Make a class `ArrayStats`. It has six private fields:

- an array of `double`,
- a `double` for the average of all the elements in the array,
- a `double` for the maximum element,
- a `double` for the minimum element,
- an `int` for the first index of the minimum element,
- an `int` for the first index of the maximum element.

If the array field is `null` or empty, the `double` fields will be 0 and the `int` fields will be -1.

The constructor will use the *vararg* notation on Slide 3.11

```
public ArrayStats(double... vals) {
// in the constructor assign vals as the value of the array field
// and compute the other 5 fields if vals is not null or empty and
// do all these 5 computations using one for loop.
}
```

Write public *getter* methods. To allow testing by the TAs, call the getters: `getData()`, `getMax`, `getMin`, `getMaxIdx`, `getMinIdx`, `getAvg`.

Write a test driver class that imports `java.util.Arrays` and has a `main` method that tests your code on the following:

```
ArrayStats test1 = new ArrayStats();
// this equivalent passing an empty array {} as the argument
ArrayStats test2 = new ArrayStats(null);
double[] arr = {7, 2, 3, 6, 8, 2, 1, 3, 6, 3, -4, 0, -5, 8};
ArrayStats test3 = new ArrayStats(arr);
```

The testing of the methods should state what the expected value is for each method call. Here is what we expect for `test2` (note that the static method `Arrays.toString` is in the class `java.util.Arrays` and gives a `String` representation of the array elements):

```
System.out.println("Testing test2.getData(), expect null");
System.out.println(Arrays.toString(test2.getData()));
System.out.println("Testing test2.getMax(), expect 0.0");
System.out.println(test2.getMax());
System.out.println("Testing test2.getMin(), expect 0.0");
System.out.println(test2.getMin());
System.out.println("Testing test2.getMaxIdx(), expect -1");
System.out.println(test2.getMaxIdx());
System.out.println("Testing test2.getMinIdx(), expect -1");
System.out.println(test2.getMinIdx());
System.out.println("Testing test2.getAvg(), expect 0.0");
System.out.println(test2.getAvg());
```

Important note When documenting the effect of a method call `obj.method(arg)`, we often have to talk about the object `obj`. It is referred either as the *"receiver"* of the method or *"this object"*

Implement all the code for the following class

```
package assignment02;
/**
 * Class to manage money in dollars and cents
 *
 * @author CS140
 */
public class DollarsAndCents {
// we need two private fields: a long called dollars and an int called cents

        /**
         * Constructor that initializes the dollars field to the given amount
         * @param dlrs The amount of the dollars in the new account. The parameter
         * must be a long. If an int is provided a different constructor is
         * called
         * @throws IllegalArgumentException if the parameter is negative
         */
        public DollarsAndCents(long dlrs) {
// this one is implemented for you check how the exception is thrown
                if (dlrs < 0) throw new IllegalArgumentException("argument cannot be
negative");
                dollars = dlrs;
        }

        /**
         * Constructor that initializes the cents field to the given amount, or the
         * dollars and cents if the number of cents is one dollar or more.
         * @param cts The amount of the cents in the new account. The parameter
         * must be an int. If a long is provided a different constructor is
         * called
         * @throws IllegalArgumentException if the parameter is negative
         */
        public DollarsAndCents(int cts) {
// this one is done for you. Note that if we called new DollarsAndCents(456),
// dollars would he 4 and cents would be 56 (cents cannot exceed 99)
                if (cts < 0) throw new IllegalArgumentException("argument cannot be
negative");
                dollars = cts/100;
                cents = cts%100;
        }

        /**
         * Constructor that initializes both the dollars and the cents field to
         * the given amounts. The cents are adjusted to be less than one dollar.
         * @param dlrs The amount of dollars in the new account.
         * @param cts The amount of the cents in the new account. if the number of
         * cents exceeds a dollar, the excess dollars are added to the dollars field
         * and only the cents less than one dollar remain.
         * @throws IllegalArgumentException if either of the parameters is negative
         */
        public DollarsAndCents(long dlrs, int cts) {
// use the ideas from the previous constructors to see how to do this
        }

        /**
         * A copy constructor that makes a new object with the same dollars and
         * cents values of the parameter
         * @param dc an object that provides the initial values of the fields
         * of the new object.
         */
        public DollarsAndCents(DollarsAndCents dc) {
// just copy the dollars and cents values from dc. Even though the fields are private
// you CAN access dc.dollars and dc.cents
        }
```

```
        /**
         * A no argument constructor that creates an object with both fields zero.
         */
        public DollarsAndCents() {
        }

        /**
         * Return a new object with the dollars and cents that are the sum of
         * the fields the receiver and all the parameters. If the amount of cents exceeds
         * one dollar the excess is passed to the dollars field.
         * @param dcs a list of zero or more DollarsAndCents objects
         * @return a DollarsAndCents object with the sum of all the dollars and
         * cents in the receiver and the parameters.
         */
        public DollarsAndCents add(DollarsAndCents... dcs) {
// note you treat dcs as an array
// the cents in the return value must be less than 100
        }

//BY THE WAY, the following method suggests we should define an equals method but
//we will do this when we have a chance to talk about the hashCode method
        /**
         * Tests if this object is less than the parameter. Consider the
         * dollars and cents as money and test if this amount is less than
         * the parameter's amount
         * @param dc a DollarsAndCents object to be compared with this object
         * @return true if this object is less than the other object in
         * terms of dollars and cents, otherwise false
         */
        public boolean lessThan(DollarsAndCents dc) {
// if dollars is less than dc.dollars the return value is true
// if dollars is equal to dc.dollars then if cents is less
// than dc.cents the return value is true
// otherwise the return value is false
        }

        /**
         * This could produce negative money, so an exception is thrown if the resulting
         * fields would be negative. Otherwise a new DollarsAndCents object is
         * returned with the parameter's dollars and cents values subtracted
         * from the receiver's fields
         * @param dc a DollarsAndCents object with values that are subtracted
         * from the receiver to produce the new object
         * @return a DollarsAndCents object with fields that are the difference between
         * the receiver and the parameter
         * @throws IllegalArgumentException if the parameter is not less than
         * this object or has fields that are equal to this object
         */
        public DollarsAndCents subtract(DollarsAndCents dc) {
// make an return-value object with two fields that are 0
// that is the return value if dollars and cents are equal to
// dc.dollars and dc.cents.
// if one or other of those values are different, throw an exception
// if(!dc.lessThan(this)) -- this means dc is not lessThan this object
// the code continues by making the dollars of the return-value object equal to
// dollars - dc.dollars and the cents of the return-value object equal to
// cents - dc.cents. Now this may make the cents of the return-value negative
// and in that case you have to reduce the dollars by 1 and add 100 to cents

        /**
         * Returns a new object that based on the receiver's dollars and cents
         * modifies by a positive factor.
```

```
         * @param factor the multiplying factor for the new object
         * @return a new object with fields that are the receiver's fields
         * modified by the factor
         * @throws IllegalArgumentException if the factor is zero
         */
        public DollarsAndCents upOrDown(double factor) {
// This computation must be done with doubles
// double temp = (d+c/100.0)*factor;
// the return value with have (long)temp for dollars and
// (int)Math.round((temp - (long)temp)*100);
        }

        @Override
        public String toString() {
                // this is a method overridden from class Object
                return "$" + dollars + "." + cents;
        }
}
```

Write a separate test driver with tests for all the methods of `DollarsAndCents`.

Here are the methods from Chapter 6 of the textbook for an `Investment` and testing class `InvestmentRunner`. Copy them over and modify `Investment` by replacing `double balance`, `double aBalance`, `double targetBalance`, and `double getBalance()` by `DollarsAndCents balance`, `DollarsAndCents aBalance`, `DollarsAndCents targetBalance`, and `DollarsAndCents getBalance()`. You will also need to make a couple of changes to `InvestmentRunner`.

```
package assignment02;
/**
   A class to monitor the growth of an investment that
   accumulates interest at a fixed annual rate.
 */
public class Investment {
        private double balance;
        private double rate;
        private int year;

        /**
          Constructs an Investment object from a starting balance and
          interest rate.
          @param aBalance the starting balance
          @param aRate the interest rate in percent
         */
        public Investment(double aBalance, double aRate) {
                balance = aBalance;
                rate = aRate;
        }

        /**
          Keeps accumulating interest until a target balance has
          been reached.
          @param targetBalance the desired balance
         */
        public void waitForBalance(double targetBalance) {
                while (balance < targetBalance)       {
                        year++;
                        double interest = balance * rate / 100;
                        balance = balance + interest;
                }
        }
```

```
        /**
          Gets the current investment balance.
          @return the current balance
         */
        public double getBalance() {
                return balance;
        }

        /**
          Gets the number of years this investment has accumulated
          interest.
          @return the number of years since the start of the investment
         */
        public int getYears() {
                return year;
        }
}
```

```
package assignment02;
/**
   This program computes how long it takes for an investment
   to double.
 */
public class InvestmentRunner {
        public static void main(String[] args) {
                final double INITIAL_BALANCE = 10000;
                final double RATE = 5;
                Investment invest = new Investment(INITIAL_BALANCE, RATE);
                invest.waitForBalance(2 * INITIAL_BALANCE);
                int years = invest.getYears();
                System.out.println("The investment doubled after "
                                + years + " years");
        }
}
```