creates a new FileResource object, fr,  prompting us to select a file via a dialogue box
4. We use this FileResource object to create a new Shape object, s.
5. Create a variable called length, and initialize it to be “the result of calling the getPerimeter method on s”, the shape object we just created.
6. The getPerimeter method creates a new double-type variable called totalPerim, and creates/initializes the point-type variable prevPt by calling the getLastPoint method on our shape, s.
7. We enter a “for each” loop by calling the getPoints method on our shape, s, and initializing the variable currPt to to be the current point in the iterable. This will loop for as many points as there are in shape s
8. We proceed through the “for each” loop, adding the distance between the last two points (currDist) to totalPerim as we go, and updating currPt at the end of each loop.
9. When all the loops are done, getPerimeter returns the variable totalPerim to the testPerimeter method, thereby initializing the variable length.
10. testPerimeter then uses the system output command to print a string, and the variable length, finishing the testPerimeter method and returning to the main method.
11. The main method finishes executing and terminates.

# Conclusion

There’s a great deal of information to encode here, but there’s a few major takeaways in terms of programming ideas that are very important.

1. We were able to use a “for each” loop with an iterable method to easily loop through every entry in a text file (this is very important and you’ll be asked to do this on your own in future activities).
2. We initialized a variable outside of a loop and then updated its value throughout a loop to keep track of a running tally (we used this to keep track of a total perimeter of a shape, but it could just as easily be used to track the number of iterations throughout a loop, or a number of other functions).
3. We saw how you can use multiple methods and classes that are loaded into a project through an import , and how to call methods inside and outside of the main method.

We definitely won’t go through future example code in this level of detail, but we felt it was necessary to further explain the example files given, and to jumpstart your imagination about how you might think about writing code on your own. In the next few exercises, you’ll need to utilize everything you’ve learned in the lecture videos, and what you’ve experienced firsthand in this example code. When you feel comfortable with the information in this reading, proceed on to the next activity!

# **Perimeter Assignment: Part One**

[**Practice with Lab Sandbox**](https://www.coursera.org/learn/java-programming/lab-sandbox)

# Introduction

In this assignment, you will complete the PerimeterAssignmentRunner class to calculate lots of interesting facts about shapes. This class has been started for you in the BlueJ project called PerimeterAssignmentRunner (**this is the same project file that we were looking at in the previous reading, so feel free to open the one you downloaded for the last reading**). This project also contains several data files. In addition, you will need to look at the documentation for the Shape class and the Point class, which we went over in the last reading.

Our goals for this exercise are to:

**1a. Complete writing the method getNumPoints** that has one parameter s that is of type Shape. This method returns an integer that is the number of points in Shape s.

**1b. Add code in the method testPerimeter** to call getNumPoints and to print the result.

**2a. Complete writing the method getAverageLength** that has one parameter s that is of type Shape. This method returns a number of type double that is the calculated average of all the sides’ lengths in the Shape S.

**2b. Add code in the method testPerimeter** to call the method getAverageLength and to print out the result.

## Discussion

### Complete the method getNumPoints

For this section, we need to complete the code for the method getNumPoints. We are instructed that the method has one parameter, s, that is a Shape-type object. This makes sense with the other code we’ve seen in this project, as the only shape-type object we’ve created is, in fact, named s. To understand how to show what parameter a method has, check out the code review we did in the last reading. (**Hint:** the getPerimeter method also has a parameter named s that is of type shape, but the getPerimeter method returns a double-type variable, whereas we want getNumPoints to return an int-type variable).

After passing the parameter along, we’ll need to write some language that will allow for this method to return an int that is the number of points in Shape s. This immediately tells us that at some point, we will need to declare an int-type variable to return at the end of this method (review the video about variables and types if you’re having trouble with variable declaration). This variable will be a counter for the number of points within a given shape, s, which means that it will need to be increased for each point in the shape, s. This tells us quite a bit about how we might translate that action into java syntax, namely that we’ll be using some sort of loop. For more information about writing code that iterates over an entire object, see the video “Seven Steps in Action: Translating to Code” and look into the code review for this exercise (Hint: the getPerimeter method has both a for each loop, and a variable that updates after each iteration).

### Add Code in the Method testPerimeter to output the result

In this case, we’re simply looking to write some code that will output the result of the getNumPoints method. This tells us that within the testPerimeter method we’ll need to add a variable to correspond to the output of the getNumPoints method. The prompt told us that this method is supposed to output an int-type variable, which means that whatever variable we declare here will also need to be int-type. We’ll then need to initialize it to the output of the getNumPoints method for the shape s. After initialization, we’ll just need to add some code that will cause the system to output some text and our variable (**Hint:** the testPerimeter method already has examples of all of these processes).

At this point, it’s worth compiling your code and running it on some of the example text files included with the BlueJ project file. For the sake of this example, we’ll only use the example1.txt. When you run your program on example1.txt, it should tell you that the shape has four points. And if we open the text file itself, we can see that the shape does indeed have four points, so that checks out. You should run it on the other text files as well, to verify that it does check for the number of points in the shape, and does not just always output 4 points.

### Complete the method getAverageLength

Before we can think about what code we would need to write to figure out the average side length for the shape s, we need to understand exactly what we’re trying to calculate, and then translate that to java syntax. In this case, a couple of things that help us, we know that the way to calculate the average side length is the (sum of all sides lengths) / (number of sides). Additionally, we know that the number of sides is equal to the number of points in a two dimensional shape. We already know how to code a method that can calculate the perimeter of a shape, and we know how to code a method that can count the number of points in a shape, so now it’s just a matter of writing a method that can do both and return a double-type variable that is the quotient of those two quantities.

One thing to keep in mind, however, is **types**. It’s important to remember that when you’ll be doing operations on more than one variable, you may need to make sure they’re the same type, or that you’re casting your variables correctly for the operation you’re attempting. For more information on types and casting, see the “Types” video. For example, the getPerimeter method uses double-type variables, and the getNumPoints method uses int-type variables.

### Add Code to testPerim to output the result of getAverageLength

You’ll do this almost exactly the same you did for the getNumPoints method.

At this point, it’s worth testing your program again, and when you compile and run it on example1.txt, it should tell you that “the average length of a side in your shape is 4.0”.

This is where we’ll leave off with this exercise, but you’ll want to **make sure you save all of your project files**, as we’ll be using the exact same project files for the next few activities.