ORACLE PRESS

Java™

JVM Performance Engineering

Inside OpenJDK and the HotSpot Java Virtual Machine

ORACLE

Monica Beckwith

FREE SAMPLE CHAPTER

JVM Performance Engineering

This page intentionally left blank

JVM Performance Engineering

Inside OpenJDK and the HotSpot Java Virtual Machine

Monica Beckwith

Addison-Wesley

Hoboken, New Jersey

Cover image: Amiak / Shutterstock
Figures 7.8–7.18, 7.22–7.29: The Apache Software Foundation

Library of Congress Control Number: 2024930211

ISBN-13: 978-0-13-465987-9
ISBN-10: 0-13-465987-2

$PrintCode

To my cherished companions, who have provided endless inspiration and comfort throughout the journey of writing this book:

In loving memory of Perl, Sherekhan, Cami, Mr. Spots, and Ruby. Their memories continue to guide and brighten my days with their lasting legacy of love and warmth.

And to Delphi, Calypso, Ivy, Selene, and little Bash, who continue to fill my life with joy, curiosity, and playful adventures. Their presence brings daily reminders of the beauty and wonder in the world around us.

This book is a tribute to all of them—those who have passed and those who are still by my side—celebrating the unconditional love and irreplaceable companionship they have graciously shared with me.

This page intentionally left blank

Contents

Preface

Welcome to my guide to JVM performance engineering, distilled from more than 20 years of expertise as a Java Champion and performance engineer. Within these pages lies a journey through the evolution of the JVM—a narrative that unfolds Java's robust capabilities and architectural prowess. This book meticulously navigates the intricacies of JVM internals and the art and science of performance engineering, examining everything from the inner workings of the HotSpot VM to the strategic adoption of modular programming. By asserting Java's pivotal role in modern computing—from server environments to the integration with exotic hardware—it stands as a beacon for practitioners and enthusiasts alike, heralding the next frontier in JVM performance engineering.

Intended Audience

This book is primarily written for Java developers and software engineers who are keen to enhance their understanding of JVM internals and performance tuning. It will also greatly benefit system architects and designers, providing them with insights into JVM's impact on system performance. Performance engineers and JVM tuners will find advanced techniques for optimizing JVM performance. Additionally, computer science and engineering students and educators will gain a comprehensive understanding of JVM's complexities and advanced features.

With the hope of furthering education in performance engineering, particularly with a focus on the JVM, this text also aligns with advanced courses on programming languages, algorithms, systems, computer architectures, and software engineering. I am passionate about fostering a deeper understanding of these concepts and excited about contributing to coursework that integrates the principles of JVM performance engineering and prepares the next generation of engineers with the knowledge and skills to excel in this critical area of technology.

Focusing on the intricacies and strengths of the language and runtime, this book offers a thorough dissection of Java's capabilities in concurrency, its strengths in multithreading, and the sophisticated memory management mechanisms that drive peak performance across varied environments.

Book Organization

Chapter 1, "The Performance Evolution of Java: The Language and the Virtual Machine," expertly traces Java's journey from its inception in the mid-1990s to the sophisticated advancements in Java 17. Highlighting Java's groundbreaking runtime environment, complete with the JVM, expansive class libraries, and a formidable set of tools, the chapter sets the stage for Java's innovative advancements, underlying technical excellence, continuous progress, and flexibility.

Key highlights include an examination of the OpenJDK HotSpot VM's transformative garbage collectors (GCs) and streamlined Java bytecode. This section illustrates Java's dedication to performance, showcasing advanced JIT compilation and avant-garde optimization techniques. Additionally, the chapter explores the synergistic relationship between the HotSpot VM's client and server compilers, and their dynamic optimization capabilities, demonstrating Java's continuous pursuit of agility and efficiency.

Another focal point is the exploration of OpenJDK's memory management with the HotSpot GCs, particularly highlighting the adoption of the "weak generational hypothesis." This concept underpins the efficiency of collectors in HotSpot, employing parallel and concurrent GC threads as needed, ensuring peak memory optimization and application responsiveness.

The chapter maintains a balance between technical depth and accessibility, making it suitable for both seasoned Java developers and those new to the language. Practical examples and code snippets are interspersed to provide a hands-on understanding of the concepts discussed.

Chapter 2, "Performance Implications of Java's Type System Evolution," seamlessly continues from the performance focus of Chapter 1, delving into the heart of Java: its evolving type system. The chapter explores Java's foundational elements—primitive and reference types, interfaces, classes, and arrays—that anchored Java programming prior to Java SE 5.0.

The narrative continues with the transformative enhancements from Java SE 5.0 onward, such as the introduction of generics, annotations, and VarHandle type reference—all further enriching the language. The chapter spotlights recent additions such as switch expressions, sealed classes, and the much-anticipated records.

Special attention is given to Project Valhalla's ongoing work, examining the performance nuances of the existing type system and the potential of future value classes. The section offers insights into Project Valhalla's ongoing endeavors, from refined generics to the conceptualization of classes for basic primitives.

Java's type system is more than just a set of types—it's a reflection of Java's commitment to versatility, efficiency, and innovation. The goal of this chapter is to illuminate the type system's past, present, and promising future, fostering a profound understanding of its intricacies.

Chapter 3, "From Monolithic to Modular Java: A Retrospective and Ongoing Evolution," provides extensive coverage of the Java Platform Module System (JPMS) and its breakthrough impact on modular programming. This chapter marks Java's bold transition into the modular era, beginning with a fundamental exploration of modules. It offers hands-on guidance through the creation, compilation, and execution of modules, making it accessible even to newcomers in this domain.

Highlighting Java's transition from a monolithic JDK to a modular framework, the chapter reflects Java's adaptability to evolving needs and its commitment to innovation. A standout section of this chapter is the practical implementation of modular services using JDK 17, which navigates the intricacies of module interactions, from service providers to consumers, enriched by working examples. The chapter addresses key concepts like encapsulation of implementation details and the challenges of JAR hell, illustrating how Jigsaw layers offer elegant solutions in the modular landscape.

Further enriching this exploration, the chapter draws insightful comparisons with OSGi, spotlighting the parallels and distinctions, to give readers a comprehensive understanding of Java's modular systems. The introduction of essential tools such as *jdeps*, *jlink*, *jdeprscan*, and *jmod*, integral to the modular ecosystem, is accompanied by thorough explanations and practical examples. This approach empowers readers to effectively utilize these tools in their developmental work.

Concluding with a reflection on the performance nuances of JPMS, the chapter looks forward to the future of Java's modular evolution, inviting readers to contemplate its potential impacts and developments.

Chapter 4, "The Unified Java Virtual Machine Logging Interface," delves into the vital yet often underappreciated world of logs in software development. It begins by underscoring the necessity of a unified logging system in Java, addressing the challenges posed by disparate logging systems and the myriad benefits of a cohesive approach. The chapter not only highlights the unification and infrastructure of the logging system but also emphasizes its role in monitoring performance and optimization.

The narrative explores the vast array of log tags and their specific roles, emphasizing the importance of creating comprehensive and insightful logs. In tackling the challenges of discerning any missing information, the chapter provides a lucid understanding of log levels, outputs, and decorators. The intricacies of these features are meticulously examined, with practical examples illuminating their application in tangible scenarios.

A key aspect of this chapter is the exploration of asynchronous logging, a critical feature for enhancing log performance with minimal impact on application efficiency. This feature is essential for developers seeking to balance comprehensive logging with system performance.

Concluding the chapter, the importance of logs as a diagnostic tool is emphasized, showcasing their role in both proactive system monitoring and reactive problem-solving. Chapter 4 not only highlights the power of effective logging in Java, but also underscores its significance in building and maintaining robust applications. This chapter reinforces the theme of Java's ongoing evolution, showcasing how advancements in logging contribute significantly to the language's capability and versatility in application development.

Chapter 5, "End-to-End Java Performance Optimization: Engineering Techniques and Microbenchmarking with JMH," focuses on the essence of performance engineering within the Java ecosystem. Emphasizing that performance transcends mere speed, this chapter highlights its critical role in crafting an unparalleled user experience. It commences with a formative exploration of performance engineering's pivotal role within the broader software development realm, highlighting its status as a fundamental quality attribute and unraveling its multifaceted layers.

With precision, the chapter delineates the metrics pivotal to gauging Java's performance, encompassing aspects from footprint to the nuances of availability, ensuring readers grasp the full spectrum of performance dynamics. Stepping in further. It explores the intricacies of response time and its symbiotic relationship with availability. This inspection provides insights into the mechanics of application timelines, intricately weaving the narrative of response time, throughput, and the inevitable pauses that punctuate them.

Yet, the performance narrative is only complete by acknowledging the profound influence of hardware. This chapter decodes the symbiotic relationship between hardware and software, emphasizing the harmonious symphony that arises from the confluence of languages, processors, and memory models. From the subtleties of memory models and their bearing on thread dynamics to the foundational principles of Java Memory Model, this chapter journeys through the maze of concurrent hardware, shedding light on the order mechanisms pivotal to concurrent computing.

Moving beyond theoretical discussions, this chapter draws on over two decades of hands-on experience in performance optimization. It introduces a systematic approach to performance diagnostics and analysis, offering insights into methodologies and a detailed investigation of subsystems and approaches to identifying potential performance issues. The methodologies are not only vital for software developers focused on performance optimization but also provide valuable insights into the intricate relationship between underlying hardware, software stacks, and application performance.

The chapter emphasizes the importance of a structured benchmarking regime, encompassing everything from memory management to the assessment of feature releases and system layers. This sets the stage for the Java Micro-Benchmark Suite (JMH), the *pièce de résistance* of JVM benchmarking. From its foundational setup to the intricacies of its myriad features, the journey encompasses the genesis of writing benchmarks, to their execution, enriched with insights into benchmarking modes, profilers, and JMH's pivotal annotations.

Chapter 5 thus serves as a comprehensive guide to end-to-end Java performance optimization and as a launchpad for further chapters. It inspires a fervor for relentless optimization and arms readers with the knowledge and tools required to unlock Java's unparalleled performance potential.

Memory management is the silent guardian of Java applications, often operating behind the scenes but crucial to their success. **Chapter 6**, "Advanced Memory Management and Garbage Collection in OpenJDK," marks a deep dive into specialized JVM improvements, showcasing advanced performance tools and techniques. This chapter offers a leap into the world of garbage collection, unraveling the techniques and innovations that ensure Java applications run efficiently and effectively.

The chapter commences with a foundational overview of garbage collection in Java, setting the stage for the detailed exploration of Thread-Local Allocation Buffers (TLABs) and Promotion Local Allocation Buffers (PLABs), and elucidating their pivotal roles in memory management. As we progress, the chapter sheds light on optimizing memory access, emphasizing the significance of the NUMA-aware garbage collection and its impact on performance.

The highlight of this chapter lies in its exploration of advanced garbage collection techniques. The narrative reviews the G1 Garbage Collector (G1 GC), unraveling its revolutionary approach to heap management. From grasping the advantages of a regionalized heap to optimizing G1 GC parameters for peak performance, this section promises a holistic cognizance of one of Java's most advanced garbage collectors. Additionally, the Z Garbage Collector (ZGC) is presented as a technological marvel with its adaptive optimization techniques, and the advancements that make it a game-changer in real-time applications.

This chapter also offers insights into the emerging trends in garbage collection, setting the stage for what lies ahead. Practicality remains at the forefront, with a dedicated section offering invaluable tips for evaluating GC performance. From sympathizing with various workloads, such as Online Analytical Processing (OLAP) to Online Transaction Processing (OLTP) and Hybrid Transactional/Analytical Processing (HTAP), to synthesizing live data set pressure and data lifespan patterns, the chapter equips readers with the apparatus and knowledge to optimize memory management effectively. This chapter is an accessible guide to advanced garbage collection techniques that Java professionals need to navigate the topography of memory management.

Chapter 7, "Runtime Performance Optimizations: A Focus on Strings, Locks, and Beyond," is dedicated to exploring the critical facets of Java's runtime performance, particularly in the realms of string handling and lock synchronization—two areas essential for efficient application performance.

The chapter excels at taking a comprehensive approach to demystifying these JVM optimizations through detailed under-the-hood analysis—utilizing a range of profiling techniques, from bytecode analysis to memory and sample-based profiling to gathering call stack views of profiled methods—to enrich the reader's understanding. Additionally, the chapter leverages JMH benchmarking to highlight the tangible improvements such optimizations bring. The practical use of *async-profiler* for method-level insights and NetBeans memory profiler further enhances the reader's granular understanding of the JVM enhancements. This chapter aims to test and illuminate the optimizations, equipping readers with a comprehensive approach to using these tools effectively, thereby building on the performance engineering methodologies and processes discussed in Chapter 5.

The journey continues with an extensive review of the string optimizations in Java, highlighting major advancements across various Java versions, and then shifts focus onto enhanced multithreading performance, highlighting Java's thread synchronization mechanisms.

Further, the chapter helps navigate the world of concurrency, with discussion of the transition from the thread-per-task model to the scalable thread-per-request model. The examination of Java's Executor Service, ThreadPools, ForkJoinPool framework, and CompletableFuture ensures a robust comprehension of Java's concurrency mechanisms.

The chapter concludes with a glimpse into the future of concurrency in Java with virtual threads. From understanding virtual threads and their carriers to discussing parallelism and integration with existing APIs, this chapter is a practical guide to advanced concurrency mechanisms and string optimizations in Java.

Chapter 8, "Accelerating Time to Steady State with OpenJDK HotSpot VM," is dedicated to optimizing start-up to steady-state performance, crucial for transient applications such as containerized environments, serverless architectures, and microservices. The chapter emphasizes the importance of minimizing JVM start-up and warm-up time to enhance efficient execution, incorporating a pivotal exploration into GraalVM's revolutionary role in this domain.

The narrative dissects the phases of JVM start-up and the journey to an application's steady-state, highlighting the significance of managing state during these phases across various architectures. An in-depth look at Class Data Sharing (CDS) sheds light on shared archive files and memory mapping, underscoring the advantages in multi-instance setups. The narrative then shifts to ahead-of-time (AOT) compilation, contrasting it with just-in-time (JIT) compilation and detailing the transformative impact of HotSpot VM's Project Leyden and its forecasted ability to manage states via CDS and AOT. This sets the stage for GraalVM and its revolutionary impact on Java's performance landscape. By harnessing advanced optimization techniques, including static images and dynamic compilation, GraalVM enhances performance for a wide array of applications. The exploration of cutting-edge technologies like GraalVM alongside a holistic survey of OpenJDK projects such as CRIU and CraC, which introduce groundbreaking checkpoint/restore functionality, adds depth to the discussion. This comprehensive coverage provides insights into the evolving strategies for optimizing Java applications, making this chapter an invaluable resource for developers looking to navigate today's cloud native environments.

The final chapter, **Chapter 9**, "Harnessing Exotic Hardware: The Future of JVM Performance Engineering," focuses on the fascinating intersection of exotic hardware and the JVM, illuminating its galvanizing impact on performance engineering. This chapter begins with an introduction to the increasingly prominent world of exotic hardware, particularly within cloud environments. It explores the integration of this hardware with the JVM, underscoring the pivotal role of language design and toolchains in this process.

Through a series of carefully detailed case studies, the chapter showcases the real-world applications and challenges of integrating such hardware accelerators. From the Lightweight Java Game Library (LWJGL), to the innovative Aparapi, which bridges Java and OpenCL, each study offers valuable insights into the complexities and triumphs of these integrations. The chapter also examines Project Sumatra's significant contributions to this realm and introduces TornadoVM, a specialized JVM tailored for hardware accelerators.

Through these case studies, the symbiotic potential of integrating exotic hardware with the JVM becomes increasingly evident, leading up to an overview of Project Panama, heralding a new horizon in JVM performance engineering. At the heart of Project Panama lies the Vector API, a symbol of innovation designed for vector computations. This API is not just about computations—it's about ensuring they are efficiently vectorized and tailored for hardware that thrives on vector operations. This ensures that developers have the tools to express parallel computations optimized for diverse hardware architectures. But Panama isn't just about vectors. The Foreign Function and Memory API emerges as a pivotal tool, a bridge that allows Java to converse seamlessly with native libraries. This is Java's answer to the age-old challenge of interoperability, ensuring Java applications can interface effortlessly with native code, breaking language barriers.

Yet, the integration is no walk in the park. From managing intricate memory access patterns to deciphering hardware-specific behaviors, the path to optimization is laden with complexities. But these challenges drive innovation, pushing the boundaries of what's possible. Looking to the future, the chapter showcases my vision of Project Panama as the gold standard for JVM interoperability. The horizon looks promising, with Panama poised to redefine performance and efficiency for Java applications.

This isn't just about the present or the imminent future. The world of JVM performance engineering is on the cusp of a revolution. Innovations are knocking at our door, waiting to be embraced—with Tornado VM's Hybrid APIs, and with HAT toolkit and Project Babylon on the horizon.

How to Use This Book

1. *Sequential Reading for Comprehensive Understanding:* This book is designed to be read from beginning to end, as each chapter builds upon the knowledge of the previous ones. This approach is especially recommended for readers new to JVM performance engineering.
2. *Modular Approach for Specific Topics:* Experienced readers may prefer to jump directly to chapters that address their specific interests or challenges. The table of contents and index can guide you to relevant sections.
3. *Practical Examples and Code:* Throughout the book, practical examples and code snippets are provided to illustrate key concepts. To get the most out of these examples, readers are encouraged to build on and run the code themselves. (See item 5.)
4. *Visual Aids for Enhanced Understanding:* In addition to written explanations, this book employs a variety of textual and visual aids to deepen your understanding.
    a. *Case Studies:* Real-world scenarios that demonstrate the application of JVM performance techniques.
    b. *Screenshots:* Visual outputs depicting profiling results as well as various GC plots, which are essential for understanding the GC process and phases.
    c. *Use-Case Diagrams:* Visual representations that map out the system's functional requirements, showing how different entities interact with each other.
    d. *Block Diagrams:* Illustrations that outline the architecture of a particular JVM or system component, highlighting performance features.
    e. *Class Diagrams:* Detailed object-oriented designs of various code examples, showing relationships and hierarchies.
    f. *Process Flowcharts:* Step-by-step diagrams that walk you through various performance optimization processes and components.
    g. *Timelines:* Visual representations of the different phases or state changes in an activity and the sequence of actions that are taken.
5. *Utilizing the Companion GitHub Repository:* A significant portion of the book's value lies in its practical application. To facilitate this, I have created JVM Performance Engineering GitHub Repository (https://github.com/mo-beck/JVM-Performance-Engineering). Here, you will find
    a. *Complete Code Listings:* All the code snippets and scripts mentioned in the book are available. This allows you to see the code and experiment with it. Use it as a launchpad for your projects and fork and improve it.
    b. *Additional Resources and Updates:* The field of JVM Performance Engineering is ever evolving. The repository will be periodically updated with new scripts, resources, and information to keep you abreast of the latest developments.
    c. *Interactive Learning:* Engage with the material by cloning the repository, running the GC scripts against your GC log files, and modifying them to see how outcomes better suit your GC learning and understanding journey.

6. *Engage with the Community:* I encourage readers to engage with the wider community. Use the GitHub repository to contribute your ideas, ask questions, and share your insights. This collaborative approach enriches the learning experience for everyone involved.
7. *Feedback and Suggestions:* Your feedback is invaluable. If you have suggestions, corrections, or insights, I warmly invite you to share them. You can provide feedback via the GitHub repository, via email (jvmbook@codekaram.com), or via social media platforms (https://www.linkedin.com/in/monicabeckwith/ or https://twitter.com/JVMPerfEngineer).

In Java's vast realm, my tale takes wing,
A narrative so vivid, of wonders I sing.
Distributed systems, both near and afar,
With JVM shining—the brightest star!

Its rise through the ages, a saga profound,
With each chronicle, inquiries resound.
"Where lies the wisdom, the legends so grand?"
They ask with a fervor, eager to understand.

This book is a beacon for all who pursue,
A tapestry of insights, both aged and new.
In chapters that flow, like streams to the seas,
I share my heart's journey, my tech odyssey.

—Monica Beckwith

Register your copy of *JVM Performance Engineering* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134659879) and click Submit. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

Reflecting on the journey of creating this book, my heart is full of gratitude for the many individuals whose support, expertise, and encouragement have been the wind beneath my wings.

At the forefront of my gratitude is my family—the unwavering pillars of support. To my husband, Ben: Your understanding and belief in my work, coupled with your boundless love and care, have been the bedrock of my perseverance.

To my children, Annika and Bodin: Your patience and resilience have been my inspiration. Balancing the demands of a teen's life with the years it took to bring this book to fruition, you have shown a maturity and understanding well beyond your years. Your support, whether it be a kind word at just the right moment or understanding my need for quiet as I wrestled with complex ideas, has meant more to me than words can express. Your unwavering faith, even when my work required sacrifices from us all, has been a source of strength and motivation. I am incredibly proud of the kind and supportive individuals you are becoming, and I hope this book reflects the values we cherish as a family.

Editorial Guidance

A special word of thanks goes to my executive editor at Pearson, Greg Doench, whose patience has been nothing short of saintly. Over the years, through health challenges, the dynamic nature of JVM release cycles and project developments, and the unprecedented times of COVID, Greg has been a beacon of encouragement. His unwavering support and absence of frustration in the face of my tardiness have been nothing less than extraordinary. Greg, your steadfast presence and guidance have not only helped shape this manuscript but have also been a personal comfort.

Chapter Contributions

The richness of this book's content is a culmination of my extensive work, research, and insights in the field, enriched further by the invaluable contributions of various experts, colleagues, collaborators, and friends. Their collective knowledge, feedback, and support have been instrumental in adding depth and clarity to each topic discussed, reflecting years of dedicated expertise in this domain.

- In Chapter 3, Nikita Lipski's deep experience in Java modularity added compelling depth, particularly on the topics of the JAR hell versioning issues, layers, and his remarkable insights on OSGi.
- Stefano Doni's enriched field expertise in Quality of Service (QoS), performance stack, and theoretical expertise in operational laws and queueing, significantly enhanced Chapter 5, bringing a blend of theoretical and practical perspectives.
- The insights and collaborative interactions with Per Liden and Stefan Karlsson were crucial in refining my exploration of the Z Garbage Collector (ZGC) in Chapter 6. Per's

numerous talks and blog posts have also been instrumental in helping the community understand the intricacies of ZGC in greater detail.

- Chapter 7 benefitted from the combined insights of Alan Bateman and Heinz Kabutz. Alan was instrumental in helping me refine this chapter's coverage of Java's locking mechanisms and virtual threads. His insights helped clarify complex concepts, added depth to the discussion of monitor locks, and provided valuable perspective on the evolution of Java's concurrency model. Heinz's thorough review ensured the relevance and accuracy of the content.
- For Chapter 8, Ludovic Henry's insistence on clarity with respect to the various terminologies and persistence to include advanced topics and Alina Yurenko's insights into GraalVM and its future developments provided depth and foresight and reshaped the chapter to its glorious state today.

  Alina has also influenced me to track the developments in GraalVM—especially the introduction of layered native images, which promises to reduce build times and enable sharing of base images.
- Last but not the least, for Chapter 9, I am grateful to Gary Frost for his thorough review of Aparapi, Project Sumatra, and insights on leveraging the latest JDK (early access) version for developing projects like Project Panama. Dr. Juan Fumero's leadership in the development of TornadoVM and insights into parallel programming challenges have been instrumental in providing relevant insights for my chapter, deepening its clarity, and enhancing its narrative.

  It was a revelation to see our visions converge and witness these industry stalwarts drive the enhancements in the integration of Java with modern hardware accelerators.

Mentors, Influencers, and Friends

Several mentors, leaders, and friends have significantly influenced my broader understanding of technology:

- Charlie Hunt's guidance in my GC performance engineering journey has been foundational. His groundbreaking work on String density has inspired many of my own approaches with methodologies and process. His seminal work *Java Performance* is an essential resource for all performance enthusiasts and is highly recommended for its depth and insight.
- Gil Tene's work on the C4 Garbage Collector and his educational contributions have deeply influenced my perspective on low pause collectors and their interactive nature. I value our check-ins, which I took as mentorship opportunities to learn from one of the brightest minds.
- Thomas Schatzl's generous insights on the G1 Garbage Collector have added depth and context to this area of study, enriching my understanding following my earlier work on G1 GC. Thomas is a GC performance expert whose work, including on Parallel GC, continues to inspire me.

- Vladimir Kozlov's leadership and work in various aspects of the HotSpot JVM have been crucial in pushing the boundaries of Java's performance capabilities. I cherish our work together on prefetching, tiered thresholds, various code generation, and JVM optimizations, and I appreciate his dedication to HotSpot VM.
- Kirk Pepperdine, for our ongoing collaborations that span from the early days of developing G1 GC parser scripts for our joint hands-on lab sessions at JavaOne, to our recent methodologies, processes, and benchmarking endeavors at Microsoft, continuously pushes the envelope in performance engineering.
- Sergey Kuksenko and Alexey Shipilev, along with my fellow JVM performance engineering experts, have been my comrades in relentless pursuit of Java performance optimizations.
- Erik Österlund's development of generational ZGC represents an exciting and forward-looking aspect of garbage collection technology.
- John Rose, for his unparalleled expertise in JVM internals and his pivotal role in the evolution of Java as a language and platform. His vision and deep technical knowledge have not only propelled the field forward but also provided me with invaluable insights throughout my career.

Each of these individuals has not only contributed to the technical depth and richness of this book but also played a vital role in my personal and professional growth. Their collective wisdom, expertise, and support have been instrumental in shaping both the content and the journey of this book, reflecting the collaborative spirit of the Java community.

This page intentionally left blank

About the Author

Monica Beckwith is a leading figure in Java Virtual Machine (JVM) performance tuning and optimizations. With a strong Electrical and Computer Engineering academic foundation, Monica has carved out an illustrious, impactful, and inspiring professional journey.

At Advanced Micro Devices (AMD), Monica refined her expertise in Java, JVM, and systems performance engineering. Her work brought critical insights to NUMA's architectural enhancements, improving both hardware and JVM performance through optimized code generation, improved footprint and advanced JVM techniques, and memory management. She continued her professional growth at Sun Microsystems, contributing significantly to JVM performance enhancements across Sun SPARC, Solaris, and Linux, aiding in the evolution of a scalable Java ecosystem.

Monica's role as a Java Champion and coauthor of *Java Performance Companion*, as well as authoring this current book, highlight her steadfast commitment to the Java community. Notably, her work in the optimization of G1 Garbage Collector went beyond optimization; she delved into diagnosing pain points, fine-tuning processes, and identifying critical areas for enhancement, thereby setting new precedents in JVM performance. Her expertise not only elevated the efficiency of the G1 GC but also showcased her intricate knowledge of JVM's complexities. At Arm, as a managed runtimes performance architect, Monica played a key role in shaping a unified strategy for the Arm ecosystem, fostering a competitive edge for performance on Arm-based servers.

Monica's significant contributions and thought leadership have enriched the broader tech community. Monica serves on the program committee for various prestigious conferences and hosts JVM and performance-themed tracks, further emphasizing her commitment to knowledge sharing and community building.

At Microsoft, Monica's expertise shines brightly as she optimizes JVM-based workloads, applications, and key services, across a diverse range of deployment scenarios, from bare metal to sophisticated Azure VMs. Her deep-seated understanding of hardware and software engineering, combined with her adeptness in systems engineering and benchmarking principles, uniquely positions her at the critical juncture of the hardware and software. This position enables her to significantly contribute to the performance, scalability and power efficiency characterization, evaluation, and analysis of both current and emerging hardware systems within the Azure Compute infrastructure.

Beyond her technical prowess, Monica embodies values that resonate deeply with those around her. She is a beacon of integrity, authenticity, and continuous learning. Her belief in the transformative power of actions, the sanctity of reflection, and the profound impact of empathy defines her interactions and approach. A passionate speaker, Monica's commitment to lifelong learning is evident in her zeal for delivering talks and disseminating knowledge.

Outside the confines of the tech world, Monica's dedication extends to nurturing young minds as a First Lego League coach. This multifaceted persona, combined with her roles as a Java Champion, author, and performance engineer at Microsoft, cements her reputation as a respected figure in the tech community and a source of inspiration for many.

This page intentionally left blank

Chapter 3

From Monolithic to Modular Java: A Retrospective and Ongoing Evolution

Introduction

In the preceding chapters, we journeyed through the significant advancements in the Java language and its execution environment, witnessing the remarkable growth and transformation of these foundational elements. However, a critical aspect of Java's evolution, which has far-reaching implications for the entire ecosystem, is the transformation of the Java Development Kit (JDK) itself. As Java matured, it introduced a plethora of features and language-level enhancements, each contributing to the increased complexity and sophistication of the JDK. For instance, the introduction of the enumeration type in J2SE 5.0 necessitated the addition of the `java.lang.Enum` base class, the `java.lang.Class.getEnumConstants()` method, `EnumSet`, and `EnumMap` to the `java.util` package, along with updates to the *Serialized Form*. Each new feature or syntax addition required meticulous integration and robust support to ensure seamless functionality.

With every expansion of Java, the JDK began to exhibit signs of unwieldiness. Its monolithic structure presented challenges such as an increased memory footprint, slower start-up times, and difficulties in maintenance and updates. The release of JDK 9 marked a significant turning point in Java's history, as it introduced the Java Platform Module System (JPMS) and transitioned Java from a monolithic structure to a more manageable, modular one. This evolution continued with JDK 11 and JDK 17, with each bringing further enhancements and refinements to the modular Java ecosystem.

This chapter delves into the specifics of this transformation. We will explore the inherent challenges of the monolithic JDK and detail the journey toward modularization. Our discussion will extend to the benefits of modularization for developers, particularly focusing on those who have adopted JDK 11 or JDK 17. Furthermore, we'll consider the impact of these changes on JVM performance engineering, offering insights to help developers optimize their applications and leverage the latest JDK innovations. Through this exploration, the goal is to demonstrate how Java applications can significantly benefit from modularization.

Understanding the Java Platform Module System

As just mentioned, the JPMS was a strategic response to the mounting complexity and unwieldiness of the monolithic JDK. The primary goal when developing it was to create a scalable platform that could effectively manage security risks at the API level while enhancing performance. The advent of modularity within the Java ecosystem empowered developers with the flexibility to select and scale modules based on the specific needs of their applications. This transformation allowed Java platform developers to use a more modular layout when managing the Java APIs, thereby fostering a system that was not only more maintainable but also more efficient. A significant advantage of this modular approach is that developers can utilize only those parts of the JDK that are necessary for their applications; this selective usage reduces the size of their applications and improves load times, leading to more efficient and performant applications.

Demystifying Modules

In Java, a module is a cohesive unit comprising packages, resources, and a module descriptor (`module-info.java`) that provides information about the module. The module serves as a container for these elements. Thus, a module

- **Encapsulates its packages:** A module can declare which of its packages should be accessible to other modules and which should be hidden. This encapsulation improves code maintainability and security by allowing developers to clearly express their code's intended usage.
- **Expresses dependencies:** A module can declare dependencies on other modules, making it clear which modules are required for that module to function correctly. This explicit dependency management simplifies the deployment process and helps developers identify problematic issues early in the development cycle.
- **Enforces strong encapsulation:** The module system enforces strong encapsulation at both compile time and runtime, making it difficult to break the encapsulation either accidentally or maliciously. This enforcement leads to better security and maintainability.
- **Boosts performance:** The module system allows the JVM to optimize the loading and execution of code, leading to improved start-up times, lower memory consumption, and faster execution.

The adoption of the module system has greatly improved the Java platform's maintainability, security, and performance.

Modules Example

Let's explore the module system by considering two example modules: `com.house.brickhouse` and `com.house.bricks`. The `com.house.brickhouse` module contains two classes, `House1` and `House2`, which calculate the number of bricks needed for houses with different levels. The `com.house.bricks` module contains a `Story` class that provides a method to count bricks based on the number of levels. Here's the directory structure for `com.house.brickhouse`:

```
src
── com.house.brickhouse
        ├── com
        │   └── house
        │       └── brickhouse
        │           ├── House1.java
        │           └── House2.java
        └── module-info.java
```

com.house.brickhouse:
   module-info.java:

```
module com.house.brickhouse {
    requires com.house.bricks;
    exports com.house.brickhouse;
}
```

com/house/brickhouse/House1.java:

```
package com.house.brickhouse;
import com.house.bricks.Story;

public class House1 {
    public static void main(String[] args) {
        System.out.println("My single-level house will need " + Story.count(1) + " bricks");
    }
}
```

com/house/brickhouse/House2.java:

```
package com.house.brickhouse;
import com.house.bricks.Story;

public class House2 {
    public static void main(String[] args) {
        System.out.println("My two-level house will need " + Story.count(2) + " bricks");
    }
}
```

Now let's look at the directory structure for `com.house.bricks`:

```
src
└── com.house.bricks
    ├── com
    │   └── house
    │       └── bricks
    │           └── Story.java
    └── module-info.java

com.house.bricks:
   module-info.java:

module com.house.bricks {
    exports com.house.bricks;
}

com/house/bricks/Story.java:

package com.house.bricks;

public class Story {
    public static int count(int level) {
        return level * 18000;
    }
}
```

Compilation and Run Details

We compile the `com.house.bricks` module first:

```
$ javac -d mods/com.house.bricks src/com.house.bricks/module-info.java src/com.house.bricks/com/
house/bricks/Story.java
```

Next, we compile the `com.house.brickhouse` module:

```
$ javac --module-path mods -d mods/com.house.brickhouse
src/com.house.brickhouse/module-info.java
src/com.house.brickhouse/com/house/brickhouse/House1.java
src/com.house.brickhouse/com/house/brickhouse/House2.java
```

Now we run the `House1` example:

```
$ java --module-path mods -m com.house.brickhouse/com.house.brickhouse.House1
```

Output:

```
My single-level house will need 18000 bricks
```

Then we run the `House2` example:

```
$ java --module-path mods -m com.house.brickhouse/com.house.brickhouse.House2
```

Output:

```
My two-level house will need 36000 bricks
```

Introducing a New Module

Now, let's expand our project by introducing a new module that provides various types of bricks. We'll call this module `com.house.bricktypes`, and it will include different classes for different types of bricks. Here's the new directory structure for the `com.house.bricktypes` module:

```
src
└── com.house.bricktypes
    ├── com
    │   └── house
    │       └── bricktypes
    │           ├── ClayBrick.java
    │           └── ConcreteBrick.java
    └── module-info.java

com.house.bricktypes:
    module-info.java:

module com.house.bricktypes {
    exports com.house.bricktypes;
}
```

The `ClayBrick.java` and `ConcreteBrick.java` classes will define the properties and methods for their respective brick types.

`ClayBrick.java`:

```
package com.house.bricktypes;

public class ClayBrick {
    public static int getBricksPerSquareMeter() {
        return 60;
    }
}
```

`ConcreteBrick.java`:

```
package com.house.bricktypes;

public class ConcreteBrick {
    public static int getBricksPerSquareMeter() {
        return 50;
    }
}
```

With the new module in place, we need to update our existing modules to make use of these new brick types. Let's start by updating the `module-info.java` file in the `com.house.brickhouse` module:

```
module com.house.brickhouse {
    requires com.house.bricks;
    requires com.house.bricktypes;
    exports com.house.brickhouse;
}
```

We modify the `House1.java` and `House2.java` files to use the new brick types.

`House1.java`:

```
package com.house.brickhouse;
import com.house.bricks.Story;
import com.house.bricktypes.ClayBrick;

public class House1 {
    public static void main(String[] args) {
        int bricksPerSquareMeter = ClayBrick.getBricksPerSquareMeter();
        System.out.println("My single-level house will need "
                                    + Story.count(1, bricksPerSquareMeter) + " clay bricks");
    }
}
```

`House2.java`:

```
package com.house.brickhouse;
import com.house.bricks.Story;
import com.house.bricktypes.ConcreteBrick;

public class House2 {
    public static void main(String[] args) {
        int bricksPerSquareMeter = ConcreteBrick.getBricksPerSquareMeter();
        System.out.println("My two-level house will need "
                        + Story.count(2, bricksPerSquareMeter) + " concrete bricks");
    }
}
```

By making these changes, we're allowing our `House1` and `House2` classes to use different types of bricks, which adds more flexibility to our program. Let's now update the `Story.java` class in the `com.house.bricks` module to accept the bricks per square meter:

```
package com.house.bricks;

public class Story {
    public static int count(int level, int bricksPerSquareMeter) {
        return level * bricksPerSquareMeter * 300;
    }
}
```

Now that we've updated our modules, let's compile and run them to see the changes in action:

- Create a new `mods` directory for the `com.house.bricktypes` module:

```
$ mkdir mods/com.house.bricktypes
```

- Compile the `com.house.bricktypes` module:

```
$ javac -d mods/com.house.bricktypes
src/com.house.bricktypes/module-info.java
src/com.house.bricktypes/com/house/bricktypes/*.java
```

- Recompile the `com.house.bricks` and `com.house.brickhouse` modules:

```
$ javac --module-path mods -d mods/com.house.bricks
src/com.house.bricks/module-info.java src/com.house.bricks/com/house/bricks/Story.java

$ javac --module-path mods -d mods/com.house.brickhouse
src/com.house.brickhouse/module-info.java
src/com.house.brickhouse/com/house/brickhouse/House1.java
src/com.house.brickhouse/com/house/brickhouse/House2.java
```

With these updates, our program is now more versatile and can handle different types of bricks. This is just one example of how the modular system in Java can make our code more flexible and maintainable.

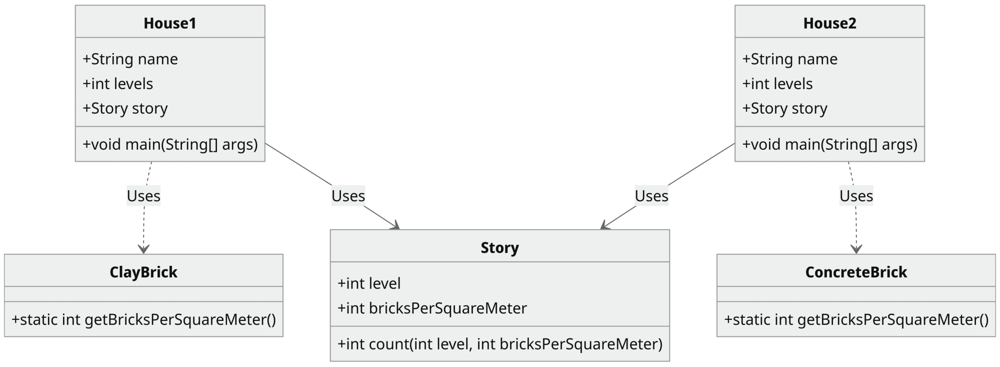
House1
+String name
+int levels
+Story story
+void main(String[] args)
House2
+String name
+int levels
+Story story
+void main(String[] args)
Uses
Uses
Uses
Uses
ClayBrick
+static int getBricksPerSquareMeter()
Story
+int level
+int bricksPerSquareMeter
+int count(int level, int bricksPerSquareMeter)
ConcreteBrick
+static int getBricksPerSquareMeter()

Figure 3.1 Class Diagram to Show the Relationships Between Modules

Let's now visualize these relationships with a class diagram. Figure 3.1 includes the new module `com.house.bricktypes`, and the arrows represent "Uses" relationships. `House1` uses `Story` and `ClayBrick`, whereas `House2` uses `Story` and `ConcreteBrick`. As a result, instances of `House1` and `House2` will contain references to instances of `Story` and either `ClayBrick` or `ConcreteBrick`, respectively. They use these references to interact with the methods and attributes of the `Story`, `ClayBrick`, and `ConcreteBrick` classes. Here are more details:

- `House1` and `House2`: These classes represent two different types of houses. Both classes have the following attributes:
  - `name`: A string representing the name of the house.
  - `levels`: An integer representing the number of levels in the house.
  - `story`: An instance of the `Story` class representing a level of the house.
  - `main(String[] args)`: The entry method for the class, which acts as the initial kick-starter for the application's execution.
- `Story`: This class represents a level in a house. It has the following attributes:
  - `level`: An integer representing the level number.
  - `bricksPerSquareMeter`: An integer representing the number of bricks per square meter for the level.
  - `count(int level, int bricksPerSquareMeter)`: A method that calculates the total number of bricks required for a given level and bricks per square meter.
- `ClayBrick` and `ConcreteBrick`: These classes represent two different types of bricks. Both classes have the following attributes:
  - `getBricksPerSquareMeter()`: A static method that returns the number of bricks per square meter. This method is called by the houses to obtain the value needed for calculations in the `Story` class.

Next, let's look at the use-case diagram of the Brick House Construction system with the House Owner as the actor and Clay Brick and Concrete Brick as the systems (Figure 3.2). This diagram illustrates how the House Owner interacts with the system to calculate the number of bricks required for different types of houses and choose the type of bricks for the construction.

Here's more information on the elements of the use-case diagram:

- **House Owner:** This is the actor who wants to build a house. The House Owner interacts with the Brick House Construction system in the following ways:
  - **Calculate Bricks for House 1:** The House Owner uses the system to calculate the number of bricks required to build House 1.
  - **Calculate Bricks for House 2:** The House Owner uses the system to calculate the number of bricks required to build House 2.
  - **Choose Brick Type:** The House Owner uses the system to select the type of bricks to be used for the construction.
- **Brick House Construction:** This system helps the House Owner in the construction process. It provides the following use cases:
  - **Calculate Bricks for House 1:** This use case calculates the number of bricks required for House 1. It interacts with both the Clay Brick and Concrete Brick systems to get the necessary data.
  - **Calculate Bricks for House 2:** This use case calculates the number of bricks required for House 2. It also interacts with both the Clay Brick and Concrete Brick systems to get the necessary data.
  - **Choose Brick Type:** This use case allows the House Owner to choose the type of bricks for the construction.
- **Clay Brick and Concrete Brick:** These systems provide the data (e.g., size, cost) to the Brick House Construction system that is needed to calculate the number of bricks required for the construction of the houses.

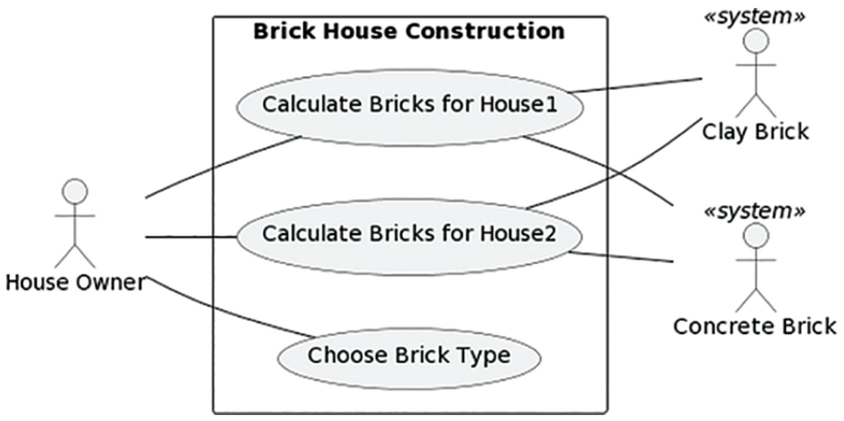

Figure 3.2 Use-Case Diagram of Brick House Construction

From Monolithic to Modular: The Evolution of the JDK

Before the introduction of the modular JDK, the bloating of the JDK led to overly complex and difficult-to-read applications. In particular, complex dependencies and cross-dependencies made it difficult to maintain and extend applications. JAR (Java Archives) hell (i.e., problems related to loading classes in Java) arose due to both the lack of simplicity and JARs' lack of awareness about the classes they contained.

The sheer footprint of the JDK also posed a challenge, particularly for smaller devices or other situations where the entire monolithic JDK wasn't needed. The modular JDK came to the rescue, transforming the JDK landscape.

Continuing the Evolution: Modular JDK in JDK 11 and Beyond

The Java Platform Module System (JPMS) was first introduced in JDK 9, and its evolution has continued in subsequent releases. JDK 11, the first long-term support (LTS) release after JDK 8, further refined the modular Java platform. Some of the notable improvements and changes made in JDK 11 are summarized here:

- **Removal of deprecated modules:** Some Java Enterprise Edition (EE) and Common Object Request Broker Architecture (CORBA) modules that had been deprecated in JDK 9 were finally removed in JDK 11. This change promoted a leaner Java platform and reduced the maintenance burden.
- **Matured module system:** The JPMS has matured over time, benefiting from the feedback of developers and real-world usage. Newer JDK releases have addressed issues, improved performance, and optimized the module system's capabilities.
- **Refined APIs:** APIs and features have been refined in subsequent releases, providing a more consistent and coherent experience for developers using the module system.
- **Continued enhancements:** JDK 11 and subsequent releases have continued to enhance the module system—for example, by offering better diagnostic messages and error reporting, improved JVM performance, and other incremental improvements that benefit developers.

Implementing Modular Services with JDK 17

With the JDK's modular approach, we can enhance the concept of services (introduced in Java 1.6) by decoupling modules that provide the service interface from their provider module, eventually creating a fully decoupled consumer. To employ services, the type is usually declared as an interface or an abstract class, and the service providers need to be clearly identified in their modules, enabling them to be recognized as providers. Lastly, consumer modules are required to utilize those providers.

To better explain the decoupling that occurs, we'll use a step-by-step example to build a `BricksProvider` along with its providers and consumers.

Service Provider

A service provider is a module that implements a service interface and makes it available for other modules to consume. It is responsible for implementing the functionalities defined in the service interface. In our example, we'll create a module called `com.example.bricksprovider`, which will implement the `BrickHouse` interface and provide the service.

Creating the `com.example.bricksprovider` Module

First, we create a new directory called `bricksprovider`; inside it, we create the `com/example/bricksprovider` directory structure. Next, we create a `module-info.java` file in the `bricksprovider` directory with the following content:

```
module com.example.bricksprovider {
    requires com.example.brickhouse;
    provides com.example.brickhouse.BrickHouse with com.example.bricksprovider.BricksProvider;
}
```

This `module-info.java` file declares that our module requires the `com.example.brickhouse` module and provides an implementation of the `BrickHouse` interface through the `com.example.bricksprovider.BricksProvider` class.

Now, we create the `BricksProvider.java` file inside the `com/example/bricksprovider` directory with the following content:

```
package com.example.bricksprovider;
import com.example.brickhouse.BrickHouse;

public class BricksProvider implements BrickHouse {
    @Override
    public void build() {
        System.out.println("Building a house with bricks...");
    }
}
```

Service Consumer

A service consumer is a module that uses a service provided by another module. It declares the service it requires in its `module-info.java` file using the `uses` keyword. The service consumer can then use the `ServiceLoader` API to discover and instantiate implementations of the required service.

Creating the `com.example.builder` Module

First, we create a new directory called `builder`; inside it, we create the `com/example/builder` directory structure. Next, we create a `module-info.java` file in the `builder` directory with the following content:

```
module com.example.builder {
    requires com.example.brickhouse;
    uses com.example.brickhouse.BrickHouse;
}
```

This `module-info.java` file declares that our module requires the `com.example.brickhouse` module and uses the `BrickHouse` service.

Now, we create a `Builder.java` file inside the `com/example/builder` directory with the following content:

```
package com.example.builder;
import com.example.brickhouse.BrickHouse;
import java.util.ServiceLoader;

public class Builder {
    public static void main(String[] args) {
        ServiceLoader<BrickHouse> loader = ServiceLoader.load(BrickHouse.class);
        loader.forEach(BrickHouse::build);
    }
}
```

A Working Example

Let's consider a simple example of a modular Java application that uses services:

- `com.example.brickhouse`: A module that defines the `BrickHouse` service interface that other modules can implement
- `com.example.bricksprovider`: A module that provides an implementation of the `BrickHouse` service and declares it in its `module-info.java` file using the `provides` keyword
- `com.example.builder`: A module that consumes the `BrickHouse` service and declares the required service in its `module-info.java` file using the `uses` keyword

The builder can then use the `ServiceLoader` API to discover and instantiate the `BrickHouse` implementation provided by the `com.example.bricksprovider` module.

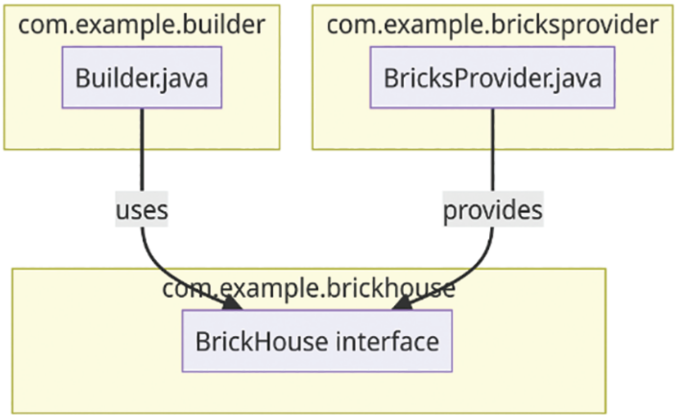

Figure 3.3 Modular Services

Figure 3.3 depicts the relationships between the modules and classes in a module diagram. The module diagram represents the dependencies and relationships between the modules and classes:

- The `com.example.builder` module contains the `Builder.java` class, which uses the `BrickHouse` interface from the `com.example.brickhouse` module.
- The `com.example.bricksprovider` module contains the `BricksProvider.java` class, which implements and provides the `BrickHouse` interface.

Implementation Details

The `ServiceLoader` API is a powerful mechanism that allows the `com.example.builder` module to discover and instantiate the `BrickHouse` implementation provided by the `com.example.bricksprovider` module at runtime. This allows for more flexibility and better separation of concerns between modules. The following subsections focus on some implementation details that can help us better understand the interactions between the modules and the role of the `ServiceLoader` API.

Discovering Service Implementations

The `ServiceLoader.load()` method takes a service interface as its argument—in our case, `BrickHouse.class`—and returns a `ServiceLoader` instance. This instance is an iterable object containing all available service implementations. The `ServiceLoader` relies on the information provided in the `module-info.java` files to discover the service implementations.

Instantiating Service Implementations

When iterating over the `ServiceLoader` instance, the API automatically instantiates the service implementations provided by the service providers. In our example, the `BricksProvider` class is instantiated, and its `build()` method is called when iterating over the `ServiceLoader` instance.

Encapsulating Implementation Details

By using the JPMS, the `com.example.bricksprovider` module can encapsulate its implementation details, exposing only the `BrickHouse` service that it provides. This allows the `com.example.builder` module to consume the service without depending on the concrete implementation, creating a more robust and maintainable system.

Adding More Service Providers

Our example can be easily extended by adding more service providers implementing the `BrickHouse` interface. As long as the new service providers are properly declared in their respective `module-info.java` files, the `com.example.builder` module will be able to discover and use them automatically through the `ServiceLoader` API. This allows for a more modular and extensible system that can adapt to changing requirements or new implementations.

Figure 3.4 is a use-case diagram that depicts the interactions between the service consumer and service provider. It includes two actors: Service Consumer and Service Provider.

- **Service Consumer:** This uses the services provided by the Service Provider. The Service Consumer interacts with the Modular JDK in the following ways:
  - **Discover Service Implementations:** The Service Consumer uses the Modular JDK to find available service implementations.
  - **Instantiate Service Implementations:** Once the service implementations are discovered, the Service Consumer uses the Modular JDK to create instances of these services.
  - **Encapsulate Implementation Details:** The Service Consumer benefits from the encapsulation provided by the Modular JDK, which allows it to use services without needing to know their underlying implementation.
- **Service Provider:** This implements and provides the services. The Service Provider interacts with the Modular JDK in the following ways:
  - **Implement Service Interface:** The Service Provider uses the Modular JDK to implement the service interface, which defines the contract for the service.
  - **Encapsulate Implementation Details:** The Service Provider uses the Modular JDK to hide the details of its service implementation, exposing only the service interface.
  - **Add More Service Providers:** The Service Provider can use the Modular JDK to add more providers for the service, enhancing the modularity and extensibility of the system.

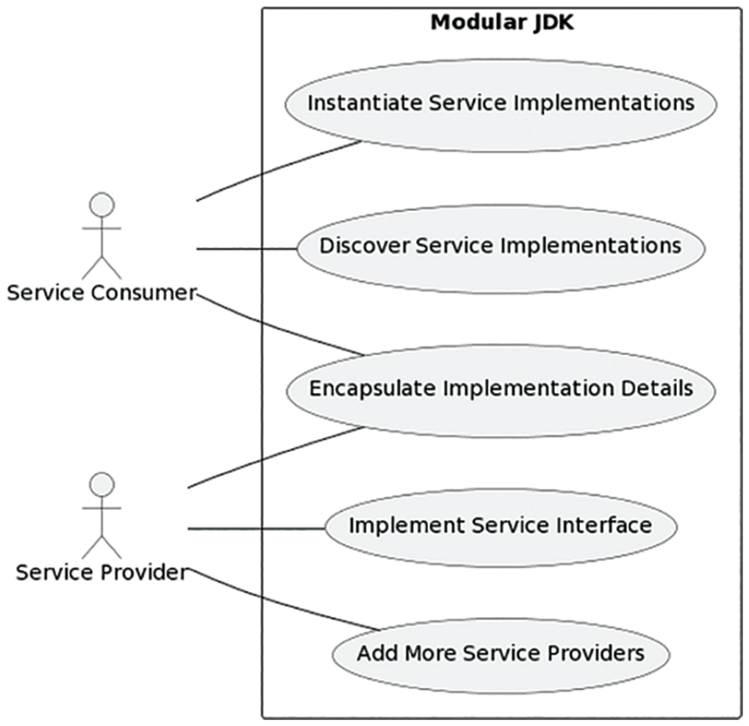
Modular JDK
Instantiate Service Implementations
Discover Service Implementations
Encapsulate Implementation Details
Implement Service Interface
Add More Service Providers
Service Consumer
Service Provider

Figure 3.4 Use-Case Diagram Highlighting the Service Consumer and Service Provider

The Modular JDK acts as a robust facilitator for these interactions, establishing a comprehensive platform where service providers can effectively offer their services. Simultaneously, it provides an avenue for service consumers to discover and utilize these services efficiently. This dynamic ecosystem fosters a seamless exchange of services, enhancing the overall functionality and interoperability of modular Java applications.

JAR Hell Versioning Problem and Jigsaw Layers

Before diving into the details of the JAR hell versioning problem and Jigsaw layers, I'd like to introduce Nikita Lipski, a fellow JVM engineer and an expert in the field of Java modularity. Nikita has provided valuable insights and a comprehensive write-up on this topic, which we will be discussing in this section. His work will help us better understand the JAR hell versioning problem and how Jigsaw layers can be utilized to address this issue in JDK 11 and JDK 17.

Java's backward compatibility is one of its key features. This compatibility ensures that when a new version of Java is released, applications built for older versions can run on the new version without any changes to the source code, and often even without recompilation. The same principle applies to third-party libraries—applications can work with updated versions of the libraries without modifications to the source code.

However, this compatibility does not extend to versioning at the source level, and the JPMS does not introduce versioning at this level, either. Instead, versioning is managed at the artifact level, using artifact management systems like Maven or Gradle. These systems handle versioning and dependency management for the libraries and frameworks used in Java projects, ensuring that the correct versions of the dependencies are included in the build process. But what happens when a Java application depends on multiple third-party libraries, which in turn may depend on different versions of another library? This can lead to conflicts and runtime errors if multiple versions of the same library are present on the classpath.

So, although JPMS has certainly improved modularity and code organization in Java, the "JAR hell" problem can still be relevant when dealing with versioning at the artifact level. Let's look at an example (shown in Figure 3.5) where an application depends on two third-party libraries (Foo and Bar), which in turn depend on different versions of another library (Baz).

If both versions of the Baz library are placed on the classpath, it becomes unclear which version of the library will be used at runtime, resulting in unavoidable version conflicts. To address this issue, JPMS prohibits such situations by detecting split packages, which are not allowed in JPMS, in support of its "reliable configuration" goal (Figure 3.6).

While detecting versioning problems early is useful, JPMS does not provide a recommended way to resolve them. One approach to address these problems is to use the latest version of the conflicting library, assuming it is backward compatible. However, this might not always be possible due to introduced incompatibilities.

To address such cases, JPMS offers the *ModuleLayer* feature, which allows for the installation of a module sub-graph into the module system in an isolated manner. When different versions of the conflicting library are placed into separate layers, both of those versions can be loaded by JPMS. Although there is no direct way to access a module of the child layer from the parent layer, this can be achieved indirectly—by implementing a service provider in the child layer module, which the parent layer module can then use. (See the earlier discussion of "Implementing Modular Services with JDK 17" for more details.)

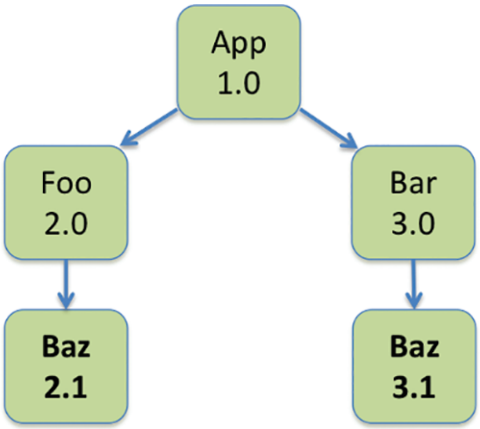

Figure 3.5 Modularity and Version Conflicts

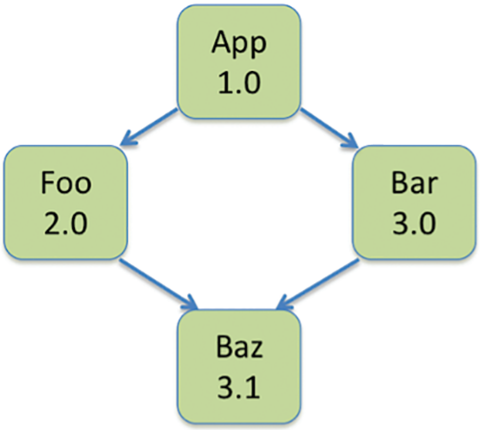

Figure 3.6 Reliable Configuration with JPMS

Working Example: JAR Hell

In this section, a working example is provided to demonstrate the use of module layers in addressing the JAR hell problem in the context of JDK 17 (this strategy is applicable to JDK 11 users as well). This example builds upon Nikita's explanation and the house service provider implementation we discussed earlier. It demonstrates how you can work with different versions of a library (termed basic and high-quality implementations) within a modular application.

First, let's take a look at the sample code provided by Java SE 9 documentation:[1]

```
1 ModuleFinder finder = ModuleFinder.of(dir1, dir2, dir3);

2 ModuleLayer parent = ModuleLayer.boot();

3 Configuration cf = parent.configuration().resolve(finder, ModuleFinder.of(),
  Set.of("myapp"));

4 ClassLoader scl = ClassLoader.getSystemClassLoader();

5 ModuleLayer layer = parent.defineModulesWithOneLoader(cf, scl);
```

In this example:

- At line 1, a `ModuleFinder` is set up to locate modules from specific directories (`dir1`, `dir2`, and `dir3`).
- At line 2, the boot layer is established as the parent layer.
- At line 3, the boot layer's configuration is resolved as the parent configuration for the modules found in the directories specified in line 1.
- At line 5, a new layer with the resolved configuration is created, using a single class loader with the system class loader as its parent.

[1] https://docs.oracle.com/javase/9/docs/api/java/lang/ModuleLayer.html

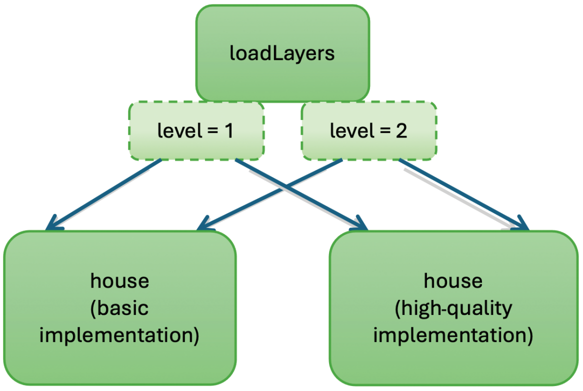
loadLayers
level = 1
level = 2
house (basic implementation)
house (high-quality implementation)

Figure 3.7 JPMS Example Versions and Layers

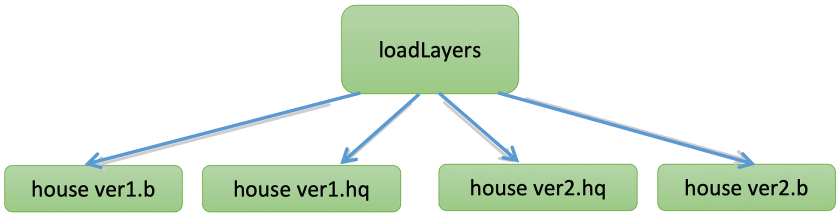

Figure 3.8 JPMS Example Version Layers Flattened

Now, let's extend our house service provider implementation. We'll have basic and high-quality implementations provided in the `com.codekaram.provider` modules. You can think of the "basic implementation" as version 1 of the house library and the "high-quality implementation" as version 2 of the house library (Figure 3.7).

For each level, we will reach out to both the libraries. So, our combinations would be level 1 + basic implementation provider, level 1 + high-quality implementation provider, level 2 + basic implementation provider, and level 2 + high-quality implementation provider. For simplicity, let's denote the combinations as *house ver1.b*, *house ver1.hq*, *house ver2.b*, and *house ver2.hq*, respectively (Figure 3.8).

Implementation Details

Building upon the concepts introduced by Nikita in the previous section, let's dive into the implementation details and understand how the layers' structure and program flow work in practice. First, let's look at the source trees:

```
ModuleLayer
├── basic
│   └── src
│       └── com.codekaram.provider
│           ├── classes
│           │   ├── com
│           │   │   └── codekaram
│           │   │       └── provider
│           │   │           └── House.java
│           │   └── module-info.java
│           └── tests
├── high-quality
│   └── src
│       └── com.codekaram.provider
│           ├── classes
│           │   ├── com
│           │   │   └── codekaram
│           │   │       └── provider
│           │   │           └── House.java
│           │   └── module-info.java
│           └── tests
└── src
    └── com.codekaram.brickhouse
        ├── classes
        │   ├── com
        │   │   └── codekaram
        │   │       └── brickhouse
        │   │           ├── loadLayers.java
        │   │           └── spi
        │   │               └── BricksProvider.java
        │   └── module-info.java
        └── tests
```

Here's the module file information and the module graph for `com.codekaram.provider`. Note that these look exactly the same for both the basic and high-quality implementations.

```
module com.codekaram.provider {
    requires com.codekaram.brickhouse;
    uses com.codekaram.brickhouse.spi.BricksProvider;
    provides com.codekaram.brickhouse.spi.BricksProvider with com.codekaram.provider.House;
}
```

The module diagram (shown in Figure 3.9) helps visualize the dependencies between modules and the services they provide, which can be useful for understanding the structure of a modular Java application:

- The `com.codekaram.provider` module depends on the `com.codekaram.brickhouse` module and implicitly depends on the `java.base` module, which is the foundational module of every Java application. This is indicated by the arrows pointing from `com.`

`codekaram.provider` to `com.codekaram.brickhouse` and the assumed arrow to `java.base`.

- The `com.codekaram.brickhouse` module also implicitly depends on the `java.base` module, as all Java modules do.

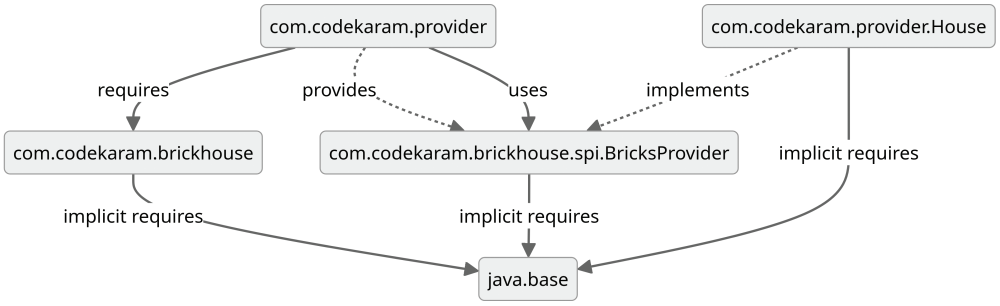

Figure 3.9 A Working Example with Services and Layers

- The `java.base` module does not depend on any other module and is the core module upon which all other modules rely.
- The `com.codekaram.provider` module provides the service `com.codekaram.brickhouse.spi.BricksProvider` with the implementation `com.codekaram.provider.House`. This relationship is represented in the graph by a dashed arrow from `com.codekaram.provider` to `com.codekaram.brickhouse.spi.BricksProvider`.

Before diving into the code for these providers, let's look at the module file information for the `com.codekaram.brickhouse` module:

```
module com.codekaram.brickhouse {
    uses com.codekaram.brickhouse.spi.BricksProvider;
    exports com.codekaram.brickhouse.spi;
}
```

The `loadLayers` class will not only handle forming layers, but also be able to load the service providers for each level. That's a bit of a simplification, but it helps us to better understand the flow. Now, let's examine the `loadLayers` implementation. Here's the creation of the layers' code based on the sample code from the "Working Example: JAR Hell" section:

```
static ModuleLayer getProviderLayer(String getCustomDir) {
   ModuleFinder finder = ModuleFinder.of(Paths.get(getCustomDir));
   ModuleLayer parent = ModuleLayer.boot();
   Configuration cf = parent.configuration().resolve(finder,
     ModuleFinder.of(), Set.of("com.codekaram.provider"));
   ClassLoader scl = ClassLoader.getSystemClassLoader();
   ModuleLayer layer = parent.defineModulesWithOneLoader(cf, scl);
   System.out.println("Created a new layer for " + layer);
   return layer;
}
```

If we simply want to create two layers, one for house version `basic` and another for house version `high-quality`, all we have to do is call `getProviderLayer()` (from the `main` method):

```
doWork(Stream.of(args)
       .map(getCustomDir -> getProviderLayer(getCustomDir)));
```

If we pass the two directories `basic` and `high-quality` as runtime parameters, the `getProviderLayer()` method will look for `com.codekaram.provider` in those directories and then create a layer for each. Let's examine the output (the line numbers have been added for the purpose of clarity and explanation):

```
1  $ java --module-path mods -m com.codekaram.brickhouse/
   com.codekaram.brickhouse.loadLayers basic high-quality

2  Created a new layer for com.codekaram.provider

3  I am the basic provider

4  Created a new layer for com.codekaram.provider

5  I am the high-quality provider
```

- Line 1 is our command-line argument with `basic` and `high-quality` as directories that provide the implementation of the `BrickProvider` service.
- Lines 2 and 4 are outputs indicating that `com.codekaram.provider` was found in both the directories and a new layer was created for each.
- Lines 3 and 5 are the output of `provider.getName()` as implemented in the `doWork()` code:

```
private static void doWork(Stream<ModuleLayer> myLayers){
myLayers.flatMap(moduleLayer -> ServiceLoader
   .load(moduleLayer, BricksProvider.class)
    .stream().map(ServiceLoader.Provider::get))
  .forEach(eachSLProvider -> System.out.println("I am the " + eachSLProvider.getName() +
" provider"));}
```

In `doWork()`, we first create a service loader for the `BricksProvider` service and load the provider from the module layer. We then print the return `String` of the `getName()` method for that provider. As seen in the output, we have two module layers and we were successful in printing the `I am the basic provider` and `I am the high-quality provider` outputs, where `basic` and `high-quality` are the return strings of the `getName()` method.

Now, let's visualize the workings of the four layers that we discussed earlier. To do so, we'll create a simple problem statement that builds a quote for basic and high-quality bricks for both levels of the house. First, we add the following code to our `main()` method:

```
int[] level = {1,2};
IntStream levels = Arrays.stream(level);
```

Next, we stream `doWork()` as follows:

```
levels.forEach(levelcount -> loadLayers
        .doWork(…
```

We now have four layers similar to those mentioned earlier (*house ver1.b*, *house ver1.hq*, *house ver2.b*, and *house ver2.hq*). Here's the updated output:

```
Created a new layer for com.codekaram.provider
My basic 1 level house will need 18000 bricks and those will cost me $6120
Created a new layer for com.codekaram.provider
My high-quality 1 level house will need 18000 bricks and those will cost me $9000
Created a new layer for com.codekaram.provider
My basic 2 level house will need 36000 bricks and those will cost me $12240
Created a new layer for com.codekaram.provider
My high-quality 2 level house will need 36000 bricks and those will be over my budget of $15000
```

NOTE The return string of the `getName()` methods for our providers has been changed to return just the `"basic"` and `"high-quality"` strings instead of an entire sentence.

The variation in the last line of the updated output serves as a demonstration of how additional conditions can be applied to service providers. Here, a budget constraint check has been integrated into the high-quality provider's implementation for a two-level house. You can, of course, customize the output and conditions as per your requirements.

Here's the updated `doWork()` method to handle both the level and the provider, along with the relevant code in the main method:

```
private static void doWork(int level, Stream<ModuleLayer> myLayers){
    myLayers.flatMap(moduleLayer -> ServiceLoader
        .load(moduleLayer, BricksProvider.class)
        .stream().map(ServiceLoader.Provider::get))
    .forEach(eachSLProvider -> System.out.println("My " + eachSLProvider.getName()
    + " " + level + " level house will need " + eachSLProvider.getBricksQuote(level)));
}
```

```
public static void main(String[] args) {
    int[] levels = {1, 2};
    IntStream levelStream = Arrays.stream(levels);

    levelStream.forEach(levelcount -> doWork(levelcount, Stream.of(args)
        .map(getCustomDir -> getProviderLayer(getCustomDir))));
}
```

Now, we can calculate the number of bricks and their cost for different levels of the house using the `basic` and `high-quality` implementations, with a separate module layer being devoted to each implementation. This demonstrates the power and flexibility that module layers provide, by enabling you to dynamically load and unload different implementations of a service without affecting other parts of your application.

Remember to adjust the service providers' code based on your specific use case and requirements. The example provided here is just a starting point for you to build on and adapt as needed.

In summary, this example illustrates the utility of Java module layers in creating applications that are both adaptable and scalable. By using the concepts of module layers and the Java `ServiceLoader`, you can create extensible applications, allowing you to adapt those applications to different requirements and conditions without affecting the rest of your codebase.

Open Services Gateway Initiative

The Open Services Gateway Initiative (OSGi) has been an alternative module system available to Java developers since 2000, long before the introduction of Jigsaw and Java module layers. As there was no built-in standard module system in Java at the time of OSGi's emergence, it addressed many modularity problems differently compared to Project Jigsaw. In this section, with insights from Nikita, whose expertise in Java modularity encompasses OSGi, we will compare Java module layers and OSGi, highlighting their similarities and differences.

OSGi Overview

OSGi is a mature and widely used framework that provides modularity and extensibility for Java applications. It offers a dynamic component model, which allows developers to create, update, and remove modules (called bundles) at runtime without restarting the application.

Similarities

- **Modularity:** Both Java module layers and OSGi promote modularity by enforcing a clear separation between components, making it easier to maintain, extend, and reuse code.
- **Dynamic loading:** Both technologies support dynamic loading and unloading of modules or bundles, allowing developers to update, extend, or remove components at runtime without affecting the rest of the application.

- **Service abstraction:** Both Java module layers (with the `ServiceLoader`) and OSGi provide service abstractions that enable loose coupling between components. This allows for greater flexibility when switching between different implementations of a service.

Differences

- **Maturity:** OSGi is a more mature and battle-tested technology, with a rich ecosystem and tooling support. Java module layers, which were introduced in JDK 9, are comparatively newer and may not have the same level of tooling and library support as OSGi.
- **Integration with Java platform:** Java module layers are a part of the Java platform, providing a native solution for modularity and extensibility. OSGi, by contrast, is a separate framework that builds on top of the Java platform.
- **Complexity:** OSGi can be more complex than Java module layers, with a steeper learning curve and more advanced features. Java module layers, while still providing powerful functionality, may be more straightforward and easier to use for developers who are new to modularity concepts.
- **Runtime environment:** OSGi applications run inside an OSGi container, which manages the life cycle of the bundles and enforces modularity rules. Java module layers run directly on the Java platform, with the module system handling the loading and unloading of modules.
- **Versioning:** OSGi provides built-in support for multiple versions of a module or bundle, allowing developers to deploy and run different versions of the same component concurrently. This is achieved by qualifying modules with versions and applying "uses constraints" to ensure safe class namespaces exist for each module. However, dealing with versions in OSGi can introduce unnecessary complexity for module resolution and for end users. In contrast, Java module layers do not natively support multiple versions of a module, but you can achieve similar functionality by creating separate module layers for each version.
- **Strong encapsulation:** Java module layers, as first-class citizens in the JDK, provide strong encapsulation by issuing error messages when unauthorized access to non-exported functionality occurs, even via reflection. In OSGi, non-exported functionality can be "hidden" using class loaders, but the module internals are still available for reflection access unless a special security manager is set. OSGi was limited by pre-JPMS features of Java SE and could not provide the same level of strong encapsulation as Java module layers.

When it comes to achieving modularity and extensibility in Java applications, developers typically have two main options: Java module layers and OSGi. Remember, the choice between Java module layers and OSGi is not always binary and can depend on many factors. These include the specific requirements of your project, the existing technology stack, and your team's familiarity with the technologies. Also, it's worth noting that Java module layers and OSGi are not the only options for achieving modularity in Java applications. Depending on your specific needs and context, other solutions might be more appropriate. It is crucial to thoroughly evaluate the pros and cons of all available options before making a decision for your project. Your choice should be based on the specific demands and restrictions of your project to ensure optimal outcomes.

On the one hand, if you need advanced features like multiple version support and a dynamic component model, OSGi may be the better option for you. This technology is ideal for complex applications that require both flexibility and scalability. However, it can be more difficult to learn and implement than Java module layers, so it may not be the best choice for developers who are new to modularity.

On the other hand, Java module layers offer a more straightforward solution for achieving modularity and extensibility in your Java applications. This technology is built into the Java platform itself, which means that developers who are already familiar with Java should find it relatively easy to use. Additionally, Java module layers offer strong encapsulation features that can help prevent dependencies from bleeding across different modules.

Introduction to Jdeps, Jlink, Jdeprscan, and Jmod

This section covers four tools that aid in the development and deployment of modular applications: *jdeps*, *jlink*, *jdeprscan*, and *jmod*. Each of these tools serves a unique purpose in the process of building, analyzing, and deploying Java applications.

Jdeps

Jdeps is a tool that facilitates analysis of the dependencies of Java classes or packages. It's particularly useful when you're trying to create a module file for JAR files. With *jdeps*, you can create various filters using regular expressions (*regex*); a regular expression is a sequence of characters that forms a search pattern. Here's how you can use *jdeps* on the `loadLayers` class:

```
$ jdeps mods/com.codekaram.brickhouse/com/codekaram/brickhouse/loadLayers.class
loadLayers.class -> java.base
loadLayers.class -> not found
   com.codekaram.brickhouse -> com.codekaram.brickhouse.spi   not found
   com.codekaram.brickhouse -> java.io                        java.base
   com.codekaram.brickhouse -> java.lang                      java.base
   com.codekaram.brickhouse -> java.lang.invoke               java.base
   com.codekaram.brickhouse -> java.lang.module               java.base
   com.codekaram.brickhouse -> java.nio.file                  java.base
   com.codekaram.brickhouse -> java.util                      java.base
   com.codekaram.brickhouse -> java.util.function             java.base
   com.codekaram.brickhouse -> java.util.stream               java.base
```

The preceding command has the same effect as passing the option `-verbose:package` to *jdeps*. The `-verbose` option by itself will list all the dependencies:

```
$ jdeps -v mods/com.codekaram.brickhouse/com/codekaram/brickhouse/loadLayers.class
loadLayers.class -> java.base
loadLayers.class -> not found
   com.codekaram.brickhouse.loadLayers -> com.codekaram.brickhouse.spi.BricksProvider  not found
   com.codekaram.brickhouse.loadLayers -> java.io.PrintStream                  java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.Class                      java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.ClassLoader                 java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.ModuleLayer                 java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.NoSuchMethodException       java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.Object                      java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.String                      java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.System                      java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.invoke.CallSite             java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.invoke.LambdaMetafactory    java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandle         java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandles        java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandles$Lookup java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodType           java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.invoke.StringConcatFactory  java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.module.Configuration        java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.module.ModuleFinder         java.base
   com.codekaram.brickhouse.loadLayers -> java.nio.file.Path                    java.base
   com.codekaram.brickhouse.loadLayers -> java.nio.file.Paths                   java.base
   com.codekaram.brickhouse.loadLayers -> java.util.Arrays                      java.base
   com.codekaram.brickhouse.loadLayers -> java.util.Collection                  java.base
   com.codekaram.brickhouse.loadLayers -> java.util.ServiceLoader               java.base
   com.codekaram.brickhouse.loadLayers -> java.util.Set                         java.base
   com.codekaram.brickhouse.loadLayers -> java.util.Spliterator                 java.base
   com.codekaram.brickhouse.loadLayers -> java.util.function.Consumer           java.base
   com.codekaram.brickhouse.loadLayers -> java.util.function.Function           java.base
   com.codekaram.brickhouse.loadLayers -> java.util.function.IntConsumer        java.base
   com.codekaram.brickhouse.loadLayers -> java.util.function.Predicate          java.base
   com.codekaram.brickhouse.loadLayers -> java.util.stream.IntStream            java.base
   com.codekaram.brickhouse.loadLayers -> java.util.stream.Stream               java.base
   com.codekaram.brickhouse.loadLayers -> java.util.stream.StreamSupport        java.base
```

Jdeprscan

Jdeprscan is a tool that analyzes the usage of deprecated APIs in modules. Deprecated APIs are older APIs that the Java community has replaced with newer ones. These older APIs are still supported but are marked for removal in future releases. *Jdeprscan* helps developers maintain their code by suggesting alternative solutions to these deprecated APIs, aiding them in transitioning to newer, supported APIs.

Here's how you can use *jdeprscan* on the `com.codekaram.brickhouse` module:

```
$ jdeprscan --for-removal mods/com.codekaram.brickhouse
No deprecated API marked for removal found.
```

In this example, *jdeprscan* is used to scan the `com.codekaram.brickhouse` module for deprecated APIs that are marked for removal. The output indicates that no such deprecated APIs are found.

You can also use `--list` to see all deprecated APIs in a module:

```
$ jdeprscan --list mods/com.codekaram.brickhouse
No deprecated API found.
```

In this case, no deprecated APIs are found in the `com.codekaram.brickhouse` module.

Jmod

Jmod is a tool used to create, describe, and list JMOD files. JMOD files are an alternative to JAR files for packaging modular Java applications, which offer additional features such as native code and configuration files. These files can be used for distribution or to create custom runtime images with *jlink*.

Here's how you can use *jmod* to create a JMOD file for the `brickhouse` example. Let's first compile and package the module specific to this example:

```
$ javac --module-source-path src -d build/modules $(find src -name "*.java")
$ jmod create --class-path build/modules/com.codekaram.brickhouse com.codekaram.brickhouse.jmod
```

Here, the `jmod create` command is used to create a JMOD file named `com.codekaram.brickhouse.jmod` from the `com.codekaram.brickhouse` module located in the `build/modules` directory. You can then use the `jmod describe` command to display information about the JMOD file:

```
$ jmod describe com.codekaram.brickhouse.jmod
```

This command will output the module descriptor and any additional information about the JMOD file.

Additionally, you can use the `jmod list` command to display the contents of the created JMOD file:

```
$ jmod list com.codekaram.brickhouse.jmod
com/codekaram/brickhouse/
com/codekaram/brickhouse/loadLayers.class
com/codekaram/brickhouse/loadLayers$1.class
…
```

The output lists the contents of the `com.codekaram.brickhouse.jmod` file, showing the package structure and class files.

By using *jmod* to create JMOD files, describe their contents, and list their individual files, you can gain a better understanding of your modular application's structure and streamline the process of creating custom runtime images with *jlink*.

Jlink

Jlink is a tool that helps link modules and their transitive dependencies to create custom modular runtime images. These custom images can be packaged and deployed without needing the entire Java Runtime Environment (JRE), which makes your application lighter and faster to start.

To use the *jlink* command, this tool needs to be added to your path. First, ensure that the `$JAVA_HOME/bin` is in the path. Next, type `jlink` on the command line:

```
$ jlink
Error: --module-path must be specified
Usage: jlink <options> --module-path <modulepath> --add-modules <module>[,<module>...]
Use --help for a list of possible options
```

Here's how you can use *jlink* for the code shown in "Implementing Modular Services with JDK 17":

```
$ jlink --module-path $JAVA_HOME/jmods:build/modules --add-modules com.example.builder --output
consumer.services --bind-services
```

A few notes on this example:

- The command includes a directory called `$JAVA_HOME/jmods` in the `module-path`. This directory contains the `java.base.jmod` needed for all application modules.
- Because the module is a consumer of services, it's necessary to link the service providers (and their dependencies). Hence, the –`bind-services` option is used.
- The runtime image will be available in the `consumer.services` directory as shown here:

  ```
  $ ls consumer.services/
  bin conf      include   legal     lib       release
  ```

Let's now run the image:

```
$ consumer.services/bin/java -m com.example.builder/com.example.builder.Builder
Building a house with bricks...
```

With *jlink*, you can create lightweight, custom, stand-alone runtime images tailored to your modular Java applications, thereby simplifying the deployment and reducing the size of your distributed application.

Conclusion

This chapter has undertaken a comprehensive exploration of Java modules, tools, and techniques to create and manage modular applications. We have delved into the Java Platform Module System (JPMS), highlighting its benefits such as reliable configuration and strong encapsulation. These features contribute to more maintainable and scalable applications.

We navigated the intricacies of creating, packaging, and managing modules, and explored the use of module layers to enhance application flexibility. These practices can help address common challenges faced when migrating to newer JDK versions (e.g., JDK 11 or JDK 17), including updating project structures and ensuring dependency compatibility.

Performance Implications

The use of modular Java carries significant performance implications. By including only the necessary modules in your application, the JVM loads fewer classes, which improves start-up performance and reduces the memory footprint. This is particularly beneficial in resource-limited environments such as microservices running in containers. However, it is important to note that while modularity can improve performance, it also introduces a level of complexity. For instance, improper module design can lead to cyclic dependencies,[2] negatively impacting performance. Therefore, careful design and understanding of modules are essential to fully reap the performance benefits.

Tools and Future Developments

We examined the use of powerful tools like *jdeps*, *jdeprscan*, *jmod*, and *jlink*, which are instrumental in identifying and addressing compatibility issues, creating custom runtime images, and streamlining the deployment of modular applications. Looking ahead, we can anticipate more advanced options for creating custom runtime images with *jlink*, and more detailed and accurate dependency analysis with *jdeps*.

As more developers adopt modular Java, new best practices and patterns will emerge, alongside new tools and libraries designed to work with JPMS. The Java community is continuously improving JPMS, with future Java versions expected to refine and expand its capabilities.

Embracing the Modular Programming Paradigm

Transitioning to modular Java can present unique challenges, especially in understanding and implementing modular structures in large-scale applications. Compatibility issues may arise with third-party libraries or frameworks that may not be fully compatible with JPMS. These challenges, while part of the journey toward modernization, are often outweighed by the benefits of modular Java, such as improved performance, enhanced scalability, and better maintainability.

In conclusion, by leveraging the knowledge gained from this chapter, you can confidently migrate your projects and fully harness the potential of modular Java applications. The future of modular Java is exciting, and embracing this paradigm will equip you to meet the evolving needs of the software development landscape. It's an exciting time to be working with modular Java, and we look forward to seeing how it evolves and shapes the future of robust and efficient Java applications.

[2] https://openjdk.org/projects/jigsaw/spec/issues/#CyclicDependences

Index