Contents

[Introduction**:** 1](#_Toc473209300)

[The spec: 1](#_Toc473209301)

[Stock calculations: 1](#_Toc473209302)

[Index Calculations: 1](#_Toc473209303)

[Trade Cache: 2](#_Toc473209304)

[Other conditions: 2](#_Toc473209305)

[Thoughts & Questions: 2](#_Toc473209306)

[GOAL: 2](#_Toc473209307)

[Design Considerations: 2](#_Toc473209308)

# Introduction**:**

## The spec:

The spec is to write the core module for GBC.

Assets/Instruments: We have two types of stocks (Common & Preferred),

Index: a volume weighted index of stocks,

We need

1. A Stock calculations module
2. An Index level computing module
3. A trade recording module ( includes a feature to fetch trades , which will be used by i) and ii)

## Stock calculations:

1. Dividend Yield:
   1. CommonStock: lastDividend / Price
   2. PreferredStock: fixedDividend/ Price
2. P/E Ratio: Price / Dividend
3. VWSP: Remains same for both types of stocks. Need to get trade records for the last five minutes.

## Index Calculations:

GBC Index:

* consists of **all** shares
* GM of vol weight stock prices ( Item iii. In Stock Calculations )

## Trade Cache:

* Record symbol, timestamp, quantity/volume, buy/sell indicator and price
* For a given symbol(s) fetch volume, price in the last 5 minutes.

## Other conditions:

No database, GUI or I/O is required, all data need only be held in memory

# Thoughts & Questions:

Given this has no db, GUI or I/O, the delivery will be a module / package with the above features and corresponding unit tests.

What are the limits of the #stocks ?

What are the limits of the no. of trades to record ?

At what rate do the requests come in ?

How many concurrent threads use this module at a given time ?

We cannot pretty much assume anything, except that the performance of module should only tied to the availability of the resources on the host.

It is a likely fact that the no. of trades are expected to be much greater than the no. of stocks, however this should not be driving the design.

# GOAL:

The goal then is to design and build the most performant module limited only by the environment/host resources. Given we are not asked to provide anything extra than what is asked, focus is to get the best computing module which will satisfy all the test cases. We need provide a concurrent API to register stocks, record trades and get Stock and Index calculations.

# Design Considerations:

**Stocks:**

We know there are two types of stocks, inheritance.. may be add an extra layer..as stock is a financial asset.. could have an inheritance like

<-- PreferredStock

base class Asset <-- Financial Asset <-- Stock

<-- CommonStock

Let’s do it assuming no financial knowledge.. this would simply be

<-- PreferredStock

Abstract Base Stock

<-- CommonStock

Abstract Base Stock will have final members Symbol, Last Dividend, Fixed Dividend, Par Value.

Abstract methods: getDividendYield, getPERation, getVWPrice

Concrete Implementations sit in PreferredStock and CommonStock

However, this inheritance begs the question.. will the subclass, passed as Stock be treated as immutable? I guess to simplify it and make Stock final, and pass around the instances between threads I would do away with inheritance. Just have class Stock.

Have a singleton registry for stocks.

**Record Stocks**

This I think is the key bit…. We want to have a simple cache to record trades. We know we are only doing inserts, so there is no delete and reclaim memory operations. However we need to traverse through trades recorded for each stock based on the timestamp.

What is the cache going to be? A map obviously comes to mind.

What should be the key? Should it be the symbol name or a reference to Stock?

If I use a Stock as a key , when creating new Stocks, all I have to add is an entry into the map with Stock and valueType, as opposed to having two data structures, one for holding the list of stocks the other being the cache map.

Value Type: We do need a container to hold trades for a given Stock. On this container

* We insert and read concurrently
* We do not do deletes
* We want fast access to elements
* Preferably sorted based on timestamp on the TradeRecord object.
* Preferably a lock-free implementation (multiple reads and inserts)
* Cannot be a queue, as I want serial access.
* Allow duplicates as I can have two trades with everything being same.. including timestamps
* Do we want a consistent view

A very simplistic view can be a List .. that will take care of insertions.. what about the performance of all the trades in the last five minutes… ? We either sort at every insertion.. which is costly…What about **ConcurrentSkipListSet** (with Trades having a unique int – because ConcurrentSkipListSet does not allow duplicates) or Guava’s **ConcurrentHashMultiSet** .. either is good for guaranteeing the order…however the iterators are not always consistent with the underlying list.. they are weakly consistent. Good bit is iterators don’t throw concurrent modification exception.

Record trades into a lockfree map possibly - <Stock, container<StockTrade>>

Using NonBlockingHashMap by Cliff Click.

**Index:**

This a composite computation of all vwaps based on the incoming timestamp.

I have developed a project in Eclipse towards this goal. All the functionality is exposed through GBCCoreStaticInterface.