FXCM Java API

Building a strategy

By Valeri Chakarov

This tutorial is the second part of the FXM Java API tutorials series. It is tightly connected to the piece of software developed in the first part named Pulling Historical Prices. In the following paragraphs I am going to explain how to build a strategy that is based on one of the most popular indicators in technical analysis – the CCI Indicator. The prerequisites for this tutorial include a fundamental to intermediate understanding of the Java language, knowledge of object-oriented concepts and principles, and having a basic idea of how financial markets and trading works.

In this document, I am going to explain thoroughly each part of the strategy in a very understandable way. In addition, at the end of the tutorial I am going to add an Appendix and place all the source code discussed in this tutorial. Both of these will be help you take a solid grasp of how to build a strategy. Have fun coding!

Before we start, please take a moment to read the following ***disclaimer***:

Trading foreign exchange on margin carries a high level of risk, and may not be suitable for all investors. Past performance is not indicative of future results. The high degree of leverage can work against you as well as for you. Before deciding to invest in foreign exchange you should carefully consider your investment objectives, level of experience, and risk appetite. The possibility exists that you could sustain a loss of some or all of your initial investment and therefore you should not invest money that you cannot afford to lose. You should be aware of all the risks associated with foreign exchange trading, and seek advice from an independent financial advisor if you have any doubts.

Building a Strategy

Important note: to fully understand the concepts of building and testing a strategy, be sure to also read the "FXCM Java API - Testing Our Strategy" as they both go hand in hand.

We start by creating an interface called Strategy and have all strategies implement it:

```
public interface Strategy {
StrategyResult runStrategy(List <Candlestick> candleStickList);
}
```

Our interface has only one method that we must implement: `runStrategy`. This method obtains a list of instances from the Candlesticks class, applies the strategy on the Candlestick's value and returns an instance of the StrategyResult class (see "FXCM Java API - Testing Our Strategy" for more details). Using an interface would later enable us to take advantage of the polymorphism mechanism when we'll want to see the results of our strategy's backtesting.

Our strategy is quite straightforward and is based on the very popular oscillator called CCI, which is short for Commodity Channel Index. The CCI was developed by Donald Lambert and featured in Commodities magazine in 1980. It is a very versatile indicator that can be used to identify a new trend or warn of extreme conditions. The indicator has been initially utilized only for commodities but it has been commonly applied to indices, ETFs, stocks, and other securities. More specifically, what CCI does is to measures the current price level relative to an average price level over a given period of time. CCI is high when the prices are above their average and low and the prices are below their average. This is why it has been widely used to identify overbought and oversold levels.

Below you can see the formula for calculating the CCI. For further information, you can have a look here and here.

```
CCI = (Typical Price - 20-period SMA of TP) / (.015 x Mean Deviation)


Typical Price (TP) = (High + Low + Close)/3


Constant = .015
```

In our strategy we will implement the CCI formula by selling the asset when its CCI is above a predefined ceiling threshold and buy the asset when its CCI is below a predefined floor threshold.
For the implementation of our strategy we will create `CCIIndicatorStrategy` class:

```
public class CCIIndicatorStrategy implements Strategy {

    private double STOP_LOSS;
    private double TAKE_PROFIT;
    public static int FLOOR_CCI;
    public static int CEILING_CCI;
    public static final double FACTOR = 0.015;
```

```
	public CCIIndicatorStrategy(double STOP_LOSS, double TAKE_PROFIT, int FLOOR_CCI) {
		super();
		STOP_LOSS = STOP_LOSS;
		TAKE_PROFIT = TAKE_PROFIT;
		FLOOR_CCI = FLOOR_CCI;
		CEILING_CCI = 0 - FLOOR_CCI;
	}
}

				//getters goes here...
```

As you can see we have four fields in this class. The `STOP_LOSS` and `TAKE_PROFIT` define the stop-loss and take-profit in percentage points. They are marked as final as they will be constant throughout the entire execution of each backtest but will vary from one backtest to another.
The `FLOOR_RSI` and `CEILING_RSI` define the RSI rates that will trigger opening a new trade to buy/sell the asset respectively. They are also marked as final. Note that the RSI is always between 0 – 100, so if the RSI argument passed to the constructor is out of this range, we'll notify it by throwing an exception.
The mechanism for backtesting a strategy is to first get a hold of historical data of the specific asset we want to test the strategy on (as we did in "FXCM Java API – Pulling Historical Data"), and then running the strategy on that data. In our case, before we can run the strategy we must calculate the RSI for each and every candlestick. To achieve that we will add a static method in `CCIIndicatorStrategy`:

As you can see we have four fields in this class. The STOP_LOSS and TAKE_PROFIT are declared to be used for stop-loss and take-profit in percentage format. The FLOOR_CCI and CELING_CCI define the limits that will trigger making an order to buy or sell the currency. Note that CCI is always between -100 and 100.
The mechanism for backtesting a strategy is to first get a hold of historical data of the specific asset we want to test the strategy on (as we did in "FXCM Java API – Pulling Historical Data"), and then running the strategy on that data. In our case, before we can run the strategy we must calculate the CCI for each and every candlestick. To achieve this we will add a static method in `CCIIndicatorStrategy`:

```
public static void calculateCciForDataSet(List<CandleStick> candleSticksList, int
duration){
	//Variable definitions
  double sum = 0;
  double typicalPriceInstance = 0;
  double smaOfTypicalPrices = 0;
  double cci = 0;
  double meanDeviation = 0;
  double sumOfDeviations = 0;
  List<Double>typicalPrices = new ArrayList<>();
	for (int j = 0; j < duration; j++) {
	  //Creation of Class instances
	TypicalPriceIndicator typicalPriceIn = newTypicalPriceIndicator(candleSticksList,
duration);
	SMAIndicator smaIndicator = new SMAIndicator();
	MeanDeviationIndicator meanDeviationIn = new MeanDeviationIndicator();
	CCIIndicator cciInd = new CCIIndicator();
```

```
                      //the method implementation continues here…
//Calculation of Typical Price
  typicalPriceIn.calculateTypicalPrice(candleSticksList, duration);
  typicalPrices.add(typicalPriceIn.calculateTypicalPrice(candleSticksList, duration))
  //Calculation of SMA of  Typical Prices
  smaOfTypicalPrices = smaIndicator.calculateSMA(typicalPrices, duration);
  meanDeviation = meanDeviationIn.calculateMeanDeviation(smaOfTypicalPrices,
typicalPrices, duration);
  //Calculation of CCI
  cci = cciInd.calculateCCI(typicalPrices, smaOfTypicalPrices,FACTOR, meanDeviation,
duration);

  //Set the CCI for each candle
  for (CandleStick candle : candleSticksList) {
       candle.setCCI(cci);
  }
 }
}
```

`calculateCciForDataSet` method gets two arguments: a list with all of the historical candlesticks of the asset and the duration of the CCI. The most typical used duration is 20 candles and that's what we are going to use for our strategy.

Each of these calculation has been developed in a different class and then the method performing the calculation is called in the calculateCciForDataSet method. This method consists of stages:

1. Start a for loop for the specific portion of prices you will analyse with CCI. In our case this portion consists of 20 candleSticks.
2. In this for loop we call a method which calculates the Typical Price called calculateTypicalPrice.

```
protected double calculateTypicalPrice(List<CandleStick> candleSticksList, int
index) {
            double typicalPrice;
            double high = 0;
            double low = 0 ;
            double closeBid = 0;
            for (int j = 0; j < index; j++) {
                   high = candleSticksList.get(j).getHigh();
                   low  = candleSticksList.get(j).getLow();
                   closeBid = candleSticksList.get(j).getCloseBid();
                   typicalPrice = (high + low + closeBid)/3;
                   return typicalPrice;
            }
            return 0;
      }
```

3. The next class instance is smalOFTypicalPrice which calls the method smaOfTypicalPrices is calculate the average of 20 typical prices and takes typicalPricesList as a parameter.

```
protected Double calculateSMA(List<Double> listOfPrices, int duration){
            double smaOfTypicalPrices =
(listOfPrices.stream().mapToDouble(Double::doubleValue).sum())/duration;
            return smaOfTypicalPrices;
            }
```

4. The next class instance is meanDeviation which calls the method calculatesMeanDeviation calculates the mean deviation of these 20 typical prices.

```
protected double calculateMeanDeviation(double average,
      List<Double> listOfPrices, int duration) {
             double sumOfDeviations = 0;
      for (int j = 0; j < duration; j++) {
              sumOfDeviations += Math.abs(average-listOfPrices.get(j));
             double meanDeviation = sumOfDeviations/duration;
             return meanDeviation;
      }
      return 0;
 }
```

5. The last class instance is cci which call the method calculatesCCI implemented below.

```
protected double calculateCCI(List<Double> listOfTypicalPrices,
              double averageOfTypicalPrices, double constant, double
                     meanDeviationOfTypicalPrices, int duration) {
      double cci = 0;
      for (int k = 0; k < duration; k++) {
        cci = (listOfTypicalPrices.get(k) - averageOfTypicalPrices/
               (constant*meanDeviationOfTypicalPrices));
             return cci;
      }
      return 0;
      }
```

6. After that we go through the whole list of typical prices with a for each loop and set the CCI values for each one of them.

One thing looks odd though, and that’s the `setCci(cci)` method we are using to set the RSI for each candlestick. It’s odd since it does not exist in the Candlestick object (see “FXCM Java API – Pulling Historical Data” for more details) and so it results in a compiler error. The Candlestick object we’ve used in the prior module describes a generic candlestick but each strategy may require its own unique candlestick with different characteristics. To tackle this issue, let’s change the Candlestick class to fit to our current needs, and add the extra fields/methods we will need to use for the execution of our strategy.

The Candlestick class should have a field called cci with its getter and setter. In addition, as we will see in a bit, for the proper execution of our strategy we should have 2 different closing prices: bid and ask. Since the generic Candlestick class had only one close field which refers to the bid price, we should add a closeAsk field to store the closing ask price of the asset. After updating it, the Candlestick class looks like this:

```
public class Candlestick{
private String date;
private double open;
private double low;
private double high;
private double closeBid;
private double closeAsk;
private double cci;
public Candlestick(String date, double open, double low, double high, double closeBid,
double closeAsk) {
this.date = date;
this.open = open;
this.low = low;
this.high = high;
this.closeBid = closeBid;
this.closeAsk = closeAsk;
}
public Candlestick(String date, double open, double low, double high, double closeBid) {
this.date = date;
this.open = open;
this.low = low;
this.high = high;
this.closeBid = closeBid;
}
// getters and setters...
}
```

There is still one problem though: in the method we use to convert historical data into candlesticks in the `HistoryMiner` class (see "FXCM Java API – Pulling Historical Data" for more details), we have only 1 closing price.Let's fix it by adding a `closeAsk` variable and changing the close to `closeBid`. We should also change the constructor we use to initialize each candlestick instance to the new one, which accepts both closing prices:

```
public void convertHistoricalRatesToCandleSticks(){
        // get the keys of the historicalRates map into a sorted list
        SortedSet<UTCDate> dateList = new TreeSet<>(historicalRates.keySet());
        // define a format for the dates
        SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yyyy HH:mm");
        // make the date formatter above convert from GMT to BST
        sdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));
        for(UTCDate date : dateList){
                MarketDataSnapshot candleData;
                candleData = historicalRates.get(date);
                // convert the key to a Date
                Date candleDate = date.toDate();
                String sdfDate = sdf.format(candleDate);
                double open = candleData.getBidOpen();
                double low = candleData.getBidLow();
                double high = candleData.getAskHigh(); // ask is the relevant price for short positions
                double closeBid = candleData.getBidClose(); // close refers to bid
                double closeAsk = candleData.getAskClose();
                Candlestick candlestick = new Candlestick(sdfDate, open, low, high, closeBid,
                closeAsk);
                candlesticksList.add(candlestick);
        }
}
```

In addition to adding a `closeAsk` field, we also change the `high` field price to ask instead of the , which makes more sense in the context of our strategy, as we will see in a bit.

Ok so now that we have made the proper adjustments to our relevant classes we can easily use the `calculateRsiForDataSet` method to calculated the CCI for each candle. The next step is to implement the logic of our strategy and we are going to do that by overriding the `runStrategy` method:

```
@Override
public StrategyResult runStrategy(List<Candlestick> candleSticksList) {
double entryBuy;
double entrySell;
double stopLossPrice=0;
double takeProfitPrice=0;
double strategyProfit=0;
double maxProfit=0;
double maxDrawdown=0;
boolean isOpenPosition=false;
```

```
                          //the method continues here…
int winCounter=0;
int lossCounter=0;
for (int i = 250; i < candleSticksList.size(); i++) {
if(isOpenPosition) {
if(stopLossPrice<takeProfitPrice) { // long position
if(candleSticksList.get(i).getLow()<stopLossPrice) {
isOpenPosition=false; // position closed at a loss lossCounter++;
strategyProfit-=STOP_LOSS; maxDrawdown=calculateMaxDrawdown(maxProfit, strategyProfit, maxDrawdown); }else
if(candleSticksList.get(i).getHigh()>takeProfitPrice) {
isOpenPosition=false; // position closed at a profit
winCounter++;
strategyProfit+=TAKE_PROFIT;
maxProfit=updateMaxProfit(strategyProfit,maxProfit);
}
}else{ // short position
if(candleSticksList.get(i).getHigh()>stopLossPrice) {
isOpenPosition=false; // position closed at a loss
lossCounter++;
strategyProfit-=STOP_LOSS;
maxDrawdown=calculateMaxDrawdown(maxProfit, strategyProfit, maxDrawdown);
}else if(candleSticksList.get(i).getLow()<takeProfitPrice) {
isOpenPosition=false; // position closed at a profit
winCounter++;
strategyProfit+=TAKE_PROFIT;
maxProfit=updateMaxProfit(strategyProfit,maxProfit);
}
}
}else if(candleSticksList.get(i).getRsi()<FLOOR_RSI){ // no open positions. check RSI
entryBuy=candleSticksList.get(i).getCloseAsk(); // use closeAsk for long position
stopLossPrice=((1-STOP_LOSS)*entryBuy);
takeProfitPrice=((1+TAKE_PROFIT)*entryBuy);
isOpenPosition=true;
}else if(candleSticksList.get(i).getRsi()>CEILING_RSI){ // no open positions. check RSI
entrySell=candleSticksList.get(i).getCloseBid (); // use close (bid) for short position
stopLossPrice=((1+STOP_LOSS)*entrySell);
takeProfitPrice=((1-TAKE_PROFIT)*entrySell);
isOpenPosition=true;
}
}
StrategyResult sr = new StrategyResult(strategyProfit, maxProfit, maxDrawdown, winCounter,
lossCounter, this);
return sr;
}
```

The logic of the strategy is implemented by the `runStrategy` method shown above, with the help of the 2 static methods `calculateMaxDrawdown` and `updateMaxProfit`:

```
private static double calculateMaxDrawdown(double maxProfit, double strategyProfit, double maxDrawdown){
double currentDrawdown = ((1+strategyProfit)-(1+maxProfit))/(1+maxProfit);
if(currentDrawdown<maxDrawdown) {
maxDrawdown=currentDrawdown;
}
return maxDrawdown;
}
private static double updateMaxProfit(double strategyProfit, double maxProfit) {
if(strategyProfit>maxProfit) {
maxProfit=strategyProfit;
}
return maxProfit;


}
```

Key notes for the logic of the strategy:

There are only 3 possible states at any given moment during the time of the strategy execution: either we have 1 open long position, 1 open short position or no open positions at all. There cannot be a situation where we have more than 1 open position.

The `runStrategy` method goes through the entire list of candlesticks and checks whether there is an open position. If there is, we need to check if we got stopped-out or if our profit target got hit. If there isn't, it means that we can open a new position provided that the asset's RSI is lower/higher than the floor/ceiling we defined when we instantiated the CCIIndicatorStrategy class. We set the stop-loss and take-profit immediately after we open a new position.

The price we use for opening a position is the appropriate closing price based on the side of the position: buy at ask and sell at bid. It is consistent with the calculation of the RSI which is based on the *closing* prices of the relevant asset. It also means that our assumption at the core of the strategy is that we can open a trade only at the end of each candle's time interval (1 minute in our case). Obviously this assumption helps simplify the backtesting process as we are limited to the data that each candlestick has, which sometimes doesn't tell the all story. In other words, there are times when the RSI breaches our floor or ceiling during the candle's time interval, but we cannot know the exact time it happened or the exact price of the asset at that time. We are limited to the open, close, low, high of each time interval. The bottom line is this: if we are consistent with the principals used for backtesting the strategy, when running the strategy on real time data, we can expect the results would be somewhat similar. But if we deviate from those principals, the results can vary considerably.

You've probably noticed that the *for* loop at the heart of `runStrategy` starts from 250 which looks a bit odd. The reason for this is that RSI is very volatile at first calculations as it takes time for the smoothing process to kick in (see the formula for more details). Therefore, it's considered good practice to use a minimum of 250 data points prior to the starting time of our backtesting.

The `runStrategy` method returns a StrategyResult instance with all of the relevant arguments (profit, maximum drawdown, etc.) plus an instance of the RsiSignals itself. We need this instance as it holds the values of the parameters used for that particular backtest i.e. stop-loss, take-profit, RSI values. We use it to analyze the results of each backtest, as we will see in the next section: "FXCM Java API – Testing Our Strategy".

Below is a full list of the classes we've used in this tutorial:
Strategy.java:

```
public interface Strategy {
StrategyResult runStrategy(List <Candlestick> candleStickList);
}
```

CCIIndicatorSrategy.class

```
public class CCIIndicatorStrategy implements Strategy {

   private double STOP_LOSS;
   private double TAKE_PROFIT;
   public static int FLOOR_CCI;
   public static int CEILING_CCI;
   public static final double FACTOR = 0.015;



   public CCIIndicatorStrategy(double sTOP_LOSS, double tAKE_PROFIT, int fLOOR_CCI) {
          super();
          STOP_LOSS = sTOP_LOSS;
          TAKE_PROFIT = tAKE_PROFIT;
          FLOOR_CCI = fLOOR_CCI;
          CEILING_CCI = 0 - fLOOR_CCI;
   }

   public static void calculateCciForDataSet(List<CandleStick> candleSticksList, int
duration){

          //Variable definitions
          double sum = 0;
          double typicalPriceInstance = 0;
          double smaOfTypicalPrices = 0;
          double cci = 0;
          double meanDeviation = 0;
          double sumOfDeviations = 0;
          List<Double>typicalPrices = new ArrayList<>();


          for (int j = 0; j < duration; j++) {
                //Creation of Class instances
                TypicalPriceIndicator typicalPriceIn = new TypicalPriceIndicator();
                SMAIndicator smaIndicator = new SMAIndicator();
                MeanDeviationIndicator meanDeviationIn = new MeanDeviationIndicator();
                CCIIndicator cciInd = new CCIIndicator();

                //Calculation of Typical Price
                typicalPriceIn.calculateTypicalPrice(candleSticksList, duration);

   typicalPrices.add(typicalPriceIn.calculateTypicalPrice(candleSticksList, duration));
```

```
                          //the method implementation continues here...

//Calculation of SMA of  Typical Prices
              smaOfTypicalPrices = smaIndicator.calculateSMA(typicalPrices, duration);
              meanDeviation = meanDeviationIn.calculateMeanDeviation(smaOfTypicalPrices,
typicalPrices, duration);

              //Calculation of CCI
              cci = cciInd.calculateCCI(typicalPrices, smaOfTypicalPrices,FACTOR,
meanDeviation, duration);

              //Set the CCI for each candle
              for (CandleStick candle : candleSticksList) {
                    candle.setCCI(cci);
              }
        }
  }
  @Override
  public StrategyResult runStrategy(List<CandleStick> candleSticksList) {

        double entryBuy;
        double entrySell;
        double stopLossPrice=0;
        double takeProfitPrice=0;
        double strategyProfit=0;
        double maxProfit=0;
        double maxDrawdown=0;
        boolean isOpenPosition=false;
        int winCounter=0;
        int lossCounter=0;


        for (int i = 250; i < candleSticksList.size(); i++) {

              if(isOpenPosition) {
                    if(stopLossPrice<takeProfitPrice) {
                          if(candleSticksList.get(i).getLow()<stopLossPrice) {
                                isOpenPosition=false;
                                lossCounter++;
                                strategyProfit-=STOP_LOSS;
                                maxDrawdown=calculateMaxDrawdown(maxProfit,
strategyProfit, maxDrawdown);
                          }else if(candleSticksList.get(i).getHigh()>takeProfitPrice) {
                                isOpenPosition=false;
                                winCounter++;
                                strategyProfit+=TAKE_PROFIT;
                                maxProfit=updateMaxProfit(strategyProfit,maxProfit);
                          }
                    }else {
                          if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                isOpenPosition=false;
                                lossCounter++;
                                strategyProfit-=STOP_LOSS;
```

```
                 //the method implementation continues here...
                   }else if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                     isOpenPosition=false;
                                     winCounter++;
                                     strategyProfit+=TAKE_PROFIT;
                                     maxProfit=updateMaxProfit(strategyProfit, maxProfit);
                               }
                        }
                        //System.out.println(candleSticksList.get(i).getCCI() + "THis is the
CCI for the FLOOR");
                 }else if(candleSticksList.get(i).getCCI()<FLOOR_CCI) {
                        entryBuy=candleSticksList.get(i).getCloseAsk();
                        stopLossPrice= ((1-STOP_LOSS)*entryBuy);
                        takeProfitPrice = ((1+TAKE_PROFIT)*entryBuy);
                        isOpenPosition=true;
                 }else if(candleSticksList.get(i).getCCI()>CEILING_CCI) {
                        entrySell=candleSticksList.get(i).getCloseBid();
                        stopLossPrice=((1+STOP_LOSS)*entrySell);
                        takeProfitPrice=((1-TAKE_PROFIT)*entrySell);
                        isOpenPosition=true;
                 }
          }
          StrategyResult sr = new StrategyResult(strategyProfit, maxProfit, maxDrawdown,
winCounter, lossCounter, this);

          return sr;
    }

    private double updateMaxProfit(double strategyProfit, double maxProfit) {

          if(strategyProfit>maxProfit) {
                 maxProfit=strategyProfit;
          }

          return maxProfit;
    }

    private double calculateMaxDrawdown(double maxProfit, double strategyProfit, double
maxDrawdown) {

          double currentDrawdown = ((1+strategyProfit) - (1+maxProfit))/(1+maxProfit);

          if (currentDrawdown<maxDrawdown) {
                 maxDrawdown=currentDrawdown;
          }

          return maxDrawdown;
    }
```

CCiIndicator.class

```
public class CCIIndicator {

protected double calculateCCI(List<Double> listOfTypicalPrices, double
averageOfTypicalPrices, double constant, double meanDeviationOfTypicalPrices, int
duration) {
                double cci = 0;
                for (int k = 0; k < duration; k++) {
                        cci = (listOfTypicalPrices.get(k) -
averageOfTypicalPrices/(constant*meanDeviationOfTypicalPrices));
                        return cci;
                }
                return 0;
        }


}
```

TypicalPriceIndicator.class

```
public class TypicalPriceIndicator {

     protected double calculateTypicalPrice(List<CandleStick> candleSticksList, int
index) {
             double typicalPrice;
             double high = 0;
             double low = 0 ;
             double closeBid = 0;
             for (int j = 0; j < index; j++) {
                     high = candleSticksList.get(j).getHigh();
                     low  = candleSticksList.get(j).getLow();
                     closeBid = candleSticksList.get(j).getCloseBid();
                     typicalPrice = (high + low + closeBid)/3;
                     return typicalPrice;
             }
             return 0;
     }
}
```

SMAIndicator.class

```
public class SMAIndicator {

     protected Double calculateSMA(List<Double> listOfPrices, int duration){
             double smaOfTypicalPrices =
(listOfPrices.stream().mapToDouble(Double::doubleValue).sum())/duration;
             return smaOfTypicalPrices;
     }
}
```

```
MeanDeviaion.class

public class MeanDeviationIndicator {

	protected double calculateMeanDeviation(double average, List<Double> 
listOfPrices, int duration) {
		double sumOfDeviations = 0;

		for (int j = 0; j < duration; j++) {
			 sumOfDeviations += Math.abs(average-listOfPrices.get(j));
			double meanDeviation = sumOfDeviations/duration;
			return meanDeviation;
		}
		return 0;
	}
}
```

HistoryMiner.java (Updated. See “FXCM Java API – Pulling Historical Data” for more details):

```
                                    // rest of HistoryMiner class...
                                  // updated part:
public void convertHistoricalRatesToCandleSticks(){
SortedSet<UTCDate> dateList = new TreeSet<>(historicalRates.keySet());
SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yyyy HH:mm");
sdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));
for(UTCDate date : dateList){
MarketDataSnapshot candleData;
candleData = historicalRates.get(date);
// convert the key to a Date
Date candleDate = date.toDate();
String sdfDate = sdf.format(candleDate);
double open = candleData.getBidOpen();
double low = candleData.getBidLow();
double high = candleData.getAskHigh(); // ask is the relevant price for short positions
double closeBid = candleData.getBidClose(); // close refers to bid
double closeAsk = candleData.getAskClose();
Candlestick candlestick = new Candlestick(sdfDate, open, low, high, closeBid, closeAsk);
candlesticksList.add(candlestick);
}
}
public void displayHistory() {
if(candlesticksList.size()<1) {
System.out.println("No data do display");
return;
}
System.out.println("Date\t Time\t\tOpen\tHigh\tLow\tClose");
for (Candlestick candlestick : candlesticksList) {
System.out.println(
candlestick.getDate() + "\t" +
candlestick.getOpen() + "\t" +
candlestick.getHigh() + "\t" +
candlestick.getLow() + "\t" +
candlestick.getCloseBid()); // the close bid for the candle
}
}
// end of updated part.
// rest of HistoryMiner class...
```