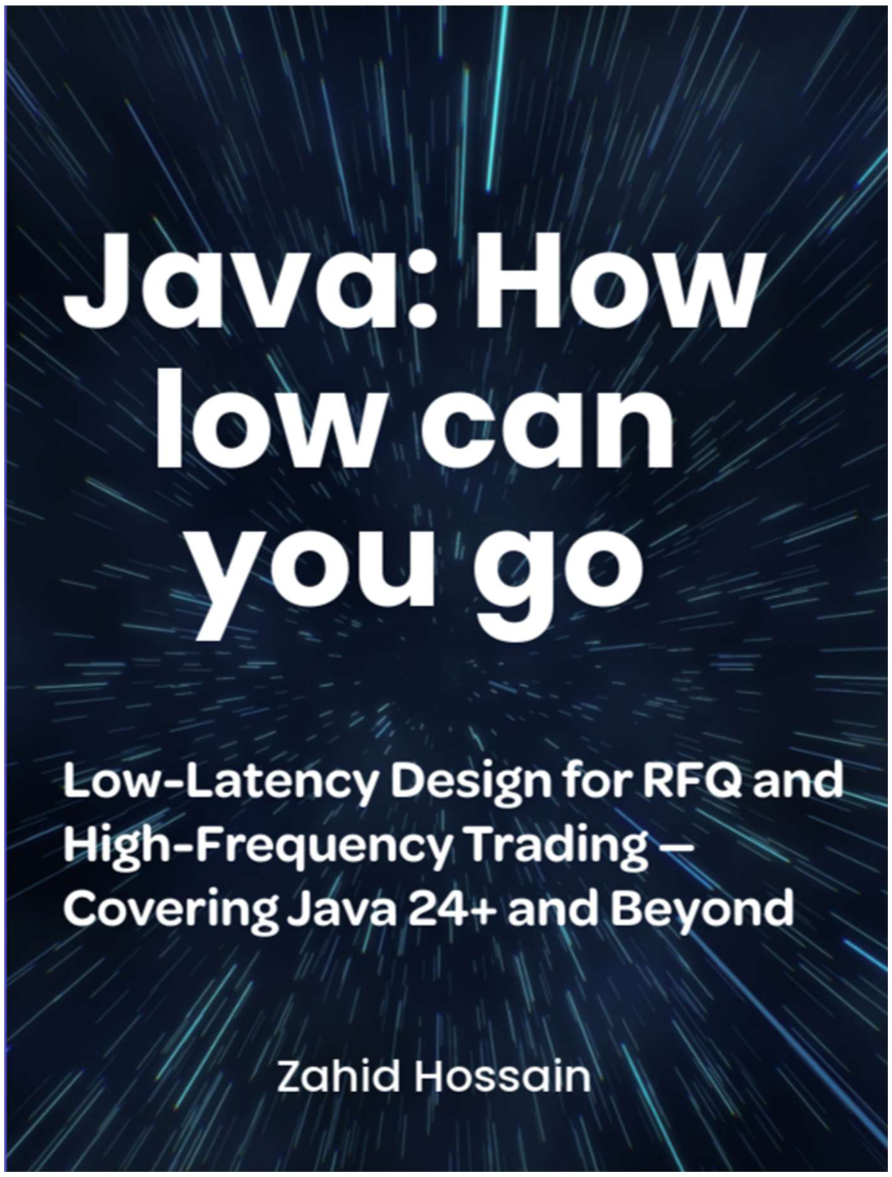

Java: How low can you go

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

QUANTEAM UK

Java: How low can you go.

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

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.

References

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