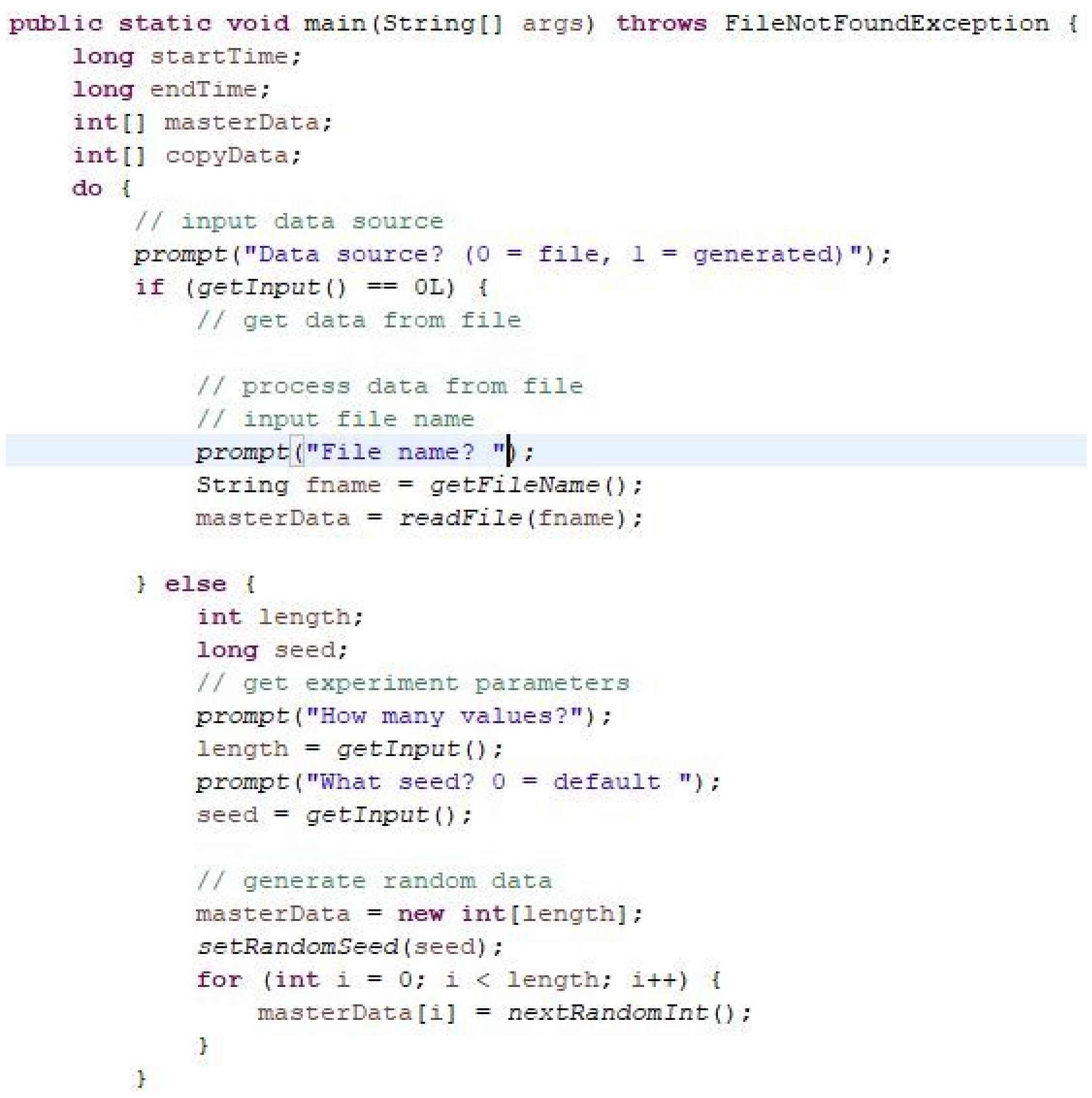
Christian Shinkle

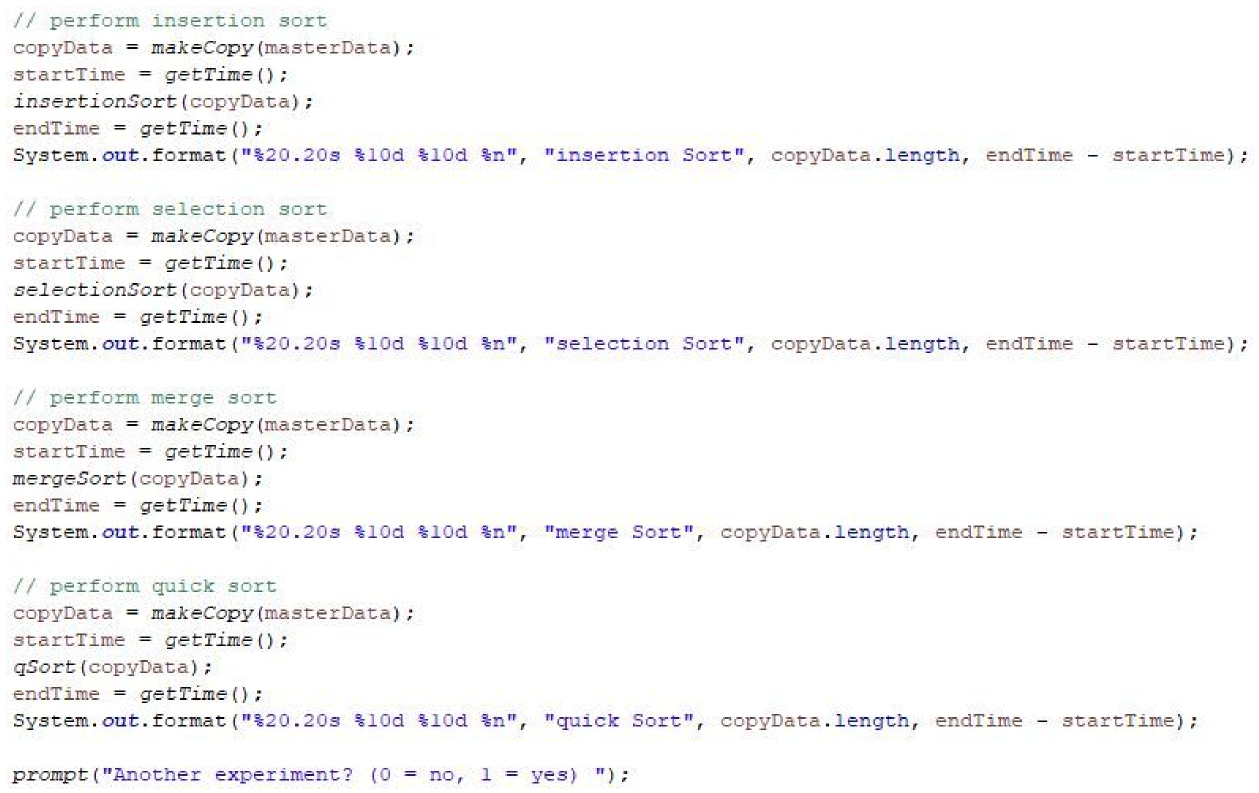
A note for the grader: I also included a copy finished of my code along with the PDF.

1.
    a. I believe the main cause of difficulty is lack of information hiding. For example, the current code for EvalSorts separates the logic of creating test data into a conditional statement. This makes it difficult to make changes to one side of the conditional statement without the need to change the other side. Another example of information hiding is all the source code is in a single file. This means there is no way for the two developers to work independently of each other without changing the same file, which is extremely tedious.
    b. Because the code isn't broken into independent modules, adding general functionality for a library would be very difficult because changing one piece of code would require several other locations of code to also be changed.
2.
    a. Each of the code smells I identified are:
        i. Duplicate code. In the if and else-blocks, after the data is gathered, almost all the code is the same. This is caused by separation of concerns. The code for running the tests could easily be placed into a helper method and then called by each block respectively. Lines 32-57 and 76-102 show this code smell.
        ii. Long methods. The main method of this class is very long for running what could be considered a fairly straightforward task. This caused by a lack of information hiding. The design decision for compiling, timing, and displaying the data for the sorts is all spread throughout the if and else-block. This information could easily be hidden away in private helper methods. Lines 10-102 show this code smell.
        iii. Divergent Change. If the user wanted to test another sorting algorithm, that would require multiple changes to the main method and an entire new method for the algorithm itself. This is caused by lack of abstraction. If we broke the sorting algorithms themselves into their own classes, we could design a method to make a collection with all the types of sorts and then run the timing on all the elements in this collection. Lines 32-57 and 76-102 show this code smell.
    b.



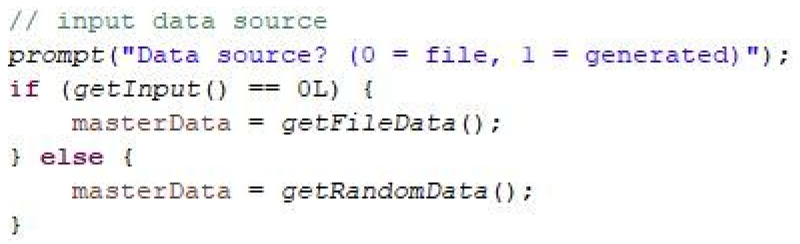
```java
public static void main(String[] args) throws FileNotFoundException {
    long startTime;
    long endTime;
    int[] masterData;
    int[] copyData;
    do {
        // input data source
        prompt("Data source? (0 = file, 1 = generated)");
        if (getInput() == 0L) {
            // get data from file

            // process data from file
            // input file name
            prompt("File name? ");
            String fname = getFileName();
            masterData = readFile(fname);

        } else {
            int length;
            long seed;
            // get experiment parameters
            prompt("How many values?");
            length = getInput();
            prompt("What seed? 0 = default ");
            seed = getInput();

            // generate random data
            masterData = new int[length];
            setRandomSeed(seed);
            for (int i = 0; i < length; i++) {
                masterData[i] = nextRandomInt();
            }
        }
```

I renamed the int arrays to masterData and copyData to more accurately reflect their roles.
Then, I removed the duplicate code from the if-blocks



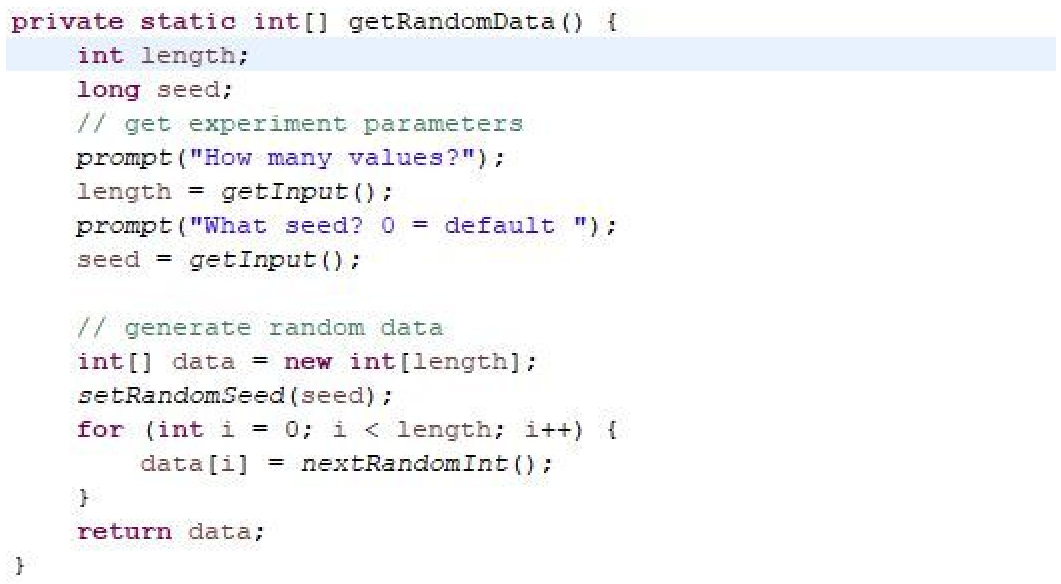
```
// perform insertion sort
copyData = makeCopy(masterData);
startTime = getTime();
insertionSort(copyData);
endTime = getTime();
System.out.format("%20.20s %10d %10d %n", "insertion Sort", copyData.length, endTime - startTime);

// perform selection sort
copyData = makeCopy(masterData);
startTime = getTime();
selectionSort(copyData);
endTime = getTime();
System.out.format("%20.20s %10d %10d %n", "selection Sort", copyData.length, endTime - startTime);

// perform merge sort
copyData = makeCopy(masterData);
startTime = getTime();
mergeSort(copyData);
endTime = getTime();
System.out.format("%20.20s %10d %10d %n", "merge Sort", copyData.length, endTime - startTime);

// perform quick sort
copyData = makeCopy(masterData);
startTime = getTime();
qSort(copyData);
endTime = getTime();
System.out.format("%20.20s %10d %10d %n", "quick Sort", copyData.length, endTime - startTime);

prompt("Another experiment? (0 = no, 1 = yes) ");
```

I moved the duplicate code to below the if-blocks and rename the variables accordingly.

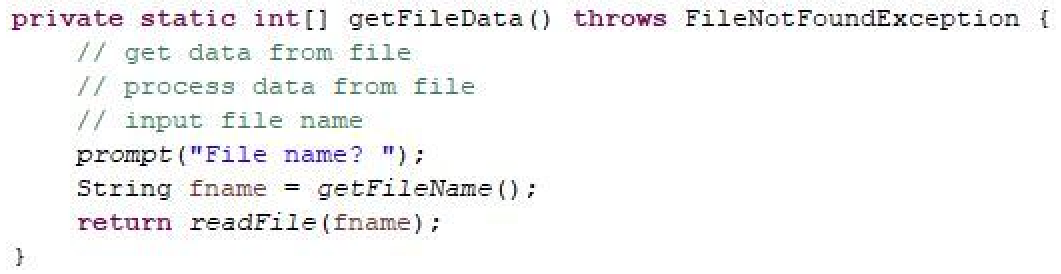


```
// input data source
prompt("Data source? (0 = file, 1 = generated)");
if (getInput() == 0L) {
    masterData = getFileData();
} else {
    masterData = getRandomData();
}
```

I made use of information hiding by moving the logic for the if-blocks into their own helper methods.

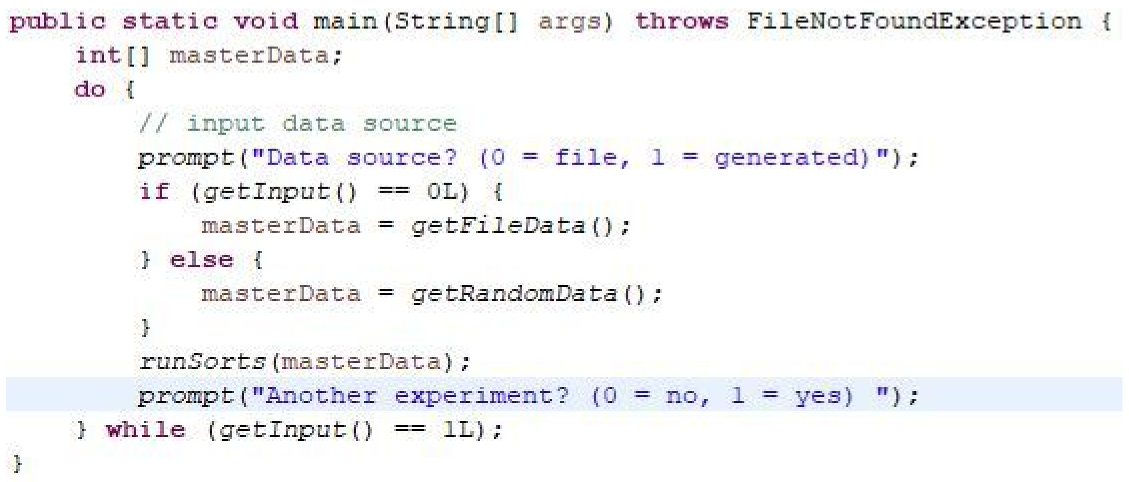


```
private static int[] getRandomData() {
    int length;
    long seed;
    // get experiment parameters
    prompt("How many values?");
    length = getInput();
    prompt("What seed? 0 = default ");
    seed = getInput();

    // generate random data
    int[] data = new int[length];
    setRandomSeed(seed);
    for (int i = 0; i < length; i++) {
        data[i] = nextRandomInt();
    }
    return data;
}
```



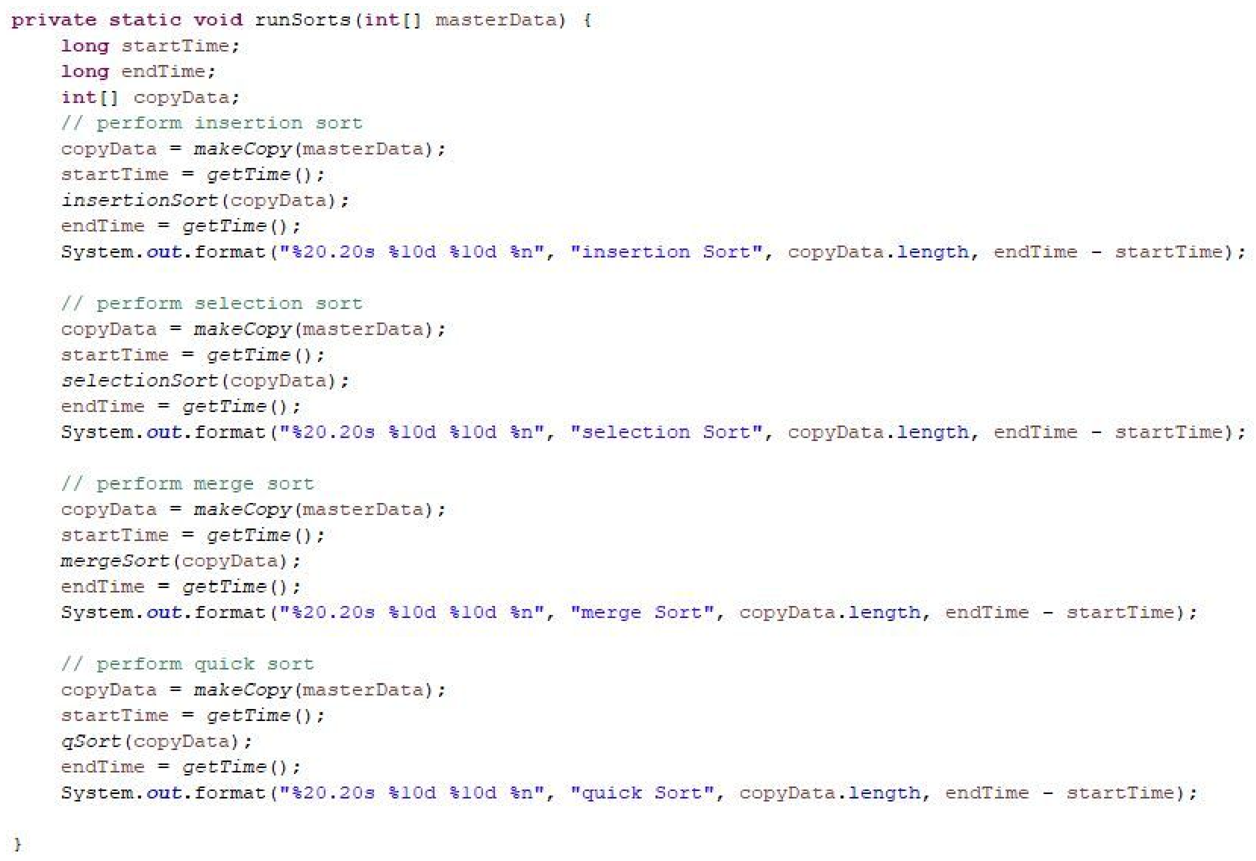
```
private static int[] getFileData() throws FileNotFoundException {
    // get data from file
    // process data from file
    // input file name
    prompt("File name? ");
    String fname = getFileName();
    return readFile(fname);
}
```

These are the two helper methods.



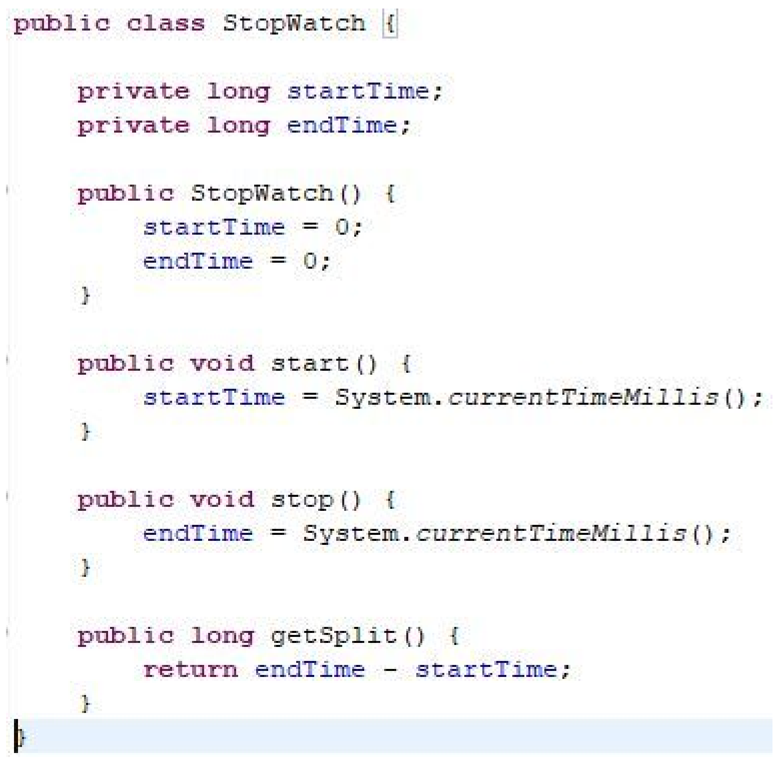
```
public static void main(String[] args) throws FileNotFoundException {
    int[] masterData;
    do {
        // input data source
        prompt("Data source? (0 = file, 1 = generated)");
        if (getInput() == 0L) {
            masterData = getFileData();
        } else {
            masterData = getRandomData();
        }
        runSorts(masterData);
        prompt("Another experiment? (0 = no, 1 = yes) ");
    } while (getInput() == 1L);
}
```

Again, I hid the logic for running the sorts in its own helper method.

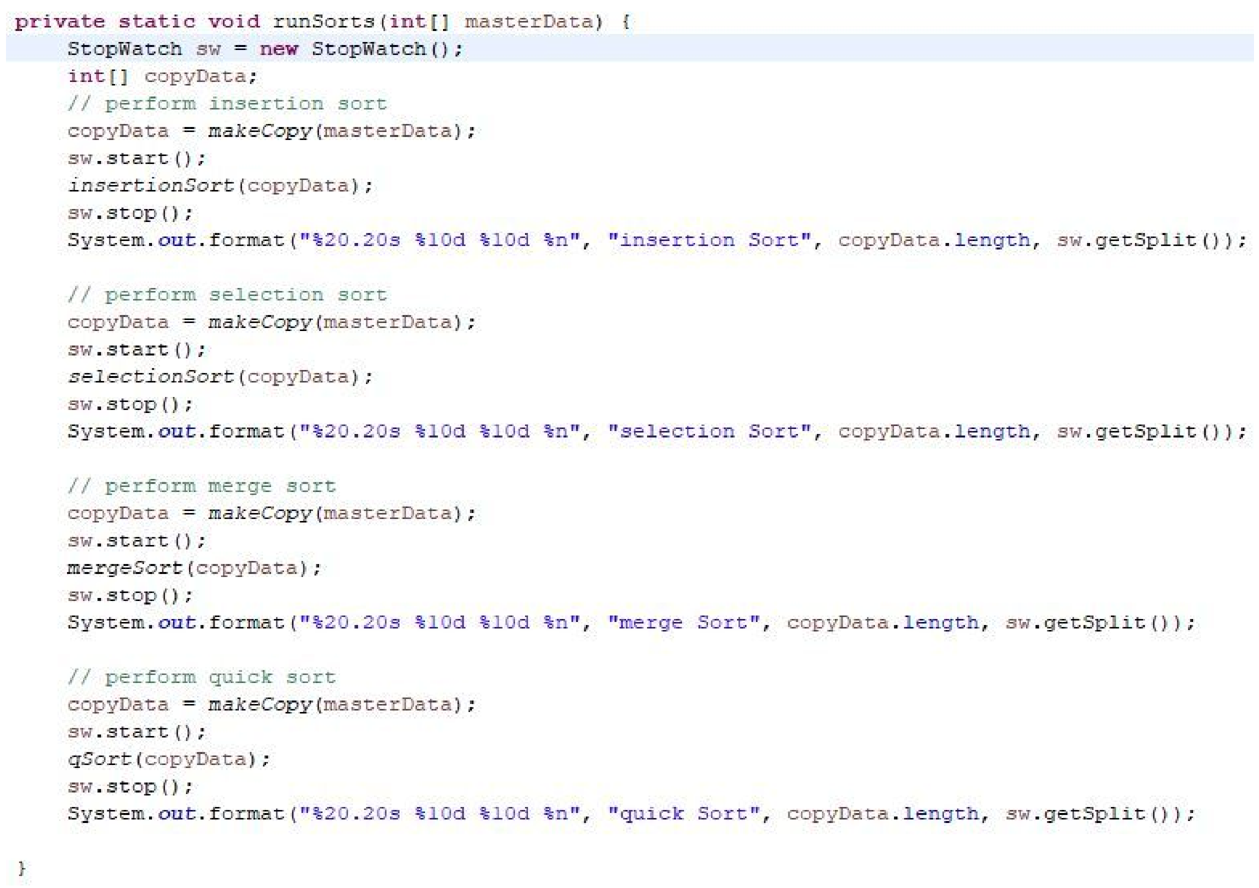


```
private static void runSorts(int[] masterData) {
    long startTime;
    long endTime;
    int[] copyData;
    // perform insertion sort
    copyData = makeCopy(masterData);
    startTime = getTime();
    insertionSort(copyData);
    endTime = getTime();
    System.out.format("%20.20s %10d %10d %n", "insertion Sort", copyData.length, endTime - startTime);

    // perform selection sort
    copyData = makeCopy(masterData);
    startTime = getTime();
    selectionSort(copyData);
    endTime = getTime();
    System.out.format("%20.20s %10d %10d %n", "selection Sort", copyData.length, endTime - startTime);

    // perform merge sort
    copyData = makeCopy(masterData);
    startTime = getTime();
    mergeSort(copyData);
    endTime = getTime();
    System.out.format("%20.20s %10d %10d %n", "merge Sort", copyData.length, endTime - startTime);

    // perform quick sort
    copyData = makeCopy(masterData);
    startTime = getTime();
    qSort(copyData);
    endTime = getTime();
    System.out.format("%20.20s %10d %10d %n", "quick Sort", copyData.length, endTime - startTime);

}
```

Here is the helper method. Note: some variables no longer need to be in the main method. I moved the variables to this method.

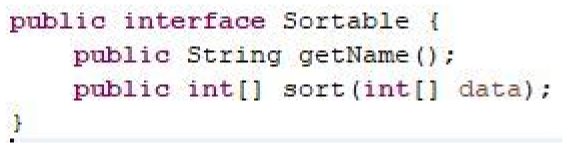

```java
public class StopWatch {

    private long startTime;
    private long endTime;

    public StopWatch() {
        startTime = 0;
        endTime = 0;
    }

    public void start() {
        startTime = System.currentTimeMillis();
    }

    public void stop() {
        endTime = System.currentTimeMillis();
    }

    public long getSplit() {
        return endTime - startTime;
    }
}
```

In order to hide away the implementation of the timing, I created a class just for timing.



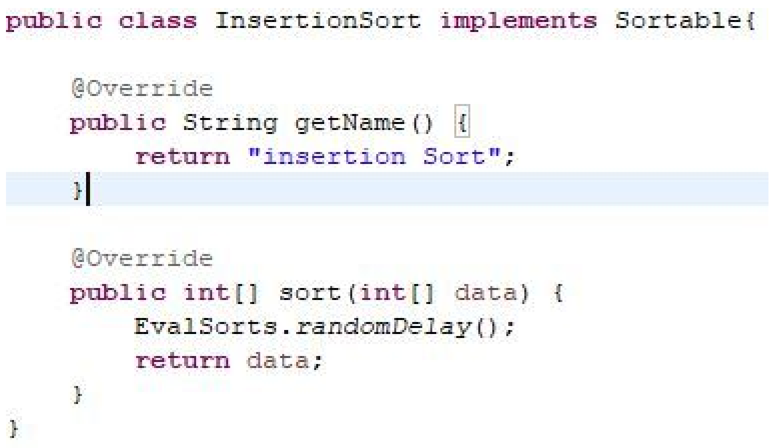
```
private static void runSorts(int[] masterData) {
    StopWatch sw = new StopWatch();
    int[] copyData;
    // perform insertion sort
    copyData = makeCopy(masterData);
    sw.start();
    insertionSort(copyData);
    sw.stop();
    System.out.format("%20.20s %10d %10d %n", "insertion Sort", copyData.length, sw.getSplit());

    // perform selection sort
    copyData = makeCopy(masterData);
    sw.start();
    selectionSort(copyData);
    sw.stop();
    System.out.format("%20.20s %10d %10d %n", "selection Sort", copyData.length, sw.getSplit());

    // perform merge sort
    copyData = makeCopy(masterData);
    sw.start();
    mergeSort(copyData);
    sw.stop();
    System.out.format("%20.20s %10d %10d %n", "merge Sort", copyData.length, sw.getSplit());

    // perform quick sort
    copyData = makeCopy(masterData);
    sw.start();
    qSort(copyData);
    sw.stop();
    System.out.format("%20.20s %10d %10d %n", "quick Sort", copyData.length, sw.getSplit());

}
```

The variables startTime and endTime are now hidden in the StopWatch object.



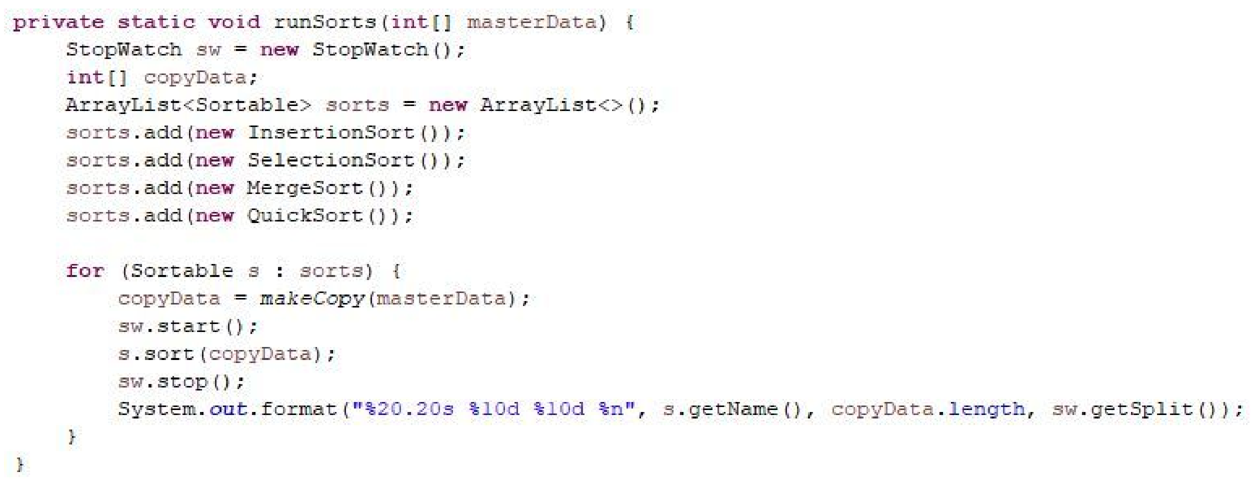
```
public interface Sortable {
    public String getName();
    public int[] sort(int[] data);
}
```

In order to account for change in the future, I created an interface to allow for extensibility. The two pieces of information are the name of the sort and the implementation of the sort itself.



```
public class InsertionSort implements Sortable{

    @Override
    public String getName() {
        return "insertion Sort";
    }

    @Override
    public int[] sort(int[] data) {
        EvalSorts.randomDelay();
        return data;
    }
}
```

Here is one example of an implementation of a sorting algorithm using the interface. Because every sort is exactly the same and they have the exact same implementation for the sort method, I won't show a capture of all the algorithms. The only change will be in the getName method.



```
private static void runSorts(int[] masterData) {
    StopWatch sw = new StopWatch();
    int[] copyData;
    ArrayList<Sortable> sorts = new ArrayList<>();
    sorts.add(new InsertionSort());
    sorts.add(new SelectionSort());
    sorts.add(new MergeSort());
    sorts.add(new QuickSort());

    for (Sortable s : sorts) {
        copyData = makeCopy(masterData);
        sw.start();
        s.sort(copyData);
        sw.stop();
        System.out.format("%20.20s %10d %10d %n", s.getName(), copyData.length, sw.getSplit());
    }
}
```

Lastly, here is the final version of the runSorts method. All the sorts are stored in a Collection (in this case, just an ArrayList), the Collection is then iterated through, and the timing and output of the sort is recorded just as before.
