```
/**
 * Base abstract class for Sorters of various kinds.
 *
 * @author Adam Smith
 * @version 1.0
 */

public abstract class Sorter {
        /**
         * Do the actual sorting. This method should be overridden.
         *
         * @param array  the array to sort
         */

        abstract public <E extends Comparable<E>> void sort(E[] array);

        /**
         * Tells whether or not an array is sorted. Useful for
         * assertions. Will also return false if one of the elements is
         * null.
         *
         * @param array  the array that may be sorted
         * @return       whether or not it's sorted
         */

        public static final <E extends Comparable<E>> boolean isSorted(E[]
array) {
                if (array[0] == null) return false;

                // go thru each element, testing for order
                for (int i=1; i<array.length; i++) {
                        if (array[i] == null) return false;
                        if (array[i].compareTo(array[i-1]) < 0) return false;
                }

                // return true if we finished the loop without problem
                return true;
        }

        /**
         * Makes an array of Integers for testing. The array has the given
         * size, and is filled with random values between 0 and the size.
         * (Thus, duplicate values are almost certainly present, but
         * uncommon.)
         *
         * @param size  the number of elements in the new array
         * @return      the new array
         */

        public final static Integer[] makeArray(int size) {
                Integer[] array = new Integer[size];
                for (int i=0; i<array.length; i++) array[i] = new
Integer((int)(Math.random() * size));
                return array;
        }

        /**
```

```
         * Times a sort of an array of a particular size. This method uses
         * a new array of Integers of the given size, generated using
         * {@link #makeArray(int) makeArray()}.  It then returns the number of
milliseconds
         * the sort took, not including time taken to allocate the array
         * and its components. It will print a warning to stderr if the
         * array was not properly sorted.
         *
         * @param size  the number of elements to be sorted
         * @return      the number of milliseconds taken
         * @see         #makeArray(int)
         */

        public final int timeSort(int size) {
                // make array
                Integer[] array = makeArray(size);

                // hash the array's elements
                int oldHash = hashSort(array);

                // do the sort now, keeping track of start & end times
                long start = System.currentTimeMillis();
                sort(array);
                long end = System.currentTimeMillis();

                // return time, but warn iff the array is not sorted
                // make warnings, in case something's gone wrong
                if (!isSorted(array)) System.err.println("WARNING: the
algorithm did not sort correctly!");
                if (hashSort(array) != oldHash) System.err.println("WARNING:
the array's elements have been changed!");

                // return the time taken
                return (int)(end - start);
        }

        // this calculate a hash code of the array's elements, to make sure it
hasn't changed
        // (this function is invariant with respect to element order)
        private <E extends Comparable<E>> int hashSort(E[] array) {
                int hash = 0;
                for (int i=0; i<array.length; i++) hash ^=
array[i].hashCode();
                return hash;
        }
}
```