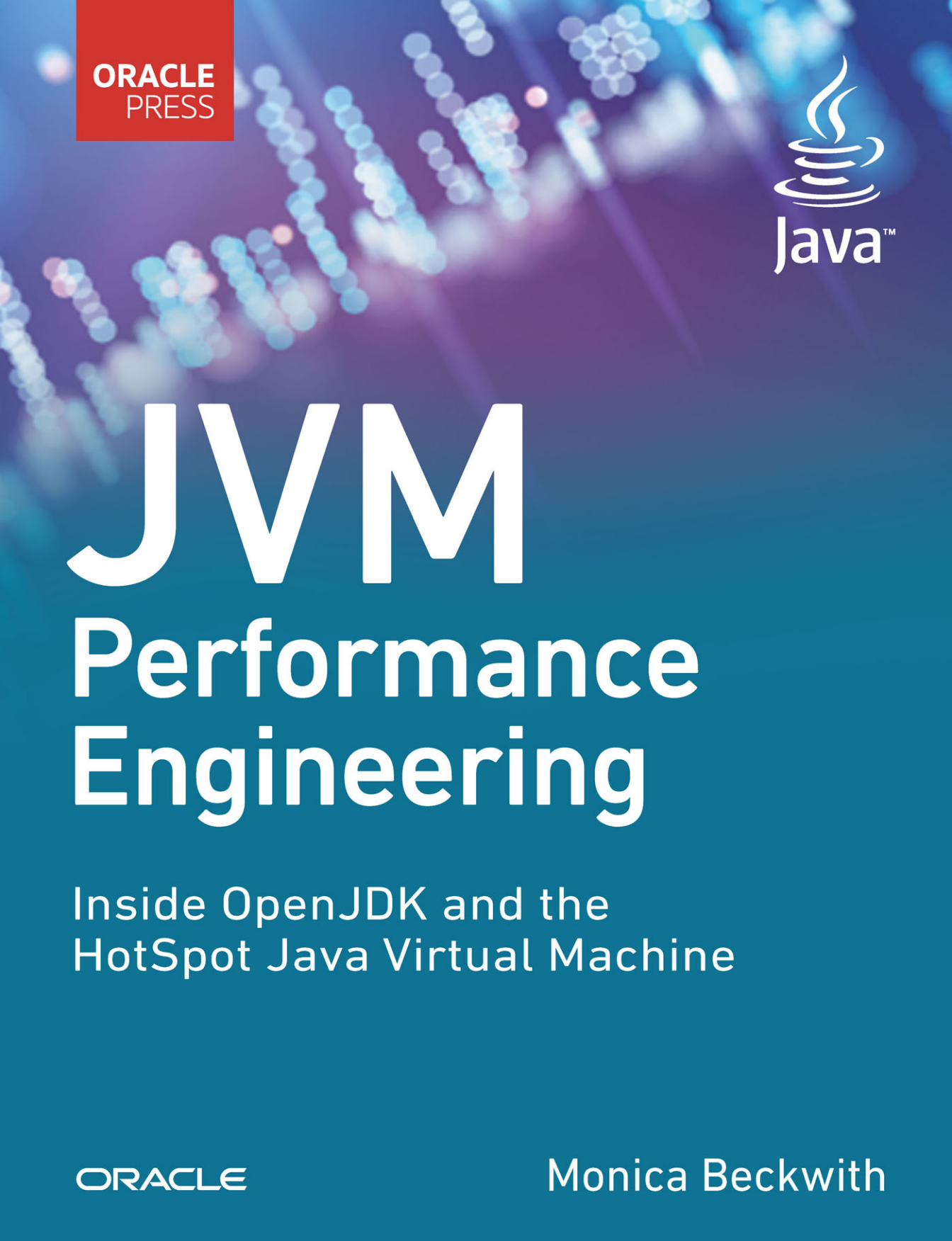
ORACLE
PRESS
Java™
JVM
Performance
Engineering
Inside OpenJDK and the
HotSpot Java Virtual Machine
ORACLE
Monica Beckwith

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

Visit us on the Web: informit.com/aw

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

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 1

The Performance Evolution of Java: The Language and the Virtual Machine

More than three decades ago, the programming languages landscape was largely defined by C and its object-oriented extension, C++. In this period, the world of computing was undergoing a significant shift from large, cumbersome mainframes to smaller, more efficient minicomputers. C, with its suitability for Unix systems, and C++, with its innovative introduction of classes for object-oriented design, were at the forefront of this technological evolution.

However, as the industry started to shift toward more specialized and cost-effective systems, such as microcontrollers and microcomputers, a new set of challenges emerged. Applications were ballooning in terms of lines of code, and the need to "port" software to various platforms became an increasingly pressing concern. This often necessitated rewriting or heavily modifying the application for each specific target, a labor-intensive and error-prone process. Developers also faced the complexities of managing numerous static library dependencies and the demand for lightweight software on embedded systems—areas where C++ fell short.

It was against this backdrop that Java emerged in the mid-1990s. Its creators aimed to fill this niche by offering a "write once, run anywhere" solution. But Java was more than just a programming language. It introduced its own runtime environment, complete with a virtual machine (Java Virtual Machine [JVM]), class libraries, and a comprehensive set of tools. This all-encompassing ecosystem, known as the Java Development Kit (JDK), was designed to tackle the challenges of the era and set the stage for the future of programming. Today, more than a quarter of a century later, Java's influence in the world of programming languages remains strong, a testament to its adaptability and the robustness of its design.

The performance of applications emerged as a critical factor during this time, especially with the rise of large-scale, data-intensive applications. The evolution of Java's runtime system has played a pivotal role in addressing these performance challenges. Thanks to the optimization in generics, autoboxing and unboxing, and enhancements to the concurrency utilities, Java applications have seen significant improvements in both performance and scalability. Moreover, the changes have had far-reaching implications for the performance of the JVM itself. In

particular, the JVM has had to adapt and optimize its execution strategies to efficiently handle these new language features. As you read this book, bear in mind the historical context and the driving forces that led to Java's inception. The evolution of Java and its virtual machine have profoundly influenced the way developers write and optimize software for various platforms.

In this chapter, we will thoroughly examine the history of Java and JVM, highlighting the technological advancements and key milestones that have significantly shaped its development. From its early days as a solution for platform independence, through the introduction of new language features, to the ongoing improvements to the JVM, Java has evolved into a powerful and versatile tool in the arsenal of modern software development.

A New Ecosystem Is Born

In the 1990s, the internet was emerging, and web pages became more interactive with the introduction of Java applets. Java applets were small applications that ran within web browsers, providing a "real-time" experience for end users.

Applets were not only platform independent but also "secure," in the sense that the user needed to trust the applet writer. When discussing security in the context of the JVM, it's essential to understand that direct access to memory should be forbidden. As a result, Java introduced its own memory management system, called the garbage collector (GC).

> **NOTE** In this book, the acronym *GC* is used to refer to both *garbage collection,* the process of automatic memory management, and *garbage collector*, the module within the JVM that performs this process. The specific meaning will be clear based on the context in which *GC* is used.

Additionally, an abstraction layer, known as Java bytecode, was added to any executable. Java applets quickly gained popularity because their bytecode, residing on the web server, would be transferred and executed as its own process during web page rendering. Although the Java bytecode is platform independent, it is interpreted and compiled into native code specific to the underlying platform.

A Few Pages from History

The JDK included tools such as a Java compiler that translated Java code into Java bytecode. Java bytecode is the executable handled by the Java Runtime Environment (JRE). Thus, for different environments, only the runtime needed to be updated. As long as a JVM for a specific environment existed, the bytecode could be executed. The JVM and the GC served as the execution engines. For Java versions 1.0 and 1.1, the bytecode was interpreted to the native machine code, and there was no dynamic compilation.

Soon after the release of Java versions 1.0 and 1.1, it became apparent that Java needed to be more performant. Consequently, a just-in-time (JIT) compiler was introduced in Java 1.2. When

combined with the JVM, it provided dynamic compilation based on hot methods and loop-back branch counts. This new VM was called the Java HotSpot VM.

Understanding Java HotSpot VM and Its Compilation Strategies

The Java HotSpot VM plays a critical role in executing Java programs efficiently. It includes JIT compilation, tiered compilation, and adaptive optimization to improve the performance of Java applications.

The Evolution of the HotSpot Execution Engine

The HotSpot VM performs *mixed-mode* execution, which means that the VM starts in interpreted mode, with the bytecode being converted into native code based on a description table. The table has a template of native code corresponding to each bytecode instruction known as the *TemplateTable*; it is just a simple lookup table. The execution code is stored in a code cache (known as *CodeCache*). *CodeCache* stores native code and is also a useful cache for storing JIT-ted code.

> **NOTE** HotSpot VM also provides an interpreter that doesn't need a template, called the C++ interpreter. Some OpenJDK ports[1] choose this route to simplify porting of the VM to non-x86 platforms.

Performance-Critical Methods and Their Optimization

Performance engineering is a critical aspect of software development, and a key part of this process involves identifying and optimizing performance-critical methods. These methods are frequently executed or contain performance-sensitive code, and they stand to gain the most from JIT compilation. Optimizing performance-critical methods is not just about choosing appropriate data structures and algorithms; it also involves identifying and optimizing the methods based on their frequency of invocation, size and complexity, and available system resources.

Consider the following `BookProgress` class as an example:

```
import java.util.*;

public class BookProgress {
    private String title;
    private Map<String, Integer> chapterPages;
    private Map<String, Integer> chapterPagesWritten;

    public BookProgress(String title) {
        this.title = title;
```

[1] https://wiki.openjdk.org/pages/viewpage.action?pageId=13729802

```
        this.chapterPages = new HashMap<>();
        this.chapterPagesWritten = new HashMap<>();
    }

    public void addChapter(String chapter, int totalPages) {
        this.chapterPages.put(chapter, totalPages);
        this.chapterPagesWritten.put(chapter, 0);
    }

    public void updateProgress(String chapter, int pagesWritten) {
        this.chapterPagesWritten.put(chapter, pagesWritten);
    }

    public double getProgress(String chapter) {
        return ((double) chapterPagesWritten.get(chapter) / chapterPages.get(chapter)) * 100;
    }

    public double getTotalProgress() {
        int totalWritten = chapterPagesWritten.values().stream().mapToInt(Integer::intValue).sum();
        int total = chapterPages.values().stream().mapToInt(Integer::intValue).sum();
        return ((double) totalWritten / total) * 100;
    }
}

public class Main {
    public static void main(String[] args) {
        BookProgress book = new BookProgress("JVM Performance Engineering");
        String[] chapters = {
            "Performance Evolution",
            "Performance and Type System",
            "Monolithic to Modular",
            "Unified Logging System",
            "End-to-End Performance Optimization",
            "Advanced Memory Management",
            "Runtime Performance Optimization",
            "Accelerating Startup",
            "Harnessing Exotic Hardware"
        };
        for (String chapter : chapters) {
            book.addChapter(chapter, 100);
        }
        for (int i = 0; i < 50; i++) {
            for (String chapter : chapters) {
                int currentPagesWritten = book.chapterPagesWritten.get(chapter);
                if (currentPagesWritten < 100) {
                    book.updateProgress(chapter, currentPagesWritten + 2);
                    double progress = book.getProgress(chapter);
```

```
                    System.out.println("Progress for chapter " + chapter + ": " + progress + "%");
                }
            }
        }
        System.out.println("Total book progress: " + book.getTotalProgress() + "%");
    }
}
```

In this code, we've defined a `BookProgress` class to track the progress of writing a book, which is divided into chapters. Each chapter has a total number of pages and a current count of pages written. The class provides methods to add chapters, update progress, and calculate the progress of each chapter and the overall book.

The `Main` class creates a `BookProgress` object for a book titled "JVM Performance Engineering." It adds nine chapters, each with 100 pages, and simulates writing the book by updating the progress of each chapter in a round-robin fashion, writing two pages at a time. After each update, it calculates and prints the progress of the current chapter and, once all pages are written, the overall progress of the book.

The `getProgress(String chapter)` and `updateProgress(String chapter, int pagesWritten)` methods are identified as performance-critical methods. Their frequent invocation makes them prime candidates for optimization by the HotSpot VM, illustrating how certain methods in a program may require more attention for performance optimization due to their high frequency of use.

Interpreter and JIT Compilation

The HotSpot VM provides an interpreter that converts bytecode into native code based on the *TemplateTable*. Interpretation is the first step in adaptive optimization offered by this VM and is considered the slowest form of bytecode execution. To make the execution faster, the HotSpot VM utilizes adaptive JIT compilation. The JIT-optimized code replaces the template code for methods that are identified as performance critical.

As mentioned in the previous section, the HotSpot VM monitors executed code for performance-critical methods based on two key metrics—method entry counts and loop-back branch counts. The VM assigns call counters to individual methods in the Java application. When the entry count exceeds a preestablished value, the method or its callee is chosen for asynchronous JIT compilation. Similarly, there is a counter for each loop in the code. Once the HotSpot VM determines that the loop-back branches (also known as loop-back edges) have crossed their threshold, the JIT optimizes that particular loop. This optimization is called on-stack replacement (OSR). With OSR, only the loop for which the loop-back branch counter overflowed will be compiled and replaced asynchronously on the execution stack.

Print Compilation

A very handy command-line option that can help us better understand adaptive optimization in the HotSpot VM is `-XX:+PrintCompilation`. This option also returns information on different optimized compilation levels, which are provided by an adaptive optimization called tiered compilation (discussed in the next subsection).

The output of the `-XX:+PrintCompilation` option is a log of the HotSpot VM's compilation tasks. Each line of the log represents a single compilation task and includes several pieces of information:

- The timestamp in milliseconds since the JVM started and this compilation task was logged.
- The unique identifier for this compilation task.
- Flags indicating certain properties of the method being compiled, such as whether it's an OSR method (%), whether it's synchronized (s), whether it has an exception handler (!), whether it's blocking (b), or whether it's native (n).
- The tiered compilation level, indicating the level of optimization applied to this method.
- The fully qualified name of the method being compiled.
- For OSR methods, the bytecode index where the compilation started. This is usually the start of a loop.
- The size of the method in the bytecode, in bytes.

Here are a few examples of the output of the `-XX:+PrintCompilation` option:

```
567  693 % !   3    org.h2.command.dml.Insert::insertRows @ 76 (513 bytes)
656  797     n 0    java.lang.Object::clone (native)
779  835  s    4    java.lang.StringBuffer::append (13 bytes)
```

These logs provide valuable insights into the behavior of the HotSpot VM's adaptive optimization, helping us understand how our Java applications are optimized at runtime.

Tiered Compilation

Tiered compilation, which was introduced in Java 7, provides multiple levels of optimized compilations, ranging from T0 to T4:

1. **T0:** Interpreted code, devoid of compilation. This is where the code starts and then moves on to the T1, T2, or T3 level.
2. **T1–T3:** Client-compiled mode. T1 is the first step where the method invocation counters and loop-back branch counters are used. At T2, the client compiler includes profiling information, referred to as profile-guided optimization; it may be familiar to readers who are conversant in static compiler optimizations. At the T3 compilation level, completely profiled code can be generated.
3. **T4:** The highest level of optimization provided by the HotSpot VM's server compiler.

Prior to tiered compilation, the server compiler would employ the interpreter to collect such profiling information. With the introduction of tiered compilation, the code reaches client compilation levels faster, and now the profiling information is generated by client-compiled methods themselves, providing better start-up times.

NOTE Tiered compilation has been enabled by default since Java 8.

Client and Server Compilers

The HotSpot VM provides two flavors of compilers: the fast client compiler (also known as the C1 compiler) and the server compiler (also known as the C2 compiler).

1. **Client compiler (C1):** Aims for fast start-up times in a client setup. The JIT invocation thresholds are lower for a client compiler than for a server compiler. This compiler is designed to compile code quickly, providing a fast start-up time, but the code it generates is less optimized.
2. **Server compiler (C2):** Offers many more adaptive optimizations and better thresholds geared toward higher performance. The counters that determine when a method/loop needs to be compiled are still the same, but the invocation thresholds are different (much lower) for a client compiler than for a server compiler. The server compiler takes longer to compile methods but produces highly optimized code that is beneficial for long-running applications. Some of the optimizations performed by the C2 compiler include *inlining* (replacing method invocations with the method's body), loop unrolling (increasing the loop body size to decrease the overhead of loop checks and to potentially apply other optimizations such as loop vectorization), dead code elimination (removing code that does not affect the program results), and range-check elimination (removing checks for index out-of-bounds errors if it can be assured that the array index never crosses its bounds). These optimizations help to improve the execution speed of the code and reduce the overhead of certain operations.[2]

Segmented Code Cache

As we delve deeper into the intricacies of the HotSpot VM, it's important to revisit the concept of the code cache. Recall that the code cache is a storage area for native code generated by the JIT compiler or the interpreter. With the introduction of tiered compilation, the code cache also becomes a repository for profiling information gathered at different levels of tiered compilation. Interestingly, even the *TemplateTable*, which the interpreter uses to look up the native code sequence for each bytecode, is stored in the code cache.

The size of the code cache is fixed at start-up but can be modified on the command line by passing the desired maximum value to `-XX:ReservedCodeCacheSize`. Prior to Java 7, the default value for this size was 48 MB. Once the code cache was filled up, all compilation would cease. This posed a significant problem when tiered compilation was enabled, as the code cache would contain not only JIT-compiled code (represented as *nmethod* in the HotSpot VM) but also profiled code. The *nmethod* refers to the internal representation of a Java method that has been compiled into machine code by the JIT compiler. In contrast, the profiled code is the code that has been analyzed and optimized based on its runtime behavior. The code cache needs

[2] "What the JIT!? Anatomy of the OpenJDK HotSpot VM." infoq.com.

to manage both of these types of code, leading to increased complexity and potential performance issues.

To address these problems, the default value for `ReservedCodeCacheSize` was increased to 240 MB in JDK 7 update 40. Furthermore, when the code cache occupancy crosses a preset `CodeCacheMinimumFreeSpace` threshold, the JIT compilation halts and the JVM runs a *sweeper*. The *nmethod* sweeper reclaims space by evacuating older compilations. However, sweeping the entire code cache data structure can be time-consuming, especially when the code cache is large and nearly full.

Java 9 introduced a significant change to the code cache: It was segmented into different regions based on the type of code. This not only reduced the sweeping time but also minimized fragmentation of the long-lived code by shorter-lived code. Co-locating code of the same type also reduced hardware-level instruction cache misses.

The current implementation of the segmented code cache includes the following regions:

- **Non-method code heap region:** This region is reserved for VM internal data structures that are not related to Java methods. For example, the *TemplateTable*, which is a VM internal data structure, resides here. This region doesn't contain compiled Java methods.
- **Non-profiled *nmethod* code heap:** This region contains Java methods that have been compiled by the JIT compiler without profiling information. These methods are fully optimized and are expected to be long-lived, meaning they won't be recompiled frequently and may need to be reclaimed only infrequently by the sweeper.
- **Profiled *nmethod* code heap:** This region contains Java methods that have been compiled with profiling information. These methods are not as optimized as those in the non-profiled region. They are considered transient because they can be recompiled into more optimized versions and moved to the non-profiled region as more profiling information becomes available. They can also be reclaimed by the sweeper as often as needed.

Each of these regions has a fixed size that can be set by their respective command-line options:

Heap Region Type	Size Command-Line Option
Non-method code heap	`-XX:NonMethodCodeHeapSize`
Non-profiled *nmethod* code heap	`-XX:NonProfiledCodeHeapSize`
Profiled *nmethod* code heap	`-XX:ProfiledCodeHeapSize`

Going forward, the hope is that the segmented code caches can accommodate additional code regions for heterogeneous code such as ahead-of-time (AOT)–compiled code and code for hardware accelerators.[3] There's also the expectation that the fixed sizing thresholds can be upgraded to utilize adaptive resizing, thereby avoiding wastage of memory.

[3] JEP 197: *Segmented Code Cache.* https://openjdk.org/jeps/197.

Adaptive Optimization and Deoptimization

Adaptive optimization allows the HotSpot VM runtime to optimize the interpreted code into compiled code or insert an optimized loop on the stack (so we could have something like an "interpreted to compiled, and back to interpreted" code execution sequence). There is another major advantage of adaptive optimization, however—in deoptimization of code. That means the compiled code could go back to being interpreted, or a higher-optimized code sequence could be rolled back into a less-optimized sequence.

Dynamic deoptimization helps Java reclaim code that may no longer be relevant. A few example use cases are when checking interdependencies during dynamic class loading, when dealing with polymorphic call sites, and when reclaiming less-optimized code. Deoptimization will first make the code "not entrant" and eventually reclaim it after marking it as "zombie" code.[4]

Deoptimization Scenarios

Deoptimization can occur in several scenarios when working with Java applications. In this section, we'll explore two of these scenarios.

Class Loading and Unloading

Consider an application containing two classes, `Car` and `DriverLicense`. The `Car` class requires a `DriverLicense` to enable drive mode. The JIT compiler optimizes the interaction between these two classes. However, if a new version of the `DriverLicense` class is loaded due to changes in driving regulations, the previously compiled code may no longer be valid. This necessitates deoptimization to revert to the interpreted mode or a less-optimized state. This allows the application to employ the new version of the `DriverLicense` class.

Here's an example code snippet:

```
class Car {
    private DriverLicense driverLicense;

    public Car(DriverLicense driverLicense) {
        this.driverLicense = driverLicense;
    }

    public void enableDriveMode() {
        if (driverLicense.isAdult()) {
            System.out.println("Drive mode enabled!");
        } else if (driverLicense.isTeenDriver()) {
            if (driverLicense.isLearner()) {
                System.out.println("You cannot drive without a licensed adult's supervision.");
            } else {
                System.out.println("Drive mode enabled!");
            }
        } else {
```

[4] https://www.infoq.com/articles/OpenJDK-HotSpot-What-the-JIT/

```
            System.out.println("You don't have a valid driver's license.");
        }
    }
}

class DriverLicense {
    private boolean isTeenDriver;
    private boolean isAdult;
    private boolean isLearner;

    public DriverLicense(boolean isTeenDriver, boolean isAdult, boolean isLearner) {
        this.isTeenDriver = isTeenDriver;
        this.isAdult = isAdult;
        this.isLearner = isLearner;
    }

    public boolean isTeenDriver() {
        return isTeenDriver;
    }

    public boolean isAdult() {
        return isAdult;
    }

    public boolean isLearner() {
        return isLearner;
    }
}

public class Main {
    public static void main(String[] args) {
        DriverLicense driverLicense = new DriverLicense(false, true, false);
        Car myCar = new Car(driverLicense);
        myCar.enableDriveMode();
    }
}
```

In this example, the `Car` class requires a `DriverLicense` to enable drive mode. The driver's license can be for an adult, a teen driver with a learner's permit, or a teen driver with a full license. The `enableDriveMode()` method checks the driver's license using the `isAdult()`, `isTeenDriver()`, and `isLearner()` methods, and prints the appropriate message to the console.

If a new version of the `DriverLicense` class is loaded, the previously optimized code may no longer be valid, triggering deoptimization. This allows the application to use the new version of the `DriverLicense` class without any issues.

Polymorphic Call Sites

Deoptimization can also occur when working with polymorphic call sites, where the actual method to be invoked is determined at runtime. Let's look at an example using the `DriverLicense` class:

```
abstract class DriverLicense {
    public abstract void drive();
}

class AdultLicense extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for driving responsibly as an adult");
    }
}

class TeenPermit extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for learning to drive responsibly as a teen");
    }
}

class SeniorLicense extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for being a valued senior citizen");
    }
}

public class Main {
    public static void main(String[] args) {
        DriverLicense license = new AdultLicense();
        license.drive(); // monomorphic call site

        // Changing the call site to bimorphic
        if (Math.random() < 0.5) {
            license = new AdultLicense();
        } else {
            license = new TeenPermit();
        }
        license.drive(); // bimorphic call site

        // Changing the call site to megamorphic
        for (int i = 0; i < 100; i++) {
            if (Math.random() < 0.33) {
                license = new AdultLicense();
```

```
            } else if (Math.random() < 0.66) {
                license = new TeenPermit();
            } else {
                license = new SeniorLicense();
            }
            license.drive(); // megamorphic call site
        }
    }
}
```

In this example, the abstract `DriverLicense` class has three subclasses: `AdultLicense`, `TeenPermit`, and `SeniorLicense`. The `drive()` method is overridden in each subclass with different implementations.

First, when we assign an `AdultLicense` object to a `DriverLicense` variable and call `drive()`, the HotSpot VM optimizes the call site to a monomorphic call site and caches the target method address in an *inline cache* (a structure to track the call site's type profile).

Next, we change the call site to a bimorphic call site by randomly assigning an `AdultLicense` or `TeenPermit` object to the `DriverLicense` variable and calling `drive()`. Because there are two possible types, the VM can no longer use the monomorphic dispatch mechanism, so it switches to the bimorphic dispatch mechanism. This change does not require deoptimization—and still provides a performance boost by reducing the number of virtual method dispatches needed at the call site.

Finally, we change the call site to a megamorphic call site by randomly assigning an `AdultLicense`, `TeenPermit`, or `SeniorLicense` object to the `DriverLicense` variable and calling `drive()` 100 times. As there are now three possible types, the VM cannot use the bimorphic dispatch mechanism and must switch to the megamorphic dispatch mechanism. This change also does not require deoptimization.

However, if we were to introduce a new subclass `InternationalLicense` and change the call site to include it, the VM could potentially deoptimize the call site and switch to a megamorphic or polymorphic call site to handle the new type. This change is necessary because the VM's type profiling information for the call site would be outdated, and the previously optimized code would no longer be valid.

Here's the code snippet for the new subclass and the updated call site:

```
class InternationalLicense extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for driving responsibly as an international driver");
    }
}

// Updated call site
for (int i = 0; i < 100; i++) {
```

```
    if (Math.random() < 0.25) {
        license = new AdultLicense();
    } else if (Math.random() < 0.5) {
        license = new TeenPermit();
    } else if (Math.random() < 0.75) {
        license = new SeniorLicense();
    } else {
        license = new InternationalLicense();
    }
    license.drive(); // megamorphic call site with a new type
}
```

HotSpot Garbage Collector: Memory Management Unit

A crucial component of the HotSpot execution engine is its memory management unit, commonly known as the garbage collector (GC). HotSpot provides multiple garbage collection algorithms that cater to a trifecta of performance aspects: application responsiveness, throughput, and overall footprint. *Responsiveness* refers to the time taken to receive a response from the system after sending a stimulus. *Throughput* measures the number of operations that can be performed per second on a given system. *Footprint* can be defined in two ways: as optimizing the amount of data or objects that can fit into the available space and as removing redundant information to save space.

Generational Garbage Collection, Stop-the-World, and Concurrent Algorithms

OpenJDK offers a variety of generational GCs that utilize different strategies to manage memory, with the common goal of improving application performance. These collectors are designed based on the principle that "most objects die young," meaning that most newly allocated objects on the Java heap are short-lived. By taking advantage of this observation, generational GCs aim to optimize memory management and significantly reduce the negative impact of garbage collection on the performance of the application.

Heap collection in GC terms involves identifying live objects, reclaiming space occupied by garbage objects, and, in some cases, compacting the heap to reduce fragmentation. Fragmentation can occur in two ways: (1) internal fragmentation, where allocated memory blocks are larger than necessary, leaving wasted space within the blocks; and (2) external fragmentation, where memory is allocated and deallocated in such a way that free memory is divided into noncontiguous blocks. External fragmentation can lead to inefficient memory use and potential allocation failures. Compaction is a technique used by some GCs to combat external fragmentation; it involves moving objects in memory to consolidate free memory into a single contiguous block. However, compaction can be a costly operation in terms of CPU usage and can cause lengthy pause times if it's done as a stop-the-world operation.

The OpenJDK GCs employ several different GC algorithms:

- **Stop-the-world (STW) algorithms:** STW algorithms pause application threads for the entire duration of the garbage collection work. Serial, Parallel, (Mostly) Concurrent Mark and Sweep (CMS), and Garbage First (G1) GCs use STW algorithms in specific phases of their collection cycles. The STW approach can result in longer pause times when the heap fills up and runs out of allocation space, especially in nongenerational heaps, which treat the heap as a single continuous space without segregating it into generations.
- **Concurrent algorithms:** These algorithms aim to minimize pause times by performing most of their work concurrently with the application threads. CMS is an example of a collector using concurrent algorithms. However, because CMS does not perform compaction, fragmentation can become an issue over time. This can lead to longer pause times or even cause a fallback to a full GC using the Serial Old collector, which does include compaction.
- **Incremental compacting algorithms:** The G1 GC introduced incremental compaction to deal with the fragmentation issue found in CMS. G1 divides the heap into smaller regions and performs garbage collection on a subset of regions during a collection cycle. This approach helps maintain more predictable pause times while also handling compaction.
- **Thread-local handshakes:** Newer GCs like Shenandoah and ZGC leverage thread-local handshakes to minimize STW pauses. By employing this mechanism, they can perform certain GC operations on a per-thread basis, allowing application threads to continue running while the GC works. This approach helps to reduce the overall impact of garbage collection on application performance.
- **Ultra-low-pause-time collectors:** The Shenandoah and ZGC aim to have ultra-low pause times by performing concurrent marking, relocation, and compaction. Both minimize the STW pauses to a small fraction of the overall garbage collection work, offering consistent low latency for applications. While these GCs are not generational in the traditional sense, they do divide the heap into regions and collect different regions at different times. This approach builds upon the principles of incremental and "garbage first" collection. As of this writing, efforts are ongoing to further develop these newer collectors into generational ones, but they are included in this section due to their innovative strategies that enhance the principles of generational garbage collection.

Each collector has its advantages and trade-offs, allowing developers to choose the one that best suits their application requirements.

Young Collections and Weak Generational Hypothesis

In the realm of a generational heap, the majority of allocations take place in the *eden* space of the *young* generation. An allocating thread may encounter an allocation failure when this eden space is near its capacity, indicating that the GC must step in and reclaim space.

During the first *young* collection, the eden space undergoes a scavenging process in which live objects are identified and subsequently moved into the *to* survivor space. The survivor

space serves as a transitional area where surviving objects are copied, aged, and moved back and forth between the *from* and *to* spaces until they cross a tenuring threshold. Once an object crosses this threshold, it is promoted to the *old* generation. The underlying objective here is to promote only those objects that have proven their longevity, thereby creating a "Teenage Wasteland," as Charlie Hunt[5] would explain. 😉

The generational garbage collection is based on two main characteristics related to the weak-generational hypothesis:

1. **Most objects die young:** This means that we promote only long-lived objects. If the generational GC is efficient, we don't promote transients, nor do we promote medium-lived objects. This usually results in smaller long-lived data sets, keeping premature promotions, fragmentation, evacuation failures, and similar degenerative issues at bay.
2. **Maintenance of generations:** The generational algorithm has proven to be a great help to OpenJDK GCs, but it comes with a cost. Because the young-generation collector works separately and more often than the old-generation collector, it ends up moving live data. Therefore, generational GCs incur maintenance/bookkeeping overhead to ensure that they mark all reachable objects—a feat achieved through the use of "write barriers" that track cross-generational references.

Figure 1.1 depicts the three key concepts of generational GCs, providing a visual reinforcement of the information discussed here. The word cloud consists of the following phrases:

- **Objects die young:** Highlighting the idea that most objects are short-lived and only long-lived objects are promoted.
- **Small long-lived data sets:** Emphasizing the efficiency of the generational GC in not promoting transients or medium-lived objects, resulting in smaller long-lived data sets.
- **Maintenance barriers:** Highlighting the overhead and bookkeeping required by generational GCs to mark all reachable objects, achieved through the use of write barriers.

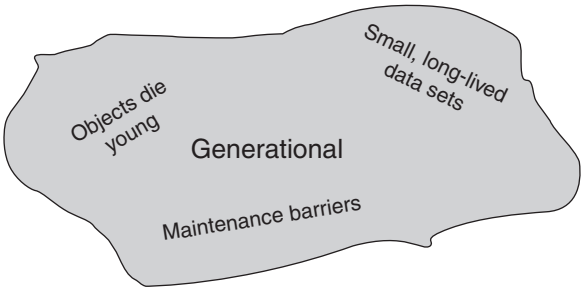

Figure 1.1 Key Concepts for Generational Garbage Collectors

[5] Charlie Hunt is my mentor, the author of *Java Performance* (https://ptgmedia.pearsoncmg.com/images/9780137142521/samplepages/0137142528.pdf), and my co-author for *Java Performance Companion* (www.pearson.com/en-us/subject-catalog/p/java-performance-companion/P200000009127/9780133796827).

Most HotSpot GCs employ the renowned "scavenge" algorithm for young collections. The Serial GC in HotSpot VM employs a single garbage collection thread dedicated to efficiently reclaiming memory within the young-generation space. In contrast, generational collectors such as the Parallel GC (throughput collector), G1 GC, and CMS GC leverage multiple GC worker threads.

Old-Generation Collection and Reclamation Triggers

Old-generation reclamation algorithms in HotSpot VM's generational GCs are optimized for throughput, responsiveness, or a combination of both. The Serial GC employs a single-threaded mark–sweep–compacting (MSC) GC. The Parallel GC uses a similar MSC GC with multiple threads. The CMS GC performs mostly concurrent marking, dividing the process into STW or concurrent phases. After marking, CMS reclaims old-generation space by performing in-place deallocation without compaction. If fragmentation occurs, CMS falls back to the serial MSC.

G1 GC, introduced in Java 7 update 4 and refined over time, is the first incremental collector. Specifically, it incrementally reclaims and compacts the old-generation space, as opposed to performing the single monolithic reclamation and compaction that is part of MSC. G1 GC divides the heap into smaller regions and performs garbage collection on a subset of regions during a collection cycle, which helps maintain more predictable pause times while also handling compaction.

After multiple young-generation collections, the old generation starts filling up, and garbage collection kicks in to reclaim space in the old generation. To do so, a full heap marking cycle must be triggered by either (1) a promotion failure, (2) a promotion of a regular-sized object that makes the old generation or the total heap cross the marking threshold, or (3) a large object allocation (also known as humongous allocation in the G1 GC) that causes the heap occupancy to cross a predetermined threshold.

Shenandoah GC and ZGC—introduced in JDK 12 and JDK 11, respectively—are ultra-low-pause-time collectors that aim to minimize STW pauses. In JDK 17, they are single-generational collectors. Apart from utilizing thread-local handshakes, these collectors know how to optimize for low-pause scenarios either by employing the application threads to help out or by asking the application threads to back off. This GC technique is known as graceful degradation.

Parallel GC Threads, Concurrent GC Threads, and Their Configuration

In the HotSpot VM, the total number of GC worker threads (also known as parallel GC threads) is calculated as a fraction of the total number of processing cores available to the Java process at start-up. Users can adjust the parallel GC thread count by assigning it directly on the command line using the `-XX:ParallelGCThreads=<n>` flag.

This configuration flag enables developers to define the number of parallel GC threads for GC algorithms that use parallel collection phases. It is particularly useful for tuning generational GCs, such as the Parallel GC and G1 GC. Recent additions like Shenandoah and ZGC, also use multiple GC worker threads and perform garbage collection concurrently with the application threads to minimize pause times. They benefit from load balancing, work sharing, and work stealing, which enhance performance and efficiency by parallelizing the garbage collection

process. This parallelization is particularly beneficial for applications running on multi-core processors, as it allows the GC to make better use of the available hardware resources.

In a similar vein, the `-XX:ConcGCThreads=<n>` configuration flag allows developers to specify the number of concurrent GC threads for specific GC algorithms that use concurrent collection phases. This flag is particularly useful for tuning GCs like G1, which performs concurrent work during marking, and Shenandoah and ZGC, which aim to minimize STW pauses by executing concurrent marking, relocation, and compaction.

By default, the number of parallel GC threads is automatically calculated based on the available CPU cores. Concurrent GC threads usually default to one-fourth of the parallel GC threads. However, developers may want to adjust the number of parallel or concurrent GC threads to better align with their application's performance requirements and available hardware resources.

Increasing the number of parallel GC threads can help improve overall GC throughput, as more threads work simultaneously on the parallel phases of this process. This increase may result in shorter GC pause times and potentially higher application throughput, but developers should be cautious not to over-commit processing resources.

By comparison, increasing the number of concurrent GC threads can enhance overall GC performance and expedite the GC cycle, as more threads work simultaneously on the concurrent phases of this process. However, this increase may come at the cost of higher CPU utilization and competition with application threads for CPU resources.

Conversely, reducing the number of parallel or concurrent GC threads may lower CPU utilization but could result in longer GC pause times, potentially affecting application performance and responsiveness. In some cases, if the concurrent collector is unable to keep up with the rate at which the application allocates objects (a situation referred to as the GC "losing the race"), it may lead to a graceful degradation—that is, the GC falls back to a less optimal but more reliable mode of operation, such as a STW collection mode, or might employ strategies like throttling the application's allocation rate to prevent it from overloading the collector.

Figure 1.2 shows the key concepts as a word cloud related to GC work:

- **Task queues:** Highlighting the mechanisms used by GCs to manage and distribute work among the GC threads.
- **Concurrent work:** Emphasizing the operations performed by the GC simultaneously with the application threads, aiming to minimize pause times.
- **Graceful degradation:** Referring to the GC's ability to switch to a less optimal but more reliable mode of operation when it can't keep up with the application's object allocation rate.
- **Pauses:** Highlighting the STW events during which application threads are halted to allow the GC to perform certain tasks.
- **Task stealing:** Emphasizing the strategy employed by some GCs in which idle GC threads "steal" tasks from the work queues of busier threads to ensure efficient load balancing.
- **Lots of threads:** Highlighting the use of multiple threads by GCs to parallelize the garbage collection process and improve throughput.

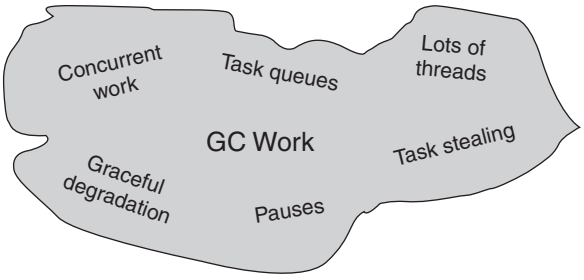

Figure 1.2 Key Concepts for Garbage Collection Work

It is crucial to find the right balance between the number of parallel and concurrent GC threads and application performance. Developers should conduct performance testing and monitoring to identify the optimal configuration for their specific use case. When tuning these threads, consider factors such as the number of available CPU cores, the nature of the application workload, and the desired balance between garbage collection throughput and application responsiveness.

The Evolution of the Java Programming Language and Its Ecosystem: A Closer Look

The Java language has evolved steadily since the early Java 1.0 days. To appreciate the advancements in the JVM, and particularly the HotSpot VM, it's crucial to understand the evolution of the Java programming language and its ecosystem. Gaining insight into how language features, libraries, frameworks, and tools have shaped and influenced the JVM's performance optimizations and garbage collection strategies will help us grasp the broader context.

Java 1.1 to Java 1.4.2 (J2SE 1.4.2)

Java 1.1, originally known as JDK 1.1, introduced JavaBeans, which allowed multiple objects to be encapsulated in a bean. This version also brought Java Database Connectivity (JDBC), Remote Method Invocation (RMI), and inner classes. These features set the stage for more complex applications, which in turn demanded improved JVM performance and garbage collection strategies.

From Java 1.2 to Java 5.0, the releases were renamed to include the version name, resulting in names like J2SE (Platform, Standard Edition). The renaming helped differentiate between Java 2 Micro Edition (J2ME) and Java 2 Enterprise Edition (J2EE).[6] J2SE 1.2 introduced two significant improvements to Java: the Collections Framework and the JIT compiler. The Collections Framework provided "a unified architecture for representing and manipulating collections",[7]

[6] www.oracle.com/java/technologies/javase/javanaming.html

[7] Collections Framework Overview. https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html.

which became essential for managing large-scale data structures and optimizing memory management in the JVM.

Java 1.3 (J2SE 1.3) added new APIs to the Collections Framework, introduced Math classes, and made the HotSpot VM the default Java VM. A directory services API was included for Java RMI to look up any directory or name service. These enhancements further influenced JVM efficiency by enabling more memory-efficient data management and interaction patterns.

The introduction of the New Input/Output (NIO) API in Java 1.4 (J2SE 1.4), based on Java Specification Request (JSR) #51,[8] significantly improved I/O operation efficiency. This enhancement resulted in reduced waiting times for I/O tasks and an overall boost in JVM performance. J2SE 1.4 also introduced the Logging API, which allowed for generating text or XML-formatted log messages that could be directed to a file or console.

J2SE 1.4.1 was soon superseded by J2SE 1.4.2, which included numerous performance enhancements in HotSpot's client and server compilers. Security enhancements were added as well, and Java users were introduced to Java updates via the Java Plug-in Control Panel Update tab. With the continuous improvements in the Java language and its ecosystem, JVM performance strategies evolved to accommodate increasingly more complex and resource-demanding applications.

Java 5 (J2SE 5.0)

The Java language made its first significant leap toward language refinement with the release of Java 5.0. This version introduced several key features, including generics, autoboxing/unboxing, annotations, and an enhanced `for` loop.

Language Features

Generics introduced two major changes: (1) a change in syntax and (2) modifications to the core API. Generics allow you to reuse your code for different data types, meaning you can write just a single class—there is no need to rewrite it for different inputs.

To compile the generics-enriched Java 5.0 code, you would need to use the Java compiler `javac`, which was packaged with the Java 5.0 JDK. (Any version prior to Java 5.0 did not have the core API changes.) The new Java compiler would produce errors if any type safety violations were detected at compile time. Hence, generics introduced type safety into Java. Also, generics eliminated the need for explicit casting, as casting became implicit.

Here's an example of how to create a generic class named `FreshmenAdmissions` in Java 5.0:

```
class FreshmenAdmissions<K, V> {
     //…
}
```

[8] https://jcp.org/en/jsr/detail?id=51

In this example, K and V are placeholders for the actual types of objects. The class `FreshmenAdmissions` is a generic type. If we declare an instance of this generic type without specifying the actual types for K and V, then it is considered to be a raw type of the generic type `FreshmenAdmissions<K, V>`. Raw types exist for generic types and are used when the specific type parameters are unknown.

```
FreshmenAdmissions applicationStatus;
```

However, suppose we declare the instance with actual types:

```
FreshmenAdmissions<String, Boolean>
```

Then `applicationStatus` is said to be a parameterized type—specifically, it is parameterized over types `String` and `Boolean`.

```
FreshmenAdmissions<String, Boolean> applicationStatus;
```

NOTE Many C++ developers may see the angle brackets < > and immediately draw parallels to C++ templates. Although both C++ and Java use generic types, C++ templates are more like a compile-time mechanism, where the generic type is replaced by the actual type by the C++ compiler, offering robust type safety.

While discussing generics, we should also touch on autoboxing and unboxing. In the `FreshmenAdmissions<K, V>` class, we can have a generic method that returns the type V:

```
public V getApprovalInfo() {
     return boolOrNumValue;
}
```

Based on our declaration of V in the parameterized type, we can perform a `Boolean` check and the code would compile correctly. For example:

```
applicationStatus = new FreshmenAdmissions<>();
if (applicationStatus.getApprovalInfo()) {
    //...
}
```

In this example, we see a generic type invocation of V as a `Boolean`. Autoboxing ensures that this code will compile correctly. In contrast, if we had done the generic type invocation of V as an `Integer`, we would get an "incompatible types" error. Thus, autoboxing is a Java compiler conversion that understands the relationship between a primitive and its object class. Just as autoboxing encapsulates a `boolean` value into its `Boolean` wrapper class, unboxing helps return a `boolean` value when the return type is a `Boolean`.

Here's the entire example (using Java 5.0):

```
class FreshmenAdmissions<K, V> {
    private K key;
    private V boolOrNumValue;

    public void admissionInformation(K name, V value) {
        key = name;
        boolOrNumValue = value;
    }

    public V getApprovalInfo() {
        return boolOrNumValue;
    }

    public K getApprovedName() {
        return key;
    }
}

public class GenericExample {

    public static void main(String[] args) {
        FreshmenAdmissions<String, Boolean> applicationStatus;
        applicationStatus = new FreshmenAdmissions<String, Boolean>();

        FreshmenAdmissions<String, Integer> applicantRollNumber;
        applicantRollNumber = new FreshmenAdmissions<String, Boolean>();

        applicationStatus.admissionInformation("Annika", true);

        if (applicationStatus.getApprovalInfo()) {
                applicantRollNumber.admissionInformation(applicationStatus.getApprovedName(), 4);
        }

        System.out.println("Applicant " + applicantRollNumber.getApprovedName() +
            " has been admitted with roll number of " + applicantRollNumber.getApprovalInfo());
    }

}
```

Figure 1.3 shows a class diagram to help visualize the relationship between the classes. In the diagram, the `FreshmenAdmissions` class has three methods: `admissionInformation(K,V)`, `getApprovalInfo():V`, and `getApprovedName():K`. The `GenericExample` class uses the `FreshmenAdmissions` class and has a `main(String[] args)` method.

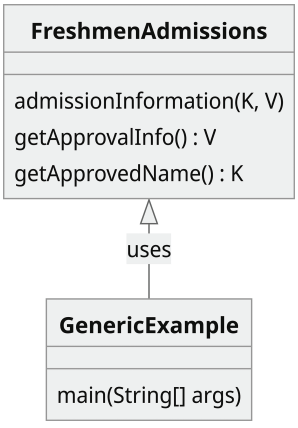

Figure 1.3 A Generic Type Example Showcasing the FreshmenAdmission Class

JVM and Packages Enhancements

Java 5.0 also brought significant additions to the `java.lang.*` and `java.util.*` packages and added most of the concurrent utilities as specified in JSR 166.[9]

Garbage collection *ergonomics* was a term coined in J2SE 5.0; it referred to a default collector for server-class machines.[10] The default collector was chosen to be the parallel GC, and default values for the initial and maximum heap sizes for parallel GC were set automatically.

> **NOTE** The Parallel GC of J2SE 5.0 days didn't have parallel compaction of the old generation. Thus, only the young generation could parallel scavenge; the old generation would still employ the serial GC algorithm, MSC.

Java 5.0 also introduced the enhanced `for` loop, often called the `for-each` loop, which greatly simplified iterating over arrays and collections. This new loop syntax automatically handled the iteration without the need for explicit index manipulation or calls to iterator methods. The enhanced `for` loop was both more concise and less error-prone than the traditional `for` loop, making it a valuable addition to the language. Here's an example to demonstrate its usage:

```
List<String> names = Arrays.asList("Monica", "Ben", "Annika", "Bodin");
for (String name : names) {
    System.out.println(name);
}
```

[9] Java Community Process. *JSRs: Java Specification Requests*, detail JSR# 166. https://jcp.org/en/jsr/detail?id=166.

[10] https://docs.oracle.com/javase/8/docs/technotes/guides/vm/server-class.html

In this example, the enhanced `for` loop iterates over the elements of the `names` list and assigns each element to the `name` variable in turn. This is much cleaner and more readable than using an index-based `for` loop or an iterator.

Java 5.0 brought several enhancements to the `java.lang` package, including the introduction of annotations, which allows for adding metadata to Java source code. Annotations provide a way to attach information to code elements such as classes, methods, and fields. They can be used by the Java compiler, runtime environment, or various development tools to generate additional code, enforce coding rules, or aid in debugging. The `java.lang.annotation` package contains the necessary classes and interfaces to define and process annotations. Some commonly used built-in annotations are `@Override`, `@Deprecated`, and `@SuppressWarnings`.

Another significant addition to the `java.lang` package was the `java.lang.instrument` package, which made it possible for Java agents to modify running programs on the JVM. The new services enabled developers to monitor and manage the execution of Java applications, providing insights into the behavior and performance of the code.

Java 6 (Java SE 6)

Java SE 6, also known as Mustang, primarily focused on additions and enhancements to the Java API,[11] but made only minor adjustments to the language itself. In particular, it introduced a few more interfaces to the Collections Framework and improved the JVM. For more information on Java 6, refer to the driver JSR 270.[12]

Parallel compaction, introduced in Java 6, allowed for the use of multiple GC worker threads to perform compaction of the old generation in the generational Java heap. This feature significantly improved the performance of Java applications by reducing the time taken for garbage collection. The performance implications of this feature are profound, as it allows for more efficient memory management, leading to improved scalability and responsiveness of Java applications.

JVM Enhancements

Java SE 6 made significant strides in improving scripting language support. The `javax.script` package was introduced, enabling developers to embed and execute scripting languages within Java applications. Here's an example of executing JavaScript code using the `ScriptEngine` API:

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class ScriptingExample {
    public static void main(String[] args) {
```

[11] Oracle. "Java SE 6 Features and Enhancements." www.oracle.com/java/technologies/javase/features.html.

[12] Java Community Process. *JSRs: Java Specification Requests*, detail JSR# 270. www.jcp.org/en/jsr/detail?id=270.

```
        FreshmenAdmissions<String, Integer> applicantGPA;
        applicantGPA = new FreshmenAdmissions<String,Integer>();
        applicantGPA.admissionInformation("Annika", 98);

        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("JavaScript");

        try {
            engine.put("score", applicantGPA.getApprovalInfo());
            engine.eval("var gpa = score / 25; print('Applicant GPA: ' + gpa);");
        } catch (ScriptException e) {
            e.printStackTrace();
        }
    }
}
```

Java SE 6 also introduced the Java Compiler API (JSR 199[13]), which allows developers to invoke the Java compiler programmatically. Using the `FreshmenAdmissions` example, we can compile our application's Java source file as follows:

```
import javax.tools.JavaCompiler;
import javax.tools.ToolProvider;
import java.io.File;

public class CompilerExample {
    public static void main(String[] args) {
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

        int result = compiler.run(null, null, null, "path/to/FreshmenAdmissions.java");
        if (result == 0) {
            System.out.println("Congratulations! You have successfully compiled your program");
        } else {
            System.out.println("Apologies, your program failed to compile");
        }
    }
}
```

Figure 1.4 is a visual representation of the two classes and their relationships.

[13] https://jcp.org/en/jsr/detail?id=199

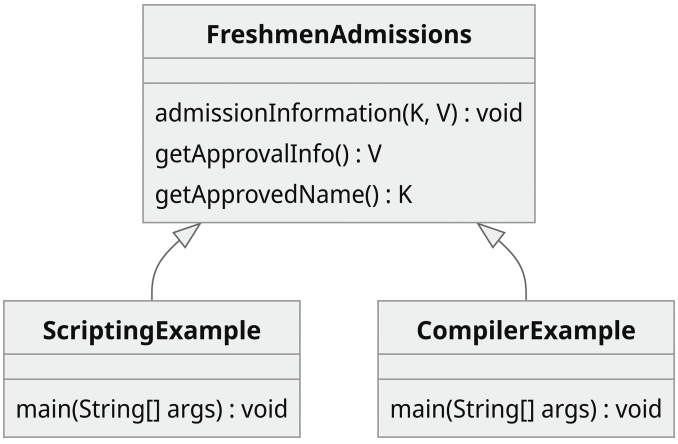

Figure 1.4 Script Engine API and Java Compiler API Examples

Additionally, Java SE 6 enhanced web services support by adding JAXB 2.0 and JAX-WS 2.0 APIs, simplifying the creation and consumption of SOAP-based web services.

Java 7 (Java SE 7)

Java SE 7 marked another significant milestone in Java's evolution, introducing numerous enhancements to both the Java language and the JVM. These enhancements are essential for understanding modern Java development.

Language Features

Java SE 7 introduced the diamond operator (<>), which simplified generic type inference. This operator enhances code readability and reduces redundancy. For instance, when declaring a `FreshmenAdmissions` instance, you no longer need to specify the type parameters twice:

```
// Before Java SE 7
FreshmenAdmissions<String, Boolean> applicationStatus = new FreshmenAdmissions<String, Boolean>();

// With Java SE 7
FreshmenAdmissions<String, Boolean> applicationStatus = new FreshmenAdmissions<>();
```

Another significant addition is the try-with-resources statement, which automates resource management, reducing the likelihood of resource leaks. Resources that implement `java.lang.AutoCloseable` can be used with this statement, and they are automatically closed when no longer needed.

Project Coin,[14] a part of Java SE 7, introduced several small but impactful enhancements, including improving literals, support for strings in switch statements, and enhanced error handling with try-with-resources and multi-catch blocks. Performance improvements such as compressed ordinary object pointers (oops), escape analysis, and tiered compilation also emerged during this time.

JVM Enhancements

Java SE 7 further enhanced the Java NIO library with NIO.2, providing better support for file I/O. Additionally, the fork/join framework, originating from JSR 166: *Concurrency Utilities*,[15] was incorporated into Java SE 7's concurrent utilities. This powerful framework enables efficient parallel processing of tasks by recursively breaking them down into smaller, more manageable subtasks and then merging the outcomes for a result.

The fork/join framework in Java reminds me of a programming competition I participated in during my student years. The event was held by the Institute of Electrical and Electronics Engineers (IEEE) and was a one-day challenge for students in computer science and engineering. My classmate and I decided to tackle the competition's two puzzles individually. I took on a puzzle that required optimizing a binary split-and-sort algorithm. After some focused work, I managed to significantly refine the algorithm, resulting in one of the most efficient solutions presented at the competition.

This experience was a practical demonstration of the power of binary splitting and sorting—a concept that is akin to the core principle of Java's fork/join framework. In the following fork/join example, you can see how the framework is a powerful tool to efficiently process tasks by dividing them into smaller subtasks and subsequently unifying them.

Here's an example of using the fork/join framework to calculate the average GPA of all admitted students:

```
// Create a list of admitted students with their GPAs
List<Integer> admittedStudentGPAs = Arrays.asList(95, 88, 76, 93, 84, 91);

// Calculate the average GPA using the fork/join framework
ForkJoinPool forkJoinPool = new ForkJoinPool();
AverageGPATask averageGPATask = new AverageGPATask(admittedStudentGPAs, 0,
                                                   admittedStudentGPAs.size());
double averageGPA = forkJoinPool.invoke(averageGPATask);

System.out.println("The average GPA of admitted students is: " + averageGPA);
```

In this example, we define a custom class `AverageGPATask` extending `RecursiveTask<Double>`. It's engineered to compute the average GPA from a subset of students by recursively splitting the list of GPAs into smaller tasks—a method reminiscent of the divide-and-conquer approach I refined during my competition days. Once the tasks are

[14] https://openjdk.org/projects/coin/

[15] https://jcp.org/en/jsr/detail?id=166

sufficiently small, they can be computed directly. The `compute()` method of `AverageGPATask` is responsible for both the division and the subsequent combination of these results to determine the overall average GPA.

Here's a simplified version of what the `AverageGPATask` class might look like:

```
class AverageGPATask extends RecursiveTask<Double> {
    private final List<Integer> studentGPAs;
    private final int start;
    private final int end;

    AverageGPATask(List<Integer> studentGPAs, int start, int end) {
        this.studentGPAs = studentGPAs;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Double compute() {
        // Split the task if it's too large, or compute directly if it's small enough
        // Combine the results of subtasks
    }
}
```

In this code, we create an instance of `AverageGPATask` to calculate the average GPA of all admitted students. We then use a `ForkJoinPool` to execute this task. The `invoke()` method of `ForkJoinPool` initiates the task and actively waits for its completion, akin to calling a `sort()` function and waiting for a sorted array, thereby returning the computed result. This is a great example of how the fork/join framework can be used to enhance the efficiency of a Java application.

Figure 1.5 shows a visual representation of the classes and their relationships. In this diagram, there are two classes: `FreshmenAdmissions` and `AverageGPATask`. The `FreshmenAdmissions` class has methods for setting admission information, getting approval information, and getting the approved name. The `AverageGPATask` class, which calculates the average GPA, has a `compute` method that returns a `Double`. The `ForkJoinPool` class uses the `AverageGPATask` class to calculate the average GPA; the `FreshmenAdmissions` class uses the `AverageGPATask` to calculate the average GPA of admitted students.

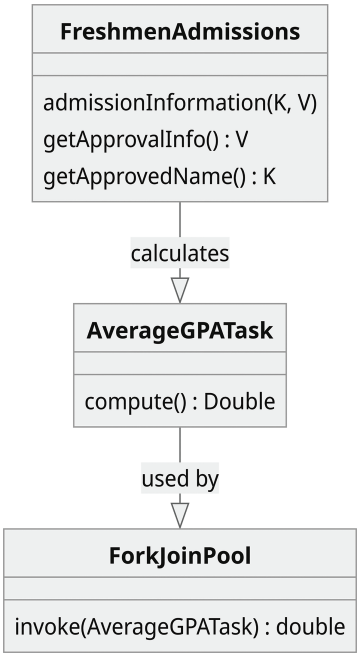

Figure 1.5 An Elaborate ForkJoinPool Example

Java SE 7 also brought significant enhancements to garbage collection. The parallel GC became NUMA-aware, optimizing performance on systems with the Non-Uniform Memory Access (NUMA) architecture. At its core, NUMA enhances memory access efficiency by taking into account the physical proximity of memory to a processing unit, such as a CPU or core. Prior to the introduction of NUMA, computers exhibited uniform memory access patterns, with all processing units sharing a single memory resource equally. NUMA introduced a paradigm in which processors access their local memory—memory that is physically closer to them—more swiftly than distant, non-local memory, which might be adjacent to other processors or shared among various cores. Understanding the intricacies of NUMA-aware garbage collection is vital for optimizing Java applications running on NUMA systems. (For an in-depth exploration of the NUMA-awareness in GC, see Chapter 6, "Advanced Memory Management and Garbage Collection in OpenJDK.")

In a typical NUMA setup, the memory access times can vary because data must traverse certain paths—or *hops*—within the system. These *hops* refer to the transfer of data between different nodes. Local memory access is direct and thus faster (fewer *hops*), while accessing non-local memory—residing farther from the processor—incurs additional *hops* resulting in increased latency due to the longer traversal paths. This architecture is particularly advantageous in

systems with multiple processors, as it maximizes the efficiency of memory utilization by leveraging the proximity factor.

NOTE My involvement with the implementation of NUMA-aware garbage collection first came during my tenure at AMD. I was involved in the performance characterization of AMD's Opteron processor,[16] a groundbreaking product that bought NUMA architecture into the mainstream. The Opteron[17] was designed with an integrated memory controller and a HyperTransport interconnect,[18] which allowed for direct, high-speed connections both between processors and between the processor and the memory. This design made the Opteron ideal for NUMA systems, and it was used in many high-performance servers, including Sun Microsystems' V40z.[19] The V40z was a high-performance server that was built by Sun Microsystems using AMD's Opteron processors. It was a NUMA system, so it could significantly benefit from NUMA-aware garbage collection. I was instrumental in providing the JDK team with data on cross-traffic and multi-JVM-local traffic, as well as insights into interleaved memory to illustrate how to amortize the latency of the *hops* between memory nodes.

To further substantiate the benefits of this feature, I implemented a prototype for the HotSpot VM with `-XX:+UseLargePages`. This prototype utilized large pages (also known as HugeTLB pages[20]) on Linux and the `numactl` API,[21] a tool for controlling NUMA policy on Linux systems. It served as a tangible demonstration of the potential performance improvements that could be achieved with NUMA-aware GC.

When the Parallel GC is NUMA-aware, it optimizes the way it handles memory to improve performance on NUMA systems. The young generation's eden space is allocated in NUMA-aware regions. In other words, when an object is created, it's placed in an eden space local to the processor that's executing the thread that created the object. This can significantly improve performance because accessing local memory is faster than accessing non-local memory.

The survivor spaces and the old generation are page interleaved in memory. Page interleaving is a technique used to distribute memory pages across different nodes in a NUMA system. It can help balance memory utilization across the nodes, leading to more efficient utilization of the memory bandwidth available on each node.

Note, however, that these optimizations are most beneficial on systems with a large number of processors and a significant disparity between local and non-local memory access times. On systems with a small number of processors or near-similar local and non-local memory access times, the impact of NUMA-aware garbage collection might be less noticeable. Also,

[16] Within AMD's Server Perf and Java Labs, my focus was on refining the HotSpot VM to enhance performance for Java workloads on NUMA-aware architectures, concentrating on JIT compiler optimizations, code generation, and GC efficiency for Opteron processor family.

[17] The Opteron was the first processor that supported AMD64, also known as x86-64ISA: https://en.wikipedia.org/wiki/Opteron.

[18] https://en.wikipedia.org/wiki/HyperTransport

[19] Oracle. "Introduction to the Sun Fire V20z and Sun Fire V40z Servers." https://docs.oracle.com/cd/E19121-01/sf.v40z/817-5248-21/chapter1.html.

[20] "HugeTLB Pages." www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html.

[21] https://halobates.de/numaapi3.pdf

NUMA-aware garbage collection is not enabled by default; instead, it is enabled using the `-XX:+UseNUMA` JVM option.

In addition to the NUMA-aware GC, Java SE 7 Update 4 introduced the Garbage First Garbage Collector (G1 GC). G1 GC was designed to offer superior performance and predictability compared to its predecessors. Chapter 6 of this book delves into the enhancements brought by G1 GC, providing detailed examples and practical tips for its usage. For a comprehensive understanding of G1 GC, I recommend the *Java Performance Companion* book.[22]

Java 8 (Java SE 8)

Java 8 brought several significant features to the table that enhanced the Java programming language and its capabilities. These features can be leveraged to improve the efficiency and functionality of Java applications.

Language Features

One of the most notable features introduced in Java 8 was lambda expressions, which allow for more concise and functional-style programming. This feature can improve the simplicity of code by enabling more efficient implementation of functional programming concepts. For instance, if we have a list of students and we want to filter out those with a GPA greater than 80, we could use a lambda expression to do so concisely.

```
List<Student> admittedStudentsGPA = Arrays.asList(student1, student2, student3, student4,
student5, student6);
List<Student> filteredStudentsList = admittedStudentsGPA.stream().filter(s -> s.getGPA() > 80).
collect(Collectors.toList());
```

The lambda expression `s -> s.getGPA() > 80` is essentially a short function that takes a `Student` object `s` and returns a `boolean` value based on the condition `s.getGPA() > 80`. The `filter` method uses this lambda expression to filter the stream of students and `collect` gathers the results back into a list.

Java 8 also extended annotations to cover any place where a type is used, a capability known as *type annotations*. This helped enhance the language's expressiveness and flexibility. For example, we could use a type annotation to specify that the `studentGPAs` list should contain only non-null `Integer` objects:

```
private final List<@NonNull Integer> studentGPAs;
```

Additionally, Java 8 introduced the Stream API, which allows for efficient data manipulation and processing on collections. This addition contributed to JVM performance improvements, as developers could now optimize data processing more effectively. Here's an example using the Stream API to calculate the average GPA of the students in the `admittedStudentsGPA` list:

```
double averageGPA = admittedStudentsGPA.stream().mapToInt(Student::getGPA).average().orElse(0);
```

[22] www.pearson.com/en-us/subject-catalog/p/java-performance-companion/P200000009127/9780133796827

In this example, the `stream` method is called to create a stream from the list, `mapToInt` is used with a method reference `Student::getGPA` to convert each `Student` object to its GPA (an integer), `average` calculates the average of these integers, and `orElse` provides a default value of 0 if the stream is empty and an average can't be calculated.

Another enhancement in Java 8 was the introduction of default methods in interfaces, which enable developers to add new methods without breaking existing implementations, thus increasing the flexibility of interface design. If we were to define an interface for our tasks, we could use default methods to provide a standard implementation of certain methods. This could be useful if most tasks need to perform a set of common operations. For instance, we could define a `ComputableTask` interface with a `logStart` default method that logs when a task starts:

```
public interface Task {
    default void logStart() {
        System.out.println("Task started");
    }
    Double compute();
}
```

This `ComputableTask` interface could be implemented by any class that represents a task, providing a standard way to log the start of a task and ensuring that each task can be computed. For example, our `AverageGPATask` class could implement the `ComputableTask` interface:

```
class AverageGPATask extends RecursiveTask<Double> implements ComputableTask {
    // ...
}
```

JVM Enhancements

On the JVM front, Java 8 removed the Permanent Generation (PermGen) memory space, which was previously used to store metadata about loaded classes and other objects not associated with any instance. PermGen often caused problems due to memory leaks and the need for manual tuning. In Java 8, it was replaced by the Metaspace, which is located in native memory, rather than on the Java heap. This change eliminated many PermGen-related issues and improved the overall performance of the JVM. The Metaspace is discussed in more detail in Chapter 8, "Accelerating Time to Steady State with OpenJDK HotSpot VM."

Another valuable optimization introduced in JDK 8 update 20 was String Deduplication, a feature was specifically designed for the G1 GC. In Java, `String` objects are immutable, which means that once created, their content cannot be changed. This characteristic often leads to multiple `String` objects containing the same character sequences. While these duplicates don't impact the correctness of the application, they do contribute to increased memory footprint and can indirectly affect performance due to higher garbage collection overhead.

String Deduplication takes care of the issue of duplicate `String` objects by scanning the Java heap during GC cycles. It identifies duplicate strings and replaces them with references to a

single canonical instance. By decreasing the overall heap size and the number of live objects, the time taken for GC pauses can be reduced, contributing to lower tail latencies (this feature is discussed in more detail in Chapter 6). To enable the String Deduplication feature with the G1 GC, you can add the following JVM options:

```
-XX:+UseG1GC -XX:+UseStringDeduplication
```

Java 9 (Java SE 9) to Java 16 (Java SE 16)

Java 9: Project Jigsaw, JShell, AArch64 Port, and Improved Contended Locking

Java 9 brought several significant enhancements to the Java platform. The most notable addition was Project Jigsaw,[23] which implemented a module system, enhancing the platform's scalability and maintainability. We will take a deep dive into modularity in Chapter 3, "From Monolithic to Modular Java: A Retrospective and Ongoing Evolution."

Java 9 also introduced JShell, an interactive Java REPL (read–evaluate–print loop), enabling developers to quickly test and experiment with Java code. JShell allows for the evaluation of code snippets, including statements, expressions, and definitions. It also supports commands, which can be entered by adding a forward slash ("/") to the beginning of the command. For example, to change the interaction mode to verbose in JShell, you would enter the command `/set feedback verbose`.

Here's an example of how you might use JShell:

```
$ jshell
|  Welcome to JShell -- Version 17.0.7
|  For an introduction type: /help intro

jshell> /set feedback verbose
|  Feedback mode: verbose

jshell> boolean trueMorn = false
trueMorn ==> false
|  Created variable trueMorn : boolean

jshell> boolean Day(char c) {
   ...> return (c == 'Y');
   ...> }
|  Created method Day(char)

jshell> System.out.println("Did you wake up before 9 AM? (Y/N)")
Did you wake up before 9 AM? (Y/N)
```

[23] https://openjdk.org/projects/jigsaw/

```
jshell> trueMorn = Day((char) System.in.read())
Y
trueMorn ==> true
|  Assigned to trueMorn : boolean

jshell> System.out.println("It is " + trueMorn + " that you are a morning person")
It is true that you are a morning person
```

In this example, we first initialize `trueMorn` as `false` and then define our `Day()` method, which returns `true` or `false` based on the value of `c`. We then use `System.out.println` to ask our question and read the char input. We provide `Y` as input, so `trueMorn` is evaluated to be `true`.

JShell also provides commands like `/list`, `/vars`, `/methods`, and `/imports`, which provide useful information about the current JShell session. For instance, `/list` shows all the code snippets you've entered:

```
jshell> /list

   1 : boolean trueMorn = false;
   2 : boolean Day(char c) {
       return (c=='Y');
       }
   3 : System.out.println("Did you wake up before 9 AM? (Y/N)")
   4 : trueMorn = Day((char) System.in.read())
   5 : System.out.println("It is " + trueMorn + " that you are a morning person")
```

`/vars` lists all the variables you've declared:

```
jshell> /vars
|    boolean trueMorn = true
```

`/methods` lists all the methods you've defined:

```
jshell> /methods
|    boolean Day(char)
```

`/imports` shows all the imports in your current session:

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
```

```
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
```

The imports will also show up if you run the `/list –all` or `/list -start` command.

Furthermore, JShell can work with modules using the `--module-path` and `--add-modules` options, allowing you to explore and experiment with modular Java code.

Java 9 also added the AArch64 (also known as Arm64) port, thereby providing official support for the Arm 64-bit architecture in OpenJDK. This port expanded Java's reach in the ecosystem, allowing it to run on a wider variety of devices, including those with Arm processors.

Additionally, Java 9 improved contended locking through JEP 143: *Improve Contended Locking*.[24] This enhancement optimized the performance of intrinsic object locks and reduced the overhead associated with contended locking, improving the performance of Java applications that heavily rely on synchronization and contended locking. In Chapter 7, "Runtime Performance Optimizations: A Focus on Strings, Locks, and Beyond" we will take a closer look at locks and strings performance.

Release Cadence Change and Continuous Improvements

Starting from Java 9, the OpenJDK community decided to change the release cadence to a more predictable six-month cycle. This change was intended to provide more frequent updates and improvements to the language and its ecosystem. Consequently, developers could benefit from new features and enhancements more quickly.

Java 10: Local Variable Type Inference and G1 Parallel Full GC

Java 10 introduced local-variable type inference with the `var` keyword. This feature simplifies code by allowing the Java compiler to infer the variable's type from its initializer. This results in less verbose code, making it more readable and reducing boilerplate. Here's an example:

```
// Before Java 10
List<Student> admittedStudentsGPA = new ArrayList<>();

// With 'var' in Java 10
var admittedStudentsGPA = new ArrayList<Student>();
```

In this example, the `var` keyword is used to declare the variable `admittedStudentsGPA`. The compiler infers the type of `admittedStudentsGPA` from its initializer `new ArrayList <Student>()`, so you don't need to explicitly declare it.

[24] https://openjdk.org/jeps/143

```
// Using 'var' for local-variable type inference with admittedStudentsGPA list
var filteredStudentsList = admittedStudentsGPA.stream().filter(s -> s.getGPA() > 80).
collect(Collectors.toList());
```

In the preceding example, the `var` keyword is used in a more complex scenario involving a stream operation. The compiler infers the type of `filteredStudentsList` from the result of the stream operation.

In addition to addressing local-variable type inference, Java 10 improved the G1 GC by enabling parallel full garbage collections. This enhancement increased the efficiency of garbage collection, especially for applications with large heaps and those that suffer from some pathological (corner) case that causes evacuation failures, which in turn evoke the fallback full GC.

Java 11: New HTTP Client and String Methods, and Epsilon and Z GC

Java 11, a long-term support (LTS) release, introduced a variety of improvements to the Java platform, enhancing its performance and utility. One significant JVM addition in this release was the Epsilon GC,[25] an experimental no-op GC designed to test the performance of applications with minimal GC interference. The Epsilon GC allows developers to understand the impact of garbage collection on their application's performance, helping them make informed decisions about GC tuning and optimization.

```
$ java -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC -jar myApplication.jar
```

Epsilon GC is a unique addition to the JVM. It manages memory allocation, but doesn't implement any actual memory reclamation process. With this GC, once the available Java heap is used up, the JVM will shut down. Although this behavior might seem unusual, it's actually quite useful in certain scenarios—for instance, with extremely short-lived microservices. Moreover, for ultra-low-latency applications that can afford the heap footprint, Epsilon GC can help avoid all marking, moving, and compaction, thereby eliminating performance overhead. This streamlined behavior makes it an excellent tool for performance testing an application's memory pressures and understanding the overhead associated with VM/memory barriers.

Another noteworthy JVM enhancement in Java 11 was the introduction of the Z Garbage Collector (ZGC), which was initially available as an experimental feature. ZGC is a low-latency, scalable GC designed to handle large heaps with minimal pause times. By providing better garbage collection performance for applications with large memory requirements, ZGC enables developers to create high-performance applications that can effectively manage memory resources. (We'll take a deep dive into ZGC in Chapter 6 of this book.)

Keep in mind that these GCs are experimental in JDK 11, so using the `+UnlockExperimentalVMOptions` flag is necessary to enable them. Here's an example for ZGC:

```
$ java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC -jar myApplication.jar
```

[25] https://blogs.oracle.com/javamagazine/post/epsilon-the-jdks-do-nothing-garbage-collector

In addition to JVM enhancements, Java 11 introduced a new HTTP client API that supports HTTP/2 and WebSocket. This API improved performance and provided a modern alternative to the `HttpURLConnection` API. As an example of its use, assume that we want to fetch some extracurricular activities' data related to our students from a remote API:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("http://students-extracurricular.com/students"))
    .build();

client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

In this example, we're making a GET request to `http://students-extracurricular.com/students` to fetch the extracurricular data. The response is handled asynchronously, and when the data is received, it's printed to the console. This new API provides a more modern and efficient way to make HTTP requests compared to the older `HttpURLConnection` API.

Furthermore, Java 11 added several new `String` methods, such as `strip()`, `repeat()`, and `isBlank()`. Let's look at our student GPA example and see how we can enhance it using these methods:

```
// Here we have student names with leading and trailing whitespaces
List<String> studentNames = Arrays.asList("  Monica  ", "  Ben  ", "  Annika  ", "  Bodin  ");

// We can use the strip() method to clean up the names
List<String> cleanedNames = studentNames.stream()
    .map(String::strip)
    .collect(Collectors.toList());

// Now suppose we want to create a String that repeats each name three times
// We can use the repeat() method for this
List<String> repeatedNames = cleanedNames.stream()
    .map(name -> name.repeat(3))
    .collect(Collectors.toList());

// And suppose we want to check if any name is blank after stripping out whitespaces
// We can use the isBlank() method for this
boolean hasBlankName = cleanedNames.stream()
    .anyMatch(String::isBlank);
```

These enhancements, along with numerous other improvements, made Java 11 a robust and efficient platform for developing high-performance applications.

Java 12: Switch Expressions and Shenandoah GC

Java 12 introduced the Shenandoah GC as an experimental feature. Like ZGC, Shenandoah GC is designed for large heaps with low latency requirements. This addition contributed to improved JVM performance for latency-sensitive applications. To enable Shenandoah GC, you can use the following JVM option:

```
$ java -XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -jar myApplication.jar
```

Java 12 also introduced switch expressions as a preview feature, simplifying `switch` statements and making them more readable and concise:

```
String admissionResult = switch (status) {
    case "accepted" -> "Congratulations!";
    case "rejected" -> "Better luck next time.";
    default -> "Awaiting decision.";
};
```

In Java 14, you also have the option to use the `yield` keyword to return a value from a switch block. This is particularly useful when you have multiple statements in a case block. Here's an example:

```
String admissionResult = switch (status) {
    case "accepted" -> {
        // You can have multiple statements here
        yield "Congratulations!";
    }
    case "rejected" -> {
        // You can have multiple statements here
        yield "Better luck next time.";
    }
    default -> {
        // You can have multiple statements here
        yield "Awaiting decision.";
    }
};
```

Java 13: Text Blocks and ZGC Enhancements

Java 13 offered a number of new features and enhancements, including the introduction of text blocks as a preview feature. Text blocks simplify the creation of multiline string literals, making it easier to write and read formatted text:

```
String studentInfo = """
                     Name: Monica Beckwith
                     GPA: 3.83
                     Status: Accepted
                     """;
```

JDK 13 also included several enhancements to ZGC. For example, the new ZGC uncommit feature allowed unused heap memory to be returned to the operating system (OS) in a more timely and efficient manner. Prior to this feature, ZGC would release memory back to the OS only during a full GC cycle,[26] which could be infrequent and might result in large amounts of memory being held by the JVM unnecessarily. With the new feature, ZGC could return memory back to the operating system immediately upon detecting that a page is no longer needed by the JVM.

Java 14: Pattern Matching for instanceof and NUMA-Aware Memory Allocator for G1

Java 14 introduced pattern matching for the `instanceof` operator as a preview feature, simplifying type checking and casting:

```
if (object instanceof Student student) {
    System.out.println("Student name: " + student.getName());
}
```

Java 14 also introduced NUMA-aware memory allocation for the G1 GC. This feature is especially useful in multi-socket systems, where memory access latency and bandwidth efficiency can vary significantly across the sockets. With NUMA-aware allocation, G1 can allocate memory with a preference for the local NUMA node, which can significantly reduce memory access latency and improve performance. This feature can be enabled on supported platforms using the command-line option `-XX:+UseNUMA`.

Java 15: Sealed and Hidden Classes

Java 15 introduced two significant features that give developers more control over class hierarchies and the encapsulation of implementation details.

Sealed classes, introduced as a preview feature,[27] enable developers to define a limited set of subclasses for a superclass, providing more control over class hierarchies. For example, in a university application system, you could have:

```
public sealed class Student permits UndergraduateStudent, GraduateStudent, ExchangeStudent {
    private final String name;
    private final double gpa;
    // Common student properties and methods
}

final class UndergraduateStudent extends Student {
    // Undergraduate-specific properties and methods
}
```

[26] https://openjdk.org/jeps/351

[27] https://openjdk.org/jeps/360

```
final class GraduateStudent extends Student {
    // Graduate-specific properties and methods
}

final class ExchangeStudent extends Student {
    // Exchange-specific properties and methods
}
```

Here, `Student` is a sealed class, and only the specified classes (`UndergraduateStudent`, `GraduateStudent`, `ExchangeStudent`) can extend it, ensuring a controlled class hierarchy.

Hidden classes,[28] in contrast, are classes that are not discoverable by their name at runtime and can be used by frameworks that generate classes dynamically. For instance, a hidden class might be generated for a specific task like a query handler in a database framework:

```
// Within a database framework method
MethodHandles.Lookup lookup = MethodHandles.lookup();
Class<?> queryHandlerClass = lookup.defineHiddenClass(
    queryHandlerBytecode, true).lookupClass();
// The queryHandlerClass is now a hidden class, not directly usable by other classes.
```

The `defineHiddenClass` method generates a class from bytecode. Since it's not referenced elsewhere, the class remains internal and inaccessible to external classes.[29]

Both sealed and hidden classes enhance the Java language by providing tools for more precise control over class design, improving maintainability, and securing class implementation details.

Java 16: Records, ZGC Concurrent Thread-Stack Processing, and Port to Windows on Arm

Java 16 introduced records, a new class type that streamlines the creation of data-centric classes. Records provide a succinct syntax for defining immutable data structures:

```
record Student(String name, double gpa) {}
```

Leading my team at Microsoft, we contributed to the Java ecosystem by enabling the JDK to function on Windows running on Arm 64 hardware. This significant undertaking was encapsulated in JEP 388: *Windows/AArch64 Port*,[30] where we successfully enabled the

[28] https://openjdk.org/jeps/371

[29] Although the hidden class example omits `MethodHandles.Lookup.ClassOption.NESTMATE` for simplicity, it's worth mentioning for a fuller understanding. `NESTMATE` enables a hidden class to access private members (like methods and fields) of its defining class, which is useful in advanced Java development. This specialized feature is not typically needed for everyday Java use but can be important for complex internal implementations.

[30] https://openjdk.org/jeps/388

template interpreter, the C1 and C2 JIT compilers, and all supported GCs. Our work broadened the environments in which Java can operate, further solidifying its position in the technology ecosystem.

In addition, ZGC in Java 15 evolved into a full supported feature. In Java 16, with JEP 376: *ZGC: Concurrent Thread-Stack Processing*,[31] ZGC underwent further enhancement. This improvement involves moving thread-stack processing to the concurrent phase, significantly reducing GC pause times—bolstering performance for applications managing large data sets.

Java 17 (Java SE 17)

Java 17, the most recent LTS release as of this book's writing, brings a variety of enhancements to the JVM, the Java language, and the Java libraries. These improvements significantly boost the efficiency, performance, and security of Java applications.

JVM Enhancements: A Leap Forward in Performance and Efficiency

Java 17 introduced several notable JVM improvements. JEP 356: *Enhanced Pseudo-Random Number Generators*[32] provides new interfaces and implementations for random number generation, offering a more flexible and efficient approach to generating random numbers in Java applications.

Another significant enhancement is JEP 382: *New macOS Rendering Pipeline*,[33] which improves the efficiency of Java two-dimensional graphics rendering on macOS. This new rendering pipeline leverages the native graphics capabilities of macOS, resulting in better performance and rendering quality for Java applications running on macOS.

Furthermore, JEP 391: *macOS/AArch64 Port*[34] adds a macOS/AArch64 port to the JDK. This port, supported by our Windows/AArch64 port, enables Java to run natively on macOS devices with M1-based architecture, further expanding the platform support for the JVM and enhancing performance on these architectures.

Security and Encapsulation Enhancements: Fortifying the JDK

Java 17 also sought to increase the security of the JDK. JEP 403: *Strongly Encapsulate JDK Internals*[35] makes it more challenging to access internal APIs that are not intended for general use. This change fortifies the security of the JDK and ensures compatibility with future releases by discouraging the use of internal APIs that may change or be removed in subsequent versions.

Another key security enhancement in Java 17 is the introduction of JEP 415: *Context-Specific Deserialization Filters*.[36] This feature provides a mechanism for defining deserialization filters based on context, offering a more granular control over the deserialization process. It addresses

[31] https://openjdk.org/jeps/376
[32] https://openjdk.org/jeps/356
[33] https://openjdk.org/jeps/382
[34] https://openjdk.org/jeps/391
[35] https://openjdk.org/jeps/403
[36] https://openjdk.org/jeps/415

some of the security concerns associated with Java's serialization mechanism, making it safer to use. As a result of this enhancement, application developers can now construct and apply filters to every deserialization operation, providing a more dynamic and context-specific approach compared to the static JVM-wide filter introduced in Java 9.

Language and Library Enhancements

Java 17 introduced several language and library enhancements geared toward improving developer productivity and the overall efficiency of Java applications. One key feature was JEP 406: *Pattern Matching for switch*,[37] which was introduced as a preview feature. This feature simplifies coding by allowing common patterns in `switch` statements and expressions to be expressed more concisely and safely. It enhances the readability of the code and reduces the chances of errors.

Suppose we're using a switch expression to generate a custom message for a student based on their status. The `String.format()` function is used to insert the student's name and GPA into the message:

```
record Student(String name, String status, double gpa) {}

// Let's assume we have a student
Student student = new Student("Monica", "accepted", 3.83);

String admissionResult = switch (student.status()) {
  case "accepted" -> String.format("%s has been accepted with a GPA of %.2f. Congratulations!",
student.name(), student.gpa());
  case "rejected" -> String.format("%s has been rejected despite a GPA of %.2f. Better luck next
time.", student.name(), student.gpa());
  case "waitlisted" -> String.format("%s is waitlisted. Current GPA is %.2f. Keep your fingers
crossed!", student.name(), student.gpa());
  case "pending" -> String.format("%s's application is still pending. Current GPA is %.2f. We hope
for the best!", student.name(), student.gpa());
  default -> String.format("The status of %s is unknown. Please check the application.", student.
name());
};

System.out.println(admissionResult);
```

Moreover, Java 17 includes JEP 412: *Foreign Function and Memory API (Incubator)*,[38] which provides a pure Java API for calling native code and working with native memory. This API is designed to be safer, more efficient, and easier to use than the existing Java Native Interface (JNI). We will delve deeper into foreign function and memory API in Chapter 9, "Harnessing Exotic Hardware: The Future of JVM Performance Engineering."

[37] https://openjdk.org/jeps/406

[38] https://openjdk.org/jeps/412

Deprecations and Removals

Java 17 is also marked by the deprecation and removal of certain features that are no longer relevant or have been superseded by newer functionalities.

JEP 398: *Deprecate the Applet API for Removal*[39] deprecates the Applet API. This API has become obsolete, as most web browsers have removed support for Java browser plug-ins.

Another significant change is the removal of the experimental ahead-of-time (AOT) and JIT compiler associated with Graal, as per JEP 410: *Remove the Experimental AOT and JIT Compiler.*[40] The Graal compiler was introduced as an experimental feature in JDK 9 and was never intended to be a long-term feature of the JDK.

Lastly, JEP 411: *Deprecate the Security Manager for Removal*[41] made the Security Manager, a feature dating back to Java 1.0, no longer relevant and marked it for removal in a future release. The Security Manager was originally designed to protect the integrity of users' machines and the confidentiality of their data by running applets in a sandbox, which denied access to resources such as the file system or the network. However, with the decline of Java applets and the rise of modern security measures, the Security Manager has become less significant.

These changes reflect the ongoing evolution of the Java platform, as it continues to adapt to modern development practices and the changing needs of the developer community. Java 17 is unequivocally the best choice for developing robust, reliable, and high-performing applications. It takes the Java legacy to the next level by significantly improving performance, reinforcing security, and enhancing developer efficiency, thereby making it the undisputed leader in the field.

Embracing Evolution for Enhanced Performance

Throughout Java's evolution, numerous improvements to the language and the JVM have optimized performance and efficiency. Developers have been equipped with innovative features and tools, enabling them to write more efficient code. Simultaneously, the JVM has seen enhancements that boost its performance, such as new garbage collectors and support for different platforms and architectures.

As Java continues to evolve, it's crucial to stay informed about the latest language features, enhancements, and JVM improvements. By doing so, you can effectively leverage these advancements in your applications, writing idiomatic and pragmatic code that fully exploits the latest developments in the Java ecosystem.

In conclusion, the advancements in Java and the JVM have consistently contributed to the performance and efficiency of Java applications. For developers, staying up-to-date with these improvements is essential to fully utilize them in your work. By understanding and applying these new features and enhancements, you can optimize your Java applications' performance and ensure their compatibility with the latest Java releases.

---

[39] https://openjdk.org/jeps/398

[40] https://openjdk.org/jeps/410

[41] https://openjdk.org/jeps/411

Chapter 2

Performance Implications of Java's Type System Evolution

The type system is a fundamental aspect of any programming language, and Java is no exception. It governs how variables are declared, how they interact with each other, and how they can be used in your code. Java is known for its (strongly) static-typed nature. To better understand what that means, let's break down this concept:

- **Java is a statically type-checked language:** That means the variable types are checked (mostly) at compile time. In contrast, dynamically type-checked languages, like JavaScript, perform type checking at runtime. This static type-checking contributes to Java's performance by catching type errors at compile time, leading to more robust and efficient code.
- **Java is a strongly typed language:** This implies that Java will produce errors at compile time if the variable's declared type does not match the type of the value assigned to it. In comparison, languages like C may implicitly convert the value's type to match the variable type. For example, because C language performs implicit conversions, it can be considered a (weakly) static-typed language. In contrast, languages such as assembly languages are untyped.

Over the years, Java's type system has evolved significantly, with each new version bringing its own set of enhancements. This evolution has made the language not only more powerful and flexible, but also easier to use and understand. For instance, the introduction of generics in Java SE 5.0 allowed developers to write more type-safe code, reducing the likelihood of runtime type errors and improving the robustness of applications. The introduction of annotations in Java SE 5.0 provided a way for developers to embed metadata directly into their code; this metadata can be used by the JVM or other tools to optimize functionality or scalability. For example, the `@FunctionalInterface` annotation introduced in Java SE 8 indicates that an interface is intended for use with lambda expressions, aiding in code clarity.

The introduction of variable handle typed references in Java 9 provided a more efficient way to access variables, such as static or instance fields, array elements, and even off-heap areas, through their handles. These handles are typed references to their variables and maintain a minimum viable access set that is compatible with the current state of the Java Memory Model

and C/C++11 atomics. These standards define how threads interact through memory and which atomic operations they can perform. This feature enhances performance and scalability for applications requiring frequent variable access.

The introduction of switch expressions and sealed classes in Java 11 to Java 17 provided more expressive and concise ways to write code, reducing the likelihood of errors and improving the readability of the code. More recently, the ongoing work on Project Valhalla aims to introduce inline classes and generics over primitive types. These features are expected to significantly improve the scalability of Java applications by reducing the memory footprint of objects and enabling flattened data structures, which provide for better memory efficiency and performance by storing data in a more compact format, which both reduces memory usage and increases cache efficiency.

In Chapter 1, "The Performance Evolution of Java: The Language and the Virtual Machine," we explored the performance evolution of Java—the language and the JVM. With that context in mind, let's delve into Java's type system and build a similar timeline of its evolution.

Java's Primitive Types and Literals Prior to J2SE 5.0

Primitive data types, along with object types and arrays, form the foundation of the simplest type system in Java. The language has reserved keywords for these primitive types, such as `int`, `short`, `long`, and `byte` for integer types. Floating-point numbers use `float` and `double` keywords; `boolean` represents boolean types; and `char` is used for characters. String objects are essentially groups of characters and are immutable in nature; they have their own class in Java: `java.lang.String`. The various types of integers and floating-point numbers are differentiated based on their size and range, as shown in Table 2.1.

Table 2.1 **Java Primitive Data Types**

Data Types: Keywords	Size	Range*
char	16-bit Unicode	\u0000–\uffff
boolean	NA	true or false
byte	8-bit signed	$-128 < b \leq 127$
short	16-bit signed	$-32768 < s \leq 32767$
int	32-bit signed	(-2^{31})–$(2^{31} - 1)$*
long	64-bit	(-2^{63})–$(2^{63} - 1)$
float	32-bit single-precision	See [1]
double	64-bit double-precision	See [1]

* Range as defined prior to J2SE 5.0.

The use of primitive types in Java contributes to both performance and scalability of applications. Primitive types are more memory efficient than their wrapper class counterparts, which

are objects that encapsulate primitive types. Thus, the use of primitive types leads to more efficient code execution.

For some primitive types, Java can handle *literals*, which represent these primitives as specific values. Here are a few examples of acceptable literals prior to J2SE 5.0:

```
String wakie = "Good morning!";
char beforeNoon = 'Y';
boolean awake = true;
byte b = 127;
short s = 32767;
int i = 2147483647;
```

In the preceding examples, `"Good morning!"` is a string literal. Similarly, `'Y'` is a char literal, `true` is a boolean literal, and `127`, `32767`, and `2147483647` are `byte`, `short`, and `int` literals, respectively.

Another important literal in Java is *null*. While it isn't associated with any object or type, *null* is used to indicate an unavailable reference or an uninitialized state. For instance, the default value for a string object is `null`.

Java's Reference Types Prior to J2SE 5.0

Java reference types are quite distinct from the primitive data types. Before J2SE 5.0, there were three types of references: interfaces, instantiable classes, and arrays. Interfaces define functional specifications for attributes and methods, whereas instantiable classes (excluding utility classes, which mainly contain static methods and are not meant to be instantiated) implement interfaces to define attributes and methods and to describe object behaviors. Arrays are fixed-size data structures. The use of reference types in Java contributes to scalability of the Java applications, as using interfaces, instantiable classes, and arrays can lead to more modular and maintainable code.

Java Interface Types

Interfaces provide the foundation for methods that are implemented in classes that might otherwise be unrelated. Interfaces can specify code execution but cannot contain any executable code. Here's a simplified example:

```
public interface WakeupDB {
    int TIME_SYSTEM = 24;

    void setWakeupHr(String name, int hrDigits);

    int getWakeupHr(String name);

}
```

In this example, both `setWakeupHr` and `getWakeupHr` are abstract methods. As you can see, there are no implementation details in these abstract methods. We will revisit interface methods when we discuss Java 8. For now, since this section covers the evolution of types only up to J2SE 5.0, just know that each class implementing the interface must include the implementation details for these interface methods. For example, a class `DevWakeupDB` that implements `WakeupDB` will need to implement both the `setWakeupHr()` and `getWakeupHr()` methods, as shown here:

```
public class DevWakeupDB implements WakeupDB {
    //…

    public void SetWakeupHr(String name, int hrDigits) {
        //…
    }

    public int getWakeupHr(String name) {
        //…
    }

    //…
}
```

The earlier example also included a declaration for `TIME_SYSTEM`. A constant declared within an interface is implicitly considered `public`, `static`, and `final`, which means it is always accessible to all classes that implement the interface and is immutable.

Figure 2.1 is a simple class diagram in which `DevWakeupDB` is a class that implements the `WakeupDB` interface. The methods `setWakeupHr` and `getWakeupHr` are defined in the `WakeupDB` interface and implemented in the `DevWakeupDB` class. The `TIME_SYSTEM` attribute is also a part of the `WakeupDB` interface.

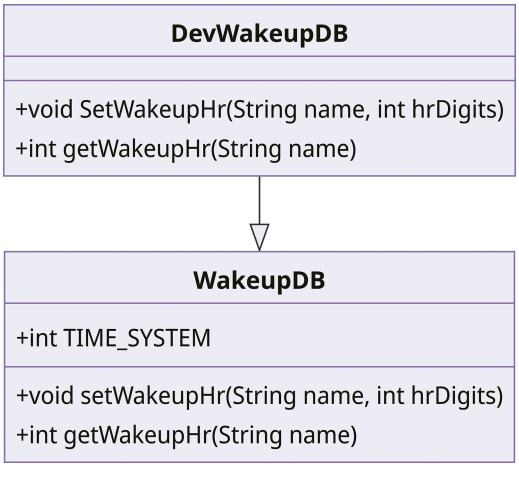

Figure 2.1 Class Diagram Showing Interface

Java Class Types

In the Java type system, a class defines a custom, aggregate data type that can contain heterogeneous data. A Java class encapsulates data and behavior and serves as a blueprint or template for creating objects. Objects have attributes (also known as fields or instance variables) that store information about the object itself. Objects also have methods, which are functions that can perform actions on or with the object's data. Let's look at an example using coding techniques prior to Java SE 5.0, with explicit field and object declarations:

```
class MorningPerson {

    private static boolean trueMorn = false;

    private static boolean Day(char c) {
        return (c == 'Y');
    }

    private static void setMornPers() throws IOException {
        System.out.println("Did you wake up before 9 AM? (Y/N)");
        trueMorn = Day((char) System.in.read());
    }

    private static void getMornPers() {
        System.out.println("It is " + trueMorn + " that you are a morning person");
    }

    public static void main(String[] args) throws IOException {
        setMornPers();
        getMornPers();
    }
}
```

This example shows a class `MorningPerson` with four methods: `Day()`, `setMornPers()`, `getMornPers()`, and `main()`. There is a static boolean class field called `trueMorn`, which stores the boolean value of a `MorningPerson` object. Each object will have access to this copy. The class field is initialized as follows:

```
private static boolean trueMorn = false;
```

If we didn't initialize the field, it would have been initialized to `false` by default, as all declared fields have their own defaults based on the data type.

Here are the two outputs depending on your input character:

```
$ java MorningPerson
Did you wake up before 9 AM? (Y/N)
N
```

```
It is false that you are a morning person

$ java MorningPerson
Did you wake up before 9 AM? (Y/N)
Y
It is true that you are a morning person
```

Java Array Types

Like classes, arrays in Java are considered aggregate data types. However, unlike classes, arrays store homogeneous data, meaning all elements in the array must be of the same type. Let's examine Java array types using straightforward syntax for their declaration and usage.

```
public class MorningPeople {

    static String[] peopleNames = {"Monica ", "Ben ", "Bodin ", "Annika"};
    static boolean[] peopleMorn = {false, true, true, false};

    private static void getMornPeople() {
        for (int i = 0; i < peopleMorn.length; i++) {
            if (peopleMorn[i]) {
                System.out.println(peopleNames[i] + "is a morning person");
            }
        }
    }

    public static void main (String[] args){
        getMornPeople();
    }

}
```

This example includes two single-dimensional arrays: a `String[]` array and a `boolean[]` array, both containing four elements. In the `for` loop, the `string.length` property is accessed, which provides the total count of elements in the array. The expected output is shown here:

```
Ben is a morning person
Bodin is a morning person
```

Arrays provide efficient access to multiple elements of the same type. So, for example, if we want to find early risers based on the day of the week, we could modify the single-dimensional boolean array by turning it into a multidimensional one. In this new array, one dimension (e.g., columns) would represent the days of the week, while the other dimension (rows) would represent the people. This structure can be visualized as shown in Table 2.2.

Table 2.2 **A Multidimensional Array**

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Monica	False	False	True	True	True	True	False
Ben	False	True	True	True	True	True	True
Bodin	True	True	True	True	True	True	False
Annika	False	True	True	True	True	True	True

This would help us get more information, so, for example, we don't disturb Monica on Saturday, Sunday, or Monday mornings. 😉

Java's Type System Evolution from J2SE 5.0 until Java SE 8

J2SE 5.0 brought significant enhancements to the Java type system, including generics, autoboxing, annotations, and enumerations. Both annotations and enumerations were new reference types for Java.

Enumerations

Enumerations provide a way to represent a related set of objects. For instance, to list common environmental allergens in the Austin, Texas, area, we could define an enumeration as follows:

```
enum Season {
    WINTER, SPRING, SUMMER, FALL
}

enum Allergens {
    MOLD(Season.WINTER),
    CEDAR(Season.WINTER),
    OAK(Season.SPRING),
    RAGWEED(Season.FALL),
    DUST(Season.WINTER);

    private Season season;

    Allergens(Season season) {
        this.season = season;
    }

    public Season getSeason() {
        return season;
    }
}
```

Enumerations enhance the type system by providing a type-safe way to define a fixed set of related objects. This can improve code readability, maintainability, and safety. Additionally, the ability to add behavior and associated data to enumerations adds flexibility and expressiveness to your code.

Annotations

Annotations (also referred to as metadata) are modifiers that are preceded by an @ symbol. Users can create their own annotation types by declaring them much as they would declare an interface. Annotation types can have methods, known as annotation type elements. However, annotation type elements can't have any parameters. Let's look at an example:

```
public @interface BookProgress {
    int chapterNum();
    String startDate(); // Format: "YYYY-MM-DD"
    String endDate(); // Format: "YYYY-MM-DD"
    boolean submittedForReview() default false;
    int revNum() default 0;
    String[] reviewerNames(); // List of names of reviewers
    boolean[] reReviewNeeded(); // Corresponding to reviewerNames, indicates if re-review needed
}
```

In this example, we have defined an annotation type called `BookProgress`. Now we can use the `BookProgress` annotation type and provide values for the elements as shown here:

```
@BookProgress(
    chapterNum = 2,
    startDate = "2018-01-01", // Format: "YYYY-MM-DD"
    endDate = "2021-02-01",   // Format: "YYYY-MM-DD"
    submittedForReview = true,
    revNum = 1,
    reviewerNames = {"Ben", "Greg", "Charlie"},
    reReviewNeeded = {true, true, false} // Each boolean corresponds to a reviewer in the same order
)

public class Chapter2 {
        //chapter content …
}
```

Annotations enhance the type system by providing a way to attach metadata to various program elements. While the annotations themselves don't directly affect program performance, they enable tools and frameworks to perform additional checks, generate code, or provide runtime behavior. This can contribute to improved code quality, maintainability, and efficiency.

Other Noteworthy Enhancements (Java SE 8)

As discussed in Chapter 1, the Java language provides some predefined annotation types, such as `@SuppressWarnings`, `@Override`, and `@Deprecated`. Java SE 7 and Java SE 8 introduced two more predefined annotation types: `@SafeVarargs` and `@FunctionalInterface`, respectively. There are a few other annotations that apply to other annotations; they are called *meta-annotations*.

In Java SE 8, generics took a step further by helping the compiler infer the parameter type using "target-type" inference. Additionally, Java SE 8 introduced type annotations that extended the usage of annotations from just declarations to anywhere a type is used. Let's look at an example of how we could use Java SE 8's "generalized target-type inference" to our advantage:

```
HashMap<@NonNull String, @NonNull List<@Readonly String>> allergyKeys;
```

In this code snippet, the `@NonNull` annotation is used to indicate that the keys and values of the `allergyKeys` *HashMap* should never be null. This can help prevent `NullPointerExceptions` at runtime. The hypothetical `@Readonly` annotation on the `String` elements of the `List` indicates that these strings should not be modified. This can be useful in situations where you want to ensure the integrity of the data in your *HashMap*.

Building on these enhancements, Java SE 8 introduced another significant feature: lambda expressions. Lambda expressions brought a new level of expressiveness and conciseness to the Java language, particularly in the context of functional programming.

A key aspect of this evolution is the concept of "target typing." In the context of lambda expressions, target typing refers to the ability of the compiler to infer the type of a lambda expression from its target context. The target context can be a variable declaration, an assignment, a return statement, an array initializer, a method or constructor invocation, or a ternary conditional expression.

For example, we can sort the allergens based on their seasons using a lambda expression:

```
List<Allergens> sortedAllergens = Arrays.stream(Allergens.values())
    .sorted(Comparator.comparing(Allergens::getSeason))
    .collect(Collectors.toList());
```

In this example, the lambda expression `Allergens::getSeason` is a method reference that is equivalent to the lambda expression `allergen -> allergen.getSeason()`. The compiler uses the target type `Comparator<Allergens>` to infer the type of the lambda expression.

Java SE 7 and Java SE 8 introduced several other enhancements, such as the use of the underscore "_" character to enhance code readability by separating digits in numeric literals. Java SE 8 added the ability to use the `int` type for unsigned 32-bit integers (range: 0 to 232 – 1) and expanded the interface types to include static and default methods, in addition to method declarations. These enhancements benefit existing classes that implemented the interface by providing default and static methods with their own implementations.

Let's look at an example of how the interface enhancements can be used:

```
public interface MorningRoutine {
    default void wakeUp() {
        System.out.println("Waking up...");
    }

    void eatBreakfast();

    static void startDay() {
        System.out.println("Starting the day!");
    }
}

public class MyMorningRoutine implements MorningRoutine {
    public void eatBreakfast() {
        System.out.println("Eating breakfast...");
    }
}
```

In this example, `MyMorningRoutine` is a class that implements the `MorningRoutine` interface. It provides its own implementation of the `eatBreakfast()` method but inherits the default `wakeUp()` method from the interface. The `startDay()` method can be called directly from the interface, like so: `MorningRoutine.startDay();`.

The enhancements in Java SE 8 provided more flexibility and extensibility to interfaces. The introduction of static and default methods allowed interfaces to have concrete implementations, reducing the need for abstract classes in certain cases and supporting better code organization and reusability.

Java's Type System Evolution: Java 9 and Java 10

Variable Handle Typed Reference

The introduction of *VarHandles* in Java 9 provided developers with a new way to access variables through their handles, offering lock-free access without the need for `java.util.concurrent.atomic`. Previously, for fast memory operations like atomic updates or prefetching on fields and elements, core API developers or performance engineers often resorted to `sun.misc.Unsafe` for JVM *intrinsics*.

With the introduction of *VarHandles*, the *MethodHandles*[1] API was updated, and changes were made at the JVM and compiler levels to support *VarHandles*. *VarHandles* themselves are represented by the `java.lang.invoke.VarHandle` class, which utilizes a signature polymorphic method. Signature polymorphic methods can have multiple definitions based on the runtime types of the arguments. This enables dynamic dispatch, with the appropriate implementation

[1] "Class MethodHandles." https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/MethodHandles.html.

being selected at runtime. These methods often leverage generic types, promoting code reusability and flexibility. Furthermore, *VarHandles*, as an example of a signature polymorphic method, can handle multiple call sites and support various return types.

The variable access modes can be classified as follows based on their type:

1. Read access (e.g., `getVolatile`): For reading variables with *volatile* memory ordering effects.
2. Write access (e.g., `setVolatile`): For updating variables with *release* memory ordering effects.
3. Read–modify–write access (e.g., `compareAndExchange`, `getAndAddRelease`, `getAndBitwiseXorRelease`): Encompasses three types:
   a. Atomic update: For performing *compare-and-set* operations on variables with *volatile* memory ordering.
   b. Numeric atomic update: For performing *get-and-add* with *release* memory-order effects for updating and *acquire* memory-order effects for reading.
   c. Bitwise atomic update: For performing *get-and-bitwise-xor* with *release* memory-order effects for updating and *volatile* memory-order effects for reading.

The goals of the JDK Enhancement Proposal (JEP) for variable handles[2] were to have predictable performance (as it doesn't involve boxing or arguments packing) equivalent to what `sun.misc.Unsafe` provides, safety (no corrupt state caused by access or update), usability (superior to `sun.misc.Unsafe`), and integrity similar to that provided by `getfield`, `putfield`, and non-updating final fields.

Let's explore an example that demonstrates the creation of a *VarHandle* instance for a static field. Note that we will follow the conventions outlined in "Using JDK 9 Memory Order Modes";[3] that is, *VarHandles* will be declared as static final fields and initialized in static blocks. Also, *VarHandle* field names will be in uppercase.

```
// Declaration and Initialization Example
class SafePin {
    int pin;
}

class ModifySafePin {
    // Declare a static final VarHandle for the "pin" field
    private static final VarHandle VH_PIN;

    static {
         try {
            // Initialize the VarHandle by finding it for the "pin" field of SafePin class
            VH_PIN = MethodHandles.lookup().findVarHandle(SafePin.class, "pin", int.class);
         } catch (Exception e) {
```

[2] JEP 193: *Variable Handles*. https://openjdk.org/jeps/193.

[3] Doug Lea. "Using JDK 9 Memory Order Modes." https://gee.cs.oswego.edu/dl/html/j9mm.html.

```
            // Throw an error if VarHandle initialization fails
            throw new Error(e);
        }
    }
}
```

In this example, we have successfully created a *VarHandle* for direct access to the `pin` field. This *VarHandle* enables us to perform a bitwise or a numeric atomic update on it.

Now, let's explore an example that demonstrates how to use a *VarHandle* to perform an atomic update:

```
// Atomic Update Example
class ModifySafePin {
    private static final VarHandle VH_PIN = ModifySafePin.getVarHandle();

    private static VarHandle getVarHandle() {
          try {
              // Find and return the VarHandle for the "pin" field of SafePin class
              return MethodHandles.lookup().findVarHandle(SafePin.class, "pin", int.class);
          } catch (Exception e) {
              // Throw an error if VarHandle initialization fails
              throw new Error(e);
          }
    }

    public static void atomicUpdatePin(SafePin safePin, int newValue) {
         int prevValue;
         do {
              prevValue = (int) VH_PIN.getVolatile(safePin);
         } while (!VH_PIN.compareAndSet(safePin, prevValue, newValue));
    }
}
```

In this example, the `atomicUpdatePin` method performs an atomic update on the `pin` field of a `SafePin` object by using the *VarHandle* `VH_PIN`. It uses a loop to repeatedly get the current value of `pin` and attempts to set it to `newValue`. The `compareAndSet` method atomically sets the value of `pin` to `newValue` if the current value is equal to `prevValue`. If another thread modifies the value of `pin` between the read operation (`getVolatile`) and the compare-and-set operation (`compareAndSet`), the `compareAndSet` fails, and the loop will repeat until the update succeeds.

This is a simple example of how to use *VarHandles* to perform atomic updates. Depending on your requirements, you might need to handle exceptions differently, consider synchronization, or utilize other methods provided by *VarHandle*. You should always exercise caution when using *VarHandles* because incorrect usage can lead to data races and other concurrency issues. Thoroughly test your code and consider using higher-level concurrency utilities, when possible, to ensure proper management of concurrent variable access.

Java's Type System Evolution: Java 11 to Java 17

Switch Expressions

Java 12 introduced a significant enhancement to the traditional `switch` statement with the concept of switch expressions. This feature was further refined and finalized in Java 14. Switch expressions simplify the coding patterns for handling multiple cases by automatically inferring the type of the returned value. The `yield` keyword (introduced in Java 13) is used to return a value from a switch block.

One of the key improvements was the introduction of the arrow syntax (->) for case labels, which replaced the traditional colon syntax (:). This syntax not only makes the code more readable but also eliminates the need for `break` statements, reducing the risk of fall-through cases.

Here's an example showing how switch expressions can be used:

```
public enum Day {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public String getDayName(Day day) {
   return switch (day) {
      case MONDAY -> "Monday";
      case TUESDAY -> "Tuesday";
      case WEDNESDAY, THURSDAY -> {
         String s = day == Day.WEDNESDAY ? "Midweek" : "Almost Weekend";
         yield s; // yield is used to return a value from a block of code
      }
      // Other cases …
      default -> throw new IllegalArgumentException("Invalid day: " + day);
   };
}
```

In this example, the switch expression is used to return the name of a day based on the value of the `day` variable. The `yield` keyword is used to return a value from a block of code in the case of `WEDNESDAY` and `THURSDAY`.

Switch expressions also improve the compiler's exhaustiveness checking. As a result, the compiler can detect whether all possible cases are covered in a switch expression and issue a warning if any are missing. This feature is especially beneficial when you are dealing with `enum` values. In a traditional `switch` statement, if a new `enum` value is introduced, it could easily be overlooked in the corresponding switch cases, leading to potential bugs or unexpected behavior. However, with switch expressions, the compiler will prompt you to explicitly handle these new cases. This not only ensures code correctness but also enhances maintainability by making it easier to accommodate later changes in your `enum` definitions.

From a maintainability perspective, switch expressions can significantly improve the readability of your code. The compiler can optimize the execution of switch expressions effectively as is

possible with traditional `switch` statements. This optimization is particularly noticeable when you are dealing with cases involving strings.

In traditional `switch` statements, handling string cases involves a sequence of operations by the Java compiler, which may include techniques such as hashing and the use of lookup tables as a part of the compiled bytecode to efficiently manage to jump to the correct case label. This process may seem less direct, but the compiler is equipped to optimize these operations for a multitude of cases.

Switch expressions allow for more concise code and eliminate the possibility of fall-through, which can simplify the compiler's task when generating efficient bytecode. This is particularly helpful with multiple case labels, enabling a streamlined way to handle cases that yield the same result. This approach drastically reduces the number of comparisons needed to find the correct case, thereby significantly streamlining code complexity when there are a substantial number of cases.

Sealed Classes

Sealed classes were introduced as a preview feature in JDK 15 and finalized in JDK 17. They provide developers with a way to define a restricted class hierarchy. This feature allows for better control over class inheritance, thereby simplifying code maintenance and comprehension.

Here's an example that illustrates the concept of sealed classes using various types of pets:

```
sealed abstract class Pet permits Mammal, Reptile {}

sealed abstract class Mammal extends Pet permits Cat, Dog, PygmyPossum {
   abstract boolean isAdult();
}

final class Cat extends Mammal {
    boolean isAdult() {
      return true;
    }
    // Cat-specific properties and methods
}

// Similar implementations for Dog and PygmyPossum

sealed abstract class Reptile extends Pet permits Snake, BeardedDragon {
    abstract boolean isHarmless();
}

final class Snake extends Reptile {
    boolean isHarmless() {
        return true;
    }
    // Snake-specific properties and methods
}

// Similar implementation for BeardedDragon
```

In this example, we have a sealed class `Pet` that permits two subclasses: `Mammal` and `Reptile`. Both of these subclasses are also sealed: `Mammal` permits `Cat`, `Dog`, and `PygmyPossum`, while `Reptile` permits `Snake` and `BeardedDragon`. Each of these classes has an `isAdult` or `isHarmless` method, indicating whether the pet is an adult or is harmless, respectively. By organizing the classes in this way, we establish a clear and logical hierarchy representing different categories of pets.

This well-structured hierarchy demonstrates the power and flexibility of sealed classes in Java, which enable developers to define restricted and logically organized class hierarchies. The JVM fully supports sealed classes and can enforce the class hierarchy defined by the developer at the runtime level, leading to robust and error-free code.

Records

Java 16 introduced records as a preview feature; they became a standard feature in Java 17. Records provide a compact syntax for declaring classes that are primarily meant to encapsulate data. They represent data as a set of properties or fields and automatically generate related methods such as accessors and constructors. By default, records are immutable, which means that their state can't be changed post creation.

Let's look at an example of `Pet` records:

```
List<Pet> pets = List.of(
    new Pet("Ruby", 2, "Great Pyrenees and Red Heeler Mix", true),
    new Pet("Bash", 1, "Maltese Mix", false),
    new Pet("Perl", 10, "Black Lab Mix", true)
);

List<Pet> adultPets = pets.stream()
    .filter(Pet::isAdult)
    .collect(Collectors.toList());

adultPets.forEach(pet -> System.out.println(pet.name() + " is an adult " + pet.breed()));
```

In this code, we create a list of `Pet` records and then use a stream to filter out the pets that are adults. The `filter` method uses the `Pet::isAdult` method reference to filter the stream. The `collect` method then gathers the remaining elements of the stream into a new list. Finally, we use a `forEach` loop to print out the names and breeds of the adult pets.

Incorporating records into your code can enhance its clarity and readability. By defining the components of your class as part of the record definition, you can avoid having to write boilerplate code for constructors, accessor methods, and `equals` and `hashCode` implementations. Additionally, because records are immutable by default, they can be safely shared between threads without the need for synchronization, which can further improve performance in multithreaded environments.

Beyond Java 17: Project Valhalla

Project Valhalla is a highly anticipated enhancement to Java's type system. It aims to revolutionize the way we understand and use the Java object model and generics. The main goals are to introduce inline classes (also known as value classes or types) and user-defined primitives. The project also aims to make generics more versatile to support these new types. These changes are expected to lead to improved performance characteristics, particularly for small, immutable objects and generic APIs. To fully appreciate the potential of these changes, we need to take a closer look at the current performance implications of Java's type system.

Performance Implications of the Current Type System

Java's type system distinguishes between primitive data types and reference types (objects). Primitive data types don't possess an identity; they represent pure data values. The JVM assigns identity only to aggregate objects, such as classes and arrays, such that each instance is distinct and can be independently tracked by the JVM.

This distinction has notable performance implications. Identity allows us to mutate the state of an object, but it comes with a cost. When considering an object's state, we need to account for the object header size and the locality implications of the pointers to this object. Even for an immutable object, we still need to account for its identity and provide support for operations like `System.identityHashCode`, as the JVM needs to be prepared for potential synchronization with respect to the object.

To better understand this, let's visualize a Java object (Figure 2.2). It consists of a header and a body of fields or, in the case of an array object, array elements. The object header includes a mark word and a `klass`, which points to the class's metadata (Figure 2.3). For arrays, the header also contains a length word.

Figure 2.2 A Java Object

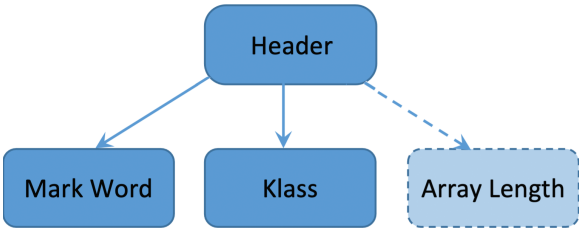

Figure 2.3 A Java Object's Header

NOTE The `klass` will be compressed if `–XX:+UseCompressedOops` is enabled.

Analyzing Object Memory Layout with Java Object Layout (JOL)

Java Object Layout (JOL)[4] is a powerful tool for understanding the memory allocation of objects. It provides a clear breakdown of the overhead associated with each object, including the object header and fields. Here's how we can analyze the layout of an 8-byte array using JOL:

```
import org.openjdk.jol.info.ClassLayout;
import org.openjdk.jol.vm.VM;
import static java.lang.System.out;

public class ArrayLayout {
    public static void main(String[] args) throws Exception {
        out.println(VM.current().details());
        byte[] ba = new byte[8];
        out.println(ClassLayout.parseInstance(ba).toPrintable());
    }
}
```

For more information on how to print internal details, refer to the following two `jol-core` files:[5]

```
org/openjdk/jol/info/ClassLayout.java
org/openjdk/jol/info/ClassData.java
```

For simplicity, let's run this code on 64-bit hardware with a 64-bit operating system and a 64-bit Java VM with `–XX:+UseCompressedOops` enabled. (This option has been enabled by default since Java 6, update 23. If you are unsure which options are enabled by default, use the `–XX:+PrintFlagsFinal` option to check.) Here's the output from JOL:

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
# WARNING | Compressed references base/shifts are guessed by the experiment!
# WARNING | Therefore, computed addresses are just guesses, and ARE NOT RELIABLE.
# WARNING | Make sure to attach Serviceability Agent to get the reliable addresses.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
```

[4] OpenJDK. "Code Tools: jol." https://openjdk.org/projects/code-tools/jol/.

[5] Maven Repository. "Java Object Layout: Core." https://mvnrepository.com/artifact/org.openjdk.jol/jol-core.

```
[B object internals:
OFFSET SIZE TYPE DESCRIPTION     VALUE
   0     4       (object header) 01 00 00 00 (00000001 …) (1)
   4     4       (object header) 00 00 00 00 (00000000 …) (0)
   8     4       (object header) 48 68 00 00 (01001000 …) (26696)
  12     4       (object header) 08 00 00 00 (00001000 …) (8)
  16     8 byte  [B.<elements>          N/A
Instance size: 24 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

Breaking down the output, we can see several sections: the JVM configuration, the byte array internals, the mark word, the `klass`, the array length, the array elements, and possible padding for alignment.

Now let's parse the output in sections.

Section 1: The first three lines

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
```

These lines confirm that we are on a 64-bit Java VM with compressed ordinary object pointers (oops) enabled, and the `klass` is also compressed. The `klass` pointer is a critical component of the JVM as it accesses metadata for the object's class, such as its methods, fields, and superclass. This metadata is necessary for many operations, such as method invocation and field access.

Compressed oops[6] and `klass` pointers are employed to reduce memory usage on 64-bit JVMs. Their use can significantly improve performance for applications that have a large number of objects or classes.

Section 2: Byte array internals

```
[B object internals:
OFFSET SIZE TYPE DESCRIPTION     VALUE
   0     4       (object header) 01 00 00 00 (00000001 …) (1)
   4     4       (object header) 00 00 00 00 (00000000 …) (0)
```

The output also shows the internals of a byte array. Referring back to Figure 2.3, which shows the Java object's header, we can tell that the two lines after the title represent the mark word. The mark word is used by the JVM for object synchronization and garbage collection.

Section 3: `klass`

```
OFFSET SIZE TYPE DESCRIPTION     VALUE
   8     4       (object header) 48 68 00 00 (01001000 …) (26696)
```

[6] https://wiki.openjdk.org/display/HotSpot/CompressedOops

The `klass` section shows only 4 bytes because when compressed oops are enabled, the compressed `klass` is automatically enabled as well. You can try disabling compressed `klass` by using the following option: `-XX:-UseCompressedClassPointers`.

Section 4: Array length

```
OFFSET SIZE TYPE DESCRIPTION     VALUE
  12     4       (object header) 08 00 00 00 (00001000 …) (8)
```

The array length section shows a value of 8, indicating this is an 8-byte array. This value is the array length field of the header.

Section 5: Array elements

```
OFFSET SIZE TYPE DESCRIPTION     VALUE
  16     8 byte  [B.<elements>   N/A
```

The array elements section displays the body/field `array.elements`. Since this is an 8-byte array, the size is shown as 8 bytes in the clarified output. Array elements are stored in a contiguous block of memory, which can improve cache locality and performance.

Understanding the composition of an array object allows us to appreciate the overhead involved. For arrays with sizes of 1 to 8 bytes, we need 24 bytes (the required padding can range from 0 to 7 bytes, depending on the elements):

```
Instance size: 24 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

The instance size and space losses are calculated to understand the memory usage of your objects better.

Now, let's examine a more complex example: an array of the `MorningPeople` class. First, we will consider the mutable version of the class:

```
public class MorningPeopleArray {

    public static void main(String[] args) throws Exception {
        out.println(VM.current().details());

        // Create an array of MorningPeople objects
        MorningPeople[] mornpplarray = new MorningPeople[8];

        // Print the layout of the MorningPeople class
        out.println(ClassLayout.parseClass(MorningPeople.class).toPrintable());

        // Print the layout of the mornpplarray instance
```

```
        out.println(ClassLayout.parseInstance(mornpplarray).toPrintable());
    }
}
class MorningPeople {
    String name;
    Boolean type;

    public MorningPeople(String name, Boolean type) {
        this.name = name;
        this.type = type;
    }
}
```

The output would now appear as follows:

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
# WARNING | Compressed references base/shifts are guessed by the experiment!
# WARNING | Therefore, computed addresses are just guesses, and ARE NOT RELIABLE.
# WARNING | Make sure to attach Serviceability Agent to get the reliable addresses.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

MorningPeople object internals:
 OFFSET  SIZE   TYPE               DESCRIPTION                VALUE
      0    12                      (object header)            N/A
     12     4   java.lang.String   MorningPeople.name         N/A
     16     4   java.lang.Boolean  MorningPeople.type         N/A
     20     4                      (loss due to the next object alignment)
Instance size: 24 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

[LMorningPeople; object internals:
 OFFSET  SIZE   TYPE              DESCRIPTION               VALUE
      0     4                     (object header)           (1)
      4     4                     (object header)           (0)
      8     4                     (object header)           (13389376)
     12     4                     (object header)           (8)
     16    32   MorningPeople     MorningPeople;.<elements> N/A
Instance size: 48 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

Inefficiency with Small, Immutable Objects

Consider the immutable `MorningPeople` class:

```
// An immutable class by declaring it final
final class MorningPeople {
    private final String name;
    private final Boolean type;

    public MorningPeople(String name, Boolean type) {
        this.name = name;
        this.type = type;
    }
    // Getter methods for name and type
    public String getName() {
         return name;
    }
    public Boolean getType() {
        return type;
    }
}
```

In this version of the `MorningPeople` class, the `name` and `type` fields are `private` and `final`, meaning they can't be modified once they're initialized in the constructor. The class is also declared as `final`, so it can't be subclassed. There are no setter methods, so the state of a `MorningPeople` object can't be changed after it's created. This makes the `MorningPeople` class immutable.

If we compile and run this code through JOL, the internals will still be the same as we saw with the mutable version. This outcome highlights an important point: The current Java type system is inefficient for certain types of data. This inefficiency is most evident in cases where we are dealing with large quantities of small, immutable data objects. For such cases, the memory overhead becomes significant, and the issue of locality of reference comes into play. Each `MorningPeople` object is a separate entity in memory, even though the latter is immutable. The references to these objects are scattered across the heap. This scattered layout can, in turn, adversely affect the performance of our Java programs due to poor hardware cache utilization.

The Emergence of Value Classes: Implications for Memory Management

Project Valhalla's planned introduction of value classes is poised to transform the JVM's approach to memory management. By reducing the need for object identity in certain cases, these classes are expected to lighten the garbage collector's workload, leading to fewer pauses and more efficient GC cycles. This change also aims to optimize storage and retrieval operations, especially for arrays, by leveraging the compact layout of value types. However, the path

to integrating these new classes poses challenges, particularly in managing the coexistence of value and reference types. The JVM must adapt its GC strategies and optimize cache usage to handle this new mixed model effectively.

Value classes represent a fundamental shift from traditional class structures. They lack identity, so they differ from objects in the conventional sense. They are proposed to be immutable, which would mean they cannot be synchronized or mutated post creation—attributes that are critical to maintaining state consistency. In the current Java type system, to create an immutable class, as in the `MorningPeople` example, you would typically declare the class as `final` (preventing extension), make all fields `private` (disallowing direct access), omit setter methods, make all mutable fields `final` (allowing only one-time assignment), initialize all fields via a constructor, and ensure exclusive access to any mutable components.

Value classes envision a streamlined process and promise a more memory-efficient storage method by eliminating the need for an object header, enabling inline storage akin to primitive types. This results in several key benefits:

- **Reduced memory footprint:** Inline storage significantly reduces the required memory footprint.
- **Improved locality of reference:** Inline storage ensures that all the fields of a value class are located close to each other in memory. This improved locality of reference can significantly enhance the performance of our programs by optimizing the CPU cache usage.
- **Versatile field types:** Value classes are predicted to support fields of any type, including primitives, references, and nested value classes. This flexibility to have user-defined primitive types allows for more expressive and efficient code.

Redefining Generics with Primitive Support

Recall our discussion of generics in Chapter 1, where we examined the generic type `FreshmenAdmissions<K, V>` using `String` and `Boolean` as type parameters for `K` and `V`, respectively. However, in current Java, generics are limited to reference types. Primitive types, if needed, are used via autoboxing with their wrapper classes, as demonstrated in the snippet from Chapter 1:

```
applicationStatus.admissionInformation("Monica", true);
```

Under Project Valhalla, the evolution of Java's type system aims to include support for generics over primitive types, eventually extending this support to value types. Consider the potential definition of the `FreshmenAdmissions` class in this new context:

```
public class FreshmenAdmissions<K, any V> {
    K key;
    V boolornumvalue;

    public void admissionInformation(K name, V value) {
```

```
        key = name;
        boolornumvalue = value;
    }
}
```

Here, the `any` keyword introduces a significant shift, allowing generics to directly accept primitive types without the need for boxing. For instance, in the modified `FreshmenAdmissions<K,any V>` class, `V` can now directly be a primitive type.[7] To demonstrate this enhancement, consider the `TestGenerics` class using the primitive type `boolean` instead of the wrapper class `Boolean`:

```
public class TestGenerics {
    public static void main(String[] args) {
        FreshmenAdmissions<String, boolean> applicationStatus = new FreshmenAdmissions<>();
        applicationStatus.admissionInformation("Monica", true);
    }
}
```

This example illustrates how the direct use of primitive types in generics could simplify code and improve performance, a key goal of Project Valhalla's advancements in the Java type system.

Exploring the Current State of Project Valhalla

Classes and Types

Project Valhalla introduces several novel concepts to the Java class hierarchy, such as inline (value) classes, primitive classes, and enhanced generics. Value classes, which were discussed in the preceding section, have a unique characteristic: When two instances of a value class are compared using the == operator, it checks for content equality rather than instance identity (because value classes do not have identity). Furthermore, synchronization on value classes is not possible. The JVM can use these features to save memory by avoiding unnecessary heap allocations.

Primitive classes are a special type of value class. They're not reference types, which means they can't be null. Instead, if you don't assign them a value, their values will default to zero. Primitive classes are stored directly in memory, rather than requiring a separate heap allocation. As a result, arrays of primitive classes don't need to point to different parts of the heap for each element, which makes programs run both more quickly and more efficiently. When necessary, the JVM can "box" a primitive class into a value class, and vice versa.[8]

[7] Nicolai Parlog. "Java's Evolution into 2022: The State of the Four Big Initiatives." Java Magazine, February 18, 2022. https://blogs.oracle.com/javamagazine/post/java-jdk-18-evolution-valhalla-panama-loom-amber.

[8] JEP 401: *Value Classes and Objects (Preview).* https://openjdk.org/jeps/401.

Significantly, Project Valhalla also introduces primitive classes for the eight basic types in the JVM: `byte`, `char`, `short`, `int`, `long`, `boolean`, `float`, and `double`. This makes the type system more elegant and easier to understand because it is no longer necessary to treat these types differently from other types. The introduction of these primitive classes for the basic types is a key aspect of Project Valhalla's efforts to optimize Java's performance and memory management, making the language more efficient for a wide range of applications.

Project Valhalla is working on improving generics so that they will work better with value classes. The goal is to help generic APIs perform better when used with value classes.

Memory Access Performance and Efficiency

One of the main goals of Project Valhalla is to improve memory access performance and efficiency. By allowing value classes and primitive classes to be stored directly in memory, and by improving generics so that they work better with these types, Project Valhalla aims to reduce memory usage and improve the performance of Java applications. This would be especially beneficial for lists or arrays, which can store their values directly in a block of memory, without needing separate heap pointers.

NOTE To stay updated on the latest advancements and track the ongoing progress of Project Valhalla, you have several resources at your disposal. The OpenJDK's Project Valhalla mailing list[9] is a great starting point. Additionally, you can export the GitHub repository[10] for in-depth insights and access to source codes. For hands-on experience, the latest early-access downloadable binaries[11] offer a practical way to experiment with OpenJDK prototypes, allowing you to explore many of these new features.

Early Access Release: Advancing Project Valhalla's Concepts

The early access release of Project Valhalla JEP 401: *Value Classes and Objects*[12] marks a significant step in realizing the project's ambitious goals. While previous sections discussed the theoretical framework of value classes, the early access (EA) release concretizes these concepts with specific implementations and functionalities. Key highlights include

- **Implementation of value objects:** In line with the Java Language Specification for Value Objects,[13] this release introduces value objects in Java, emphasizing their identity-free behavior and the ability to have inline, allocation-free encodings in local variables or method arguments.

[9] https://mail.openjdk.org/mailman/listinfo/valhalla-dev

[10] https://github.com/openjdk/valhalla/tree/lworld

[11] https://jdk.java.net/valhalla/

[12] https://openjdk.org/jeps/401

[13] https://cr.openjdk.org/~dlsmith/jep8277163/jep8277163-20220830/specs/value-objects-jls.html#jls-8.1.1.5

- **Enhancements and experimental features:** The EA release experiments with features like flattened fields and arrays, and developers can use the `.ref` suffix to refer to the corresponding reference type of a primitive class. This approach aims to provide null-free primitive value types, potentially transforming Java's approach to memory management and data representation.
- **Command-line options for experimentation:** Developers can use options like `-XDenablePrimitiveClasses` and `-XX:+EnablePrimitiveClasses` to unlock experimental support for *Primitive Classes*, while other options like `–XX:InlineFieldMaxFlatSize=n` and `–XX:FlatArrayElementMaxSize=n` allow developers to set limits on flattened fields and array components.
- **HotSpot optimizations:** Significant optimizations in HotSpot's C2 compiler specifically target the efficient handling of value objects, aiming to reduce the need for heap allocations.

With these enhancements, Project Valhalla steps out of the theoretical realm and into a more tangible form, showcasing the practical implications and potential of its features. As developers explore these features, it's important to note the compatibility implications. Refactoring an identity class to a value class may affect existing code. Additionally, value classes can now be declared as records and made serializable, expanding their functionality. Core reflection has also been updated to support these new modifiers. Finally, it's crucial for developers to understand that value class files generated by *javac* in this EA release are specifically tailored for this version. They may not be compatible with other JVMs or third-party tools, indicating the experimental nature of these features.

Use Case Scenarios: Bringing Theory to Practice

In high-performance computing and data processing applications, value classes can offer substantial benefits. For example, financial simulations that handle large volumes of immutable objects can leverage the memory efficiency and speed of value classes for more efficient processing.[14] Enhanced generics enable API designers to create more versatile yet less complex APIs, promoting broader applicability and easier maintenance.

Concurrent applications stand to benefit significantly from the inherent thread safety of value classes. The immutability of value classes aligns with thread safety, reducing the complexity associated with synchronization in multithreaded environments.

A Comparative Glance at Other Languages

Java's upcoming value classes, part of Project Valhalla, aim to optimize memory usage and performance similar to C#'s value types. However, Java's value classes are expected to offer enhanced features for methods and interfaces, diverging from C#'s more traditional approach with structs. While C#'s structs provide efficient memory management, they face performance costs associated with boxing when used as reference types. Java's approach is focused

[14] https://openjdk.java.net/projects/valhalla/

on avoiding such overhead and enhancing runtime efficiency, particularly in collections and arrays.[15]

Kotlin's data classes offer functional conveniences akin to Java's records, automating common methods like `equals()` and `hashCode()`, but they aren't tailored for memory optimization. Instead, Kotlin's inline classes are the choice for optimizing memory. Inline classes avoid overhead by embedding values directly in the code during compilation, which is comparable to the goals of Java's proposed value classes in Project Valhalla for reducing runtime costs.[16]

Scala's case classes are highly effective for representing immutable data structures, promoting functional programming practices. They inherently provide pattern matching, `equals()`, `hashCode()`, and `copy()` methods. However, unlike Java's anticipated value classes, Scala's case classes do not specialize in memory optimization or inline storage. Their strength lies in facilitating immutable, pattern-friendly data modeling, rather than in low-level memory management or performance optimization typical of value types.[17]

Conclusion

In this chapter, we've explored the significant strides made in Java's type system, leading up to the advancements now offered by Project Valhalla. From the foundational use of primitive and reference types to the ambitious introduction of value types and enhanced generics, Java's evolution reflects a consistent drive toward efficiency and performance.

Project Valhalla, in particular, marks a notable point in this journey, bringing forward concepts that promise to refine Java's capabilities further. These changes are pivotal for both new and experienced Java developers, offering tools for more efficient and effective coding. Looking ahead, Project Valhalla sets the stage for Java's continued relevance in the programming world—it represents a commitment to modernizing the language while maintaining its core strengths.

[15] https://docs.microsoft.com/en-us/dotnet/csharp/

[16] https://kotlinlang.org/docs/reference/data-classes.html

[17] https://docs.scala-lang.org/tour/case-classes.html

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");
    }
}
```