```java
/*
 ************************************************************
 *
 * Notes:
 *    This is only a first draft, and I'm putting it out there
 *    so that you have something to go on.  I'll likely add
 *    more in the future, or take stuff away, so just remember
 *    while you code to keep the style goals in mind more than
 *    the details.  Fundamentally, your style should portray
 *    how much you care about the code, and how much time
 *    you spent making it elegant and beautiful.
 *
 *    Also, some of the instructions in this are specific
 *    to this class, and would not necessarily constitute
 *    good style outside of it.  For this class we are looking
 *    for your stylistic effort, and understanding of the code
 *    as well as actual style.  Your code should also be
 *    written with the goal of being printed out and read in
 *    mind, as well as being graded- something that might not
 *    be important for your future coding.
 *
 *    Furthermore, this is simply a guide to help you write
 *    beautiful, elegant, maintainable, and readable code.
 *    If you do decide to sway from instruction, please
 *    supply a good reason in comments.
 *
 *    While I will not take points off for not conforming to it
 *    necessarily the official java style guide is at:
 *    http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
 *    You should conform to this outline.
 *
 ************************************************************
 * Style Goals:
 *    Make your code easy to read / understand.
 *    Make your code easy to write.
 *    Make your code easy to print.
 *    Make your code easy to view.
 *    Convey how much you love your code - You love it.
 *    Make your code look exactly how you designed it to look
 *         from any reasonable editor
 *         (emacs, vim, eclipse, Dr. Java, notepad...)
 *    Make your code fast.
 *    Make your code easy to debug.
 *    Remove redundancies.
 *    Protect code from bad programming.
 *    Conform to agreed standards.
 *    Make it easy for other people to write using your code.
 *    Make it easy for other people to extend your code.
 *    Convince others your code is good.
 *    Be consistent. (Consistency reduces confusion and error)
 **************************************************************
 *
 * Do not use whitespace (more than one new line)
 * to keep code clean.
 *
 * Use organized, formated comments.
 *
 * Use comments to organize code.
 *
 * It is more important that you keep the width
 * of comments and lines of code short,
 * then have line breaks be logical.
 * (about the width of the longest line here)
 *
 * I will not take points off for spelling.
 * In general though, people might get annoyed
 * if your spelling is bad.
 *
 ***************************************************************
 *
 * when writing comments:
 * use /**
 *      *
 *      *
 * for javadoc comments that can be read outside the source,
 * such as what something does and how to use it.
 * You don't need to use javadoc annotations or HTML-
 * just good formating principles
 *
 ***************************************************************
 *
 * use /*
 *      *
 *      *
 * for developer comments that can only be read within the source:
 *     algorithm descriptions
 *     data descriptions
 *     notes and clarifications
 *     discussion about how to write more code
 *     things that need work (TODO)
 *
 ***************************************************************
 *
 * use //
 * for single line note comments.
 * It is usually not permissible to include this
 * type of comment on the same line as code
 * Except when you have a very short line of type declaration.
 *    Ex:
 *       int x=5; //x is not an index, but 5 is a pretty number.
 *
 */


/*
 *  If you import more than 3 or 4 classes from the
 *  same class I recommend you use .*
 */
import java.util.*;


/**
 * A general description of:
 *     What the class is,
 *     What it should/can be used for,
 *     How it should be used.
 *
 * A more detailed description of:
 *     How it should be used,
 *     How the algorithms work.
 *
 * Notes:
 *     Precautions,
 *     depreciations,
 *     things that need work (TODO)
 *
 * @author Matthew B. Mirman ( mmirman@andrew.cmu.edu )
 * @version 1
 * @date (last edited date)
 */
public class Style {


      /*
       * On curly brackets:
       *    It does not matter if you put
       *    your curly brackets on a new line
       *      ex:
       *          if (i<3)
       *          {
       *          ..do something..
       *          }
       *          else
       *          {
       *          ..do something else..
       *          }
       *
       *   or if you put your curly brackets
       *   on the same line
       *     ex:
       *        if (i<3) {
       *            ..do something..
       *        }else{
       *            ..do something else..
       *        }
       *   as long as you are consistent.
       *
       *   The only things that should ever come after
       *   a curly bracket on the same line are "else ... "
       *   or "catch ... ".  Not comments or other type
       *   of code though.  If you think of something else
       *   that I am forgetting, tell me.
       *
       *   If you decide not to use curly brackets for a
       *   block, I recommend you put the code in a block
       *   on a new line after an indentation.
       *     ex:
       *        if (i<3)
       *           i++;
       *
       ***********************************************************
       *
       * Indentations:
       *     It does not matter whether you use 4 space or tab,
       *     as long as you are very very consistent.
       *     Both have advantages and disadvantages.
       *
       *     4 space:
       *         advantages:
       *             -super consistent. It looks the
       *              same on every text editor.
       *             -Easier to format code how you want.
       *              You can also use 1,2,3,4,5... spaces if
       *              you so wish.
       *         disadvantages:
       *             -can take longer to write.
       *             -fewer editors have support
       *              for auto-formating using 4 spaces.
       *     Tabs:
       *         advantages:
       *             -very quick to type.
       *             -more auto-formatters
       *              have support for them
       *             -can have size altered depending on editor.
       *         disadvantages:
       *             -They look different depending on editor.
       *             -They encourage formating laziness.
       *             -can not be mixed with 4-space.
       *
       */


      /*          Data            *
       ****************************/

      // Try to make data private and use getters and setters
      private double x;


      /*   Getters and Setters    *
       *  Place just below data   *
       ****************************/

      /**
       * @return double x
       */
      public double getX() {
            return x;
      }

      /**
       * @param x
       */
      public void setX(final double x) {
            this.x = x;
      }


      /*  Equals, HashCode, toString    *
       *  Try to always implement these *
       *  Note: These have been         *
       *  autogenerated by eclipse      *
       **********************************/

      /*
       * Overridden methods do not necessarily need
       * javadoc comments.
       * (non-Javadoc)
       * @see java.lang.Object#equals(java.lang.Object)
       */
      @Override
      public String toString() {
            return "Style [x=" + x + "]";
      }

      @Override
      public int hashCode() {
            final int prime = 31;
            final long temp = Double.doubleToLongBits(x);
            return prime  + (int) (temp ^ (temp >>> 32));
      }

      @Override
      public boolean equals(final Object obj) {
            if (this == obj) {
                  return true;
            }
            if (obj == null) {
                  return false;
            }
            if (!(obj instanceof Style)) {
                  return false;
            }
            final Style other = (Style) obj;
            return this.x==other.x;
      }


      /*            Constructors              *
       ****************************************/

      /*
       * If you have more than one constructor that
       * does mostly the same thing but with different
       * combinations of arguments, it might be helpful
       * to create an initialization method,
       * with all possible arguments, and simply call that
       * from each different constructor.  The initializer
       * should be placed at the beginning or end of
       * the constructors.
       *
       */

      /**
       * Always include the empty constructor.
       * If you want there to be no empty constructor
       * make one that is private.
       */
      public Style() {
            setX(0);
      }

      /**
       * This constructor simply sets the only data
       * member, x, to what we expect it to be set as.
       * @param x
       */
      public Style(final double x){
            setX(x);
      }

      /**
       * This is a copy constructor.  for data based
       * classes it is very nice to have.
       *
       */
      public Style(final Style other){

            /*
             * Note that I do not use setX or getX here
             * because I want this Style to be an exact
             * copy of other Style, not whatever getX
             * and setX might change x to.
             * Be careful when setting references to objects
             * if those objects have copy constructors,
             * use them.  Think about memory, and write
             * comments explaining how memory is handled.
             */
            this.x=other.x;
      }


      /*          Methods           *
       ******************************/

      /*
       * Methods should be organized by what they do.
       */

      /**
       * Here we return a new style object so that you can
       * call add within lines of equations, use that value
       * but not effect any of the original values.
       * ex: suppose we had a Vec3D class with add, cross
       *     scale, and dot that all behaved this way.
       *     we could then write
       *         Vec3D angle=a.add(b).cross(c).scale(b.dot(c));
       *     which translates easily into math, as opposed to
       *         Vec3D composite=new Vec3D(a);
       *         composite.add(b);
       *         composite.cross(c);
       *         composite.scale(Vec3D.dot(b,c));
       * @param b
       * @return the sum of this's "x" value and b's "x" value.
       * @throws NullPointerException if b is null
       */
      public Style add(final Style b){

            /*
             * Note that here we throw a null pointer exception
             * within the method, but we don't have
             * "throws NullPointerException" in the method name.
             * While if a programmer is stupid and puts a null into this method
             * it shouldn't continue blindly computing stuff
             * (more of an issue with more code), we don't want the programmer
             * to have to surround this method with a try/catch every time he
             * wants to call it.  We do declare that it is possible for
             * this method to throw a nullPointerException in the method description
             * though, because it makes using it a bit easier.
             */
            if (b==null)
                  throw new NullPointerException();

            //create a new style object with the sum of this.getX() and b.getX()
            return new Style(this.getX() +b.getX());

            /*
             * Here, we used getX() because what if some time in the future
             * the programmer should decide either to extend style such that
             * x is really a function of x, or change that here.
             * It is much easier to do if you only ever use getX and setX.
             * There are cases where this isn't necessary
             */
      }

      /**
       * This method is just an example of using holder variables
       * so that repeated operations only need to happen once.
       * Not doing this can vastly increase the time complexity.
       * You should try to reduce the number of divides,
       * Math.sqrt and Math.pow as much as possible, because these
       * are really slow.  But optimize smart. If removing a sqrt
       * means you have to put in 15 other lines of code,
       * it is probably not more efficient, and much uglier.
       * Readability usually takes precedence over efficiency,
       * but something like
       *     for(int i=0;i<Math.sqrt(n-i);i++)
       * could easily be changed to
       *     for(int i=0;i*i+i<n;i++)
       * without effecting readability.
       *
       * @param c is the square route of the number
       *        of times x should be squared.
       */
      public void power(final int c){
            int c_squared=c*c;
            for (int i=0;i<c_squared;i++){
                  //getX could be very slow, we use a holder variable.
                  final double x_hold = this.getX();
                  setX(x_hold*x_hold);
            }
      }

      /**
       * If you ever use the same math/code twice or more,
       * and that code is more than 3- 5 operations or calls
       * long or is unexplained,
       * don't hesitate to write a helper method.
       *
       * You should do this to
       *    -Eliminate code redundancy (even preemptively)
       *    -Reduce amount of code that needs to be changed
       *     in the case of a bug
       *    -Make code easier to read by giving sequences of
       *     operations descriptive names
       *
       * In general, these make the most sense
       * as private or protected.
       *
       * @param i is the row number
       * @param j is the column number
       * @return the index into
       * @throws ArrayIndexOutOfBoundsException
       *         if i or j are smaller than zero
       */
      private int getIndexHelper(final int i, final int j){
            if (i<0 || j<0)
                  throw new ArrayIndexOutOfBoundsException();
            return (int)Math.floor(i*x+j);
      }
}


/*
 * Using final:
 *     if a variable isn't changed within a method,
 *     don't hesitate to use final, but try to anticipate
 *     it not being changed.  Do hesitate to make class
 *     data final, methods final, and classes final.
 *     Making things final helps keep you from accidentally
 *     changing a variable, then trying to access it's original
 *     value after it has been changed.
 *
 *
 *     If you don't know what final is, don't worry about it,
 *     and don't use it.
 */
```