```
/* *********************************
 *   BuyAgent.java
 *     Handles the buying of Derivitives.
 * *********************************/
package player.playeragent;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Date;
import java.util.Iterator;

import player.*;
import edu.neu.ccs.demeterf.demfgen.lib.List;
import gen.*;

/** Class for buying a derivative */
public class BuyAgent implements PlayerI.BuyAgentI{

        static int maxDerivsToBuy = 5;
        private int numProfitable = 0;

        public List<Derivative> buyDerivatives(List<Derivative> forSale, double account){
                Date d_start = new Date();
        long starttime = d_start.getTime();

                List<Derivative> profitable = profitableDerivatives(forSale, account);

                //make sure we are buying at least one Derivative...don't want to reoffer all
                //only buy least non-profitable derivative, while making sure we don't buy our own

                if(profitable.isEmpty()) {
                   double this_gain, this_bEven;
                   Derivative this_d;
                   Iterator<Derivative> iter = forSale.iterator();

                   ArrayList<Pair<Derivative, Double>> list = new ArrayList<Pair<Derivative, Double>>()
;

                   while(iter.hasNext()){
                      this_d = iter.next();
                      this_bEven = Util.breakEven(this_d);
                      this_gain = this_bEven - this_d.price.val;
                      list.add(new Pair<Derivative, Double>(this_d, this_gain));
                   }

                   //SORT the 'sortedList' by which is more profitable
              Collections.sort(list, new MoreProfitable());

              profitable = profitable.push(list.get(0).a);
              numProfitable++;
                }

                Iterator<Derivative> iter = profitable.iterator();

                ArrayList<Pair<Derivative, Double>> sortedList = new ArrayList<Pair<Derivative, Double>
>();

                double this_gain, this_bEven;
                Derivative this_d;

                while(iter.hasNext()){
                        this_d = iter.next();
                        this_bEven = Util.breakEven(this_d);
                        this_gain = this_bEven - this_d.price.val;
                        sortedList.add(new Pair<Derivative, Double>(this_d, this_gain));

                }

                // SORT the 'sortedList' by which is more profitable
                Collections.sort(sortedList, new MoreProfitable());

                // Trim the list to maxDerivsToBuy
                java.util.List<Pair<Derivative, Double>> trimmed_list = sortedList.subList(0, (Math.min
(maxDerivsToBuy, profitable.length())));
                Iterator<Pair<Derivative, Double>> p_iter = trimmed_list.iterator();

                List<Derivative> toBuy = List.<Derivative> create();

                while(p_iter.hasNext()){
                        this_d = p_iter.next().a;
                        toBuy = toBuy.push(this_d);
```

```
                }

        Date d_end = new Date();
        long endtime = d_end.getTime();
        System.out.println("Bought " + Math.min(numProfitable, 5) + " Derivatives");
                return toBuy;
        }



    /** Returns the profitable derivatives from those on sale */
    public List<Derivative> profitableDerivatives(List<Derivative> forSale, double account){
        return forSale.foldl(new List.Fold<Derivative, BuyChoice>(){
            public BuyChoice fold(Derivative d, BuyChoice choice){
                // Should this one be bought??
                if(shouldBuy(d, choice.account)) {
                    numProfitable++;
                    return choice.buy(d); }

                // I guess not...
                return choice;
            }}, new BuyChoice(account)).toBuy;
    }

    /** Is the given Derivative profitable? Do I have enough Money? */
    public boolean shouldBuy(Derivative der, double account){
       double beven = Util.breakEven(der);
       if (der.type.kind.isClassic()) {
          //System.out.println("The price " + der.price + " > " + beven);
          return (der.price.val < beven - 0.01) && der.price.val < account;
       }
       else return (der.price.val < beven - .1) && der.price.val < account;
    }

    /** Collects buying decisions based on the shouldBuy "predicate" */
    private class BuyChoice{
        double account;
        List<Derivative> toBuy;
        BuyChoice(double acc){ this(acc, List.<Derivative>create()); }
        BuyChoice(double acc, List<Derivative> buy){ account = acc; toBuy = buy; }
        BuyChoice buy(Derivative der){
            return new BuyChoice(account-der.price.val, toBuy.push(der));
        }
    }
}
```