



# **Java: How low can you go**

**Low-Latency Design for RFQ and  
High-Frequency Trading –  
Covering Java 24+ and Beyond**

**Zahid Hossain**

# Java: How low can you go.

*Low-latency design for RFQ and high-frequency trading – covering Java 24+ and beyond.*

**Zahid HOSSAIN**





## **Java: How low can you go.**

Copyright © 2025.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the author and publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor the publisher, its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

First published in 2025.

*This book is dedicated to **Quanteam UK**, whose inspiration and sponsorship made it possible. Quanteam UK is a consultancy firm specializing in the Financial Markets industry.*

— Zahid HOSSAIN

## **Contributors**

### **About the author**

Zahid HOSSAIN is an entrepreneur, software engineer, and technology leader with over 18 years of experience in e-trading, quantitative development, and building low-latency trading systems at Charles River / State Street, Jefferies, Citi, Credit Suisse, Barclays Investment Bank, and Bloomberg, in London and New York. He has a strong educational background, holding both a Master's and a Bachelor's degree in Computer and Electrical Engineering where he graduated top of his class.

Throughout his career, he has been recognized multiple times as an exceptional performer for his contributions to high-impact technology initiatives. Currently based in London, Zahid specializes in low-latency Java development, and he has a passion for algorithms, data structures, financial technology, and AI.

### **About the reviewers**

Antoine DUCHER, founder and CEO of Quanteam UK, a London-based consultancy dedicated to financial markets and quantitative modelling. Antoine brings over 20 years of experience in capital markets and consulting.

Eugenie STEELE, Head of Marketing at Quanteam UK. Eugenie has over 18 years of experience in marketing, having previously worked for BNP Paribas and RBC BlueBay Asset Management, both in France and the UK.

## ***TABLE OF CONTENTS***

Preface .....	13
Summary .....	13
Interview Questions.....	14
References .....	15
Java Types .....	16
Primitive Types .....	16
boolean .....	18
Wrapper Types in Java.....	24
JVM Optimizations in Wrappers .....	27
JVM caching Mechanism for wrapper types .....	29
String Type .....	33
JVM Optimization Around String.....	35
Streamlined Deduplication Process:.....	37
Record.....	41
Some strategies for achieving low latency. ....	42
Type Checking and Arrays in Low Latency.....	43
Conclusion .....	50
Java Collections and Their Underlying Mechanics.....	51
HashMap .....	52
JVM HashMap Optimization .....	54
Resizing Issue .....	55
ConcurrentHashMap .....	56
ConcurrentHashMap Synchronization Tricks.....	58

HashMap Vs ConcurrentHashMap Resizing .....	60
Fail-Fast vs. Fail-Safe Iteration .....	62
Internal Mechanics of an ArrayList.....	64
Low Latency Application's Hot Path.....	66
CopyOnWriteArrayList .....	67
Here are some Tricky Interview Questions on collections ....	70
Threading .....	74
JVM Thread Deep Dive .....	74
ForkJoinPool vs ThreadPoolExecutor .....	77
Virtual Threads .....	79
ForkJoinPool in JDK.....	87
Thread Pinning In JVM/Java (CPU Affinity Explained) .....	90
Thread Priority .....	92
Some Interview Questions .....	93
Synchronization.....	97
How do they work .....	97
Biased Locking.....	104
Reader Writer Lock.....	106
Compare-And-Swap (CAS).....	109
CAS vs ReentrantLock Performance .....	112
Low Latency Synchronization Test .....	117
Some Tricky Interview Questions .....	118
The Evolution of Garbage Collection .....	121
The First People.....	121



The Reason for G1GC .....	122
Shenandoah Garbage Collection: The Brooks Forwarding Pointer and Regioned Heap .....	125
Z Garbage Collection: Colored Pointers and Concurrent Relocation .....	130
Azul's C4 Garbage Collection: Hardware Concurrency .....	136
Typical Pause Times of Major Java Garbage Collectors .....	140
Epsilon GC .....	141
No-GC Off-Heap Storage .....	144
Eliminate All Transient Allocations .....	144
Utilize Off-Heap Memory or Direct Buffers .....	145
Pre-allocate Object .....	146
Java's built-in collections (HashMap, ArrayList, etc.) .....	146
Reuse Charset Encoders and StringBuilders .....	146
Use Pooled Network Buffers .....	147
Avoid Lambdas and Anonymous Classes in Hot Code .....	147
Avoid String Creation and Exception Throwing .....	148
GC free, Lock free Ring Buffer .....	149
Conclusion .....	151
Java Recent Projects .....	153
Valhalla .....	153
Leydon .....	153
Babylon .....	154
Panama .....	154

Loom.....	154
Project Leydon: The JVM's Speculative Nature.....	155
Project Babylon: Code Reflection and the GPU Future of Java .....	157
Project Amber and Low latency Application .....	161
Azul's Zing in HFT .....	162
Trading Systems .....	164
RFQ System Architecture .....	166
Market Making Architecture .....	170
How Exchanges match orders .....	173
Darkpool.....	177
Smart Order Router .....	185
Algorithmic Execution Strategies .....	189
VWAP - Volume Weighted Average Price .....	190
TWAP - Time Weighted Average Price .....	192
POV Participation of Volume .....	194
Client Order Lifecycle - Combining the Parts .....	195
Ultra Low Latency Code .....	199
What is ultra-low latency .....	199
Low latency business in finance .....	200
Programming Language .....	201
Challenges of Java and other consideration .....	202
Low Latency libraries in Java .....	203
Zero Copy .....	204

CPU and Memory consideration .....	206
Method Inlining .....	213
Loop unrolling .....	215
Escape analysis .....	217
Branch Prediction.....	217
SIMD operation (Java 24+) .....	220
Thread Pinning (CPU Affinity) .....	223
General Optimizations for Low-Latency Systems .....	225
OS (LINUX) Tuning.....	229
The importance of OS tuning .....	229
The concept of Jitter.....	230
Measure Before You Tune.....	231
Red Hat Linux – Tuned Profiles.....	231
NUMA Awareness.....	233
BIOS Configuration .....	236
CPU Isolation .....	239
Cage your GC Thread .....	243
Avoiding Interference (De-Jittering) .....	244
Huge Pages — Making Memory Access Faster .....	245
Kernel Timer Tick (nohz_full) .....	246
Conclusion .....	249
Market Data & Pricing Engine .....	251
Market Data – UDP vs TCP .....	252
Market Data Format.....	254

Architecture of Market Data – Pricing Engine .....	255
Conclusion .....	265
Network Tuning .....	267
Typical Packet Flow .....	268
Kernel Overhead .....	270
Kernel Bypass .....	271
Interrupt mode vs Poll Mode .....	272
Smart NIC cards.....	274
Linux Tuning.....	274
Conclusion .....	277
KDB+ A Low-Latency Database .....	280
High-frequency Trading.....	281
KDB+ Architecture .....	282
KDB+ processes.....	284
Why KDB+ is fast .....	285
How Compression Works in kdb+ .....	286
Understanding the Q Language .....	288
Conclusion .....	291
INDEX .....	292

# ***PREFACE***

## SUMMARY

This book reflects many years of building and tuning low-latency trading systems on the premises of some of the most prominent investment banks in the world. It includes firsthand engineering decisions and production environments where every microsecond counts. For most of my professional life, I have straddled the domains of Java performance, electronic trading, and system architecture. This book, or rather the subsequent chapters, sets out to teach the readers the advanced features of Java types, memory layouts, JVM optimizations and how the types like wrappers, primitives, records and strings work internally along with insight into JIT compiler and the memory model. It uses practical case studies to demonstrate the principles of bounded garbage, predictable performance in latency sensitive environments, reliable, and cache aware.

The subsequent chapter delves into the core libraries in addition to the concepts of threading and synchronization, detailing the operational mechanics of collections, the processes during resizing, and the significance of fail-fast and fail-safe practices in overload conditions. Then, the book describes the use of concurrency mechanisms in Java from biased locking and CAS to modern virtual threads along with CPUs pinned to threads. These themes are augmented by the history of garbage collection from G1GC to Shenandoah, ZGC, and Azul's C4, along with techniques to remove GC pauses through off-heap and zero-copy methods. Every chapter provides an equilibrium of theory and low-level practical tuning, supported by reasoning from benchmarks.

The final chapters unify the theory underpinning Java performance with actual trading infrastructure followed by OS and network tuning. It describes the processes of order matching in addition to the architecture of RFQ and market-making systems, the execution of market algorithms such as VWAP and TWAP on low-latency stacks, and the structuring of algorithmic strategies, Smart Order Router (SOR) etc. It has demonstrated consistent performance through OS-level tuning, NUMA, CPU isolation, and network optimization with Kernel bypass. The book concludes with an outline of Java code optimization and an integrated perspective on JVM systems competing in high-frequency and algorithmic trading where every microsecond is pivotal to profit-making.

## INTERVIEW QUESTIONS

This volume contains more than just system design, it also contains hundreds of real interview questions alongside intellectual exercises - not fictitious enigmas, rather questions I have asked, and I have been asked, during my decades of employment mentoring and hiring global engineering trading teams. Engineers trying the challenging field of low-latency systems will find these questions valuable, as will interviewers seeking to assess a candidate's real-world capability. Each question touches on a real-life design problem, system performance obstacle, or JVM instance that is observable in production. This book tries to equip the readers with practical insights that only come with years spent in highly demanding, low-latency, and high-pressure work situations in addition to technical knowledge.

While writing this book, I undertook some additional research on the topics, including watching several YouTube lectures. I found that some of these sources had already conducted similar types of benchmark tests that I aimed to conduct. To avoid unnecessary duplication of work, I have appropriately integrated those benchmark findings. I have transformed some diagrams in the book based on similar ideas found in those lectures, although I have changed them to better reflect my explanations by adding or removing elements. I would like to thank these creators for their valuable contributions to the low latency community. Lastly, the title of this book was inspired by one of the lectures that particularly resonated with me. For those who wish to pursue these materials, I have collected the list of videos below.

Understanding the Disruptor

<https://www.youtube.com/watch?v=DCdGlxBbKU4>

Kernel-bypass techniques for high-speed

<https://www.youtube.com/watch?v=MpjlWt7fvrw>

Ultra-low latency Java in the real world - Daniel Shaya

<https://www.youtube.com/watch?v=BD9cRbxWQx8>

Low Latency Market Data

<https://www.youtube.com/watch?v=y-BSb045LNk>

Network Performance Tuning

<https://www.youtube.com/watch?v=ZYCKSN4xf84>

# ***JAVA TYPES***

The most central part of the Java type system is its small set of primitive types, which form the foundation of all data structures. In Java, any variable that is declared must first be given a type, which is a form of pact. This pact defines the total amount of memory allocated, the maximum and minimum values that can be stored, and the functions that can be performed on the variable.

Since Java was developed for use on any type of hardware, the size of each primitive type is bound by the language specification itself. This is determined by the Java Virtual Machine, not by the processor or the operating system. A 32-bit int is thus, and for example, exactly 32 bits on a desktop Java Virtual Machine (JVM) and on an embedded controller. This was one Java's main selling points back in the day, the concept of write once, run anywhere.

## PRIMITIVE TYPES

Primitive types are not objects and exist in system memory as opposed to the heap. They do not possess method tables, identity, headers, or any other form of metadata. Primitive types are composed of pure values, which is the reason as to why they are extremely fast and compact. Advanced Java programmers are thus able to perform any numeric operation and use loops or even counters without any form of reduced performance by using primitive types instead of wrapper classes.

Unlike other programming languages, Java is unique in the sense that it chooses to categorize its primitives into two main categories: numeric and non-numeric.



The numeric types represent numbers as integers or in floating-point formats, whereas types that are not numeric manage logic and character data.

The types of integers are byte, short, int and long. They vary only in the number of bits and range of value each can contain. The decision of which one to use is usually between precision or memory. For example, byte is appropriate for the storing of compact data such as in the case of network protocols or file I/O buffers whereas int is the general-purpose choice for counters, indices and most arithmetic.

The floating-point formats, float and double, follow the IEEE 754 standard of representation of real numbers. They can express fractions, special values such as infinities and even NaN - an outcome of mathematical operations which are undefined. In double, however, is the dominating type in modern Java due to the higher precision and only moderately higher storage required.

The char type is the only one that is 16 bits and contains one Unicode code unit. Java unlike older languages such as C, does not consider characters as bytes. Java characters are global since they can cover accented letters and other scripts that are not part of ASCII.

A boolean holds the value of true or false and is the basis of every conditional as well as every logical test. While its internal representation is specific to the JVM, language documentation states that it acts as a one-bit logical flag.

In addition to the aforementioned types, there is a unique type called **returnAddress** which exists only in the JVM's bytecode. It is used by the VM to keep track of the return points of methods, and as such, never shows up in Java source code.

VWAP (Volume-Weighted Average Price): Consider a **Trade** table

time timestamp  
sym symbol  
price float  
size int

the basic VWAP for each symbol is:

```
select vwap: sum price * size % sum size by sym from trade
```

Output may look like this:

sym	vwap
AAPL	150.146
MSFT	305.553

A time-bucketed VWAP (for example, 1-minute intervals) is just as straightforward:

```
select vwap: sum price * size % sum size by sym, 1 xbar time.minute  
from trade
```

The expression 1 xbar groups trades into 1-minute buckets before calculating VWAP within each group.

Q functions as the query and control language, which manages how data is loaded, transformed, stored and queried, now or in the past, when working with kdb+. Coming running over off the heels of the data load, q's ability to run directly on columnar memory arrays and shun unnecessary overheads gives developers the chance to write lightning-fast analytics code with the absolute minimum amount of code.

The fusion of language, engine and storage model in kdb+ is the secret to its success in providing extremely low-latency, high-volume time-series analytics.

## CONCLUSION

Speed is not just an advantage, it's basically a necessity, when working in the high-stakes world of high-frequency trading and electronic markets. Financial firms count on databases that can seamlessly keep up with the torrent of live market data, crank out microsecond-level analytics, and deliver reliable results even under the most extreme loads.

Well-known as the de-facto standard in major banks and trading companies, kdb+ has become the go-to for all these reasons. Its in-memory, column-structured layout, merged with the expressiveness of the q language, allows teams to cut through the noise of market ticks, compute rolling statistics and dissect risk in real time without hitting the brakes on the trading engine. Today, numerous global banks, market makers, and quant desks use kdb+ as the anchor of their analytics stacks.

In my time at Citi working in Equity Derivatives, kdb+ was at the heart of real-time monitoring of Greeks, intraday risk and market-sensitive signals and I have found that its low-latency intake, lightning-fast vectorised queries and efficient storage make it one of the handful of systems that can handle the scale and speed needed in modern low-latency application.

# INDEX

:+PrintInlining, 210  
+AlwaysPreTouch, 242  
Aeron, 143, 145, 163, 179, 181, 192, 194, 200, 201, 202  
Affinity, 90  
Agrona, 65, 84, 143, 145, 163, 180, 200, 201, 202  
Algorithmic Execution Strategies, 186  
Anonymous Classes, 145  
ArrayList, 64  
Autoboxing, 29  
Babylon, 151  
Biased Locking, 104  
BigDecimal, 24  
BIOS, 225, 227, 232, 233  
BitSet, 21  
boolean, 17, 19, 20, 21, 22, 26, 41, 68, 106, 109, 110, 216  
Boolean, 19, 26, 27, 30  
boxing, 40, 51, 65, 144, 223  
Branch Prediction, 214  
C4 Garbage Collection, 134  
Cache filling, 207  
CAS, 13, 56, 58, 59, 60, 67, 98, 99, 103, 109, 110, 112, 113, 117, 118, 222  
Challenges of Java, 199  
Chronicle Queue, 181, 200, 202  
Citi, 177, 186  
Client Order Lifecycle, 192  
Compare-And-Swap, 109  
Concurrent Mark-Sweep, 121  
ConcurrentHashMap, 56  
contiguous memory, 44, 46, 204  
Continuous Compaction, 136  
CopyOnWriteArrayList, 67, 69, 70, 72, 73  
**CountdownLatch**, 114

CPU Affinity, 90, 219  
CPU Isolation, 235  
Credit Suisse, 177, 186, 263  
Crossfinder, 177  
C-states, 233  
**CyclicBarrier**, 114, 115  
dark pool, 161, 174, 175, 176, 177, 178, 179, 180, 182, 184, 193  
Darkpool, 174, 178  
Deduplication, 37, 38  
De-Jittering, 240  
Direct Buffers, 143  
Drop Copy, 185, 186, 193, 194  
Encoders, 144  
Epsilon GC, 140  
Escape analysis, 213, 214  
Exception Throwing, 146  
Fail-Fast, 62  
Fail-Safe, 62  
**Fairness**, 102  
False sharing, 95  
*FixTag*, 40  
ForkJoinPool, 77, 78, 79, 80, 86, 87, 88, 89, 90  
Forwarding Pointer, 124, 125  
G1GC, 121  
Garbage Collection, 120  
GC free, 147  
GC overhead, 31  
Goldman Sachs, 178  
HashMap, 33, 52, 53, 54, 55, 56, 60, 61, 62, 63, 71, 72, 106, 144, 222  
HashMap Optimization, 54  
Hot Path, 66  
Huge Pages, 241, 242  
IRQ affinity, 219, 228, 273  
isolcpus, 235, 236, 237, 244, 245  
Java Collections, 25, 51

- Jitter, 92, 226, 240
- JVM caching, 29
- JVM Optimization, 35, 103
- JVM Optimizations, 28
- JVM Parameters, 36
- JVM's Speculative Nature, 152
- Kernel Timer Tick, 242
- Lambdas, 145
- Lead Market Maker, 173
- Leydon, 150
- LMAX, 147, 148, 163
- Load Barriers, 135
- Lock Acquisition, 102
- Lock flag**, 98
- Lock free, 147
- Lock optimizations, 152
- Loom, 151
- Loop optimizations, 152
- Loop unrolling, 212
- Low latency business, 197
- Low Latency libraries, 200
- Mark word, 23, 24, 98, 127
- Market Data, 247
- Market Making, 162, 167, 168
- match orders, 170
- Method inlining, 152
- Method Inlining, 210
- Midpoint Calculation, 180
- modCount, 62
- NBBO, 175, 177, 180, 181
- No-GC, 142
- nohz\_full, 245
- NUMA Awareness, 229
- NUMA node, 229, 230, 232
- NUMA server, 232
- Off-Heap Memory, 143

- OpenHFT**, 91
- OTC derivatives, 163
- Panama, 151
- Polluted data, 254
- POV, 184, 186, 187, 191, 192, 193, 194, 195
- Predictable Performance, 123
- Pricing Engine, 247
- Primitive Types, 16
- Pro Rata, 171
- P-states, 233
- Question**, 21, 22, 23, 24, 29, 31, 32, 33, 34, 38, 46, 47, 48, 49, 55, 57, 60, 65, 82, 83, 86, 88, 93, 94, 95, 97, 100, 102, 106, 108, 113, 114
- Reader Write Lock, 106
- record*, 41, 113, 158, 159, 205
- Red Hat Linux, 227, 243
- Reentrant Lock**, 99
- ReentrantLock, 112
- Reporting and Compliance, 181
- Request for Quote, 162
- Resizing Issue, 55
- RFQ, 2, 14, 85, 140, 162, 163, 164, 165, 201, 207, 221
- Ring Buffer, 147
- Shenandoah, 124
- SIMD, 36, 66, 71, 151, 157, 160, 199, 213, 217, 218, 219
- Smart Order Router, 14, 182, 183, 187, 193, 194
- SOR, 183, 184, 187, 189, 190, 193, 194
- stamped Lock, 113
- String Type, 33
- Synchronization, 97
- Synchronization Tricks, 58
- synchronize**, 97
- System Architecture, 163
- System.arraycopy, 44, 45, 66, 71
- TCP, 248
- Thread Pinning, 90, 219

- Thread Priority, 92
- Threading, 74
- ThreadPoolExecutor, 78
- Time Weighted Average Price, 189
- Tuned Profiles, 227
- Turbos Boost, 233
- TWAP, 14, 186, 187, 189, 190, 194, 195
- Type Checking, 43
- UBS ATS, 178
- UDP vs TCP, 248
- Ultra Low Latency, 196
- ultra-low latency, 140, 160, 196, 225, 245, 268, 269, 271
- unboxing, 27, 29, 51, 223
- UNIX tunning, 220
- UnlockExperimentalVMOptions, 141
- UseEpsilonGC, 141
- Valhalla, 150
- Value Class, 39
- Virtual Threads, 79
- Volume Weighted Average Price, 187
- VWAP, 14, 164, 184, 186, 187, 188, 189, 190, 192, 193, 194, 195
- Warm-up, 207
- Wrapper Class**, 26
- Wrapper Types, 25
- XX
  - +OptimizeStringConcat, 37
  - +UseStringDeduplication, 37
  - MaxInlineSize, 211
  - StringTableSize=<entries>, 37
- XX:+PrintStringTableStatistics, 36
- XX:+UseLargePages, 242
- XX:AutoBoxCacheMax, 30
- XX:MaxInlineSize, 211
- Z Garbage Collection, 128
- Zero Copy, 201
- Zero-GC, 142



Zing, 139, 159, 213