Name:_____ Date: ___/___/___   Period: _____

Objective: The student will be able to write a simple program to perform basic arithmetic operations.
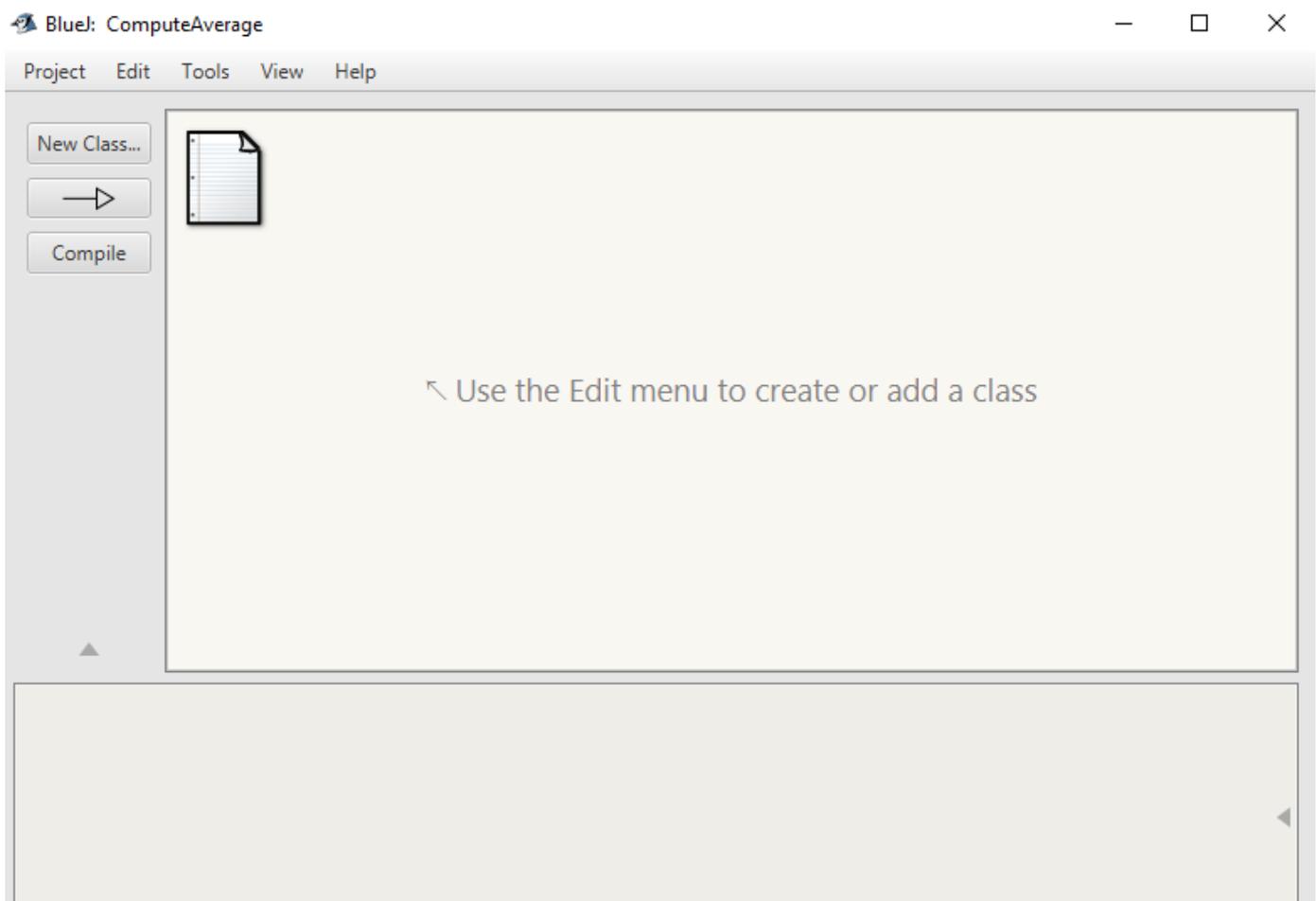
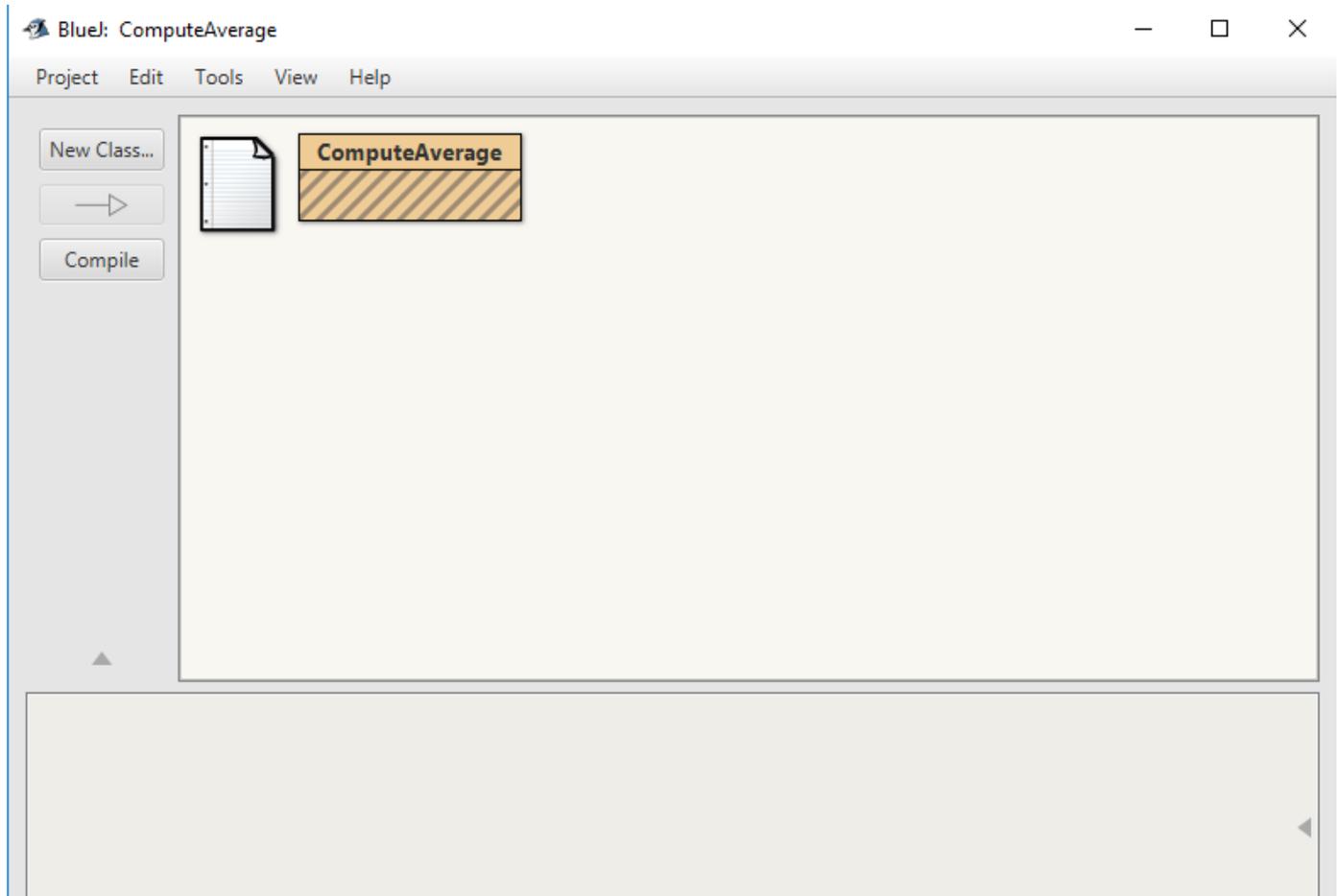Directions: Perform the following steps.

Step 1: Start BlueJ.

Step 2: Click on New Project.

Step 3: Enter a Project Name. In this case ComputeAverage. The project name and the class name will be the same so the first letter should be a capital letter.

Step 4: Click on New Class and enter ComputeAverage.

BlueJ: ComputeAverage   — ☐ ✕

Project   Edit   Tools   View   Help

New Class...

⟶

Compile

↖ Use the Edit menu to create or add a class

Step 5: Double click on the ComputeAverage box to start the editor.

Step 6: Press Control-A to select all text in the editor screen. Press the Backspace key to remove this text.

Step7: Enter the program code shown. When declaring public class ComputeAverage do not press the backspace key after entering the closing brace "}", this will lock the editor. Once you put in the closing brace and hit the enter key you can go back and edit any errors without any problems. (It is a program bug with BlueJ).

public class ComputeAverage

{

   /* Program to Compute the Average of a Sequence of Numbers

   *

   * This program reads a sequence of positive integers input

   * by the user, and it will print out the average of those

   * integers. The user is prompted to enter one integer at a

   * time. The user must enter 0 to mark the end of the data.

```
 */

// Mr. Ellsworth    Period: 0   November 11, 2017

public static void main(String[] args)
{
   // Declare Variables

   int inputNumber = 0;         // One of the integers input by the user.
   int sum = 0;                 // The sum of the positive integers.
   int count = 0;               // The number of positive integers.
   double average;              // The average of the positive integers.

   // Input Section

   System.out.print("Enter your first positive integer: ");
   inputNumber = TextIO.getlnInt();

   while (inputNumber != 0)
   {
      sum = sum + inputNumber;
      count = count + 1;
      System.out.print("Enter your next positive integer, or 0 to end: ");
      inputNumber = TextIO.getlnInt();

   } // end of while loop

   // Output Section

   if (count == 0)
   {
      System.out.println("You didn't enter any data!");
```
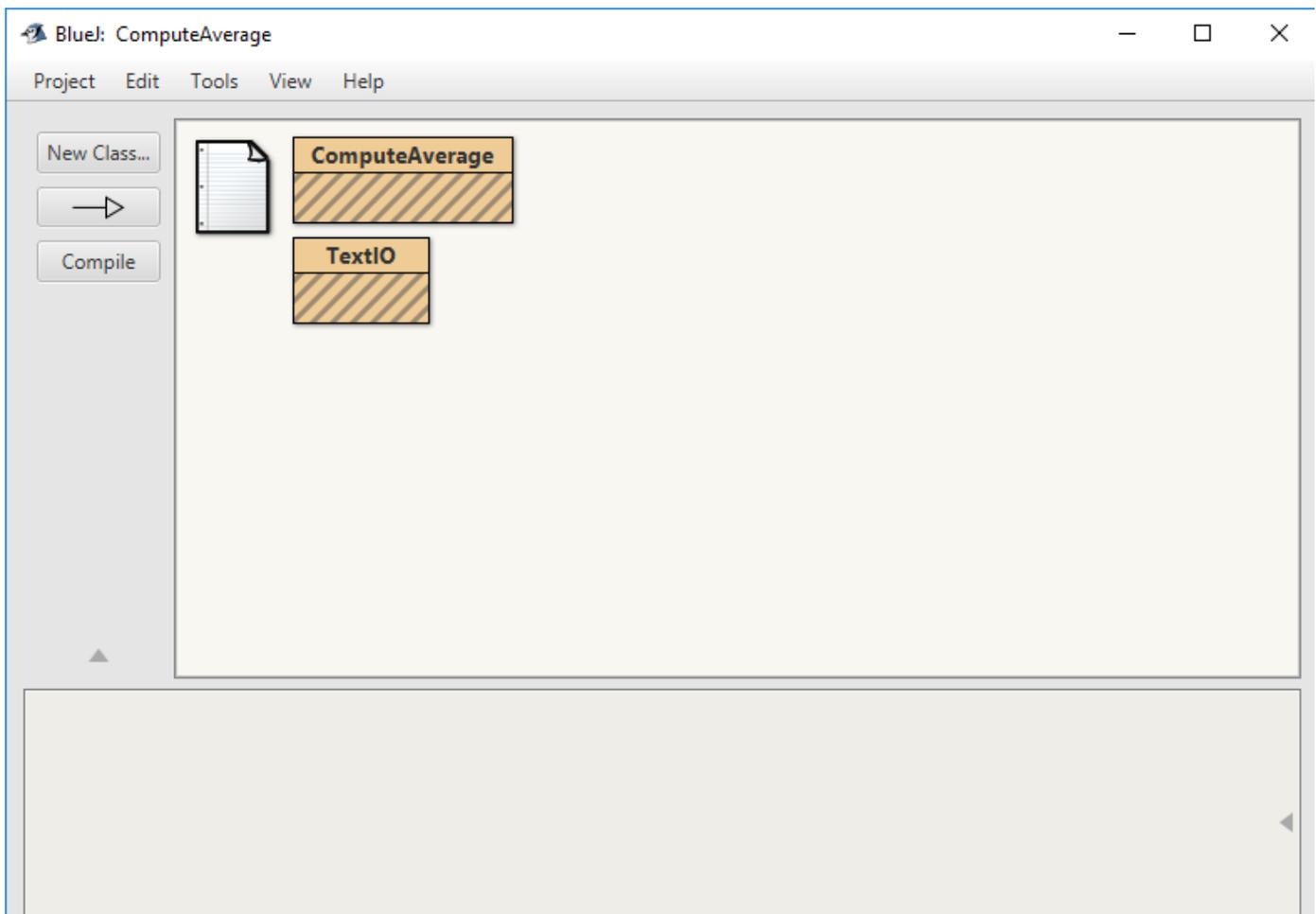
```
        }
        else
        {
            average = ((double)sum) / count;
            System.out.println();
            System.out.println("You entered " + count + " positive integers.");
            System.out.printf("Their average is %1.3f. \n", average);
        }
    } // end of main
} // end of class ComputeAverage
```

Step 8: Add a new class to your project named TextIO.

Step 9: Add the following code to the TextIO class. It is a lot of code so if you can copy this class from another project.

```java
import java.io.*;
import java.util.IllegalFormatException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;


import javax.swing.JFileChooser;
import javax.swing.JOptionPane;


//* TextIO provides a set of static methods for reading and writing text.
public class TextIO
{
   /**
    * The value returned by the peek() method when the input is at end-of-file.
    * (The value of this constant is (char)0xFFFF.)
    */
   public final static char EOF = (char)0xFFFF;


   /**
    * The value returned by the peek() method when the input is at end-of-line.
    * The value of this constant is the character '\n'.
    */
   public final static char EOLN = '\n';          // The value returned by peek() when at end-of-line.



   /**
    * After this method is called, input will be read from standard input (as it
    * is in the default state).  If a file or stream was previously the input source, that file
    * or stream is closed.
    */
```

```java
public static void readStandardInput() {
   if (readingStandardInput)
      return;
   try {
      in.close();
   }
   catch (Exception e) {
   }
   emptyBuffer();  // Added November 2007
   in = standardInput;
   inputFileName = null;
   readingStandardInput = true;
   inputErrorCount = 0;
}


/**
 * After this method is called, input will be read from inputStream, provided it
 * is non-null.  If inputStream is null, then this method has the same effect
 * as calling readStandardInput(); that is, future input will come from the
 * standard input stream.
 */
public static void readStream(InputStream inputStream) {
   if (inputStream == null)
      readStandardInput();
   else
      readStream(new InputStreamReader(inputStream));
}


/**
 * After this method is called, input will be read from inputStream, provided it
 * is non-null.  If inputStream is null, then this method has the same effect
 * as calling readStandardInput(); that is, future input will come from the
```

```
 * standard input stream.
 */
public static void readStream(Reader inputStream) {
   if (inputStream == null)
      readStandardInput();
   else {
      if ( inputStream instanceof BufferedReader)
         in = (BufferedReader)inputStream;
      else
         in = new BufferedReader(inputStream);
      emptyBuffer();  // Added November 2007
      inputFileName = null;
      readingStandardInput = false;
      inputErrorCount = 0;
   }
}


/**
 * Opens a file with a specified name for input.  If the file name is null, this has
 * the same effect as calling readStandardInput(); that is, input will be read from standard
 * input.  If an
 * error occurs while trying to open the file, an exception of type IllegalArgumentException
 * is thrown, and the input source is not changed.  If the file is opened
 * successfully, then after this method is called, all of the input routines will read
 * from the file, instead of from standard input.
 */
public static void readFile(String fileName) {
   if (fileName == null) // Go back to reading standard input
      readStandardInput();
   else {
      BufferedReader newin;
      try {
```

```
                newin = new BufferedReader( new FileReader(fileName) );
            }
            catch (Exception e) {
                throw new IllegalArgumentException("Can't open file \"" + fileName + "\" for input.\n"
                        + "(Error :" + e + ")");
            }
            if (! readingStandardInput) { // close current input stream
                try {
                    in.close();
                }
                catch (Exception e) {
                }
            }
            emptyBuffer();  // Added November 2007
            in = newin;
            readingStandardInput = false;
            inputErrorCount = 0;
            inputFileName = fileName;
        }
    }


    /**
     * Puts a GUI file-selection dialog box on the screen in which the user can select
     * an input file.  If the user cancels the dialog instead of selecting a file, it is
     * not considered an error, but the return value of the subroutine is false.
     * If the user does select a file, but there is an error while trying to open the
     * file, then an exception of type IllegalArgumentException is thrown.  Finally, if
     * the user selects a file and it is successfully opened, then the return value of the
     * subroutine is true, and  the input routines will read from the file, instead of
     * from standard input.   If the user cancels, or if any error occurs, then the
     * previous input source is not changed.
     * <p>NOTE: Calling this method starts a GUI user interface thread, which can continue
```

```
 * to run even if the thread that runs the main program ends.  If you use this method
 * in a non-GUI program, it might be necessary to call System.exit(0) at the end of the main()
 * routine to shut down the Java virtual machine completely.
 */
public static boolean readUserSelectedFile() {
   if (fileDialog == null)
      fileDialog = new JFileChooser();
   fileDialog.setDialogTitle("Select File for Input");
   int option = fileDialog.showOpenDialog(null);
   if (option != JFileChooser.APPROVE_OPTION)
      return false;
   File selectedFile = fileDialog.getSelectedFile();
   BufferedReader newin;
   try {
      newin = new BufferedReader( new FileReader(selectedFile) );
   }
   catch (Exception e) {
      throw new IllegalArgumentException("Can't open file \"" + selectedFile.getName() + "\" for input.\n"
               + "(Error :" + e + ")");
   }
   if (!readingStandardInput) { // close current file
      try {
         in.close();
      }
      catch (Exception e) {
      }
   }
   emptyBuffer();  // Added November 2007
   in = newin;
   inputFileName = selectedFile.getName();
   readingStandardInput = false;
   inputErrorCount = 0;
```

```
        return true;
    }


    /**
     * After this method is called, output will be written to standard output (as it
     * is in the default state).  If a file or stream was previously open for output, it
     * will be closed.
     */
    public static void writeStandardOutput() {
        if (writingStandardOutput)
            return;
        try {
            out.close();
        }
        catch (Exception e) {
        }
        outputFileName = null;
        outputErrorCount = 0;
        out = standardOutput;
        writingStandardOutput = true;
    }


    /**
     * After this method is called, output will be sent to outputStream, provided it
     * is non-null.  If outputStream is null, then this method has the same effect
     * as calling writeStandardOutput(); that is, future output will be sent to the
     * standard output stream.
     */
    public static void writeStream(OutputStream outputStream) {
        if (outputStream == null)
            writeStandardOutput();
```

```
    else
        writeStream(new PrintWriter(outputStream));
}


/**
 * After this method is called, output will be sent to outputStream, provided it
 * is non-null.  If outputStream is null, then this method has the same effect
 * as calling writeStandardOutput(); that is, future output will be sent to the
 * standard output stream.
 */
public static void writeStream(PrintWriter outputStream) {
    if (outputStream == null)
        writeStandardOutput();
    else {
        out = outputStream;
        outputFileName = null;
        outputErrorCount = 0;
        writingStandardOutput = false;
    }
}



/**
 * Opens a file with a specified name for output.  If the file name is null, this has
 * the same effect as calling writeStandardOutput(); that is, output will be sent to standard
 * output.  If an
 * error occurs while trying to open the file, an exception of type IllegalArgumentException
 * is thrown.  If the file is opened successfully, then after this method is called,
 * all of the output routines will write to the file, instead of to  standard output.
 * If an error occurs, the output destination is not changed.
 * <p>NOTE: Calling this method starts a GUI user interface thread, which can continue
 * to run even if the thread that runs the main program ends.  If you use this method
```

```
 * in a non-GUI program, it might be necessary to call System.exit(0) at the end of the main()
 * routine to shut down the Java virtual machine completely.
 */
public static void writeFile(String fileName) {
   if (fileName == null)  // Go back to reading standard output
      writeStandardOutput();
   else {
      PrintWriter newout;
      try {
         newout = new PrintWriter(new FileWriter(fileName));
      }
      catch (Exception e) {
         throw new IllegalArgumentException("Can't open file \"" + fileName + "\" for output.\n"
               + "(Error :" + e + ")");
      }
      if (!writingStandardOutput) {
         try {
            out.close();
         }
         catch (Exception e) {
         }
      }
      out = newout;
      writingStandardOutput = false;
      outputFileName = fileName;
      outputErrorCount = 0;
   }
}


/**
 * Puts a GUI file-selection dialog box on the screen in which the user can select
 * an output file.  If the user cancels the dialog instead of selecting a file, it is
```

```java
 * not considered an error, but the return value of the subroutine is false.
 * If the user does select a file, but there is an error while trying to open the
 * file, then an exception of type IllegalArgumentException is thrown.  Finally, if
 * the user selects a file and it is successfully opened, then the return value of the
 * subroutine is true, and  the output routines will write to the file, instead of
 * to standard output.  If the user cancels, or if an error occurs, then the current
 * output destination is not changed.
 */
public static boolean writeUserSelectedFile() {
   if (fileDialog == null)
      fileDialog = new JFileChooser();
   fileDialog.setDialogTitle("Select File for Output");
   File selectedFile;
   while (true) {
      int option = fileDialog.showSaveDialog(null);
      if (option != JFileChooser.APPROVE_OPTION)
         return false;  // user canceled
      selectedFile = fileDialog.getSelectedFile();
      if (selectedFile.exists()) {
         int response = JOptionPane.showConfirmDialog(null,
               "The file \"" + selectedFile.getName() + "\" already exists.  Do you want to replace it?",
               "Replace existing file?",
               JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE);
         if (response == JOptionPane.YES_OPTION)
            break;
      }
      else {
         break;
      }
   }
   PrintWriter newout;
   try {
```

```java
            newout = new PrintWriter(new FileWriter(selectedFile));
        }
    catch (Exception e) {
            throw new IllegalArgumentException("Can't open file \"" + selectedFile.getName() + "\" for
output.\n"
                    + "(Error :" + e + ")");
    }
    if (!writingStandardOutput) {
        try {
            out.close();
        }
        catch (Exception e) {

        }
    }
    out = newout;
    writingStandardOutput = false;
    outputFileName = selectedFile.getName();
    outputErrorCount = 0;
    return true;
    }



    /**
     * If TextIO is currently reading from a file, then the return value is the name of the file.
     * If the class is reading from standard input or from a stream, then the return value is null.
     */
    public static String getInputFileName() {
        return inputFileName;
    }



    /**
     * If TextIO is currently writing to a file, then the return value is the name of the file.
```

* If the class is writing to standard output or to a stream, then the return value is null.

* /

```java
public static String getOutputFileName() {

    return outputFileName;

}
```

// *************************** Output Methods *********************************

/**

* Write a single value to the current output destination, using the default format

* and no extra spaces. This method will handle any type of parameter, even one

* whose type is one of the primitive types.

*/

```java
public static void put(Object x) {

    out.print(x);

    out.flush();

    if (out.checkError())

        outputError("Error while writing output.");

}
```

/**

* Write a single value to the current output destination, using the default format

* and outputting at least minChars characters (with extra spaces added before the

* output value if necessary). This method will handle any type of parameter, even one

* whose type is one of the primitive types.

* @param x The value to be output, which can be of any type.

* @param minChars The minimum number of characters to use for the output. If x requires fewer

* then this number of characters, then extra spaces are added to the front of x to bring

* the total up to minChars. If minChars is less than or equal to zero, then x will be printed

* in the minimum number of spaces possible.

*/

```java
public static void put(Object x, int minChars)  {
    if (minChars <= 0)
        out.print(x);
    else
        out.printf("%" + minChars + "s", x);
    out.flush();
    if (out.checkError())
        outputError("Error while writing output.");
}


/**
 * This is equivalent to put(x), followed by an end-of-line.
 */
public static void putln(Object x) {
    out.println(x);
    out.flush();
    if (out.checkError())
        outputError("Error while writing output.");
}


/**
 * This is equivalent to put(x,minChars), followed by an end-of-line.
 */
public static void putln(Object x, int minChars) {
    put(x,minChars);
    out.println();
    out.flush();
    if (out.checkError())
        outputError("Error while writing output.");
}


/**
```

```java
 * Write an end-of-line character to the current output destination.
 */
public static void putln() {
   out.println();
   out.flush();
   if (out.checkError())
      outputError("Error while writing output.");
}


/**
 * Writes formatted output values to the current output destination.  This method has the
 * same function as System.out.printf(); the details of formatted output are not discussed
 * here.  The first parameter is a string that describes the format of the output.  There
 * can be any number of additional parameters; these specify the values to be output and
 * can be of any type.  This method will throw an IllegalArgumentException if the
 * format string is null or if the format string is illegal for the values that are being
 * output.
 */
public static void putf(String format, Object... items) {
   if (format == null)
      throw new IllegalArgumentException("Null format string in TextIO.putf() method.");
   try {
      out.printf(format,items);
   }
   catch (IllegalFormatException e) {
      throw new IllegalArgumentException("Illegal format string in TextIO.putf() method.");
   }
   out.flush();
   if (out.checkError())
      outputError("Error while writing output.");
}
```

```java
// ************************ Input Methods *******************************

/**
 * Test whether the next character in the current input source is an end-of-line.  Note that
 * this method does NOT skip whitespace before testing for end-of-line -- if you want to do
 * that, call skipBlanks() first.
 */
public static boolean eoln() {
   return peek() == '\n';
}


/**
 * Test whether the next character in the current input source is an end-of-file.  Note that
 * this method does NOT skip whitespace before testing for end-of-line -- if you want to do
 * that, call skipBlanks() or skipWhitespace() first.
 */
public static boolean eof()  {
   return peek() == EOF;
}


/**
 * Reads the next character from the current input source.  The character can be a whitespace
 * character; compare this to the getChar() method, which skips over whitespace and returns the
 * next non-whitespace character.  An end-of-line is always returned as the character '\n', even
 * when the actual end-of-line in the input source is something else, such as '\r' or "\r\n".
 * This method will throw an IllegalArgumentException if the input is at end-of-file (which will
 * not ordinarily happen if reading from standard input).
 */
public static char getAnyChar() {
   return readChar();
}
```

```
/**
 * Returns the next character in the current input source, without actually removing that
 * character from the input.  The character can be a whitespace character and can be the
 * end-of-file character (specified by the constant TextIO.EOF).An end-of-line is always returned
 * as the character '\n', even when the actual end-of-line in the input source is something else,
 * such as '\r' or "\r\n".  This method never causes an error.
 */
public static char peek() {
    return lookChar();
}


/**
 * Skips over any whitespace characters, except for end-of-lines.  After this method is called,
 * the next input character is either an end-of-line, an end-of-file, or a non-whitespace character.
 * This method never causes an error.  (Ordinarily, end-of-file is not possible when reading from
 * standard input.)
 */
public static void skipBlanks() {
    char ch=lookChar();
    while (ch != EOF && ch != '\n' && Character.isWhitespace(ch)) {
        readChar();
        ch = lookChar();
    }
}


/**
 * Skips over any whitespace characters, including for end-of-lines.  After this method is called,
 * the next input character is either an end-of-file or a non-whitespace character.
 * This method never causes an error. (Ordinarily, end-of-file is not possible when reading from
 * standard input.)
 */
private static void skipWhitespace() {
```

```
    char ch=lookChar();
    while (ch != EOF && Character.isWhitespace(ch)) {
        readChar();
        if (ch == '\n' && readingStandardInput && writingStandardOutput) {
            out.print("? ");
            out.flush();
        }
        ch = lookChar();
    }
}


/**
 * Skips whitespace characters and then reads a value of type byte from input, discarding the rest of
 * the current line of input (including the next end-of-line character, if any).  When using standard IO,
 * this will not produce an error; the user will be prompted repeatedly for input until a legal value
 * is input.  In other cases, an IllegalArgumentException will be thrown if a legal value is not found.
 */
public static byte getlnByte() {
    byte x=getByte();
    emptyBuffer();
    return x;
}


/**
 * Skips whitespace characters and then reads a value of type short from input, discarding the rest of
 * the current line of input (including the next end-of-line character, if any).  When using standard IO,
 * this will not produce an error; the user will be prompted repeatedly for input until a legal value
 * is input.  In other cases, an IllegalArgumentException will be thrown if a legal value is not found.
 */
public static short getlnShort() {
    short x=getShort();
    emptyBuffer();
```

```
      return x;
   }


   /**
    * Skips whitespace characters and then reads a value of type int from input, discarding the rest of
    * the current line of input (including the next end-of-line character, if any).  When using standard IO,
    * this will not produce an error; the user will be prompted repeatedly for input until a legal value
    * is input.  In other cases, an IllegalArgumentException will be thrown if a legal value is not found.
    */
   public static int getlnInt() {
      int x=getInt();
      emptyBuffer();
      return x;
   }


   /**
    * Skips whitespace characters and then reads a value of type long from input, discarding the rest of
    * the current line of input (including the next end-of-line character, if any).  When using standard IO,
    * this will not produce an error; the user will be prompted repeatedly for input until a legal value
    * is input.  In other cases, an IllegalArgumentException will be thrown if a legal value is not found.
    */
   public static long getlnLong() {
      long x=getLong();
      emptyBuffer();
      return x;
   }


   /**
    * Skips whitespace characters and then reads a value of type float from input, discarding the rest of
    * the current line of input (including the next end-of-line character, if any).  When using standard IO,
    * this will not produce an error; the user will be prompted repeatedly for input until a legal value
    * is input.  In other cases, an IllegalArgumentException will be thrown if a legal value is not found.
```

```java
*/
public static float getlnFloat() {

    float x=getFloat();

    emptyBuffer();

    return x;

}


/**
 * Skips whitespace characters and then reads a value of type double from input, discarding the rest of
 * the current line of input (including the next end-of-line character, if any).  When using standard IO,
 * this will not produce an error; the user will be prompted repeatedly for input until a legal value
 * is input.  In other cases, an IllegalArgumentException will be thrown if a legal value is not found.
 */
public static double getlnDouble() {

    double x=getDouble();

    emptyBuffer();

    return x;

}


/**
 * Skips whitespace characters and then reads a value of type char from input, discarding the rest of
 * the current line of input (including the next end-of-line character, if any).  Note that the value
 * that is returned will be a non-whitespace character; compare this with the getAnyChar() method.
 * When using standard IO, this will not produce an error.  In other cases, an error can occur if
 * an end-of-file is encountered.
 */
public static char getlnChar() {

    char x=getChar();

    emptyBuffer();

    return x;

}
```

```
/**
 * Skips whitespace characters and then reads a value of type boolean from input, discarding the rest of
 * the current line of input (including the next end-of-line character, if any).  When using standard IO,
 * this will not produce an error; the user will be prompted repeatedly for input until a legal value
 * is input.  In other cases, an IllegalArgumentException will be thrown if a legal value is not found.
 * <p>Legal inputs for a boolean input are: true, t, yes, y, 1, false, f, no, n, and 0; letters can be
 * either upper case or lower case. One "word" of input is read, using the getWord() method, and it
 * must be one of these; note that the "word"  must be terminated by a whitespace character (or end-of-file).
 */
public static boolean getlnBoolean() {
    boolean x=getBoolean();
    emptyBuffer();
    return x;
}


/**
 * Skips whitespace characters and then reads one "word" from input, discarding the rest of
 * the current line of input (including the next end-of-line character, if any).  A word is defined as
 * a sequence of non-whitespace characters (not just letters!).   When using standard IO,
 * this will not produce an error.  In other cases, an IllegalArgumentException will be thrown
 * if an end-of-file is encountered.
 */
public static String getlnWord() {
    String x=getWord();
    emptyBuffer();
    return x;
}


/**
 * This is identical to getln().
 */
public static String getlnString() {
```

```
      return getln();
  }


  /**
   * Reads all the characters from the current input source, up to the next end-of-line.  The end-of-line
   * is read but is not included in the return value.  Any other whitespace characters on the line are retained,
   * even if they occur at the start of input.  The return value will be an empty string if there are
   * no characters before the end-of-line.  When using standard IO, this will not produce an error.
   * In other cases, an IllegalArgumentException will be thrown if an end-of-file is encountered.
   */
  public static String getln() {
      StringBuffer s = new StringBuffer(100);
      char ch = readChar();
      while (ch != '\n') {
         s.append(ch);
         ch = readChar();
      }
      return s.toString();
  }


  /**
   * Skips whitespace characters and then reads a value of type byte from input.  Any additional characters on
   * the current line of input are retained, and will be read by the next input operation.  When using standard IO,
   * this will not produce an error; the user will be prompted repeatedly for input until a legal value
   * is input.  In other cases, an IllegalArgumentException will be thrown if a legal value is not found.
   */
  public static byte getByte()   {
      return (byte)readInteger(-128L,127L);
  }


  /**
   * Skips whitespace characters and then reads a value of type short from input.  Any additional characters on
```

* the current line of input are retained, and will be read by the next input operation. When using standard IO,

    * this will not produce an error; the user will be prompted repeatedly for input until a legal value

    * is input. In other cases, an IllegalArgumentException will be thrown if a legal value is not found.

    */

    public static short getShort() {

        return (short)readInteger(-32768L,32767L);

    }

    /**

    * Skips whitespace characters and then reads a value of type int from input. Any additional characters on

    * the current line of input are retained, and will be read by the next input operation. When using standard IO,

    * this will not produce an error; the user will be prompted repeatedly for input until a legal value

    * is input. In other cases, an IllegalArgumentException will be thrown if a legal value is not found.

    */

    public static int getInt()   {

        return (int)readInteger(Integer.MIN_VALUE, Integer.MAX_VALUE);

    }

    /**

    * Skips whitespace characters and then reads a value of type long from input. Any additional characters on

    * the current line of input are retained, and will be read by the next input operation. When using standard IO,

    * this will not produce an error; the user will be prompted repeatedly for input until a legal value

    * is input. In other cases, an IllegalArgumentException will be thrown if a legal value is not found.

    */

    public static long getLong()   {

        return readInteger(Long.MIN_VALUE, Long.MAX_VALUE);

    }

    /**

    * Skips whitespace characters and then reads a single non-whitespace character from input. Any additional characters on

```java
   * the current line of input are retained, and will be read by the next input operation.  When using standard IO,
   * this will not produce an error.  In other cases, an IllegalArgumentException will be thrown if an end-of-file
   * is encountered.
   */
   public static char getChar() {
      skipWhitespace();
      return readChar();
   }


   /**
    * Skips whitespace characters and then reads a value of type float from input.  Any additional characters on
    * the current line of input are retained, and will be read by the next input operation.  When using standard IO,
    * this will not produce an error; the user will be prompted repeatedly for input until a legal value
    * is input.  In other cases, an IllegalArgumentException will be thrown if a legal value is not found.
    */
   public static float getFloat() {
      float x = 0.0F;
      while (true) {
         String str = readRealString();
         if (str == null) {
            errorMessage("Floating point number not found.",
                  "Real number in the range " + (-Float.MAX_VALUE) + " to " + Float.MAX_VALUE);
         }
         else {
            try {
               x = Float.parseFloat(str);
            }
            catch (NumberFormatException e) {
               errorMessage("Illegal floating point input, " + str + ".",
                     "Real number in the range " +  (-Float.MAX_VALUE) + " to " + Float.MAX_VALUE);
               continue;
```

```
        }
        if (Float.isInfinite(x)) {
            errorMessage("Floating point input outside of legal range, " + str + ".",
                    "Real number in the range " +  (-Float.MAX_VALUE) + " to " + Float.MAX_VALUE);
            continue;
        }
        break;
    }
}
inputErrorCount = 0;
return x;
}


/**
 * Skips whitespace characters and then reads a value of type double from input.  Any additional characters on
 * the current line of input are retained, and will be read by the next input operation.  When using standard IO,
 * this will not produce an error; the user will be prompted repeatedly for input until a legal value
 * is input.  In other cases, an IllegalArgumentException will be thrown if a legal value is not found.
 */
public static double getDouble() {
    double x = 0.0;
    while (true) {
        String str = readRealString();
        if (str == null) {
            errorMessage("Floating point number not found.",
                    "Real number in the range " + (-Double.MAX_VALUE) + " to " + Double.MAX_VALUE);
        }
        else {
            try {
                x = Double.parseDouble(str);
            }
```

```
         catch (NumberFormatException e) {
            errorMessage("Illegal floating point input, " + str + ".",
                "Real number in the range " + (-Double.MAX_VALUE) + " to " + Double.MAX_VALUE);
            continue;
         }
         if (Double.isInfinite(x)) {
            errorMessage("Floating point input outside of legal range, " + str + ".",
                "Real number in the range " + (-Double.MAX_VALUE) + " to " + Double.MAX_VALUE);
            continue;
         }
         break;

      }
   }
   inputErrorCount = 0;
   return x;
}


/**
 * Skips whitespace characters and then reads one "word" from input. Any additional characters on
 * the current line of input are retained, and will be read by the next input operation.  A word is defined as
 * a sequence of non-whitespace characters (not just letters!).   When using standard IO,
 * this will not produce an error.  In other cases, an IllegalArgumentException will be thrown
 * if an end-of-file is encountered.
 */
public static String getWord() {
   skipWhitespace();
   StringBuffer str = new StringBuffer(50);
   char ch = lookChar();
   while (ch == EOF || !Character.isWhitespace(ch)) {
      str.append(readChar());
      ch = lookChar();
   }
```

```
      return str.toString();
  }


  /**
   * Skips whitespace characters and then reads a value of type boolean from input.  Any additional characters on
   * the current line of input are retained, and will be read by the next input operation.  When using standard IO,
   * this will not produce an error; the user will be prompted repeatedly for input until a legal value
   * is input.  In other cases, an IllegalArgumentException will be thrown if a legal value is not found.
   * <p>Legal inputs for a boolean input are: true, t, yes, y, 1, false, f, no, n, and 0; letters can be
   * either upper case or lower case. One "word" of input is read, using the getWord() method, and it
   * must be one of these; note that the "word"  must be terminated by a whitespace character (or end-of-file).
   */
  public static boolean getBoolean() {
      boolean ans = false;
      while (true) {
         String s = getWord();
         if ( s.equalsIgnoreCase("true") || s.equalsIgnoreCase("t") ||
              s.equalsIgnoreCase("yes")  || s.equalsIgnoreCase("y") ||
              s.equals("1") ) {
            ans = true;
            break;
         }
         else if ( s.equalsIgnoreCase("false") || s.equalsIgnoreCase("f") ||
              s.equalsIgnoreCase("no")  || s.equalsIgnoreCase("n") ||
              s.equals("0") ) {
            ans = false;
            break;
         }
         else
            errorMessage("Illegal boolean input value.",
            "one of:  true, false, t, f, yes, no, y, n, 0, or 1");
```

```
    }
    inputErrorCount = 0;
    return ans;
}
```

// **************** Everything beyond this point is private implementation detail ******************

private static String inputFileName; // Name of file that is the current input source, or null if the source is not a file.

private static String outputFileName; // Name of file that is the current output destination, or null if the destination is not a file.

private static JFileChooser fileDialog; // Dialog used by readUserSelectedFile() and writeUserSelectedFile()

private final static BufferedReader standardInput = new BufferedReader(new InputStreamReader(System.in)); // wraps standard input stream

private final static PrintWriter standardOutput = new PrintWriter(System.out); // wraps standard output stream

private static BufferedReader in = standardInput; // Stream that data is read from; the current input source.

private static PrintWriter out = standardOutput;   // Stream that data is written to; the current output destination.

private static boolean readingStandardInput = true;

private static boolean writingStandardOutput = true;

private static int inputErrorCount; // Number of consecutive errors on standard input; reset to 0 when a successful read occurs.

private static int outputErrorCount; // Number of errors on standard output since it was selected as the output destination.

private static Matcher integerMatcher; // Used for reading integer numbers; created from the integer Regex Pattern.

private static Matcher floatMatcher;   // Used for reading floating point numbers; created from the floatRegex Pattern.

```java
private final static Pattern integerRegex = Pattern.compile("(\\+|-)?[0-9]+");
private final static Pattern floatRegex = Pattern.compile("(\\+|-)?(([0-9]+(\\.[0-9]*)?)|(\\.[0-9]+))((e|E)(\\+|-)?[0-9]+)?");


private static String buffer = null;  // One line read from input.
private static int pos = 0;          // Position of next char in input line that has not yet been processed.


private static String readRealString() {   // read chars from input following syntax of real numbers
   skipWhitespace();
   if (lookChar() == EOF)
      return null;
   if (floatMatcher == null)
      floatMatcher = floatRegex.matcher(buffer);
   floatMatcher.region(pos,buffer.length());
   if (floatMatcher.lookingAt()) {
      String str = floatMatcher.group();
      pos = floatMatcher.end();
      return str;
   }
   else
      return null;
}


private static String readIntegerString() {  // read chars from input following syntax of integers
   skipWhitespace();
   if (lookChar() == EOF)
      return null;
   if (integerMatcher == null)
      integerMatcher = integerRegex.matcher(buffer);
   integerMatcher.region(pos,buffer.length());
   if (integerMatcher.lookingAt()) {
      String str = integerMatcher.group();
      pos = integerMatcher.end();
```

```java
        return str;
    }
    else
        return null;
}


private static long readInteger(long min, long max) {  // read long integer, limited to specified range
    long x=0;
    while (true) {
        String s = readIntegerString();
        if (s == null){
            errorMessage("Integer value not found in input.",
                    "Integer in the range " + min + " to " + max);
        }
        else {
            String str = s.toString();
            try {
                x = Long.parseLong(str);
            }
            catch (NumberFormatException e) {
                errorMessage("Illegal integer input, " + str + ".",
                        "Integer in the range " + min + " to " + max);
                continue;
            }
            if (x < min || x > max) {
                errorMessage("Integer input outside of legal range, " + str + ".",
                        "Integer in the range " + min + " to " + max);
                continue;
            }
            break;
        }
    }
```

```java
        inputErrorCount = 0;

        return x;

    }



    private static void errorMessage(String message, String expecting) {  // Report error on input.
        if (readingStandardInput && writingStandardOutput) {
            // inform user of error and force user to re-enter.
            out.println();
            out.print("  *** Error in input: " + message + "\n");
            out.print("  *** Expecting: " + expecting + "\n");
            out.print("  *** Discarding Input: ");
            if (lookChar() == '\n')
                out.print("(end-of-line)\n\n");
            else {
                while (lookChar() != '\n')    // Discard and echo remaining chars on the current line of input.
                    out.print(readChar());
                out.print("\n\n");
            }
            out.print("Please re-enter: ");
            out.flush();
            readChar();  // discard the end-of-line character
            inputErrorCount++;
            if (inputErrorCount >= 10)
                throw new IllegalArgumentException("Too many input consecutive input errors on standard input.");
        }
        else if (inputFileName != null)
            throw new IllegalArgumentException("Error while reading from file \"" + inputFileName + "\":\n"
                    + message + "\nExpecting " + expecting);
        else
            throw new IllegalArgumentException("Error while reading from inptu stream:\n"
                    + message + "\nExpecting " + expecting);
```

```java
    }

    private static char lookChar() {  // return next character from input
        if (buffer == null || pos > buffer.length())
            fillBuffer();
        if (buffer == null)
            return EOF;
        else if (pos == buffer.length())
            return '\n';
        else
            return buffer.charAt(pos);
    }

    private static char readChar() {  // return and discard next character from input
        char ch = lookChar();
        if (buffer == null) {
            if (readingStandardInput)
                throw new IllegalArgumentException("Attempt to read past end-of-file in standard input???");
            else
                throw new IllegalArgumentException("Attempt to read past end-of-file in file \"" + inputFileName
+ "\".");
        }
        pos++;
        return ch;
    }

    private static void fillBuffer() {    // Wait for user to type a line and press return,
        try {
            buffer = in.readLine();
        }
        catch (Exception e) {
            if (readingStandardInput)
                throw new IllegalArgumentException("Error while reading standard input???");
```

```java
        else if (inputFileName != null)
            throw new IllegalArgumentException("Error while attempting to read from file \"" + inputFileName
+ "\".");
        else
            throw new IllegalArgumentException("Errow while attempting to read form an input stream.");
    }
    pos = 0;
    floatMatcher = null;
    integerMatcher = null;
}


private static void emptyBuffer() {   // discard the rest of the current line of input
    buffer = null;
}


private static void outputError(String message) {  // Report an error on output.
    if (writingStandardOutput) {
        System.err.println("Error occurred in TextIO while writing to standard output!!");
        outputErrorCount++;
        if (outputErrorCount >= 10) {
            outputErrorCount = 0;
            throw new IllegalArgumentException("Too many errors while writing to standard output.");
        }
    }
    else if (outputFileName != null){
        throw new IllegalArgumentException("Error occurred while writing to file \""
            + outputFileName+ "\":\n   " + message);
    }
    else {
        throw new IllegalArgumentException("Error occurred while writing to output stream:\n   " + message);
    }
}
```

} // end of class TextIO

Step 8: Click on the Compile button and the program will be compiled. If there are any errors correct the errors and compile the program again.
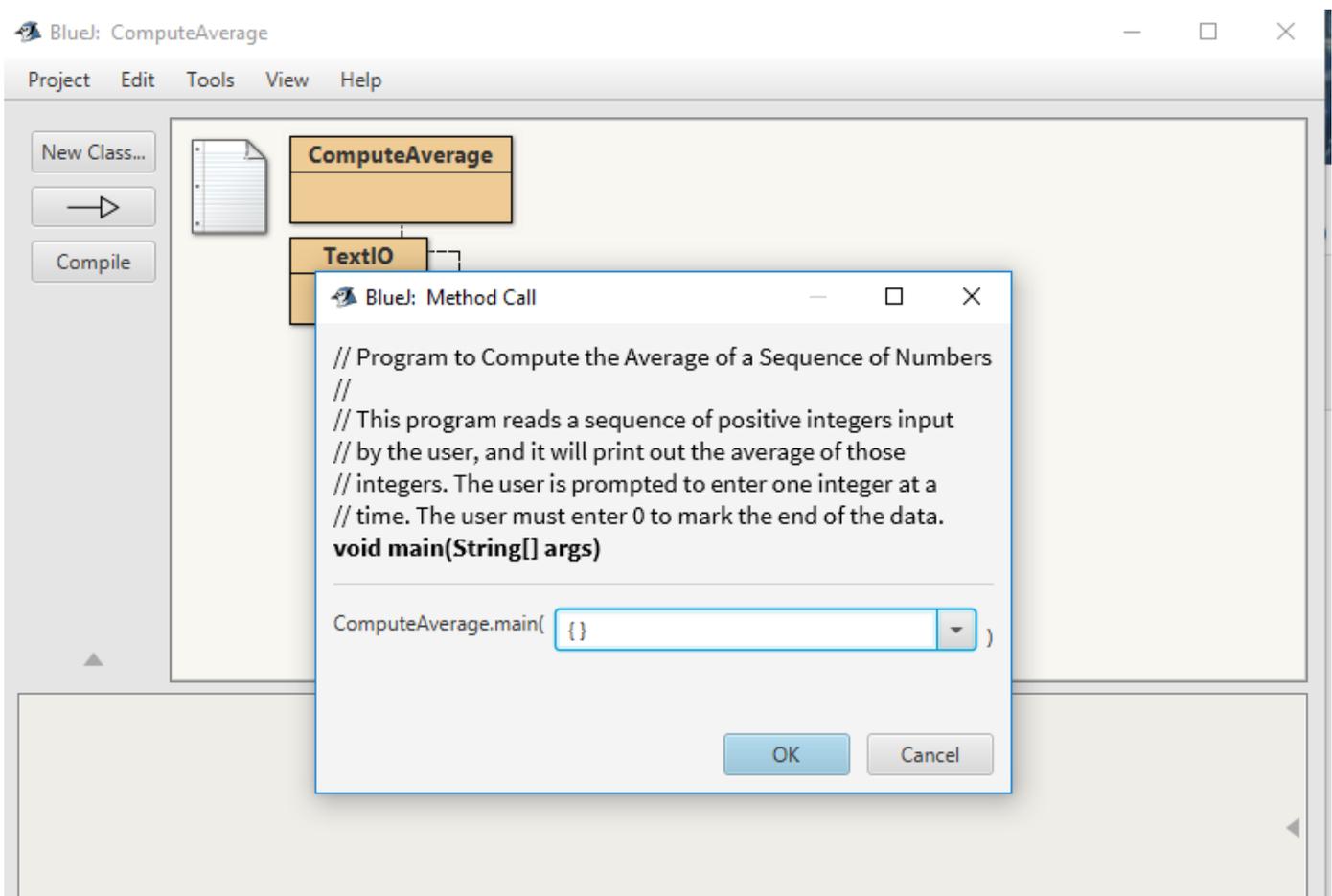
Step 9: Minimize the editor by click on the "_" button in the upper right hand corner of the screen.

Step 10: Right click on the EnumDemo box and a drop down menu will appear.

Step 11: Click on void main(String[ ] args) to start the program.

Step 12: Click on the OK button to run the program.

Step 13: A BlueJ Terminal Window will open and you will see the output displayed.

Output:

BlueJ: Terminal Window - ComputeAverage

Options

```
Enter your first positive integer: 12
Enter your next positive integer, or 0 to end: 18
Enter your next positive integer, or 0 to end: 32
Enter your next positive integer, or 0 to end: 16
Enter your next positive integer, or 0 to end: 22
Enter your next positive integer, or 0 to end: 0

You entered 5 positive integers.
Their average is 20.000.
```