```
package
com.dataartisans.flinktraining.exercises.datastream_java.process;

import
com.dataartisans.flinktraining.exercises.datastream_java.datatypes.C
onnectedCarEvent;
import
com.dataartisans.flinktraining.exercises.datastream_java.utils.Compa
reByTimestampAscending;
import
com.dataartisans.flinktraining.exercises.datastream_java.utils.Conne
ctedCarAssigner;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.state.ValueState;
import org.apache.flink.api.common.state.ValueStateDescriptor;
import org.apache.flink.api.common.typeinfo.TypeHint;
import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.TimeCharacteristic;
import org.apache.flink.streaming.api.TimerService;
import org.apache.flink.streaming.api.datastream.DataStream;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironmen
t;
import org.apache.flink.streaming.api.functions.ProcessFunction;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

import java.util.PriorityQueue;

public class CarEventSort {
        public static void main(String[] args) throws Exception {

                // read parameters
```

```
                 ParameterTool params =
ParameterTool.fromArgs(args);
                 String input = params.getRequired("input");

                 // set up streaming execution environment
                 StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

                 // connect to the data file
                 DataStream<String> carData =
env.readTextFile(input);

                 // map to events
                 DataStream<ConnectedCarEvent> events = carData
                                  .map(new MapFunction<String,
ConnectedCarEvent>() {
                                          @Override
                                          public ConnectedCarEvent
map(String line) throws Exception {
                                                  return
ConnectedCarEvent.fromString(line);
                                          }
                                  })
                                  .assignTimestampsAndWatermarks(new
ConnectedCarAssigner());

                 // sort events
                 events
                                  .keyBy("carId")
                                  .process(new SortFunction())
                                  .print()


                 env.execute("Sort Connected Car Events");
         }

         public static class SortFunction extends
ProcessFunction<ConnectedCarEvent, ConnectedCarEvent> {
                 private
ValueState<PriorityQueue<ConnectedCarEvent>> queueState = null;

                 @Override
                 public void open(Configuration config) {

ValueStateDescriptor<PriorityQueue<ConnectedCarEvent>> descriptor =
new ValueStateDescriptor<>(
                                          // state name
                                          "sorted-events",
                                          // type information of
state
                                          TypeInformation.of(new
TypeHint<PriorityQueue<ConnectedCarEvent>>() {
                                }));
                        queueState = 
getRuntimeContext().getState(descriptor);
                }

                @Override
                public void processElement(ConnectedCarEvent event, 
Context context, Collector<ConnectedCarEvent> out) throws Exception 
{
                        TimerService timerService = 
context.timerService();

                        if (context.timestamp() > 
timerService.currentWatermark()) {
                                PriorityQueue<ConnectedCarEvent> 
queue = queueState.value();
                                if (queue == null) {
                                        queue = new 
PriorityQueue<>(10, new CompareByTimestampAscending());
                                }
                                queue.add(event);
                                queueState.update(queue);

timerService.registerEventTimeTimer(event.timestamp);
                        }
                }

                @Override
                public void onTimer(long timestamp, OnTimerContext 
context, Collector<ConnectedCarEvent> out) throws Exception {
                        PriorityQueue<ConnectedCarEvent> queue = 
queueState.value();
                        Long watermark = 
context.timerService().currentWatermark();
                        ConnectedCarEvent head = queue.peek();
                        while (head != null && head.timestamp <= 
watermark) {
                                out.collect(head);
                                queue.remove(head);
                                head = queue.peek();
                        }
                }
        }
}
```