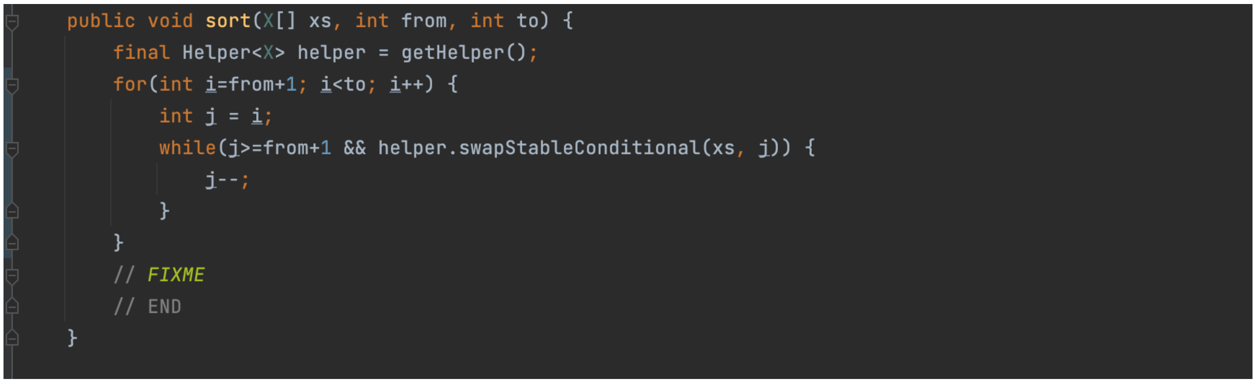
Name: KEERTHANA SATHEEESH
NUID: 002747795

Task:

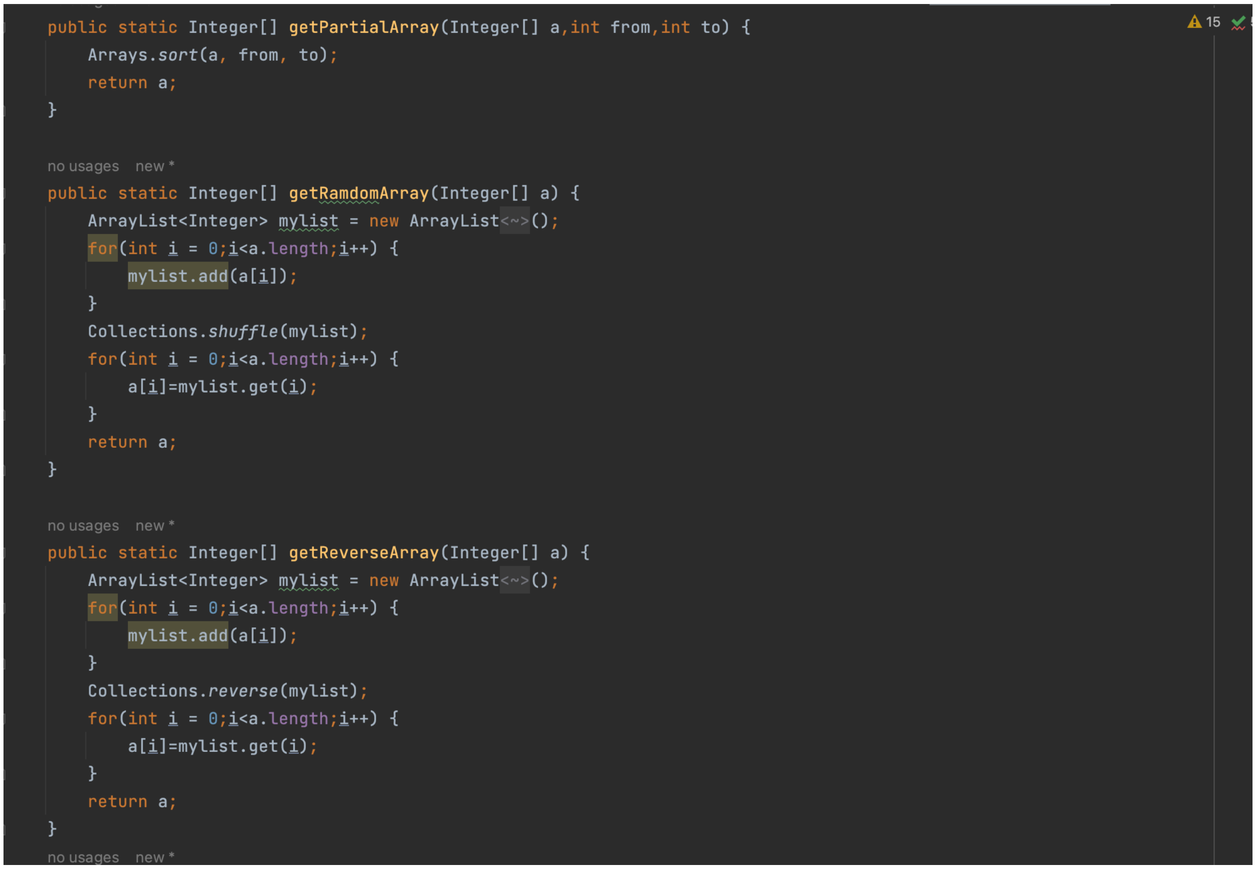
1) You are to implement three (3) methods (*repeat*, *getClock*, and *toMillisecs*) of a class called *Timer*.
2) Implement *InsertionSort* (in the *InsertionSort* class) by simply looking up the insertion code used by *Arrays.sort*.
3) Implement a main program (or you could do it via your own unit tests) to actually run the following benchmarks: measure the running times of this sort, using four different initial array ordering situations: random, ordered, partially-ordered and reverse-ordered.

Code:

Insertion Code:

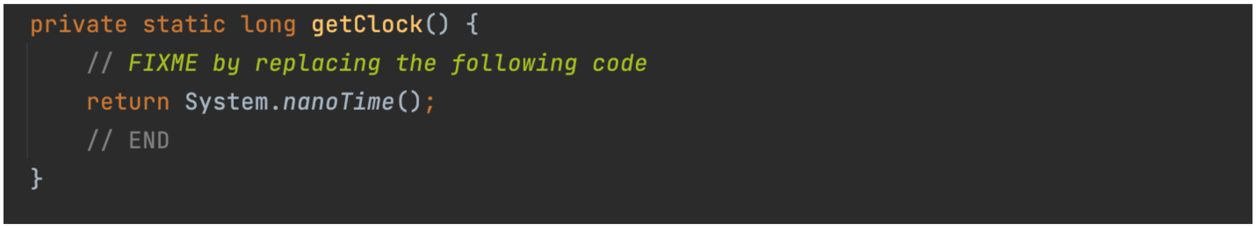


```java
public void sort(X[] xs, int from, int to) {
    final Helper<X> helper = getHelper();
    for(int i=from+1; i<to; i++) {
        int j = i;
        while(j>=from+1 && helper.swapStableConditional(xs, j)) {
            j--;
        }
    }
    // FIXME
    // END
}
```

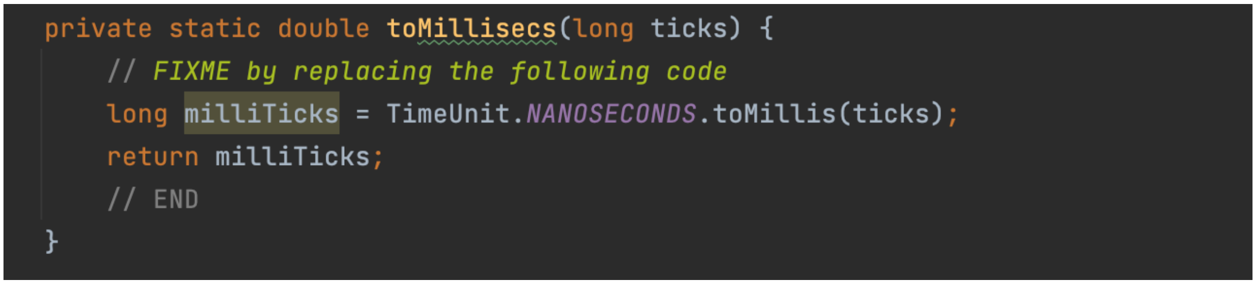


```
public static Integer[] getPartialArray(Integer[] a,int from,int to) {
    Arrays.sort(a, from, to);
    return a;
}

no usages   new *
public static Integer[] getRamdomArray(Integer[] a) {
    ArrayList<Integer> mylist = new ArrayList<~>();
    for(int i = 0;i<a.length;i++) {
        mylist.add(a[i]);
    }
    Collections.shuffle(mylist);
    for(int i = 0;i<a.length;i++) {
        a[i]=mylist.get(i);
    }
    return a;
}

no usages   new *
public static Integer[] getReverseArray(Integer[] a) {
    ArrayList<Integer> mylist = new ArrayList<~>();
    for(int i = 0;i<a.length;i++) {
        mylist.add(a[i]);
    }
    Collections.reverse(mylist);
    for(int i = 0;i<a.length;i++) {
        a[i]=mylist.get(i);
    }
    return a;
}
no usages   new *
```

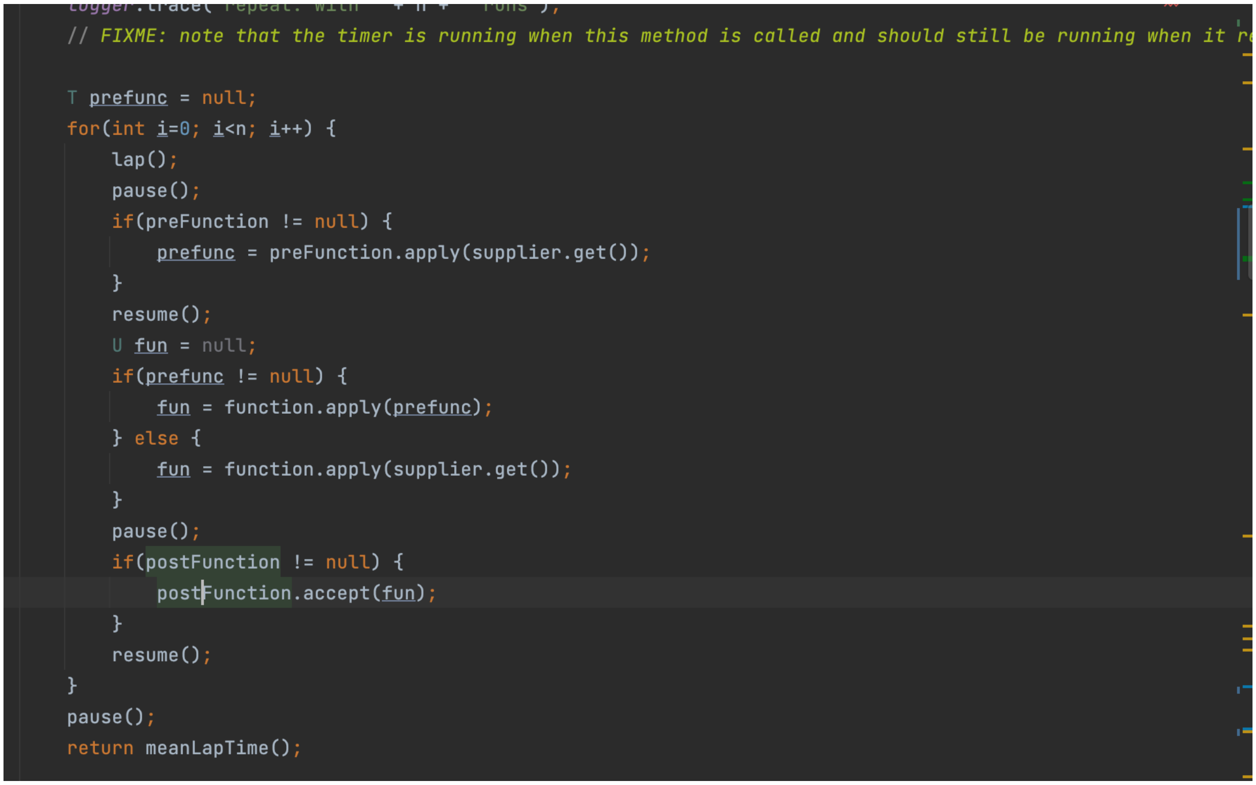
Timer Class Code:



```
private static long getClock() {
    // FIXME by replacing the following code
    return System.nanoTime();
    // END
}
```



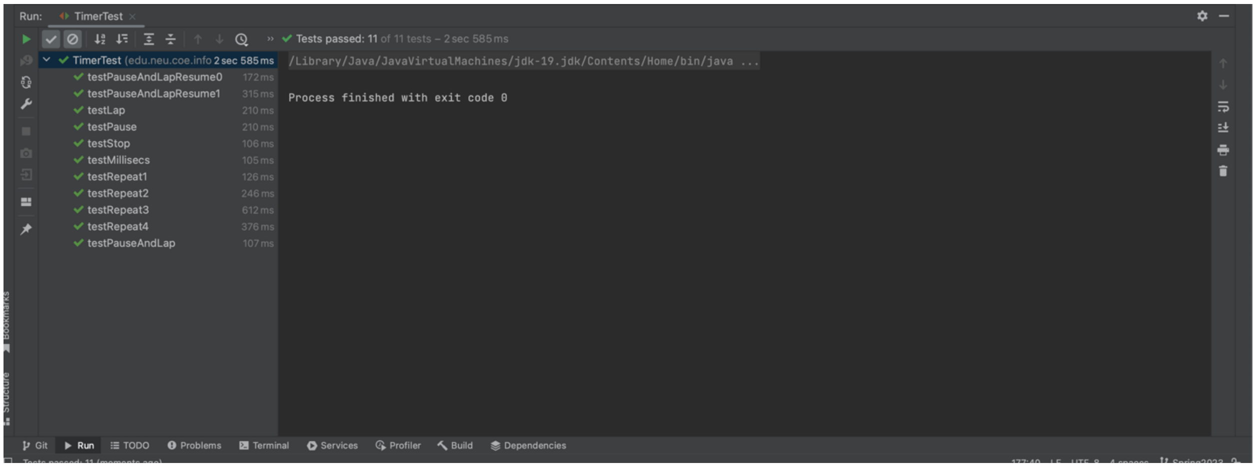
```
private static double toMillisecs(long ticks) {
    // FIXME by replacing the following code
    long milliTicks = TimeUnit.NANOSECONDS.toMillis(ticks);
    return milliTicks;
    // END
}
```



```java
logger.trace("repeat: with " + n + " runs");
// FIXME: note that the timer is running when this method is called and should still be running when it re

T prefunc = null;
for(int i=0; i<n; i++) {
    lap();
    pause();
    if(preFunction != null) {
        prefunc = preFunction.apply(supplier.get());
    }
    resume();
    U fun = null;
    if(prefunc != null) {
        fun = function.apply(prefunc);
    } else {
        fun = function.apply(supplier.get());
    }
    pause();
    if(postFunction != null) {
        postFunction.accept(fun);
    }
    resume();
}
pause();
return meanLapTime();
```

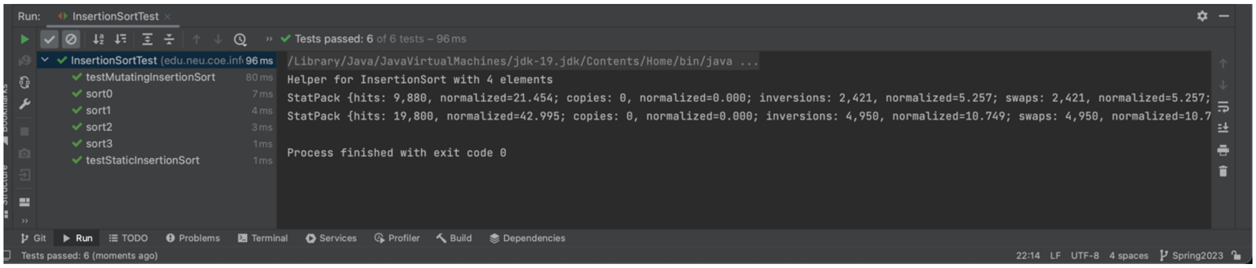
Unit Test Case:

1) Timer Test Case



```
✓ Tests passed: 11 of 11 tests – 2 sec 585 ms

✓ TimerTest (edu.neu.coe.info 2 sec 585 ms
    ✓ testPauseAndLapResume0    172 ms
    ✓ testPauseAndLapResume1    315 ms
    ✓ testLap                   210 ms
    ✓ testPause                 210 ms
    ✓ testStop                  106 ms
    ✓ testMillisecs             105 ms
    ✓ testRepeat1               126 ms
    ✓ testRepeat2               246 ms
    ✓ testRepeat3               612 ms
    ✓ testRepeat4               376 ms
    ✓ testPauseAndLap           107 ms

/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

2) Insertion Test Case



Tests passed: 6 of 6 tests – 96 ms

```
InsertionSortTest (edu.neu.coe.inf  96 ms
  testMutatingInsertionSort         80 ms
  sort0                              7 ms
  sort1                              4 ms
  sort2                              3 ms
  sort3                              1 ms
  testStaticInsertionSort            1 ms
```

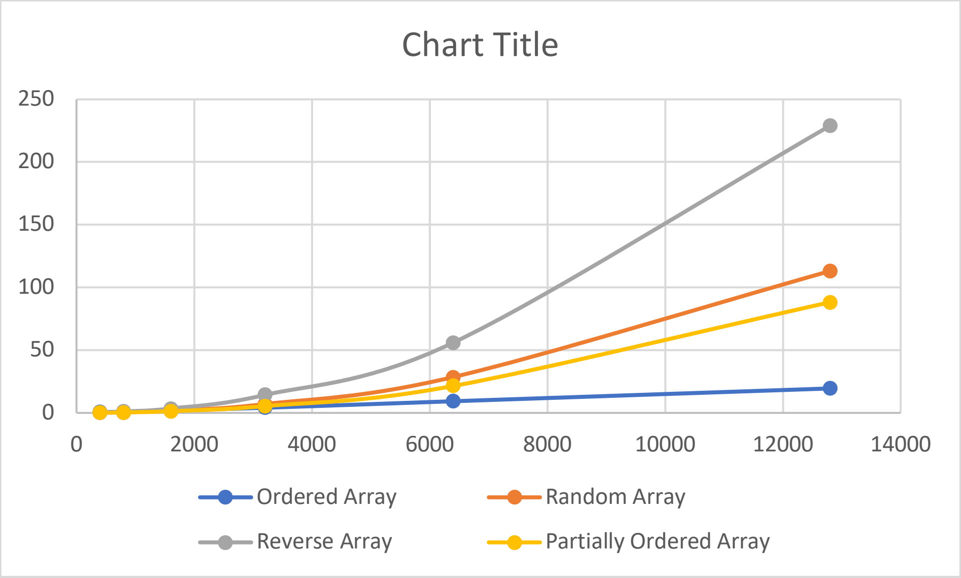
```
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bin/java ...
Helper for InsertionSort with 4 elements
StatPack {hits: 9,880, normalized=21.454; copies: 0, normalized=0.000; inversions: 2,421, normalized=5.257; swaps: 2,421, normalized=5.257;
StatPack {hits: 19,800, normalized=42.995; copies: 0, normalized=0.000; inversions: 4,950, normalized=10.749; swaps: 4,950, normalized=10.7

Process finished with exit code 0
```

Tests passed: 6 (moments ago)

Evidence:

Partially ordered array time increase based on the size of n - 0(n^2).
Reverse ordered array time increase based on the size of n - 0(n^2).
Ordered array time increase based on the size of n - 0(n).
Random ordered array time increase based on the size of n - 0(n^2).

n	Ordered Array	Random Array	Reverse Array	Partially Ordered Array
400	0.53	0.26	0.28	0.15
800	0.92	0.48	1.11	0.38
1600	2.06	1.85	3.49	1.36
3200	4.08	7.03	14.3	5.39
6400	9.29	28.57	56	21.45
12800	19.49	113.1	228.96	88.23



Conclusion:

According to the graph given above we can see that the Reverse array takes the largest sorting time. Next is random array followed by partially ordered array and finally ordered array. Ordered array takes the least time.