

Dr. Edward Lavieri, Peter Verhas

Mastering Java 9.

Write reactive, modular, concurrent, and secure code



Packt

Mastering Java 9

Write reactive, modular, concurrent, and secure code

Dr. Edward Lavieri
Peter Verhas

Packt

BIRMINGHAM - MUMBAI

Mastering Java 9

Copyright © 2017 Packt Publishing

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

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

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2017

Production reference: 1031017

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street

Birmingham
B3 2PB, UK.

ISBN 978-1-78646-873-4

www.packtpub.com

Credits

Authors Dr. Edward Lavieri Peter Verhas	Copy Editor Safis Editing
Reviewer Mandar Jog	Project Coordinator Prajakta Naik
Commissioning Editor Kunal Parikh	Proofreader Safis Editing

Acquisition Editor

Denim Pinto

Indexer

Francy Puthiry

Content Development Editor

Lawrence Veigas

Graphics

Jason Monteiro

Technical Editor

Abhishek Sharma

Production Coordinator

Arvindkumar Gupta

About the Authors

Dr. Edward Lavieri is a veteran developer with a strong academic background. He earned a doctorate in computer science from Colorado Technical University, an MS in management information systems (Bowie State University), an MS in education (Capella University), and an MS in operations management (University of Arkansas).

He has been creating and teaching computer science courses since 2002. Edward retired from the U.S. Navy as a Command Master Chief after 25 years of service. As the founder and creative director of three19, a software design and development studio, Edward is constantly designing and developing software. He uses a variety of game engines and development tools. His passions include developing adaptive learning systems, educational games, and mobile apps.

Edward has authored *Adaptive Learning for Educational Game Design* (CreateSpace), *Getting Started with Unity 5* (Packt), *Learning AWS Lumberyard Game Development* (Packt), *LiveCode Mobile Development HOTSHOT* (Packt), *LiveCode Mobile Development Cookbook* (Packt), and *Software Consulting: A Revolutionary Approach* (CreateSpace). He was the technical editor for *Excel Formulas and Functions for Dummies* (Wiley Publishing). He has also developed numerous college courses involving computer science, information systems, and game development.

Peter Verhas is a senior developer and software system architect with more than 30 years of software development experience. He currently works for EPAM as a senior developer, where he is involved in many customer projects and actively participates in the education activities of the company. Peter writes a technical blog and is committed to open source software development. He has been using Java since 2005

and is also an author at Java Code Geeks.

About the Reviewer

Mandar Jog is an expert IT trainer with over 15 years of training experience. He is an expert in technologies such as Java, J2EE, and Android. He also holds SCJP and SCWCD certifications. He is an occasional blogger where he makes the readers feel "I can" for the complex concepts in Java and J2EE. He is a regular speaker at many engineering colleges for technical seminars and workshops.

He was also the technical reviewer of the book, *Modular Programming in Java 9* by Packt.

Thanks a lot Tejaswini, you have been my inspiration throughout this journey. I am equally grateful to my son, Ojas; his lovely smiles have always made me push myself further.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1786468735>.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

- What this book covers
- What you need for this book
- Who this book is for
- Conventions
- Reader feedback
- Customer support
 - Downloading the example code
 - Errata
 - Piracy
 - Questions

1. The Java 9 Landscape

- Java 9 at 20,000 feet
- Breaking the monolith
- Playing around with the Java Shell
- Taking control of external processes
- Boosting performance with G1
- Measuring performance with JMH
- Getting started with HTTP 2.0
- Encompassing reactive programming
- Expanding the wish list
- Summary

2. Discovering Java 9

- Improved Contended Locking [JEP 143]
 - Improvement goals
 - Segmented code cache [JEP 197]
 - Memory allocation
 - Smart Java compilation, phase two [JEP 199]
 - Resolving Lint and Doclint warnings [JEP 212]
 - Tiered attribution for javac [JEP 215]
 - Annotations pipeline 2.0 [JEP 217]
 - New version-string scheme [JEP 223]
 - Generating run-time compiler tests automatically [JEP 233]
 - Testing class-file attributes generated by Javac [JEP 235]
 - Storing interned strings in CDS archives [JEP 250]
 - The problem
 - The solution
 - Preparing JavaFX UI controls and CSS APIs for modularization [JEP 253]
 - JavaFX overview

Implications for Java 9
Compact strings [JEP 254]

Pre-Java 9 status

New with Java 9

Merging selected Xerces 2.11.0 updates into JAXP [JEP 255]

Updating JavaFX/Media to newer version of GStreamer [JEP 257]

HarfBuzz Font-Layout Engine [JEP 258]

HiDPI graphics on Windows and Linux [JEP 263]

Marlin graphics renderer [JEP 265]

Unicode 8.0.0 [JEP 267]

New in Unicode 8.0.0

Updated Classes in Java 9

Reserved stack areas for critical sections [JEP 270]

The pre-Java 9 situation

New in Java 9

Dynamic linking of language-defined object models [JEP 276]

Proof of concept

Additional tests for humongous objects in G1 [JEP 278]

Improving test-failure troubleshooting [JEP 279]

Environmental information

Java process information

Optimizing string concatenation [JEP 280]

HotSpot C++ unit-test framework [JEP 281]

Enabling GTK 3 on Linux [JEP 283]

New HotSpot build system [JEP 284]

Summary

3. Java 9 Language Enhancements

Working with variable handlers [JEP 193]

Working with the AtoMiC Toolkit

Using the sun.misc.Unsafe class

Eliding depreciation warnings on import statements [JEP 211]

Milling Project Coin [JEP 213]

Using the @SafeVarargs annotation

The try-with-resource statement

Using the diamond operator

Discontinuing use of the underscore

Making use of private interface methods

Processing import statements correctly [JEP 216]

Summary

4. Building Modular Applications with Java 9

A modular primer

Reviewing Java's platform module system [JEP-200]

Modularizing JDK source code [JEP-201]
Pre-Java 9 JDK source code organization
Development tools

- Deployment
- Internationalization
- Monitoring
- RMI
- Security
- Troubleshooting
- Web services

JavaFX tools

Java runtime environment

Source code

- Libraries
- C header files
- Database

JDK source code reorganized

Understanding modular run-time images [JEP-220]

- Runtime format adoption
- Runtime image restructure
- Supporting common operations
- De-privileging JDK classes
- Preserving existing behaviors

Getting to know the module system [JEP-261]

- Module paths
- Access-control boundary violations

Runtime

Modular Java application packaging [JEP-275]

- Advanced look at the Java Linker
- Java Packager options

JLink - The Java Linker [JEP-282]

Encapsulating most internal APIs [JEP-260]

Summary

5. Migrating Applications to Java 9

- Quick review of Project Jigsaw

Classpath

The monolithic nature of the JDK

How modules fit into the Java landscape

- Base module
- Reliable configuration
- Strong encapsulation

Migration planning

Testing a simple Java application

Potential migration issues

The JRE
Access to internal APIs
Accessing internal JARs
JAR URL depreciation
Extension mechanism
The JDK's modularization
Advice from Oracle
Preparatory steps
Getting the JDK 9 early access build
Running your program before recompiling
Updating third-party libraries and tools
Compiling your application
 Pre-Java 9 -source and -target options
 Java 9 -source and -target options
 Running jdeps on your code
Breaking encapsulation
 The --add-opens option
 The --add-exports option
 The --permit-illegal-access option
Runtime image changes
 Java version schema
 JDK and JRE layout
What has been removed
Updated garbage collection
Deployment
 JRE version selection
 Serialized applets
 JNLP update
 Nested resources
 FX XML extension
 JNLP file syntax
 Numeric version comparison
Useful tools
 Java environment - jEnv
 Maven
 Obtaining the M2Eclipse IDE
Summary

6. Experimenting with the Java Shell

What is JShell?
Getting Started with JShell
Practical uses of JShell
 Feedback modes
 Creating a custom feedback mode
 Listing your assets

- Editing in the JShell
 - Modifying text
 - Basic navigation
 - Historical navigation
 - Advanced editing commands
 - Working with scripts
 - Start up scripts
 - Loading scripts
 - Saving scripts
 - Advanced scripting with JShell
 - Summary
7. Leveraging the New Default G1 Garbage Collector
- Overview of garbage collection
 - Object life cycle
 - Object creation
 - Object mid-life
 - Object destruction
 - Garbage collection algorithms
 - Mark and sweep
 - Concurrent mark sweep (CMS) garbage collection
 - Serial garbage collection
 - Parallel garbage collection
 - G1 garbage collection
 - Garbage collection options
 - Java methods relevant to garbage collection
 - The System.gc() method
 - The finalize() method
 - Pre-Java 9 garbage collection
 - Visualizing garbage collection
 - Garbage collection upgrades in Java 8
 - Case study - Games written with Java
 - Collecting garbage with the new Java platform
 - Default garbage collection
 - Deprecated garbage collection combinations
 - Unified garbage collection logging
 - Unified JVM logging (JEP-158)
 - Tags
 - Levels
 - Decorations
 - Output
 - Command-line options
 - Unified GC logging (JEP-271)
 - Garbage collection logging options
 - The gc tag

- Macros
- Additional considerations
- Persistent issues
 - Making objects eligible for garbage collection
- Summary
- 8. Microbenchmarking Applications with JMH
 - Microbenchmarking overview
 - Approach to using JMH
 - Installing Java 9 and Eclipse with Java 9 support
 - Hands-on experiment
 - Microbenchmarking with Maven
 - Benchmarking options
 - Modes
 - Time units
 - Techniques for avoiding microbenchmarking pitfalls
 - Power management
 - OS schedulers
 - Time sharing
 - Eliminating dead-code and constant folding
 - Run-to-run variance
 - Cache capacity
 - Summary
- 9. Making Use of the ProcessHandle API
 - What are processes?
 - The new ProcessHandle interface
 - Getting the PID of the current process
 - Getting information about a process
 - Listing processes
 - Listing children
 - Listing descendants
 - Listing all processes
 - Waiting for processes
 - Terminating processes
 - A small process controller application
 - Main class
 - Parameters class
 - The ParamsAndHandle class
 - The ControlDaemon class
 - Summary
- 10. Fine-Grained Stack Tracing
 - Overview of the Java Stack
 - The importance of stack information

- Example - Restricting callers
- Example - Getting logger for caller
- Working with StackWalker
 - Getting an instance of StackWalker
 - RETAIN_CLASS_REFERENCE
 - SHOW_REFLECT_FRAMES
 - SHOW_HIDDEN_FRAMES
 - Final thoughts on enum constants
- Accessing classes
- Walking methods
- StackFrame
- Performance
- Summary

11. New Tools and Tool Enhancements

- The new HTTP client [JEP-110]
 - The HTTP client before Java 9
 - Java 9's new HTTP client
 - New API limitations
- Simplified Doclet API [JEP-221]
 - The pre-Java 9 Doclet API
 - API enums
 - API classes
 - API interfaces
 - Problems with the pre-existing Doclet API
 - Java 9's Doclet API
 - Compiler tree API
 - Language model API
 - The AnnotatedConstruct interface
 - The SourceVersion enum
 - The UnknownEntityException exception
- HTML5 Javadoc [JEP-224]
- Javadoc search [JEP-225]
 - Introducing camel case search
- Remove launch-time JRE version selection [JEP-231]
- Parser API for Nashorn [JEP-236]
 - Nashorn
 - Using Nashorn as a command-line tool
 - Using Nashorn as an embedded interpreter
 - EMCAScript
 - Parser API
- Multi-release JAR files [JEP-238]
 - Identifying multi-release JAR files
 - Related JDK changes
- Java-level JVM compiler interface [JEP-243]

BeanInfo annotations [JEP-256]
 JavaBean
 BeanProperty
 SwingContainer
 BeanInfo classes
TIFF image input/output [JEP-262]
Platform logging API and service [JEP-264]
 The java.util.logging package
 Logging in Java 9
XML Catalogs [JEP-268]
 The OASIS XML Catalog standard
 JAXP processors
 XML Catalogs prior to Java 9
 Java 9 platform changes
Convenience factory methods for collections [JEP-269]
 Using collections before Java 9
 Using new collection literals
Platform-specific desktop features [JEP-272]
Enhanced method handles [JEP-274]
 Reason for the enhancement
 Lookup functions
 Argument handling
 Additional combinations
Enhanced deprecation [JEP-277]
 What the @Deprecated annotation really means
Summary

12. Concurrency and Reactive Programming

 Reactive Programming
 Reactive programming standardization
 The New Flow API
 The Flow.Publisher interface
 The Flow.Subscriber interface
 The Flow.Subscription interface
 The Flow.Processor interface
 Sample implementation
 Additional Concurrency Updates
 Java concurrency
 Concurrency explained
 System configurations
 Java threads
 Concurrency improvements
 CompletableFuture API enhancements
 Class details
 Enhancements

- Spin-Wait Hints
 - Summary
- 13. Security Enhancements
 - Datagram Transport Layer Security
 - DTLS protocol version 1.0
 - DTLS protocol version 1.2
 - DTLS support in Java 9
 - Creating PKCS12 keystores
 - Keystore primer
 - Java Keystore (JKS)
 - Builder
 - The CallbackHandlerProtection class
 - The PasswordProtection class
 - The PrivateKeyEntry class
 - The SecretKeyEntry class
 - The TrustedCertificateEntry class
 - PKCS12 default in Java 9
 - Improving security application performance
 - Security policy enforcement
 - Permission evaluation
 - The java.Security.CodeSource package
 - Package checking algorithm
 - TLS application-layer protocol negotiation extension
 - TLS ALPN extension
 - The javax.net.ssl package
 - The java.net.ssl package extension
 - Leveraging CPU instructions for GHASH and RSA
 - Hashing
 - OCSP stapling for TLS
 - OCSP stapling primer
 - Changes for the Java 9 platform
 - DRBG-based SecureRandom implementations
 - Summary
- 14. Command Line Flags
 - Unified JVM Logging [JEP 158]
 - Command-line options
 - Decorations
 - Levels
 - Output
 - Tags
 - Compiler control [JEP 165]
 - Compilation modes
 - C1 compilation mode

- C2 compilation mode
- Tiered compilation
- Compiler control in Java 9
- Diagnostic commands [JEP 228]
- Heap profiling agent [JEP 240]
- Removing your JHAT [JEP 241]
- JVM command-line flag argument validation [JEP 245]
- Compile for older platform versions [JEP 247]

Summary

15. Best Practices In Java 9

Support for UTF-8

- The ResourceBundle class
- The nested class
- Fields and constructors
- Methods

Changes in Java 9

Unicode 7.0.0

- The java.lang package
- The java.text package
- Additional significance

The Linux/AArch64 port

Multi-resolution Images

Common Locale Data Repository (CLDR)

Summary

16. Future Directions

Future Changes to the JDK

JDK changes targeted for Java 10

- Repository consolidation
- Native-header tool removal
- JDK-related submitted proposals

Parallelize the Full GC Phase in CMS

REST APIs for JMX

Support heap allocation

JDK-related drafted proposals

Finalization promptness

Java memory model

Foreign Function Interfaces

Isolated methods

Reducing metaspace waste

Improving IPv6 support

Unboxed argument lists for method handles

Enhanced MandelblotSet demo using value types

Efficient array comparison intrinsics
Future changes to the Java Compiler
Policy for retiring javac -source and -target options
Pluggable static analyzers
Future Changes to the Java Virtual Machine
JVM-related submitted proposals
Container aware Java
Enable execution of Java methods on GPU
Epsilon GC - The arbitrarily low overhead garbage (non-) collector
JVM-related drafted proposals
Provide stable USDT probe points on JVM compiled methods
Concurrent monitor deflation
Provide a low-overhead way of sampling Java heap allocations
Diagnostic Command Framework
Enhanced Class Redefinition
Enable NUMA mode by default when appropriate
Value objects
Align JVM Access Checks

Future Changes to JavaX
JMX specific annotations for registration of managed resources
Modernize the GTK3 Look and Feel Implementation

Ongoing Special Projects
Annotations pipeline 2.0
Audio Synthesis Engine
Caciocavallo
Common VM Interface
Compiler Grammar
Da Vinci Machine
Device I/O
Graal
HarfBuzz Integration
Kona
OpenJFX
Panama
Shenandoah

Summary

Preface

Java 9 and its new features add to the richness of the language--one of the most-used languages to build robust software applications. Java 9 comes with a special emphasis on modularity, implemented from Project Jigsaw. This book is your one-stop guide to mastering the changes made to the Java platform.

The book gives an overview and explanation of the new features introduced in Java 9 and the importance of the new APIs and enhancements. Some of the new features of Java 9 are groundbreaking, and if you are an experienced programmer, you will be able to make your enterprise application leaner by implementing these new features. You will be provided with practical guidance in applying the newly acquired knowledge in regards to Java 9 and further information on future developments of the Java platform. This book will improve your productivity, making your applications faster. By learning the best practices in Java, you will become the *go-to* person in your organization for Java 9.

By the end of this book, you will not only know the important concepts of Java 9 but you will also have a nuanced understanding of the important aspects of programming with this great language.

What this book covers

[Chapter 1](#), *The Java 9 Landscape*, explores the most significant features introduced in Java 9, including Project Jigsaw, the Java Shell, G1 garbage collection, and reactive programming. This chapter provides introductions to these topics, priming them for deeper coverage in the subsequent chapters.

[Chapter 2](#), *Discovering Java 9*, covers several changes to the Java platform to include heap space efficiencies, memory allocation, compilation process improvements, type testing, annotations, automated runtime compiler tests, and improved garbage collection.

[Chapter 3](#), *Java 9 Language Enhancements*, focuses on the changes made to the Java language. These changes impact variable handlers, depreciation warnings, improvements on Project Coin changes implemented in Java 7, and import statement processing.

[Chapter 4](#), *Building Modular Applications with Java 9*, examines the structure of a Java module as specified by Project Jigsaw and how Project Jigsaw was implemented as part of the Java platform. This chapter also reviews the key internal changes to the Java platform as they relate to the new modular system.

[Chapter 5](#), *Migrating Applications to Java 9*, explores how to migrate Java 8 applications to the Java 9 platform. Both manual and semi-automated migration processes are covered.

[Chapter 6](#), *Experimenting with the Java Shell*, covers JShell, the new command-line read-eval-print loop tool in Java 9. Coverage includes information regarding the tool, the read-eval-print loop concept, and the commands and command-line options for use with JShell.

[Chapter 7](#), *Leveraging the New Default G1 Garbage Collector*, takes an in-depth look at garbage collection and how it is handled in Java 9.

[Chapter 8](#), *Microbenchmarking Applications with JMH*, examines how to write performance tests using the Java Microbenchmark Harness (JMH), a Java harness library for writing benchmarks for the Java Virtual Machine (JVM). Maven is used

along with JMH to help illustrate the power of microbenchmarking with the new Java 9 platform.

[Chapter 9](#), *Making Use of the ProcessHandle API*, reviews new class APIs that enable the management of operating system processes.

[Chapter 10](#), *Fine-Grained Stack Tracing*, covers the new API that permits an effective means of stack walking. The chapter includes detailed information on how to access stack trace information.

[Chapter 11](#), *New Tools and Tool Enhancements*, covers 16 Java Enhancement Proposals (JEPs) that were incorporated into the Java 9 platform. These JEPs cover a wide range of tools and updates to APIs to make development with Java easier with greater optimization possibilities for our Java applications.

[Chapter 12](#), *Concurrency Enhancements*, covers concurrency enhancements introduced with the Java 9 platform. The primary focus is the support for reactive programming, a concurrency enhancement that is provided by the Flow class API. Additional concurrency enhancements introduced in Java 9 are also covered.

[Chapter 13](#), *Security Enhancements*, covers several small changes made to the JDK that involve security. The security enhancements introduced with the Java 9 platform provide developers with a greater ability to write and maintain applications that are more secure than previously possible.

[Chapter 14](#), *Command-Line Flags*, explores the command-line flag changes in Java 9. Concepts covered in this chapter include unified JVM logging, compiler control, diagnostic commands, heap-profiling agent, JHAT, command-line flag argument validation, and compiling for older platform versions.

[Chapter 15](#), *Best Practices in Java 9*, focuses on working with utilities provided by the Java 9 platform to include UTF-8 property files, Unicode 7.0.0, Linux/AArch64 port, multiresolution images, and common locale data repository.

[Chapter 16](#), *Future Directions*, provides an overview of the future developments of the Java platform, beyond Java 9. This includes a specific look at what is planned for Java 10 and what further changes we are likely to see in the future.

What you need for this book

To work with this text, you will need at least a basic knowledge of the Java programming language.

You will also need the following software components:

- Java SE Development Kit 9 (JDK)
 - <http://www.oracle.com/technetwork/java/javase/downloads/>
- An Integrated Development Environment (IDE) for coding. Here are suggestions:
 - Eclipse
 - <https://www.eclipse.org>
 - IntelliJ
 - <https://www.jetbrains.com/idea/>
 - NetBeans
 - <https://netbeans.org>

Who this book is for

This book is for enterprise developers and existing Java developers. Basic knowledge of Java is necessary.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, and user input are shown as follows: "Under the subdirectory structure of `c:\chapter8-benchmark\src\main\java\com\packt` is the `MyBenchmark.java` file."

A block of code is set as follows:

```
public synchronized void protectedMethod()
{
    . . .
}
```

New terms and important words are shown in bold.



Warnings or important notes appear like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book--what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply email feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Java-9>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books--maybe a mistake in the text or the code--we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

The Java 9 Landscape

Java is already a fully-grown adult in its own right more than two decades since its first release. With a stunning community of developers and wide adoption in a number of industries, the platform continues to evolve and keep up with the rest of the world in terms of performance, security, and scalability. We will begin our journey by exploring the most significant features introduced in Java 9, what are the biggest drivers behind them, and what more we can expect in subsequent developments of the platform, along with some of the things that did not make it in this release.

In this chapter, we will cover the following topics:

- Java 9 at 20,000 feet
- Breaking the monolith
- Playing around with the Java Shell
- Taking control of external processes
- Boosting performance with G1
- Measuring performance with JMH
- Getting ready for HTTP 2.0
- Encompassing reactive programming
- Expanding the wish list

Java 9 at 20,000 feet

You might be asking yourself--isn't Java 9 just a maintenance release with a set of features that did not make it into Java 8? There is plenty of new stuff in Java 9 that makes it a distinct version in its own right.

Inarguably, the modularization of the Java platform (developed as part of project Jigsaw) is the biggest piece of work that makes it successfully in Java 9. Initially planned for Java 8, but postponed, project Jigsaw is also one of the main reasons why the final release of Java 9 was further postponed. Jigsaw also introduces a few notable changes to the Java platform and is one of the reasons Java 9 is considered a major release. We will explore these features in detail in the subsequent chapters.

The **JCP (Java Community Process)** provides the mechanisms to turn a set of feature proposals (also known as **Java Enhancement Proposals** or **JEPs**) into formal specifications that provide the basis to extend the platform with new functionality. Java 9 is no different in that regard. Apart from the Jigsaw-related Java enhancement proposals, there is a long list of other enhancements that made it in Java 9. Throughout this book, we will discuss the various features in terms of logical groups based on the corresponding enhancement proposals, including the following:

- The **Java Shell** (also called **JShell**)--an interactive shell for the Java platform
- New APIs to work with operating system processes in a portable manner
- The **Garbage-first (G1)** garbage collector introduced in Java 7 is made the default garbage collector in Java 9
- Adding the **Java Microbenchmark Harness (JMH)** tool that can be used to run performance benchmarks against Java applications is included as part of the Java distribution
- Support for the HTTP 2.0 and WebSocket standards by means of a new client API
- Concurrency enhancements among which is the definition of the `Flow` class, which describes an interface for the reactive streams specification in the Java platform

Some of the initial proposals that were accepted for release 9 did not make it there and were postponed for a later release, along with other interesting things that developers may expect in the future.

You can download the JDK 9 distribution for your system from <http://www.oracle.com/technetwork/java/javase/downloads/index.html>, if you are eager to get your hands dirty before trying to move through the other chapters and experimenting with the newly introduced samples and concepts.

Breaking the monolith

Over the years, the utilities of the Java platform have continued to evolve and increase, making it one big monolith. In order to make the platform more suitable for embedded and mobile devices, the publication of stripped down editions such as Java CDC and Java ME was necessary. These, however, did not prove to be flexible enough for modern applications with varying requirements in terms of functionality provided by the JDK. In that regard, the need for a modular system came in as a viral requirement, not only to address modularization of the Java utilities (overall, more than 5000 Java classes and 1500 C++ source files with more than 25,0000 lines of code for the Hotspot runtime), but also to provide a mechanism for developers to create and manage modular applications using the same module system used in the JDK. Java 8 provided an intermediate mechanism to enable applications to use only a subset of the APIs provided by the entire JDK, and that mechanism was named compact profiles. In fact, compact profiles also provided the basis for further work that had to be done in order to break dependencies between the various distinct components of the JDK required to enable implementation of a module system in Java.

The module system itself has been developed under the name of project Jigsaw on the basis of which several Java enhancement proposals and a target JSR (376) were formed. Much was put in place to address the requirements of project Jigsaw--there was evidence of concept implementation with more features proposed than the ones that successfully made it into Java 9. Apart from that, a complete restructuring of the JDK code base has been made along with a complete reorganization of the JDK distributable images.

There was considerable controversy in the community as to whether an existing and mature Java module system such as OSGi should be adopted as part of the JDK instead of providing a completely new module system. However, OSGI targets runtime behavior such as the resolution of module dependencies, installation, uninstallation, starting and stopping of modules (also named bundles in terms of OSGI), custom module classloaders, and so on. Project Jigsaw however targets a compile-time module system where resolution of dependencies happen when the application is compiled. Moreover, installing and uninstalling a module as part of the JDK eliminates the need to include it as a dependency explicitly during compilation.

Furthermore, loading of module classes is made possible through the existing hierarchy of classloaders (the bootstrap and the extension and system classloaders), although, there was a possibility of using custom module classloaders pretty much similar to the module classloaders of OSGI. The latter was, however, abandoned; we will discuss Java module classloading in more detail when we talk about the details of the module system in Java.

Additional benefits from the Java module system include enhanced security and performance. By modularizing the JDK and applications into Jigsaw modules, we are able to create well-defined boundaries between components and their corresponding domains. This separation of concerns aligns with the security architecture of the platform and is an enabler of better resource utilization. We have dedicated two detailed chapters to all of the preceding points, and to the topic of adopting Java 9 as well, which also requires a degree of understanding on the possible approaches to migrating existing projects to Java 9.

Playing around with the Java Shell

For a long time, there has been no standard shell shipped with the Java programming language to experiment with new language features or libraries or for rapid prototyping. If you wanted to do this, you could write a test application with a main method, compile it with `javac`, and run it. This could be done either at the command line or using a Java IDE; however, in both cases, this is not as convenient as having an interactive shell for the purpose.

Starting an interactive shell in JDK 9 is as simple as running the following command (assuming the `bin` directory of your JDK 9 installation is in the current path):

```
| jshell
```

You may find it somewhat puzzling that an interactive shell has not been introduced earlier in the Java platform as many programming languages, such as Python, Ruby, and a number of others, already come with an interactive shell in their earliest versions; However, this had still not made it on the priority features list for the earlier Java releases, until now, and it is out there and ready for use. The Java shell makes use of a JShell API that provides capabilities to enable autocompletion or evaluation of expressions and code snippets, among other features. A full chapter is dedicated to discussing the details of the Java shell so that developers can make the best use out of it.

Taking control of external processes

Up to JDK 9, if you wanted to create a Java process and handle process input/output, you had to use either the `Runtime.getRuntime.exec()` method, which allows us to execute a command in a separate OS process and get a `java.lang.Process` instance over which to provide certain operations in order to manage the external process, or use the new `java.lang.ProcessBuilder` class with some more enhancements in regard to interacting with the external process and also create a `java.lang.Process` instance to represent the external process. Both mechanisms were inflexible and also non-portable as the set of commands executed by the external processes were highly dependent on the operating system (additional effort had to be exerted in order to make the particular process operations portable across multiple operating systems). A chapter is dedicated to the new process API, providing developers with the knowledge of creating and managing external processes in a much easier way.

Boosting performance with G1

The G1 garbage collector was already introduced in JDK 7 and is now enabled by default in JDK 9. It is targeted for systems with multiple processing cores and a lot of available memory. What are the benefits of the G1 compared to previous types of garbage collectors? How does it achieve these improvements? Is there a need to manually tune it, and in what scenarios? These, and several more questions regarding G1, will be discussed in a separate chapter.

Measuring performance with JMH

On many occasions, Java applications may suffer from performance degradation. Exacerbating the issue is a lack of performance tests that can provide at least a minimal set of guarantees that performance requirements are met and, moreover, the performance of certain features will not degrade over time. Measuring performance of Java applications is not trivial, especially due to the fact that there is a number of compiler and runtime optimizations that may affect performance statistics. For that reason, additional measures such as warm-up phases and other tricks must be used in order to provide more accurate performance measurements. The Java Microbenchmark Harness is a framework that incorporates a number of techniques along with a convenient API that can be used for this purpose. It is not a new tool, but is included with the distribution of Java 9. If you have not added JMH to your toolbox yet, read the detailed chapter on the usage of JMH in the context of Java 9 application development.

Getting started with HTTP 2.0

HTTP 2.0 is the successor of the HTTP 1.1 protocol, and this new version of the protocol addresses some limitations and drawbacks of the previous one. HTTP 2.0 improves performance in several ways and provides capabilities such as request/response multiplexing in a single TCP connection, sending of responses in a server-push, flow control, and request prioritization, among others.

Java provides the `java.net.HttpURLConnection` utility that can be used to establish a non-secure HTTP 1.1 connection. However, the API was considered difficult to maintain and further extended with the support for HTTP 2.0 and, so, an entirely new client API was introduced in order to establish a connection via the HTTP 2.0 or the web socket protocols. The new HTTP 2.0 client, along with the capabilities it provides, will be covered in a dedicated chapter.

Encompassing reactive programming

Reactive programming is a paradigm used to describe a certain pattern for propagation of changes in a system. Reactiveness is not built in Java itself, but reactive data flows can be established using third-party libraries such as RxJava or project Reactor (part of the Spring Framework). JDK 9 also addresses the need for an API that aids the development of highly-responsive applications built around the idea of reactive streams by providing the `java.util.concurrent.Flow` class for the purpose. The `Flow` class, along with other related changes introduced in JDK 9, will be covered in a separate chapter.

Expanding the wish list

Apart from all of the new stuff in JDK 9, a whole new set of features is expected in future releases of the platform. Among these are the following:

- **Generics over primitive types:** This is one of the features planned for JDK 10 as part of project Valhalla. Other language enhancements, such as value handles, are already part of Java 9 and will be introduced later in this book.
- **Reified generics:** This is another featured part of project Valhalla that aims to provide the ability to preserve generic types at runtime. The related goals are listed as follows:
 - The foreign functional interface aims to introduce a new API to call and manage native functions. The API addresses some of the drawbacks of JNI and especially a lack of simplicity for use by application developers. The foreign functional interface is developed as part of project Panama in the JDK ecosystem.
 - New money and currency API (developed under JSR 354) was initially planned for Java 9, but was postponed.
 - New lightweight JSON API (developed under JSR 353) was also planned for Java 9, but postponed to Java 10.

These are just some of the new things one may expect in subsequent releases of the JDK. Project Penrose aims to bridge the gap between the module system in Java and the OSGi module system, and to provide different methodologies for interoperability between the two systems.

The Graal VM is another interesting research project that is a potential candidate for subsequent releases of the Java platform. It aims to bring the runtime performance of Java to dynamic languages such as JavaScript or Ruby.

A chapter dedicated to the future of JDK discusses all of these points in detail.

Summary

In this brief introductory chapter, we revealed the small universe of capabilities provided by JDK 9. The module system introduced in this release of the platform is indisputably a cornerstone in the development of Java applications. We also discovered that a number of other major features and changes are introduced in JDK 9 that deserve special attention and will be discussed in great detail in subsequent chapters.

In the next chapter, we will take a look at 26 internal changes to the Java platform.

Discovering Java 9

Java 9 represents a major release and consists of a large number of internal changes to the Java platform. Collectively, these internal changes represent a tremendous set of new possibilities for Java developers, some stemming from developer requests, others from Oracle-inspired enhancements. In this chapter, we will review 26 of the most important changes. Each change is related to a **JDK Enhancement Proposal (JEP)**. JEPs are indexed and housed at openjdk.java.net/jeps/0. You can visit this site for additional information on each JEP.



The JEP program is part of Oracle's support for open source, open innovation, and open standards. While other open source Java projects can be found, OpenJDK is the only one supported by Oracle.

In this chapter, we will cover changes to the Java platform. These changes have several impressive implications, including:

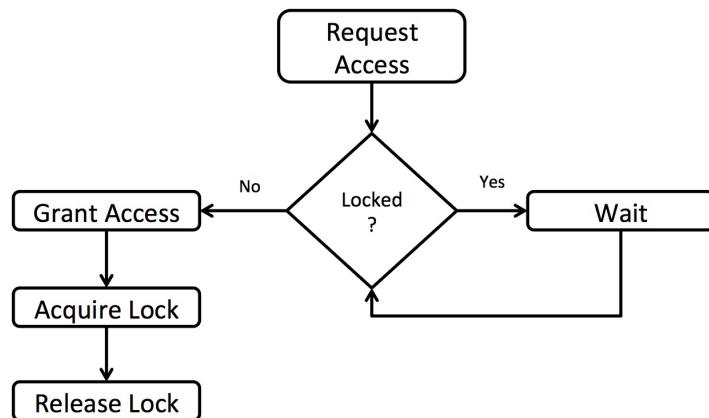
- Heap space efficiencies
- Memory allocation
- Compilation process improvements
- Type testing
- Annotations
- Automated runtime compiler tests
- Improved garbage collection

Improved Contended Locking [JEP 143]

The JVM uses Heap space for classes and objects. The JVM allocates memory on the heap whenever we create an object. This helps facilitate Java's garbage collection which releases memory previously used to hold objects that no longer have a reference to it. Java Stack memory is a bit different and is usually much smaller than heap memory.

The JVM does a good job of managing data areas that are shared by multiple threads. It associates a monitor with every object and class; these monitors have locks that are controlled by a single thread at any one time. These locks, controlled by the JVM, are, in essence, giving the controlling thread the object's monitor.

So, what is contended locking? When a thread is in a queue for a currently locked object, it is said to be in contention for that lock. The following diagram shows a high-level view of this contention:



As you can see in the preceding illustration, any threads in waiting cannot use a locked object until it is released.

Improvement goals

The general goal of JEP 143 was to increase the overall performance of how the JVM manages contention over locked Java object monitors. The improvements to contended locking were all internal to the JVM and do not require any developer actions to benefit from them. The overall improvement goals were related to faster operations. These include:

- Faster monitor enter
- Faster monitor exit
- Faster notifications

The notifications are the `notify()` and `notifyAll()` operations that are called when the locked status of an object is changed. Testing this improvement is not something you can easily accomplish. Greater efficiency, at any level, is welcome, so this improvement is one we can be thankful for even without any easily observable testing.

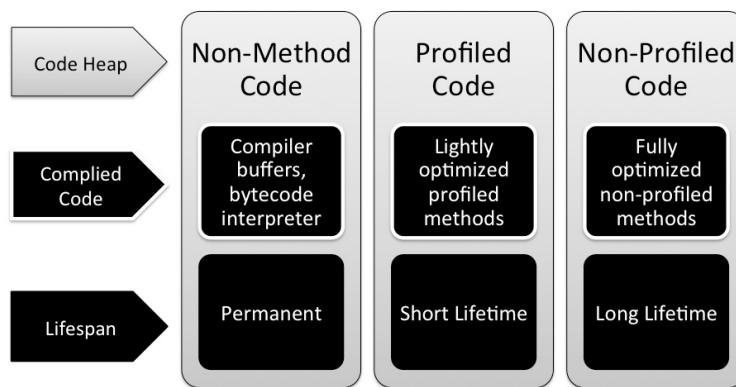
Segmented code cache [JEP 197]

The segmented code cache JEP (197) upgrade was completed and results in faster, more efficient execution time. At the core of this change was the segmentation of the code cache into three distinct segments--non-method, profiled, and non-profiled code.



A code cache is the area of memory where the Java Virtual Machine stores generated native code.

Each of the aforementioned code cache segments will hold a specific type of compiled code. As you can see in the following diagram, the code heap areas are segmented by type of compiled code:



Memory allocation

The code heap containing non-method code is for JVM internal code and consists of a 3 MB fixed memory block. The rest of the code cache memory is equally allocated for the profiled code and non-profiled code segments. You have control of this via command-line commands.

The following command can be used to define the code heap size for the non-method compiled code:

```
| -XX:NonMethodCodeCodeHeapSize
```

The following command can be used to define the code heap size for the profiled compiled methods:

```
| -XX:ProfiledCodeHeapSize
```

The following command can be used to define the code heap size for the non-profiled compiled methods:

```
| -XX:NonProfiledCodeHeapSize
```

This Java 9 feature certainly stands to improve Java application efficiency. It also impacts other processes that employ the code cache.

Smart Java compilation, phase two [JEP 199]

The JDK Enhancement Proposal 199 is aimed at improving the code compilation process. All Java developers will be familiar with the **javac** tool for compiling source code to bytecode, which is used by the JVM to run Java programs. **Smart Java Compilation**, also referred to as Smart Javac and **sjavac**, adds a *smart* wrapper around the javac process. Perhaps the core improvement sjavac adds is that only the necessary code is recompiled. Necessary code, in this context, is code that has changed since the last compile cycle.

This enhancement might not get developers excited if they only work on small projects. Consider, however, the tremendous gains in efficiency when you continuously have to recompile your code for medium and large projects. The time developers stand to save is enough reason to embrace JEP 199.

How will this change how you compile your code? It probably will not, at least not yet. Javac will remain the default compiler. While sjavac offers efficiencies regarding incremental builds, Oracle has deemed it to not have sufficient stability to become part of the standard compilation workflow.



You can read more information about the smart javac wrapper tool here: <http://cr.openjdk.java.net/~briangoetz/JDK-8030245/webrev/src/share/classes/com/sun/tools/sjavac/Main.java-.html>.

Resolving Lint and Doclint warnings [JEP 212]

Do not worry if you are not familiar with Lint or Doclint in Java. As you can determine from the section title, they are sources that report warnings to javac. Let's take a look at each one:

- **Lint** analyzes byte code and source code for javac. The goal of Lint is to identify security vulnerabilities in the code being analyzed. Lint can also provide insights into scalability and thread locking concerns. There is more to Lint, and the overall purpose is to save developers time.



You can read more about Lint here: [https://en.wikipedia.org/wiki/Lint_\(software\)](https://en.wikipedia.org/wiki/Lint_(software)).

- **Doclint** is similar to Lint and is specific to javadoc. Both Lint and Doclint report errors and warnings during the compile process. Resolution of these warnings was the focus of JEP 212. When using core libraries, there should not be any warnings. This mindset led to JEP 212, which has been resolved and implemented in Java 9.



A comprehensive list of the Lint and Doclint warnings can be reviewed in the <https://bugs.openjdk.java.net> JDK Bug System.

Tiered attribution for javac [JEP 215]

JEP 215 represents an impressive undertaking to streamline javac's type checking schema. Let's first review how type checking works in Java 8; then we will explore the changes in Java 9.

In Java 8, type checking of poly expressions is handled by a **speculative attribution** tool.



Speculative attribution is a method of type checking as part of javac's compilation process. It has a significant processing overhead.

Using the speculative attribution approach to type checking is accurate, but lacks efficiency. These checks include argument position, and are exponentially slower when testing in the midst of recursion, polymorphism, nested loops, and lambda expressions. So the goal with JEP 215 was to change the type checking schema to create faster results. The results themselves were not inaccurate with speculative attribution; they were just not generated rapidly.

The new approach, released with Java 9, uses a tiered attribution tool. This tool implements a tiered approach for type checking argument expressions for all method calls. Permissions are also made for method overriding. In order for this new schema to work, new structural types are created for each of the following listed types of method arguments:

- Lambda expressions
- Poly expressions
- Regular method calls
- Method references
- Diamond instance creation expressions

The changes to javac from JEP 215 are more complex than what has been highlighted in this section. There is no immediate impact to developers other than a more efficient javac and time saved.

Annotations pipeline 2.0 [JEP 217]

Java annotations refer to a special kind of metadata that resides inside your Java source code files. They are not stripped by javac, so that they can remain available to the JVM at runtime.

Annotations look similar to JavaDocs references because they start with the @ symbol. There are three types of annotations. Let's examine each:

- The most basic form of annotation is a **marker** annotation. These are standalone annotations with the only component being the name of the annotation. Here is an example:

```
@thisIsAMarkerAnnotation
public double computeSometing(double x, double y)
{
    // do something and return a double
}
```

- The second type of annotation is one that contains a *single value*, or piece of data. As you can see in the following code, the annotation, which starts with the @ symbol, is followed by parentheses containing data:

```
@thisIsAMarkerAnnotation (data="compute x and y
                           coordinates")
public double computeSometing(double x, double y)
{
    // do something and return a double
}
```

An alternative way of coding the single value annotation type is to omit the data= component, as illustrated in the following code:

```
@thisIsAMarkerAnnotation ("compute x and y coordinates")
public double computeSometing(double x, double y)
{
    // do something and return a double
}
```

- The third type of annotation is when there is *more than one data component*. With this type of annotation, the data= component cannot be omitted. Here is an example:

```
@thisIsAMarkerAnnotation (data="compute x and y
```

```
coordinates", purpose="determine intersecting point")
public double computeSometing(double x, double y)
{
    // do something and return a double
}
```

So, what has changed in Java 9? To answer this question, we need to recall a couple of changes introduced with Java 8 that impacted Java annotations:

- Lambda expressions
- Repeated annotations
- Java type annotations

These Java 8 related changes impacted Java annotations, but did not usher in a change to how javac processed them. There were some hardcoded solutions that allowed javac to handle the new annotations, but they were not efficient. Moreover, this type of coding (hardcoding workarounds) is difficult to maintain.

So, JEP 217 focused on refactoring the javac annotation pipeline. This refactoring was all internal to javac, so it should not be evident to developers.

New version-string scheme [JEP 223]

Prior to Java 9, the release numbers did not follow industry standard versioning-- **semantic versioning**. For example, at the time of writing, the last four JDK releases were:

- JDK 8 update 131
- JDK 8 update 121
- JDK 8 update 112



Semantic versioning uses a major, minor, patch (0.0.0) schema:

Major equates to new API changes that are not backwards compatible.

Minor is when functionality is added that is backwards compatible.

Patch refers to bug fixes or minor changes that are backwards compatible.

Oracle has embraced semantic versioning for Java 9 and beyond. For Java, a **major-minor-security** schema will be used for the first three elements of Java version numbers:

- **Major**: A major release consisting of a significant new set of features
- **Minor**: Revisions and bug fixes that are backwards compatible
- **Security**: Fixes deemed critical to improve security

This description of JEP 223 might make the versioning schema seem basic. To the contrary, a very detailed set of rules and practices have been developed to manage the future version numbers. To demonstrate the complexity, see the following example:

| 1.9.0._32.b19

Generating run-time compiler tests automatically [JEP 233]

Java is arguably the most used programming language and resides on an increasingly diverse number of platforms. This exacerbates the problem of running targeted compiler tests in an efficient manner. The purpose of JEP 233 was to create a tool that could automate the runtime compiler tests.

The tool that was created starts by generating a random set of Java source code and/or byte code. The generated code will have three key characteristics:

- Be syntactically correct
- Be semantically correct
- Use a random seed that permits reusing the same randomly-generated code

The source code that is randomly generated will be saved in the following directory:

```
|   hotspot/test/testlibrary/jit-tester
```

These test cases will be stored for later re-use. They can be run from the `j-treg` directory or from the tool's makefile. One of the benefits of re-running saved tests is to test the stability of your system.

Testing class-file attributes generated by Javac [JEP 235]

The lack of, or insufficient, capability to create tests for class-file attributes was the impetus behind JEP 235. The goal is to ensure javac creates a class-file's attributes completely and correctly. This suggests that even if some attributes are not used by the class-file, all class-files should be generated with a complete set of attributes. There also needs to be a way of testing that the class-files were created correctly, in regards to the file's attributes.

Prior to Java 9, there was no method of testing a class-file's attributes. Running a class and testing the code for anticipated or expected results was the most commonly used method of testing javac generated class-files. This technique falls short of testing to validate the file's attributes.

There are three categories of class-file attributes--attributes used by the JVM, optional attributes, and attributes not used by the JVM.

Attributes used by the JVM include:

- BootstrapMethods
- Code
- ConstantValue
- Exceptions
- StackMapTable

Optional attributes include:

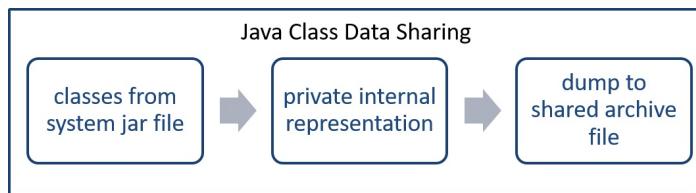
- Deprecated
- LineNumberTable
- LocalVariableTable
- LocalVariableTypeTable
- SourceDebugExtension
- SourceFile

Attributes not used by the JVM include:

- AnnotationDefault
- EnclosingMethod
- InnerClasses
- MethodParameters
- RuntimeInvisibleAnnotations
- RuntimeInvisibleParameterAnnotations
- RuntimeInvisibleTypeAnnotations
- RuntimeVisibleAnnotations
- RuntimeVisibleParameterAnnotations
- RuntimeVisibleTypeAnnotations
- Signature
- Synthetic

Storing interned strings in CDS archives [JEP 250]

The method in which strings are stored and accessed to and from **Class Data Sharing (CDS)** archives is inefficient, excessively time consuming, and wastes memory. The following diagram illustrates the method in which Java stores interned strings in a CDS archive:



The inefficiency stems from the current storage schema. Especially when the **Class Data Sharing** tool dumps the classes into the shared archive file, the constant pools containing `CONSTANT_String` items have a UTF-8 string representation.

UTF-8 is an 8-bit variable-length character encoding standard.



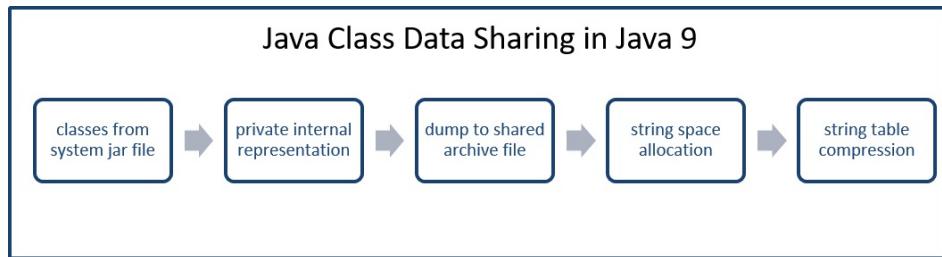
The problem

With the current use of UTF-8, the strings must be converted to string objects, instances of the `java.lang.String` class. This conversion takes place on-demand which can result in slower systems and unnecessary memory usage. The processing time is extremely short, but the memory usage cannot be overlooked. Every character in an interned string requires at least 3 bytes of memory and potentially more.

A related problem is that the stored strings are not accessible to all JVM processes.

The solution

CDS archives now allocate specific space on the heap for strings:



The string space is mapped using a shared-string table, hash tables, and deduplication.



Deduplication is a data compression technique that eliminates duplicative information in an archive.

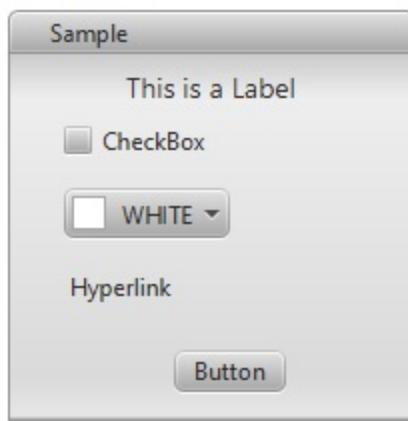
Preparing JavaFX UI controls and CSS APIs for modularization [JEP 253]

JavaFX is a set of packages that permits the design and development of media-rich graphical user interfaces. JavaFX applications provide developers with a great API for creating a consistent interface for applications. **Cascading Style Sheets (CSS)** can be used to customize the interfaces. One of the great things about JavaFX is that the tasks of programming and interface design can easily be separated.

JavaFX overview

There is a wonderful visual scripting tool called Scene Builder that allows you to create graphical user interfaces by using drag and drop and property settings. Scene Builder generates the necessary FXML files that are used by your **Integrated Development Environment (IDE)** such as NetBeans.

Here is a sample UI created with Scene Builder:



And, here is the FXML file created by Scene Builder:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.paint.*?>
<?import javafx.scene.text.*?>

<AnchorPane id="AnchorPane" maxHeight="-Infinity"
    maxWidth="-Infinity" minHeight="-Infinity"
    minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0"
    xmlns:fx="http://javafx.com/fxml/1"
    xmlns="http://javafx.com/javafx/2.2">
    <children>
        <TitledPane animated="false" collapsible="false"
            layoutX="108.0" layoutY="49.0" text="Sample">
            <content>
                <AnchorPane id="Content" minHeight="0.0" minWidth="0.0"
                    prefHeight="180.0" prefWidth="200.0">
                    <children>
                        <CheckBox layoutX="26.0" layoutY="33.0"
                            mnemonicParsing="false" prefWidth="94.0"
                            text="CheckBox" />
                        <ColorPicker layoutX="26.0" layoutY="65.0" />
                    </children>
                </AnchorPane>
            </content>
        </TitledPane>
    </children>
</AnchorPane>
```

```
<Hyperlink layoutX="26.0" layoutY="103.0"
    text="Hyperlink" />
<Label alignment="CENTER" layoutX="14.0" layoutY="5.0"
    prefWidth="172.0" text="This is a Label"
    textAlignment="CENTER">
    <font>
        <Font size="14.0" />
    </font>
</Label>
<Button layoutX="81.0" layoutY="146.0"
    mnemonicParsing="false" text="Button" />
</children>
</AnchorPane>
</content>
</TitledPane>
</children>
</AnchorPane>
```

Implications for Java 9

Prior to Java 9, JavaFX controls as well as CSS functionality were only available to developers by interfacing with internal APIs. Java 9's modularization has made the internal APIs inaccessible. Therefore, JEP 253 was created to define public, instead of internal, APIs.

This was a larger undertaking than it might seem. Here are a few actions that were taken as part of this JEP:

- Moving javaFX control skins from the internal to public API (`javafx.scene.skin`)
- Ensuring API consistencies
- Generation of a thorough javadoc

The following classes were moved from internal packages to a public `javafx.scene.control.skin` package:

AccordionSkin	ButtonBarSkin	ButtonSkin	CellSkinBase
CheckBoxSkin	ChoiceBoxSkin	ColorPickerSkin	ComboBoxBaseSkin
ComboBoxListViewSkin	ComboBoxPopupControl	ContextMenuSkin	DateCellSkin
DatePickerSkin	HyperLinkSkin	LabelSkin	LabeledSkinBase
ListCellSkin	ListViewSkin	MenuBarSkin	MenuButtonSkin
MenuButtonSkinbase	NestedTableColumHeader	PaginationSkin	ProgressbarSkin
ProgressIndicatorSkin	RadioButtonSkin	ScrollBarSkin	ScrollPaneSkin
SeparatorSkin	SliderSkin	SpinnerSkin	SplitMenuItem
SplitPaneSkin	TabPaneSkin	TableCellSkin	TableCellSkinBase
TableColumnHeader	TableHeaderRow	TableHeaderSkin	TableRowSkinBase
TableViewSkin	TableViewSkinBase	TextAreaSkin	TextFieldSkin
TextInputControlSkin	TitledPaneSkin	ToggleButtonSkin	ToolBarSkin
TooltipSkin	TreeCellSkin	TreeTableCellSkin	TreeTableRowSkin
TreeTableViewSkin	TreeViewSkin	VirtualContainerBase	VirtualFlow

The public `javafx.css` package now has the additional classes:

- CascadingStyle.java:public class CascadingStyle implements Comparable<CascadingStyle>
- CompoundSelector.java:final public class CompoundSelector extends Selector
- CssError.java:public class CssError
- Declaration.java:final public class Declaration
- Rule.java:final public class Rule
- Selector.java:abstract public class Selector
- SimpleSelector.java:final public class SimpleSelector extends Selector
- Size.java:final public class Size
- Style.java:final public class Style
- Stylesheet.java:public class Stylesheet
- CssParser.java:final public class CssParser

Compact strings [JEP 254]

The string data type is an important part of nearly every Java app. While JEP 254's aim was to make strings more space-efficient, it was approached with caution so that existing performance and compatibilities would not be negatively impacted.

Pre-Java 9 status

Prior to Java 9, string data was stored as an array of chars. This required 16 bits for each char. It was determined that the majority of String objects could be stored with only 8 bits, or 1 byte of storage. This is due to the fact that most strings consist of Latin-1 characters.



The ISO Latin-1 Character Set is a single-byte set of character's encodings.

New with Java 9

Starting with Java 9, strings are now internally represented using a byte array along with a flag field for encoding references.

Merging selected Xerces 2.11.0 updates into JAXP [JEP 255]

Xerces is a library used for parsing XML in Java. It was updated to 2.11.0 in late 2010, so JEP 255's aim was to update JAXP to incorporate changes in Xerces 2.11.0.



JAXP is Java's API for XML processing.

Prior to Java 9, the JDK's latest update regarding XML processing was based on Xerces 2.7.1. There were some additional changes to JDK 7 based on Xerces, 2.10.0. JEP 255 is a further refinement of the JAXP based on Xerces 2.11.0.

Xerces 2.11.0 supports the following standards:

- XML 1.0, Fourth Edition
- Namespaces in XML 1.0, Second Edition
- XML 1.1, Second Edition
- Namespaces in XML 1.1, Second Edition
- XML Inclusions 1.0, Second Edition
- **Document Object Model (DOM)**
 - Level 3
 - Core
 - Load & save
 - Level 2
 - Core
 - Events
- Traversal & Range
- Element Traversal, First Edition
- Simple API for XML 2.0.2
- **Java APIs for XML Processing (JAXP) 1.4**
- Streaming API for XML 1.0
- XML Schema 1.0
- XML Schema 1.1
- XML Schema Definition Language

The JDK was updated to include the following Xerces 2.11.0 categories:

- Catalog resolver
- Datatypes
- Document Object Model Level 3
- XML Schema Validation
- XPointer

The public API for JAXP was not changed in Java 9.

Updating JavaFX/Media to newer version of GStreamer [JEP 257]

JavaFX is used for creating desktop and web applications. JavaFX was created to replace Swing as Java's standard GUI library. The `Media` class, `javafx.scene.media.Media`, is used to instantiate an object representing a media resource. JavaFX/Media refers to the following class:

```
| public final class Media extends java.lang.Object
```

This class provides referential data to a media resource. The `javafx.scene.media` package provides developers with the ability to incorporate media into their JavaFX applications. JavaFX/Media utilizes a GStreamer pipeline.



GStreamer is a multimedia processing framework that can be used to build systems that take in media from several different formats and, after processing, export them in selected formats.

The purpose of JEP 257 was to ensure JavaFX/Media was updated to include the latest release of GStreamer for stability, performance, and security assurances.

HarfBuzz Font-Layout Engine [JEP 258]

Prior to Java 9, the layout engine used to handle font complexities; specifically fonts that have rendering behaviors beyond what the common Latin fonts have. Java used the uniform client interface, also referred to as ICU, as the defacto text rendering tool. The ICU layout engine has been deprecated and, in Java 9, has been replaced with the HarfBuzz font layout engine.

HarfBuzz is an **OpenType** text rendering engine. This type of layout engine has the characteristic of providing script-aware code to help ensure text is laid out as desired.



OpenType is an HTML formatted font format specification.

The impetus for the change from the ICU Layout Engine to the HarfBuzz Font Layout Engine was IBM's decision to cease supporting the ICU Layout Engine. Therefore, the JDK was updated to contain the HarfBuzz Font Layout Engine.

HiDPI graphics on Windows and Linux [JEP 263]

JEP 263 was focused on ensuring the crispness of on-screen components, relative to the pixel density of the display. The following terms are relevant to this JEP and are provided along with the below listed descriptive information:

- **DPI-aware application:** An application that is able to detect and scale images for the display's specific pixel density
- **DPI-unaware application:** An application that makes no attempt to detect and scale images for the display's specific pixel density
- **HiDPI graphics:** High dots-per-inch graphics
- **Retina display:** This term was created by Apple to refer to displays with a pixel density of at least 300 pixels per inch

Displaying graphics, both images and graphical user interface components, to the user is typically of paramount performance. Displaying this imagery in high quality can be somewhat problematic. There is large variability in computer monitor DPIs. There are three basic approaches to developing for displays:

- Develop apps without regard for the potential different display dimensions. In other words, create a DPI-unaware application.
- Develop a DPI-aware application that selectively uses pre-rendered image sizes for a given display.
- Develop a DPI-aware application that properly scales images up/down to account for the specific display the application is run on.

Clearly, the first two approaches are problematic, and for different reasons. With the first approach, the user experience is not considered. Of course, if the application was being developed for a very specific display with no expected pixel density variability, then this approach could be viable.

The second approach requires a lot of work on the design and development end to ensure images for each expected display density are created and implemented programmatically. In addition to the tremendous amount of work, the app size will unnecessarily increase, and new and different pixel densities will not have been

accounted for.

The third approach is to create a DPI-aware application with efficient and effective scaling capabilities. This approach works well and has been proven with the Mac retina displays.

Prior to Java 9, automatic scaling and sizing was already implemented in Java for the Mac OS X operating system. This capability was added in Java 9 for Windows and Linux operating systems.

Marlin graphics renderer [JEP 265]

JEP 265 replaced the Pisces graphics rasterizer with the Marlin graphics renderer in the Java 2D API. This API is used to draw 2D graphics and animations.

The goal was to replace Pisces with a rasterizer/renderer that was much more efficient and without any quality loss. This goal was realized in Java 9. An intended collateral benefit was to include a developer-accessible API. Previously, the means of interfacing with the AWT and Java 2D was internal.

Unicode 8.0.0 [JEP 267]

Unicode 8.0.0 was released on June 17, 2015. JEP 267 focused on updating the relevant APIs to support Unicode 8.0.0.

New in Unicode 8.0.0

Unicode 8.0.0 added nearly 8,000 characters. Here are the highlights of the release:

- Ahom script for the Tai Ahom language (India)
- Arwi, Tamil language (Arabic)
- Cherokee symbols
- CJK unified ideographs
- Emoji symbols along with flesh-tone symbol modifiers
- Georgian lari currency symbol
- lk language (Uganda)
- Kulango language (Côte d'Ivoire)

Updated Classes in Java 9

In order to fully comply with the new Unicode standard, several Java classes were updated. The following listed classes were updated for Java 9 to comply with the new Unicode standard:

- `java.awt.font.NumericShaper`
- `java.lang.Character`
- `java.lang.String`
- `java.text.Bidi`
- `java.text.BreakIterator`
- `java.text.Normalizer`

Reserved stack areas for critical sections [JEP 270]

The goal of JEP 270 was to mitigate problems stemming from stack overflows during the execution of critical sections. This mitigation took the form of reserving additional thread stack space.

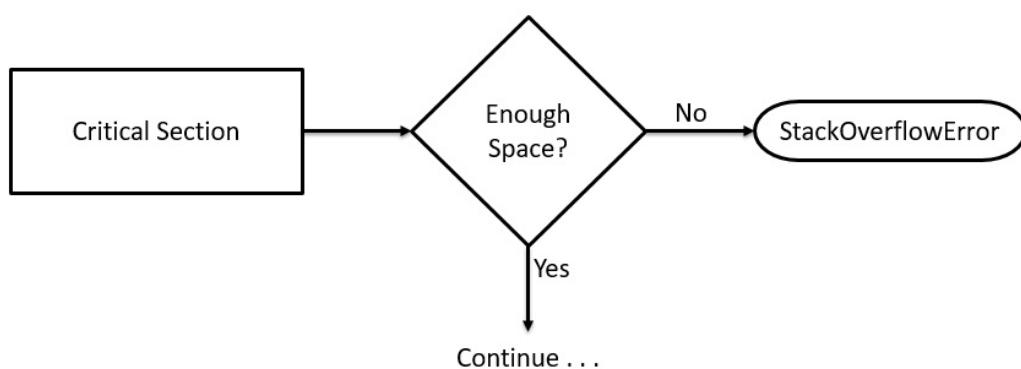
The pre-Java 9 situation

The JVM throws a `StackOverflowError` when it is asked to perform data computation in a thread that has insufficient stack space and does not have permission to allocate additional space. This is an asynchronous exception. The JVM can also throw the `StackOverflowError` exception synchronously when a method is invoked.

When a method is invoked, an internal process is used to report the Stack Overflow. While the current schema works sufficiently for reporting the error, there is no room for the calling application to easily recover from the error. This can result in being more than a nuisance for developers and users. If the `StackOverflowError` was thrown during a critical computational operation, the data might be corrupted, causing additional problems.

While not the sole cause of these problems, the effected status of locks from the `ReentrantLock` class were a common cause of undesirable outcomes. This issue was evident in Java 7 because the `ConcurrentHashMap` code implemented the `ReentrantLock` class. The `ConcurrentHashMap` code was modified for Java 8, but problems still persisted for any implementation of the `ReentrantLock` class. Similar problems existed beyond just `ReentrantLock` class usage.

The following diagram provides a broad overview of the `StackOverflowError` problem:



In the next section, we will look at how this issue was resolved for Java 9.

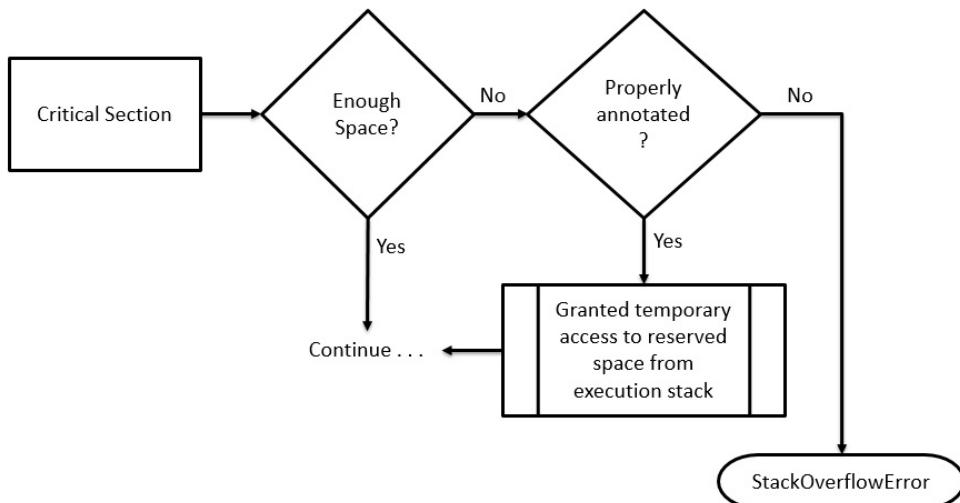
New in Java 9

With the JEP 270 changes for Java 9, a critical section will automatically be given additional space so that it can complete its execution and not suffer from the `StackOverflowError`. This is predicated on the additional space allocation needs being small. The necessary changes have been made to the JVM to permit this functionality.

The JVM actually delays the `StackOverflowError`, or at least attempts to, while critical sections are executing. In order to capitalize on this new schema, methods must be annotated with the following:

```
|     jdk.internal.vm.annotation.ReservedStackAccess
```

When a method has this annotation and a `StackOverflowError` condition exists, temporary access to the reserved memory space is granted. The new process is, at a high level of abstraction, presented as follows:



Dynamic linking of language-defined object models [JEP 276]

Java interoperability was enhanced with JEP 276. The necessary JDK changes were made to permit runtime linkers from multiple languages to coexist in a single JVM instance. This change applies to high-level operations, as you would expect. An example of a relevant high-level operation is the reading or writing of a property with elements such as accessors and mutators.

The high-level operations apply to objects of unknown types. They can be invoked with `INVOKEDYNAMIC` instructions. Here is an example of calling an object's property when the object's type is unknown at compile time:

```
| INVOKEDYNAMIC "dyn:getProp:age"
```

Proof of concept

Nashorn is a lightweight, high-performance, JavaScript runtime that permits embedding JavaScript in Java applications. This was created for Java 8 and replaced the previous JavaScript scripting engine that was based on Mozilla Rhino. Nashorn already has this functionality. It provides linkage between high-level operations on any object of unknown type, such as `obj.something`, where it produces the following:

```
| INVOKEDYNAMIC "dyn.getProp.something"
```

The dynamic linker springs into action and provides, when possible, the appropriate implementation.

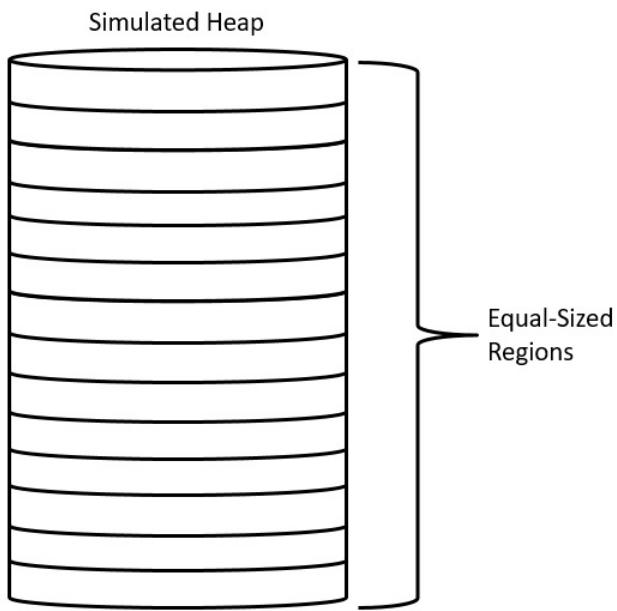
Additional tests for humongous objects in G1 [JEP 278]

One of the long-favored features of the Java platform is the behind the scenes garbage collection. JEP 278's focus was to create additional WhiteBox tests for humongous objects as a feature of the G1 garbage collector.



WhiteBox testing is an API used to query JVM internals. The WhiteBox testing API was introduced in Java 7 and upgraded in Java 8 and Java 9.

The G1 garbage collector works extremely well, but there was room for some improved efficiency. The way the G1 garbage collector worked is based on first dividing the heap into regions of equal size, illustrated as follows:



The problem with the G1 garbage collector was how humongous objects were handled.



A humongous object in the context of garbage collection, is any object that takes up more than one region on the heap.

The problem with humongous objects was that if they took up any part of a region on the heap, the remaining space was not able to be allocated for other objects. In Java 9, the WhiteBox API was extended with four types of new methods:

- Methods with the purpose of blocking full garbage collection and to initiate concurrent marking.
- Methods that can access individual G1 garbage collection heap regions. Access to these regions consist of attribute reading, such as with the current state of the region.
- Methods with direct access to the G1 garbage collection internal variables.
- Methods that can determine if humongous objects reside on the heap and, if so, in what regions.

Improving test-failure troubleshooting [JEP 279]

For developers that do a lot of testing, JEP 279 is worth reading about. Additional functionality has been added in Java 9 to automatically collect information to support troubleshooting test failures as well as timeouts. Collecting readily available diagnostic information during tests stands to provide developers and engineers with greater fidelity in their logs and other output.

There are two basic types of information in the context of testing--environmental and process.

Environmental information

When running tests, the testing environment information can be important for troubleshooting efforts. This information includes the following:

- CPU loads
- Disk space
- I/O loads
- Memory space
- Open files
- Open sockets
- Processes running
- System events
- System messages

Java process information

There is also information available during the testing process directly related to Java processes. These include:

- C stacks
- Core dumps
- Mini dumps
- Heap statistics
- Java stacks



For additional information on this concept, read about the JDK's regression test harness (jtreg).

Optimizing string concatenation

[JEP 280]

JEP 280 is an interesting enhancement for the Java platform. Prior to Java 9, string concatenation was translated by javac into `StringBuilder` `:: append` chains. This was a sub-optimal translation methodology often requiring `StringBuilder` presizing.

The enhancement changed the string concatenation bytecode sequence, generated by javac, so that it uses `INVOKEDYNAMIC` calls. The purpose of the enhancement was to increase optimization and to support future optimizations without the need to reformat the javac's bytecode.



See JEP 276 for more information on `INVOKEDYNAMIC`.

The use of `INVOKEDYNAMIC` calls to `java.lang.invoke.StringConcatFactory` allows us to use a methodology similar to lambda expressions instead of using `StringBuilder`'s step-wise process. This results in more efficient processing of string concatenation.

HotSpot C++ unit-test framework

[JEP 281]

HotSpot is the name of the JVM. This Java enhancement was intended to support the development of C++ unit tests for the JVM. Here is a partial, non-prioritized, list of goals for this enhancement:

- Command-line testing
- Create appropriate documentation
- Debug compile targets
- Framework elasticity
- IDE support
- Individual and isolated unit testing
- Individualized test results
- Integrate with existing infrastructure
- Internal test support
- Positive and negative testing
- Short execution time testing
- Support all JDK 9 build platforms
- Test compile targets
- Test exclusion
- Test grouping
- Testing that requires the JVM to be initialized
- Tests co-located with source code
- Tests for platform-dependent code
- Write and execute unit testing (for classes and methods)

This enhancement is evidence of the increasing extensibility.

Enabling GTK 3 on Linux [JEP 283]

GTK+, formally known as the GIMP toolbox, is a cross-platform tool used for creating **Graphical User Interfaces (GUI)**. The tool consists of widgets accessible through its API. JEP 283's focus was to ensure GTK 2 and GTK 3 were supported on Linux when developing Java applications with graphical components. The implementation supports Java apps that employ JavaFX, AWT, and Swing.

We can create Java graphical applications with JavaFX, AWT, and Swing. Here is a table to summarize those three approaches as they relate to GTK, prior to Java 9:

Approach	Remarks
JavaFX	<ul style="list-style-type: none">• Uses a dynamic GTK function lookup• Interacts with AWT and Swing via JFXPanel• Uses AWT printing functionality
AWT	<ul style="list-style-type: none">• Uses a dynamic GTK function lookup
Swing	<ul style="list-style-type: none">• Uses a dynamic GTK function lookup

So, what changes were necessary to implement this JEP? For JavaFX, three specific things were changed:

- Automated testing was added for both GTK 2 and GTK 3
- Functionality was added to dynamically load GTK 2
- Support was added for GTK 3

For AWT and Swing, the following changes were implemented:

- Automated testing was added for both GTK 2 and GTK 3
- `AwtRobot` was migrated to GTK 3

- `FileChooserDialog` was updated for GTK 3
- Functionality was added to dynamically load GTK 3
- The Swing GTK LnF was modified to support GTK 3



Swing GTK LnF is short for Swing GTK look and feel.

New HotSpot build system [JEP 284]

The Java platform used, prior to Java 9, was a build system riddled with duplicate code, redundancies, and other inefficiencies. The build system has been reworked for Java 9 based on the build-infra framework. In this context, infra is short for infrastructure. The overarching goal for JEP 284 was to upgrade the build system to one that was simplified. Specific goals included:

- Leverage existing build system
- Maintainable code
- Minimize duplicate code
- Simplification
- Support future enhancements

You can learn more about Oracle's infrastructure framework at this site: <http://www.oracle.com/technetwork/oem/frmwrk-infra-496656.html>

Summary

In this chapter, we covered some impressive new features of the Java platform, with specific focus on javac, JDK libraries, and various test suites. Memory management improvements, including heap space efficiencies, memory allocation, and improved garbage collection represent a powerful new set of Java platform enhancements. Changes regarding the compilation process resulting in greater efficiencies were part of our chapter. We also covered important improvements, such as with the compilation process, type testing, annotations, and automated runtime compiler tests.

In the next chapter, we will look at several minor language enhancements introduced in Java 9.

Java 9 Language Enhancements

In the previous chapter, we gained insight into some exciting new features contained in Java 9. Our focus was on javac, the JDK libraries, and test suites. We learned about memory management improvements including memory allocation, heap optimizations, and enhanced garbage collection. We also covered changes to the compilation process, type testing, annotations, and runtime compiler tests.

This chapter covers some changes in Java 9 that impact variable handlers, depreciation warnings, improvements on Project Coin changes implemented in Java 7, and import statement processing. These represent changes to the Java language itself.

The topics we will cover here are:

- Variable handlers
- Import statement depreciation warnings
- Project Coin
- Import statement processing

Working with variable handlers

[JEP 193]

Variable handlers are typed references to variables and are governed by the `java.lang.invoke.VarHandle` abstract class. The `VarHandle` method's signature is polymorphic. This provides for great variability in both method signatures and return types. Here is a code sample demonstrating how a `VarHandle` might be used:

```
 . . .

class Example
{
    int myInt;
    . . .
}

. . .
class Sample
{
    static final VarHandle VH_MYINT;

    static
    {
        try
        {
            VH_MYINT =
                MethodHandles.lookup().in(Example.class)
                    .findVarHandle(Example.class, "myInt", int.class);
        }
        catch (Exception e)
        {
            throw new Error(e);
        }
    }
}

. . .
```

As you can see in the preceding code snippet, the `VarHandle.lookup()` performs the same operation as those that are performed by a `MethodHandle.lookup()` method.

The aim of this JEP was to standardize the way in which methods of the following classes are invoked:

- `java.util.concurrent.atomic`
- `sun.misc.Unsafe`

Specifically, methods that:

- accessed/mutated object fields
- accessed/mutated elements of an array

In addition, this JEP resulted in two fence operations for memory ordering and object reachability. In the spirit of due diligence, special attention was given to ensure the JVM's safety. It was important to ensure that memory errors did not result from these changes. Data integrity, usability, and, of course, performance were key components of the aforementioned due diligence and are explained as follows:

- **Safety:** Corrupt memory states must not be possible.
- **Data integrity:** Ensure access to an object's field uses identical rules used by:
 - `getfield` byte code
 - `putfield` byte code
- **Usability:** The benchmark for usability was the `sun.misc.Unsafe` API. The goal was to make the new API easier to use than the benchmark.
- **Performance:** There could be no degradation of performance compared to the use of the `sun.misc.Unsafe` API. The goal was to outperform that API.



In Java, a fence operation is what javac does to force a constraint on memory in the form of a barrier instruction. These operations occur before and after the barrier instruction, essentially fencing them in.

Working with the AtoMiC Toolkit

The `java.util.concurrent.atomic` package is a collection of 12 sub-classes that support operations on single variables that are thread-safe and lock-free. In this context, thread-safe refers to code that accesses or mutates a shared single variable without impeding on other threads executing on the variable at the same time. This superclass was introduced in Java 7.

Here is a list of the 12 sub-classes in the AtoMiC Toolkit. The class names, as you would expect, are self-descriptive:

Atomic subclass
<code>java.util.concurrent.atomic.AtomicBoolean</code>
<code>java.util.concurrent.atomic.AtomicInteger</code>
<code>java.util.concurrent.atomic.AtomicIntegerArray</code>
<code>java.util.concurrent.atomic.AtomicIntegerFieldUpdater<T></code>
<code>java.util.concurrent.atomic.AtomicLong</code>
<code>java.util.concurrent.atomic.AtomicLongArray</code>
<code>java.util.concurrent.atomic.AtomicLongFieldUpdater<T></code>
<code>java.util.concurrent.atomic.AtomicMarkableReference<V></code>
<code>java.util.concurrent.atomic.AtomicReference<V></code>
<code>java.util.concurrent.atomic.AtomicReferenceArray<E></code>
<code>java.util.concurrent.atomic.AtomicReferenceFieldUpdater<T,V></code>
<code>java.util.concurrent.atomic.AtomicStampedReference<V></code>

Volatile variables, fields, and array elements can be asynchronously modified by concurrent threads.



In Java, the `volatile` keyword is used to inform the `javac` utility to read the value, field, or array element from the main memory and not to cache them.

Here is a code snippet that demonstrates the use of the `volatile` keyword for an

instance variable:

```
public class Sample
{
    private static volatile Sample myVolatileVariable; // a
    volatile instance variable

    public static Sample getVariable() // getter method
    {
        if (myVolatileVariable != null)
        {
            return myVolatileVariable;
        }
        // this section executes if myVolatileVariable == null
        synchronized(Sample.class)
        {
            if (myVolatileVariable == null)
            {
                myVolatileVariable = new Sample();
            }
        }
    }
}
```

Using the sun.misc.Unsafe class

The `sun.misc.Unsafe` class, like other `sun` classes, is not officially documented or supported. It has been used to circumvent some of Java's built-in memory management safety features. While this can be viewed as a window to greater control and flexibility in our code, it is a terrible programming practice.

The class had a single private constructor, so an instance of the class could not easily be instantiated. So, if we tried to instantiate an instance with `myUnsafe = new Unsafe()`, a `SecurityException` would be thrown in most circumstances. This somewhat unreachable class has over 100 methods that permitted operations on arrays, classes, and objects. Here is a brief sampling of those methods:

Arrays	Classes	Objects
arrayBaseOffset	defineAnonymousClass	allocateInstance
arrayIndexScale	defineClass	objectFieldOffset
	ensureClassInitialized	
	staticFieldOffset	

Here is a secondary grouping of the `sun.misc.Unsafe` class method for information, memory, and synchronization:

Information	Memory	Synchronization
addressSize	allocateMemory	compareAndSwapInt
pageSize	copyMemory	monitorEnter
	freeMemory	monitorExit
	getAddress	putOrderedEdit
	getInt	tryMonitorEnter
	putInt	

The `sun.misc.Unsafe` class was earmarked for removal in Java 9. There was actually some opposition to this decision in the programming industry. To put their concerns to rest, the class has been deprecated, but will not be completely removed. A special

flag can be sent to the JVM to utilize the original API.

Eliding depreciation warnings on import statements [JEP 211]

This is one of the more simplistic JEPs for Java 9. Quite often, when we compile our programs, we receive many warnings and errors. The compiler errors must be fixed as they are typically syntactical in nature. The warnings, on the other hand, should be reviewed and appropriately addressed. Some of the warning messages are ignored by developers.

This JEP provides slight relief in the number of warnings we receive. Specifically, depreciation warnings caused by import statements are no longer generated. Prior to Java 9, we could suppress deprecated warning messages with the following annotation:

```
|     @SuppressWarnings
```

Now, with Java 9, the compiler will suppress depreciated warnings if one or more of the following cases is true:

- If the `@Deprecated` annotation is used
- If the `@SuppressWarnings` annotation is used
- If the use of the warning-generating code and the declaration are within the ancestor class
- If the use of the warning-generating code is within an import statement

The fourth condition listed was an addition in Java 9.

Milling Project Coin [JEP 213]

Project Coin was a feature set of minor changes introduced in Java 7. These changes are listed as follows:

- Strings in `switch` statements
- Binary integral literals
- Using underscores in numeric literals
- Implementing multi-catch
- Allowing for more precise re-throwing of exceptions
- Generic instance creation improvements
- Addition of the `try-with-resources` statement
- Improvements to invoking `varargs` methods

Detailed information can be found in the following Oracle presentation: <http://www.oracle.com/us/technologies/java/project-coin-428201.pdf>.

JEP 213 focused on improvements to Project Coin's enhancements. There were five such enhancements, each detailed as follows.

Using the `@SafeVarargs` annotation

In Java 9, we can use the `@SafeVarargs` annotation with private instance methods.

When we use this annotation, we are asserting that the method does not contain any harmful operations on the `varargs` passed as parameters to the method.

The syntax for usage is:

```
@SafeVarargs // this is the annotation
static void methodName(...)
{
    /*
    The contents of the method or constructor must not
    perform any unsafe or potentially unsafe operations
    on the varargs parameter or parameters.
    */
}
```

Use of the `@SafeVarargs` annotation is restricted to:

- Static methods
- Final instance methods
- Private instance methods

The try-with-resource statement

The `try-with-resource` statement previously required a new variable to be declared for each resource in the statement when a final variable was used. Here is the syntax for the `try-with-resource` statement prior to Java 9 (in Java 7 or 8):

```
try ( // open resources )
{
    // use resources
} catch ( // error)
{
    // handle exceptions
}
// automatically close resources
```

Here is a code snippet using the preceding syntax:

```
try ( Scanner xmlScanner = new Scanner(new File(xmlFile)) ;
{
    while (xmlScanner.hasNext())
    {
        // read the xml document and perform needed operations
    }
    xmlScanner.close();
} catch (FileNotFoundException fnfe)
{
    System.out.println("Your XML file was not found.");
}
```

Now, with Java 9, the `try-with-resource` statement can manage final variables without requiring a new variable declaration. So, we can now rewrite the earlier code, as shown here in Java 9:

```
Scanner xmlScanner = new Scanner(new File(xmlFile));
try ( while (xmlScanner.hasNext())
{
    {
        // read the xml document and perform needed operations
    }
    xmlScanner.close();
} catch (FileNotFoundException fnfe)
{
    System.out.println("Your XML file was not found.");
})
```

As you can see, the `xmlScanner` object reference is contained inside the `try-with-resource` statement block, which provides for automatic resource management. The resource will automatically be closed as soon as the `try-with-resource` statement block is exited.



You can also use a `finally` block as part of the `try-with-resource` statement.

Using the diamond operator

Introduced in Java 9, the diamond operator can be used with anonymous classes if the inferred data type is denotable. When a data type is inferred, it suggests that the Java Compiler can determine the data types in a method's invocation. This includes the declaration and any included arguments.



The diamond operator is the less-than and greater-than symbol pair (<>). It is not new to Java 9; rather, the specific use with anonymous classes is.

The diamond operator was introduced in Java 7 and made instantiating generic classes simpler. Here is a pre-Java 7 example:

```
| ArrayList<Student> roster = new ArrayList<Student>();
```

Then, in Java 7, we could rewrite it:

```
| ArrayList<Student> roster = new ArrayList<>();
```

The problem was that this method could not be used for anonymous classes. Here is an example in Java 8 that works fine:

```
public interface Example<T>
{
    void aMethod()
    {
        // interface code goes here
    }
}

Example example = new Example<Integer>()
{
    @Override
    public void aMethod()
    {
        // code
    }
};
```

While the preceding code works fine, when we change it to use the diamond operator, as shown here, a compiler error will occur:

```
public interface Example<T>
{
```

```
void aMethod()
{
    // interface code goes here
}
}

Example example = new Example<>()
{
    @Override
    public void aMethod()
    {
        // code
    }
};
```

The error results from using the diamond operator with anonymous inner classes. Java 9 to the rescue. While the preceding code results in a compile time error in Java 8, it works fine in Java 9.

Discontinuing use of the underscore

The underscore character (`_`) can no longer be used as a legal identifier name. Earlier attempts to remove the underscore in an identifier name were incomplete. The use of such would generate a combination of errors and warnings. With Java 9, the warnings are now errors. Consider the following sample code:

```
public class Java9Tests
{
    public static void main(String[] args)
    {
        int _ = 319;
        if (_ > 300 )
        {
            System.out.println("Your value us greater than 300.");
        }
        else
        {
            System.out.println("Your value is not greater than 300.");
        }
    }
}
```

The preceding code, in Java 8, will result in compiler warnings for `int _ = 319;` and `if (_ > 300)` statements. The warning is *as of release 9, '_ is a keyword, and may not be used as an identifier.* So, in Java 9, you will not be able to use the underscore by itself as a legal identifier.



It is considered bad programming practice to use identifier names that are not self-descriptive. So, the use of the underscore character by itself as an identifier name should not be a problematic change.

Making use of private interface methods

Lambda expressions were a big part of the Java 8 release. As a follow-up to that improvement, private methods in interfaces are now feasible. Previously, we could not share data between non-abstract methods of an interface. With Java 9, this data sharing is possible. Interface methods can now be private. Let's look at some sample code.

This first code snippet is how we might code an interface in Java 8:

```
 . . .
public interface characterTravel
{
    public default void walk()
    {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter desired pacing: ");
        int p = scanner.nextInt();
        p = p +1;
    }
    public default void run()
    {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter desired pacing: ");
        int p = scanner.nextInt();
        p = p +4;
    }
    public default void fastWalk()
    {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter desired pacing: ");
        int p = scanner.nextInt();
        p = p +2;
    }
    public default void retreat()
    {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter desired pacing: ");
        int p = scanner.nextInt();
        p = p - 1;
    }
    public default void fastRetreat()
    {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter desired pacing: ");
        int p = scanner.nextInt();
        p = p - 4;
    }
}
```

Now, in Java 9, we can rewrite this code. As you can see next, the redundant code has been moved into a single private method called `characterTravel`:

```
 . . .
public interface characterTravel
{
    public default void walk()
    {
        characterTravel("walk");
    }
    public default void run()
    {
        characterTravel("run");
    }
    public default void fastWalk()
    {
        characterTravel("fastWalk");
    }
    public default void retreat()
    {
        characterTravel("retreat");
    }
    public default void fastRetreat()
    {
        characterTravel("fastRetreat");
    }
    private default void characterTravel(String pace)
    {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter desired pacing: ");
        int p = scanner.nextInt();
        if (pace.equals("walk"))
        {
            p = p +1;
        }
        else if (pace.equals("run"))
        {
            p = p + 4;
        }
        else if (pace.equals("fastWalk"))
        {
            p = p + 2;
        }
        else if (pace.equals("retreat"))
        {
            p = p - 1;
        }
        else if (pace.equals("fastRetreat"))
        {
            p = p - 4;
        }
        else
        {
            //
        }
    }
}
```

Processing import statements correctly [JEP 216]

JEP 216 was issued as a fix to javac in regards to how import statements are processed. Prior to Java 9, there were instances where the order of import statements would impact if the source code was accepted or not.

When we develop applications in Java, we typically add import statements as we need them, resulting in an unordered list of import statements. IDEs do a great job of color-coding import statements that are not used, as well as informing us of import statements we need but that have not been included. It should not matter what order the import statements are in; there is no applicable hierarchy.

javac compiles classes in two primary steps. Specific to handling import statements, these steps are type resolution and member resolution. The type resolution consists of a review of the abstract syntax tree to identify declarations of classes and interfaces. The member resolution includes determining the class hierarchy and individual class variables and members.

With Java 9, the order we list import statements in our classes and files will no longer impact the compilation process. Let's look at an example:

```
package samplePackage;

import static SamplePackage.OuterPackage.Nested.*;
import SamplePackage.Thing.*;

public class OuterPackage
{
    public static class Nested implements Inner
    {
        // code
    }
}

package SamplePackage.Thing;

public interface Inner
{
    // code
}
```

In the preceding example, type resolution occurs and results in the following

realizations:

- `SamplePackage`.`OuterPackage` exists
- `SamplePackage`.`OuterPackage`.`Nested` exists
- `SamplePackage`.`Thing`.`Innner` exists

The next step is member resolution, and this is where the problem existed prior to Java 9. Here is an overview of the sequential steps javac would use to conduct the member resolution for our sample code:

1. Resolution of `SamplePackage`.`OuterPackage` begins.
2. The `SamplePackage`.`OuterPackage`.`Nested` import is processed.
3. Resolution of the `SamplePackage`.`Outer`.`Nested` class begins.
4. The inner interface is type checked, although, because it is not in scope at this point, inner cannot be resolved.
5. Resolution of `SamplePackage`.`Thing` begins. This step includes importing all member types of `SamplePackage`.`Thing` into scope.

So the error occurs, in our example, because `Innner` is out of scope when resolution is attempted. If steps 4 and 5 were swapped, it would not have been a problem.

The solution to the problem, implemented in Java 9, was to break the member resolution steps into additional sub-steps. Here are those steps:

1. Analyze the import statements.
2. Create the hierarchy (class and interfaces).
3. Analyze class headers and type parameters.

Summary

In this chapter, we covered changes in Java 9 with regards to variable handlers and how they relate to the Atomic Toolkit. We also covered depreciation warnings and why they are now suppressed under specific circumstances. Five enhancements to changes introduced with Java 7 as part of Project Coin were also reviewed. Finally, we explored the improvements to import statement processing.

In the next chapter, we will examine the structure of a Java module as specified by Project Jigsaw. We will take a deep dive into how Project Jigsaw is implemented as part of the Java platform. Code snippets from a sample e-commerce application are used throughout the chapter to demonstrate Java 9's modular system. Internal changes to the Java platform, in regards to the modular system, are also discussed.

Building Modular Applications with Java 9

In the last chapter, we covered changes in Java 9 with regards to variable handlers and how they related to the AtoMiC Toolkit. We also covered depreciation warnings and why they are now suppressed under specific circumstances. Five enhancements to changes introduced with Java 7 as part of *Project Coin* were also reviewed. Finally, we explored the improvements to import statement processing.

In this chapter, we will examine the structure of a Java module as specified by *Project Jigsaw*. We will take a deep-dive into how *Project Jigsaw* is implemented as part of the Java platform. We will also review key internal changes to the Java platform as they relate to the modular system.

The topics we will cover here are:

- An introduction to Java modularity
- Review of the Java platform's module system
- Modularizing JDK source code
- Modular runtime images
- Getting to know the module system
- Modular Java application packaging
- The Java linker
- Encapsulation of internal APIs

A modular primer

Before we dive into the Java 9 enhancements in this chapter, let's examine what modularity is in the context of Java.

We can define the term **modular** as a type of design or construction, in our context, of computer software. This type of software design involves a set of modules that collectively comprise the whole. A house, for example, can be built as a single structure or in a modular fashion where each room is constructed independently and joined to create a home. With this analogy, you could selectively add or not add modules in the creation of your home. The collection of modules, in our analogy, becomes the design of your home. Your design does not need to use every module, only the ones you want. So, for example, if there are basement and bonus room modules and your design does not include those modular rooms, those modules are not used to build your home. The alternative would be that every home would include every room, not just the ones that are used. This, of course, would be wasteful. Let's see how that correlates to software.

This concept can be applied to computer architecture and software systems. Our systems can be comprised of several components instead of one behemoth system. As you can likely imagine, this provides us with some specific benefits:

- We should be able to scale our Java applications to run on small devices
- Our Java applications will be smaller
- Our modular code can be more targeted
- Increased use of the object-oriented programming model
- Additional opportunities for encapsulation
- Our code will be more efficient
- Java applications will have increased performance
- Overall system complexity is reduced
- Testing and debugging is easier
- Code maintenance is easier

The shift to a modular system for Java was necessary for several reasons. Here are the primary conditions of the Java platform as of Java 9 that led to the creation of the module system for the Java 9 platform:

- The **Java Development Kit (JDK)** was simply too big. This made it difficult to support small devices. Even with the compact profiles discussed in the next section, supporting some small devices was difficult at best and, in some cases, not possible.
- Due to the over-sized JDK, it was difficult to support truly optimized performance with our Java applications. In this case, smaller is better.
- The **Java Runtime Environment (JRE)** was too large to efficiently test and maintain our Java applications. This results in time consuming, inefficient testing, and maintenance operations.
- The **Java Archive (JAR)** files were also too large. This made supporting small devices problematic.
- Because the JDK and JRE were all encompassing, security was of great concern. Internal APIs, for example, that were not used by the Java application, were still available due to the nature of the public access modifier.
- Finally, our Java applications were unnecessarily large.

Modular systems have the following requirements:

- There must be a common interface to permit interoperability among all connected modules
- Isolated and connected testing must be supported
- Compile time operations must be able to identify which modules are in use
- Runtime support for modules

A module is a new concept and component in Java 9; it is a named collection of data and code. Specifically, modules are a collection of:

- Packages
- Classes
- Interfaces
- Code
- Data
- Resources

Key to successful implementation, a module in Java 9 is self-described in its modular declaration. Module names must be unique and typically use the reverse domain name schema. Here is an example declaration:

```
|   module com.three19.irisScan { }
```

Module declarations are contained in a `module-info.java` file that should be in the

module's root folder. As one might expect, this file is compiled into a `module-info.class` file and will be placed in the appropriate output directory. These output directories are established in the module source code.

In the next sections, we will look at specific changes for Java 9 in regards to modularity.

Reviewing Java's platform module system [JEP-200]

The core aim of JEP-200 was to modularize the **Java Development Kit (JDK)** using the **Java Platform Module System (JPMS)**. Prior to Java 9, our familiarity with the JDK includes awareness of its major components:

- Java runtime environment (JRE)
- The interpreter (java)
- Compiler (javac)
- The archiver (jar)
- Document generator (javadoc)

The task of modularizing the JDK was to break it into components that could be combined at compile time or runtime. The modular structure is based on the following modular profiles established as compact profiles in Java 8. Each of the three profiles is detailed in the following tables:

Compact profile 1:

java.io	java.lang.annotation	java.lang.invoke
java.lang.ref	java.lang.reflect	java.math
java.net	java.nio	java.nio.channels
java.nio.channels.spi	java.nio.charset	java.nio.charset.spi
java.nio.file	java.nio.file.attribute	java.nio.file.spi
java.security	java.security.cert	java.security.interfaces
java.security.spec	java.text	java.text.spi
java.time	java.time.chrono	java.time.format
java.time.temporal	java.time.zone	java.util
java.util.concurrent	java.util.concurrent.atomic	java.util.concurrent.locks
java.util.function	java.util.jar	java.util.logging
java.util.regex	java.util.spi	java.util.stream
java.util.zip	javax.crypto	javax.crypto.interfaces
javax.crypto.spec	javax.net	javax.net.ssl
javax.script	javax.security.auth	javax.security.auth.callback

javax.security.auth.login	javax.security.auth.spi	javax.security.auth.spi
javax.security.auth.x500	javax.security.cert	

Compact profile 2:

java.rmi	java.rmi.activation	java.rmi.dgc
java.rmi.registry	java.rmi.server	java.sql
javax.rmi.ssl	javax.sql	javax.transaction
javax.transaction.xa	javax.xml	javax.xml.database
javax.xml.namespace	javax.xml.parsers	javax.xml.stream
javax.xml.stream.events	javax.xml.stream.util	javax.xml.transform
javax.xml.transform.dom	javax.xml.transform.sax	javax.xml.transform.stax
javax.xml.transform.stream	javax.xml.validation	javax.xml.xpath
org.w3c.dom	org.w3c.dom.bootstrap	org.w3c.dom.events
org.w3c.dom.ls	org.xml.sax	org.xml.sax.ext
org.xml.sax.helpers		

Compact profile 3:

java.lang.instrument	java.lang.management	java.security.acl
java.util.prefs	javax.annotation.processing	javax.lang.model
javax.lang.model.element	javax.lang.model.type	javax.lang.model.util
javax.management	javax.management.loading	javax.management.modelmbe
javax.management.monitor	javax.management.openmbean	javax.management.relation
javax.management.remote	javax.management.remote.rmi	javax.management.timer
javax.naming	javax.naming.directory	javax.naming.event
javax.naming.ldap	javax.naming.spi	javax.security.auth.kerbe
javax.security.sasl	javax.sql.rowset	javax.sql.rowset.serial
javax.sql.rowest.spi	javax.tools	javax.xml.crypto
javax.xml.crypto.dom	javax.xml.crypto.dsig	javax.xml.crypto.dsig.dom
javax.xml.crypto.dsig.keyinfo	javax.xml.crypto.dsig.spec	org.ietf.jgss

The three compact module profiles represent the basis for the standardized modular

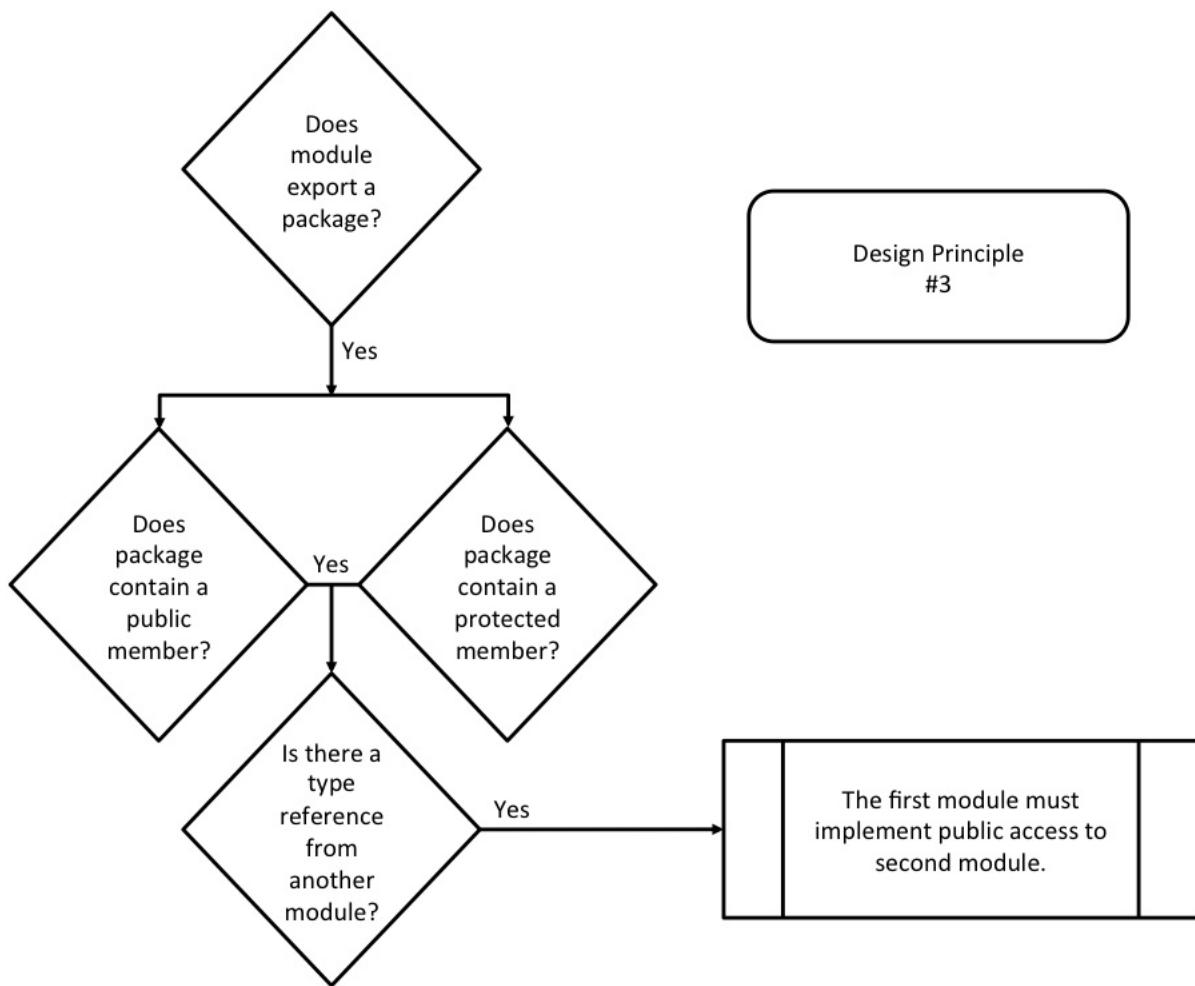
system in Java 9. The effectiveness of this standardization relies on the following six principles:

- All JCP-governed modules must start with the string `java..` So, if a module on spatial utilities was being developed it would have a name such as `java.spatial.util.`



JCP refers to the **Java Community Process**. JCP allows developers to create technical specifications for Java. You can learn more about JCP and become a member at the official JCP website--<http://www.jcp.org>.

- Non-JCP modules are considered part of the JDK and their names must start with the string `jdk..`
- Ensure method invocation chaining works properly. This is best illustrated with the following flowchart:

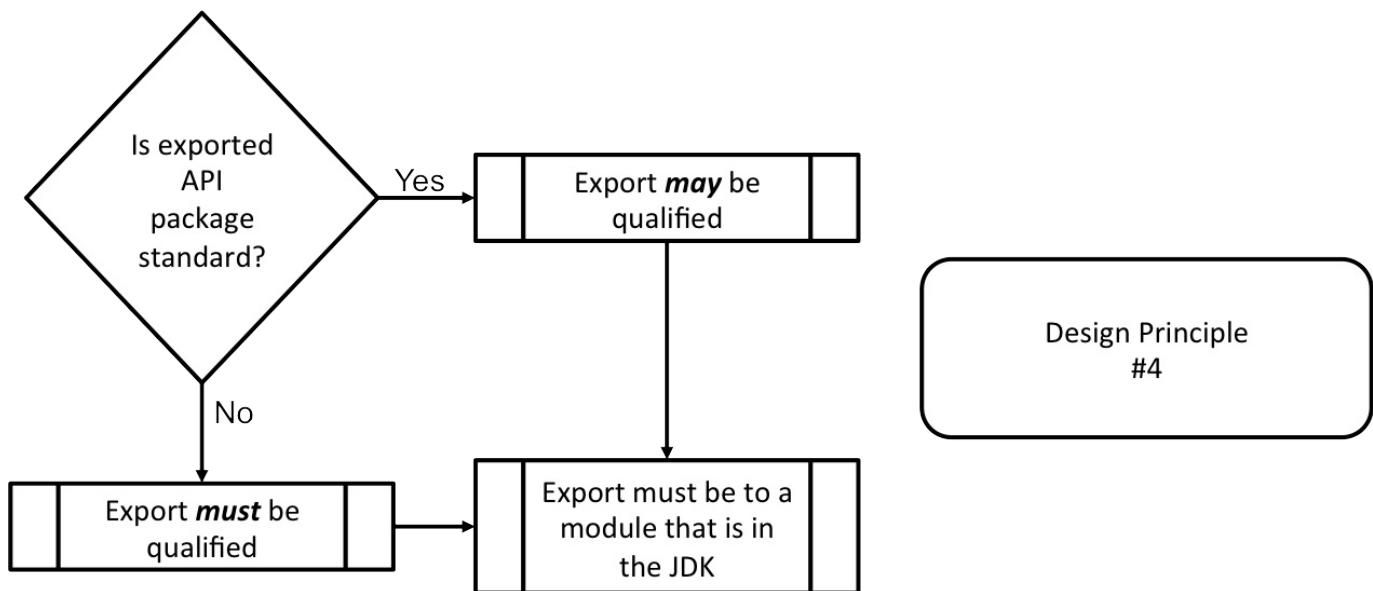


As you can see in the preceding flowchart, it only applies to modules



that export a package.

- The fourth principle deals with both standard and non-standard API packages being used in a standard module. The following flowchart illustrates the implementation of this principle's covenants:



- The fifth design principle is that standard modules can be dependent upon more than one non-standard module. While this dependency is permitted, implied readability access to non-standard modules is not.
- The final design principle ensures non-standard modules do not export standard API packages.

Modularizing JDK source code

[JEP-201]

As previously mentioned, Project Jigsaw had the goal of modularization. The envisioned standard modular system would be applied to the Java SE platform and the JDK. In addition to efficiency gains, the modular shift would result in better security and ease maintainability. The enhancement detailed in JEP-201 focused on JDK source code reorganization. Let's take a closer look.

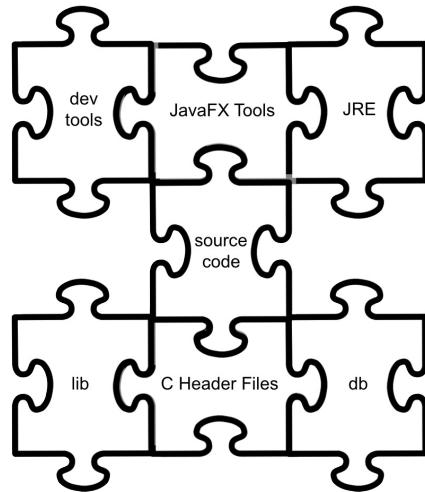
Reorganizing the JDK's source code is a significant task and was accomplished with the following subset of goals:

- Provide JDK developers insights and familiarity with the new Java 9 modular system. So, this goal was aimed at developers of the JDK, not mainstream developers.
- Ensure modular boundaries are established and maintained throughout the JDK build process. This was a necessary precaution so the modular system would be stable throughout Java 9's enhancements and, more specifically, in implementing the modular system.
- The third goal was to ensure future enhancements, specifically with *Project Jigsaw*, could be easily integrated into the new modular system.

The significance of this source code reorganization cannot be overstated. The pre-Java 9 source code organization is 20 years old. This overdue JDK source code reorganization will make the code much easier to maintain. Let's look at the previous organization of the JDK source code and then examine the changes.

Pre-Java 9 JDK source code organization

The JDK is a compilation of code files, tools, libraries, and more. The following illustration provides an overview of the JDK components:



The pre-Java 9 organization of the JDK components in the preceding illustration are detailed in the next seven subsections.

Development tools

The development tools are located in the `\bin` directory. These tools include seven broad categorizations, each detailed in the subsequent sections.

Deployment

This is a set of tools intended to help deploy Java applications:

- `appletviewer`: This tool gives you the ability to run and debug Java applets without the need for a web browser.
- `extcheck`: This tool allows you to find conflicts in JAR files.
- `jar`: This tool is used for creating and manipulating JAR files. JAR files are Java Archive files.
- `java`: This is the Java application launcher.
- `javac`: This is the Java Compiler.
- `javadoc`: This tool generates API documentation.
- `javah`: This tool allows you to write native methods; it generates C header files.
- `javap`: This tool disassembles class files.
- `javapackager`: For signing and packaging Java applications, including JavaFX.
- `jdb`: This is the Java debugger.
- `jdeps`: This is an analyzer for Java class dependencies.
- `pack200`: This is a tool that compresses JAR files into `pack200` files. The compression ratio using this tool is impressive.
- `unpack200`: This tool unpacks `pack200` files resulting in JAR files.

Internationalization

If you are interested in creating localizable applications, the following tool might come in handy:

- `native2ascii`: This tool creates Unicode Latin-1 from normal text.

Monitoring

Monitoring tools used for providing JVM performance data include:

- `jps`: This is the **JVM process status tool (jps)**. It provides a list of HotSpot JVMs on a specific system.
- `jstat`: This is the JVM statistics monitoring tool. It collects log data and performance information from a machine with a HotSpot JVM.
- `jstatd`: This is the **jstat** daemon tool. It runs an RMI server app for monitoring HotSpot JVM operations.

RMI

RMI tools are **Remote Method Invocation** tools. They help developers create applications that operate over a network to include the internet:

- `rmic`: This tool can generate stubs and skeletons for objects over a network
- `rmiregistry`: This is a registry service for remote objects
- `rmid`: This tool is an activation system daemon for RMI
- `serialver`: This tool returns the class `serialVersionUID` value

Security

This set of security tools empowers developers to create security policies that can be enforced on the developer's computer system as well as on remote systems:

- `keytool`: This tool manages security certificates and keystores
- `jarsigner`: This tool generates and verifies JAR signatures for creating/opening JAR files
- `policytool`: This tool has a graphical user interface that helps developers manage their security policy files

Troubleshooting

These experimental troubleshooting tools are useful for very specific troubleshooting. They are experimental and, therefore, not officially supported:

- `jinfo`: This tool provides configuration information for specific processes, files, or servers.
- `jhat`: This is a heap dump tool. It instantiates a web server so that a heap can be viewed with a browser.
- `jmap`: This displays heap and shared object memory maps from a process, file, or server.
- `jsadebugd`: This is Java's Serviceability Agent Debug Daemon. It acts as a debug server for a process or file.
- `jstack`: This is a Java Stack Trace tool that provides a thread stack trace for a process, file, or server.

Web services

This set of tools provides a utility that can be used with **Java Web Start** and other web services:

- `javaws`: This is a command line tool that launches Java Web Start.
- `schemagen`: This tool generates schemas for Java architecture. These schemas are used for XML binding.
- `wsgen`: This tool is used for generating JAX-WS artifacts that are portable.
- `wsimport`: This tool is used for importing portable JAX-WS artifacts.
- `xjc`: This is the binding compiler that is used for XML binding.

JavaFX tools

The JavaFX tools are located in a few different places including `\bin`, `\man`, and `\lib` directories.

Java runtime environment

The **Java runtime environment (JRE)** is located in the `\jre` directory. Key contents include the **Java Virtual Machine (JVM)** and class libraries.

Source code

The JDK's source code, pre-Java 9, has the following basic organizational schema:

```
| source code / [shared, OS-specific] / [classes / native] / Java API  
| package name / [.file extension]
```

Let's look at this a bit closer. After the source code, we have two options. If the code is cross-platform, then it is a shared directory; otherwise, it is operating system specific. For example:

```
| src/share/...  
| src/windows/...
```

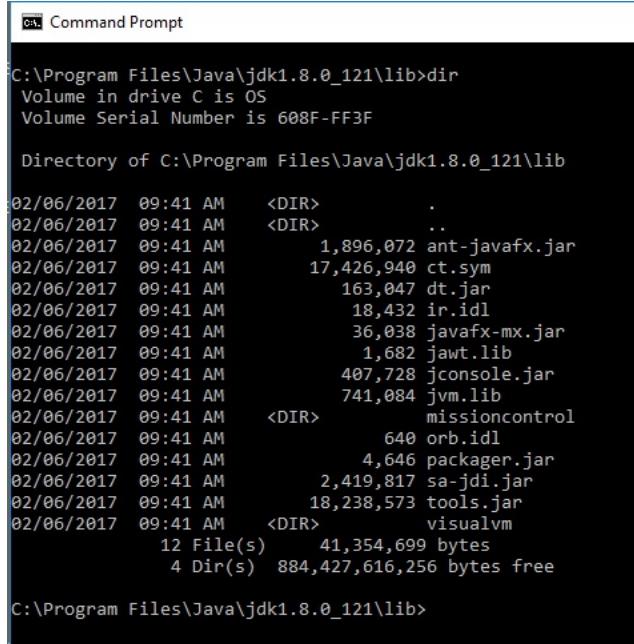
Next, we have the classes directory or a native language directory. For example:

```
| src/share/classes/...  
| src/share/classes/java/...
```

Next, we have the name of the Java API package followed by the file extension. The file extensions depend on content such as `.java`, `.c`, and more.

Libraries

The `\lib` directory houses class libraries that are needed by one or more of the development tools in the `\bin` directory. Here is a list of files in a typical Java 8 `\lib` directory:



```
C:\> Command Prompt
C:\Program Files\Java\jdk1.8.0_121\lib>dir
 Volume in drive C is OS
 Volume Serial Number is 608F-FF3F

 Directory of C:\Program Files\Java\jdk1.8.0_121\lib

02/06/2017  09:41 AM    <DIR>      .
02/06/2017  09:41 AM    <DIR>      ..
02/06/2017  09:41 AM           1,896,072 ant-javafx.jar
02/06/2017  09:41 AM           17,426,940 ct.sym
02/06/2017  09:41 AM           163,047 dt.jar
02/06/2017  09:41 AM           18,432 ir.idl
02/06/2017  09:41 AM           36,038 javafx-mx.jar
02/06/2017  09:41 AM           1,682 jawt.lib
02/06/2017  09:41 AM           407,728 jconsole.jar
02/06/2017  09:41 AM           741,084 jvm.lib
02/06/2017  09:41 AM    <DIR>      missioncontrol
02/06/2017  09:41 AM           640 orb.idl
02/06/2017  09:41 AM           4,646 packager.jar
02/06/2017  09:41 AM           2,419,817 sa-jdi.jar
02/06/2017  09:41 AM           18,238,573 tools.jar
02/06/2017  09:41 AM    <DIR>      visualvm
               12 File(s)   41,354,699 bytes
               4 Dir(s)  884,427,616,256 bytes free

C:\Program Files\Java\jdk1.8.0_121\lib>
```

Reviewing the directory listing does not provide a great level of granular insight. We can list the classes contained in any of the `.jar` files with the following command--
`jar tvf fileName.jar`. As an example, here is the class listing generated from executing `jar tvf javafx-mx.jar` at the command line:

Command Prompt

```
C:\Program Files\Java\jdk1.8.0_121\lib>jar tvf javafx-mx.jar
  0 Mon Dec 12 12:01:10 CST 2016 META-INF/
  25 Mon Dec 12 12:01:10 CST 2016 META-INF/MANIFEST.MF
  0 Mon Dec 12 12:00:54 CST 2016 com/
  0 Mon Dec 12 12:00:54 CST 2016 com/oracle/
  0 Mon Dec 12 12:00:54 CST 2016 com/oracle/javafx/
  0 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/
  963 Mon Dec 12 12:00:54 CST 2016 com/oracle/javafx/jmx/MXExtensionImpl.class
  667 Mon Dec 12 12:00:54 CST 2016 com/oracle/javafx/jmx/SGMXBean.class
  728 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/SGMXBeanImpl$1.class
  12268 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/SGMXBeanImpl.class
  0 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/
  744 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/ImmutableJSONDocument.class
  1052 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/JSONDocument$IteratorWrapper.class
  1068 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/JSONDocument$Type.class
  11846 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/JSONDocument.class
  893 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/JSONException.class
  936 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/JSONFactory.class
  1973 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/JSONReader$EventType.class
  800 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/JSONReader.class
  590 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/JSONWriter$Container.class
  1115 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/JSONWriter$ContainerType.class
  4781 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/JSONWriter.class
  0 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/impl/
  1288 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/impl/JSONMessages.class
  4966 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/impl/JSONScanner.class
  2273 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/impl/JSONStreamReaderImpl$1.class
  8241 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/impl/JSONStreamReaderImpl.class
  5486 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/impl/JSONSymbol.class
  916 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/impl/JSONMessagesBundle.properties
  1968 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/impl/JSONMessagesBundle_ja.properties
  1284 Mon Dec 12 12:00:56 CST 2016 com/oracle/javafx/jmx/json/impl/JSONMessagesBundle_zh_CN.properties

C:\Program Files\Java\jdk1.8.0_121\lib>
```

C header files

The `/include` directory contains C header files. These files primarily support the following:

- **Java Native Interface (JNI)**: This is used for native-code programming support. The JNI is used to embed Java native methods and the JVM into native apps.
- **JVM Tool Interface (JVM TI)**: This is used by tools for state inspections and execution control for apps running the JVM.

Database

The Apache Derby relational database is stored in the `/db` directory. You can learn more about Java DB at the following sites:

<http://docs.oracle.com/javadb/support/overview.html>

http://db.apache.org/derby/manuals/#docs_10.11

JDK source code reorganized

In a previous section, you learned that the pre-Java 9 source code organization schema was as follows:

```
| source code / [shared, OS-specific] / [classes / native] / Java API  
| package name / [.file extension]
```

In Java 9, we have a new modular schema. That schema follows:

```
| source code / module / [shared, OS-specific] / [classes / native /  
| configuration] / [ package / include / library ] /  
| [.file extension]
```

There are a few differences in the new schema, most notably the module name. After the shared or OS-specific directory, there is either the classes directory, the native directory for C or C++ source files, or a configuration directory. This seemingly rudimentary organization schema changes results in a much more maintainable code base.

Understanding modular run-time images [JEP-220]

Java 9's modular system required changes to the runtime images for compatibility. Benefits of these changes include enhancements in the following areas:

- Maintainability
- Performance
- Security

Core to these changes was a new URI schema used for resource naming. These resources include modules and classes.



*A Uniform Resource Identifier (**URI**) is similar to a **URL (Uniform Resource Locator)** in that it identifies the name and location of something. For a URL, that something is a web page; for a URI, it is a resource.*

There were five primary goals for JEP-220 and these are detailed in the following sections.

Runtime format adoption

A run-time format was created for Java 9, for adoption by stored classes and other resource files. This format is applicable for stored classes and resources under the following circumstances:

- When the new run-time format has greater efficiencies (time and space) than the pre-Java 9 JAR format.



A JAR file is a Java ARchive file. This is a compressed file format based on the legacy ZIP format.

- When stored classes and other resources can be individually isolated and loaded.
- When JDK and library classes and resources can be stored. This includes app modules as well.
- When they are devised in such a way as to promote future enhancements. This requires them to be extensible, documented, and flexible.

Runtime image restructure

There are two types of runtime images in Java--JDK and JRE. With Java 9, both of these image types were restructured to differentiate between files that can be used and modified by users to internal files that can be used but not modified by developers and their apps.

The JDK build system, prior to Java 9, produces both a JRE and a JDK. The JRE is a complete implementation of the Java platform. The JDK includes the JRE as well as other tools and libraries. A notable change in Java 9 is that the JRE subdirectory is no longer part of the JDK image. This change was made, in part, to ensure both image types (JDK and JRE) have identical image structures. With a common and reorganized structure, future changes will be more efficiently integrated.



If you created custom plugins prior to Java 9 that address a specific structure, your app might not work in Java 9. This is also true if you are explicitly addressing `tools.jar`.

The following diagram provides a high-level view of the contents of each image before Java 9's release:

JRE Image	JDK Image
<ul style="list-style-type: none">• bin• lib	<ul style="list-style-type: none">• jre• bin• demo• sample• man• include• lib

The Java 9 runtime images are illustrated in the following diagram. As shown, a full JDK image contains the same directories as a modular runtime image as well as demo, sample, man, and includes directories:

Modular RunTime Image	Full JDK Image
<ul style="list-style-type: none">• bin• conf• lib	<ul style="list-style-type: none">• bin• conf• lib• demo• sample• man• include

There is no longer a difference between a JRE or JDK image. Now, with Java 9, a JDK image is a JRE image that contains a full set of dev tools.

Supporting common operations

Developers occasionally must write code that performs operations requiring access to the runtime image. Java 9 includes support for these common operations. This is possible due to the restructuring and standardized JDK and JRE runtime image structures.

De-privileging JDK classes

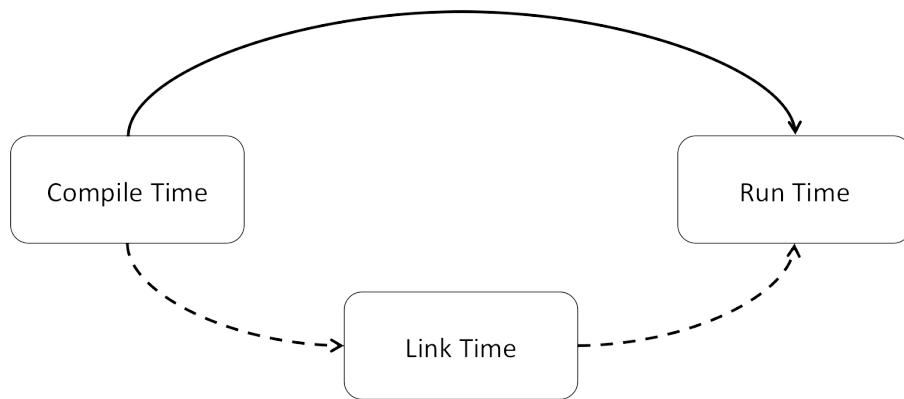
Java 9 allows privilege revocation for individual JDK classes. This change strengthens system security in that it ensures JDK classes only receive the permissions required for system operations.

Preserving existing behaviors

The final goal of the JEP-220 was to ensure currently existing classes are not negatively impacted. This refers to applications that do not have dependencies on internal JDK or JRE runtime images.

Getting to know the module system [JEP-261]

The purpose of this JEP was the implementation of the new module system for the Java platform. You will recall that the modular system was created to provide reliable configuration and strong encapsulation for Java programs. Key to this implementation was the concept of link time. As illustrated here, link time is an optional phase in between compile time and runtime. This phase allows the assembly of the appropriate modules into an optimized runtime image. This is possible, in part, due to the jlink linking tool which you will learn more about later in this chapter:



Module paths

It is important to organize modules so that they can be easily located. The module path, a sequence of module components or directories, provides the organizational structure used by searches. These path components are searched for in order, returning the first path component that comprises a module.

Modules and their paths should not be considered to be the same as packages or class paths. They are indeed different and have a greater level of fidelity. The key difference is that, with classpaths, a singular component is searched for. Module path searches return complete modules. This type of search is possible by searching the following paths, in the presented order, until a module is returned:

- Compilation module path
- Upgrade module path
- System modules
- Application module path

Let's briefly review each of these paths. The compilation module path is only applicable at compile time and contains the module definitions. The upgrade module path has the compiled module definitions. The system modules are built-in and include Java SE and JDK modules. The final path, the application module path, has the compiled module definitions from the application modules as well as the library modules.

Access-control boundary violations

As a professional developer, you always want your code to be secure, portable, and bug-free, which requires strict adherence to Java constructs such as encapsulation. There are occasions, such as with white box testing, that you need to break the encapsulation that the JVM mandates. This mandate permits cross-modular access.

To permit breaking the encapsulation, you can add an `add-exports` option in your module declaration. Here is the syntax you will use:

```
module com.three19.irisScan
{
    - - add-exports <source-module>/<package> = <target-module>
    (, <target-module> )*
}
```

Let's take a closer look at the preceding syntax. The `<source-module>` and `<target-module>` are module names and `<package>` is the name of the package. Using the `add-exports` option permits us to violate access-control boundaries.

There are two rules regarding using the `add-exports` option:

- It can be used multiple times in a module
- Each use must be of a unique pairing of `<source-module>` and `<target-module>`



It is not recommended that the `add-exports` option be used unless absolutely necessary. Its use permits dangerous access to a library module's internal API. This type of use makes your code dependent on the internal API not changing, which is beyond your control.

Runtime

The HotSpot virtual machine implements the <options> for the `jmod` and `jlink` command-line tools. Here is the list of <options> for the `jmod` command-line tool:

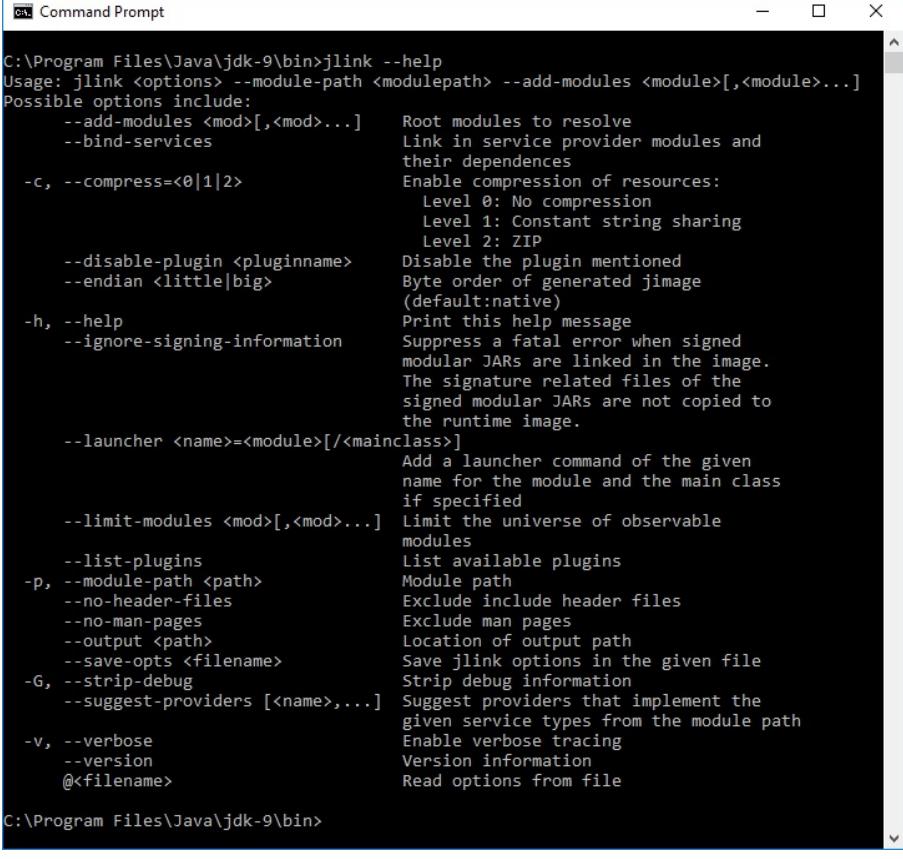
```
C:\Program Files\Java\jdk-9\bin>jmod --help
Usage: jmod (create|extract|list|describe|hash) <OPTIONS> <jmod-file>

Main operation modes:
create    - Creates a new jmod archive
extract   - Extracts all the files from the archive
list      - Prints the names of all the entries
describe  - Prints the module details
hash      - Records hashes of tied modules.

Option           Description
-----
--class-path <path>          Application jar files|dir containing
                               classes
--cmds <path>                Location of native commands
--config <path>              Location of user-editable config files
--dir <path>                 Target directory for extract
--dry-run                  Dry run of hash mode
--exclude <pattern-list>     Exclude files matching the supplied
                               comma separated pattern list, each
                               element using one the following
                               forms: <glob-pattern>, glob:<glob-
                               pattern> or regex:<regex-pattern>
-h, --help            Print this usage message
--hash-modules <regex-pattern> Compute and record hashes to tie a
                               packaged module with modules
                               matching the given <regex-pattern>
                               and depending upon it directly or
                               indirectly. The hashes are recorded
                               in the JMOD file being created, or a
                               JMOD file or modular JAR on the
                               module path specified the jmod hash
                               command.
--header-files <path>        Location of header files
--help-extra           Print help on extra options
--legal-notices <path>       Location of legal notices
--libs <path>              Location of native libraries
--main-class <class-name>    Main class
--man-pages <path>          Location of man pages
--module-version <module-version> Module version
-p, --module-path <path>    Module path
--target-platform <target-platform> Target platform
--version               Version information
@<filename>             Read options from the specified file

C:\Program Files\Java\jdk-9\bin>
```

Here is the list of <options> for the jlink command-line tool:



```
C:\Program Files\Java\jdk-9\bin>jlink --help
Usage: jlink <options> --module-path <modulepath> --add-modules <module>[,<module>...]
Possible options include:
  --add-modules <mod>[,<mod>...]      Root modules to resolve
  --bind-services                          Link in service provider modules and
                                           their dependences
  -c, --compress=<0|1|2>                 Enable compression of resources:
                                           Level 0: No compression
                                           Level 1: Constant string sharing
                                           Level 2: ZIP
  --disable-plugin <pluginname>          Disable the plugin mentioned
  --endian <little|big>                  Byte order of generated jimage
                                           (default:native)
  -h, --help                            Print this help message
  --ignore-signing-information          Suppress a fatal error when signed
                                           modular JARs are linked in the image.
                                           The signature related files of the
                                           signed modular JARs are not copied to
                                           the runtime image.
  --launcher <name>=<module>[/<mainclass>]    Add a launcher command of the given
                                           name for the module and the main class
                                           if specified
  --limit-modules <mod>[,<mod>...]       Limit the universe of observable
                                           modules
  --list-plugins                         List available plugins
  -p, --module-path <path>              Module path
  --no-header-files                     Exclude include header files
  --no-man-pages                       Exclude man pages
  --output <path>                      Location of output path
  --save-opt <filename>                Save jlink options in the given file
  -G, --strip-debug                    Strip debug information
  --suggest-providers [<name>,...]     Suggest providers that implement the
                                           given service types from the module path
  -v, --verbose                         Enable verbose tracing
  --version                           Version information
  @<filename>                        Read options from file

C:\Program Files\Java\jdk-9\bin>
```

Modular Java application packaging [JEP-275]

One of the great improvements in Java 9 is the size of the runtime binaries generated by the **Java Packager**. This is possible in part due to the **Java Linker**, which is covered in the next section. The Java Packager's workflow has essentially remained the same in Java 9 as it was in Java 8. There have been, as you will see later in this section, new tools added to the workflow.

The Java Packager solely creates JDK 9 applications. This change to the Java Packager is intended to streamline and make the process of generating runtime images more efficient. So, the Java Packager will only create runtime images for the SDK version that it is associated with.

Advanced look at the Java Linker

Prior to the Java Linker tool, `jlink`, introduced in Java 9, runtime image creation included copying the entire JRE. Then, unused components are removed. Simply put, `jlink` facilitates the creation of runtime images with only the required modules. `jlink` is used by the Java Packager to generate an embedded runtime image.

Java Packager options

The syntax for the Java Packager is as follows:

```
|   javapackager -command [-options]
```

There are five different commands (`-command`) that can be used. They are described as follows:

command	Description
<code>-createbss</code>	This command is used for converting files from CSS to binary
<code>-createjar</code>	This command, used along with additional parameters, creates a JAR archive file
<code>-deploy</code>	This command is used to generate jnlp and HTML files
<code>-makeall</code>	Combines the <code>-createjar</code> , <code>-deploy</code> , and compilation steps
<code>-signJar</code>	This command creates and signs a JAR file

The `[-options]` for the `-createbss` command include:

```
-outdir <dir>
    name of the directory to generate output file to.
-srcdir <dir>
    Base dir of the files to pack.
-srcfiles <files>
    List of files in srcdir. If omitted, all files in srcdir (which
    is a mandatory argument in this case) will be used.
```

The `[-options]` for the `-createjar` command include:

```
-appclass <application class>
    qualified name of the application class to be executed.
-preloader <preloader class>
    qualified name of the preloader class to be executed.
-paramfile <file>
    properties file with default named application parameters.
-argument arg
    An unnamed argument to be put in <fx:argument> element in the JNLP
    file.
-classpath <files>
    list of dependent jar file names.
-manifestAttrs <manifest attributes>
    List of additional manifest attributes. Syntax: "name1=value1,
    name2=value2,name3=value3.
-noembedlauncher
    If present, the packager will not add the JavaFX launcher classes
    to the jarfile.
-nocss2bin
    The packager won't convert CSS files to binary form before copying
    to jar.
-runtimeversion <version>
    version of the required JavaFX Runtime.
-outdir <dir>
    name of the directory to generate output file to.
-outfile <filename>
    The name (without the extension) of the resulting file.
-srcdir <dir>
    Base dir of the files to pack.
-srcfiles <files>
    List of files in srcdir. If omitted, all files in srcdir (which
    is a mandatory argument in this case) will be packed.
```

The [-options] for the -deploy command include:

```
-native <type>
    generate self-contained application bundles (if possible).
    If type is specified then only bundle of this type is created.
    List of supported types includes: installer, image, exe, msi, dmg, pkg, rpm, deb.
-name <name>
    name of the application.
-appclass <application class>
    qualified name of the application class to be executed.
-outdir <dir>
    name of the directory to generate output file to.
-outfile <filename>
    The name (without the extension) of the resulting file.
-srcdir <dir>
    Base dir of the files to pack.
-srcfiles <files>
    List of files in srcdir. If omitted, all files in srcdir (which
    is a mandatory argument in this case) will be used.
-m <modulename>[/<mainclass>]
--module <modulename>[/<mainclass>]
    the initial module to resolve, and the name of the main class
    to execute if not specified by the module
-p <module path>
--module-path <module path>...
    A : separated list of directories, each directory
    is a directory of modules.
--add-modules <modulename>[,<modulename>...]
    root modules to resolve in addition to the initial module
--limit-modules <modulename>[,<modulename>...]
    limit the universe of observable modules
--strip-native-commands <true/false>
    include or exclude the native commands
-title <title>
    title of the application.
--vendor <vendor>
    vendor of the application.
--description <description>
    description of the application.
--embedjnlp
    If present, the jnlp file will be embedded in the html document.
--embedCertificates
    If present, the certificates will be embedded in the jnlp file.
--allPermissions
    If present, the application will require all security permissions
    in the jnlp file.
--updateMode <updatemode>
    sets the update mode for the jnlp file.
--isExtension
    if present, the srcfiles are treated as extensions.
--
```

Here are the remaining [-options] for the -deploy command:

```
-callbacks
    specifies user callback methods in generated HTML. The format is
    "name1:value1,name2:value2,..."
-templateInFilename
    name of the html template file. Placeholders are in form of
    #XXXX.YYYY(APPID)#
-templateOutFilename
    name of the html file to write the filled-in template to.
-templateId
    Application ID of the application for template processing.
-argument arg
    An unnamed argument to be put in <fx:argument> element in the JNLP
    file.
-preloader <preloader class>
    qualified name of the preloader class to be executed.
-paramfile <file>
    properties file with default named application parameters.
-htmlparamfile <file>
    properties file with parameters for the resulting applet.
-width <width>
    width of the application.
-height <height>
    height of the application.
```

The [-options] for the `-makeall` command include:

```
-appclass <application class>
    qualified name of the application class to be executed.
-preloader <preloader class>
    qualified name of the preloader class to be executed.
-classpath <files>
    list of dependent jar file names.
-name <name>
    name of the application.
-width <width>
    width of the application.
-height <height>
    height of the application.
-v      enable verbose output.
```

The [-options] for the `-signJar` include:

```
-keyStore <file>
          Keystore filename.
-alias
          Alias for the key.
-storePass
          Password to check integrity of the keystore or unlock the keystore.
-keyPass
          Password for recovering the key.
-storeType
          Keystore type, the default value is "jks".
-outdir <dir>
          name of the directory to generate output file(s) to.
-srcdir <dir>
          Base dir of the files to signed.
-srcfiles <files>
          List of files in srcdir. If omitted, all files in srcdir (which
          is a mandatory argument in this case) will be signed.
```

The Java Packager is divided into two modules:

```
| jdk.packager
| jdk.packager.services
```

JLink - The Java Linker [JEP-282]

The Java Linker, commonly referred to as JLink, is a tool that was created to create custom runtime images. This tool collects the appropriate modules along with their dependencies, then optimizes them to create the image. This represents a big change for Java, with the release of Java 9. Before the Java Linker tool, `jlink`, was available, runtime image creation included initially copying the entire JRE. In a subsequent step, the unused components were removed. In Java 9, `jlink` creates runtime images with only the needed modules. `jlink` is used by the Java Packager to generate an embedded runtime image.

As illustrated in a previous section, JEP-282 resulted in link time as an optional phase between compile time and runtime. It is in this phase that the appropriate modules are assembled into an optimized runtime image.

JLink is a command-line linking tool that permits the creation of runtime images containing a smaller subset of the JDK modules. This results in smaller runtime images. The following syntax consists of four components--the `jlink` command, options, the module path, and the output path:

```
| $ jlink <options> ---module-path <modulepath> --output <path>
```

Here is a list of the options that can be used with the `jlink` tool along with brief descriptions of each:

```
C:\ Command Prompt
C:\Program Files\Java\jdk-9\bin>jlink --help
Usage: jlink <options> --module-path <modulepath> --add-modules <module>[,<module>...]
Possible options include:
  --add-modules <mod>[,<mod>...]      Root modules to resolve
  --bind-services                          Link in service provider modules and
                                            their dependences
  -c, --compress=<0|1|2>                  Enable compression of resources:
                                            Level 0: No compression
                                            Level 1: Constant string sharing
                                            Level 2: ZIP
  --disable-plugin <pluginname>          Disable the plugin mentioned
  --endian <little|big>                  Byte order of generated jimage
                                            (default:native)
  -h, --help                            Print this help message
  --ignore-signing-information          Suppress a fatal error when signed
                                            modular JARs are linked in the image.
                                            The signature related files of the
                                            signed modular JARs are not copied to
                                            the runtime image.
  --launcher <name>=<module>/<mainclass>  Add a launcher command of the given
                                            name for the module and the main class
                                            if specified
  --limit-modules <mod>[,<mod>...]       Limit the universe of observable
                                            modules
  --list-plugins                         List available plugins
  -p, --module-path <path>              Module path
  --no-header-files                     Exclude include header files
  --no-man-pages                        Exclude man pages
  --output <path>                      Location of output path
  --save-opts <filename>                Save jlink options in the given file
  -G, --strip-debug                    Strip debug information
  --suggest-providers [<name>,...]     Suggest providers that implement the
                                            given service types from the module path
  -v, --verbose                         Enable verbose tracing
  --version                           Version information
  @<filename>                         Read options from file

C:\Program Files\Java\jdk-9\bin>
```

The module path tells the linker where to find the modules. The linker will not use exploded modules or JAR/JMOD files.

The output path simply informs the linker where to save the custom run-time image.

Encapsulating most internal APIs [JEP-260]

JEP-260 was implemented to make the Java platform more secure. The core of this JEP's goal was to encapsulate the majority of internal APIs. Specifically, most of the JDK's internal APIs are no longer accessible by default. Currently, internal APIs deemed to be *critical* and *widely-used* remain accessible. In the future, we are likely to see functionality to replace them, and at that time, those internal APIs will not be accessible by default.

So, why is this change necessary? There are a few widely-used APIs that are unstable and, in some cases, not standardized. Unsupported APIs should not have access to internal details of the JDK. Therefore, JEP-260 resulted in increased security of the Java platform. Generally speaking, you should not use unsupported APIs in your development projects.

The aforementioned critical APIs (internal to the JDK) are:

- sun.misc
- sun.misc.Unsafe
- sun.reflect.Reflection
- sun.reflect.ReflectionFactory.newConstructorForSerialization

The aforementioned critical internal APIs are still accessible in JDK 9. They will be accessible with the `jdk.unsupported` JDK module. Full JRE and JDK images will contain the `jdk.unsupported` module.



You can use the Java Dependency Analysis Tool, `jdeps`, to help determine if your Java program has any dependencies on JDK internal APIs.

This is an interesting change to watch. It is likely that the currently accessible internal APIs will not be accessible by default when Java 10 is released.

Summary

In this chapter, we examined the structure of Java modules as specified by *Project Jigsaw* and took an in-depth look at how *Project Jigsaw* was implemented to improve the Java platform. We also reviewed key internal changes to the Java platform as they relate to the modular system. Our review started with a modular primer where we learned about Java 9's modular system in terms of benefits and requirements.

We explored how Java 9 introduced modularity to the JDK including its source code and organization of the same. The seven primary tool categories that make up the JDK were also explored. As we learned, modularity in Java 9 also extends to runtime images resulting in more maintainability, better performance, and increased security. The concept of **link time** was introduced as an optional phase between compile-time and runtime. We concluded the chapter with a look at the Java Linker and how Java 9 encapsulates internal APIs.

In the next chapter, we will explore how to migrate our existing applications to the Java 9 platform. We will look at both manual and semi-automated migration processes.

Migrating Applications to Java 9

In the previous chapter, we took a close look at the structure of Java modules as specified by Project Jigsaw and examined how Project Jigsaw was implemented to improve the Java platform. We also reviewed key internal changes to the Java platform with specific focus on the new modular system. We started with a modular primer where we learned about Java 9's modular system in terms of benefits and requirements. Next, we explored how Java 9 introduced modularity to the JDK. This included a look at how the source code was reorganized for Java 9. We also explored the JDK's seven primary tool categories and learned that Java 9 modularity extends to runtime images resulting in more maintainability, better performance, and increased security. The concept of **link time** was introduced as an optional phase between compile-time and runtime. We concluded the chapter with a look at the **Java linker** and how Java 9 encapsulates internal APIs.

In this chapter, we will explore how to migrate our existing applications to the Java 9 platform. We will look at both manual and semi-automated migration processes. Java 9 is a major release with numerous changes to the JDK so developers should not be surprised if their Java 8 code no longer works with Java 9. This chapter aims to provide you with insights and processes to get your Java 8 code working with Java 9.

The topics we will cover in this chapter are:

- Quick review of Project Jigsaw
- How modules fit into the Java landscape
- Migration planning
- Advice from Oracle
- Useful tools

Quick review of Project Jigsaw

Project Jigsaw is the Java project that encompasses several change recommendations to the Java platform. As you have read in earlier chapters, Java 9's greatest changes involve modules and modularity. The initiative to move to modules in Java was driven by Project Jigsaw. The need for modularity stemmed from two major challenges with Java:

- Classpath
- JDK

Next, we will review both of those challenges and see how they were addressed and overcome with the new release to the Java platform, Java 9.

Classpath

Prior to Java 9, the classpath was problematic and the source of developer anguish. This was evident in the numerous developer forums and, fortunately, Oracle was paying attention. Here are the several instances in which the classpath can be problematic; here are two primary cases:

- The first case involves having two or more versions of a library on your development computer. The way this was previously handled by the Java system was inconsistent. Which library was used during the class loading process was anyone's guess. This resulted in an undesired lack of specificity--not enough details regarding which library was loaded.
- The second case is in exercising the most advanced features of the class loader. Often times, this type of class loader usage resulted in the most errors and bugs. These were not always easy to detect and resulted in a lot of extra work for developers.

Classpaths, before Java 9, were almost always very lengthy. Oracle, in a recent presentation, shared a classpath that contained 110 JAR files. This type of unwieldy classpath makes it difficult to detect conflicts or even determine if anything was missing and if so, what might be missing. The re-envisioning of the Java platform as a modular system made these classpath issues a thing of the past.



Modules solve the pre-Java 9 classpath problem by providing reliable configuration.

The monolithic nature of the JDK

Java has continually evolved in an impressive fashion since 1995 and with each evolutionary step, the JDK grew larger. As with Java 8, the JDK had become prohibitively large. Prior to Java 9, there were several problematic issues stemming from the monolithic nature of the JDK, including:

- Because the JDK is so large, it does not fit on very small devices. In some development sectors this is enough reason to find a non-Java solution for software engineering problems.
- The oversized JDK resulted in waste. It was wasteful in terms of processing and memory when running on devices, networks, and the cloud. This stems from the fact that the entire JDK is loaded, even when only a small subset of the JDK is required.
- While the Java platform has great performance when running, the startup performance, in terms of load and launch times, leaves much to be desired.
- The vast number of internal APIs has also been a pain point. Because so many internal APIs existed and were used by developers, the system has been difficult to evolve.
- The existence of internal APIs has made it difficult to make the JDK secure and scalable. With so many internal dependencies, isolating security and scalability issues has been overly problematic.

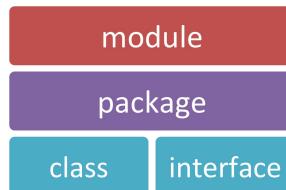
The answer to the monolithic woes of the JDK is the module. Java 9 introduced the module and its own modular system. One of the great updates to the platform is that only the modules needed are compiled, as opposed to the entire JDK. This modular system is covered throughout this book.



Modules solve the pre-Java 9 JDK monolithic issue by providing strong encapsulation.

How modules fit into the Java landscape

As you can see from the following illustration, packages are comprised of classes and interfaces, and modules are comprised of packages. Modules are a container of packages. This is the basic premise, at a very high level, of Java 9's new modular system. It is important to view modules as part of the modular system and not simply as a new level of abstraction above packages, as the illustration might suggest.



So, modules are new to Java 9 and they, as you would expect, require declaration before they can be used. A module's declaration includes names of other modules in which it has a dependency. It also exports packages for other modules that have dependencies to it. Modular declarations are arguably the most important modular issue to address as you start developing with Java 9. Here is an example:

```
module com.three19.irisScan
{
    // modules that com.three19.irisScan depends upon
    requires com.three19.irisCore;
    requires com.three19.irisData;

    // export packages for other modules that are dependent
    // upon com.three19.irisScan
    exports com.three19.irisScan.biometric;
}
```

When programming a Java 9 application, your module declarations will be placed in a `module-info.java` file. Once this file is completed, you simply run `javac`, the Java Compiler, to generate the `module-info.class` Java class file. You accomplish this task in the same manner that you currently compile your `.java` files into `.class` files.

You can also create modular JAR files that have your `module-info.class` file at its root. This represents a great level of flexibility.

Base module

When programming Java 9 applications, or porting existing applications programmed with older versions of Java, the base module (`java.base`) must be used. Every module requires the `java.base` module because it defines the critical, or foundational, Java platform APIs. Here are the contents of the `java.base` module:

```
module java.base
{
    exports java.io;
    exports java.lang;
    exports java.lang.annotation;
    exports java.lang.invoke;
    exports java.lang.module;
    exports java.lang.ref;
    exports java.lang.reflect;
    exports java.math;
    exports java.net;
    exports java.net.spi;
    exports java.nio;
    exports java.nio.channels;
    exports java.nio.channels.spi;
    exports java.nio.charset;
    exports java.nio.charset.spi;
    exports java.nio.file;
    exports java.nio.file.attribute;
    exports java.nio.file.spi;
    exports java.security;
    exports java.security.aci;
    exports java.security.cert;
    exports java.security.interfaces;
    exports java.security.spec;
    exports java.text;
    exports java.text.spi;
    exports java.time;
    exports java.time.chrono;
    exports java.time.format;
    exports java.time.temporal;
    exports java.time.zone;
    exports java.util;
    exports java.util.concurrent;
    exports java.util.concurrent.atomic;
    exports java.util.concurrent.locks;
    exports java.util.function;
    exports java.util.jar;
    exports java.util.regex;
    exports java.util.spi;
    exports java.util.stream;
    exports java.util.zip;
    exports java.crypto;
    exports java.crypto.interfaces;
    exports java.crypto.spec;
    exports java.net;
    exports java.net.ssi;
```

```
    exports java.security.auth;
    exports java.security.auth.callback;
    exports java.security.auth.login;
    exports java.security.auth.spi;
    exports java.security.auth.x500;
    exports java.security.cert;
}
```

As you can see, the `java.base` module does not require any modules and it exports numerous packages. It can be useful to have a list of these exports handy so you know what is available to you as you start creating applications using the new Java platform, Java 9.

You will notice that in the previous section, we did not include the `requires java.base;` line of code in our declaration of our `com.three19.irisScan` module. The updated code is provided as follows and now includes the `requires java.base;` line of code:

```
module com.three19.irisScan
{
    // modules that com.three19.irisScan depends upon
    requires java.base; // optional inclusion
    requires com.three19.irisCore;
    requires com.three19.irisData;

    // export packages for other modules that are dependent
    // upon com.three19.irisScan
    exports com.three19.irisScan.biometric;
}
```

If you do not include the `requires java.base;` line of code in your module declarations, the Java Compiler will automatically include it.

Reliable configuration

As suggested earlier in this chapter, modules provide a reliable configuration of our Java 9 applications that solves the classpath problem in earlier versions of the Java platform.

Java reads and interprets modular declarations making the modules readable. These readable modules permit the Java platform to determine if any modules are missing, if there are duplicate libraries declared, or there are any other conflicts. In Java 9, very specific error messages will be generated and output by the compiler or at runtime. Here is an example of a compile-time error:

```
| src/com.three19.irisScan/module-info.java: error: module not found: com.three19.iris:  
| requires com.three19.irisCore;  
| ^  
1 error
```

Here is an example of a runtime error that would occur if the module `com.three19.irisCore` was not found, but required by the `com.three19.irisScan` app:

```
| Error occurred during initialization of VM  
| java.lang.module.ResolutionException: Module com.three19.irisCore not found, required
```

Strong encapsulation

Earlier in this chapter, you read that Java 9's strong encapsulation remedied the monolithic JDK issue. Encapsulation, in Java 9, is driven by the information in the `module-info.java` file. The information in this file lets Java know what modules are dependent upon others and what each of them exports. This underscores the importance of ensuring our `module-info.java` files are properly configured. Let's look at an example written with standard Java code, nothing new in Java 9 in the way this was coded:

```
com.three19.irisScan          com.three19.access

package com.three19.irisScanner.internal;

public class irisScanResult {
    ...
}

package com.three19.access;

import com.three19.irisScanner.internal.irisScanResult;

public class Main {
    private irisScanResult scan1 = new irisScanResult();
    ...
}
```

In the preceding example, the `com.three19.irisScan` module has an `irisScanner` package intended for internal use and an `irisScanResult` class. If the `com.three19.access` application tries to import and use the `irisScanResult` class, the following error message will be produced by the Java Compiler:

```
| src/com.three19.access/com/three19/access/Main.java: error: irisScanResult is not ac
|   private irisSanResult scan1 = new irisScanResult();
|                                         ^
| 1 error
```

If for some reason the compiler does not catch this error, although it would be very unlikely, the following runtime error would occur:

```
| Exception in thread "main" java.lang.IllegalAccessError: class com.three19.access.Ma:
```

The detailed error messages will make debugging and troubleshooting much easier.

Migration planning

The changes to the Java platform are significant and Java 9 is considered a major release. It would be naive to think our current Java applications will work seamlessly on Java 9. While that might be the case, at least for simple programs, it is prudent to plan ahead and consider the issues you are most likely to encounter. Before we look at these issues, let's test a simple Java application in the next section.

Testing a simple Java application

The following code consists of a single Java class, `GeneratePassword`. This class prompts the user for a desired password length and then generates a password based on the user's requested length. If the user asks for a length shorter than 8, the default length of 8 will be used. This code was written with the Java SE 1.7 JRE System Library:

```
/*
 * This is a simple password generation app
 */

import java.util.Scanner;

public class GeneratePassword
{
    public static void main(String[] args)
    {
        // passwordLength int set up to easily change the schema
        int passwordLength = 8; //default value

        Scanner in = new Scanner(System.in);
        System.out.println("How long would you like your
                           password (min 8)?");
        int desiredLength;
        desiredLength = in.nextInt();

        // Test user input
        if (desiredLength >8)
        {
            passwordLength = desiredLength;
        }

        // Generate new password
        String newPassword = createNewPassword(passwordLength);

        // Prepare and provide output
        String output = "\nYour new " + passwordLength
                      + "-character password is: ";
        System.out.println(output + newPassword);
    }

    public static String createNewPassword(int lengthOfPassword)
    {
        // Start with an empty String
        String newPassword = "";

        // Populate password
        for (int i = 0; i < lengthOfPassword; i++)
        {
            newPassword = newPassword + randomizeFromSet(
                "aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ
                0123456789+-*/?@#$%&+");
        }
    }
}
```

```

        }

        return newPassword;
    }

    public static String randomizeFromSet(String characterSet)
    {
        int len = characterSet.length();
        int ran = (int) (len * Math.random());
        return characterSet.substring(ran, ran + 1);
    }
}

```

In the following screenshot, we test the `GeneratePassword` app on a Mac running Java 8. As you can see, we start by querying Java to verify the current version. In this test, Java `1.8.0_121` was used. Next, we compile the `GeneratePassword` Java file using the `javac` utility. Lastly, we run the app:



```

Edwards-iMac:~ edljr$ java -version
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
Edwards-iMac:~ edljr$
Edwards-iMac:~ edljr$
Edwards-iMac:~ edljr$ javac GeneratePassword.java
Edwards-iMac:~ edljr$
Edwards-iMac:~ edljr$
Edwards-iMac:~ edljr$ java GeneratePassword
How long would you like your password (min 8)?
32

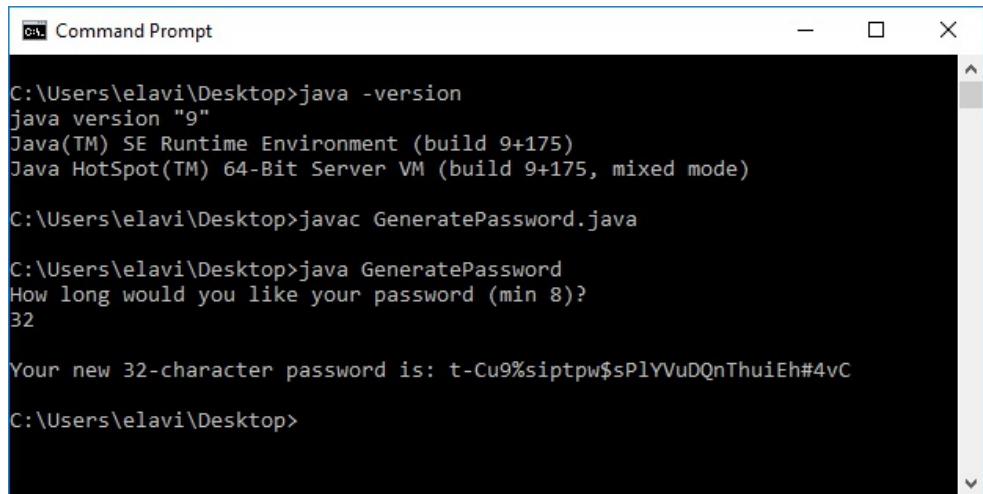
Your new 32-character password is: B#CZy0z1Mq0WI@dkFFiuG9BrHw$w9KFg
Edwards-iMac:~ edljr$

```

As you can see from the preceding test, `GeneratePassword.java` was successfully compiled with the `GeneratePassword.class` file resulting. The application was run using the `java GeneratePassword` command. The user was prompted for a desired password length and `32` was entered. The application then successfully generated a 32-character random password and provided the appropriate output.

This test demonstrated the example application works successfully using JDK 1.8. Next, let's test the same application using JDK 9.

We start with the `java -version` command to show that we are using JDK 9 on this computer. The following screenshot shows that we successfully compiled the `.java` file to a `.class` file. When the application was run, it functioned as expected and provided the proper results:



```
C:\Users\elavi\Desktop>java -version
java version "9"
Java(TM) SE Runtime Environment (build 9+175)
Java HotSpot(TM) 64-Bit Server VM (build 9+175, mixed mode)

C:\Users\elavi\Desktop>javac GeneratePassword.java

C:\Users\elavi\Desktop>java GeneratePassword
How long would you like your password (min 8)?
32

Your new 32-character password is: t-Cu9%siptpw$sPlYVuDQnThuiEh#4vC

C:\Users\elavi\Desktop>
```

As you can see, we clearly demonstrated that a pre-Java 9 application has the potential to successfully run on Java 9 without having to make any modifications. This is a simple case study and featured a very basic Java program. This is, of course, the best case scenario, and cannot be assumed. You will want to test your applications to ensure they run as expected on the Java 9 platform.

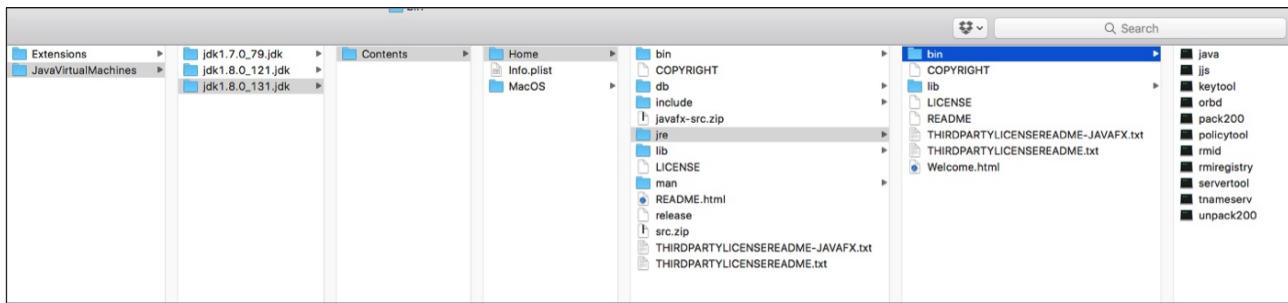
In the next section, we will review some potential issues you might encounter when testing your pre-Java 9 applications using the new Java platform with JDK 9.

Potential migration issues

The potential migration issues featured in this section include direct access to the JRE, access to internal APIs, accessing internal JARs, JAR URL depreciation, the extension mechanism, and the JDK's modularization. Let's look at each of these potential migration issues.

The JRE

Creating the Java 9's modular system resulted in some simplification in respect to the number and location of development and utility tools. One such example is the JDK's consumption of the JRE. In all pre-Java 9 versions, the Java platform included the JDK and JRE as two separate components. In Java 9, these components have been combined. This is a significant change and one that developers should be keenly aware of. If you have an application that specifically points to the JRE directory, you will need to make changes to avoid problems. The JRE contents are shown as follows:



Access to internal APIs

The Java 9 platform has encapsulated internal APIs to increase security of the platform and applications written in Java. Applications that you program in Java 9 will not have default access to the JDK's internal APIs, unlike with previous versions of the Java platform. Oracle has identified some internal APIs as critical; those APIs remain accessible via the `jdk.unsupported` JDK module.

The aforementioned critical APIs (internal to the JDK) are:

- `sun.misc`
- `sun.misc.Unsafe`
- `sun.reflect.Reflection`
- `sun.reflect.ReflectionFactory.newConstructorForSerialization`

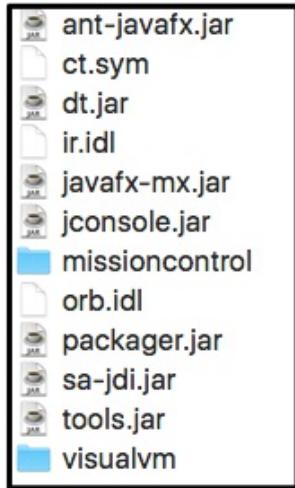
If you have pre-Java 9 applications that implement any `sun.*` or `com.sun.*` package, you will likely run into problems migrating your applications to Java 9. In order to address this issue, you should review your class files for use of `sun.*` and `com.sun.*` packages. Alternatively, you can use the Java dependency analysis tool, `jdeps`, to help determine if your Java program has any dependencies on JDK internal APIs.



The `jdeps` tool is the Java dependency analysis tool, that can be used to help determine if your Java program has any dependencies on JDK internal APIs.

Accessing internal JARs

Java 9 does not permit access to internal JARs such as `lib/ant-javax.jar`, `lib/dt.jar`, and others listed in the `lib` directory shown here:



The key thing to note here is that if you have Java applications that are dependent on one of these tools residing in the `lib` folder, you will need to modify your code accordingly.



It is recommended that you test your IDE once you start using Java 9 to ensure the IDE is updated and officially supports Java 9. If you use more than one IDE for Java development, test each one to avoid surprises.

JAR URL depreciation

JAR file URLs were, prior to Java 9, used by some APIs to identify specific files in the runtime image. These URLs contain a `jar:file:` prefix with two paths; one to the `jar` and one to the specific resource file within the `jar`. Here is the syntax for the pre-Java 9 JAR URL:

```
|   jar:file:<path-to-jar>!<path-to-file-in-jar>
```

With the advent of Java 9's modular system, containers will house resource files instead of individual JARs. The new syntax for accessing resource files is as follows:

```
|   jrt:/<module-name>/<path-to-file-in-module>
```

A new URL schema, `jrt`, is now in place for naming resources within a runtime image. These resources include classes and modules. The new schema allows for the identification of a resource without introducing a security risk to the runtime image. This increased security ensures that the runtime image's form and structure remain concealed. Here is the new schema:

```
|   jrt:/[$MODULE[/$PATH]]
```

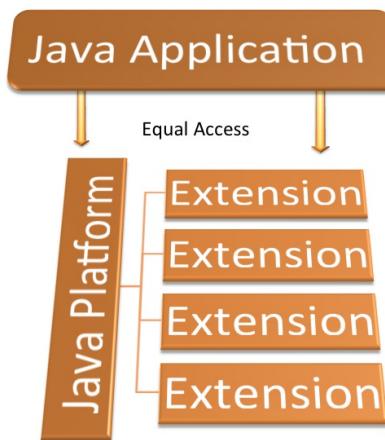
Interestingly, a `jrt` URL's structure determines its meaning, suggesting that the structure can take one of several forms. Here are three examples of different `jrt` URL structures:

- `jrt:/$MODULE/$PATH`: This structure provides access to the resource file, identified with the `$PATH` parameter, within the module specified with the `$MODULE` parameter
- `jrt:/$MODULE`: This structure provides reference to all resource files within the module specified with the `$MODULE` parameter
- `jrt:/`: This structure provides reference to all resource files in the runtime image

If you have preexisting code that uses URL instances, returned by APIs, you should not have any problems. On the other hand, if your code is dependent on the `jar` URL structure, you will have problems.

Extension mechanism

The Java platform previously had an extension mechanism that gave developers the ability to make custom APIs available to all applications. As you can see in the following illustration, extensions are plugins of sorts, or add-ons to the Java platform. The APIs and classes in each extension are, by default, automatically available:



As the illustration suggests, Java applications have access both to the Java platform and extensions without requiring classpaths. This feature was deprecated in Java 8 and no longer exists in Java 9.

The JDK's modularization

By now, you have a firm appreciation of Java 9's modularization. The old adage in Java, and other object-oriented programming language, is *everything is a class*. Now, with Java 9, *everything is a module* is the new adage. There are three type of modules as explained as follows:

Module type	Description
Automatic	When a JAR is placed on a new module path, modules are automatically created
Explicit/Named	These modules are manually defined by editing the <code>module-info.java</code> file
Unnamed	When a JAR is placed on a classpath, unnamed modules are created

When you migrate your applications to Java 9, your application and its libraries become unnamed modules. So, you will need to ensure all the modules are in the module path.

Another thing to be aware of is that your runtime image will not contain the entire JDK. Instead, it will only contain the modules your application requires. It is worth reviewing how the JDK is modularized in Java 9. The following table contains the API specification for the JDK in Java 9:

<code>jdk.accessibility</code>	<code>jdk.attach</code>	<code>jdk.charsets</code>	<code>jdk.compiler</code>
<code>jdk.crypto.cryptoki</code>	<code>jdk.crypto.ec</code>	<code>jdk.dynalink</code>	<code>jdk.editpad</code>
<code>jdk.hotspot.agent</code>	<code>jdk.httpserver</code>	<code>jdk.incubator.httpclient</code>	<code>jdk.jartool</code>
<code>jdk.javadoc</code>	<code>jdk.jcmd</code>	<code>jdk.jconsole</code>	<code>jdk.jdeps</code>
<code>jdk.jdi</code>	<code>jdk.jdwp.agent</code>	<code>jdk.jlink</code>	<code>jdk.jshell</code>
<code>jdk.jsobject</code>	<code>jdk.jstard</code>	<code>jdk.localedata</code>	<code>jdk.manageme</code>
<code>jdk.management.agent</code>	<code>jdk.naming.dns</code>	<code>jdk.naming.rmi</code>	<code>jdk.net</code>
<code>jdk.pack</code>	<code>jdk.packager.services</code>	<code>jdk.policytool</code>	<code>jdk.rmic</code>

jdk.scripting.nashorn	jdk.sctp	jdk.security.auth	jdk.security
jdk.snmp	jdk.xml.dom	jdk.zipfs	

The following table contains the API specification for Java SE in Java 9:

java.activation	java.base	java.compiler	java.cobra
java.datatransfer	java.desktop	java.instrument	java.logging
java.management	java.management.rmi	java.naming	java.prefs
java.rmi	java.scripting	java.se	java.se.ee
java.security.jgss	java.security.sasl	java.sql	java.sql.rowset
java.transaction	java.xml	java.xml.bind	java.xml.crypto
java.xml.ws	java.xml.ws	java.xml.ws.annotation	



Remember, all applications will have access to `java.base` as it is in the module path by default.

The following table contains the API specification for JavaFX in Java 9:

javafx.base	javafx.controls	javafx.fxml	javafx.graphics
javafx.media	javafx.swing	javafx.web	

There are two additional modules:

- `java.jnlp` defines the API for **JNLP (Java Network Launch Protocol)**
- `java.smartcardio` defines the API for the Java Smart Card Input/Output



For details on any of these modules, visit Oracle's Java® Platform, Standard Edition & Java Development Kit Version 9 API Specification website: <http://download.java.net/java/jdk9/docs/api/overview-summary.html>.

Advice from Oracle

Oracle has done a great job in bringing us this major update, version 9, to the Java platform. Their insights into getting ready for Java 9 and how to migrate to the new JDK is worth reviewing. In this section, we will look at preparatory steps, breaking encapsulation, changes to the runtime image, components such as tools and APIs that have been removed, changes to garbage collection, and deployment.

Preparatory steps

Oracle provides a five-step process to help developers migrate their Java applications to version 9. These steps are listed as follows and then covered in subsequent sections:

1. Get the JDK 9 early access build.
2. Run your program before recompiling.
3. Update third-party libraries and tools.
4. Compile your application.
5. Run `jdeps` on your code.

Getting the JDK 9 early access build

If you are reading this book before Java 9 is officially released, then you can obtain a JDK 9 early access build from here--<http://jdk.java.net/9/>. Early release builds are available for Windows (32 and 64), macOS (64), Linux (32 and 64) and various Linux ARM, Solaris, and Alpine Linux versions.

Taking the time to test your applications for Java 9 and get them migrated before Java 9 is officially released, helps ensure you will not experience any downtime for services that rely on your Java applications.

Running your program before recompiling

As indicated earlier in this chapter, there is a chance that your existing Java applications will run without modification on the Java 9 platform. So, before you make any changes, try running your current application on the Java 9 platform. If your application works fine on Java 9, that is great, but your work is not complete. Review the next three sections on updating third-party libraries and tools, compiling your application, and running `jdeps` on your code.

Updating third-party libraries and tools

Third-party libraries and tools can help extend our applications and shorten development time. For Java 9 compatibility, it is important to ensure that each third-party library and tool you use is compatible with and supports version 9 of the JDK. Running your application on Java 9 will not provide you with the level of insight you need to ensure you do not have compatibility issues down the road. It is recommended that you review the official website for each library and tool to verify compatibility with and support of JDK 9.

If a library or tool that you use does have a version that supports JDK 9, download and install it. If you find one that does not yet support JDK 9, consider finding a replacement for it.

In our context, tools includes **Integrated Development Environments (IDE)**. NetBeans, Eclipse, and IntelliJ all have IDE versions that support JDK 9. Links to those sites are provided as follows:

- **NetBeans:** <http://bits.netbeans.org/download/trunk/nightly/latest/>
- **Eclipse:** https://www.eclipse.org/community/eclipse_newsletter/2015/june/article4.php
- **IntelliJ:** <https://www.jetbrains.com/idea/nextversion/>

Compiling your application

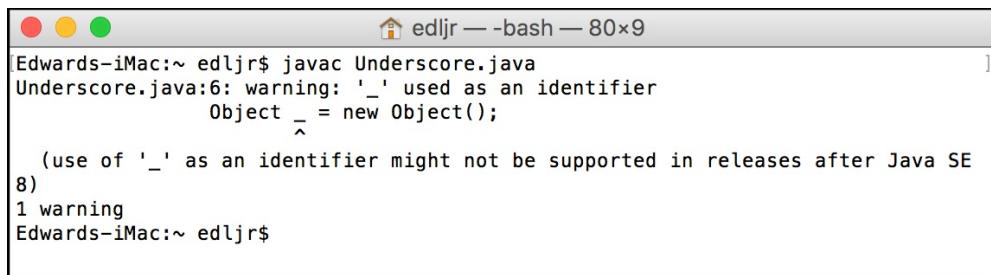
Your next step is to compile your application using JDK 9's `javac`. This is important, even if your app works fine on JDK 9. You might not receive compiler errors, but watch for warnings too. Here are the most common reasons your applications might not compile with JDK 9, assuming they compiled fine prior to Java 9.

First, as indicated earlier in this chapter, most of the JDK 9 internal APIs are not accessible by default. Your indication will be an `IllegalAccessException` error at runtime or compile time. You will need to update your code so that you are using accessible APIs.

A second reason your pre-Java 9 applications might not compile with JDK 9 is if you use the underscore character as a single character identifier. According to Oracle, this practice generates a warning in Java 8 and an error in Java 9. Let's look at an example. The following Java class instantiates an Object named `_` and prints a singular message to the console:

```
public class Underscore
{
    public static void main(String[] args)
    {
        Object _ = new Object();
        System.out.println("This ran successfully.");
    }
}
```

When we compile this program with Java 8, we receive a warning that use of `'_'` as an identifier might not be supported in releases after Java SE 8:



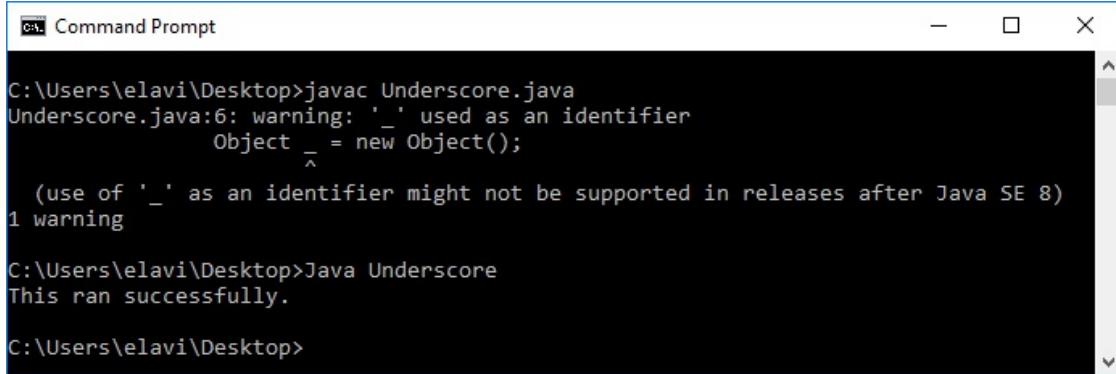
```
edwards-iMac:~ edljr$ javac Underscore.java
Underscore.java:6: warning: '_' used as an identifier
Object _ = new Object();
          ^
(use of '_' as an identifier might not be supported in releases after Java SE
8)
1 warning
Edwards-iMac:~ edljr$
```

As you can see in the following screenshot, that is just a warning and the application runs fine:



```
Edwards-iMac:~ edljr$ java Underscore
This ran successfully.
Edwards-iMac:~ edljr$
```

Now, let's try compiling the same class using JDK 9:



```
C:\Users\elavi\Desktop>javac Underscore.java
Underscore.java:6: warning: '_' used as an identifier
        Object _ = new Object();
                  ^
(use of '_' as an identifier might not be supported in releases after Java SE 8)
1 warning

C:\Users\elavi\Desktop>Java Underscore
This ran successfully.

C:\Users\elavi\Desktop>
```

As you can see, use of the underscore as a single character identifier still only resulted in a warning and not an error. The application ran successfully. This test was run when JDK 9 was still in early release. It is assumed that running this test once JDK 9 has been officially released will result in an error instead of just a warning. The error that would likely be thrown is as follows:

```
| Underscore.java:2: error: as of release 9, '_' is a keyword, and may not be used as :
```

Even if this issue is not resolved with the formal release of JDK 9, use of an underscore as a single character identifier is not good programming practice, so you should steer away from using it.

A third potential reason for your pre-Java 9 programmed application not to compile with JDK 9 is if you are using the `-source` and `-target` compiler options. Let's take a look at the `-source` and `-target` compiler options pre-Java 9 and with Java 9.

Pre-Java 9 -source and -target options

The `-source` option specifies the Java SE version and has the following acceptable values:

Value	Description
1.3	<code>javac</code> does not support features introduced after Java SE 1.3.
1.4	<code>javac</code> accepts code with language features introduced in Java SE 1.4.
1.5 or 5	<code>javac</code> accepts code with languages features introduced in Java SE 1.5.
1.6 or 6	<code>javac</code> reports encoding errors as errors instead of warnings. Of note, no new language features were intruded with Java SE 1.6.
1.7 or 7	<code>javac</code> accepts code with language features introduced in Java SE 1.7. This is the default value if the <code>-source</code> option is not used.

The `-target` option tells `javac` what version of the JVM to target. The acceptable values for the `-target` option are--1.1, 1.2, 1.3, 1.4, 1.5 or 5, 1.6 or 6 and 1.7 or 7. If the `-target` option is not used, the default JVM target is dependent on the value used with the `-source` option. Here is a table of `-source` values with their associated `-target`:

-source value	default -target
unspecified	1.7
1.2	1.4
1.3	1.4
1.4	1.4
1.5 or 5	1.7
1.6 or 6	1.7
1.7	1.7

Java 9 -source and -target options

In Java 9, the supported values are shown as follows:

Supported values	Remarks
9	This is the default, should no value be specified
8	Sets support to 1.8
7	Sets support to 1.7
6	Sets support to 1.6 and generates a warning (not an error) to indicate JDK 6 is deprecated

Running jdeps on your code

The `jdeps` class dependency analysis tool is not new to Java 9, but perhaps has never been as important to developers with the advent of Java 9. An important step to migrating your applications to Java 9 is to run the `jdeps` tool to determine the dependencies your applications and its libraries have. The `jdeps` tool does a great job of suggesting replacements if your code has dependencies on any internal APIs.

The following screenshot shows the options available to you when using the `jdeps` analyzer:

```
C:\ Command Prompt
C:\Users\elavi\Desktop>jdeps -help
Usage: jdeps <options> <classes...>
where <classes> can be a pathname to a .class file, a directory, a JAR file,
or a fully-qualified class name. Possible options include:
  -dotoutput <dir>          Destination directory for DOT file output
  -s                  -summary          Print dependency summary only
  -v                  -verbose           Print all class level dependencies
                                         Equivalent to -verbose:class -filter:none.
  -verbose:package      Print package-level dependencies excluding
                                         dependencies within the same package by default
  -verbose:class        Print class-level dependencies excluding
                                         dependencies within the same package by default
  -cp <path>  -classpath <path>    Specify where to find class files
  -p <pkgname> -package <pkgname>   Finds dependences matching the given package name
                                         (may be given multiple times)
  -e <regex>  -regex <regex>       Finds dependences matching the given pattern
                                         (-p and -e are exclusive)
  -f <regex>  -filter <regex>      Filter dependences matching the given pattern
                                         If given multiple times, the last one will be used.
  -filter:package      Filter dependences within the same package (default)
  -filter:archive      Filter dependences within the same archive
  -filter:none         No -filter:package and -filter:archive filtering
                                         Filtering specified via the -filter option still applies.
  -include <regex>     Restrict analysis to classes matching pattern
                                         This option filters the list of classes to
                                         be analyzed. It can be used together with
                                         -p and -e which apply pattern to the dependences
  -P                  -profile          Show profile or the file containing a package
  -apionly            Restrict analysis to APIs i.e. dependences
                                         from the signature of public and protected
                                         members of public classes including field
                                         type, method parameter types, returned type,
                                         checked exception types etc
  -R                  -recursive        Recursively traverse all dependencies.
                                         The -R option implies -filter:none. If -p, -e, -f
                                         option is specified, only the matching dependences
                                         are analyzed.
  -jdkinternals       Finds class-level dependences on JDK internal APIs.
                                         By default, it analyzes all classes on -classpath
                                         and input files unless -include option is specified.
                                         This option cannot be used with -p, -e and -s options.
                                         WARNING: JDK internal APIs may not be accessible in
                                         the next release.
  -version            Version information

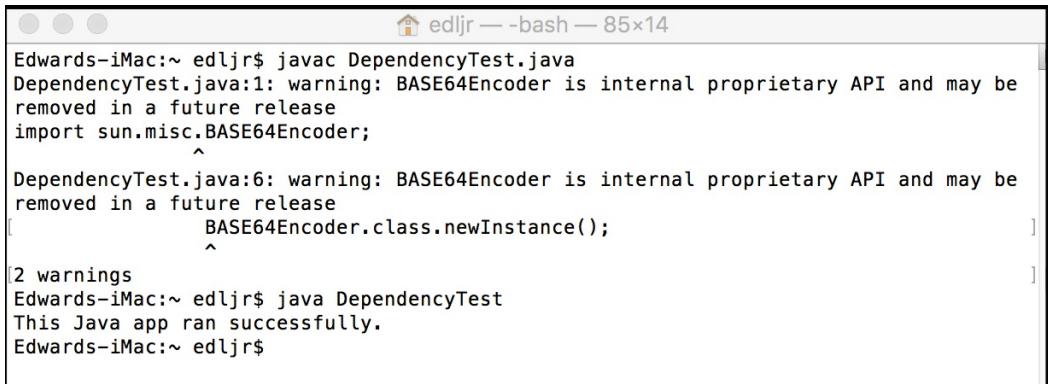
C:\Users\elavi\Desktop>
```

Let's take a look at an example. Here is a simple Java class called `DependencyTest`:

```
import sun.misc.BASE64Encoder;

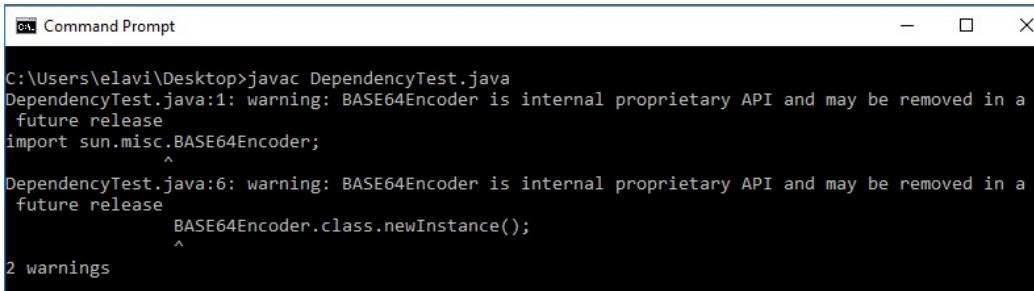
public class DependencyTest
{
    public static void main(String[] args) throws
        InstantiationException, IllegalAccessException
    {
        BASE64Encoder.class.newInstance();
        System.out.println("This Java app ran successfully.");
    }
}
```

Now, let's use `javac` to compile this class using Java 8:



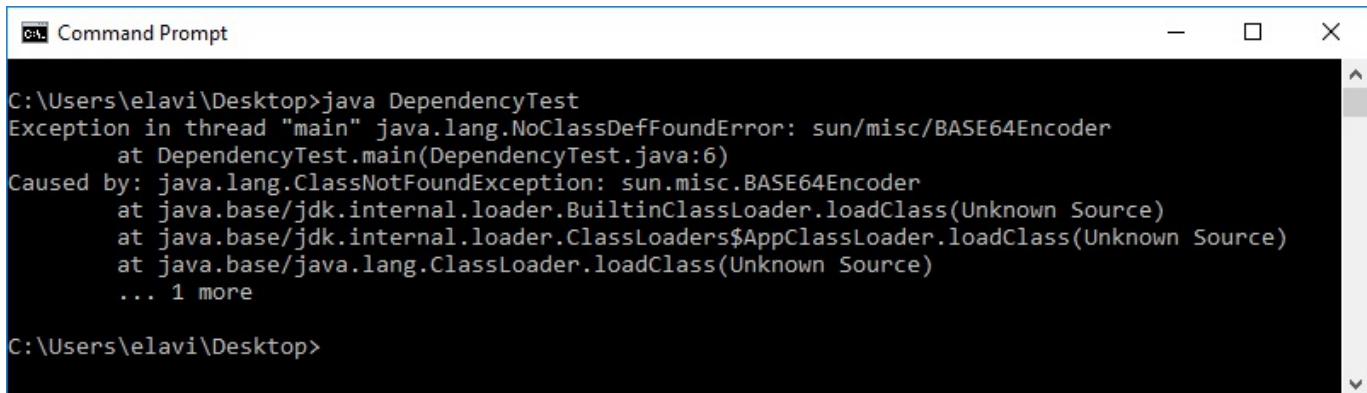
```
edljr ~ -bash — 85x14
Edwards-iMac:~ edljr$ javac DependencyTest.java
DependencyTest.java:1: warning: BASE64Encoder is internal proprietary API and may be
removed in a future release
import sun.misc.BASE64Encoder;
^
DependencyTest.java:6: warning: BASE64Encoder is internal proprietary API and may be
removed in a future release
[     BASE64Encoder.class.newInstance();
^
]2 warnings
Edwards-iMac:~ edljr$ java DependencyTest
This Java app ran successfully.
Edwards-iMac:~ edljr$
```

As you can see, Java 8 successfully compiled the class and the application ran. The compiler did give us a `DependencyTest.java:6: warning: BASE64Encoder is internal proprietary API and may be removed in a future release` warning. Now, let's see what happens when we try to compile this class using Java 9:



```
Command Prompt
C:\Users\elavi\Desktop>javac DependencyTest.java
DependencyTest.java:1: warning: BASE64Encoder is internal proprietary API and may be removed in a
future release
import sun.misc.BASE64Encoder;
^
DependencyTest.java:6: warning: BASE64Encoder is internal proprietary API and may be removed in a
future release
[     BASE64Encoder.class.newInstance();
^
]2 warnings
```

In this case, with Java 9, the compiler gave us two warnings instead of one. The first warning is for the `import sun.misc.BASE64Encoder;` statement and the second for the `BASE64Encoder.class.newInstance();` method call. As you can see, these are just warnings and not errors, so the `DependencyTest.java` class file is successfully compiled. Next, let's run the application:

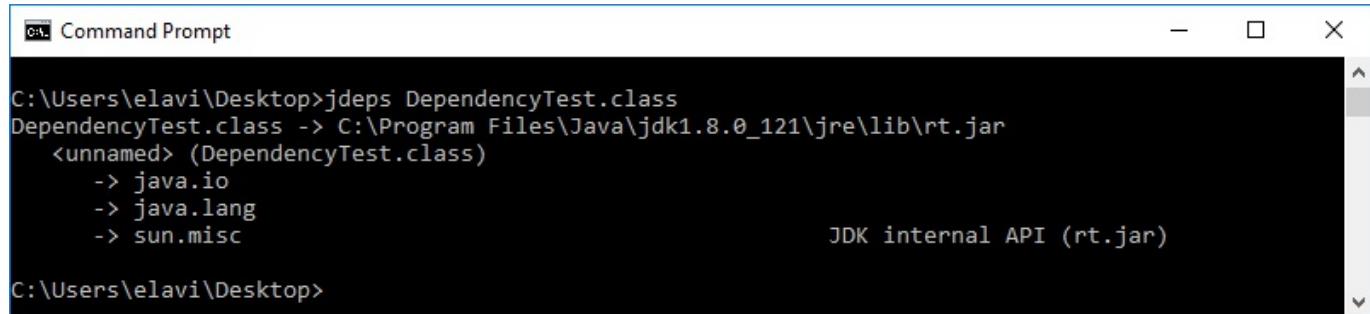


```
Command Prompt
C:\Users\elavi\Desktop>java DependencyTest
Exception in thread "main" java.lang.NoClassDefFoundError: sun/misc/BASE64Encoder
        at DependencyTest.main(DependencyTest.java:6)
Caused by: java.lang.ClassNotFoundException: sun.misc.BASE64Encoder
        at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(Unknown Source)
        at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(Unknown Source)
        at java.base/java.lang.ClassLoader.loadClass(Unknown Source)
        ... 1 more

C:\Users\elavi\Desktop>
```

Now, we can clearly see that Java 9 will not allow us to run the application. Next,

let's run a dependency test using the `jdeps` analyzer tool. We will use the following command line syntax--`jdeps DependencyTest.class`:



```
Command Prompt

C:\Users\elavi\Desktop>jdeps DependencyTest.class
DependencyTest.class -> C:\Program Files\Java\jdk1.8.0_121\jre\lib\rt.jar
 <unnamed> (DependencyTest.class)
   -> java.io
   -> java.lang
   -> sun.misc                               JDK internal API (rt.jar)

C:\Users\elavi\Desktop>
```

As you can see, we have three dependencies: `java.io`, `java.lang`, and `sun.misc`. Here we are given the suggestion to replace our `sun.misc` dependency with `rt.jar`.

Breaking encapsulation

The Java 9 platform is more secure than its predecessor versions due to, in part, the increased encapsulation that resulted from the modular reorganization. That being said, you might have a requirement to break through the modular system's encapsulation. Breaking through these access control boundaries is permitted by Java 9.

As you read earlier in this chapter, most internal APIs are strongly encapsulated. As previously suggested, you might look for replacement APIs when updating your source code. Of course, that is not always feasible. There are three additional approaches you can take--using the `--add-opens` option at runtime; employing the `--add-exports` option; and `--permit-illegal-access` command-line option. Let's look at each of those options.

The --add-opens option

You can use the `--add-opens` runtime option to allow your code to access non-public members. This can be referred to as deep reflection. Libraries that do this deep reflection are able to access all members, private and public. To grant this type of access to your code, you use the `--add-opens` option. Here is the syntax:

```
|   --add-opens module/package=target-module(,target-module)*
```

This allows the given module to open the specified package. The compiler will not produce any errors or warnings when this is used.

The --add-exports option

You can use `--add-exports` to break encapsulation so that you can use an internal API whose default is to be inaccessible. Here is the syntax:

```
| --add-exports <source-module>/<package>=<target-module>(  
|   ,<target-module>)*
```

This command-line option gives code in the `<target-module>` access to types in the `<source-module>` package.

Another method of breaking encapsulation is with a JAR file's manifest. Here is an example:

```
| --add-exports:java.management/sun.management
```

 *The `--add-exports` command-line option should only be used if deemed absolutely necessary. It is not advisable to use this option except for short-term solutions. The danger of using it routinely is that any updates to referenced Internal APIs could result in your code not working properly.*

The --permit-illegal-access option

A third option for breaking encapsulation is to use the `--permit-illegal-access` option. Of course, it is prudent to check with third-party library creators to see if they have an updated version. If that is not an option, you use `--permit-illegal-access` to gain illegal access to operations to be implemented on the classpath. Due to the significantly illegal operation here, you will receive warnings each time one of these operations occurs.

Runtime image changes

Java 9 represents a major change to the JDK and the JRE. Much of these changes are related to modularity and have been covered in other chapters. There are still a few more things you should consider.

Java version schema

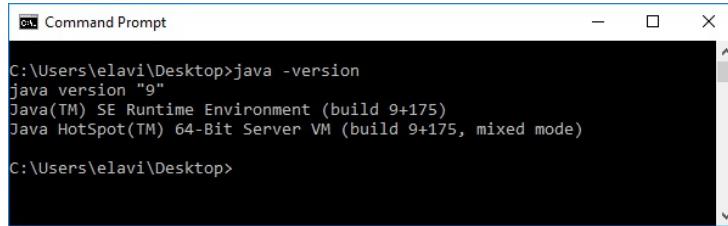
With Java 9, the way the Java platform's version is displayed has changed. Here is an example of a pre-Java 9 version format:



A screenshot of a terminal window titled "edljr -- bash -- 65x8". The window shows the output of the command "java -version". The output is as follows:

```
Last login: Sat Jul 15 15:04:45 on ttys000
Edwards-iMac:~ edljr$ java -version
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
Edwards-iMac:~ edljr$
```

Now, let's look at how Java 9 reports its version:



A screenshot of a Command Prompt window titled "Command Prompt". The window shows the output of the command "java -version". The output is as follows:

```
C:\Users\elavi\Desktop>java -version
java version "9"
Java(TM) SE Runtime Environment (build 9+175)
Java HotSpot(TM) 64-Bit Server VM (build 9+175, mixed mode)

C:\Users\elavi\Desktop>
```

As you can see, with Java 9, the version schema is now `$MAJOR.$MINOR.$SECURITY.$PATCH`. This is markedly different than with previous versions of Java. This will only impact your applications if you have code that parses the string returned by the `java -version` command and option.

JDK and JRE layout

How files are organized in the JDK and the JRE have changed in the new version of Java. It is worth your time to familiarize yourself with the new filesystem layout. The following screenshot shows the file structure of the JDK's `/bin` folder:

The screenshot shows a Windows Command Prompt window with the title "Command Prompt - dir /s/w/p". The current directory is "C:\Program Files\Java\jdk-9\bin". The output lists numerous files and sub-directories, including Java executables like java.exe, javac.exe, and javadoc.exe, as well as various DLLs and system files. The output is as follows:

```
Volume in drive C is OS
Volume Serial Number is 600F-FF3F

Directory of C:\Program Files\Java\jdk-9\bin

[...]          [...]          ...
appletviewer.exe      attach.dll
aut.dll             bci.dll
dcrn.dll            decorasse.dll
deploy.dll           [dtplugin]
dt_shmem.dll         dt_socket.dll
eula.dll            fontmanager.dll
fplugins.dll        glass.dll
giz-lite.dll        gstreamer-lite.dll
i18n.dll            instck11.dll
j2DSC.dll           jpkc11.dll
jas_nt.dll          jabswitch.exe
jaccessinspector.exe jaccesswalker.exe
jar.exe              jarsigner.exe
java.dll            java.exe
javaccessbridge.dll javac.exe
javapl.exe          javadoc.exe
javafx_font.dll    javafx_font_t2k.dll
javafx_iio.dll     javah.exe
javajpeg.dll       javap.exe
javapackager.exe   javaw.exe
javaws.exe          jaut.dll
jdb.exe             jconsole.exe
jdeps.exe           jdeprscn.exe
jmedia.dll          jdpwd.dll
jsdb.exe            jfxwebkit.dll
image.exe           jimage.dll
js.exe               jinfo.exe
jlink.exe           jli.dll
jmod.exe            jmap.exe
jp2native.dll       jp2ssv.dll
jps.exe              jrunscript.exe
jse.exe              jsound.dll
jounds.dll          jstack.exe
jstat.exe           jstard.exe
jublauncher.exe    kcmsg.dll
keytool.exe         kinit.exe
list.exe            ktab.exe
lcms.dll           le.dll
management.dll     management_agent.dll
management_ext.dll mlib_image.dll
msvcpi20.dll        msrvr120.dll
net.dll             nio.dll
orbd.exe            pack200.exe
[plugin2]           policytool.exe
pref.dll            prism_common.dll
prism_d3d.dll      prism_w.dll
rmi.dll             rmid.exe
rmid.exe            rmiregistry.exe
swindbg.dll         schemagen.exe
serialver.exe       [server]
servicetool.exe    splashscreen.dll
svagent.exe         sunec.dll
sumscapi.dll        t2k.dll
tameserv.exe        unpack.dll
unpack200.exe       verify.dll
w2k_lsa_auth.dll   windowsaccessbridge-64.dll
wsdetect.dll        wsigen.exe
wimport.exe         xjc.exe
zip.dll             116 File(s)      56,430,864 bytes

Directory of C:\Program Files\Java\jdk-9\bin\dtplugin
[...]          [...]          ...
npdeployJava1.dll   deployJava1.dll
2 File(s)        2,200,872 bytes

Directory of C:\Program Files\Java\jdk-9\bin\plugin2
[...]          [...]          ...
3 File(s)        msvcp120.dll  msvcr120.dll  npjp2.dll
1,841,000 bytes

Directory of C:\Program Files\Java\jdk-9\bin\server
[...]          [...]          ...
1 File(s)        jvm.dll      10,332,712 bytes

Total Files Listed:
 122 File(s)      70,814,448 bytes
 11 Dir(s)        896,921,709,568 bytes free
```

Here is the layout of the `\lib` folder:

```
cmd Command Prompt - dir /s/w/p
Volume in drive C is OS
Volume Serial Number is 608F-FF3F

Directory of C:\Program Files\Java\jdk-9\lib

[.]          [..]
ant-javafx.jar      classlist
ct.sym            [deploy]
deploy.jar        fontconfig.bfc
fontconfig.properties.src [fonts]
java.jnlp.jar     javacl.cpl
javafx-swt.jar    javafx.properties
javaws.jar       jdk.deploy.jar
jdk.javaws.jar   jdk.plugin.dom.jar
jdk.plugin.jar   jrt-fs.jar
jvm.cfg          jvm.lib
modules          plugin-legacy.jar
plugin.jar       psfont.properties.ja
psfontjd.properties sawindbg.dll.manifest
[security]        [server]
src.zip          tzdb.dat
tzmappings

27 File(s)    249,507,799 bytes

Directory of C:\Program Files\Java\jdk-9\lib\deploy

[.]          [..]
messages.properties  messages_de.properties
messages_es.properties messages_fr.properties
messages_it.properties messages_ja.properties
messages_ko.properties messages_pt_BR.properties
messages_sv.properties messages_zh_CN.properties
messages_zh_HK.properties messages_zh_TW.properties
splash.gif

13 File(s)    60,861 bytes

Directory of C:\Program Files\Java\jdk-9\lib\fonts

[.]          [..]
LucidaBrightDemiBold.ttf  LucidaBrightDemiItalic.ttf
LucidaBrightItalic.ttf    LucidaBrightRegular.ttf
LucidaSansDemiBold.ttf   LucidaSansRegular.ttf
LucidaTypewriterBold.ttf LucidaTypewriterRegular.ttf
8 File(s)    2,068,932 bytes

Directory of C:\Program Files\Java\jdk-9\lib\security

[.]          [..]          blacklist
blacklisted.certs    cacerts      default.policy
public_suffix_list.dat trusted.libraries
6 File(s)    254,000 bytes

Directory of C:\Program Files\Java\jdk-9\lib\server

[.]          [..]          xusage.txt
1 File(s)    1,383 bytes

Total Files Listed:
 55 File(s)  251,892,975 bytes
 14 Dir(s)  890,920,189,952 bytes free

C:\Program Files\Java\jdk-9\lib>
```

What has been removed

Another area of change for the new version of the Java platform is that many platform components have been removed. The following sections represent the most significant components.

Notably, the `rt.jar` and `tools.jar` and `dt.jar` have been removed. These JAR files contained class and other resources files and all resided in the `/lib` directory.

The *endorsed standards override mechanism* has been removed. In Java 9, both `javac` and `java` will exit if they detect that mechanism. The mechanism was used for application servers to override some JDK components. In Java 9, you can use upgradeable modules to achieve the same result.

As previously covered in this chapter, the *extension mechanism* has also been removed.

The following listed APIs were previously deprecated and have been removed and are not accessible in Java 9. Removal of these APIs is the result of the modularization of the Java platform:

- `apple.applescript`
- `com.apple.concurrent`
- `com.sun.image.codec.jpeg`
- `java.awt.dnd.peer`
- `java.awt.peer`
- `java.rmi.server.disableHttp`
- `java.util.logging.LogManager.addPropertyChangeListener`
- `java.util.logging.LogManager.removePropertyChangeListener`
- `java.util.jar.Packer.addPropertyChangeListener`
- `java.util.jar.Packer.removePropertyChangeListener`
- `java.util.jar.Pack200.Unpacker.addPropertyChangeListener`
- `java.util.jar.Pack200.Unpacker.removePropertyChangeListener`
- `javax.management.remote.rmi.RMIIOPServerImpl`
- `sun.misc.BASE64Encoder`
- `sun.misc.BASE64Decoder`
- `sun.rmi.transport.proxy.connectTimeout`

- sun.rmi.transport.proxy.eagerHttpFallback
- sun.rmi.transport.proxy.logLevel
- sun.rmi.transport.tcp.proxy

The following listed tools have been removed. In each case, the tool was previously deprecated or its functionality superseded by better alternatives:

- hprof
- java-rmi.cgi
- java-rmi.exe
- JavaDB
- jhat
- native2ascii

Two additional things that have been removed in Java 9 are:

- AppleScript engine. This engine was deemed as unusable and is dropped without replacement.
- Windows 32-bit client virtual machine. JDK 9 does support a 32-bit server JVM, but not a 32-bit client VM. This change was made to focus on the increased performance of 64-bit systems.

Updated garbage collection

Garbage collection has been one of Java's great claims to fame. In Java 9, the **Garbage-First (G1)** garbage collector is now the default garbage collector on both 32- and 64-bit servers. In Java 8, the default garbage collector was the parallel garbage collector. Oracle reports that there are three garbage collection combinations that will prohibit your application from starting in Java 9. Those combinations are:

- DefNew + CMS
- Incremental CMS
- ParNew + SerialOld

We will take an in-depth look at Java 9 garbage collection in [Chapter 7, Leveraging the New Default G1 Garbage Collector](#).

Deployment

There are three issues that you should be aware of, in the context of migrating to Java 9, when you are deploying your applications. These issues are JRE version selection, serialized applets, and the update to the JNLP.



JNLP is the acronym for **Java Network Launch Protocol** and is covered in a later section of this chapter.

JRE version selection

Prior to Java 9, developers could request a JRE version other than the version being launched when launching an application. This could be accomplished with a command-line option or with a proper JAR file manifest configuration. This feature has been removed in JDK 9 because of the way we typically deploy applications. Here are the three primary methods:

- Active installers
- **Java Web Start** using JNLP
- Native OS packaging systems

Serialized applets

Java 9 does not support the ability to deploy applets as serialized objects. In the past, applets were deployed as serialized objects to compensate for slow compression and JVM performance issues. With Java 9, compression techniques are advanced and the JVM has great performance.

If you attempt to deploy your applets as serialized objects, your object attributes and parameter tags will simply be ignored when your applet launches. Starting with Java 9, you can deploy your applets using standard deployment strategies.

JNLP update

The JNLP is used for launching applications on a desktop client using resources located on a web server. JNLP clients include Java Web Start and Java Plug-in software because they are able to launch applets that are remotely hosted. This protocol is instrumental in launching RIAs.

RIAs are **Rich Internet Applications** and when launched with JNLP have access to the various JNLP APIs that, with user permission, can access the user's desktop.

In Java 9, the JNLP specification has been updated. There are four specific updates as detailed in the next sections.

Nested resources

The ability to use component extensions with nest resources in Java or j2se elements was previously supported, but not documented in the specification. The specification has now been updated to reflect this support. The previous specification read:

No java elements can be specified as part of the resources.

The updated specification for Java 9 now reads:

A java element in a component extension will not govern what version of java is used, but may be used containing nested resource elements, and then those resources may be used only when using a Java version that matches the given version as specified in section 4.6

This specific change ensures that extension JLP files must have `java` or `j2se` resources and those resources will not dictate what JRE is used. Nested resources are permitted when using the specified version.

FX XML extension

When using the JNLP, you create a JNLP file. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="" href="">
  <information>
    <title>Sample</title>
    <vendor>The Sample Vendor</vendor>
    <icon href="sample-icon.jpg"/>
    <offline-allowed/>
  </information>
  <resources>
    <!-- Application Resources -->
    <j2se version="1.6+"
      href="http://java.sun.com/products/autodl/j2se"/>
    <jar href="Sample-Set.jar" main="true" />
  </resources>
  <application-desc
    name="Sample Application"
    main-class="com.vendor.SampleApplication"
    width="800"
    height="500">
    <argument>Arg1</argument>
    <argument>Arg2</argument>
    <argument>Arg3</argument>
  </application-desc>
  <update check="background"/>
</jnlp>
```

Two changes have been made to the `<application-desc>` element. First, the optional `type` attribute has been added so the type of application can be annotated. The default type is `Java`, so if your program is a Java app, you need not include the `type` attribute. Alternatively, you can specify `Java` as your type as follows:

```
<application-desc
  name="Another Sample Application"
  type="Java" main-class="com.vendor.SampleApplication2"
  width="800"
  height="500">
  <argument>Arg1</argument>
  <argument>Arg2</argument>
  <argument>Arg3</argument>
</application-desc>
```

We can indicate other application types to include `JavaFX` as shown here:

```
<application-desc
  name="A Great JavaFX Application"
  type="JavaFX" main-class="com.vendor.GreatJavaFXApplication"
  width="800"
```

```
height="500">
<argument>Arg1</argument>
<argument>Arg2</argument>
<argument>Arg3</argument>
</application-desc>
```

 *If you indicate an application type that is not supported by the JNLP client, your application launch will fail. For more information about JNLP, you can consult the official documentation: <http://docs.oracle.com/javase/7/docs/technotes/guides/javaws/developersguide/faq.html>.*

The second change to the `<application-desc>` element in Java 9 is the addition of the `param` sub-element. This allows us to provide the name of parameters along with their value using the `value` attribute. Here is an example of how an `<application-desc>` element of a JNLP file looks with the `param` sub-element and the `value` attribute included. This example shows three sets of parameters:

```
<application-desc
    name="My JRuby Application"
    type="JRuby"
    main-class="com.vendor.JRubyApplication"
    width="800"
    height="500">
    <argument>Arg1</argument>
    <argument>Arg2</argument>
    <argument>Arg3</argument>
    <param name="Parameter1" value="Value1"/>
    <param name="Parameter2" value="Value2"/>
    <param name="Parameter3" value="Value3"/>
</application-desc>
```

 *If the application type is Java, then any param sub-elements you use will be ignored.*

JNLP file syntax

JNLP file syntax is now in complete compliance with XML specifications. Prior to Java 9, you could use & to create complex comparisons. That is not supported with standard XML. You can still create complex comparisons in JNLP files. Now you will use & instead of &.

Numeric version comparison

The JNLP specification has been changed to reflect how numeric version elements were compared against non-numeric version elements. Previous to the change, version elements were compared lexicographically by ASCII value. With Java 9 and this JNLP specification change, elements are still compared lexicographically by ASCII value. The change is evident when the two strings have different lengths. In new comparisons, the shorter string will be padded with leading zeros to match the length of the longer string.



Lexicographical comparisons use a mathematical model that is based on alphabetical order.

Useful tools

The first thing you will need to do before migrating your applications to Java 9 is to download JDK 9. You can download the early access builds at this URL--<http://jdk.java.net/9/>. You will need to accept the license agreement and then select which build to download. As you can see in the following screenshot, there are several options based on your operating system:

Builds			
		JRE	JDK
Windows	32	exe (sha256) 83.59 MB	exe (sha256) 298.30 MB
	64	exe (sha256) 88.62 MB	exe (sha256) 309.18 MB
Mac OS	64	dmg (sha256) 72.28 MB	dmg (sha256) 319.94 MB
	Linux	tar.gz (sha256) 77.95 MB	tar.gz (sha256) 271.22 MB
Linux ARM	32	tar.gz (sha256) 78.96 MB	tar.gz (sha256) 279.76 MB
	64		tar.gz (sha256) 176.19 MB
Solaris SPARC	64	tar.gz (sha256) 52.26 MB	tar.gz (sha256) 206.72 MB
Solaris x86	64	tar.gz (sha256) 51.94 MB	tar.gz (sha256) 205.77 MB
		Server JRE	JDK
Alpine Linux	64	tar.gz (sha256) 200.00 MB	tar.gz (sha256) 200.00 MB

Now that you have JDK 9 installed on your development computer, let's look at a couple of tools that can help facilitate migrating your applications to Java 9.

Java environment - jEnv

If you develop on a computer with Linux or macOS, you might consider using `jEnv`, an open source Java environment management tool. This is a command-line tool, so do not expect a GUI. You can download the tool at this URL--<https://github.com/gcuisinier/jenv>.

Here is the installation command for Linux:

```
| $ git clone https://github.com/gcuisinier/jenv.git ~/.jenv
```

To download using macOS with Homebrew, use this command:

```
| $ brew install jenv
```

You can also install on Linux or macOS using `Bash` as follows:

```
| $ echo 'export PATH="$HOME/.jenv/bin:$PATH"' >> ~/.bash_profile
| $ echo 'eval "$(jenv init -)"' >> ~/.bash_profile
```

Alternatively, you can install on Linux or macOS using `Zsh` as follows:

```
| $ echo 'export PATH="$HOME/.jenv/bin:$PATH"' >> ~/.zshrc
| $ echo 'eval "$(jenv init -)"' >> ~/.zshrc
```

After you have `jEnv` installed, you will need to configure it on your system as shown here. You will need to modify the script to reflect your actual path:

```
| $ jenv add /Library/Java/JavaVirtualMachines/jdk17011.jdk/Contents/Home
```

You will want to repeat the `jenv add` command for each version of the JDK on your system. With each `jenv add` command, you will receive confirmation that the specific JDK version was added to `jEnv` as follows:

```
| $ jenv add /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
| oracle64-1.6.0.39 added
| $ jenv add /Library/Java/JavaVirtualMachines/jdk17011.jdk/Contents/Home
| oracle64-1.7.0.11 added
```

You can check to see what JDK versions you have added to your `jEnv` by using `$ jenv versions` at the Command Prompt. This will result in an output list.

Here are three additional `jenv` commands:

- `jenv global <version>`: this sets the global version
- `jenv local <version>`: this sets the local version
- `jenv shell <version>`: this sets the instance version for the shell

Maven

Maven is an open source tool that can be used for building and managing Java-based projects. It already supports Java 9 and is part of the *Apache Maven Project*. If you are not already using Maven and you do a lot of Java development you might be enticed by the following Maven objectives:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features

You can read more specifics about each of the Maven objectives at this site--<https://maven.apache.org/what-is-maven.html>. To download Maven, visit this site--<https://maven.apache.org/download.cgi>. Installation instructions for Windows, macOS, Linux, and Solaris are available here--<https://maven.apache.org/install.html>.

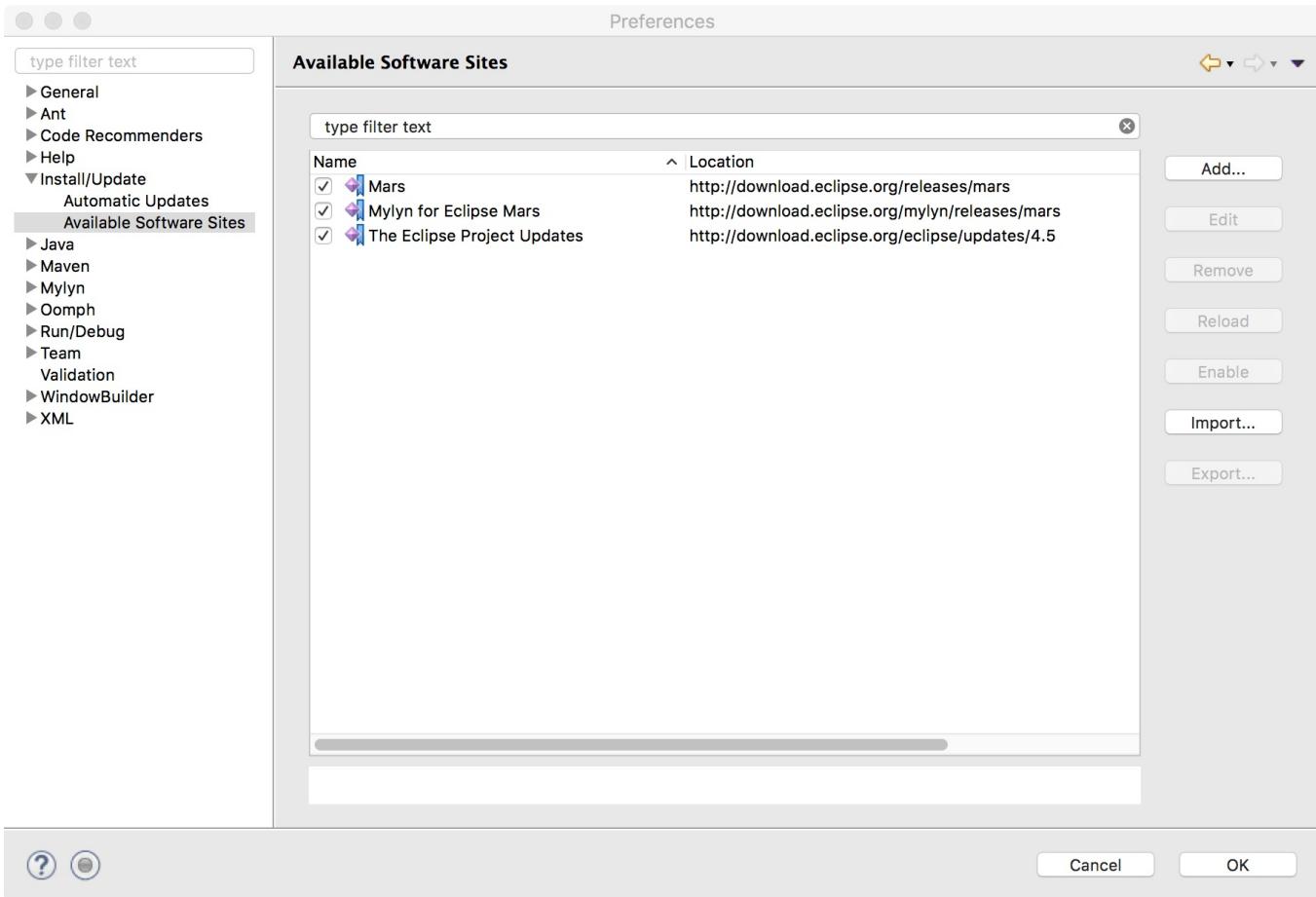
Maven can be integrated with Eclipse (M2Eclipse), JetBrains IntelliJ IDEA, and the Netbeans IDE. The M2Eclipse IDE, as an example, provides rich integration with Apache Maven and boasts the following features:

- You can launch Maven builds from within Eclipse
- Manage your dependencies for the Eclipse build path
- Easily resolve Maven dependencies (you can do this directly from Eclipse and not have to install a local Maven repository)
- Automatically download required dependencies (from remote Maven repositories)
- Use software wizards to create new Maven projects, create `pom.xml` files, and to enable Maven support for your plain Java projects
- Rapid dependency search of Maven remote repositories

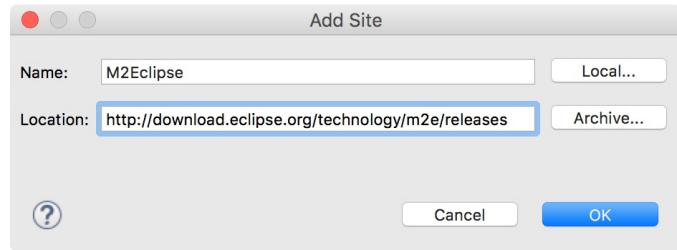
Obtaining the M2Eclipse IDE

To obtain the M2Eclipse IDE, you must first have Eclipse installed. Here are the steps:

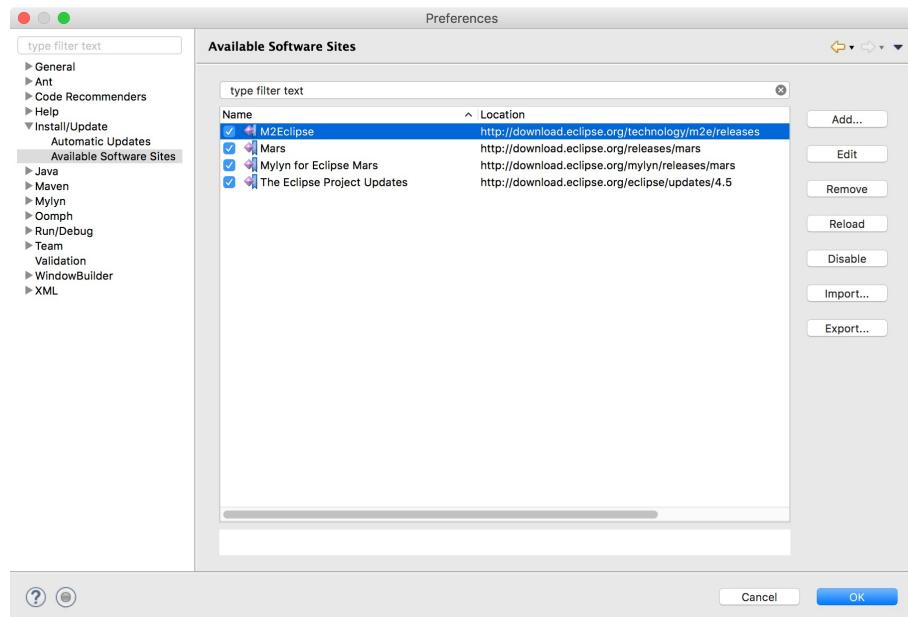
1. Start by opening your current Eclipse IDE. Next, select Preferences | Install/Update | Available Software Sites as shown in the following screenshot:



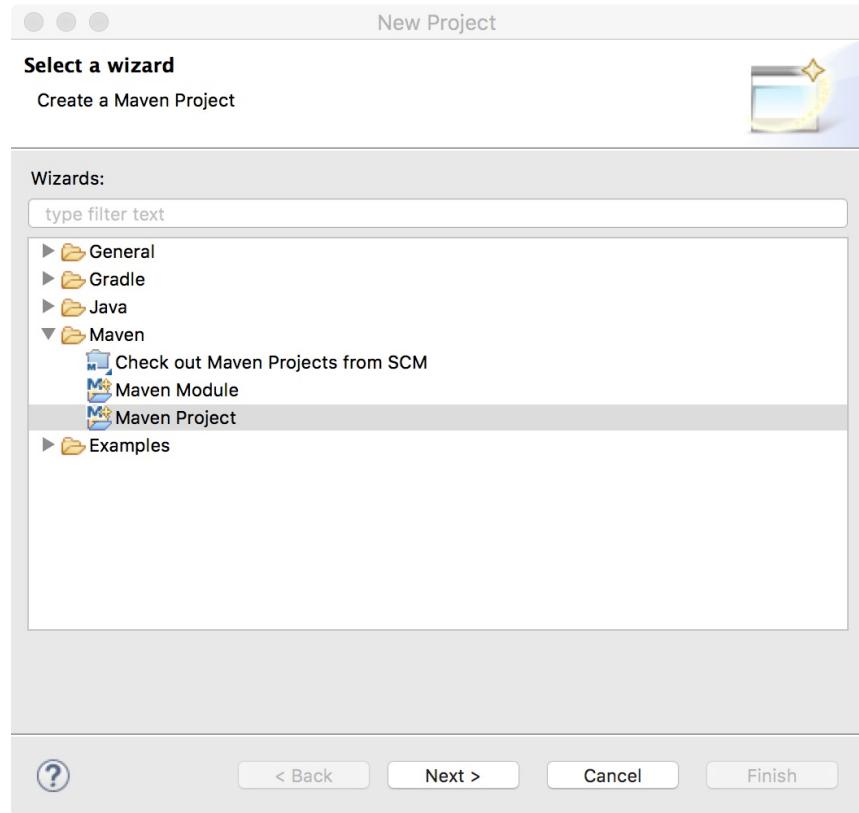
2. The next task is to add the M2Eclipse repository site to your list of Available Software Sites. To accomplish this, click the Add button and enter values in the Name and Location text input boxes. For Name, enter something to help you remember that M2Eclipse is available at this site. For Location, enter the URL--<http://download.eclipse.org/technology/m2e/releases>. Then, click the OK button:



3. You should now see the M2Eclipse site listed in your list of Available Software Sites as shown in the following screenshot. Your final step is to click the OK button:



4. Now, when you start a new project, you will see `Maven Project` as an option:



Maven is a proven tool for Java developers. You might consider obtaining additional information on Maven with one of the following resources:

- *Apache Maven Project*: <https://maven.apache.org/index.html>
- *Apache Maven Cookbook*: <https://www.packtpub.com/application-development/apache-maven-cookbook>
- *Apache Maven 3.0 Cookbook*: <https://www.packtpub.com/application-development/apache-maven-3-cookbook>
- *Getting Started with Apache Maven [Video]*: <https://www.packtpub.com/application-development/getting-started-apache-maven-video>

Summary

In this chapter, we explored potential issues involved in migrating our existing applications to the Java 9 platform. We looked at both manual and semi-automated migration processes. This chapter provided you with insights and processes to get your Java 8 code working with Java 9. Specifically, we conducted a quick review of Project Jigsaw, looked at how modules fit into the Java landscape, provided tips for migration planning, shared advice from Oracle regarding migration, and shared tools that you can use to help you as you get started with Java 9.

In the next chapter, we will take a close look at the Java shell and the JShell API. We demonstrate the JShell API and the JShell tool's ability to interactively evaluate declarations, statements, and expressions of the Java programming language. We will demonstrate features and use of this command-line tool.

Experimenting with the Java Shell

In the previous chapter, we explored how to migrate pre-Java 9 applications to the new Java platform. We examined several issues that might cause your current applications to have problems running on Java 9. We started with a review of Project Jigsaw and then looked at how modules fit into the new Java platform. We provided you with insights and processes to get your Java 8 code working with Java 9. Specifically, we provided tips for migration planning, shared advice from Oracle regarding migration, and shared tools that you can use to help you as you get started with Java 9.

In this chapter, we will take our first look at the new command line, **read-eval-print loop** (also referred to as **REPL**) tool in Java 9, the **Java Shell (JShell)**. We will start with introductory information regarding the tool, the read-eval-print loop concept, and move into the commands and command-line options for use with JShell. We will take a practitioner's approach to our review of the Java Shell and include examples you can try on your own.

The following topics are covered in this chapter:

- What is JShell?
- Getting started with JShell
- Practical uses of JShell
- Working with scripts

What is JShell?

JShell is a new tool introduced with Java 9. It is an interactive read-eval-print loop tool that is used to evaluate the following Java programming language components-- declarations, statements, and expressions. It has its own API so that it can be used by external applications.



Read-Eval-Print Loop is often referred to as REPL, taking the first letter from each word in the phrase. It is also known language shell or interactive top-level.

The introduction of JShell was a result of **Java Enhancement Program (JEP) 222**. Here are the stated goals of this JEP in regards to the Java Shell command-line tool:

- Facilitate rapid investigation
- Facilitate rapid coding
- Provide an edit history

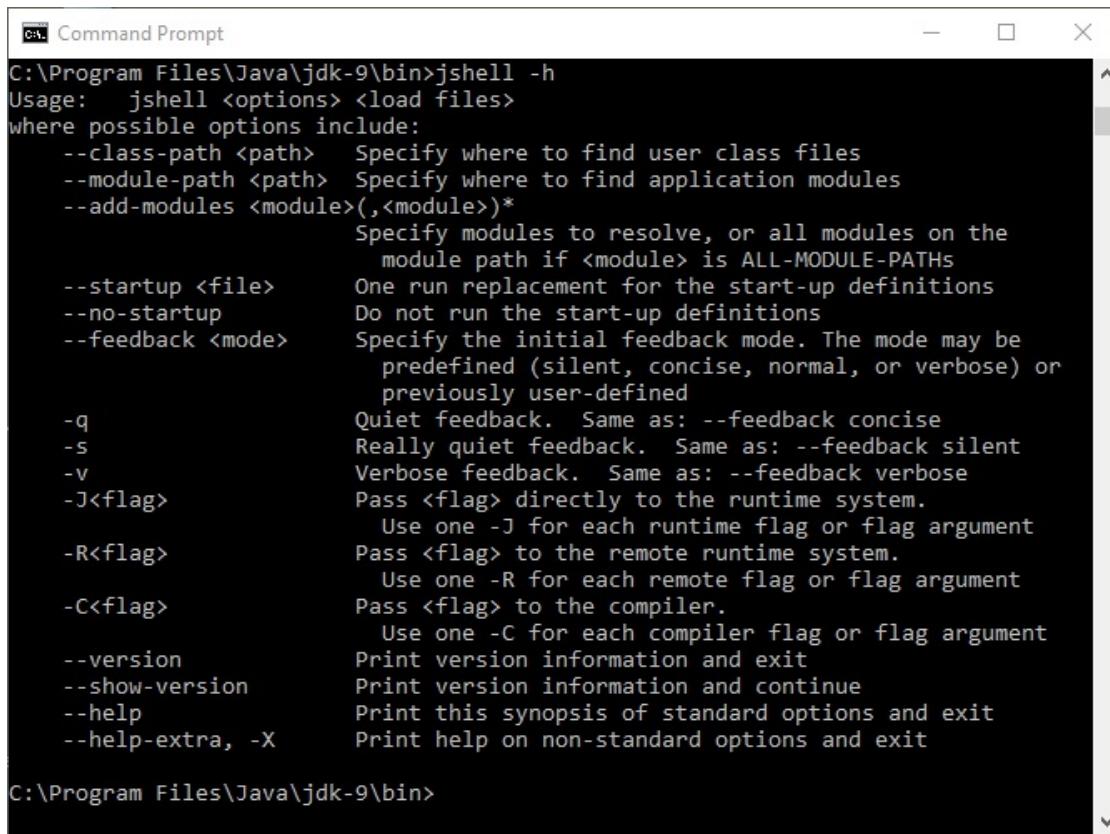
The rapid investigation and coding listed previously includes statements and expressions. Impressively, these statements and expressions do not need to be part of a method. Furthermore, variables and methods are not required to be part of a class, making this tool especially dynamic.

In addition, the following listed features were included to make JShell much easier to use and to make your time using JShell as time-efficient as possible:

- Tab-completion
- Auto-completion for end-of-statement semicolons
- Auto-completion for imports
- Auto-completion for definitions

Getting Started with JShell

JShell is a command-line tool that is located in the `/bin` folder. The syntax for this tool is `jshell <options> <load files>`. As you can see here, there are several options that can be used with this tool:

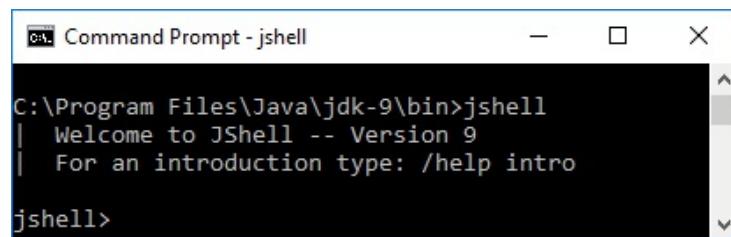


```
Command Prompt
C:\Program Files\Java\jdk-9\bin>jshell -h
Usage: jshell <options> <load files>
where possible options include:
--class-path <path> Specify where to find user class files
--module-path <path> Specify where to find application modules
--add-modules <module>(<module>)*
                                Specify modules to resolve, or all modules on the
                                module path if <module> is ALL-MODULE-PATHS
--startup <file> One run replacement for the start-up definitions
--no-startup Do not run the start-up definitions
--feedback <mode> Specify the initial feedback mode. The mode may be
                    predefined (silent, concise, normal, or verbose) or
                    previously user-defined
-q Quiet feedback. Same as: --feedback concise
-s Really quiet feedback. Same as: --feedback silent
-v Verbose feedback. Same as: --feedback verbose
-J<flag> Pass <flag> directly to the runtime system.
          Use one -J for each runtime flag or flag argument
-R<flag> Pass <flag> to the remote runtime system.
          Use one -R for each remote flag or flag argument
-C<flag> Pass <flag> to the compiler.
          Use one -C for each compiler flag or flag argument
--version Print version information and exit
--show-version Print version information and continue
--help Print this synopsis of standard options and exit
--help-extra, -X Print help on non-standard options and exit

C:\Program Files\Java\jdk-9\bin>
```

You have already seen the `-h` option, that we executed with `jshell -h`. This provided the listing of JShell options.

To log into your JShell, you simply use the `jshell` command. You will see that the prompt in the command window changes accordingly:



```
Command Prompt - jshell
C:\Program Files\Java\jdk-9\bin>jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro
jshell>
```

Exiting the shell is as easy as entering `/exit`. Once inside the JShell, you can enter any of the following commands:

Command	Functionality
<code>/drop</code>	<p>Use this command to delete a source entry that is referenced by <code>name</code> or <code>id</code>. Here is the syntax:</p> <pre> /drop <name or id></pre>
<code>/edit</code>	<p>With this command, you can edit a source entry using a <code>name</code> or <code>id</code> reference. Here is the syntax:</p> <pre> /edit <name or id></pre>
<code>/env</code>	<p>This powerful command allows you to view or change the evaluation context. Here is the syntax:</p> <pre> /env [-class-path <path>] [-module-path <path>] [-add-modules <modules>]</pre>
<code>/exit</code>	<p>This command is used to exit the JShell. The syntax is simply <code>/exit</code> without any options or parameters available.</p>
<code>/history</code>	<p>The history command provides a history of what you have typed. The syntax is simply <code>/history</code> without any options or parameters available.</p>
<code>/<i>id</i>></code>	<p>This command is used to rerun a previous snippet by referencing the <code>id</code>. Here is the syntax: <code>/<i>id</i>></code></p> <p> <i>You can also run a specific snippet by referencing the n^{th} previous snippet with <code>/-<n></code>.</i></p>

/imports	You can use this command to list the imported items. The syntax is /imports and it does not accept any options or parameters.
/list	This command will list the source you typed. Here is the syntax: /list [<name or id> -all -start]
/methods	This command lists all declared methods as well as their signatures. Here is the syntax: /methods [<name or id> -all -start]
/open	Using this command, you can open a file as source input. Here is the syntax: /open <file>
/reload	The reload command gives you the ability to reset and replay relevant history. Here is the syntax: /reload [-restore] [-quiet] [-class-path <path>] [-module-path <path>]
/reset	This command resets the JShell. Here is the syntax: /reset [-class-path <path>] [-module-path <path>] [-add-modules <modules>]
/save	This command saves the snippet source to a file specified by you. Here is the syntax: /save [-all -history -start] <file>
/set	This command is used to set the JShell configuration information. Here is the syntax: /set editor start feedback mode prompt

	<pre> truncation format</pre>
/types	<p>This command simply lists declared types. Here is the syntax:</p> <pre> /types [<name or id> -all -start]</pre>
/vars	<p>This command lists all declared variables as well as their values. Here is the syntax:</p> <pre> /vars [<name or id> -all -start]</pre>
!*	<p>This command will rerun the last snippet. The syntax is simply <code>!*</code></p>

*Several of the previously listed commands use the term **snippet**. In the context of Java 9 and JShell, a snippet is one of the following:*

- *ClassDeclaration*
- *Expression*
- *FieldDeclaration*
- *ImportDeclaration*
- *InterfaceDeclaration*
- *MethodDeclaration*



Entering the `/help` or `/?` command in the JShell provides a complete list of commands and syntax that can be used in the shell. That list is provided as follows:



```
jshell> /?
| Type a Java language expression, statement, or declaration.
| Or type one of the following commands:
/list [<name or id>|-all|-start]
    list the source you have typed
/edit <name or id>
    edit a source entry referenced by name or id
/drop <name or id>
    delete a source entry referenced by name or id
/save [-all|-history|-start] <file>
    Save snippet source to a file.
/open <file>
    open a file as source input
/vars [<name or id>|-all|-start]
    list the declared variables and their values
/methods [<name or id>|-all|-start]
    list the declared methods and their signatures
/types [<name or id>|-all|-start]
    list the declared types
/imports
    list the imported items
/exit
    exit jshell
/env [-class-path <path>] [-module-path <path>] [-add-modules <modules>] ...
    view or change the evaluation context
/reset [-class-path <path>] [-module-path <path>] [-add-modules <modules>]...
    reset jshell
/reload [-restore] [-quiet] [-class-path <path>] [-module-path <path>]...
    reset and replay relevant history -- current or previous (-restore)
/history
    history of what you have typed
/help [<command>|<subject>]
    get information about jshell
/set editor|start|feedback|mode|prompt|truncation|format ...
    set jshell configuration information
/? [<command>|<subject>]
    get information about jshell
/!
    re-run last snippet
/<id>
    re-run snippet by id
/-n>
    re-run n-th previous snippet

For more information type '/help' followed by the name of a
command or a subject.
For example '/help /list' or '/help intro'.

Subjects:

intro
    an introduction to the jshell tool
shortcuts
    a description of keystrokes for snippet and command completion,
    information access, and automatic code generation
context
    the evaluation context options for /env /reload and /reset

jshell>
```

The `/help` command can be especially helpful if you are still new to JShell. As you can see in the following screenshot, we can obtain an introduction to JShell by simply entering the `/help intro` command:

```
Command Prompt - jshell
jshell> /help intro
intro

The jshell tool allows you to execute Java code, getting immediate results.
You can enter a Java definition (variable, method, class, etc), like: int x = 8
or a Java expression, like: x + x
or a Java statement or import.
These little chunks of Java code are called 'snippets'.

There are also jshell commands that allow you to understand and
control what you are doing, like: /list

For a list of commands: /help

jshell>
```

If you find yourself using JShell often you might benefit from one or more of the following listed shortcuts. These can be listed at any time from within JShell by using the `/help shortcuts` command:

```
Command Prompt - jshell
jshell> /help shortcuts
shortcuts

Supported shortcuts include:

<tab>
    After entering the first few letters of a Java identifier,
    a jshell command, or, in some cases, a jshell command argument,
    press the <tab> key to complete the input.
    If there is more than one completion, then possible completions will be shown.
    Will show documentation if available and appropriate.

Shift-<tab> v
    After a complete expression, hold down <shift> while pressing <tab>,
    then release and press "v", the expression will be converted to
    a variable declaration whose type is based on the type of the expression.

Shift-<tab> i
    After an unresolvable identifier, hold down <shift> while pressing <tab>,
    then release and press "i", and jshell will propose possible imports
    which will resolve the identifier based on the content of the specified classpath.

jshell>
```

Additional help can be obtained from within the JShell by using the `/help` command followed by the command you want additional help on. For example, entering `/help reload` provides detailed information regarding the `/reload` command. That information is provided as follows:

```
Command Prompt - jshell
jshell> /help reload
/reload
Reset the jshell tool code and execution state then replay each valid snippet
and any /drop commands in the order they were entered.

/reload
    Reset and replay the valid history since jshell was entered, or
    a /reset, or /reload command was executed -- whichever is most
    recent.

/reload -restore
    Reset and replay the valid history between the previous and most
    recent time that jshell was entered, or a /reset, or /reload
    command was executed. This can thus be used to restore a previous
    jshell tool session.

/reload [-restore] -quiet
    With the '-quiet' argument the replay is not shown. Errors will display.

Each of the above accepts context options, see:
/help context

For example:
/reload -add-modules com.greetings -restore

jshell>
```

Practical uses of JShell

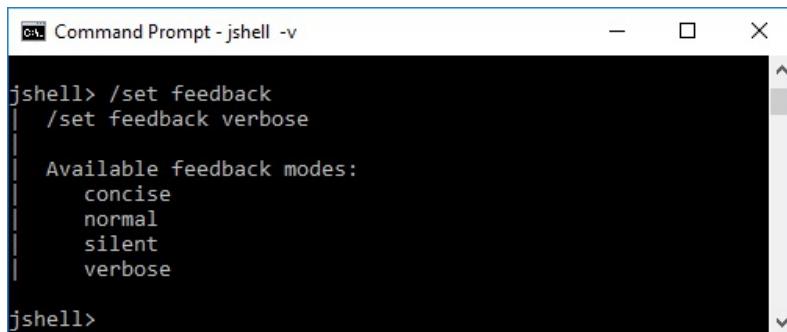
Whether you are a new or seasoned developer or just new to Java, you are bound to find the JShell very useful. In this section, we will look at some practical uses of JShell. Specifically, we will cover:

- Feedback modes
- Listing your assets
- Editing in the JShell

Feedback modes

Command-line tools usually provide relatively sparse feedback in an effort to not overcrowd the screen or otherwise become a nuisance to developers. JShell has several feedback modes in addition to giving developers the ability to create their own custom modes.

As you can see from the following screenshot, there are four feedback modes-- `concise`, `normal`, `silent`, and `verbose`. Here, we entered the `/set feedback` command without any parameters to list the feedback modes as well as to identify what the current feedback mode is. The first line of output displays the command-line command and argument set that would be used to set the mode to the currently set mode. So, in the following screenshot, the current feedback mode is set to `verbose` and the other three modes are listed:



```
c:\ Command Prompt - jshell -v
jshell> /set feedback
      /set feedback verbose

      Available feedback modes:
          concise
          normal
          silent
          verbose

jshell>
```

We can dictate which mode we want to enter when we first enter JShell by including an option when we launch JShell. Here are the command-line options:

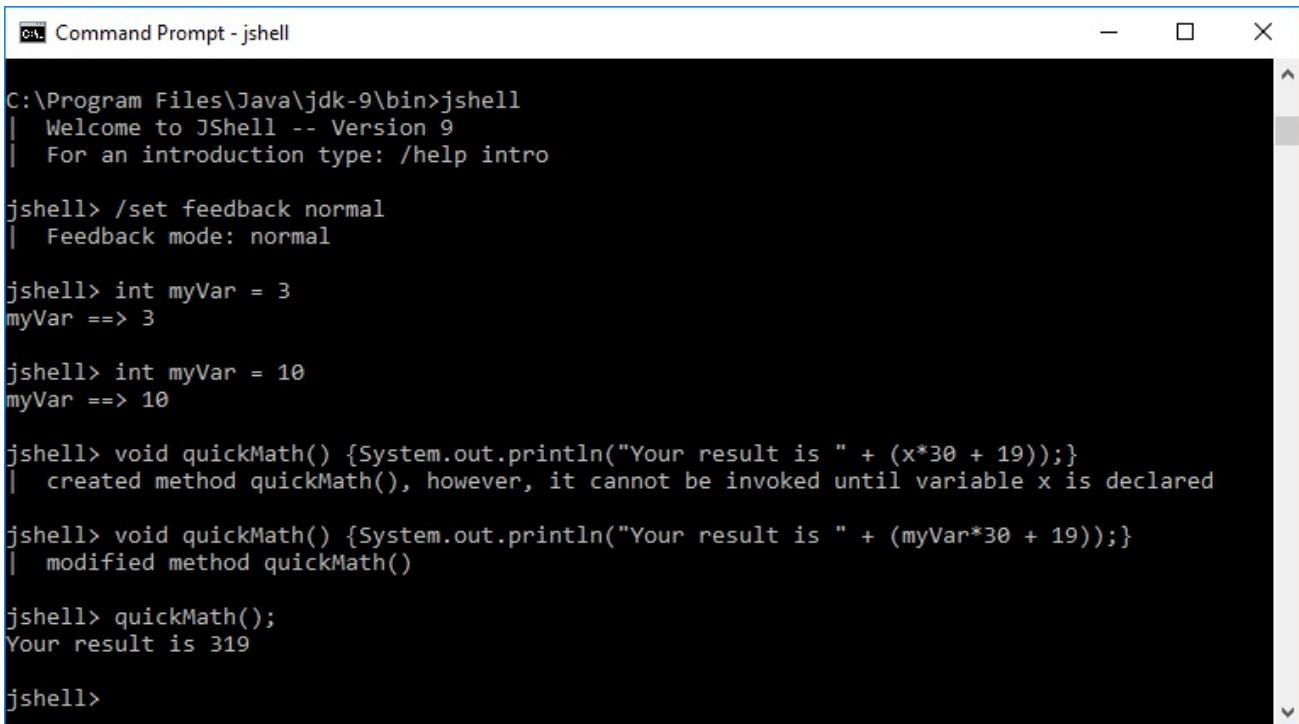
Command-line command and option	Feedback mode
<code>jshell -q</code>	<code>concise</code>
<code>jshell -n</code>	<code>normal</code>
<code>jshell -s</code>	<code>silent</code>
<code>jshell -v</code>	<code>verbose</code>

You will notice that we use `-q` for `concise` mode instead of `-c`. The `-c<flag>` syntax and is used to pass `<flag>` to the compiler.

The best way to review the differences between the feedback modes is to use examples. Starting with the `normal` mode, we will execute command-line commands to accomplish the following ordered feedback demonstration:

1. Create a variable.
2. Update the variable's value.
3. Create a method.
4. Update the method.
5. Run the method.

To start our first test, we will execute the `/set feedback normal` command at the `jshell>` prompt, which sets the JShell feedback mode to `normal`. After entering the `normal` feedback mode, we will enter the necessary commands to run our demonstration:



```
C:\Program Files\Java\jdk-9\bin>jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro

jshell> /set feedback normal
| Feedback mode: normal

jshell> int myVar = 3
myVar ==> 3

jshell> int myVar = 10
myVar ==> 10

jshell> void quickMath() {System.out.println("Your result is " + (x*30 + 19));}
| created method quickMath(), however, it cannot be invoked until variable x is declared

jshell> void quickMath() {System.out.println("Your result is " + (myVar*30 + 19));}
| modified method quickMath()

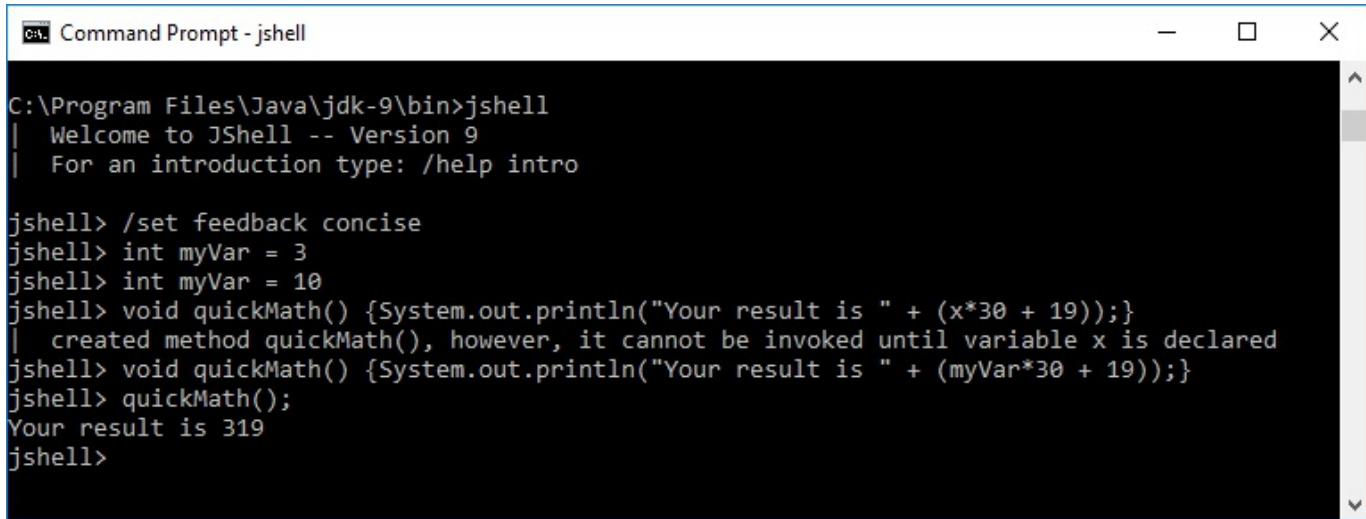
jshell> quickMath();
Your result is 319

jshell>
```

After entering `normal` feedback mode, we entered `int myVar = 3` to and received `myVar ==> 3` as feedback. In our next command, we changed the value of the same variable and received the same output with the new value. Our next statement, `void quickMath() {System.out.println("Your result is " + (x*30 + 19));}`, used a variable that was not declared and you see the resulting two-part feedback--one part indicating that the method was created and the other to inform that the method cannot be invoked until the undeclared variable is declared. Next, we changed our method to include the `myVar` variable and the feedback reported that the method was modified. Our last step was to run the method using `quickMath();` and the results are as we

expected.

Let's try this same feedback demonstration in `concise` mode:

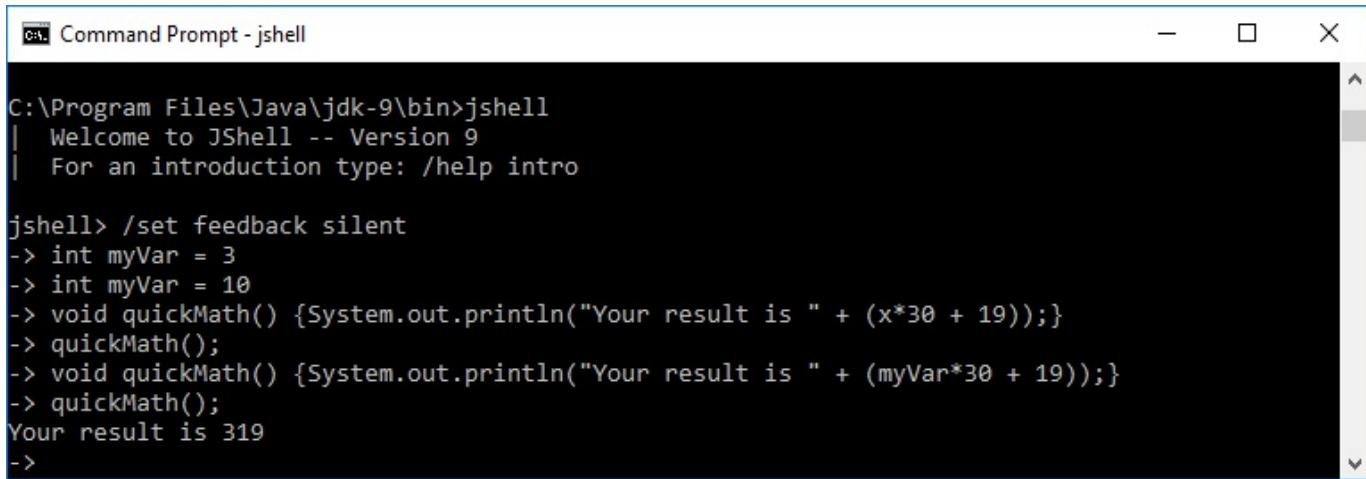


```
C:\Program Files\Java\jdk-9\bin>jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro

jshell> /set feedback concise
jshell> int myVar = 3
jshell> int myVar = 10
jshell> void quickMath() {System.out.println("Your result is " + (x*30 + 19));}
| created method quickMath(), however, it cannot be invoked until variable x is declared
jshell> void quickMath() {System.out.println("Your result is " + (myVar*30 + 19));}
jshell> quickMath();
Your result is 319
jshell>
```

As you can see from the preceding screenshot, the `concise` feedback mode provides us with much less feedback. We created and modified the variables and received no feedback. When we created the method with an undeclared variable, we received the same feedback that we did in `normal` mode. We updated the method without confirmation or other feedback.

Our next use of the feedback demonstration will be in `silent` mode:



```
C:\Program Files\Java\jdk-9\bin>jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro

jshell> /set feedback silent
-> int myVar = 3
-> int myVar = 10
-> void quickMath() {System.out.println("Your result is " + (x*30 + 19));}
-> quickMath();
-> void quickMath() {System.out.println("Your result is " + (myVar*30 + 19));}
-> quickMath();
Your result is 319
->
```

When we entered `silent` feedback mode, as you can see in the preceding screenshot, the JShell prompt changed from `jshell>` to `->`. There was no feedback provided when we created the `myVar` variable, modified the `myVar` variable, or created the `quickMath()` method. We intentionally created the `quickMath()` method to use an undeclared

variable. Because we were in `silent` feedback mode, we were not informed that the method had an undeclared variable. Based on this lack of feedback, we ran the method and were not provided any output or feedback. Next, we updated the method to include the `myVar` declared variable and then ran the method.



The `silent` feedback mode might seem pointless as no feedback is provided, but there is a great utility with this mode. Using the `silent` mode might be appropriate for pipelining or simply when you want to minimize the amount of terminal output. You can include specific, conditional, outputs with implicit `System.out.println` commands, as an example.

Our last use of the feedback demonstration is in `verbose` feedback mode. This feedback mode, as you would assume from its name, provides the most amount of feedback. Here are our test results:

```
Command Prompt - jshell
C:\Program Files\Java\jdk-9\bin>jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro

jshell> /set feedback verbose
| Feedback mode: verbose

jshell> int myVar = 3
myVar ==> 3
| created variable myVar : int

jshell> int myVar = 10
myVar ==> 10
| modified variable myVar : int
|     update overwrote variable myVar : int

jshell> void quickMath() {System.out.println("Your result is " + (x*30 + 19));}
| created method quickMath(), however, it cannot be invoked until variable x is declared

jshell> void quickMath() {System.out.println("Your result is " + (myVar*30 + 19));}
| modified method quickMath()
|     update overwrote method quickMath()

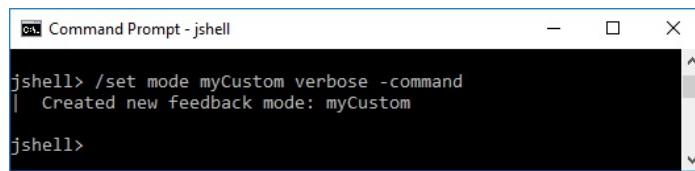
jshell> quickMath();
Your result is 319

jshell>
```

In our feedback demonstration, using `verbose` feedback mode, we receive a bit more feedback as well as a nicer format for the feedback.

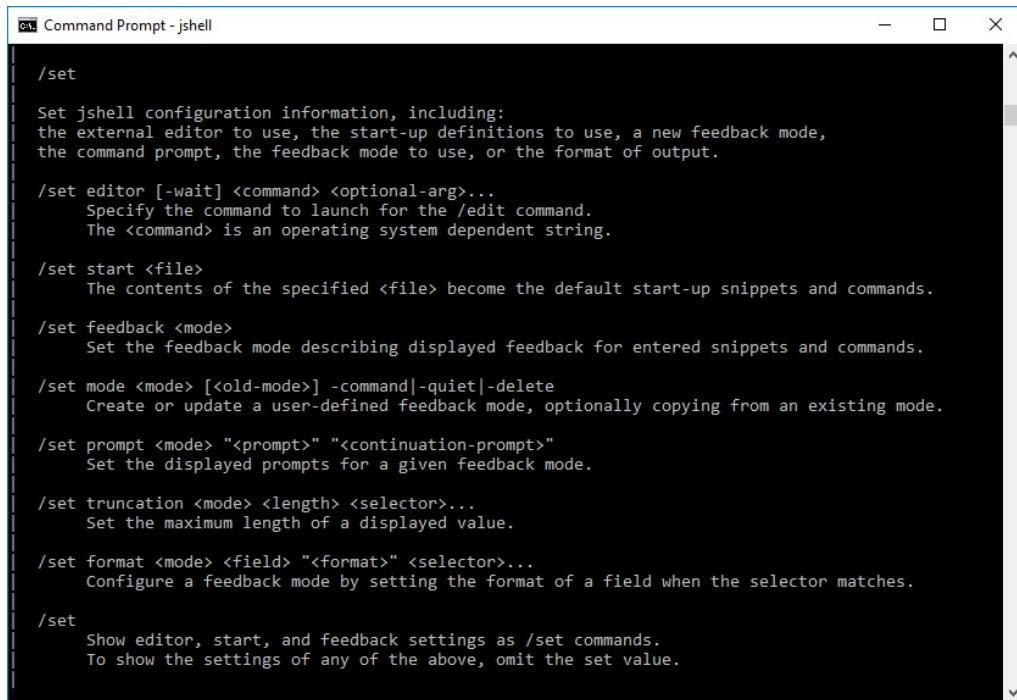
Creating a custom feedback mode

While the internal feedback modes (`normal`, `concise`, `silent`, and `verbose`) cannot be modified, you can create your own custom feedback mode. The first step in this process is to copy an existing mode. The following example demonstrates how to copy the `verbose` mode to a `myCustom` mode with the `/set mode myCustom verbose -command` command string:



```
jshell> /set mode myCustom verbose -command
| Created new feedback mode: myCustom
jshell>
```

We used the `-command` option to ensure we would receive the command feedback. You can make various changes to your feedback mode using the `/set` command along with one of the options listed in the following screenshot:



```
/set

Set jshell configuration information, including:
the external editor to use, the start-up definitions to use, a new feedback mode,
the command prompt, the feedback mode to use, or the format of output.

/set editor [-wait] <command> <optional-arg>...
    Specify the command to launch for the /edit command.
    The <command> is an operating system dependent string.

/set start <file>
    The contents of the specified <file> become the default start-up snippets and commands.

/set feedback <mode>
    Set the feedback mode describing displayed feedback for entered snippets and commands.

/set mode <mode> [<old-mode>] -command|-quiet|-delete
    Create or update a user-defined feedback mode, optionally copying from an existing mode.

/set prompt <mode> "<prompt>" "<continuation-prompt>"
    Set the displayed prompts for a given feedback mode.

/set truncation <mode> <length> <selector>...
    Set the maximum length of a displayed value.

/set format <mode> <field> "<format>" <selector>...
    Configure a feedback mode by setting the format of a field when the selector matches.

/set
    Show editor, start, and feedback settings as /set commands.
    To show the settings of any of the above, omit the set value.
```

As an example, let's walk through the truncation setting that mandates how many characters are displayed on each output line. Using the `/set truncation` command, as illustrated in the following screenshot, shows the current truncation settings:

```
Command Prompt - jshell

jshell> /set truncation
| /set truncation myCustom 80
| /set truncation myCustom 1000 expression,varvalue
| /set truncation normal 80
| /set truncation normal 1000 expression,varvalue
| /set truncation silent 80
| /set truncation silent 1000 expression,varvalue
| /set truncation concise 80
| /set truncation concise 1000 expression,varvalue
| /set truncation verbose 80
| /set truncation verbose 1000 expression,varvalue

jshell>
```

As you can see, our `myCustom` feedback mode has a truncation of `80`. We will change that to `60` with the `/set truncation myCustom 60` command and then use the `/set truncation` command for verification:

```
Command Prompt - jshell

jshell> /set truncation myCustom 60

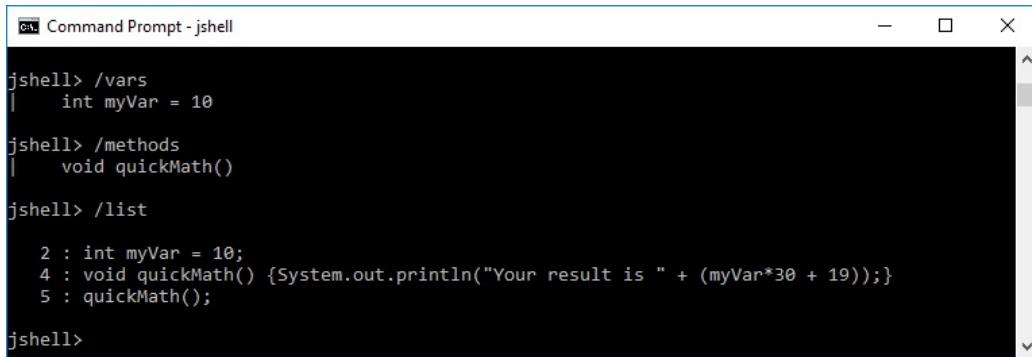
jshell> /set truncation
| /set truncation myCustom 60
| /set truncation normal 80
| /set truncation normal 1000 expression,varvalue
| /set truncation silent 80
| /set truncation silent 1000 expression,varvalue
| /set truncation concise 80
| /set truncation concise 1000 expression,varvalue
| /set truncation verbose 80
| /set truncation verbose 1000 expression,varvalue

jshell>
```

As you can see in the previous screenshot, the truncation for our `myCustom` feedback mode was successfully changed from the `80` inherited from the `verbose` mode to `60`, based on our use of the `/set truncation myCustom 60` JShell command.

Listing your assets

There are a few JShell commands that are convenient for listing assets that you have created. Using the feedback demonstration from the previous section, we executed the `/vars`, `/methods`, and `/list` commands to provide a list of variables, methods, and all sources respectively:



```
jshell> /vars
|   int myVar = 10

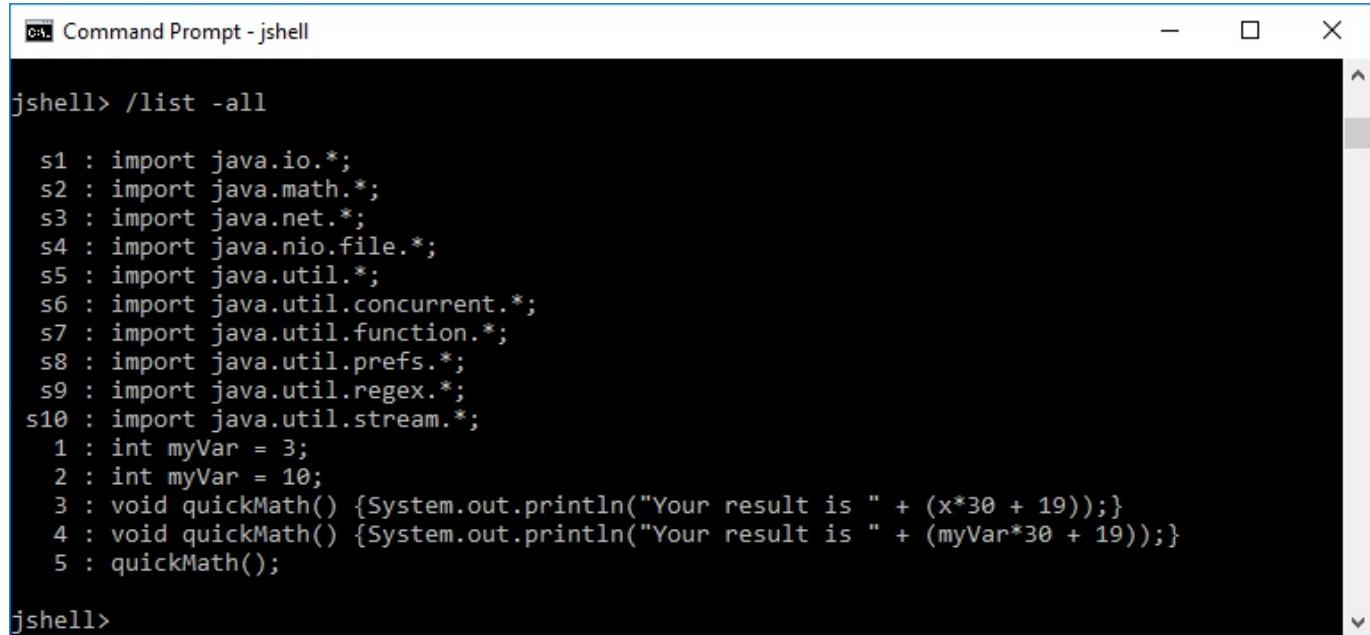
jshell> /methods
|   void quickMath()

jshell> /list

2 : int myVar = 10;
4 : void quickMath() {System.out.println("Your result is " + (myVar*30 + 19));}
5 : quickMath();

jshell>
```

We can also use the `/list -all` command and option combination to see what packages the JShell imported. As you can see in the following screenshot, JShell imported several packages that make our work within the shell more convenient, saving us time from having to import these standard packages in our methods:



```
jshell> /list -all

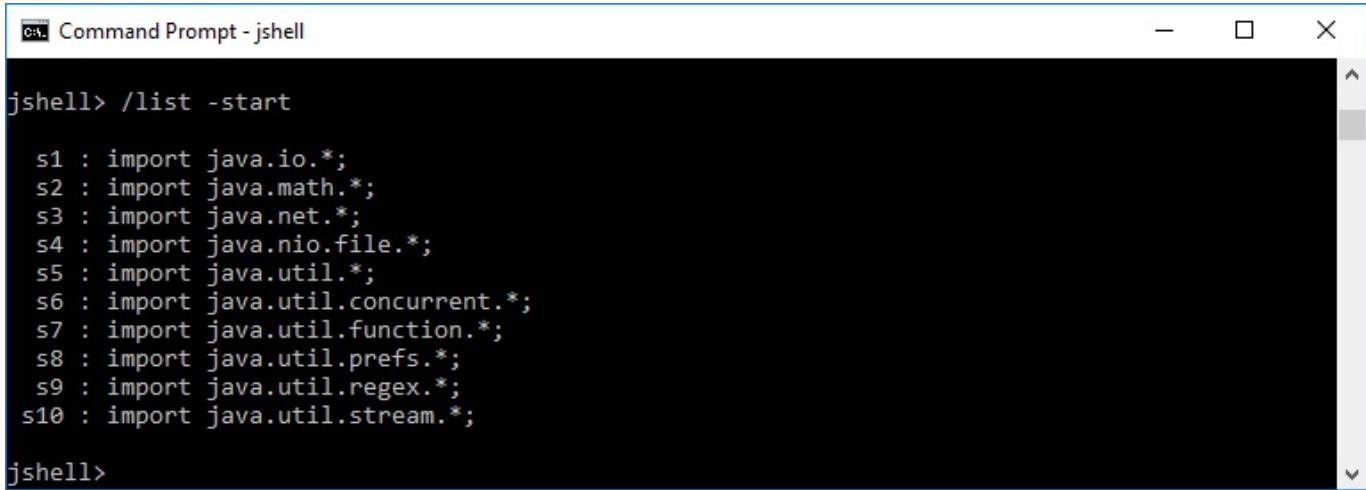
s1 : import java.io.*;
s2 : import java.math.*;
s3 : import java.net.*;
s4 : import java.nio.file.*;
s5 : import java.util.*;
s6 : import java.util.concurrent.*;
s7 : import java.util.function.*;
s8 : import java.util.prefs.*;
s9 : import java.util.regex.*;
s10 : import java.util.stream.*;

1 : int myVar = 3;
2 : int myVar = 10;
3 : void quickMath() {System.out.println("Your result is " + (x*30 + 19));}
4 : void quickMath() {System.out.println("Your result is " + (myVar*30 + 19));}
5 : quickMath();

jshell>
```

If you just want to list the startup imports, you can use the `/list -start` command and option combination. As you can see in the following screenshot, each start up import

has an "s" prefix and is numerically ordered:



```
jshell> /list -start

s1 : import java.io.*;
s2 : import java.math.*;
s3 : import java.net.*;
s4 : import java.nio.file.*;
s5 : import java.util.*;
s6 : import java.util.concurrent.*;
s7 : import java.util.function.*;
s8 : import java.util.prefs.*;
s9 : import java.util.regex.*;
s10 : import java.util.stream.*;

jshell>
```

Editing in the JShell

JShell is not a full-featured text editor, but there are several things you can do within the shell. This section provides you with the editing techniques grouped into modifying text, basic navigation, historical navigation, and advanced editing commands.

Modifying text

The default text edit/entry mode is that the text you type will appear at the current cursor position. You have several options available to you when you want to delete text. Here is a complete list:

Delete action	PC keyboard combination	Mac keyboard combination
Delete the character at the current cursor location	<i>Delete</i>	<i>Delete</i>
Delete the character to the left of the cursor	<i>Backspace</i>	<i>Backspace</i>
Delete the text from the cursor location to the end of the line	<i>Ctrl + K</i>	<i>Cmd + K</i>
Delete the text from the cursor location to the end of the current word	<i>Alt + D</i>	<i>Alt/Opt + D</i>
Delete from the cursor location to the previous white space	<i>Ctrl + W</i>	<i>Cmd + W</i>
Paste the most recently deleted text at the cursor location	<i>Ctrl + Y</i>	<i>Cmd + Y</i>
When the <i>Ctrl + Y</i> (or <i>Cmd + Y</i> on Mac) is used, you will be able to use the <i>Alt + Y</i> keyboard combination to cycle through previously deleted text	<i>Alt + Y</i>	<i>Alt/Opt + Y</i>

Basic navigation

While navigational control inside the JShell is similar to most command-line editors, it is helpful to have a list of basic navigational controls:

Key/Key combination	Navigation action
<i>Left arrow</i>	Move backward one character
<i>Right arrow</i>	Move forward one character
<i>Up arrow</i>	Move up one line through history
<i>Down arrow</i>	Move down one line forward through history
<i>Return</i>	Enter (submit) the current line
<i>Ctrl + A</i> (<i>cmd - A</i> on Mac)	Jump to the beginning of the current line
<i>Ctrl + E</i> (<i>cmd - E</i> on Mac)	Jump to the end of the current line
<i>Alt + B</i>	Jump back one word
<i>Alt + F</i>	Jump forward one word

Historical navigation

JShell remembers the snippets and commands that you enter. It maintains this history so that you can reuse snippets and commands you already entered. To cycle through snippets and commands, you can hold down the *Ctrl* key (*cmd* on Mac) and then use the up and down arrow keys until you see the snippet or command you want.

Advanced editing commands

There are several more editing options to include search functionality, macros creation and use, and more. JShell's editor is based on JLine2, a Java library for parsing console input and editing. You can learn more about JLine2 at this URL: <https://github.com/jline/jline2/wiki/JLine-2.x-Wiki>.

Working with scripts

Up to this point, you have entered data directly into JShell from the keyboard. You also have the ability to work with JShell scripts which are a sequence of JShell commands and snippets. The format is the same as other scripting formats with one command per line.

In this section, we will look at start up scripts, examine how to load scripts, how to save scripts, and then end with a look at advanced scripting with JShell.

Start up scripts

Each time the JShell is launched, the start up scripts are loaded. This also occurs each time the `/reset`, `/reload`, and `/env` commands are used.

By default, the DEFAULT start up script is used by JShell. If you want to use a different start up script, you merely need to use the `/set start <script>` command. Here is an example--`/set start MyStartupScript.jsh`. Alternatively, you can use the `jshell --start MyStartupScript.jsh` command at the Command Prompt to launch JShell and load the `MyStartupScript.jsh` JShell start up script.

When you use the `/set start <script>` command with the `-retain` option, you are telling JShell to use the new start up script the next time you launch JShell.

Loading scripts

Loading scripts in the JShell can be accomplished with one of the following methods:

- You can use the `/open` command along with the name of the script as a parameter. For example, if our script name is `MyScript`, we would use `/open MyScript`.
- A second option for loading scripts is to use the `jshell MyScript.jsh` at the Command Prompt. This will launch JShell and load the `MyScript.jsh` JShell script.

Saving scripts

In addition to creating JShell scripts in external editors, we can create them within the JShell environment as well. When taking this approach, you will need to use the `/save` command to save your scripts. As you can see in the following screenshot, the `/save` command requires, at a minimum, a file name argument:

```
c:\ Command Prompt - jshell
jshell> /help save
/save
    Save the specified snippets and/or commands to the specified file.

/save <file>
    Save the source of current active snippets to the file.

/save -all <file>
    Save the source of all snippets to the file.
    Includes source including overwritten, failed, and start-up code.

/save -history <file>
    Save the sequential history of all commands and snippets entered since jshell was launched.

/save -start <file>
    Save the current start-up definitions to the file.

jshell>
```

There are three options available to you with the `/save` command:

- The `-all` option can be used to save the source of all snippets to the specified file.
- The `-history` option saves a sequential history of all commands and snippets you entered since JShell was launched. JShell's ability to perform this operation informs you that it maintains a history of everything you enter.
- The `-start` option saves the current start up definitions to the specified file.

Advanced scripting with JShell

What are the limits of JShell? There is so much you can do with this tool, and you are virtually only limited by your imagination and programming abilities.

Let's look at an advanced code base that can be used to compile and run Java programs from a JShell script:

```
import java.util.concurrent.*
import java.util.concurrent.*
import java.util.stream.*
import java.util.*

void print2Console(String thetext)
{
    System.out.println(thetext);
    System.out.println("");
}

void runSomeProcess(String... args) throws Exception
{
    String theProcess =
        Arrays.asList(args).stream().collect(
            Collectors.joining(" "));
    print2Console("You asked me to run: '"+theProcess+"'");
    print2Console("");
    ProcessBuilder compileBuilder = new
        ProcessBuilder(args).inheritIO();
    Process compileProc = compileBuilder.start();
    CompletableFuture<Process> compileTask =
        compileProc.onExit();
    compileTask.get();
}

print2Console("JShell session launched.")
print2Console("Preparing to compile Sample.java. . . ")

// run the Java Compiler to complete Sample.java
runSomeProcess("javac", "Sample.java")
print2Console("Compilation complete.")
print2Console("Preparing to run Sample.class...")

// run the Sample.class file
runSomeProcess("java", "Sample")
print2Console("Run Cycle compete.")

// exit JShell
print2Console("JShell Termination in progress...")
print2Console("Session ended.")

/exit
```

As you can see with this script, we created a `runSomeProcess()` method and can use it to explicitly compile and run external Java files.

Summary

In this chapter, we examined JShell, Java 9's new read-eval-print loop command-line tool. We started with introductory information regarding the tool and looked closely at the read-eval-print loop concept. We spent considerable time reviewing JShell commands and command-line options. Our coverage included practical guides to feedback modes, asset listing, and editing in the shell. We also gained experience working with scripts.

In the next chapter, we will look at Java 9's new default garbage collector. Specifically, we will look at the default garbage collection, deprecated garbage collection combinations, and examine garbage collection logging.

Leveraging the New Default G1 Garbage Collector

In the previous chapter, we examined **Java Shell (JShell)**, Java 9's new **read-eval-print loop (REPL)** command-line tool. We started with introductory information regarding the tool and looked closely at the read-eval-print loop concept. We spent considerable time reviewing JShell commands and command-line options. Our coverage included practical guides to feedback modes, asset listing, and editing in the shell. We also gained experience working with scripts.

In this chapter, we will take an in-depth look at **garbage collection** and how it is handled in Java 9. We will start with an overview of garbage collection, and then look at specifics in the pre-Java 9 realm. Armed with that foundational information, we will look at specific garbage collection changes in the Java 9 platform. Lastly, we will look at some garbage collection issues that persist, even after Java 9.

The following topics are covered in this chapter:

- Overview of garbage collection
- The pre-Java 9 garbage collection schema
- Collecting garbage with the new Java platform
- Persistent issues

Overview of garbage collection

Garbage collection is the mechanism used in Java to deallocate unused memory. Essentially, when an object is created, memory space is allocated and dedicated to that object until it no longer has any references pointing to it. At that time, the system deallocates the memory. Java performs this garbage collection automatically for us, which can lead to a lack of attention to memory usage and poor programming practices in the area of memory management and system performance.

Java's garbage collection is considered an automatic memory management schema because programmers do not have to designate objects as ready to be deallocated. The garbage collection runs on a low-priority thread and, as you will read later in this chapter, has variable execution cycles.

In our overview of garbage collection, we will look at the following concepts:

- Object life cycle
- Garbage collection algorithms
- Garbage collection options
- Java methods relevant to garbage collection

We will look at each of these concepts in the sections that follow.

Object life cycle

In order to fully understand Java's garbage collection, we need to look at the entire life cycle of an object. Because the core of garbage collection is automatic in Java, it is not uncommon to see the terms *garbage collection* and *memory management* as assumed components of the object life cycle.

We will start our review of the object life cycle with object creation.

Object creation

Objects are declared and created. When we write an object declaration, or declare an object, we are declaring a name or identifier so that we can refer to an object. For example, the following line of code declares `myObjectName` as the name of an object of type `CapuchinMonkey`. At this point, no object was created and no memory allocated for it:

```
| CapuchinMonkey myObjectName;
```

We use the `new` keyword to create an object. The following example illustrates how to invoke the `new` operation to create an object. This operation results in:

```
| myObjectName = new CapuchinMonkey();
```

Of course, we can combine the declaration and creation statements together by using `CapuchinMonkey myObjectName = new CapuchinMonkey();` instead of `CapuchinMonkey myObjectName;` and `myObjectName = new CapuchinMonkey();`. They were separated in the preceding example for illustrative purposes.

When an object is created, a specific amount of memory is allocated for storing that object. The amount of memory allocated can differ based on architecture and JVM.

Next look at the mid-life of an object.

Object mid-life

Objects are created and Java allocates system memory for storing that object. If the object is not used, the memory allocated to it is considered wasted. This is something we want to avoid. Even with small applications, this type of wasted memory can lead to poor performance and even out-of-memory issues.

Our goal is to deallocate or release the memory, any previously allocated memory that we no longer need. Fortunately, with Java, there is a mechanism for handling this issue. It is called garbage collection.

When an object, such as our `myObjectName` example, no longer has any references pointing to it, the system will reallocate the associated memory.

Object destruction

The idea of Java having a garbage collector running in the dark shadows of your code (usually a low-priority thread) and deallocated memory currently allocated to unreferenced objects, is appealing. So, how does this work? The garbage collection system monitors objects and, as feasible, counts the number of references to each object.

When there are no references to an object, there is no way to get to it with the currently running code, so it makes perfect sense to deallocate the associated memory.



*The term **memory leak** refers to small memory chunks to be lost or improperly deallocated. These leaks are avoidable with Java's garbage collection.*

Garbage collection algorithms

There are several garbage collection algorithms, or types, for use by the Java virtual machine. In this section, we will cover the following garbage collection algorithms:

- Mark and sweep
- CMS garbage collection
- Serial garbage collection
- Parallel garbage collection
- G1 garbage collection

Mark and sweep

Java's initial garbage collection algorithm, *mark and sweep*, used a simple two-step process:

1. Java first step, mark, is to step through all objects that have accessible references, marking those objects as alive.
2. The second step, sweep, involves scanning the sea for any object that is not marked.

As you can readily determine, the mark and sweep algorithm seems effective, but probably not very efficient due to the two-step nature of this approach. This eventually lead to a Java garbage collection system with vastly improved efficiencies.

Concurrent mark sweep (CMS) garbage collection

The **concurrent mark sweep (CMS)** algorithm for garbage collection scans heap memory using multiple threads. Similar to the mark and sweep method, it marks objects for removal and then makes a sweep to actually remove those objects. This method of garbage collection is essentially an upgraded mark and sweep method. It was modified to take advantage of faster systems and had performance enhancements.

To manually invoke the concurrent mark sweep garbage collection algorithm for your application, use the following command-line option:

```
| -XX:+UseConcMarkSweepGC
```

If you want to use the concurrent mark sweep garbage collection algorithm and dictate the number of threads to use, you can use the following command-line option. In the following example, we are telling the Java platform to use the concurrent mark sweep garbage collection algorithm with eight threads:

```
| -XX:ParallelCMSThreads=8
```

Serial garbage collection

Java's serial garbage collection works on a single thread. When executing, it freezes all other threads until garbage collection operations have concluded. Due to the thread-freezing nature of serial garbage collection, it is only feasible for very small programs.

To manually invoke the serial garbage collection algorithm for your application, use the following command-line option:

```
| -XX:+UseSerialGC
```

Parallel garbage collection

Prior to Java 9, the parallel garbage collection algorithm was the default garbage collector. It uses multiple threads but freezes all non-garbage collection threads in the application until garbage collection functions have completed, just like the serial garbage collection algorithm.

G1 garbage collection

The G1 garbage collection algorithm was created for use with large memory heaps. This approach involves segmenting the memory heap into regions. Garbage collection, using the G1 algorithm, takes place in parallel with each heap region.

Another part of the G1 algorithm is that when memory is deallocated, the heap space is compacted. Unfortunately, the compacting operation takes place using the *Stop the World* approach.

The G1 garbage collection algorithm also prioritizes the regions based on those that have the most garbage to be collected.



The G1 name refers to Garbage First.

To manually invoke the G1 garbage collection algorithm for your application, use the following command-line option:

```
| -XX:+UseG1GC
```

Garbage collection options

Here is a list of JVM sizing options:

Sizing description	JVM option flag
Sets the initial heap size (young space plus tenured space).	-XX:InitialHeapSize=3g
Sets the maximum heap size (young space plus tenured space).	-XX:MaxHeapSize=3g
Sets the initial and maximum heap size (young space plus tenured space).	-Xms2048m -Xmx3g
Sets the initial size of young space.	-XX:NewSize=128m
Sets the maximum size of young space.	-XX:MaxNewSize=128m
Sets young space size. Uses ration of young versus tenured space. In the sample flag to the right, 3 means that young space will be three times smaller than tenured space.	-XX:NewRatio=3
Sets the size of single survivor space as a portion of Eden space size.	-XX:SurvivorRatio=15
Sets the initial size of the permanent space.	-XX:PermSize=512m
Sets the maximum size of the permanent space.	-XX:MaxPermSize=512m
Sets the size of the stack area dedicated to each thread in bytes.	-Xss512k
Sets the size of the stack area dedicated to each thread in Kbytes.	-XX:ThreadStackSize=512
Sets the maximum size of off-heap memory available to the JVM.	-XX:MaxDirectMemorySize=3g

Here is a list of young garbage collection options:

Young garbage collection tuning option	Flag
Sets the initial value for the number of collections before an object will be promoted from young to tenured space. This is referred to as the tenuring threshold .	-XX:InitialTenuringThreshold=16
Sets the maximum value for tenuring threshold.	-XX:MaxTenuringThreshold=30
Sets the maximum object size allowed to be allocated in young space. If an object is larger than the maximum size it will be allocated to tenured space and bypass young space.	-XX:PretenureSizeThreshold=3m
This can be used to promote all young objects surviving the young collection to tenured space.	-XX:+AlwaysTenure
With this tag, objects from young space never get promoted to tenured space as long as the survivor space has sufficient room for them.	-XX:+NeverTenure
We can indicate that we want to use thread local allocation blocks in the young space. This is enabled by default.	-XX:+UseTLAB
Toggle this to allow the JVM to adaptively resize the TLAB (Thread Local Allocation Blocks) for threads.	-XX:+ResizeTLAB
Sets the initial size of TLAB for a thread.	-XX:TLABSize=2m
Sets the minimum allowable size of TLAB.	-XX:MinTLABSize=128k

Here is a list of **concurrent mark sweep (CMS)** tuning options:

CMS tuning option	Flag
Indicates that you want to solely use occupancy as a criterion for starting a CMS collection operation.	-XX:+UseCMSInitiatingOccupancyOnly
Sets the percentage CMS	

<p>generation occupancy to start a CMS collection cycle. If you indicate a negative number, you are telling the JVM you want to use <code>CMSTriggerRatio</code>.</p>	<p><code>-XX:CMSInitiating\OccupancyFraction=70</code></p>
<p>Sets the percentage CMS generation occupancy that you want to initiate a CMS collection for bootstrapping collection statistics.</p>	<p><code>-XX:CMSBootstrap\Occupancy=10</code></p>
<p>This is the percentage of <code>MinHeapFreeRatio</code> in CMS generation that is allocated prior to a CMS cycle starts.</p>	<p><code>-XX:CMSTriggerRatio=70</code></p>
<p>Sets the percentage of <code>MinHeapFreeRatio</code> in the CMS permanent generation that is allocated before starting a CMS collection cycle.</p>	<p><code>-XX:CMSTriggerPermRatio=90</code></p>
<p>This is the wait duration after a CMS collection is triggered. Use the parameter to specify how long the CMS is allowed to wait for young collection.</p>	<p><code>-XX:CMSWaitDuration=2000</code></p>
<p>Enables parallel remark.</p>	<p><code>-XX:+CMSParallel\RemarkEnabled</code></p>
<p>Enables parallel remark of survivor space.</p>	<p><code>-XX:+CMSParallel\SurvivorRemarkEnabled</code></p>
<p>You can use this to force young collection before the remark phase.</p>	<p><code>-XX:+CMSScavengeBeforeRemark</code></p>
<p>Use this to prevent scheduling remark if Eden used is below the threshold value.</p>	<p><code>-XX:+CMSScheduleRemark\EdenSizeThreshold</code></p>
<p>Sets the Eden occupancy percentage that you want CMS</p>	<p><code>-XX:CMSScheduleRemark\EdenPenetration=20</code></p>

to try and schedule a remark pause.	
This is where you want to start sampling Eden top at least before young generation occupancy reaches $1/4^{\text{th}}$ (in our sample to the right) of the size at which you want to schedule remark.	-XX:CMSScheduleRemark\SamplingRatio=4
You can select <code>variant=1</code> or <code>variant=2</code> of verification following remark.	-XX:CMSRemarkVerifyVariant=1
Elects to use the parallel algorithm for young space collection.	-XX:+UseParNewGC
Enables the use of multiple threads for concurrent phases.	-XX:+CMSConcurrentMTEnabled
Sets the number of parallel threads used for the concurrent phases.	-XX:ConcGCThreads=2
Sets the number of parallel threads you want used for <i>stop-the-world</i> phases.	-XX:ParallelGCThreads=2
You can enable incremental CMS (iCMS) mode.	-XX:+CMSIncrementalMode
If this is not enabled, CMS will not clean permanent space.	-XX:+CMSClassUnloadingEnabled
This allows <code>System.gc()</code> to trigger concurrent collection instead of a full garbage collection cycle.	-XX:+ExplicitGCInvokes\Concurrent
This allows <code>System.gc()</code> to trigger concurrent collection of permanent space.	-XX:+ExplicitGCInvokes\ConcurrentAndUnloadsClasses



iCMS (incremental concurrent mark sweep) mode is intended for servers with a small number of CPUs. It should not be employed on modern hardware.

Here are some miscellaneous garbage collection options:

Miscellaneous garbage collection options	Flag
This will cause the JVM to ignore any <code>System.gc()</code> method invocations by an application.	<code>-XX:+DisableExplicitGC</code>
This is the (soft reference) time to live in milliseconds per MB of free space in the heap.	<code>-XX:SoftRefLRU\PolicyMSPerMB=2000</code>
This is the use policy used to limit the time spent in garbage collection before an <code>OutOfMemory</code> error is thrown.	<code>-XX:+UseGCOverheadLimit</code>
This limits the proportion of time spent in garbage collection before an <code>OutOfMemory</code> error is thrown. This is used with <code>GCHHeapFreeLimit</code> .	<code>-XX:GCTimeLimit=95</code>
This sets the minimum percentage of free space after a full garbage collection before an <code>OutOfMemory</code> error is thrown. This is used with <code>GCHHeapFreeLimit</code> .	<code>-XX:GCHHeapFreeLimit=5</code>

Finally, here are some G1 specific options. Note that, these are all supported starting with JVM 6u26:

G1 garbage collection options	Flag
Size of the heap region. The default is 2,048 and the acceptable range is 1 MiB to 32 MiB.	<code>-XX:G1HeapRegionSize=16m</code>
This is the confidence coefficient pause prediction heuristics.	<code>-XX:G1ConfidencePercent=75</code>
This determines the minimum reserve in the heap.	<code>-XX:G1ReservePercent=5</code>
This is the garbage collection time per MMU--time	

slice in milliseconds.	-XX:MaxGCPauseMillis=100
This is the pause interval time slice per MMU in milliseconds.	- XX:GCPauseIntervalMillis=200



MiB stands for Mebibyte which is a multiple of bytes for digital information.

Java methods relevant to garbage collection

Let's look at two specific methods associated with garbage collection.

The System.gc() method

Although garbage collection is automatic in Java, you can make explicit calls to the `java.lang.System.gc()` method to aid in the debugging process. This method does not take any parameters and does not return any value. It is an explicit call that runs Java's garbage collector. Here is a sample implementation:

```
System.gc();
System.out.println("Garbage collected and unused
memory has been deallocated.");
```

Let's look at a more in-depth example. In the following code, we start by creating an instance of the `Runtime`, using `Runtime myRuntime = Runtime.getRuntime();` which returns a singleton. This gives us access to the JVM. After printing some header information and initial memory stats, we create an `ArrayList` with a size of `300000`. Then, we create a loop that generates `100000` array list objects. Lastly, we provide output in three passes, asking the JVM to invoke the garbage collector with `1` second pauses in between. Here is the source code:

```
package MyGarbageCollectionSuite;

import java.util.ArrayList;
import java.util.concurrent.TimeUnit;

public class GCVerificationTest
{
    public static void main(String[] args) throws
        InterruptedException
    {
        // Obtain a Runtime instance (to communicate
        // with the JVM)
        Runtime myRuntime = Runtime.getRuntime();

        // Set header information and output initial
        // memory stats
        System.out.println("Garbage Collection
        Verification Test");
        System.out.println("-----
        -----");
        System.out.println("Initial JVM Memory: " +
        myRuntime.totalMemory() +
        "\tFree Memory: " + myRuntime.freeMemory());

        // Use a bunch of memory
        ArrayList<Integer> AccountNumbers = new
        ArrayList<>(300000);
        for (int i = 0; i < 100000; i++)
        {
            AccountNumbers = new ArrayList<>(3000);
```

```

        AccountNumbers = null;
    }

    // Provide update with with three passes
    for (int i = 0; i < 3; i++)
    {
        System.out.println("-----");
        System.out.println("Free Memory before
collection number " +
(i+1) + ":" + myRuntime.freeMemory());
        System.gc();
        System.out.println("Free Memory after
collection number " +
(i+1) + ":" + myRuntime.freeMemory());
        TimeUnit.SECONDS.sleep(1); // delay thread
        1 second
    }

}

}

```

As you can see from the following output, the garbage collector did not reallocate all of the 'garbage' during the first or even the second pass:

Garbage Collection Verification Test	
Initial JVM Memory: 514850816 Free Memory: 509439928	

Free Memory before collection number 1: 768241776	
Free Memory after collection number 1: 888052656	

Free Memory before collection number 2: 888052656	
Free Memory after collection number 2: 887536992	

Free Memory before collection number 3: 887536992	
Free Memory after collection number 3: 888061280	

There is an alternative to using the `System.gc()` method to invoke the garbage collector. In our example, we could have used `myRuntime.gc()`, our earlier singleton example.

The `finalize()` method

You can think of Java's garbage collector as a death dealer. When it removes something from memory, it is gone. This so-called death dealer is not without compassion as it provides each method with their final last words. The objects give their *last words* through a `finalize()` method. If an object has a `finalize()` method, the garbage collector invokes it before the object is removed and the associated memory deallocated. The method takes no parameters and has a return type of `void`.

The `finalize()` method is only called once and there can be variability when it is run. Certainly, the method is invoked before it is removed, but when the garbage collector runs is dependent on the system. If, as an example, you have a relatively small app that is running a memory-rich system, the garbage collector might not run at all. So, why include a `finalize()` method at all? It is considered poor programming practice to override the `finalize()` method. That being said, you can use the method if needed. In fact, you can add code there to add a reference to your object to ensure it is not removed by the garbage collector. Again, this is not advisable.

Because all objects in Java, even the ones you create yourself are child classes of `java.lang.Object`, every object in Java has a `finalize()` method.

The garbage collector, as sophisticated as it is, might not close databases, files, or network connections the way you want it done. If your application requires specific considerations when its objects are collected, you can override the object's `finalize()` method.

Here is an example implementation that demonstrates a use case for when you might want to override an object's `finalize()` method:

```
public class Animal
{
    private static String animalName;
    private static String animalBreed;
    private static int objectTally = 0;

    // constructor
    public Animal(String name, String type)
    {
        animalName = name;
        animalBreed = type;
    }

    // finalize()
    protected void finalize()
    {
        System.out.println("Object destroyed: " + animalName);
        objectTally--;
    }
}
```

```

// increment count of object
    ++objectTally;
}

protected void finalize()
{
    // decrement object count each time this method
    // is called by the garbage collector
    --objectTally;

    //Provide output to user
    System.out.println(animalName + " has been
        removed from memory.");

    // condition for 1 animal (use singular form)
    if (objectTally == 1)
    {
        System.out.println("You have " + objectTally + "
            animal remaining.");
    }

    // condition for 0 or greater than 1
    // animals (use plural form)
    else
    {
        System.out.println("You have " + objectTally + "
            animals remaining.");
    }
}
}

```

As you can see in the preceding code, the `objectTally` count is incremented each time an object of type `Animal` is created and decremented when one is removed by the garbage collector.



Overriding an object's `finalize()` method is usually discouraged. The `finalize()` method should normally be declared as `protected`.

Pre-Java 9 garbage collection

Java's garbage collection is not new to Java 9, it has existed since the initial release of Java. Java has long had a sophisticated garbage collection system that is automatic and runs in the background. By running in the background, we are referring to garbage collection processes running during idle times.



Idle times refer to the time in between input/output such as between keyboard input, mouse clicks, and output generation.

This automatic garbage collection has been one of the key factors in developers selecting Java for their programming solutions. Other programming languages such as C# and Objective-C have implemented garbage collection following the success of the Java platform.

Let's next take a look at the following listed concepts before we look at the changes to garbage collection in the Java 9 platform:

- Visualizing garbage collection
- Garbage collection upgrades in Java 8
- Case study - Games written with Java

Visualizing garbage collection

It can be helpful to visualize how garbage collection works and, perhaps more importantly, the need for it. Consider the following code snippet that progressively creates the string `Garbage`:

```
001 String var = new String("G");
002 var += "a";
003 var += "r";
004 var += "b";
005 var += "a";
006 var += "g";
007 var += "e";
008 System.out.println("Your completed String
is: " + var + ".");
```

Clearly, the preceding code generates the output provided as follows:

```
| Your completed String is Garbage.
```

What might not be clear is that the sample code results in five unreferenced string objects. This is due, in part, because strings are immutable. As you can see in the following illustration, with each successive line of code, the referenced object is updated and an additional object becomes unreferenced:

Line of Code	001	002	003	004	005	006	007
Code	<code>String var = new String("G");</code>	<code>var+="a";</code>	<code>var+="r";</code>	<code>var+="b";</code>	<code>var+="a";</code>	<code>var+="g";</code>	<code>var+="e";</code>
Referenced memory	<code>var → "G"</code>	<code>var → "Ga"</code>	<code>var → "Gar"</code>	<code>var → "Garb"</code>	<code>var → "Garba"</code>	<code>var → "Garbag"</code>	<code>var → "Garbage"</code>
Unreferenced memory		<code>"G"</code>	<code>"G" "Ga"</code>	<code>"G" "Ga" "Gar"</code>	<code>"G" "Ga" "Gar" "Garb"</code>	<code>"G" "Ga" "Gar" "Garb" "Garba"</code>	<code>"G" "Ga" "Gar" "Garb" "Garbag"</code>

The preceding unreferenced objects listed certainly will not break the memory bank, but it is indicative of how quickly a large number of unreferenced objects can accumulate.

Garbage collection upgrades in Java 8

As of Java 8, the default garbage collection algorithm was the parallel garbage collector. Java 8 was released with some improvements to the G1 garbage collection system. One of these improvements was the ability to use the following command-line option to optimize the heap memory by removing duplicative string values:

```
| -XX:+UseStringDeduplication
```

The G1 garbage collector can view the character arrays when it sees a string. It then takes the value and stores it with a new, weak reference to the character array. If the G1 garbage collector finds a string with the same hash code, it will compare the two strings with a character-by-character review. If a match is found, both strings end up pointing to the same character array. Specifically, the first string will point to the character array of the second string.

This method can require substantial processing overhead and should only be used if deemed beneficial or absolutely necessary.

Case study - Games written with Java

Multiplayer games require extensive management techniques, both for server and client systems. The JVM runs the garbage collection thread in a low-priority thread and periodically runs. Server administrators previously used an incremental garbage collection schema using the now deprecated `-Xincgc` command-line option to avoid **server stalls** that occur when the server is overloaded. The goal is to have garbage collection run more frequently and with much shorter execution cycles each time.



When considering memory usage and garbage collection, it is important to use as little memory on the target system as possible and to limit pauses for garbage collection to the extent feasible. These tips are especially important for games, simulations, and other applications that require real-time performance.

The JVM manages the heap where Java memory is stored. The JVM starts with a small heap by default and grows as additional objects are created. The heap has two partitions--young and tenured. When objects are initially created, they are created in the young partition. Persistent objects are moved to the tenure partition. The creation of objects is usually very quick with not much more than pointer incrementation. Processing in the young partition is much faster than that of the tenured partition. This is important because it applies to the overall app, or in our case, a game's efficiency.

It becomes important for us to monitor our game's memory usage and when garbage collection occurs. To monitor garbage collection, we can add the verbose flag (`-verbose:gc`) when we launch our game such as with the following example:

```
| java -verbose:gc MyJavaGameClass
```

The JVM will then provide a line of formatted output for each garbage collection. Here is the format of the verbose GC output:

```
| [<TYPE> <MEMORY USED BEFORE> -> MEMORY USED AFTER  
| (TOTAL HEAP SIZE), <TIME>]
```

Let's look at two examples. In this first example, we see `GC` for type which refers to the young partition we previously discussed:

```
| [GC 31924K -> 29732K(42234K), 0.0019319 secs]
```

In this second example, `Full GC` indicates that the garbage collection action was taken on the tenured partition of the memory heap:

```
| [Full GC 29732K -> 10911K(42234K), 0.0319319 secs]
```

You can obtain more detailed information from the garbage collector using the `-XX:+PrintGCDetails` option as shown here:

```
| java -verbose:gc -XX:+PrintGCDetails MyJavaGameClass
```

Collecting garbage with the new Java platform

Java came out of the gate with automatic garbage collection, making it a development platform of choice for many programmers. It was commonplace to want to avoid manual memory management in other programming languages. We have looked in-depth at the garbage collection system to include the various approaches, or algorithms, used by the JVM. Java 9 includes some relevant changes to the garbage collection system and was the focus of three **Java Enhancement Program (JEP)** issues. Those issues are listed here:

- Default garbage collection (JEP 248)
- Deprecated garbage collection combinations (JEP 214)
- Unified garbage collection logging (JEP 271)

We will review each one of these garbage collection concepts and their corresponding **Java Enhancement Plan (JEP)** issue in the following sections.

Default garbage collection

We previously detailed the following garbage collection approaches used by the JVM prior to Java 9. These are still plausible garbage collection algorithms:

- CMS garbage collection
- Serial garbage collection
- Parallel garbage collection
- G1 garbage collection

Let's briefly recap each of these approaches:

- **CMS garbage collection:** The CMS garbage collection algorithm scans heap memory using multiple threads. Using this approach, the JVM marks objects for removal and then makes a sweep to actually remove them.
- **Serial garbage collection:** This approach uses a thread-freezing schema on a single thread. When the garbage collection is in progress, it freezes all other threads until garbage collection operations have concluded. Due to the thread-freezing nature of serial garbage collection, it is only feasible for very small programs.
- **Parallel garbage collection:** This approach uses multiple threads but freezes all non-garbage collection threads in the application until garbage collection functions have completed, just like the serial garbage collection algorithm.
- **G1 garbage collection:** This is the garbage collection algorithm with the following characteristics:
 - Is used with large memory heaps
 - Involves segmenting the memory heap into regions
 - Takes place in parallel with each heap region
 - Compacts the heap space when memory is deallocated
 - Compacting operations take place using the *Stop the World* approach
 - Prioritizes the regions based on those that have the most garbage to be collected

Prior to Java 9, the parallel garbage collection algorithm was the default garbage collector. In Java 9, the G1 garbage collector is the new default implementation of Java's memory management system. This is true for both 32 and 64-bit server configurations.

Oracle assessed that the G1 garbage collector, mostly due to its low-pause nature, was a better performing garbage collection method than the parallel approach. This change was predicated on the following concepts:

- It is important to limit latency
- Maximizing throughput is less important than limiting latency
- The G1 garbage collection algorithm is stable

There are two assumptions involved with making the G1 garbage collection method the default method over the parallel approach:

- Making G1 the default garbage collection method will significantly increase its use. This increased usage might unveil performance or stability issues not realized before Java 9.
- The G1 approach is more processor-intensive than the parallel approach. In some use cases, this could be somewhat problematic.

On the surface this change might seem like a great step for Java 9 and that very well might be the case. Caution, however, should be used when blindly accepting this new default collection method. It is recommended that systems be tested if switching to G1 to ensure your applications do not suffer from performance degradation or have unexpected issues that are caused by the use of G1. As previously suggested, G1 has not benefited from the widespread testing that the parallel method has.

This last point about the lack of widespread testing is significant. Making G1 the default automatic memory management (garbage collection) system with Java 9 is tantamount to turning developers into unsuspecting testers. While no major problems are expected, knowing that there is potential for performance and stability issues when using G1 with Java 9 will place greater emphasis on testing your Java 9 applications.

Deprecated garbage collection combinations

Oracle has been great about depreciating features, APIs, and libraries before removing them from a new release to the Java platform. With this schema in place, language components that were deprecated in Java 8 are subject for removal in Java 9. There are a few garbage collection combinations that were deemed to be rarely used and depreciated in Java 8. Those combinations, listed here, have been removed in Java 9:

- DefNew + CMS
- ParNew + SerialOld
- Incremental CMS

These combinations, in addition to having been rarely used, introduced an unneeded level of complexity to the garbage collection system. This resulted in an extra drain on system resources without providing a commensurate benefit to the user or developer.

The following listed garbage collection configurations were affected by the aforementioned depreciation in the Java 8 platform:

Garbage collection configuration	Flag(s)
DefNew + CMS	-XX:+UseParNewGC -XX:UseConcMarkSweepGC
ParNew + SerialOld	-XX:+UseParNewGC
ParNew + iCMS	-Xincgc
ParNew + iCMS	-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC

Defnew + ICMS	<pre>-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC -XX:-UseParNewGC</pre>
---------------	---

The **Java Enhancement Program 214 (JEP 214)** removed garbage collection combinations deprecated in JDK 8. Those combinations are listed above along with the flags that control those combinations. In addition, the flags to enable CMS foreground collections were removed and are not present in JDK 9. Those flags are listed as follows:

Garbage collection combinations	Flag
CMS foreground	<code>-XX:+UseCMSCompactAtFullCollection</code>
CMS foreground	<code>-XX+CMSFullGCsBeforeCompaction</code>
CMS foreground	<code>-XX+UseCMSCollectionPassing</code>

The only assessed downside to the removal of the deprecated garbage collection combinations is that applications that use JVM start up files with any of the flags listed in this section, will need to have their JVM start up files modified to remove or replace the old flags.

Unified garbage collection logging

The **Java Enhancement Program #271 (JEP-271)** titled, *Unified GC Logging*, is intended to re-implement garbage collection logging using the unified JVM logging framework that was previously introduced with JEP-158. So, let's first review the Unified JVM Logging (JEP-158).

Unified JVM logging (JEP-158)

Creating a unified logging schema for the JVM was the central goal of JEP-158. Here is a high-level list of the goals of the JEP:

- Create a JVM-wide set of command-line options for all logging operations
- Use categorized tags for logging
- Provide six levels of logging:
 - Error
 - Warning
 - Information
 - Debug
 - Trace
 - Develop



This is not an exhaustive list of goals. We will discuss JEP-158 in greater detail in [Chapter 14, Command Line Flags](#).

The changes to the JVM, in the context of logging, can be categorized into:

- Tags
- Levels
- Decorations
- Output
- Command-line options

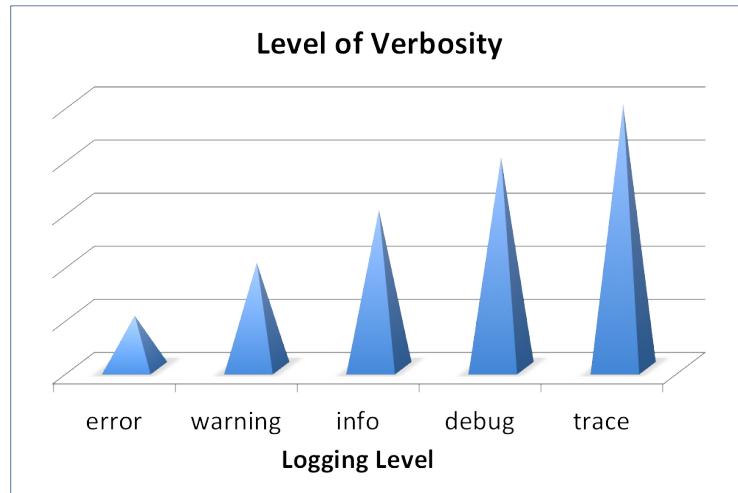
Let's briefly look at these categories.

Tags

Logging tags are identified in the JVM and can be changed in source code if needed. The tags should be self-identifying, such as `gc` for garbage collection.

Levels

Each log message has an associated level. As previously listed, the levels are error, warning, information, debug, trace, and develop. The following chart shows how the levels have an increasing level of verbosity in respect to how much information is logged:



Decorations

In the context of Java 9's logging framework, decorations is metadata about the log message. Here is the alphabetic list of decorations that are available:

- level
- pid
- tags
- tid
- time
- timemillis
- timenanos
- uptime
- uptimemillis
- uptimenanos

For an explanation of these decorations, please refer to [Chapter 14, *Command Line Flags*](#).

Output

The Java 9 logging framework supports three types of output:

- stderr: Provides output to stderr
- stdout: Provides output to stdout
- text file: Writes the output to text files

Command-line options

A new command-line option was added to the logging framework to provide overall control of the JVM's logging operations. The `-xlog` command-line option has an extensive array of parameters and possibilities. Here is one example:

```
| -xlog:gc+rt*=debug
```

In this example, we are telling the JVM to take the following actions:

- Log all messages tagged with, at a minimum, the `gc` and `rt` tags
- Use the `debug` level
- Provide output to `stdout`

Unified GC logging (JEP-271)

Now that we have a general understanding of the changes to Java 9's logging framework, let's look at what changes JEP-271 introduced. In this section we will look at the following areas:

- Garbage collection logging options
- The `gc` tag
- Macros
- Additional considerations

Garbage collection logging options

Here is a list of garbage collection logging options and flags we had available to us before the introduction of Java 9's logging framework:

Garbage collection logging option	JVM option flag(s)
This prints the basic garbage collection information.	<code>-verbose:gc</code> or <code>-XX:+PrintGC</code>
This will print more detailed garbage collection information.	<code>-XX:+PrintGCDetails</code>
You can print timestamps for each garbage collection event. The seconds are sequential and begin from the JVM start time.	<code>-XX:+PrintGCTimeStamps</code>
You can print date stamps for each garbage collection event. Sample format: 2017-07-26T03:19:00.319+400:[GC . . .]	<code>-XX:+PrintGCDateStamps</code>
You can use this flag to print timestamps for individual garbage collection work thread tasks.	<code>-XX:+PrintGC\TaskTimeStamps</code>
Using this you can redirect garbage collection output to a file instead of the console.	<code>-Xloggc:</code>
You can print detailed information regarding young space following each collection cycle.	<code>-XX:+Print\TenuringDistribution</code>
You can use this flag to print TLAB allocation statistics.	<code>-XX:+PrintTLAB</code>
Using this flag, you can print the times for reference processing (that is, weak, soft,	<code>-XX:+PrintReferenceGC</code>

and so on) during <i>stop-the-world</i> pauses.	
This reports if the garbage collection is waiting for native code to unpin objects in memory.	-XX:+PrintJNIGCStalls
This will print a pause summary after each <i>stop-the-world</i> pause.	-XX:+PrintGC\ApplicationStoppedTime
This flag will print time for each concurrent phase of garbage collection.	-XX:+PrintGC\ApplicationConcurrentTime
Using this flag will print a class histogram after a full garbage collection.	-XX:+Print\ClassHistogramAfterFullGC
Using this flag will print a class histogram before a full garbage collection.	-XX:+Print\ClassHistogramBeforeFullGC
This creates a heap dump file after full garbage collection.	-XX:+HeapDump\AfterFullGC
This creates a heap dump file before full garbage collection.	-XX:+HeapDump\BeforeFullGC
This creates a heap dump file in an out-of-memory condition.	-XX:+HeapDump\OnOutOfMemoryError
You use this flag to specify the path where you want your heap dumps saved on your system.	-XX:HeapDumpPath=<path>
You can use this to print CMS statistics, <i>if n >= 1</i> . Applies specifically to CMS only.	-XX:PrintCMSStatistics=2
This will print CMS initialization details. Applies specifically to CMS only.	-XX:+Print\CMSInitiationStatistics
You can use this flag to print additional information concerning free lists. Applies specifically to CMS only.	-XX:PrintFLSStatistics=2
You can use this flag to print additional information concerning free lists. Applies specifically to CMS only.	-XX:PrintFLSCensus=2
You can use this flag to print detailed diagnostic information following a	-XX:+PrintPromotionFailure

<p>promotion (young to tenure) failure. Applies specifically to CMS only.</p>	
<p>This flag allows you to dump useful information regarding the state of the CMS old generation when a promotion (young to tenure) failure occurs. Applies specifically to CMS only.</p>	<p>-XX:+CMSDumpAt\PromotionFailure</p>
<p>When the <code>-XX:+CMSDumpAt\PromotionFailure</code> flag is used, you can use - <code>XX:+CMSPrint\ChunksInDump</code> to include additional details regarding free chunks. Applies specifically to CMS only.</p>	<p>-XX:+CMSPrint\ChunksInDump</p>
<p>When using the <code>-XX:+CMSPrint\ChunksInDump</code> flag, you can include additional information about the allocated objects using the <code>-XX:+CMSPrint\ObjectsInDump</code> flag. Applies specifically to CMS only.</p>	<p>-XX:+CMSPrint\ObjectsInDump</p>

The gc tag

We can use the `gc` tag with the `-xlog` option to inform the JVM to only log `gc` tagged items at the info level. As you will recall, this is similar to using `-xx:+PrintGC`. With both options, the JVM will log one line for each garbage collection operation.

It is important to note that the `gc` tag was not intended to be used on its own; rather, it is recommended that it be used in conjunction with other tags.

Macros

We can create macros to add logic to our garbage collection logging. Here is the general syntax for the log macro:

```
| log_<level>(Tag1[,...]) (fmtstr, ...)
```

Here is an example of a log macro:

```
| log_debug(gc, classloading) ("Number of objects  
| loaded: %d.", object_count)
```

The following example skeleton log macro shows how you can use the new Java 9 logging framework to create scripts for greater fidelity in logging:

```
LogHandle(gc, rt, classunloading) log;  
if (log.is_error())  
{  
    // do something specific regarding the 'error' level  
}  
  
if (log.is_warning())  
{  
    // do something specific regarding the 'warning'  
    level  
}  
  
if (log.is_info())  
{  
    // do something specific regarding the 'info' level  
}  
  
if (log.is_debug())  
{  
    // do something specific regarding the 'debug' level  
}  
  
if (log.is_trace())  
{  
    // do something specific regarding the 'trace' level  
}
```

Additional considerations

Here are some additional items to be considered in regards to garbage collection logging:

- Using the new `-xlog:gc` should produce similar results to the `-xx:+PrintGCDetails` command-line option and flag pairing
- The new `trace` level provides the level of detail previously provided with the `verbose` flag

Persistent issues

Even with the advent of Java 9, there were downsides to Java's garbage collection system. Because it is an automatic process, we do not have complete control of when the collector runs. We, as developers, are not in control of garbage collection, the JVM is. The JVM makes the decision when to run garbage collection. As you have seen earlier in this chapter, we can ask the JVM to run garbage collection using the `System.gc()` method. Despite our use of this method, we are guaranteed that our request will be honored or that it will be complied with in a timely manner.

Earlier in this chapter, we reviewed several approaches and algorithms for garbage collection. We discussed how we, as developers, can take control of the process. That assumes that we have the ability to take control of garbage collection. Even when we specify a specific garbage collection technique, for example using -
`xx:+UseConcMarkSweepGC` for CMS garbage collection, we are not guaranteed that the JVM will use that implementation. So, we can do our best to control how the garbage collector works, but should remember that the JVM has the ultimate authority regarding how, when, and if garbage collection occurs.

Our lack of complete control over garbage collection underscores the importance of writing efficient code with memory management in mind. In the next sections, we will examine how to write code to explicitly make objects eligible for garbage collection by the JVM.

Making objects eligible for garbage collection

An easy method for making objects available for garbage collection is to assign `null` to the reference variable that refers to the object. Let's review this example:

```
package MyGarbageCollectionSuite;

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

        // Declare and create new object.
        String junk = new String("Pile of Junk");

        // Output to demonstrate that the object
        has an active reference
        // and is not eligible for garbage collection.
        System.out.println(junk);

        // Set the reference variable to null.
        junk = null;

        // The String object junk is now eligible
        for garbage collection.

    }
}
```

As indicated in the in-code comments, once the string object reference variable is set to null, in this case using the `junk = null;` statement, the object becomes available for garbage collection.

In our next example, we will abandon an object by setting its reference variable to point to a different object. As you can see in the following code, that results in the first object being available for garbage collection:

```
package MyGarbageCollectionSuite;

public class GarbageCollectionExperimentTwo
{
    public static void main(String[] args)
    {
        // Declare and create the first object.
        String junk1 = new String("The first pile of
        Junk");
```

```

// Declare and create the second object.
String junk2 = new String("The second pile of
Junk");

// Output to demonstrate that both objects have
active references
// and are not eligible for garbage collection.
System.out.println(junk1);
System.out.println(junk2);

// Set the first object's reference to the
second object.
junk1 = junk2;

// The String "The first pile of Junk" is now
eligible for garbage collection.

}

}

```

Let's review one final method of making objects available for garbage collection. In this example, we have a single instance variable (`objectNbr`) that is a reference variable to an instance of the `GarbageCollectionExperimentThree` class. The class does not do anything interesting other than create additional reference variables to instances of the `GarbageCollectionExperimentThree` class. In our example, we set the `objectNbr2`, `objectNbr3`, `objectNbr4`, and `objectNbr5` references to `null`. Although these objects have instance variables and can refer to each other, their accessibility outside of the class has been terminated by setting their references to `null`. This makes them (`objectNbr2`, `objectNbr3`, `objectNbr4`, and `objectNbr5`) eligible for garbage collection:

```

package MyGarbageCollectionSuite;
{

    // instance variable
    GarbageCollectionExperimentThree objectNbr;

    public static void main(String[] args)
    {
        GarbageCollectionExperimentThree objectNbr2 = new
            GarbageCollectionExperimentThree();
        GarbageCollectionExperimentThree objectNbr3 = new
            GarbageCollectionExperimentThree();
        GarbageCollectionExperimentThree objectNbr4 = new
            GarbageCollectionExperimentThree();
        GarbageCollectionExperimentThree objectNbr5 = new
            GarbageCollectionExperimentThree();
        GarbageCollectionExperimentThree objectNbr6 = new
            GarbageCollectionExperimentThree();
        GarbageCollectionExperimentThree objectNbr7 = new
            GarbageCollectionExperimentThree();

        // set objectNbr2 to refer to objectNbr3
        objectNbr2.objectNbr = objectNbr3;
    }
}

```

```
// set objectNbr3 to refer to objectNbr4
objectNbr3.objectNbr = objectNbr4;

// set objectNbr4 to refer to objectNbr5
objectNbr4.objectNbr = objectNbr5;

// set objectNbr5 to refer to objectNbr2
objectNbr5.objectNbr = objectNbr2;

// set selected references to null
objectNbr2 = null;
objectNbr3 = null;
objectNbr4 = null;
objectNbr5 = null;

}

}
```

Summary

In this chapter we took an in-depth review of garbage collection as a critical pre-Java 9 platform component. Our review included object life cycle, garbage collection algorithms, garbage collection options, and methods related to garbage collection. We looked at upgrades to garbage collection in Java 8 and looked at a case study to help our understanding of modern garbage collection. We then turned our focus to the changes to garbage collection with the new Java 9 platform. Our exploration of garbage collection in Java 9 included looks at default garbage collection, depreciated garbage collection combinations, and unified garbage collection logging. We concluded our exploration of garbage collection by looking at a few garbage collection issues that persist, even after Java 9.

In the next chapter we will look at how to write performance tests using the **Java Microbenchmark Harness (JMH)**, a Java harness library for writing benchmarks for the JVM.

Microbenchmarking Applications with JMH

In the previous chapter, we took an in-depth review of **garbage collection** to include an object life cycle, garbage collection algorithms, garbage collection options, and methods related to garbage collection. We took a brief look at upgrades to garbage collection in Java 8 and focused on changes with the new Java 9 platform. Our exploration of garbage collection in Java 9 included looks at default garbage collection, deprecated garbage collection combinations, unified garbage collection logging, and garbage collection issues that persist, even after Java 9.

In this chapter, we will look at how to write performance tests using the **Java Microbenchmark Harness (JMH)**, a Java harness library for writing benchmarks for the **Java Virtual Machine (JVM)**. We will use Maven along with JMH to help illustrate the power of microbenchmarking with the new Java 9 platform.

Specifically, we will cover the following topics:

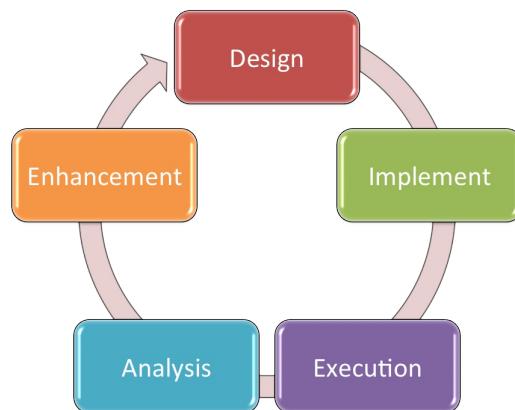
- Microbenchmarking overview
- Microbenchmarking with Maven
- Benchmarking options
- Techniques for avoiding microbenchmarking pitfalls

Microbenchmarking overview

Microbenchmarking is used to test the performance of a system. This differs from macrobenchmarking which runs tests on different platforms for efficiency comparison and subsequent analysis. With microbenchmarking, we typically target a specific slice of code on one system such as a method or loop. The primary purpose of microbenchmarking is to identify optimization opportunities in our code.

There are multiple approaches to benchmarking and we will focus on using the JMH tool. So, why benchmark at all? Developers do not always concern themselves with performance issues unless performance is a stated requirement. This can lead to post-deployment surprises that could have been avoided if microbenchmarking was conducted as part of the development process.

Microbenchmarking takes place across several phases of a process. As shown in the following diagram, the process involves design, implementation, execution, analysis, and enhancement:



In the **Design** phase, we determine our goals and design our microbenchmark accordingly. In the **Implement** phase, we are writing the microbenchmark and then, in the **Execution** phase, we actually run the test. With microbenchmarking results in hand, we interpret and analyze the results in the **Analysis** phase. This leads to code improvements in the **Enhancement** phase. Once our code has been updated, we redesign the microbenchmarking test, adjust the implementation, or go straight to the **Execution** phase. This is a cyclical process that continues until we have achieved the performance optimization we identified in our goals.

Approach to using JMH

Oracle's documentation indicates that the most ideal JMH use case is to use a Maven project that is dependent on the application's JAR files. They further recommend that microbenchmarking take place via the command-line and not from within an **Integrated Development Environment (IDE)**, as that could impact the results.



Maven, also referred to as Apache Maven, is a project management and comprehension tool that we can use to manage our application project build, reporting, and documentation.

To use JMH, we will use bytecode processors (annotations) to generate the benchmark code. We use Maven archetypes to enable JMH.

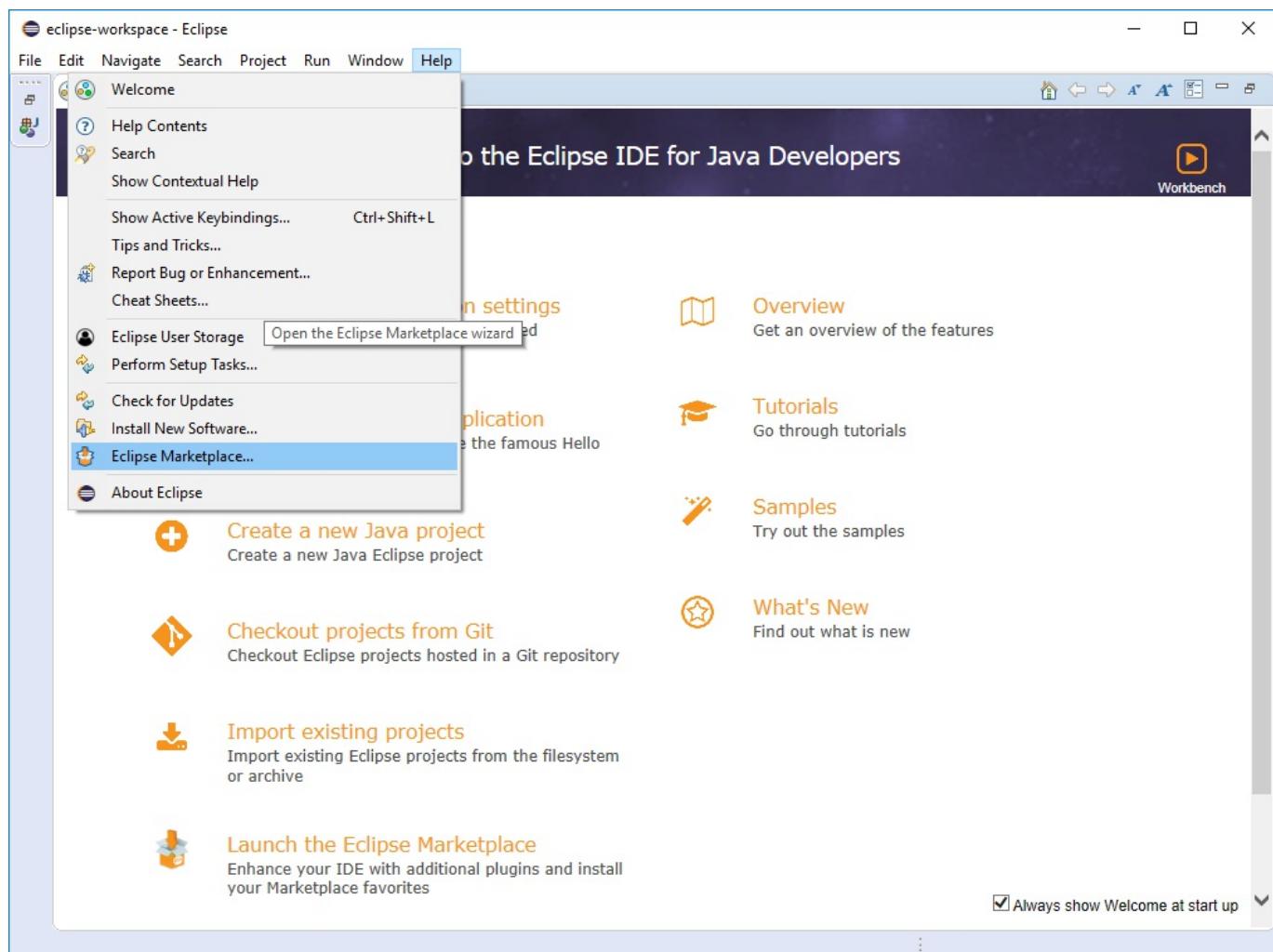
In order to test the JMH, we require an IDE with support for Maven and Java 9. If you do not yet have Java 9 or an IDE with Java 9 support, you can follow the steps in the next section.

Installing Java 9 and Eclipse with Java 9 support

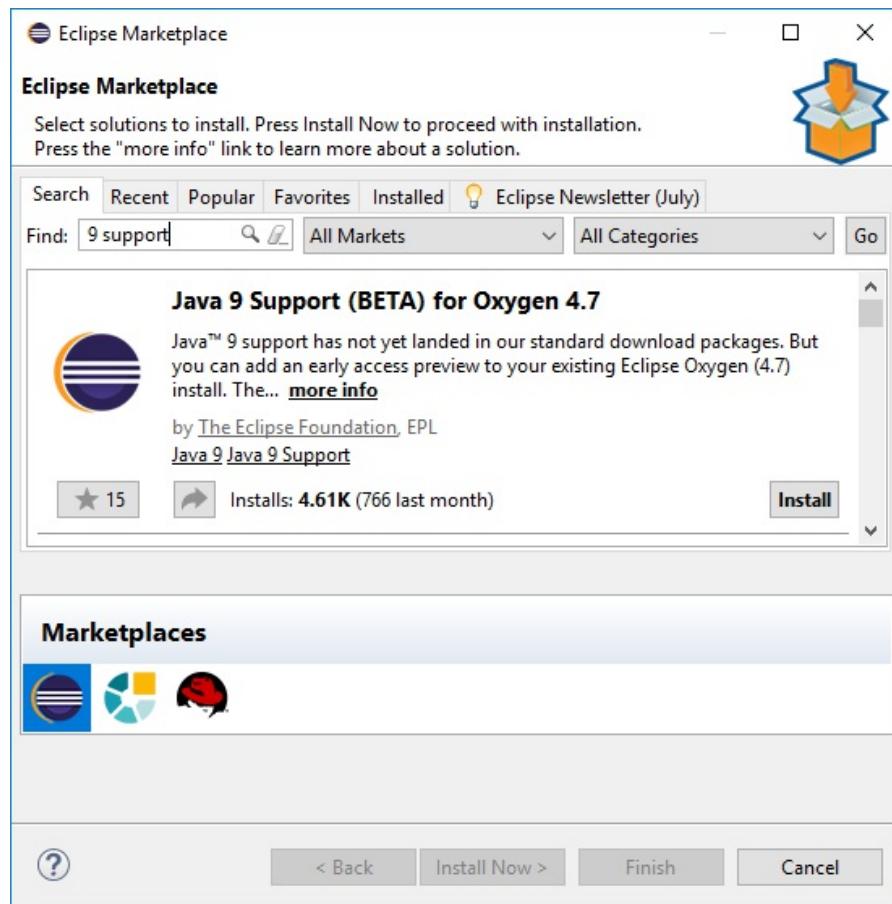
You can download and install Java 9 from the JDK 9 early access builds page--<http://jdk.java.net/9/>.

Once you have Java 9 installed, download the latest version of Eclipse. At the time of writing this book, that was Oxygen. Here is the relevant link--<https://www.eclipse.org/downloads/>.

The next step is to enable Java 9 support in your IDE. Launch Eclipse Oxygen and select Help | Eclipse Marketplace... , as shown in the following screenshot:



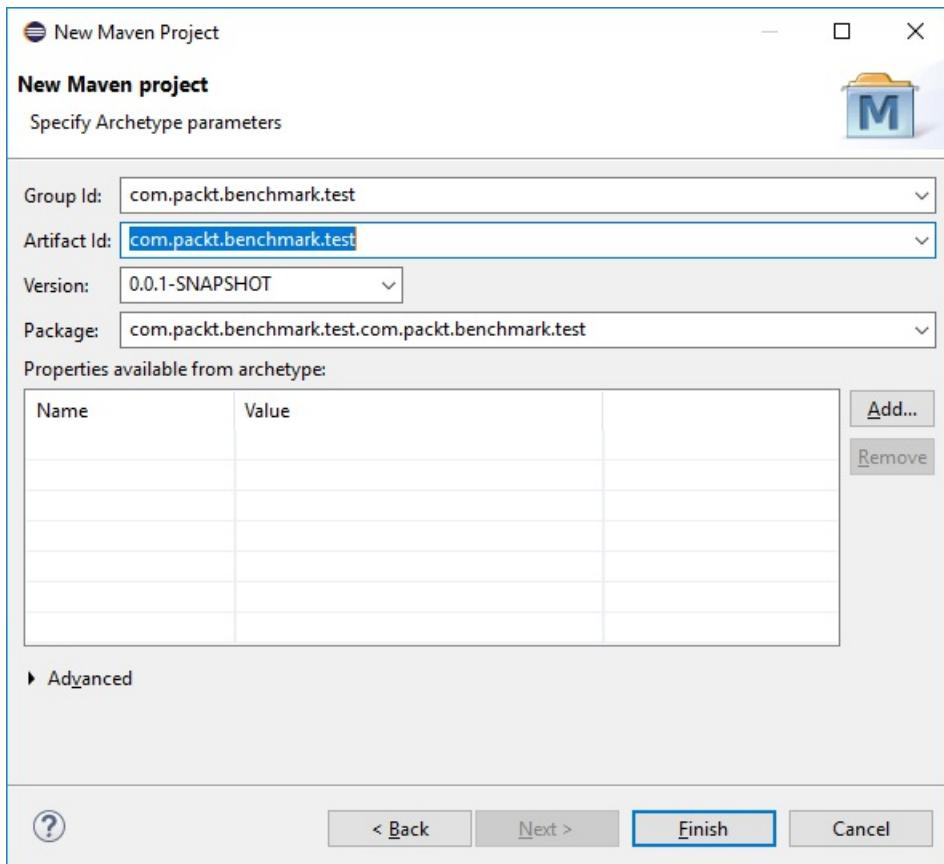
With the Eclipse Marketplace dialog window present search for Java 9 support using the search box. As you can see in the following screenshot, you will be presented with an Install button:



During the installation process, you will be required to accept the license agreement and, upon completion, you will be required to restart Eclipse.

Hands-on experiment

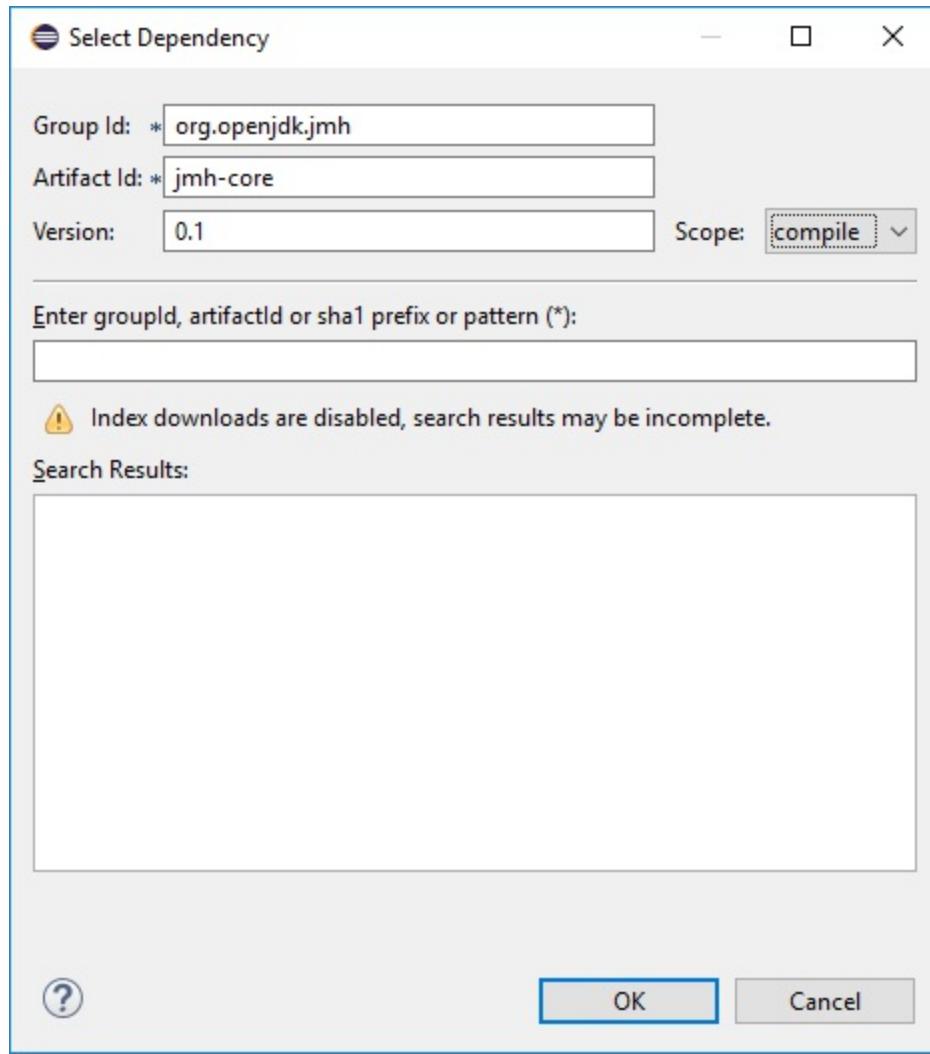
Now that we have Eclipse updated to support Java 9, you can run a quick test to determine if JMH is working on your development computer. Start by creating a new Maven project as illustrated in the following screenshot:



Next, we need to add a dependency. We can do this by editing the `pom.xml` file directly with the following code:

```
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-core</artifactId>
    <version>0.1</version>
</dependency>
```

Alternatively, we can use the dependencies tab to enter the data in a dialog window, as shown in the following screenshot. Using this form updates the `pom.xml` file with the preceding code:



Next, we need to write a class that contains a JMH method. This is just as an initial test to confirm our recently updated development environment. Here is sample code you can use for your test:

```
package com.packt.benchmark.test.com.packt.benchmark.test;

import org.openjdk.jmh.Main;

public class Test
{
    public static void main(String[] args)
    {
        Main.main(args);
    }
}
```

We can now compile and run our very simple test program. The results are provided in the Console tab, or the actual console if you are using the command-line. Here is what you will see:

```
<terminated> Test [Java Application] C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe (Jul 30, 2017, 8:04:30 PM)
No matching benchmarks. Miss-spelled regexp? Use -v for verbose output.
```

You can see that the program worked sufficiently to let us know that JMH is working. Of course, there was, as the output indicates, no benchmarks set up. We will take care of that in the next section.

Microbenchmarking with Maven

One approach to getting started with JMH is to use the JMH Maven archetype. The first step is to create a new JMH project. At our system's command prompt, we will enter the `mvn` command followed by a long set of parameters to create a new Java project and the necessary Maven `pom.xml` file:

```
| mvn archetype:generate -DinteractiveMode=false -DarchetypeGroupId=org.openjdk.jmh -DarchetypeArtifactId=jmh-jar -DgroupId=org.openjdk.jmh -DartifactId=jmh-benchmark -Dversion=1.19 -DarchetypeVersion=3.0.1 -DremoteRepositories=https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-archetype-plugin/3.0.1/
```

Once you enter the `mvn` command and the preceding detailed parameters, you will see the results reported to you via the Terminal. Depending on your level of use, you might see a large number of downloads from <https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-archetype-plugin/3.0.1/> and other similar repository sites.

You will also see an information section that informs you about the project build process:

```
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
[INFO] >>> maven-archetype-plugin:3.0.1:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO] <<< maven-archetype-plugin:3.0.1:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:3.0.1:generate (default-cli) @ standalone-pom ---
```

There will likely be additional plugin and other resources downloaded from the <https://repo.maven.apache.org/maven2/> repositories. Then, you will see an informational feedback component that lets you know the project is being generated in batch mode:

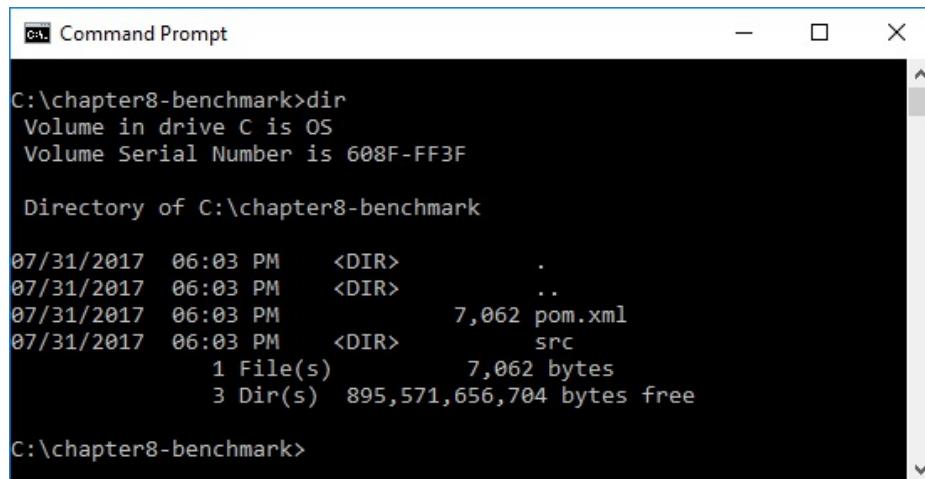
```
[INFO] Generating project in Batch mode
[INFO] Archetype [org.openjdk.jmh:jmh-java-benchmark-archetype:1.19] found in catalog remote
```

Finally, you will be presented with a set of parameters and a note that your project build was successful. As you can see with the following example, the process took less than 21 seconds to complete:

```
[INFO] -----
[INFO] Using following parameters for creating project from Archetype: jmh-java-benchmark-archetype:1.19
[INFO] -----
[INFO] Parameter: groupId, Value: com.packt
[INFO] Parameter: artifactId, Value: chapter8-benchmark
[INFO] Parameter: version, Value: 1.0
[INFO] Parameter: package, Value: com.packt
[INFO] Parameter: packageInPathFormat, Value: com/packt
[INFO] Parameter: package, Value: com.packt
[INFO] Parameter: groupId, Value: com.packt
[INFO] Parameter: artifactId, Value: chapter8-benchmark
[INFO] Parameter: version, Value: 1.0
[INFO] Project created from Archetype in dir: C:\chapter8-benchmark
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 20.753 s
[INFO] Finished at: 2017-07-31T18:03:27-05:00
[INFO] Final Memory: 18M/62M
[INFO] -----
```

C:\>

A folder will be created based on the parameter we included in the `-DartifactId` option. In our example, we used `-DartifactId=chapter8-benchmark`, and Maven created a `chapter8-benchmark` project folder:



```
Command Prompt

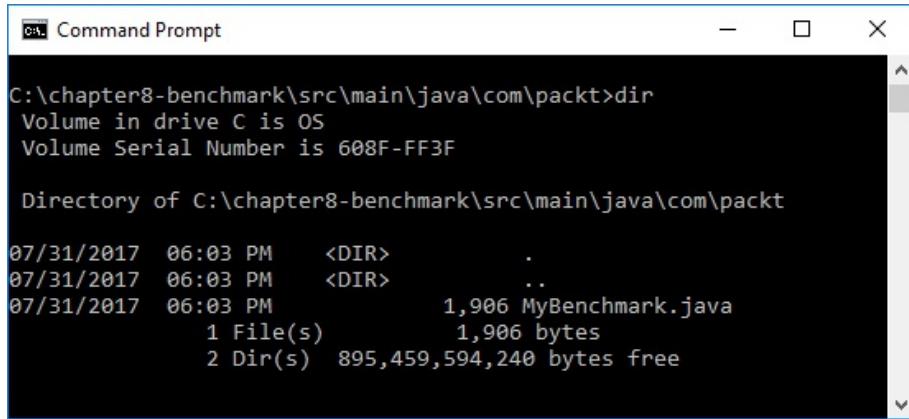
C:\chapter8-benchmark>dir
 Volume in drive C is OS
 Volume Serial Number is 608F-FF3F

 Directory of C:\chapter8-benchmark

07/31/2017  06:03 PM    <DIR>        .
07/31/2017  06:03 PM    <DIR>        ..
07/31/2017  06:03 PM            7,062 pom.xml
07/31/2017  06:03 PM    <DIR>        src
                           1 File(s)     7,062 bytes
                           3 Dir(s)   895,571,656,704 bytes free

C:\chapter8-benchmark>
```

You will see that Maven created the `pom.xml` file as well as a source (`src`) folder. In that folder, under the subdirectory structure of `C:\chapter8-benchmark\src\main\java\com\packt`, is the `MyBenchmark.java` file. Maven created a benchmark class for us:



```
C:\chapter8-benchmark\src\main\java\com\packt>dir
 Volume in drive C is OS
 Volume Serial Number is 608F-FF3F

 Directory of C:\chapter8-benchmark\src\main\java\com\packt

 07/31/2017  06:03 PM    <DIR>      .
 07/31/2017  06:03 PM    <DIR>      ..
 07/31/2017  06:03 PM           1,906 MyBenchmark.java
               1 File(s)     1,906 bytes
              2 Dir(s)   895,459,594,240 bytes free
```

Here are the contents of the `MyBenchmark.java` class created by the JMH Maven project creation process:

```
/*
 * Copyright (c) 2014, Oracle America, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or
 * without
 * modification, are permitted provided that the following
 * conditions are met:
 *
 * * Redistributions of source code must retain the above
 *   copyright notice,
 * this list of conditions and the following disclaimer.
 *
 * * Redistributions in binary form must reproduce the above
 *   copyright
 * notice, this list of conditions and the following
 * disclaimer in the
 * documentation and/or other materials provided with the
 * distribution.
 *
 * * Neither the name of Oracle nor the names of its
 *   contributors may be used
 * to endorse or promote products derived from this software
 * without
 * specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
 * CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
 * CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY,
 * OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS
```

```

* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
* WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
* OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
* ADVISED OF
* THE POSSIBILITY OF SUCH DAMAGE.
*/



package com.packt;

import org.openjdk.jmh.annotations.Benchmark;

public class MyBenchmark
{
    @Benchmark
    public void testMethod()
    {

        // This is a demo/sample template for building your JMH
        // benchmarks.
        //Edit as needed.
        // Put your benchmark code here.
    }
}

```

Our next step is to modify the `testMethod()` so that there is something to test. Here is the modified method we will use for the benchmark test:

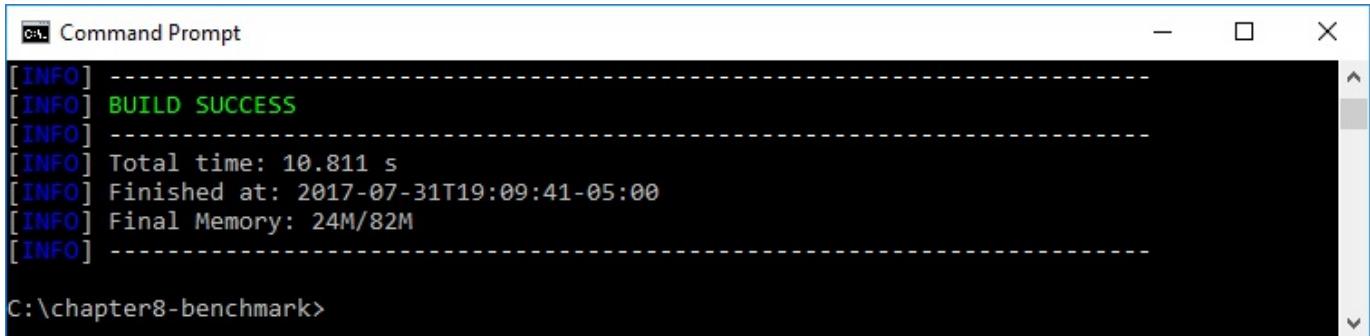
```

@Benchmark
public void testMethod()
{
    int total = 0;
    for (int i=0; i<100000; i++)
    {
        total = total + (i * 2 );
    }
    System.out.println("Total: " + total);
}

```

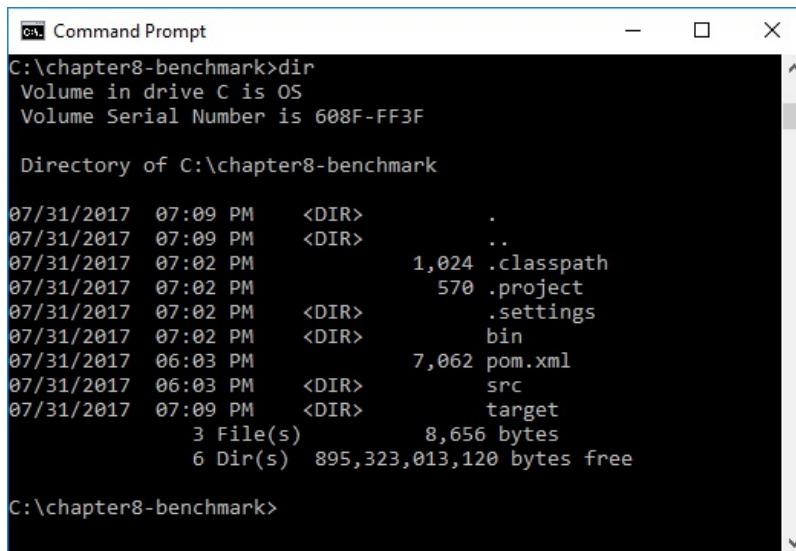
With our code edited, we will navigate back to the project folder, `C:\chapter8-benchmark`, in our example, and execute `mvn clean install` at the command prompt.

You will see several repository downloads, source compilations, plugin installations and, finally the Build Success indicator, as shown here:



```
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 10.811 s  
[INFO] Finished at: 2017-07-31T19:09:41-05:00  
[INFO] Final Memory: 24M/82M  
[INFO] -----  
  
C:\chapter8-benchmark>
```

You will now see `.classpath` and `.project` files as well as a new `.settings` and target subfolders in the project directory:



```
C:\chapter8-benchmark>dir  
Volume in drive C is OS  
Volume Serial Number is 608F-FF3F  
  
Directory of C:\chapter8-benchmark  
  
07/31/2017  07:09 PM    <DIR>          .  
07/31/2017  07:09 PM    <DIR>          ..  
07/31/2017  07:02 PM           1,024 .classpath  
07/31/2017  07:02 PM            570 .project  
07/31/2017  07:02 PM    <DIR>          .settings  
07/31/2017  07:02 PM    <DIR>          bin  
07/31/2017  06:03 PM           7,062 pom.xml  
07/31/2017  06:03 PM    <DIR>          src  
07/31/2017  07:09 PM    <DIR>          target  
                  3 File(s)        8,656 bytes  
                 6 Dir(s)   895,323,013,120 bytes free  
  
C:\chapter8-benchmark>
```

If you navigate to the `\target` subfolder, you will see that our `benchmarks.jar` file was created. This JAR contains what we need to run our benchmarks.

We can update our `MyBenchmark.java` file in an IDE, such as Eclipse. Then, we can execute `mvn clean install` again to overwrite our files. After the initial time, our builds will be much faster, as nothing will need to be downloaded. Here is a look at the output from the build process other than the first time:

```
C:\chapter8-benchmark>mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building JMH benchmark sample: Java 1.0
[INFO] -----
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ chapter8-benchmark ---
[INFO] Deleting C:\chapter8-benchmark\target
[INFO] -----
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ chapter8-benchmark ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\chapter8-benchmark\src\main\resources
[INFO] -----
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ chapter8-benchmark ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to C:\chapter8-benchmark\target\classes
[INFO] -----
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ chapter8-benchmark ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\chapter8-benchmark\src\test\resources
[INFO] -----
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ chapter8-benchmark ---
[INFO] No sources to compile
[INFO] -----
[INFO] --- maven-surefire-plugin:2.17:test (default-test) @ chapter8-benchmark ---
[INFO] No tests to run.
[INFO] -----
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ chapter8-benchmark ---
[INFO] Building jar: C:\chapter8-benchmark\target\chapter8-benchmark-1.0.jar
[INFO] -----
[INFO] --- maven-shade-plugin:2.2:shade (default) @ chapter8-benchmark ---
[INFO] Including org.openjdk.jmh:jmh-core:jar:1.19 in the shaded jar.
[INFO] Including net.sf.jopt-simple:jopt-simple:jar:4.6 in the shaded jar.
[INFO] Including org.apache.commons:commons-math3:jar:3.2 in the shaded jar.
[INFO] Replacing C:\chapter8-benchmark\target\benchmarks.jar with C:\chapter8-benchmark\target\chapter8-benchmark-1.0-shaded.jar
[INFO] -----
[INFO] --- maven-install-plugin:2.5.1:install (default-install) @ chapter8-benchmark ---
[INFO] Installing C:\chapter8-benchmark\target\chapter8-benchmark-1.0.jar to C:\Users\elavi\.m2\repository\com\packt\chapter8-benchmark\1.0\chapter8-benchmark-1.0.jar
[INFO] Installing C:\chapter8-benchmark\pom.xml to C:\Users\elavi\.m2\repository\com\packt\chapter8-benchmark\1.0\chapter8-benchmark-1.0.pom
[INFO] -----
[INFO] [BUILDSUCCESS]
[INFO] -----
[INFO] Total time: 3.388 s
[INFO] Finished at: 2017-07-31T19:26:01-05:00
[INFO] Final Memory: 22M/73M
[INFO] -----
```

Our last step is to run the benchmark tool. We can do that with the following command--`java -jar benchmarks.jar`. Even for small benchmarks on simplistic code, as with our example, the benchmarks could take some time to run. There will likely be several iterations including warmups to provide a more concise and valid set of benchmark results.

Our benchmark results are provided here. As you can see, the test ran for 8 minutes and 8 seconds:

```
cmd Command Prompt
Total: 319
24676.388 ops/s

Result "com.packt.MyBenchmark.testMethod":
23847.961 ±(99.9%) 772.746 ops/s [Average]
(min, avg, max) = (14864.509, 23847.961, 35528.242), stdev = 3271.857
CI (99.9%): [23075.215, 24620.708] (assumes normal distribution)

# Run complete. Total time: 00:08:08

Benchmark           Mode  Cnt    Score    Error  Units
MyBenchmark.testMethod  thrpt  200  23847.961 ± 772.746  ops/s

C:\chapter8-benchmark\target>
```

Benchmarking options

In the previous section, you learned how to run a benchmark test. In this section, we will look at the following configurable options for running our benchmarks:

- Modes
- Time units

Modes

The output of our benchmark results, from the previous section, included a **Mode** column that had the value of **thrpt** which is short for **throughput**. This is the default mode and there are an additional four modes. All JMH benchmark modes are listed and described as follows:

Mode	Description
All	Measures all other modes inclusively.
Average	This mode measures the average time for a single benchmark to run.
Sample Time	This mode measures the benchmark execution time and includes min and max times.
Single Shot Time	With this mode, there is no JVM warm up and the test is to determine how long a single benchmark method takes to run.
Throughput	This is the default mode and measures the number of operations per second the benchmark could be run.

To dictate which benchmark mode to use, you will modify your `@Benchmark` line of code to one of the following:

```
@Benchmark @BenchmarkMode (Mode.All)
@Benchmark @BenchmarkMode (Mode.Average)
@Benchmark @BenchmarkMode (Mode.SampleTime)
@Benchmark @BenchmarkMode (Mode.SingleShotTime)
@Benchmark @BenchmarkMode (Mode.Throughput)
```

Time units

In order to gain greater fidelity in benchmark output, we can designate a specific unit of time, listed here from shortest to longest:

- NANOSECONDS
- MICROSECONDS
- MILLISECONDS
- SECONDS
- MINUTES
- HOURS
- DAYS

In order to make this designation, we simply add the following code to our `@Benchmark` line:

```
| @Benchmark @BenchmarkMode(Mode.Average)  
| @OutputTimeUnit(TimeUnit.NANOSECONDS)
```

In the preceding example, we have designated the average mode and nanoseconds as the time unit.

Techniques for avoiding microbenchmarking pitfalls

Microbenchmarking is not something that every developer will have to worry about, but for those that do, there are several pitfalls that you should be aware of. In this section we will review the most common pitfalls and suggest strategies for avoiding them.

Power management

There are many subsystems that can be used to help you manage the balance between power and performance (that is, `cpufreq`). These systems can alter the state of time during benchmarks.

There are two suggested strategies to this pitfall:

- Disable any power management systems before running tests
- Run the benchmarks for longer periods

OS schedulers

Operating system schedulers, such as Solaris schedulers, help determine which software processes gain access to a system's resources. Use of these schedulers can result in unreliable benchmarking results.

There are two suggested strategies to this pitfall:

- Refine your system scheduling policies
- Run the benchmarks for longer periods

Time sharing

Time-sharing systems are used to help balance system resources. Use of these systems often results in irregular gaps between a thread's start and stop time. Also, CPU load will not be uniform and our benchmarking data will not be as useful to us.

There are two suggested strategies to avoid this pitfall:

- Test all code before running benchmarks to ensure things work as they should
- Use JMH to measure only after all threads have started or all threads have stopped

Eliminating dead-code and constant folding

Dead-code and constant folding are often referred to as redundant code and our modern compilers are pretty good at eliminating them. An example of dead-code is code that will never be reached. Consider the following example:

```
    . . .
    int value = 10;

    if (value != null)
    {
        System.out.println("The value is " + value + ".");
    } else
    {
        System.out.println("The value is null."); // This is
        a line of Dead-Code
    }

    . . .
```

In our preceding example, the line identified as dead-code is never reached since the variable `value` will never be equal to `null`. It is set to `10` immediately before the conditional `if` statement evaluates the variable.

The problem is that benchmarking code can sometimes be removed in the attempt to eliminate dead-code.

Constant folding is the compiler operation that occurs when compile-time constraints are replaced with actual results. The compiler performs constant folding to remove any redundant runtime computations. In the following example, we have a `final int` followed by a second `int` based on a mathematical calculation involving the first `int`:

```
    . . .
    static final int value = 10;

    int newValue = 319 * value;

    . . .
```

The constant folding operation would convert the two lines of the preceding code to the following:

```
|     int newValue = 3190;
```

There is one suggested strategy to this pitfall:

- Use the JMH API support to ensure your benchmarking code is not eliminated

Run-to-run variance

There are a plethora of issues that can drastically impact the run-to-run variance in benchmarking.

There are two suggested strategies to this pitfall:

- Run the JVM multiple times within every subsystem
- Use multiple JMH folks

Cache capacity

Dynamic randomly accessed memory (DRAM) is very slow. This can result in very different performance results during benchmarking.

There are two suggested strategies to this pitfall:

- Run multiple benchmarks with varying problem sets. Keep track of your memory footprint during tests.
- Use the `@State` annotation to dictate the JMH state. This annotation is used to define the instance's scope. There are three states:
 - `Scope.Benchmark`: The instance is shared across all threads that are running the same test.
 - `Scope.Group`: One instance is allocated per thread group.
 - `Scope.Thread`: Each thread will have its own instance. This is the default state.

Summary

In this chapter, we learned that the JMH is a Java harness library for writing benchmarks for the JVM. We experimented with writing performance tests using Maven along with JMH to help illustrate the procedures of microbenchmarking with the new Java 9 platform. We started with a microbenchmarking overview, then dove deep into microbenchmarking with Maven, reviewed benchmarking options, and concluded with a few techniques for avoiding microbenchmarking pitfalls.

In the next chapter, we will learn to write an application that is managing other processes and utilizes the modern process management API of the Java 9 platform.

Making Use of the ProcessHandle API

In the previous chapter, we discovered the **Java Microbenchmark Harness (JMH)**. We explored performance tests and how to write them using JMH, the Java library for writing benchmarks for the Java Virtual Machine. We started with an overview of microbenchmarking, then looked at microbenchmarking with Maven, reviewed benchmarking options, and concluded with techniques for avoiding microbenchmarking pitfalls.

In this chapter, we will focus on the updates to the `Process` class and the new `java.lang.ProcessHandle` API. Prior to Java 9, managing processes in Java was never easy, because Java was rarely used to automate the controlling of other processes. The API was insufficient with some features lacking and some tasks needed to be solved in a system specific manner. For example, in Java 8, giving a process access to its own **process identifier (PID)** was an unnecessarily difficult task.

In this chapter, the reader will gain all the knowledge that is needed to write an application that manages other processes and utilizes the modern process management API of Java.

We will cover the following topics in this chapter:

- What is and how to use the new `ProcessHandle` interface
- How to get the PID of the current process
- How to list different processes that run in the operating system
- How to effectively wait for an external process to finish
- How to terminate external processes

What are processes?

In this section, we will review what processes are in the context of Java application programming. If you are already familiar with processes, you might consider skipping this section.

Processes are executional units in the operating system. When you start a program, you start a process. When the machine boots the code, the first thing it does is, execute the boot process. This process then starts other processes that become the child of the boot process. These child processes may start other processes. This way, when the machine runs there are trees of processes running. When the machine does something, it is done in some code executing inside some process. The operating system also runs as several processes that execute simultaneously. Applications are executed as one or more processes. Most of the applications run as a single process but as an example, the Chrome browser starts several processes to do all the rendering and network communication operations that finally function as a browser.

To get a better idea about what processes are, start the task manager on Windows or the Activity Monitor on OS X and click on the Process tab. You will see the different processes that currently exist on the machine. Using these tools, you can look at the parameters of the processes, or you can kill an individual process.

The individual processes have their memory allocated for their work and they are not allowed to freely access each other's memory.

The execution unit scheduled by the operating system is a thread. A process consists of one or more threads. These threads are scheduled by the operating system scheduler and are executed in time slots.

With every operating system, processes have a process identifier, which is a number that identifies the process. No two processes can be active at a time, sharing the same PID. When we want to identify an active process in the operating system we use the PID. On Linux and other Unix-like operating systems, the `kill` command terminates a process. The argument to be passed to this program is the PID of the process, to terminate. Termination can be graceful. It is something like asking the process to exit. If the process decides not to, it can keep running. Programs can be prepared to stop upon such requests. For example, a Java application may add a `Thread` object

calling the `Runtime.getRuntime().addShutdownHook(Thread t)` method. The thread passed is supposed to start when the process is asked to stop and the thread can perform all tasks that the program has to do before it exits. However, there is no guarantee that it does start. It depends on the actual implementation.

The new ProcessHandle interface

There are two new interfaces and also their implementations in Java 9 supporting the handling of operating system processes. One of them is `ProcessHandle`, the other one is `ProcessHandle.Info`, a nested interface of the prior.

A `ProcessHandle` object identifies an operating system process and provides methods to manage the process. In prior versions of Java, this was possible only using operating system specific methods using the PID to identify the process. The major problem with this approach is that the PID is unique only while the process is active. When a process finishes, the operating system is free to reuse the PID for a new process. When we know only the PID of a process and check to see if the process is still running, what we are really doing is checking if there is an active process with that PID. Our process may be alive when we check it, but the next time our program queries the process state, it might be a different process.

The desktop and server operating systems try not to reuse the PID values for as long as possible. On some embedded systems the operating system may only use 16-bit to store the PID. When only 16-bit values are used, there is a greater chance that the PIDs will be reused. We can avoid this problem using the `ProcessHandle` API. We can receive a `ProcessHandle` and can call the `handle.isAlive()` method. This method will return `false` when the process finishes. This works even if the PID was reused.

Getting the PID of the current process

We can gain access to the PID of the processes via the handle. The `handle.getPid()` method returns a `Long` representing the numerical value of the PID. Since it is safer to access the processes through the handle, the importance of this method is limited. It may come in handy when our code wants to give information about itself to some other management tool. It is a common practice for programs to create a file that has the numeric PID as the name of the file. It may be a requirement that a certain program does not run in multiple processes. In that case, the code writes its own PID file to a specific directory. If a PID file with that name already exists, processing stops. If the previous process crashed and terminated without deleting the PID file, then the system manager can easily delete the file and start the new process. If the program hangs, then the system manager can easily kill the dead process if s/he knows the PID.

To get the PID of the current process, the call chain `ProcessHandle.current().getPid()` can be used.

Getting information about a process

To get information about a process, we need access to the `Info` object of the process. This is available through a `ProcessHandle`. We use a call to the `handle.info()` method to return it.

The `Info` interface defines query methods that deliver information about the process. These are:

- `command()` returns an `Optional<String>` containing the command that was used to start the process
- `arguments()` returns an `Optional<String[]>` that contains the arguments that were used on the command-line after the command to start the process
- `commandLine()` returns an `Optional<String>` that contains the whole command-line
- `startInstant()` returns an `Optional<Instant>`, which essentially represents the time the process was started
- `totalCpuDuration()` returns an `Optional<Duration>`, which represents the CPU time used by the process since it was started
- `user()` returns an `Optional<String>` that holds the name of the user the process belongs to

The values returned by these methods are all `Optional` because there is no guarantee that the actual operating system or the Java implementation can return the information. However, on most operating systems it should work and the returned values should be present.

The following sample code displays the information on a given process:

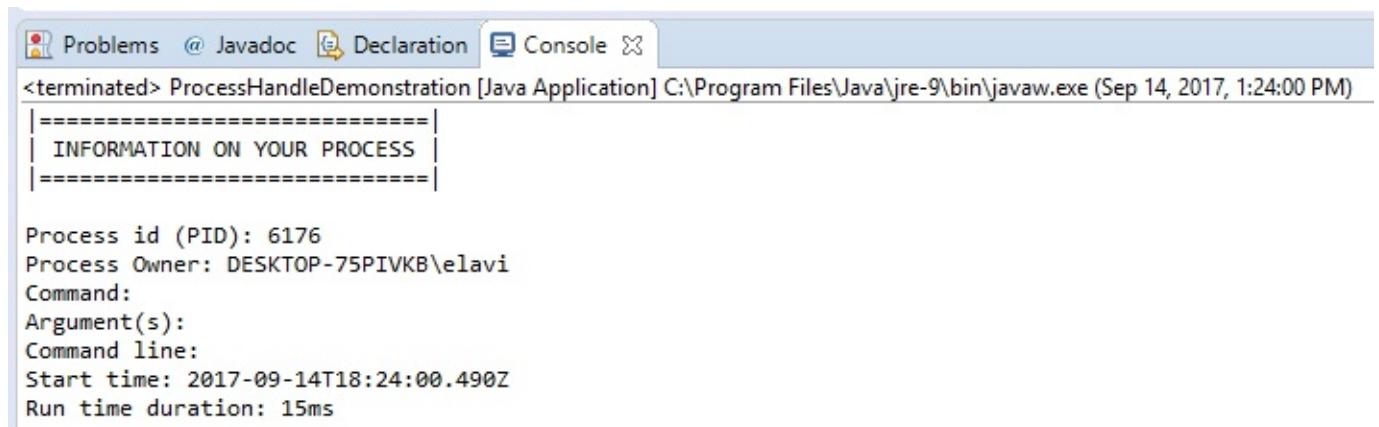
```
import java.io.IOException;
import java.time.Duration;
import java.time.Instant;
public class ProcessHandleDemonstration
{
    public static void main(String[] args) throws
        InterruptedException, IOException
    {
        provideProcessInformation(ProcessHandle.current());
        Process theProcess = new
            ProcessBuilder("SnippingTool.exe").start();
        provideProcessInformation(theProcess.toHandle());
```

```

        theProcess.waitFor();
        provideProcessInformation(theProcess.toHandle());
    }
    static void provideProcessInformation(ProcessHandle theHandle)
    {
        // get id
        long pid = ProcessHandle.current().pid();
        // Get handle information (if available)
        ProcessHandle.Info handleInformation = theHandle.info();
        // Print header
        System.out.println("|=====|");
        System.out.println("| INFORMATION ON YOUR PROCESS |");
        System.out.println("|=====|\n");
        // Print the PID
        System.out.println("Process id (PID): " + pid);
        System.out.println("Process Owner: " +
            handleInformation.user().orElse(""));
        // Print additional information if available
        System.out.println("Command: " +
            handleInformation.command().orElse(""));
        String[] args = handleInformation.arguments().orElse
            (new String[]{});
        System.out.println("Argument(s): ");
        for (String arg: args) System.out.printf("\t" + arg);
        System.out.println("Command line: " + handleInformation.
            commandLine().orElse(""));
        System.out.println("Start time: " +
            handleInformation.startInstant().
            orElse(Instant.now()).toString());
        System.out.printf("Run time duration: %sms%n",
            handleInformation.totalCpuDuration()
            .orElse(Duration.ofMillis(0)).toMillis());
    }
}

```

Here is the console output for the preceding code:



The screenshot shows a Java application window with the title bar 'ProcessHandleDemonstration [Java Application] C:\Program Files\Java\jre-9\bin\javaw.exe (Sep 14, 2017, 1:24:00 PM)'. The 'Console' tab is selected. The output in the console is:

```

=====
| INFORMATION ON YOUR PROCESS |
=====

Process id (PID): 6176
Process Owner: DESKTOP-75PIVKB\elavi
Command:
Argument(s):
Command line:
Start time: 2017-09-14T18:24:00.490Z
Run time duration: 15ms

```

Listing processes

Prior to Java 9, we did not have the means to obtain a list of active processes. With Java 9 it is now possible to get the processes in a stream. There are three methods that return a `Stream<ProcessHandle>`. One lists the child processes. The other lists all the descendants; the children and the children of those recursively. The third lists all the processes.

Listing children

To get the stream of process handles that can be used to control the children, the static method `processHandle.children()` should be used. This will create a snapshot of the children processes of the process represented by `processHandle` and create the `Stream`. Since processes are dynamic there is no guarantee that during the code execution, while our program attends to the handles, that all children processes are still active. Some of them may terminate and our process may spawn new children, perhaps from a different thread. Thus the code should not assume that each of the `ProcessHandle` elements of the stream represents an active and running process.

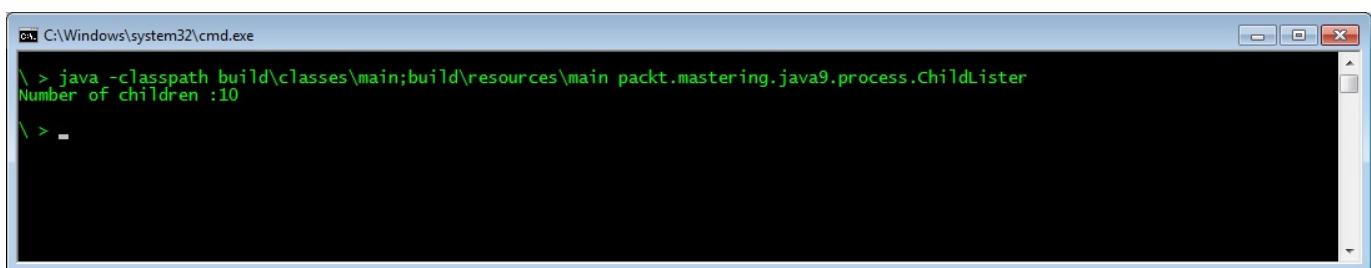
The following program starts 10 command prompts in Windows and then counts the number of children processes and prints it to standard output:

```
package packt.mastering.java9.process;

import java.io.IOException;

public class ChildLister {
    public static void main(String[] args) throws IOException {
        for (int i = 0; i < 10; i++) {
            new ProcessBuilder().command("cmd.exe").start();
        }
        System.out.println("Number of children :" +
            ProcessHandle.current().children().count());
    }
}
```

Executing the program will result in the following:



Listing descendants

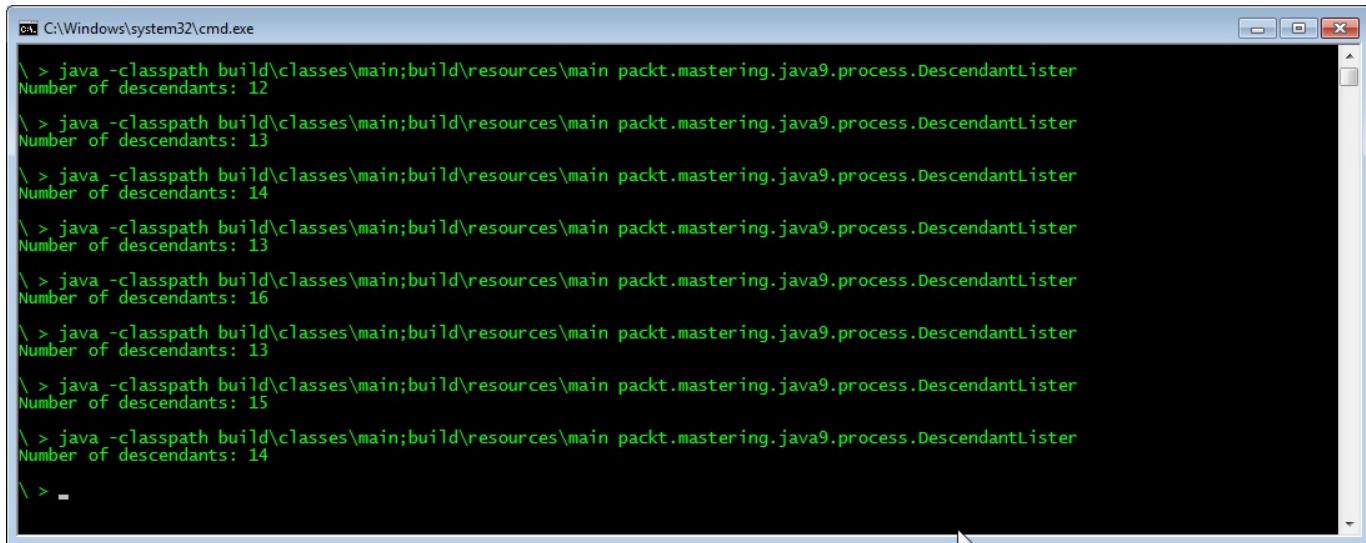
Listing the descendants is very similar to listing children, but if we call the `processHandle.descendants()` method then the `Stream` will contain all the children processes and the children processes of those processes and so on. The following program starts command prompts with command-line arguments so that they also spawn another `cmd.exe` that terminates:

```
package packt.mastering.java9.process;

import java.io.IOException;
import java.util.stream.Collectors;

public class DescendantLister {
    public static void main(String[] args) throws IOException {
        for (int i = 0; i < 10; i++) {
            new ProcessBuilder().command("cmd.exe", "/K", "cmd") .
                start();
        }
        System.out.println("Number of descendants: " +
            ProcessHandle.current().descendants().count());
    }
}
```

Running the command a few times will result in the following, non-deterministic output:



The screenshot shows a Windows command prompt window titled 'cmd.exe' with the path 'C:\Windows\system32\cmd.exe'. The window contains several lines of text output from running the Java program. Each line starts with a backslash and greater than symbol (\>) followed by the command and its output. The output shows the number of descendants for each run, which varies between 12 and 16. The window has a standard Windows title bar and a scroll bar on the right side.

```
\> java -classpath build\classes\main;build\resources\main packt.mastering.java9.process.DescendantLister
Number of descendants: 12
\> java -classpath build\classes\main;build\resources\main packt.mastering.java9.process.DescendantLister
Number of descendants: 13
\> java -classpath build\classes\main;build\resources\main packt.mastering.java9.process.DescendantLister
Number of descendants: 14
\> java -classpath build\classes\main;build\resources\main packt.mastering.java9.process.DescendantLister
Number of descendants: 13
\> java -classpath build\classes\main;build\resources\main packt.mastering.java9.process.DescendantLister
Number of descendants: 16
\> java -classpath build\classes\main;build\resources\main packt.mastering.java9.process.DescendantLister
Number of descendants: 13
\> java -classpath build\classes\main;build\resources\main packt.mastering.java9.process.DescendantLister
Number of descendants: 15
\> java -classpath build\classes\main;build\resources\main packt.mastering.java9.process.DescendantLister
Number of descendants: 14
\> -
```

The output clearly demonstrates that when the `Stream` of the descendants is created not all processes are alive. The sample code starts 10 processes and each of them starts another. The `Stream` does not have 20 elements because some of these sub-

processes were terminated during processing.

Listing all processes

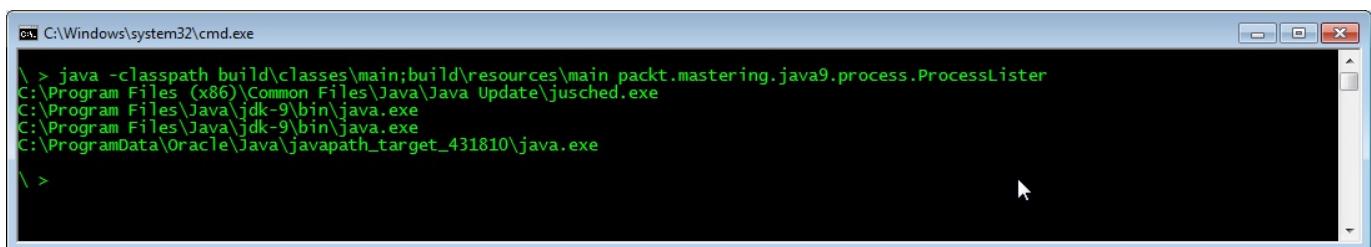
Listing all the processes is slightly different from listing descendants and children. The method `allProcesses()` is static and returns a `Stream` of handles of all processes that are active in the operating system at the time of execution.

The following sample code prints the process commands to the console that seem to be Java processes:

```
package packt.mastering.java9.process;
import java.lang.ProcessHandle.Info;
public class ProcessLister {
    private static void out(String format, Object... params) {
        System.out.println(String.format(format, params));
    }
    private static boolean looksLikeJavaProcess(Info info) {
        return info.command().isPresent() &&
            info.command().get().
                toLowerCase().indexOf("java") != -1;
    }

    public static void main(String[] args) {
        ProcessHandle.allProcesses().
            map(ProcessHandle::info).
            filter(info -> looksLikeJavaProcess(info)).
            forEach(
                (info) -> System.out.println(
                    info.command().orElse("---"))
            );
    }
}
```

The output of the program lists all the process commands that have the string `java` inside:



A screenshot of a Windows Command Prompt window titled 'cmd' with the path 'C:\Windows\system32\cmd.exe'. The window contains the following text:

```
C:\ > java -classpath build\classes\main;build\resources\main packt.mastering.java9.process.ProcessLister
C:\Program Files (x86)\Common Files\Java\Java Update\jusched.exe
C:\Program Files\Java\jdk-9\bin\java.exe
C:\Program Files\Java\jdk-9\bin\java.exe
C:\ProgramData\Oracle\Java\javapath_target_431810\java.exe
\ >
```

Your actual output may, of course, be different.

Waiting for processes

When a process starts another process, it may wait for the process many times because it needs the result of the other program. If the structure of the task can be organized in a way that the parent program can do something else while waiting for the child process to finish, then the parent process can invoke the `isAlive()` method on the process handle. Many times, the parent process has nothing to do until the spawned process finishes. Old applications implemented loops that called the `Thread.sleep()` method so CPU was not excessively wasted and from time to time the process was checked to see if it was still alive. Java 9 offers a much better approach to the waiting process.

The `ProcessHandle` interface has a method called `onExit` that returns a `CompletableFuture`. This class was introduced in Java 8 and makes it possible to wait for a task to be finished without looping. If we have the handle to a process we can simply call the `handle.onExit().join()` method to wait until the process finishes. The `get()` method of the returned `CompletableFuture` will return the `ProcessHandle` instance that was used to create it in the first place.

We can call the `onExit()` method on the handle many times and each time it will return a different `CompletableFuture` object, each related to the same process. We can call the `cancel()` method on the object but it will only cancel the `CompletableFuture` object and not the process and also does not have any effect on the other `CompletableFuture` objects that were created from the same `ProcessHandle` instance.

Terminating processes

To terminate a process we can call the `destroy()` method or the `destroyForcibly()` method on the `ProcessHandle` instance. Both of these methods will terminate the process. The `destroy()` method is expected to terminate the process gracefully executing the process shutdown sequence. In this case the shutdown hooks added to the run time are executed if the actual implementation supports the graceful, normal termination of processes. The `destroyForcibly()` method will enforce process termination, and in this case the shutdown sequence will not be executed.

If the process managed by the handle is not alive then nothing happens when the code calls any of these methods. If there are any `CompletableFuture` objects created calling the `onExit()` method on the handle then they will be completed after the call to the `destroy()` or `destroyForcefully()` method when the process has terminated. This means that the `CompletableFuture` object will return from a `join()` or some similar method after some time when the process termination is complete and not immediately after `destroy()` or `destroyForcefully()` returned.

It is also important to note that process termination may depend on many things. If the actual process that is waiting to terminate another does not have the right to terminate the other process then the request will fail. In this case the return value of the method is `false`. It is also important to understand that a return value of `true` does not mean that the process has actually terminated. It only means that the termination request was accepted by the operating system and that the operating system will terminate the process at some point in the future. This will actually happen rather soon, but not instantaneously and thus it should not be a surprise if the method `isAlive()` returns `true` for some time after the method `destroy()` or `destroyForcefully()` returned the value `true`.

The difference between `destroy()` and `destroyForcefully()` is implementation specific. The Java standard does not state that `destroy()` does terminate the process letting the shutdown sequence be executed. It only *requests the process be killed. Whether the process represented by this ProcessHandle object is normally terminated or not is implementation dependent* (<http://download.java.net/java/jdk9/docs/api/java/lang/ProcessHandle.html#supportsNormalTermination-->).



To learn more about `ProcessHandle` interface, visit <http://download.java.net/t/java/jdk9/docs/api/java/lang/ProcessHandle.html>.

This is because some operating systems do not implement the graceful process termination feature. In such situations, the implementation of `destroy()` is the same as calling `destroyForcefully()`. The system specific implementation of the interface `ProcessHandle` must implement the method `supportsNormalTermination()` that is `true` only if the implementation supports normal (not forceful) process termination. The method is expected to return the same value for all invocations in an actual implementation and should not change the return value during the execution of a JVM instance. There is no need to call the method multiple times.

The following examples demonstrate process starting, process termination, and waiting for the process to terminate. In our example, we use two classes. This first class demonstrates the `.sleep()` method:

```
package packt.mastering.java9.process;

public class WaitForChildToBeTerminated
{
    public static void main(String[] args)
        throws InterruptedException
    {
        Thread.sleep(10_000);
    }
}
```

The second class in our example calls the `WaitForChildToBeTerminated` class:

```
package packt.mastering.java9.process;

import java.io.IOException;
import java.util.Arrays;
import java.util.concurrent.CompletableFuture;
import java.util.stream.Collectors;

public class TerminateAProcessAfterWaiting {
    private static final int N = 10;

    public static void main(String[] args)
        throws IOException, InterruptedException {
        ProcessHandle ph[] = new ProcessHandle[N];

        for (int i = 0; i < N; i++) {
            final ProcessBuilder pb = new ProcessBuilder()
                .command("java", "-cp", "build/classes/main",
                    "packt.mastering.java9.process."
                    "WaitForChildToBeTerminated");
        }
    }
}
```

```

        Process p = pb.start();
        ph[i] = p.toHandle();
    }
    long start = System.currentTimeMillis();
    Arrays.stream(ph).forEach(ProcessHandle::destroyForcibly);

    CompletableFuture.allOf(Arrays.stream(ph) .
        map(ProcessHandle::onExit) .
        collect(Collectors.toList()) .
        toArray(new CompletableFuture[ph.length])) .
        join();
    long duration = System.currentTimeMillis() - start;
    System.out.println("Duration " + duration + "ms");
}
}

```

The preceding code starts 10 processes, each executing the program that sleeps ¹⁰ seconds. Then it forcibly destroys the processes, more specifically, the operating system is asked to destroy them. Our example joins the `CompletableFuture` that is composed from the array of `CompletableFuture` objects, which are created using the handles of the individual processes.

When all the processes are finished then it prints out the measured time in milliseconds. The time interval starts when the processes are created and the process creation loop finished. The end of the measured time interval is when the processes are recognized by the JVM returning from the `join()` method.

The sample code sets the sleeping time to 10 seconds. This is a more noticeable time period. Running the code twice and deleting the line that destroys the processes can result a much slower printout. Actually the measured and printed elapsed times will also show that terminating the processes has an effect.

A small process controller application

To summarize and put into use all that we have learned in this chapter we look at a sample process control application. The functionality of the application is very simple. It reads from a series of configuration file(s) parameters how to start some processes and then if any of them stops, it tries to restart the process.

Even a real life application can be created from this demo version. You can extend the set of parameters of the process with environment variable specifications. You can add default directory for the process, input and output redirection or even how much CPU a process is allowed to consume without the controlling application killing and restarting it.

The application consists of four classes.

- `Main`: This class that contains the public static void main method and is used to start up the daemon.
- `Parameters`: This class contains the configuration parameters for a process. In this simple case it will only contain one field, the `commandLine`. If the application gets extended this class will contain the default directory, the redirections, and CPU use limiting data.
- `ParamsAndHandle`: This class that is nothing else than a data tuple holding a reference to a `Parameters` object and also a process handle. When a process dies and gets restarted the process handle is replaced by the new handle, but the reference to the `Parameters` object never changes, it is configuration.
- `ControlDaemon`: This class implements the interface `Runnable` and is started as a separate thread.

In the code we will use most of the process API we discussed in the previous sections, *Terminating processes* and we will use a lot of threading code and stream operations. Understanding the threading work of the JVM is important independent of the process management also. It has, however emphasized importance when used together with the process API.

Main class

The main method takes the name of the directory from the command line argument. It treats this as relative to the current working directory. It uses a separate method from the same class to read the set of configurations from the files in the directory and then starts the control daemon. The following code if the `main` method of the program:

```
public static void main(String[] args) throws IOException,
    InterruptedException
{
    // DemoOutput.out() simulated - implementation no shown
    DemoOutput.out(new File(".").getAbsolutePath().toString());
    if (args.length == 0) {
        System.err.println("Usage: daemon directory");
        System.exit(-1);
    }
    Set<Parameters> params = parametersSetFrom(args[0]);
    Thread t = new Thread(new ControlDaemon(params));
    t.start();
}
```

Although this is a daemon, we are starting it as a normal thread and not as a daemon thread. When a thread is set to be a daemon thread it will not keep the JVM alive. When all other non-daemon threads stop, the JVM will just exit and the daemon threads will be stopped. In our case, the daemon thread we execute is the only one that keeps the code running. After that was started the main thread has nothing more to do, but the JVM should stay alive, until it is killed by the operator issuing a Unix `kill` command or pressing *Control + C* on the command line.

Getting the list of the files that are in the directory specified and getting the parameters from the file is simple using the new `Files` and `Paths` classes from the JDK:

```
private static Set<Parameters>
    GetListOfFilesInDirectory(String directory) throws IOException
{
    return Files.walk(Paths.get(directory))
        .map(Path::toFile)
        .filter(File::isFile)
        .map(file -> Parameters.fromFile(file))
        .collect(Collectors.toSet());
}
```

We get a stream of the files in the form of `Path` objects, map it to `File` objects, then

we filter out the directories if there are any in the configuration directory and map the remaining plain files to `Parameters` objects using the static method `fromFile` of the `Parameters` class. Finally, we return a `Set` of the objects.

Parameters class

Our `Parameters` class has a field and a constructor as listed as follows:

```
final String[] commandLine;

public Parameters(String[] commandLine) {
    this.commandLine = commandLine;
}
```

The parameter class has two methods. The first method, `getCommandLineStrings`, gets the command line strings from the properties. This array contains the command and the command line parameters. If it was not defined in the file then we return an empty array:

```
private static String[] getCommandLineStrings(Properties props)
{
    return Optional
        .ofNullable(props.getProperty("commandLine"))
        .orElse("")
        .split("\\s+");
}
```

The second method is the `static fromFile` that reads the properties from a properties file:

```
public static Parameters fromFile(final File file)
{
    final Properties props = new Properties();
    try (final InputStream is = new FileInputStream(file)) {
        props.load(is);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    return new Parameters(getCommandLineStrings(props));
}
```

If the set of parameters handled by the program is extended then this class should also be modified.

The ParamsAndHandle class

The `ParamsAndHandle` is a very simple class that holds two fields. One for the parameters and the other is the handle to the process handle that is used to access the process started using the parameters:

```
public class ParamsAndHandle
{
    final Parameters params;
    ProcessHandle handle;

    public ParamsAndHandle(Parameters params,
        ProcessHandle handle) {
        this.params = params;
        this.handle = handle;
    }

    public ProcessHandle toHandle() {
        return handle;
    }
}
```

Since the class is closely tied to the `ControlDaemon` class from where it is used there is no mutator or accessor associated with the field. We see the two classes as something inside the same encapsulation boundaries. The `toHandle` method is there so that we can use it as a method handle, as we will see in the next chapter.

The ControlDaemon class

The `ControlDaemon` class implements the `Runnable` interface and is started as a separate thread. The constructor gets the set of the parameters that were read from the properties files and converts them to a set of `ParamsAndHandle` objects:

```
private final Set<ParamsAndHandle> handlers;

public ControlDaemon(Set<Parameters> params) {
    handlers = params
        .stream()
        .map( s -> new ParamsAndHandle(s,null))
        .collect(Collectors.toSet());
}
```

Because the processes are not started at this point, the handles are all `null`. The `run()` method starts the processes:

```
@Override
public void run() {
    try {
        for (ParamsAndHandle pah : handlers) {
            log.log(DEBUG, "Starting {}", pah.params);
            ProcessHandle handle = start(pah.params);
            pah.handle = handle;
        }
        keepProcessesAlive();
        while (handlers.size() > 0) {
            allMyProcesses().join();
        }
    } catch (IOException e)
    {
        log.log(ERROR, e);
    }
}
```

Processing goes through the set of parameters and uses the method (implemented in this class later) to start the processes. The handles to each process get to the `ParamsAndHandle` object. After that, the `keepProcessesAlive` method is called and waits for the processes to finish. When a process stops it gets restarted. If it cannot be restarted it will be removed from the set.

The `allMyProcesses` method (also implemented in this class) returns a `CompletableFuture` that gets completed when all the started processes have stopped. Some of the processes may have been restarted by the time the `join()` method returns. As long as there is at least one process running, the thread should run.

Using the `CompletableFuture` to wait for the processes and the `while` loop, we use minimal CPU to keep the thread alive so long as long there is at least one process we manage running, presumably even after a few restarts. We have to keep this thread alive even if it does not use CPU and executes no code most of the time to let the `keepProcessesAlive()` method do its work using `CompletableFutures`. The method is shown in the following code snippet:

```
private void keepProcessesAlive()
{
    anyOfMyProcesses()
        .thenAccept(ignore -> {
            restartProcesses();
            keepProcessesAlive();
        });
}
```

The `keepProcessesAlive()` method calls the `anyOfMyProcesses()` method that returns a `CompletableFuture`, which is completed when any of the managed processes exits. The method schedules to execute the lambda passed as an argument to the `thenAccept()` method for the time the `CompletableFuture` is completed. The lambda does two things:

- Restarts the processes that are stopped (probably only one)
- Calls the `keepProcessesAlive()` method

It is important to understand that this call is not performed from within the `keepProcessesAlive()` method itself. This is not a recursive call. This is scheduled as a `CompletableFuture` action. We are not implementing a loop in a recursive call, because we would run out of stack space. We ask the JVM executors to execute this method again when the processes are restarted.

It is important to know that the JVM uses the default `ForkJoinPool` to schedule these tasks and this pool contains daemon threads. That is the reason we have to wait and keep the method running because that is the only non-daemon thread that prevents the JVM from exiting.

The next method is `restartProcesses()`:

```
private void restartProcesses()
{
    Set<ParamsAndHandle> failing = new HashSet<>();
    handlers.stream()
        .filter(pah -> !pah.toHandle().isAlive())
        .forEach(pah -> {
            try {
                pah.handle = start(pah.params);
            } catch (IOException e) {

```

```

        failing.add(pah);
    }
});
handlers.removeAll(failing);
}
}

```

This method starts the processes that are in our set of managed processes and which are not alive. If any of the restarts fail it removes the failing processes from the set. (Be aware not to remove it in the loop to avoid `ConcurrentModificationException`.)

The `anyOfMyProcesses()` and `allMyProcesses()` methods are using the auxiliary `completableFuturesOfTheProcessesand()` method and are straight forward:

```

private CompletableFuture anyOfMyProcesses()
{
    return CompletableFuture.anyOf(
        completableFuturesOfTheProcesses());
}

private CompletableFuture allMyProcesses() {
    return CompletableFuture.allOf(
        completableFuturesOfTheProcesses());
}

```

The `completableFuturesOfTheProcesses()` method returns an array of `CompletableFuture`s created from the currently running managed processes calling their `onExit()` method. This is done in a compact and easy to read functional programming style, as shown here:

```

private CompletableFuture[] completableFuturesOfTheProcesses()
{
    return handlers.stream()
        .map(ParamsAndHandle::toHandle)
        .map(ProcessHandle::onExit)
        .collect(Collectors.toList())
        .toArray(new CompletableFuture[handlers.size()]);
}

```

The set is converted to a `stream`, mapped to a `stream` of `ProcessHandle` objects (this is why we needed the `toHandle()` method in the `ParamsAndHandle` class). Then the handles are mapped to `CompletableFuture` stream using the `onExit()` method and finally we collect it to a list and convert to an array.

Our last method to complete our sample application is as follows:

```

private ProcessHandle start(Parameters params)
throws IOException {
    return new ProcessBuilder(params.commandLine)
        .start()
        .toHandle();
}

```

```
|     }
```

This method starts the process using a `ProcessBuilder` and returns the `ProcessHandle` so that we can replace the old one in our set and manage the new process.

Summary

In this chapter we discussed how Java 9 better enables us to manage processes. Prior to Java 9, process management from within Java required OS specific implementations and was less than optimal in terms of CPU use and coding practice. The modern API, with new classes like `ProcessHandle`, makes it possible to handle almost all aspects of processes. We listed the new API, and had simple example codes for the use of each of them. In the second half of the chapter we put together a whole application managing processes where the learned API was put into practice.

In the next chapter, we will take a detailed look at the new Java Stack Walking API released with Java 9. We will use code samples to illustrate how to use the API.

Fine-Grained Stack Tracing

Java 9 comes with a new stack walker API that lets the program walk the calling stack. This is a very special functionality that is rarely needed by ordinary programs. The API can be useful for some very special cases--for functionality that is delivered by framework. So, if you want an efficient means of stack walking that gives you filterable access to stack trace information, you will enjoy this new stack walker API.

The API gives fast and optimized access to the call stack, implementing lazy access to the individual frames.

In this chapter, we cover the following topics:

- Overview of the Java Stack
- The importance of stack information
- Using `StackWalker`
- The `StackFrame`
- Performance

Overview of the Java Stack

Before we dive into the stack walker, let's start by covering the Java Stack. This is basic stack information, not specific to the stack walker.

The Java runtime has a class named `Stack`, which can be used to store objects using the **last-in-first-out (LIFO)** policy.

When arithmetic expressions are calculated they are done using a stack. If we add A and B in our code first A is pushed on the **Operand Stack**, then B is pushed on the Operand Stack and finally the addition operation is executed, which fetches the two topmost elements of the Operand Stack and pushes the result, $A + B$ there.

The JVM is written in C and executes calling C functions and returning from there. This call-return sequence is maintained using the Native Method Stack just like any other C program.

Finally, when the JVM creates a new thread it also allocates a call stack containing frames that in turn contain the local variables, reference to the previous frame, and reference to the class that contains the executing method. When a method is invoked a new frame is created. The frame is destroyed when a method finishes its execution, in other words, returns or throws an exception. This stack, the Java Virtual Machine Stack, is the one that the stack walker API manages.

The importance of stack information

Generally speaking we need the stack information when we want to develop caller dependent code. Having information about the caller allows our code to make decisions based on that information. In general practice, it is not a good idea to make functionality dependent on the caller. Information that affects the behavior of a method should be available via parameters. Caller dependent code development should be fairly limited.

The JDK accesses stack information with native methods that are not available to Java applications. The `SecurityManager` is a class that defines an application's security policy. This class checks that the caller of a reflection API is allowed to access the non-public members of another class. To do that it has to have access to the caller class and it does that through a protected native method.

This is an example of implementing some security measures without having to walk through a stack. We open our code for external developers to use it as a library. We also call methods of classes provided by the library user and they may call back to our code. There is some code that we want to allow library users to call but only if they were not called from our code. If we did not want to allow some of the code to be accessed directly by the library using code we could use Java 9's modular structure not exporting the package containing the classes not to be invoked. This is the reason we set the extra condition that the code is available for the callers from outside, except if they were called by our code:



Another example is when we want to get access to a logger. Java applications use many different loggers and the logging system is usually very flexible so that the output of the different loggers can be switched on and off based on the actual need to introspect into the code. The most common practice is to use a different logger for each class and the name of the logger is usually the name of the class. The practice is so common that the logging framework even provides logger access methods that accept the reference to the class itself instead of the name. It essentially means that the call to get the handle of a logger looks something like the following:

```
| private static final Logger LOG = Logger.getLogger(MyClass.class);
```

A problem can arise when we create new classes from existing classes if we forget to alter the name of the class name in the call for getting a new logger. This is not a serious problem, but it is common. In that case our code will use the logger of the other class and it will actually work, but may create confusion when we analyze the log files. It would be much nicer if we had a method that returns the logger that is named as the class of the caller.

Let's continue our exploration of stack information in the next two sections with code snippet examples.

Example - Restricting callers

In this section we develop a sample library with two methods. The `hello()` method prints `hello` to the standard output. The `callMe()` method accepts a `Runnable` as an argument and runs it. The first method however is restricted. It executes only if the caller is purely outside of the library. It throws an `IllegalCallerException` if the caller obtained the control in a way that the library was calling out, presumably via the second method invoking the passed `Runnable`. The implementation of the API is simple:

```
package packt.java9.deep.stackwalker.myrestrictivelibrary;
public class RestrictedAPI {
    public void hello(){
        CheckEligibility.itIsNotCallBack();
        System.out.println("hello");
    }
    public void callMe(Runnable cb){
        cb.run();
    }
}
```

The code that performs the eligibility checking is implemented in a separate class to keep things simple. We will examine that code shortly, but before that we look at the main code we use to start the demonstration. The main program code we use to demonstrate the functionality is the following:

```
package packt.java9.deep.stackwalker.externalcode;

import
    packt.java9.deep.stackwalker.myrestrictivelibrary.RestrictedAPI;

public class DirectCall {

    public static void main(String[] args) {
        RestrictedAPI api = new RestrictedAPI();
        api.hello();
        api.callMe(() -> {
            api.hello();
        });
    }
}
```

This code creates an instance of our API class and then directly invokes the `hello()` method. It should work and should print the characters `hello` on screen. The next code line asks the `callMe()` method to call back the `Runnable` provided in form of a lambda expression. In this case the call will fail, because the caller is outside but was

called from inside the library.

Let's now look at how the eligibility check is implemented:

```
package packt.java9.deep.stackwalker.myrestrictivelibrary;

import static java.lang.StackWalker.Option.RETAIN_CLASS_REFERENCE;

public class CheckEligibility {
    private static final String packageName
        = CheckEligibility.class.getPackageName();

    private static boolean notInLibrary(StackWalker.StackFrame f) {
        return !inLibrary(f);
    }

    private static boolean inLibrary(StackWalker.StackFrame f) {
        return f.getDeclaringClass().getPackageName()
            .equals(packageName);
    }

    public static void itIsNotCallBack() {
        boolean eligible = StackWalker
            .getInstance(RETAIN_CLASS_REFERENCE)
            .walk(s -> s.dropWhile(CheckEligibility::inLibrary)
                .dropWhile(CheckEligibility::notInLibrary)
                .count() == 0
            );
        if (!eligible) {
            throw new IllegalCallerException();
        }
    }
}
```

The `itIsNotCallBack()` method is the one called from the `hello()` method. This method creates a stack walker and invokes the `walk()` method. The argument of the `walk()` method is a `Function` that converts a `Stream` of `StackFrame` objects to some other value that the `walk()` method will return.

At first this argument setting might seem complex and difficult to understand. It would be more logical to return a `Stream` that provides the `StackFrame` objects instead of forcing the caller to define a `Function` that will get this as an argument.

The sample code uses a lambda expression to define the `Function` as an argument to the `walk()` method. The argument to the lambda expression `s` is the stream. Since the first element of this stream is the actual call we drop it. Because these calls should also be refused if the caller is not eligible even though the call to method `hello()` was through some other class and method that is already inside the library, we drop all elements from the frame that belong to classes inside the package of the class

`CheckEligibility`. This package is `packt.java9.deep.stackwalker.myrestrictivelibrary` and

in the code this string is stored in the field `packageName`. The resulting stream contains only the `StackFrame` objects that are from outside of the library. We drop these also until the stream exhausts or until we find a `StackFrame` that again belongs to the library. If all elements were dropped we are good. In this case the result of `count()` is zero. If we find some class in the `StackFrame` that belongs to the library it means that the outside code was called from the library and in this case we have to refuse working. In this case the variable `eligible` will be `false` and we throw an exception, as can be seen in the following screenshot:

The screenshot shows the IntelliJ IDEA interface with two tabs open: `DirectCall.java` and `CheckEligibility.java`. The `DirectCall.java` tab is active, displaying the following code:

```

1 package packt.java9.deep.stackwalker.externalcode;
2
3 import packt.java9.deep.stackwalker.myrestrictivelibrary.RestrictedAPI;
4
5 public class DirectCall {
6
7     public static void main(String[] args) {
8         RestrictedAPI api = new RestrictedAPI();
9         api.hello();
10        api.callMe(() -> {
11            api.hello();
12        });
13    }
14}

```

The line `api.hello();` at position 11 is highlighted with a yellow background. The `Run` tool window below shows the output of the run:

```

Exception in thread "main" java.lang.IllegalCallerException
    at stackwalker/packt.java9.deep.stackwalker.myrestrictivelibrary.CheckEligibility.itIsNotCallBack(CheckEligibility.java:31)
    at stackwalker/packt.java9.deep.stackwalker.myrestrictivelibrary.RestrictedAPI.hello(RestrictedAPI.java:6)
    at stackwalker/packt.java9.deep.stackwalker.externalcode.DirectCall.lambda$main$0(DirectCall.java:11)
    at stackwalker/packt.java9.deep.stackwalker.myrestrictivelibrary.RestrictedAPI.callMe(RestrictedAPI.java:11)
    at stackwalker/packt.java9.deep.stackwalker.externalcode.DirectCall.main(DirectCall.java:10)

Process finished with exit code 1

```

Example - Getting logger for caller

To get a logger, Java 9 has a new API. Using this API a module can provide an implementation for the service `LoggerFinder`, which in turn can return a `Logger` implementing the `getLogger()` method. This eliminates the dependency of libraries on specific loggers or logger facades, which is a huge advantage. The smaller but still annoying issue requiring us to write the name of the class again as the parameter to the method `getLogger()` is still there.

To avoid this cumbersome task, we create a helper class that looks up the caller class and retrieves the logger that is suitable for the caller class and module. Because in this case there is no need for all the classes referenced in the stack trace we will call the `getCallerClass()` method of the `StackWalker` class. We create a class named `Labrador` in the package `packt.java9.deep.stackwalker.logretriever`:

```
package packt.java9.deep.stackwalker.logretriever;

import java.lang.System.Logger;
import java.lang.System.LoggerFinder;

import static java.lang.StackWalker.Option.RETAIN_CLASS_REFERENCE;

public class Labrador {
    public static Logger retrieve() {
        final Class clazz = StackWalker
            .getInstance(RETAIN_CLASS_REFERENCE)
            .getCallerClass();
        return LoggerFinder.getLoggerFinder().getLogger(
            clazz.getCanonicalName(), clazz.getModule());
    }
}
```

Before Java 9 the solution for this issue was getting the `StackTrace` array from the `Thread` class and looking up the name of the caller class from there. Another approach was extending the `SecurityManager` that has a protected method `getClassContext()` that returns an array of all the classes on the stack. Both solutions walk through the stack and compose an array although we only need one element from the array. In case of logger retrieval it may not be a significant performance penalty since loggers are usually stored in `private static final` fields and thus are initialized once per class during class initialization. In other use cases the performance penalty may be significant.

After we have seen two examples we will look at the details of `StackWalker` inner working.

Working with StackWalker

In this section you will become more familiar with how to work with `StackWalker`. We will explore the following topics in this section:

- Getting an instance of `StackWalker`
- Stack walking options

Getting an instance of StackWalker

To perform the walking over the stack elements we need an instance of the stack walker. To do that, we invoke the `getInstance()` method. As shown here, there are four overloaded versions of this method:

- `static StackWalker getInstance()`
- `static StackWalker getInstance(StackWalker.Option option)`
- `static StackWalker getInstance(Set<StackWalker.Option> options)`
- `static StackWalker getInstance(Set<StackWalker.Option> options, int estimateDepth)`

The first version does not take any arguments and returns a `StackWalker` instance that will let us walk through normal stack frames. This is usually what we would be interested in. The other versions of the method accept a `StackWalker.Option` value or values. The enum `StackWalker.Option`, as the name suggests, is inside the class `StackWalker` and has three values:

- `RETAIN_CLASS_REFERENCE`
- `SHOW_REFLECT_FRAMES`
- `SHOW_HIDDEN_FRAMES`

These `enum` options have self-descriptive names and are explained in the next sections.

RETAIN_CLASS_REFERENCE

If we specify the first option `enum constant, RETAIN_CLASS_REFERENCE, as an argument to the getInstance()` method then the returned instance grants us access to the classes that the individual stack frames reference during the walking.

SHOW_REFLECT_FRAMES

The `SHOW_REFLECT_FRAMES` enum constant will generate a walker that includes the frames that source from some reflective calling.

SHOW_HIDDEN_FRAMES

Finally the `enum` constant option, `SHOW_HIDDEN_FRAMES` will include all the hidden frames, which contain reflective calls as well as call frames that are generated for lambda function calls.

Here is a simple demonstration of reflective and hidden frames:

```
package packt;
import static java.lang.StackWalker.Option.SHOW_HIDDEN_FRAMES;
import static java.lang.StackWalker.Option.SHOW_REFLECT_FRAMES;
public class Main {
```

The main method allowing us to execute this code directly calls the method `simpleCall()`:

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

The method `simpleCall()` simply calls on as the name suggests:

```
static void simpleCall() {
    reflectCall();
}
```

The next method in the chain is a bit more complex. Although this also only calls the next one, it does so using reflection:

```
static void reflectCall() {
    try {
        Main.class.getDeclaredMethod("lambdaCall",
            new Class[0])
            .invoke(null, new Object[0]);
    } catch (Exception e) {
        throw new RuntimeException();
    }
}
```

In this next example, we have a method that calls using a lambda:

```
static void lambdaCall() {
    Runnable r = () -> {
        walk();
    };
    r.run();
}
```

The last method before the actual walking is called `walk()`:

```
static void walk() {
    noOptions();
    System.out.println();
    reflect();
    System.out.println();
    hidden();
}
```

The preceding `walk()` method calls three methods, one after the other. These methods are very similar to each other and provided here:

```
static void noOptions() {
    StackWalker
        .getInstance()
        .forEach(System.out::println);
}

static void reflect() {
    StackWalker
        .getInstance(SHOW_REFLECT_FRAMES)
        .forEach(System.out::println);
}

static void hidden() {
    StackWalker
        // shows also reflect frames
        .getInstance(SHOW_HIDDEN_FRAMES)
        .forEach(System.out::println);
}
```

The preceding three methods print out the frames to the standard output. They use the `forEach()` method of the stack walker. Here is the output of the stack walking program:

```
stackwalker/packt.Main.noOptions(Main.java:45)
stackwalker/packt.Main.walk(Main.java:34)
stackwalker/packt.Main.lambda$lambdaCall$0(Main.java:28)
stackwalker/packt.Main.lambdaCall(Main.java:30)
stackwalker/packt.Main.reflectCall(Main.java:19)
stackwalker/packt.Main.simpleCall(Main.java:12)
stackwalker/packt.Main.main(Main.java:8)
```

This output only contains the frames that belong to calls that are in our code. The `main()` method calls `simpleCall()`, which calls `reflectCall()`, that in turn calls `lambdaCall()`, which calls a lambda expression, that calls `walk()` and so on. The fact that we did not specify any option does not delete the lambda call from the stack. We performed that call, thus it must be there. What it deletes are the extra stack frames that are needed by the JVM to implement the lambda. We can see on the next output, when the option was `SHOW_REFLECT_FRAMES`, that the reflective frames are already there:

```
stackwalker/packt.Main.reflect(Main.java:58)
stackwalker/packt.Main.walk(Main.java:36)
stackwalker/packt.Main.lambda$lambdaCall$0(Main.java:28)
stackwalker/packt.Main.lambdaCall(Main.java:30)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorI
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodA
java.base/java.lang.reflect.Method.invoke(Method.java:547)
stackwalker/packt.Main.reflectCall(Main.java:19)
stackwalker/packt.Main.simpleCall(Main.java:12)
stackwalker/packt.Main.main(Main.java:8)
```

In this case the difference is that we can see that the call from the `reflectCall()` method to `lambdaCall()` method is not direct. The `reflectCall()` method calls the `invoke()` method that calls another method of the same name defined in a different class that in turn calls the `invoke0()` method, which is a native method provided by the JVM. After that we finally get to the `lambdaCall()` method.

In the output we can also see that these reflective calls belong to the module `java.base` and not our `stackwalker` module.

If we include the hidden frames in addition to the reflective frames, specifying the option `SHOW_HIDDEN_FRAMES`, then we will see the following output:

```
stackwalker/packt.Main.hidden(Main.java:52)
stackwalker/packt.Main.walk(Main.java:38)
stackwalker/packt.Main.lambda$lambdaCall$0(Main.java:28)
stackwalker/packt.Main$$Lambda$46/269468037.run(Unknown Source)
stackwalker/packt.Main.lambdaCall(Main.java:30)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorI
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodA
java.base/java.lang.reflect.Method.invoke(Method.java:547)
stackwalker/packt.Main.reflectCall(Main.java:19)
stackwalker/packt.Main.simpleCall(Main.java:12)
stackwalker/packt.Main.main(Main.java:8)
```

This includes an extra hidden frame that the JVM is using to execute the lambda call. In addition, the reflective frames are also included.

Final thoughts on enum constants

We can also specify more than one option giving a set of the options. The simplest way of doing that is to use the static `of()` method of the `java.util.Set` interface. This way the `RETAIN_CLASS_REFERENCE` option can be combined with either the `SHOW_REFLECT_FRAMES` option or the `SHOW_HIDDEN_FRAMES` option.

Although it is technically possible to combine `SHOW_REFLECT_FRAMES` and `SHOW_HIDDEN_FRAMES` as an option set, there is really no advantage in doing that. The latter includes the first, so the combination of the two is exactly same as the second.

Accessing classes

When we want to access the class objects during a stack walk, we have to specify the `RETAIN_CLASS_REFERENCE` option. Although the `StackFrame` interface defines the `getClassName()` method, that could be used to access a class of the name using the `Class.forName()` method, doing so would not guarantee that the class the `StackFrame` object refers to was loaded by the same class loader as the code calling `Class.forName()`. In some special cases, we could end up with two different classes of the same name loaded by two different class loaders.

When the option is not used during the creation of the `StackWalker` instance then the methods that otherwise return a class object will throw an `UnsupportedOperationException` exception. That way `getDeclaringClass()` cannot be used on the `StackFrame` and `getCallerClass()` on the `StackWalker`.

Walking methods

The `StackWalker` defines the `forEach()` method that expects a `Consumer` (preferably in the form of a lambda expression) that is invoked for each element of the stack trace walking up the stack. The argument to the `Consumer` method is a `StackFrame` object.

Although a method named `forEach` is also defined by the `Stream` interface and the method `walk()` passes a `Stream` object to the `Function` it gets as argument, we should not confuse the two. The `forEach()` method of `StackWalker` is a simpler, and most of the time less effective way to get through all the elements of the stack trace.

It is less effective, in most cases, because it forces the `StackWalker` instance to get all the elements of the stack trace so that the `forEach()` method can traverse through each element to the end. If we know that we will not traverse through the stack trace to the end we should use the `walk()` method that is accessing the stack the lazy way and thus leave more room for performance optimization.

The `StackWalker` class has the `walk()` method, which is the defining method that makes it a walker. The method accepts a `Function` that is called by the `StackWalker`. The return value of the `walk()` method will be the object returned by the `Function`. The argument to the `Function` is a `Stream<StackFrame>` that delivers the stack frames. The first frame is the one that contains the `walk()` method call, the next is the one that was calling method that contains the call to `walk()`, and so on.

The `Function` can be used to calculate some value based on the `StackFrame` objects that come from the stream and decide if a caller is eligible calling our code or not.

You might ponder, after reviewing the `walk()` method that needs a `Function` that in turn gets a `Stream<StackFrame>` as argument, why it is so complicated. We might wish we could get a `Stream<StackFrame>` from the `StackWalker` instance directly. The simplest approach would be to pass the stream back from the `Function`. Consider the following example:

```
// EXAMPLE OF WHAT NOT TO DO!!!!
public static void itIsNotCallBack() {
    Stream<StackWalker.StackFrame> stream =
        StackWalker
            .getInstance(RETAIN_CLASS_REFERENCE)
            .walk(s -> s);
```

```
boolean eligible = // YOU GET EXCEPTION!!!!
    stream.dropWhile(CheckEligibility::inLibrary)
        .dropWhile(CheckEligibility::notInLibrary)
        .count() == 0;
if (!eligible) {
    throw new IllegalCallerException();
}
```

What we were doing is simply returning the stream directly from the walker call and walking through it afterwards doing the same calculation. Our results are an `IllegalStateException` exception instead of the eligibility check.

The reason for that is that the implementation of the `StackWalker` is highly optimized. It does not copy the whole stack to provide source information for the stream. It works from the actual, living stack. To do that it has to be sure that the stack is not modified while the stream is in use. This is something very similar to the `ConcurrentModificationException` that we might get if we alter a collection while we iterate over it. If we passed the stream up in the call stack and then wanted to get the `StackFrame` out of it, the stream would try to get the information from the stack frame that is long gone, since we returned from the method that it belonged to. That way the `StackWalker` does not make a snapshot of the whole stack but rather it works from the actual one and it must ensure that the part of the stack it needs does not change. We may call methods from the `Function` and that way we can dig deeper in the call chain but we cannot get higher while the stream is in use.

Also do not try to play other tricks, like extending the `StackWalker` class. You cannot. It is a `final` class.

StackFrame

In previous sections, we iterated through the `StackFrame` elements, and provided sample code snippets, but did not take the time to examine it more closely. `StackFrame` is an interface defined inside the `StackWalker` class. It defines accessors, and a converter that can be used to convert the information to `StackTraceElement`.

The accessors the interface defines are the following:

- `getClassName()` will return the binary name of the class of the method represented by the `StackFrame`.
- `getMethodName()` will return the name of the method represented by the `StackFrame`.
- `getDeclaringClass()` will return the class of the method represented by the `StackFrame`. If the `Option.RETAIN_CLASS_REFERENCE` was not used during the creation of the `StackWalker` instance then the method will throw `UnsupportedOperationException`.
- `getByteCodeIndex()` gets the index to the code array containing the execution point of the method represented by the `StackFrame`. The use of this value can be helpful during bug hunting when looking at the disassembled Java code that the command line tool `javap` can give us. The programmatic use of this value can only be valuable for applications that have direct access to the byte code of the code, java agents or libraries that generate byte code during run-time. The method will return a negative number in case the method is native.
- `getFileName()` returns the name of the source file the method represented by the `StackFrame` was defined.
- `getLineNumber()` returns the line number of the source code.
- `isNativeMethod()` returns `true` if the method represented by the `StackFrame` is native and `false` otherwise.

The `StackFrame` does not provide any means to access the object that the method belongs to. You cannot access the arguments and the local variables of the method represented by the `StackFrame` and there is no other way you can accomplish that. This is important. Such access would be too invasive and is not possible.

Performance

Our coverage of `StackWalker` would not be complete without a look at performance considerations.

`StackWalker` is highly optimized and does not create huge memory structures that go unused. That is the reason why we have to use that `Function` passed to the method `walker()` as an argument. This is also the reason why a `StackTrace` is not automatically converted to a `StackTraceElement` when created. This only happens if we query the method name, the line number of the specific `StackTraceElement`. It is important to understand that this conversion takes a significant amount of time and if it was used for some debug purpose in the code it should not be left there.

To make the `StackWalker` even faster we can provide an estimate about the number of `StackFrame` elements that we will work with in the stream. If we do not provide such an estimate, the current implementation in the JDK will use eight `StackFrame` objects pre-allocated and when that is exhausted, the JDK will allocate more. The JDK will allocate the number of elements based on our estimate unless we estimate a value larger than 256. In that case, the JDK will use 256.

Summary

In this chapter, we learned how to use the `StackWalker` and provided example code. Our detailed review of the API included different usage scenarios, options, and information. We explained the API's complexity and shared how and how not to use the class. We closed with some related performance issues that the user has to be aware of.

In our next chapter, we will cover over a dozen Java Enhancement Proposals that were incorporated in the Java 9 platform. The featured changes will cover a wide range of tools and updates to APIs that are aimed at making developing with Java easier and the ability to create optimized Java applications. We will look at the new HTTP client, changes to the Javadoc and Doclet API, the new JavaScript parser, JAR and JRE changes, the new Java-level JVM compiler interface, support for TIFF images, platform logging, XML catalog support, collections, new platform-specific desktop features, and enhancements to method handling and the depreciation annotation.

New Tools and Tool Enhancements

In the previous chapter, we explored Java 9's new stack walker API and learned how it enables Java applications to walk the calling stack. This is a specialized functionality that is not often implemented in Java applications. That being said, the API may be good for some very special cases, such as for functionality that is delivered by a framework. You learned that if you develop framework-supporting application programming and you want code that depends on the caller context, then the stack walker API is for you. We also discovered that the API gives fast and optimized access to the call stack, implementing lazy access to the individual frames.

In this chapter, we will cover 16 **Java Enhancement Proposals (JEPs)** that were incorporated into the Java 9 platform. These JEPs cover a wide range of tools and updates to APIs to make developing with Java easier, with greater optimization possibilities for our resulting programs.

Our review of new tools and tool enhancements will include the following:

- The new HTTP client
- Javadoc and the Doclet API
- mJRE changes
- JavaScript parser
- Multi-release JAR files
- The Java-level JVM compiler interface
- TIFF support
- Platform logging
- XML Catalogs
- Collections
- Platform-specific desktop features
- Enhanced method handling
- Enhanced deprecation

The new HTTP client [JEP-110]

In this section, we will review Java's **Hypertext Transfer Protocol (HTTP)** client, starting with a pre-Java 9 look and then diving into the new HTTP client that is part of the Java 9 platform. This approach is needed to support an understanding of the changes made in Java 9.

The HTTP client before Java 9

JDK version 1.1 introduced the `HttpURLConnection` API that supported HTTP-specific features. This was a robust class that included the fields listed here:

- `chunkLength`
- `fixedContentLength`
- `fixedContentLengthLong`
- `HTTP_ACCEPTED`
- `HTTP_BAD_GATEWAY`
- `HTTP_BAD_METHOD`
- `HTTP_BAD_REQUEST`
- `HTTP_CLIENT_TIMEOUT`
- `HTTP_CONFLICT`
- `HTTP_CREATED`
- `HTTP_ENTITY_TOO_LARGE`
- `HTTP_FORBIDDEN`
- `HTTP_GONE`
- `HTTP_INTERNAL_ERROR`
- `HTTP_LENGTH_REQUIRED`
- `HTTP_MOVED_PERM`
- `HTTP_MOVED_TEMP`
- `HTTP_MULT_CHOICE`
- `HTTP_NO_CONTENT`
- `HTTP_NOT_ACCEPTABLE`
- `HTTP_NOT_AUTHORITATIVE`
- `HTTP_NOT_FOUND`
- `HTTP_NOT_IMPLEMENTED`
- `HTTP_NOT_MODIFIED`
- `HTTP_OK`
- `HTTP_PARTIAL`
- `HTTP_PAYMENT_REQUIRED`
- `HTTP_PRECON_FAILED`
- `HTTP_PROXY_AUTH`
- `HTTP_REQ_TOO_LONG`
- `HTTP_RESET`
- `HTTP_SEE_OTHER`

- HTTP_SERVER_ERROR
- HTTP_UNAUTHORIZED
- HTTP_UNAVAIABLE
- HTTP_UNSUPPORTED_TYPE
- HTTP_USE_PROXY
- HTTP_VERSION
- instanceFollowRedirects
- method
- responseCode
- responseMessage

As you can see from the list of fields, there was a great support for HTTP. In addition to a constructor, there are a plethora of available methods, including the following ones:

- disconnect()
- getErrorStream()
- getFollowRedirects()
- getHeaderField(int n)
- getHeaderFieldDate(String name, long Default)
- getHeaderFieldKey(int n)
- getInstanceFollowRedirects()
- getPermission()
- getRequestMethod()
- getResponseCode()
- getResponseMessage()
- setChunkedStreamingMode(int chunklen)
- setFixedLengthStreamingMode(int contentLength)
- setFixedlengthStreamingMode(long contentLength)
- setFollowRedirects(boolean set)
- setInstanceFollowRedirects(boolean followRedirects)
- setRequestMethod(String method)
- usingProxy()

The class methods listed earlier are in addition to the methods inherited from the `java.net.URLConnection` class and the `java.lang.Object` class.

There were problems with the original HTTP client that made it ripe for updating with the new Java platform. Those problems were as follows:

- The base `URLConnection` API had, defunct protocols such as Gopher and FTP increasingly over the years
- The `HttpURLConnection` API predated HTTP 1.1 and was overly abstract, making it less usable
- The HTTP client was woefully under documented, making the API frustrating and difficult to use
- The client only functioned on one thread at a time
- The API was extremely difficult to maintain due to the above points about it predating HTTP 1.1 and it lacking sufficient documentation

Now that we know what was wrong with the HTTP client, let's look at what's in store for Java 9.

Java 9's new HTTP client

There were several goals associated with creating the new HTTP client for the Java 9 platform. JEP-110 was the organizing proposal for the new HTTP client. The primary goals of JEP-110 are listed here and featured the creation of the new HTTP client presented. These goals are presented in the broad categories of ease of use, core capabilities, additional capabilities, and performance:

- Ease of use:
 - The API was designed to provide up to 90 percent of HTTP-related application requirements.
 - The new API is usable, without unnecessary complexity, for the most common use cases.
 - A simplistic blocking mode is included.
 - The API supports modern Java language features. Lambda expressions, a major new introduction released with Java 8, are an example.
- Core capabilities:
 - Supports HTTPS/TLS
 - Supports HTTP/2
 - Provides visibility on all details related to HTTP protocol requests and responses
 - Supports standard/common authentication mechanisms
 - Provides **headers received** event notifications
 - Provides **response body received** event notifications
 - Provides error event notifications
- Additional capabilities:
 - The new API can be used for WebSocket handshakes
 - It performs security checks in concert with the current networking API
- Performance:
 - For HTTP/1.1:
 - The new API must perform at least as efficiently as the previous API.
 - Memory consumption must not exceed that of Apache HttpClient, Netty, and Jetty, when being used as a client API.
 - For HTTP/2:
 - Performance must exceed that of HTTP/1.1.
 - The new performance must match or exceed that of Netty and Jetty when being used as a client API. No performance degradation should

be a result of the new client.

- Memory consumption must not exceed that of Apache HttpClient, Netty, and Jetty, when being used as a client API.
- Avoids running timer threads

New API limitations

There are some intentional shortcomings of the new API. While this might sound counter-intuitive, the new API was not intended to completely replace the current `HttpURLConnection` API. Instead, the new API is intended to eventually replace the current one.

The following code snippet provides an example of how to implement the `HttpURLConnection` class to open and read a URL in a Java application:

```
/*
import statements
*/

public class HttpURLConnectionExample
{
    public static void main(String[] args)
    {
        new HttpURLConnectionExample();
    }

    public HttpURLConnectionExample()
    {
        URL theUrl = null;
        BufferedReader theReader = null;
        StringBuilder theStringBuilder;

        // put the URL into a String
        String theUrl = "https://www.packtpub.com/";

        // here we are creating the connection
        theUrl = new URL(theUrl);
        HttpURLConnection theConnection = (HttpURLConnection)
            theUrl.openConnection();

        theConnection.setRequestMethod("GET");

        // add a delay
        theConnection.setReadTimeout(30000); // 30 seconds
        theConnection.connect();

        // next, we can read the output
        theReader = new BufferedReader(
            new InputStreamReader(theConnection.getInputStream()));
        theStringBuilder = new StringBuilder();

        // read the output one line at a time
        String theLine = null;
        while ((theLine = theReader.readLine()) != null)
        {
            theStringBuilder.append(line + "\n");
        }
    }
}
```

```
// echo the output to the screen console  
System.out.println(theStringBuilder.toString());  
  
// close the reader  
theReader.close();  
}  
}  
. . .
```

The preceding code does not include exception handling for brevity.



Here are some specific limitations of the new API:

- Not all HTTP-related functionality is supported. It is estimated that about 10 percent of the HTTP's protocol is not exposed by the API.
- Standard/common authentication mechanisms have been limited to basic authentication.
- The overarching goal of the new API was the simplicity of use, which means that performance improvements might not be realized. Certainly, there will be no performance degradation, but there is not likely to be an overwhelming level of improvement, either.
- There is no support for filtering on requests.
- There is no support for filtering on responses.
- The new API does not include a pluggable connection cache.
- There is a lack of a general upgrade mechanism.

The new API is delivered as part of the Java 9 platform in the incubator mode. This suggests that the API will be standardized in a future Java platform, perhaps with Java 10.

Simplified Doclet API [JEP-221]

The Doclet API and Javadoc are closely related. Javadoc is a documentation tool and the Doclet API provides functionality so that we can inspect the javadoc comments embedded at the source-levels of libraries and programs. In this section, we will review the pre-Java 9 status of the Doclet API and then explore the changes introduced to the Doclet API in the Java 9 platform. In the following section, we will review Javadoc.

The pre-Java 9 Doclet API

The pre-Java 9 Doclet API, or the `com.sun.javadoc` package, gives us access to look at javadoc comments located in the source code. Invoking a Doclet is accomplished by using the `start` method. That method's signature is `public static boolean start(RootDoc root)`. We will use the `RootDoc` instance as a container for the program structure information.

In order to call javadoc, we need to pass the following:

- Package names
- Source file names (for classes and interfaces)
- An access control option--one of the following:
 - `package`
 - `private`
 - `protected`
 - `public`

When the preceding listed items are used to call javadoc, a **documented set** is provided as a filtered list. If our aim is to obtain a comprehensive, unfiltered list, we can use `allClasses(false)`.

Let's review an example Doclet:

```
// Mandatory import statement.  
import com.sun.javadoc.*;  
  
// We will be looking for all the @throws documentation tags.  
public class AllThrowsTags extends Doclet  
{  
    // This is used to invoke the Doclet.  
    public static boolean start(Rootdoc myRoot)  
    {  
        // "ClassDoc[]" here refers to classes and interfaces.  
        ClassDoc[] classesAndInterfaces =  
            myRoot.classesAndInterfaces();  
        for (int i = 0; i < classesAndInterfaces.length; ++i)  
        {  
            ClassDoc tempCD = classesAndInterfaces[i];  
            printThrows(tempCD.contructors());  
            printThrows(tempCD.methods());  
        }  
        return true;  
    }  
}
```

```

static void printThrows(ExecutableMemberDoc[] theThrows)
{
    for (int i = 0; i < theThrows.length; ++i)
    {
        ThrowsTag[] throws = theThrows[i].throwsTags();

        // Print the "qualified name" which will be a the
        // class or
        // interface name.
        System.out.println(theThrows[i].qualifiedName());

        // A loop to print all comments with the Throws Tag that
        // belongs to the previously printed class or
        // interface name
        for (int j = 0; j < throws.length; ++j)
        {
            // A println statement that calls three methods
            // from the
            // ThrowsTag Interface: exceptionType(),
            // exceptionName(),
            // and exceptionComment().
            System.out.println("--> TYPE: " +
                throws[j].exceptionType() +
                " | NAME: " + throws[j].exceptionName() +
                " | COMMENT: " + throws[j].exceptionComment());
        }
    }
}

```

As you can see by the thoroughly commented code, gaining access to the javadoc content is relatively easy. In our preceding example, we would invoke the `AllThrows` class using the following code in the command line:

```
| javadoc -doclet AllThrowsTags -sourcepath <source-location> java.util
```

The output of our result will consist of the following structure:

```

<class or interface name>
    TYPE: <exception type> | NAME: <exception name> | COMMENT: <exception comment>
    TYPE: <exception type> | NAME: <exception name> | COMMENT: <exception comment>
    TYPE: <exception type> | NAME: <exception name> | COMMENT: <exception comment>
<class or interface name>
    TYPE: <exception type> | NAME: <exception name> | COMMENT: <exception comment>
    TYPE: <exception type> | NAME: <exception name> | COMMENT: <exception comment>

```

API enums

The API consists of one enum, `LanguageVersion`, which provides the Java programming language version. The constants for this enum are `Java_1_1` and `Java_1_5`.

API classes

The `Doclet` class provides an example of how to create a class to start a Doclet. It contains an empty `Doclet()` constructor and the following methods:

- `languageVersion()`
- `optionLength(String option)`
- `start(RootDoc root)`
- `validOptions(String[][] options, DocErrorReporter reporter)`

API interfaces

The Doclet API contains the following listed interfaces. The interface names are self-describing. You can consult the documentation for additional details:

- `AnnotatedType`
- `AnnotationDesc`
- `AnnotationDesc.ElementValuePair`
- `AnnotationTypeDoc`
- `AnnotationTypeElementDoc`
- `AnnotationValue`
- `ClassDoc`
- `ConstructorDoc`
- `Doc`
- `DocErrorReporter`
- `ExecutableMemberDoc`
- `FieldDoc`
- `MemberDoc`
- `MethodDoc`
- `PackageDoc`
- `Parameter`
- `ParameterizedType`
- `ParamTag`
- `ProgramElementDoc`
- `RootDoc`
- `SeeTag`
- `SerialFieldTag`
- `SourcePosition`
- `Tag`
- `ThrowsTag`
- `Type`
- `TypeVariable`
- `WildcardType`

Problems with the pre-existing Doclet API

Fueling the need for a new Doclet API were several issues with the pre-existing Doclet API:

- It was not ideal for testing or concurrent usage. This stemmed from its implementation of static methods.
- The language model used in the API had several limitations and became more problematic with each successive Java upgrade.
- The API was inefficient, largely due to its heavy use of substring matching.
- There was no reference provided regarding the specific location of any given comment. This made diagnostics and troubleshooting difficult.

Java 9's Doclet API

Now that you have a good handle on the Doclet API as it existed prior to Java 9, let's look at what changes have been made and delivered with the Java 9 platform. The new Doclet API is in the `jdk.javadoc.doclet` package.

At a high level, the changes to the Doclet API are as follows:

- Updates the `com.sun.javadoc` Doclet API to take advantage of several Java SE and JDK APIs
- Updates the `com.sun.tools.doclets.standard.Standard` Doclet to use the new API
- Supports the updated Taglet API that is used to create custom javadoc tags

In addition to the changes listed earlier, the new API uses the two APIs listed here:

- Compiler tree API
- Language model API

Let's explore each of these in the following sections.

Compiler tree API

The compiler tree API is in the `com.sun.source.doctree` package. It provides several interfaces to document source-level comments. These APIs are represented as **Abstract Syntax Trees (ASTs)**.

There are two enums:

- `AttributeTree.ValueKind` with the following constants:
 - DOUBLE
 - EMPTY
 - SINGLE
 - UNQUOTED
- `DocTree.Kind` with the following constants:
 - ATTRIBUTE
 - AUTHOR
 - CODE
 - COMMENT
 - DEPRECATED
 - DOC_COMMENT
 - DOC_ROOT
 - END_ELEMENT
 - ENTITY
 - ERRONEOUS
 - EXCEPTION
 - IDENTIFIER
 - INHERIT_DOC
 - LINK
 - LINK_PLAIN
 - LITERAL
 - OTHER
 - PARAM
 - REFERENCE
 - RETURN
 - SEE
 - SERIAL
 - SERIAL_DATA

- SERIAL_FIELD
- SINCE
- START_ELEMENT
- TEXT
- THROWS
- UNKNOWN_BLOCK_TAG
- UNKNOWN_INLINE_TAG
- VALUE
- VERSION

The `com.sun.source.doctree` package contains several interfaces. They are detailed in the following table:

Interface name	Extends	A tree node for:	Non-inherited methods
AttributeTree	DocTree	HTML element	<code>getName()</code> , <code>getValue()</code> , <code>getValueK</code>
AuthorTree	BlockTagTree, DocTree	@author block tag	<code>getName()</code>
BlockTagTree	DocTree	Base class for different types of block tags	<code>getTagName()</code>
CommentTree	DocTree	An embedded HTML comment with the following HTML tags-- <code><!--text--></code>	<code>getBody()</code>
DeprecatedTree	BlockTagTree	@deprecated block tag	<code>getBody()</code>
DocCommentTree	DocTree	Body block tags	<code>getBlockTags()</code> , <code>getBody()</code> , <code>getFirstSentence()</code>
DocRootTree	InlineTagTree	@docroot inline tag	N/A

DocTree	N/A	Common interface for all	accept(DocTreeVisitor<R,D>visit getKind())
DocTreeVisitor<R,P>	N/A	R = return type of visitor's methods; P = type of the additional parameter	visitAttribute(AttributeTree node, P p), visitAuthor(AuthorTree node, P p), visitComment(CommentTree node, P p), visitDeprecated(DeprecatedTree node, P p), visitDocComment(DocCommentTree node, P p), visitDocRoot(DocRootTree node, P p), visitEndElement(EndElementTree node, P p), visitEntity(EntityTree node, P p), visitErroneous(ErroneousTree node, P p), visitIdentifier(IdentifierTree node, P p), visitInheritDoc(InheritDocTree node, P p), visitLink(LinkTree node, P p), visitLiteral(LiteralTree node, P p), visitOther(DocTree node, P p), visitParam(ParamTree node, P p), visitReference(ReferenceTree node, P p), visitReturn(ReturnTree node, P p), visitSee(SeeTree node, P p), visitSerial(SerialTree node, P p), visitSerialData(SerialDataTree node, P p), visitSerialField(SerialFieldTree node, P p), visitSince(SinceTree node, P p), visitStartElement(StartElementTree node, P p), visitText(TextTree node, P p), visitThrows(ThrowsTree node, P p), visitUnknownBlockTag(UnknownBlockTag node, P p), visitUnknownInlineTag(UnknownInlineTag node, P p), visitValue(ValueTree node, P p), visitVersion(VersionTree node, P p)
EndElementTree	DocTree	End of an HTML element </name>	getName()
EntityTree	DocTree	An HTML entity	getName()
		This is for	

ErroneousTree	TextTree	malformed text	getDiagnostic()
IdentifierTree	DocTree	An identifier in a comment	getName()
InheritDocTree	InlineTagTree	@inheritDoc inline tag	N/A
InlineTagTree	DocTree	Common interface for inline tags	getTagName()
LinkTree	InlineTagTree	@link or @linkplan inline tags	getLabel(), getReference()
LiteralTree	InlineTagTree	@literal or @code inline tags	getBody()
ParamTree	BlockTagTree	@param block tags	getDescription(), getName(), isTypeParameter()
ReferenceTree	DocTree	Used to reference a Java lang element	getSignature()
ReturnTree	BlockTagTree	@return block tags	getDescription()
SeeTree	BlockTagTree	@see block tags	getReference()
SerialDataTree	BlockTagTree	@serialData block tags	getDescription()
SerialFieldTree	BlockTagTree	@serialData block tags and @serialField field names	getDescription(), getName(), get

		and descriptions	
SerialTree	BlockTagTree	@serial block tags	getDescription()
SinceTree	BlockTagTree	@since block tags	getBody()
StartElementTree	DocTree	Start of an HTML element < name [attributes] [/] >	getAttributes(), getName(), isSe
TextTree	DocTree	Plain text	getBody()
ThrowsTree	BlockTagTree	@exception or @throws block tags	getDescription(), getExceptionna
UnknownBlockTagTree	BlockTagTree	Unrecognized inline tags	getContent()
UnknownInlineTagTree	InlineTagTree	Unrecognized inline tags	getContent()
ValueTree	InlineTagTree	@value inline tags	getReference()
VersionTree	BlockTagTree	@version block tags	getBody()

Language model API

The language model API is in the `java.lang.model` package. It includes packages and classes that are used for language processing and language modeling. It consists of the following components:

- `AnnotatedConstruct` interface
- `SourceVersion` enum
- `UnknownEntityException` exception

Each of these language model API components is further explored in the next three sections.

The AnnotatedConstruct interface

The `AnnotatedConstruction` interface provides an annotatable construct to the language model API that has been part of the Java platform since version 1.8. It is applicable to constructs that are either an element (Interface `Element`) or a type (Interface `TypeMirror`). The annotations for each of these constructs differ, as shown in this table:

Construct type	Interface	Annotation
element	<code>Element</code>	Declaration
type	<code>TypeMirror</code>	Based on use of a type name

The `AnnotatedConstruction` interface has three methods:

- `getAnnotation(Class<A> annotationType)`: This method returns the type of the construct's annotation
- `getAnnotationMirrors()`: This method returns a list of annotations that are on the construct
- `getAnnotationsByType(Class<A> annotationType)`: This method returns the construct's associated annotations

The SourceVersion enum

The `SourceVersion` enum consists of the following constants:

- `RELEASE_0`
- `RELEASE_1`
- `RELEASE_2`
- `RELEASE_3`
- `RELEASE_4`
- `RELEASE_5`
- `RELEASE_6`
- `RELEASE_7`
- `RELEASE_8`



It is anticipated that the `SourceVersion` enum will be updated to include `RELEASE_9` once the Java 9 platform has been officially released.

This enum also contains several methods, which are as follows:

Method name: `isIdentifier`

```
public static boolean isIdentifier(CharSequence name)
```

This method returns `true` if the parameter string is a Java identifier or keyword.

Method name: `isKeyword`

```
public static boolean isKeyword(CharSequence s)
```

This method returns `true` if the given `CharSequence` is a literal or keyword.

Method name: `isName`

```
public static boolean isName(CharSequence name)
```

This method returns `true` if the `CharSequence` is a valid name.

Method name: `latest`

```
public static SourceVersion latest()
```

This method returns the latest source version for modeling purposes.

Method name: latestSupported

```
public static SourceVersion latestSupported()
```

This method returns the latest source version that can be fully supported for modeling.

Method name: valueOf

```
public static SourceVersion valueOf(String name)
```

This method returns the enum constant based on the parameter string provided.



You should be aware that the `valueOf(String name)` method throws two exceptions: `IllegalArgumentException` and `NullPointerException`.

Method name: values

```
public static SourceVersion[] values()
```

This method returns an array of the enum constants.

The UnknownEntityException exception

The `UnknownEntityException` class extends `RuntimeException` and is a superclass of unknown exceptions. The class constructor is as follows:

```
|     protected UnknownEntityException(String message)
```

The constructor creates a new instance of `UnknownEntityException` with the message provided as a string argument. The method does not take additional arguments.

This class does not have its own methods, but inherits methods from both `java.lang.Throwable` and `class.java.lang.Object` classes as shown here:

The `java.lang.Throwable` class methods:

- `addSuppressed()`
- `fillInStackTrace()`
- `getCause()`
- `getLocalizedMessage()`
- `getMessage()`
- `getStackTrace()`
- `getSuppressed()`
- `initCause()`
- `printStackTrace()`
- `setStackTrace()`
- `toString()`

The `java.lang.Object` class methods:

- `clone()`
- `equals()`
- `finalize()`
- `getClass()`
- `hashCode()`
- `notify()`
- `notifyAll()`

- `wait()`

HTML5 Javadoc [JEP-224]

The Javadoc tool has been updated for the Java 9 platform. It can now generate HTML 5 markup output in addition to HTML 4. The new Javadoc tool provides support for both HTML 4 and HTML 5.

HTML 4 will continue, even with the advent of the Java 9 platform, to be the default Javadoc output format. HTML 5 will be an option and will not become the default output markup format until Java 10.

The following short Java application simply generates a 319-wide by 319-high frame. It is shown here without any Javadoc tags, which we will discuss later in this section:

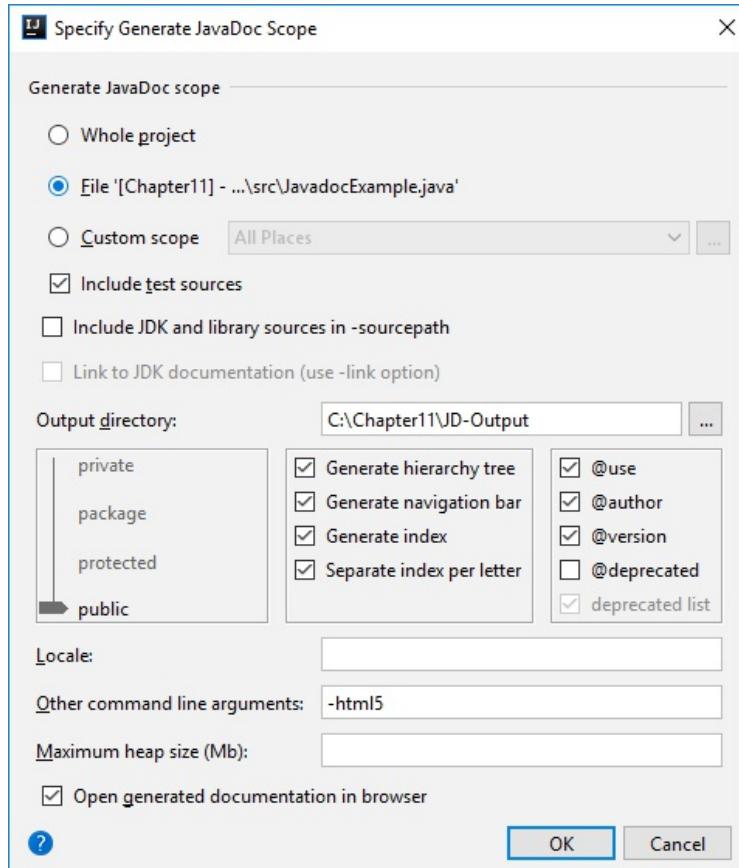
```
/import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class JavadocExample
{
    public static void main(String[] args)
    {
        drawJFrame();
    }

    public static void drawJFrame()
    {
        JFrame myFrame = new JFrame("Javadoc Example");
        myFrame.setSize(319,319);
        myFrame.setDefaultCloseOperation(
            WindowConstants.EXIT_ON_CLOSE);
        myFrame.setVisible(true);
    }
}
```

Once your package or class is completed, you can generate a Javadoc using the Javadoc tool. You can run the Javadoc tool, located in your JDK /bin directory, from the command line or from within your **Integrated Development Environment (IDE)**. Each IDE handles Javadoc generation differently. For example, in Eclipse, you would select Project from the pull-down menu and then Generate Javadoc. In the IntelliJ IDEA IDE, you select the Tools pull-down menu and then Generate Javadoc.

The following screenshot shows the IntelliJ IDEA interface for the Generate Javadoc functionality. As you can see, the `-html5` command-line argument has been included:



When the OK button is clicked, you will see a series of status messages, as shown in the following example:

```
"C:\Program Files\Java\jdk-9\bin\javadoc.exe" -public -splitindex -use -author -vers:  
Loading source file C:\Chapter11\src\JavadocExample.java...  
Constructing Javadoc information...  
Standard Doclet version 9  
Building tree for all the packages and classes...  
Generating C:\Chapter11\JD-Output\JavadocExample.html...  
Generating C:\Chapter11\JD-Output\package-frame.html...  
Generating C:\Chapter11\JD-Output\package-summary.html...  
Generating C:\Chapter11\JD-Output\package-tree.html...  
Generating C:\Chapter11\JD-Output\constant-values.html...  
Generating C:\Chapter11\JD-Output\class-use\JavadocExample.html...  
Generating C:\Chapter11\JD-Output\package-use.html...  
Building index for all the packages and classes...  
Generating C:\Chapter11\JD-Output\overview-tree.html...  
Generating C:\Chapter11\JD-Output\index-files\index-1.html...  
Generating C:\Chapter11\JD-Output\index-files\index-2.html...  
Generating C:\Chapter11\JD-Output\index-files\index-3.html...  
Building index for all classes...  
Generating C:\Chapter11\JD-Output\allclasses-frame.html...  
Generating C:\Chapter11\JD-Output\allclasses-frame.html...  
Generating C:\Chapter11\JD-Output\allclasses-noframe.html...  
Generating C:\Chapter11\JD-Output\allclasses-noframe.html...  
Generating C:\Chapter11\JD-Output\index.html...  
Generating C:\Chapter11\JD-Output\help-doc.html...  
  
javadoc exited with exit code 0
```

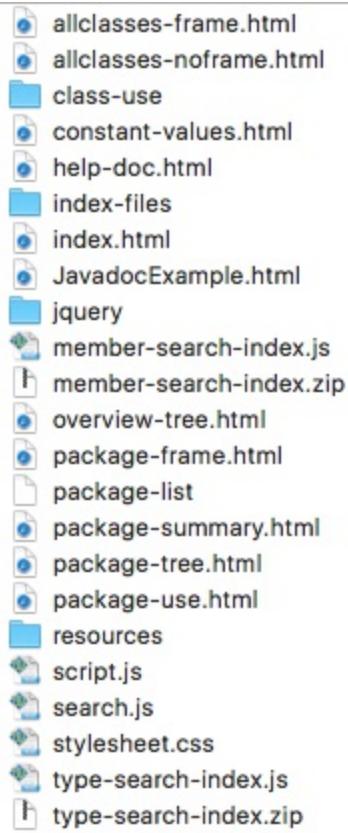
Once the Javadoc tool exits, you are ready to view the Javadoc. Here is a screenshot of what was generated based on the previously provided code. As you can see, it is formatted in the same manner in which the formal Java documentation from Oracle is documented:

The screenshot shows a Javadoc-generated HTML page for the `JavadocExample` class. The top navigation bar includes links for PACKAGE, CLASS (which is selected), USE, TREE, INDEX, and HELP. Below the navigation is a search bar labeled "SEARCH: Search". The main content area starts with the package declaration: `java.lang.Object` `JavadocExample`. A horizontal line separates this from the class definition: `public class JavadocExample extends java.lang.Object`. The page is organized into several sections:

- Constructor Summary:** Contains a table with columns for Constructors, Constructor, and Description. It lists the constructor `JavadocExample()`.
- Method Summary:** Contains tabs for All Methods, Static Methods, and Concrete Methods. It lists methods `drawJFrame()` and `main(String[] args)`. Below this, it lists methods inherited from `java.lang.Object`: `equals`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, and `wait`.
- Constructor Detail:** Shows the definition of the constructor `public JavadocExample()`.
- Method Detail:** Shows the definitions for the methods `drawJFrame()` and `main(String[] args)`.

At the bottom of the page is a footer with links for PACKAGE, CLASS (selected), USE, TREE, INDEX, and HELP, along with links for PREV CLASS, NEXT CLASS, FRAMES, NO FRAMES, ALL CLASSES, and a summary of nested, field, constr, and method details.

When we generated the Javadoc, multiple documents were created, as illustrated by the directory tree provided in the following screenshot:



You can also add optional tags that are recognized by the Javadoc tool. Those tags are provided here:

- @author
- @code
- @deprecated
- @docRoot
- @exception
- @inheritDoc
- @link
- @linkplain
- @param
- @return
- @see
- @serial
- @serialData
- @serialField
- @since
- @throws
- @value
- @version



For more information on how to write document comments for the Javadoc tool, you can visit Oracle's official instructions at <http://www.oracle.com/technetwork/articles/java/index-137868.html>.

Javadoc search [JEP-225]

Prior to Java 9, the standard Doclet generated API documentation pages that made navigating them difficult. Unless you are very familiar with the layout of these documentation pages, you will likely use browser-based find functionality to search text. This is considered clunky and suboptimal.

The Java 9 platform includes a search box as part of the API documentation. This search box is granted by the standard Doclet and can be used to search for text within the documentation. This represents a great convenience for developers and is likely to change our usage of Doclet-generated documentation.

With the new Javadoc search functionality, we have the ability to search for the following indexed components:

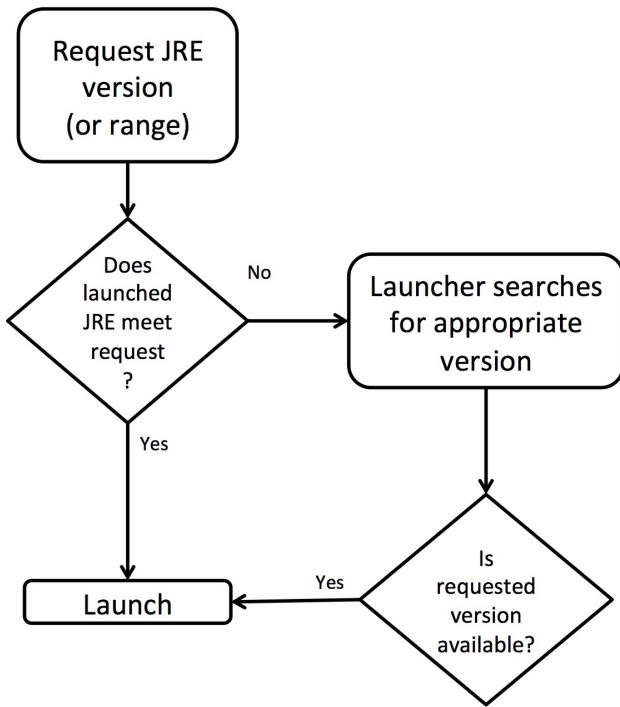
- Module names
- Package names
- Types
- Members
- Terms/phrases indexed using the new `@index` inline tag

Introducing camel case search

The new Javadoc search functionality includes a great shortcut using camel case search. As an example, we can search for `openED` to find the `openExternalDatabase()` method.

Remove launch-time JRE version selection [JEP-231]

Prior to Java 9, we could use the **mJRE** (**M**ultiple **J**RE) feature to specify a specific JRE version, or range of versions, for launching our applications. We would accomplish this via the command-line option `-version` or with an entry in the JAR file's manifest. The following flowchart illustrates what happens based on our selection:



This functionality was introduced with JDK 5 and was not fully documented in that release or any subsequent release prior to JDK 9.

The following specific changes were introduced with the Java 9 platform:

- The mJRE feature has been removed.
- The launcher will now produce an error whenever the `-version` command-line option is used. This is a terminal errors in that processing will not continue.
- A warning will be produced if there is a `-version` entry in a JARs manifest. The warning will not stop execution.

Interestingly, the presence of a `-version` entry in a manifest file will only generate a warning. This is by design, to take into account the likelihood of the entry being in older JAR file. It is estimated that this warning will be changed into a terminal error when the Java 10 platform is released.

Parser API for Nashorn [JEP-236]

The focus of JEP 236 was to create an API for Nashorn's EMCAbstract syntax tree. In this section, we will individually look at Nashorn, EMCAbstract and then the Parser API.

Nashorn

Oracle Nashorn is a JavaScript engine for the JVM developed in Java by Oracle. It was released with Java 8. It was created to provide developers with a highly efficient and lightweight JavaScript runtime engine. Using this engine, developers were able to embed JavaScript code in their Java applications. Prior to Java 8, developers had access to the JavaScript engine created by Netscape. That engine, introduced in 1997, was maintained by Mozilla.

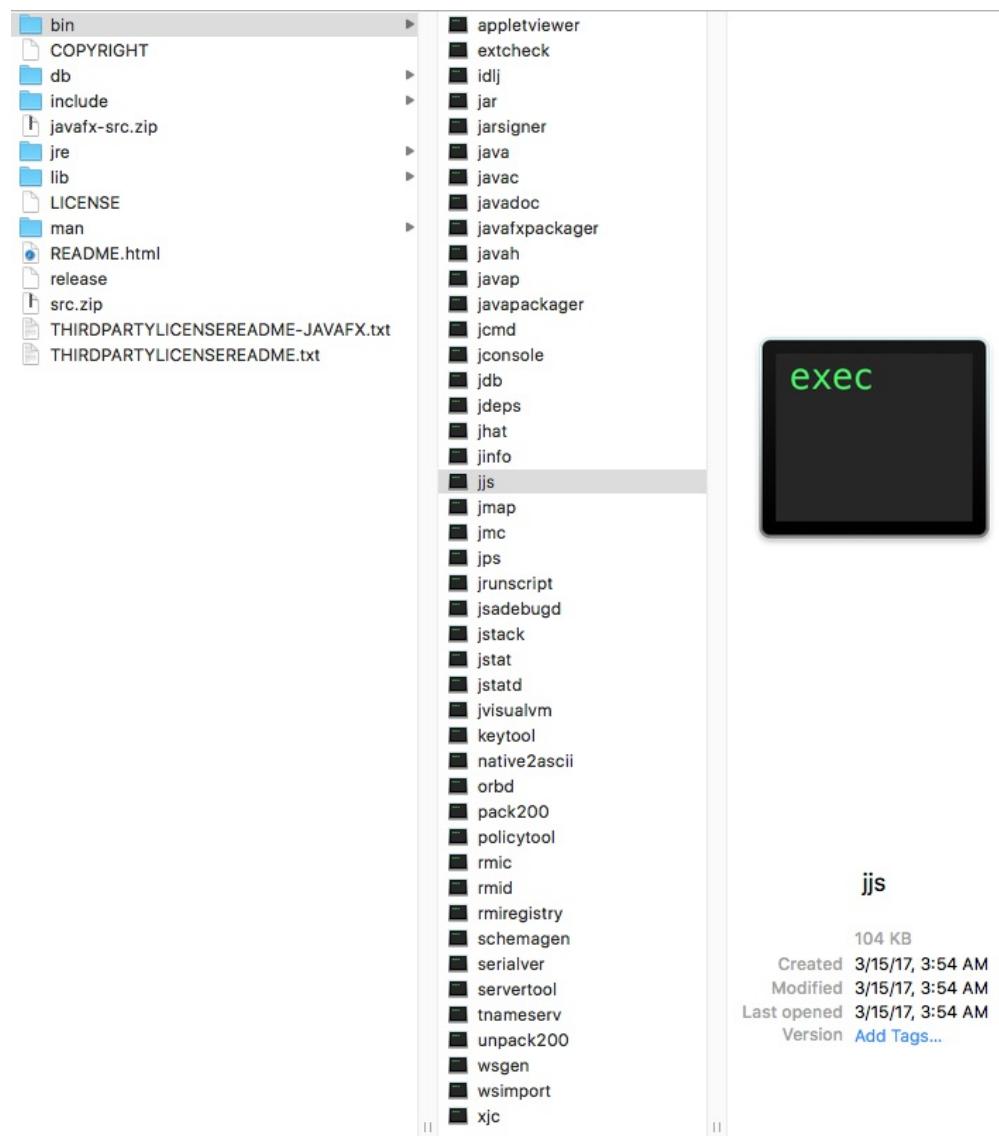
Nashorn can be used both as a command-line tool and as an embedded interpreter in Java applications. Let's look at examples of both.



*Nashorn is the German word for rhinoceros. The name spawned from the Rhino-named JavaScript engine from the Mozilla Foundation. Rhino is said to have originated from the picture of the animal on a JavaScript book cover. File this one under **interesting facts**.*

Using Nashorn as a command-line tool

The Nashorn executable file, `jjs.exe`, resides in the `\bin` folder. To access it, you can navigate to that folder or, if your system path is set up appropriately, you can launch into the shell by entering the `jjs` command in a Terminal / Command Prompt window on your system:



Here, you can see an open terminal window that first checks the version of Java and then uses the `jjs -version` command to launch the Nashorn shell. In this example, both Java and Nashorn are version 1.8.0.121. Alternatively, we can simply launch

Nashorn with the `jjs` command, and the shell will open without the version identification:



```
[  ] Edwards-iMac:~ edljr$ java -version
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
[Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
Edwards-iMac:~ edljr$ jjs -version
nashorn 1.8.0_121
jjs>
```

Next, let's create a short JavaScript and run it using Nashorn. Consider the following simple JavaScript code that has three simple lines of output.

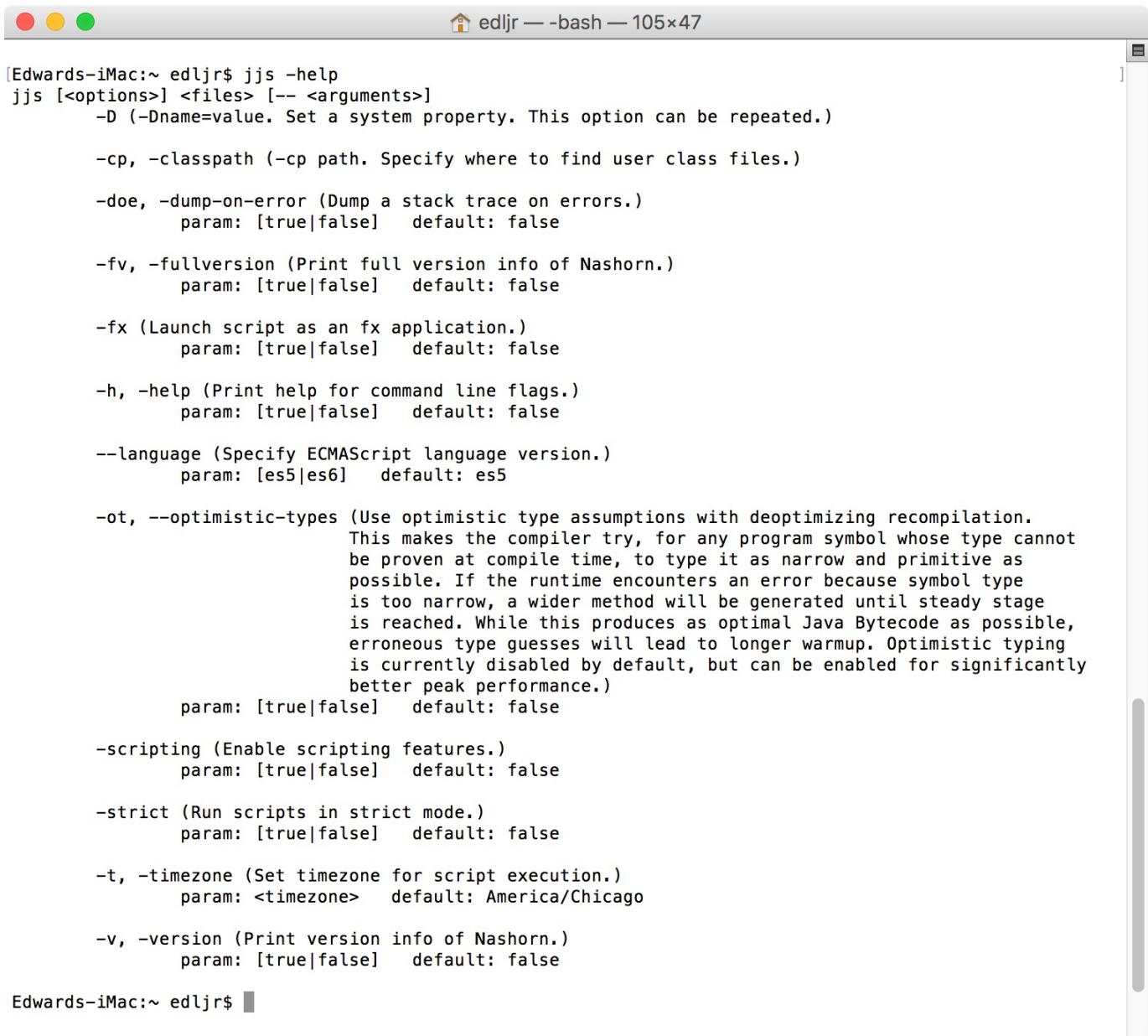
```
var addtest = function()
{
    print("Simple Test");
    print("This JavaScript program adds the numbers 300
        and 19.");
    print("Addition results = " + (300 + 19));
}
addtest();
```

To have Java run this JavaScript application, we will use the `jjs address.js` command. Here is the output:



```
[  ] Edwards-iMac:~ edljr$ jjs addtest.js
Simple Test
This JavaScript program adds the numbers 300 and 19.
Addition results = 319
Edwards-iMac:~ edljr$
```

There is a lot you can do with Nashorn. From the Command Prompt/Terminal window, we can execute `jjs` with the `-help` option to see a full list of command-line commands:



```
[Edwards-iMac:~ edljr$ jjs -help
jjs [<options>] <files> [-- <arguments>]
  -D (-Dname=value. Set a system property. This option can be repeated.)
  -cp, -classpath (-cp path. Specify where to find user class files.)
  -doe, -dump-on-error (Dump a stack trace on errors.)
    param: [true|false]  default: false
  -fv, -fullversion (Print full version info of Nashorn.)
    param: [true|false]  default: false
  -fx (Launch script as an fx application.)
    param: [true|false]  default: false
  -h, -help (Print help for command line flags.)
    param: [true|false]  default: false
  --language (Specify ECMAScript language version.)
    param: [es5|es6]  default: es5
  -ot, --optimistic-types (Use optimistic type assumptions with deoptimizing recompilation.
    This makes the compiler try, for any program symbol whose type cannot
    be proven at compile time, to type it as narrow and primitive as
    possible. If the runtime encounters an error because symbol type
    is too narrow, a wider method will be generated until steady stage
    is reached. While this produces as optimal Java Bytecode as possible,
    erroneous type guesses will lead to longer warmup. Optimistic typing
    is currently disabled by default, but can be enabled for significantly
    better peak performance.)
    param: [true|false]  default: false
  -scripting (Enable scripting features.)
    param: [true|false]  default: false
  -strict (Run scripts in strict mode.)
    param: [true|false]  default: false
  -t, -timezone (Set timezone for script execution.)
    param: <timezone>  default: America/Chicago
  -v, -version (Print version info of Nashorn.)
    param: [true|false]  default: false

Edwards-iMac:~ edljr$ ]
```

As you can see, using the `-scripting` option gives us the ability to create scripts using Nashorn as a text editor. There are several built-in functions that are useful when using Nashorn:

- `echo()`: This is similar to a `System.out.print()` Java method
- `exit()`: This exits Nashorn
- `load()`: This loads a script from a given path or URL
- `print()`: This is similar to a `System.out.print()` Java method
- `readFull()`: This reads a file's contents
- `readLine()`: This reads a single line from `stdin`
- `quit()`: This exits Nashorn

Using Nashorn as an embedded interpreter

A more common use of Nashorn, compared to using it as a command-line tool, is using it as an embedded interpreter. The `javax.script` API is public and can be accessed via the `nashorn` identifier. The following code demonstrates how we can gain access to Nashorn, define a JavaScript function, and obtain the results--all from within a Java application:

```
// required imports
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class EmbeddedAddTest
{
    public static void main(String[] args) throws Throwable
    {
        // instantiate a new ScriptEngineManager
        ScriptEngineManager myEngineManager =
            new ScriptEngineManager();

        // instantiate a new Nashorn ScriptEngine
        ScriptEngine myEngine = myEngineManager.getEngineByName(
            "nashorn");

        // create the JavaScript function
        myEngine.eval("function addTest(x, y) { return x + y; }");

        // generate output including a call to the addTest function
        // via the engine
        System.out.println("The addition results are:
            " + myEngine.eval("addTest(300, 19);"));
    }
}
```

Here is the output provided in the console window:



The screenshot shows a Java application named "EmbeddedAddTest" running in an IDE. The console tab displays the output of the program, which includes the message "The addition results are: 319.0". The rest of the console window is mostly empty, showing standard IDE interface elements like tabs for Problems, Javadoc, Declaration, and Console.

```
<terminated> EmbeddedAddTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (Aug 8, 2017, 1:44:24 PM)
The addition results are: 319.0
```

This is a simplistic example to give you an idea of what is possible with embedded use of Nashorn. There are ample examples in Oracle's official documentation.

EMCAScript

EMCA (European Computer Manufacturers Association) was formed in 1961 as a standards organization for both information systems and communications systems. Today, the EMCA continues to develop standards and issue technical reports to help standardize how consumer electronics, information systems, and communications technology are used. They are over 400 ECMA standards, most of which have been adopted.



You will notice that EMCA is not spelled with all capital letters as it is no longer considered an acronym. In 1994, the European Computer Manufacturers Association formally changed its name to EMCA.

EMCAScript, also referred to as ES, was created in 1997 as a scripted-language specification. JavaScript implements this specification. The specification includes the following:

- Complementary technologies
- Libraries
- Scripting language syntax
- Semantics

Parser API

One of the changes in the Java platform with version 9 is to provide specific support for Nashorn's EMCAScript abstract syntax tree. The goals of the new API are to provide the following:

- Interfaces to represent Nashorn syntax tree nodes
- The ability to create parser instances that can be configured with command-line options
- A visitor pattern API for interfacing with AST nodes
- Test programs to use the API

The new API, `jdk.nashorn.api.tree`, was created to permit future changes to the Nashorn classes. Prior to the new Parser API, IDEs used Nashorn's internal AST representations for code analysis. According to Oracle, use of the `idk.nashorn.internal.ir` package prevented the modernization of Nashorn's internal classes.

Here is a look at the class hierarchy of the new `jdk.nashorn.api.tree` package:

Class Hierarchy

- `java.lang.Object`
 - `jdk.nashorn.api.tree.SimpleTreeVisitorES5_1<R,P>` (implements `jdk.nashorn.api.tree.TreeVisitor<R,P>`)
 - `jdk.nashorn.api.tree.SimpleTreeVisitorES6<R,P>`
 - `java.lang.Throwable` (implements `java.io.Serializable`)
 - `java.lang.Exception`
 - `java.lang.RuntimeException`
 - `jdk.nashorn.api.tree.UnknownTreeException`

The following graphic illustrates the complexity of the new API, featuring a full interface hierarchy:

Interface Hierarchy

- jdk.nashorn.api.tree.**Diagnostic**
- jdk.nashorn.api.tree.**DiagnosticListener**
- jdk.nashorn.api.tree.**LineMap**
- jdk.nashorn.api.tree.**Parser**
- jdk.nashorn.api.tree.**Tree**
 - jdk.nashorn.api.tree.**CaseTree**
 - jdk.nashorn.api.tree.**CatchTree**
 - jdk.nashorn.api.tree.**CompilationUnitTree**
 - jdk.nashorn.api.tree.**ExportEntryTree**
 - jdk.nashorn.api.tree.**ExpressionTree**
 - jdk.nashorn.api.tree.**ArrayAccessTree**
 - jdk.nashorn.api.tree.**ArrayLiteralTree**
 - jdk.nashorn.api.tree.**AssignmentTree**
 - jdk.nashorn.api.tree.**BinaryTree**
 - jdk.nashorn.api.tree.**ClassExpressionTree**
 - jdk.nashorn.api.tree.**CompoundAssignmentTree**
 - jdk.nashorn.api.tree.**ConditionalExpressionTree**
 - jdk.nashorn.api.tree.**ErroneousTree**
 - jdk.nashorn.api.tree.**FunctionCallTree**
 - jdk.nashorn.api.tree.**FunctionExpressionTree**
 - jdk.nashorn.api.tree.**IdentifierTree**
 - jdk.nashorn.api.tree.**InstanceOfTree**
 - jdk.nashorn.api.tree.**LiteralTree**
 - jdk.nashorn.api.tree.**MemberSelectTree**
 - jdk.nashorn.api.tree.**NewTree**
 - jdk.nashorn.api.tree.**ObjectLiteralTree**
 - jdk.nashorn.api.tree.**ParenthesizedTree**
 - jdk.nashorn.api.tree.**RegExpLiteralTree**
 - jdk.nashorn.api.tree.**SpreadTree**
 - jdk.nashorn.api.tree.**TemplateLiteralTree**
 - jdk.nashorn.api.tree.**UnaryTree**
 - jdk.nashorn.api.tree.**YieldTree**
 - jdk.nashorn.api.tree.**ImportEntryTree**
 - jdk.nashorn.api.tree.**ModuleTree**
 - jdk.nashorn.api.tree.**PropertyTree**
 - jdk.nashorn.api.tree.**StatementTree**
 - jdk.nashorn.api.tree.**BlockTree**
 - jdk.nashorn.api.tree.**ClassDeclarationTree**
 - jdk.nashorn.api.tree.**DebuggerTree**
 - jdk.nashorn.api.tree.**EmptyStatementTree**
 - jdk.nashorn.api.tree.**ExpressionStatementTree**
 - jdk.nashorn.api.tree.**FunctionDeclarationTree**
 - jdk.nashorn.api.tree.**GotoTree**
 - jdk.nashorn.api.tree.**BreakTree**
 - jdk.nashorn.api.tree.**ContinueTree**
 - jdk.nashorn.api.tree.**IfTree**
 - jdk.nashorn.api.tree.**LabeledStatementTree**
 - jdk.nashorn.api.tree.**LoopTree**
 - jdk.nashorn.api.tree.**ConditionalLoopTree**
 - jdk.nashorn.api.tree.**DoWhileLoopTree**
 - jdk.nashorn.api.tree.**ForLoopTree**
 - jdk.nashorn.api.tree.**WhileLoopTree**
 - jdk.nashorn.api.tree.**ForInLoopTree**
 - jdk.nashorn.api.tree.**ForOfLoopTree**
 - jdk.nashorn.api.tree.**ReturnTree**
 - jdk.nashorn.api.tree.**SwitchTree**
 - jdk.nashorn.api.tree.**ThrowTree**
 - jdk.nashorn.api.tree.**TryTree**
 - jdk.nashorn.api.tree.**VariableTree**
 - jdk.nashorn.api.tree.**WithTree**
- jdk.nashorn.api.tree.**TreeVisitor**<R,P>

The last component of the `jdk.nashorn.api.tree` package is the enum hierarchy, shown here:

Enum Hierarchy

- `java.lang.Object`
- `java.lang.Enum<E>` (implements `java.lang.Comparable<T>`, `java.io.Serializable`)
 - `jdk.nashorn.api.tree.Diagnostic.Kind`
 - `jdk.nashorn.api.tree.Tree.Kind`

Multi-release JAR files [JEP-238]

The JAR file format has been extended in the Java 9 platform and now permits multiple versions of class files to exist in a single JAR file. The class versions can be specific to a Java release version. This enhancement allows developers to use a single JAR file to house multiple releases of their software.

The JAR file enhancement includes the following:

- Support for the `JarFile` API
- Support for standard class loaders

The changes to the JAR file format resulted in necessary changes to core Java tools so that they are able to interpret the new multiple-release JAR files. These core tools include the following:

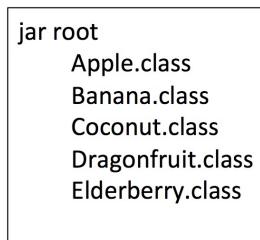
- `javac`
- `javap`
- `jdeps`

Finally, the new JAR file format supports modularity as the key characteristic of the Java 9 platform. The changes to the JAR file format have not resulted in reduced performance of related tools or processes.

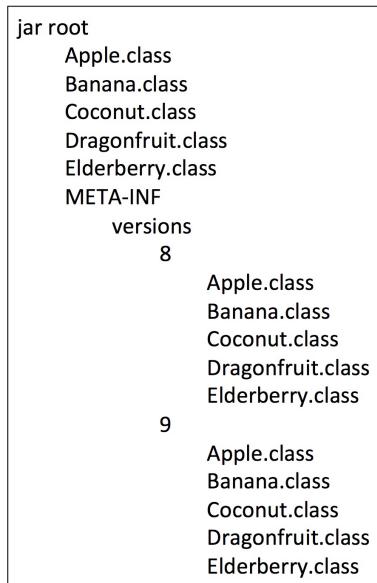
Identifying multi-release JAR files

Multi-release JAR files will have a new attribute, `Multi-Release: true`. This attribute will be located in the JAR `MANIFEST.MF` main section.

The directory structure will differ between standard JAR files and multi-release JAR files. Here is a look at a typical JAR file structure:



This illustration shows the new multi-release JAR file structure with Java version-specific class files for both Java 8 and Java 9:



Related JDK changes

Several changes had to be made to the JDK to support the new multi-release JAR file format. These changes include the following:

- The `URLClassLoader` is JAR-based and was modified so that it can read class files from the specified version.
- The new module-based class loader, new to Java 9, was written so that it can read class files from the specified version.
- The `java.util.jar.JarFile` class was modified so that it selects the appropriate class version from the multi-release JAR files.
- The JAR URL scheme's protocol handler was modified so that it selects the appropriate class version from the multi-release JAR files.
- The Java Compiler, `javac`, was made to read identified versions of the class files. These version identifications are made using the `-target` and `-release` command-line options with the `JavacFileManager` API and the `ZipFileSystem` API.
- The following tools were modified to take advantage of the changes to the `JavacFileManager` API and the `ZipFileSystem` API:
 - `javah`: This generates C header and source files
 - `schemagen`: This is the schema generator for namespaces in Java classes
 - `wsgen`: This is the parser for web service deployment
- The `javap` tool was updated to support the new versioning schema.
- The `jdeps` tool was modified to support the new versioning schema.
- The JAR packing tool set was updated accordingly. This tool set consists of `pack200` and `unpack200`.
- Of course, the JAR tool was enhanced so that it can create the multi-release JAR files.

All related documentation has been updated to support all the changes involved in establishing and supporting the new multi-release JAR file format.

Java-level JVM compiler interface [JEP-243]

The JEP-243 was to create a Java-based **JVM Compiler Interface (JVMCI)**. The JVMCI enables a Java compiler (which must have been written in Java) to be used as a dynamic compiler by the JVM.

The reasoning behind the desire for the JVMCI is that it would be a highly optimized compiler that does not require low-level language features. Some JVM subsystems require low-level functionality, such as with garbage collection and bytemode interpretation. So, the JVMCI was written in Java instead of C or C++. This provides the collateral benefit of some of Java's greatest features, such as the following ones:

- Exception handling
- IDEs that are both free and robust
- Memory management
- Runtime extensibility
- Synchronization
- Unit testing support

As JVMCI was written in Java, it will arguably be easier to maintain.

There are three primary components of the JVMCI API:

- Virtual machine data structure access
- Installing compiled code with its metadata
- Using the JVM's compilation system

The JVMCI actually existed, to some extent, in Java 8. The JVMCI API was only accessible via a class loader that worked for code on the boot class path. In Java 9, this changes. It will still be experimental in Java 9, but more accessible. In order to enable the JVMCI, the following series of command-line options must be used:

```
| -XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI -XX:+UseJVMCICompiler - Djvmci.Com
```

Oracle is keeping the JVMCI experimental in Java 9 to permit further testing and to afford the greatest level of protection for developers.

BeanInfo annotations [JEP-256]

The JEP-256 focused on replacing `@beaninfo` javadoc tags with more appropriate annotations. Furthermore, these new annotations are now processed at runtime so that `BeanInfo` classes can be generated dynamically. The modularity of Java 9 resulted in this change. The creation of custom `BeanInfo` classes has been simplified and the client library has been modularized.

In order to fully grasp this change, we will review `JavaBean`, `BeanProperty`, and `SwingContainer` before going any further into this JEP.

JavaBean

A JavaBean is a Java class. Like other Java classes, JavaBeans are reusable code. They are unique in their design because they encapsulate several objects into one. There are three conventions a JavaBean class must follow:

- The constructor should not take any arguments
- It must be serializable
- It must contain mutator and accessor methods for its properties

Here is an example JavaBean class:

```
public class MyBean implements java.io.Serializable
{
    // instance variables
    private int studentId;
    private String studentName;

    // no-argument constructor
    public MyBean()
    {

    }

    // mutator/setter
    public void setStudentId(int theID)
    {
        this.studentId = theID;
    }

    // accessor/getter
    public int getStudentId()
    {
        return studentId;
    }

    // mutator/setter
    public void setStudentName(String theName)
    {
        this.studentName = theName;
    }

    // accessor/getter
    public String getStudentName()
    {
        return studentName;
    }
}
```

Accessing `JavaBean` classes is as simple as using the mutator and accessor methods. This is likely not new to you, but there is a good chance you did not know that those carefully coded classes you created were called `JavaBean` classes.

BeanProperty

`BeanProperty` is an annotation type. We use this annotation to specify a property so that we can automatically generate `BeanInfo` classes. This is a new annotation for Java 9.

The `BeanProperty` annotation has the following optional elements:

- boolean bound
- String description
- String[] enumerationValues
- boolean expert
- boolean hidden
- boolean preferred
- boolean required
- boolean visualUpdate

SwingContainer

`SwingContainer` is an annotation type. We use this annotation to specify a swing-related property so that we can automatically generate `BeanInfo` classes. This is a new annotation for Java 9.

The `SwingContainer` annotation has the following optional elements:

- `String delegate`
- `boolean value`

Now that we have reviewed `JavaBean`, `BeanProperty`, and `SwingContainer`, let's take a look at the `BeanInfo` classes.

BeanInfo classes

For the most part, `BeanInfo` classes are automatically generated at runtime. The exception is with Swing classes. Those classes generate `BeanInfo` classes based on the `@beaninfo` javadoc tags. This is done at compile time, not runtime. In Java 9, the `@beaninfo` tags have been replaced with `@interface JavaBean`, `@interface BeanProperty`, and `@interface SwingContainer` annotations.

These new annotations are used to set the corresponding attributes based on the optional elements noted in the previous sections. As an example, the following code snippet sets the attributes for a `SwingContainer`:

```
package javax.swing;

public @interface SwingContainer
{
    boolean value() default false;
    String delegate() default "";
}
```

This provides us with three benefits:

- It will be much easier to specify attributes in Bean classes instead of having to create individual `BeanInfo` classes
- We will be able to remove auto-generated classes
- The client library is much more easily modularized with this approach

TIFF image input/output [JEP-262]

JEP-262 is pretty straight forward. For Java 9, the image input/output plugins have been extended to include support for the TIFF image format. The `ImageIO` class extends the `Object` class and is part of Java SE. The class contains several methods for encoding and decoding images. Here is a list of static methods:

Method	Return value
<code>createImageInputStream(Object input)</code>	<code>ImageInputStream</code>
<code>createImageOutputStream(Object output)</code>	<code>ImageOutputStream</code>
<code>getCacheDirectory()</code>	Current value of the <code>CacheDirectory</code>
<code>getImageReader(ImageWriter writer)</code>	<code>ImageReader</code>
<code>getImageReaders(Object input)</code>	Iterator of current <code>ImageReaders</code>
<code>getImageReadersByFormatName(String formatName)</code>	Iterator of current <code>ImageReaders</code> with the specified format name
<code>getImageReadersByMimeType(String mimeType)</code>	Iterator of current <code>ImageReaders</code> of the specified MIME type
<code>getImageReadersBySuffix(String fileSuffix)</code>	Iterator of current <code>ImageReaders</code> with the specified suffix.
<code>getImageTranscoders(ImageReader reader)</code>	Iterator of current <code>ImageTranscoders</code>
<code>getImageWriter(ImageReader reader)</code>	<code>ImageWriter</code>
<code>getImageWriters(ImageTypeSpecifier type, String formatName)</code>	Iterator of current <code>ImageWriters</code> that can encode to the specified type
<code>getImageWritersByFormatName(String formatName)</code>	Iterator of current <code>ImageWriters</code> with the specified format name
<code>getImageWritersByMimeType(String mimeType)</code>	Iterator of current <code>ImageWriters</code> of the specified MIME type
<code>getImageWritersBySuffix(String fileSuffix)</code>	Iterator of current <code>ImageWriters</code> with the specified suffix.
<code>getReaderFileSuffixes()</code>	String array with file suffixes understood

	by current readers
getReaderFormatNames()	String array with format names understood by current readers
getReaderMIMETypes()	String array with MIME types understood by current readers
getUseCache()	UseCache value
getWriterFileSuffixes()	String array of file suffixes understood by current writers
getWriterFormatNames()	String array with format names understood by current writers
getWriterMIMETypes()	String array with MIME types understood by current writers
read(File input)	BufferedImage with an ImageReader
read(ImageInputStream stream)	BufferedImage with ImageInputStream and an ImageReader
read(InputStream input)	BufferedImage with InputStream and ImageReader
read(URL input)	BufferedImage with an ImageReader

There are also a few static methods that do not return a value or return a Boolean:

Method	Description
scanForPlugins()	<p>Performs the following actions:</p> <ul style="list-style-type: none"> • Scans the application classpath for plugins • Loads plugin service provider classes • Registers service provide instances in the IIORRegistry
setCacheDirectory(File cacheDirectory)	This is where the cache files will be stored.
	This method toggles if the cache will be disk-based

<code>setUseCache(boolean useCache)</code>	or not. This applies to <code>ImageInputStream</code> and <code>ImageOutputStream</code> instances.
<code>write(RenderedImage im, String formatName, File output)</code>	Writes an image to the specified file.
<code>write(RenderedImage im, String formatName, ImageOutputStream output)</code>	Writes an image to an <code>ImageOutputStream</code> .
<code>write(RenderedImage im, String formatName, OutputStream output)</code>	Writes an image to an <code>OutputStream</code> .

As you can glean from the provided methods, the image input/output framework provides us with a convenient way of using image codecs. As of Java 7, the following image format plugins were implemented by `javax.imageio`:

- BMP
- GIF
- JPEG
- PNG
- WBMP

The TIFF is, as you can see, not on the list of image file formats. TIFFs are a common file format and, in 2001, macOS, with the release of MacOS X, used the format extensively.

The Java 9 platform includes `ImageReader` and `ImageWriter` plugins for the TIFFs. These plugins have been written in Java and have been bundled in the new `javax.imageio.plugins.tiff` package.

Platform logging API and service [JEP-264]

The Java 9 platform includes a new logging API enabling platform classes to log messages. It has a commensurate service for manipulating the logs. Before we go too far into what is new regarding the logging API and service, let's review `java.util.logging.api` which was introduced in Java 7.

The `java.util.logging` package

The `java.util.logging` package includes classes and interfaces that collectively comprise Java's core logging features. This functionality was created with the following goals:

- Problem diagnosis by end users and system administrators
- Problem diagnosis by field service engineers
- Problem diagnosis by the development organization

As you can see, the primary purpose was to enable maintenance of remote software.

The `java.util.logging` package has two interfaces:

- `public interface Filter`
 - Purpose: This provides fine-grain control over logged data
 - Method:
 - `isLoggable(LogRecord record)`
- `public interface LoggingMXBean`
 - Purpose: This is the logging facility's management interface
 - Methods:
 - `getLoggerLevel(String loggerName)`
 - `getLoggerNames()`
 - `getparentLoggerName(String loggerName)`
 - `setLoggerLevel(String loggerName, String levelName)`

The following table provides the `java.util.logging` package classes, along with a brief description regarding what each class provides in respect to logging functionality and management:

Class	Definition	Description
ConsoleHandler	<pre>public class ConsoleHandler extends StreamHandler</pre>	Publishes log records to <code>System.err</code>
ErrorManager	<pre>public class ErrorManager extends Object</pre>	Used to process errors during logging
	<pre>public class</pre>	

FileHandler	FileHandler extends StreamHandler	File logging
Formatter	public abstract class Formatter extends Object	For formatting LogRecords
Handler	public abstract class Handler extends Object	Exports Logger messages
Level	public class Level extends Object implements Serializable	Controls level of logging. The levels, in descending order, are--severe, warning, info, config, fine, finer, and finest
Logger	public class Logger extends Object	Logs messages
LoggingPermission	public final class LoggingPermission extends BasicPermission	SecurityManager checks this
LogManager	public class LogManager	For maintaining shared state between loggers and logging services
LogRecord	public class LogRecord extends Object implements Serializable	Passed between handlers
MemoryHandler	public class MemoryHandler extends Handler	Buffers requests in memory
SimpleFormatter	public class SimpleFormatter extends Formatter	Provides human-readable LogRecord metadata
SocketHandler	public class SocketHandler extends StreamHandler	Network logging handler
StreamHandler	public class StreamHandler extends Handler	Stream-based logging handler
XMLFormatter	public class XMLFormatter extends Formatter	Formats logs into XML

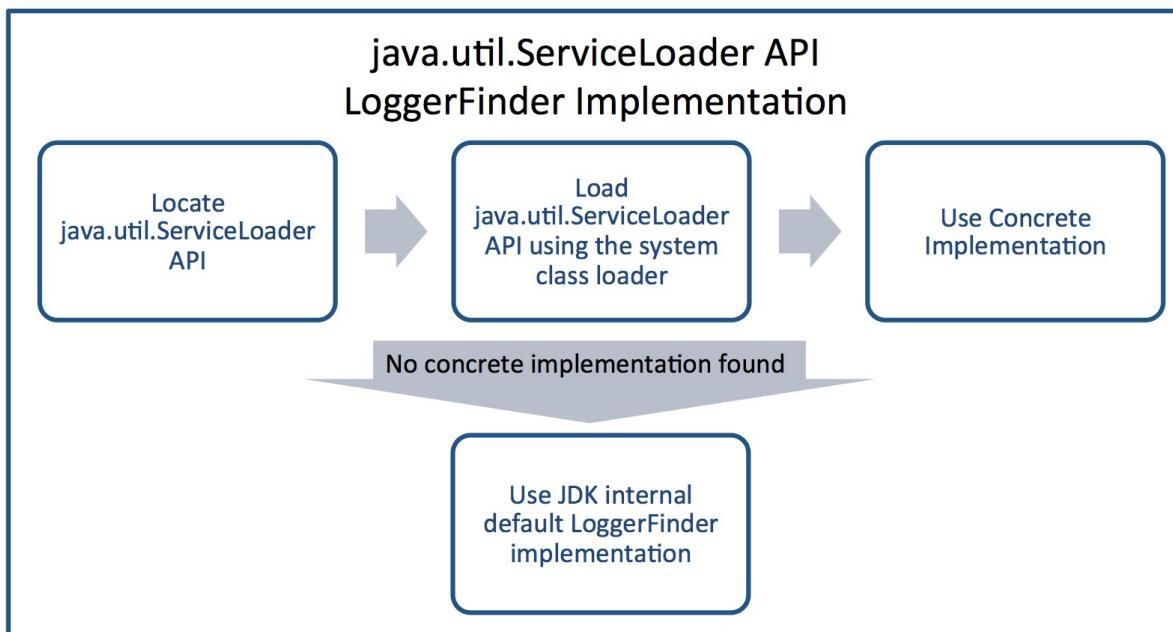
Next, let's review what changes were made in Java 9.

Logging in Java 9

Prior to Java 9, there were multiple logging schemas available, including `java.util.logging`, `SLF4J`, and `Log4J`. The latter two are third-party frameworks that have separate facade and implementation components. This pattern has been replicated in the new Java 9 platform.

Java 9 introduced changes to the `java.base` module so that it would handle logging functions and not rely on the `java.util.logging` API. It has separate facade and implementation components. This means that when using third-party frameworks, the JDK only needs to provide the implementation component and return platform loggers that work with the requesting logging framework.

As you can see in the following illustration, we use the `java.util.ServiceLoader` API to load our `LoggerFinder` implementation. The JDK uses a default implementation if a concrete implementation is not found using the system class loader:



XML Catalogs [JEP-268]

JEP 268, titled XML Catalogs, focused on creating a standard XML Catalog API to support the OASIS XML Catalogs Standard v1.1. The new API defines catalog and catalog-resolve abstractions so that JAXP processors can use them. In this section, we will look at the following:

- The OASIS XML Catalog standard
- JAXP processors
- XML Catalogs prior to Java 9
- Java 9 platform changes

The OASIS XML Catalog standard

XML (eXtensible Markup Language) Catalogs are XML documents consisting of catalog entries. Each entry pairs an identifier to another location. OASIS is a not-for-profit consortium with the mission of advancing open standards. They published the XML catalog standard, version 1.1., in 2005. This standard has two basic use cases:

- Map an external identifier to a URI reference
- Map a URI reference to another URI reference

Here is a sample XML catalog entry:

```
| <public publicId="-//Packt Publishing Limited//Mastering Java 9//EN"  
|   uri="https://www.packtpub.com/application-development/mastering-java-9"/>
```

The complete OASIS XML Catalog standard can be found at the official site: <https://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html>

JAXP processors

The Java API for XML processing is referred to as JAXP. As its name suggests, this API is used for parsing XML documents. There are four related interfaces:

- DOM: Document Object Model parsing
- SAX: Simple API for XML parsing
- StAX: Streaming API for XML parsing
- XSLT: Interface to transform XML documents

XML Catalogs prior to Java 9

The Java platform has had an internal catalog resolver since JDK 6. There was no public API, so external tools and libraries were used to access the functionality. Moving into Java 9, the goal was to make the internal catalog resolver a standard API for common use and ease of support.

Java 9 platform changes

The new XML Catalog API, delivered with Java 9, follows the OASIS XML Catalogs standard, v1.1. Here are the feature and capability highlights:

- Implements `EntityResolver`
- Implements `URIResolver`
- Creation of XML Catalogs is possible via the `CatalogManager`
- `CatalogManager` will be used to create `CatalogResolvers`
- OASIS open catalog file semantics will be followed
 - Map an external identifier to a URI reference
 - Map a URI reference to another URI reference
- `CatalogResolvers` will implement the JAXP `EntityResolver` interface
- `CatalogResolvers` will implement the JAXP `URIResolver` interface
- The SAX `XMLFilter` will be supported by the resolver.

Since the new XML Catalog API will be public, the pre-Java 9 internal catalog resolver will be removed, as it will no longer be necessary.

Convenience factory methods for collections [JEP-269]

The Java programming language does not support collection literals. Adding this feature to the Java platform was proposed in 2013 and revisited in 2016, but it only gained exposure as a research proposal, not for future implementation.



Oracle's definition of a collection literal is "a syntactic expression form that evaluates to an aggregate type, such as an array, list, or map" (<http://openjdk.java.net/jeps/186>).

Of course, that is until Java 9 is released. Implementing collection literals in the Java programming language is reported to have the following benefits:

- Performance improvement
- Increased safety
- Reduction of boilerplate code

Even without being part of the research group, our knowledge of the Java programming language clues us in to additional benefits:

- Ability to write shorter code
- Ability to write space-efficient code
- Ability to make collection literals immutable

Let's look at two cases--using collections before Java 9, and then with the new support for collection literals in the new Java platform.

Using collections before Java 9

Here is an example of how we would create our own collections prior to Java 9. This first class defines the structure for `PlanetCollection`. It has the following components:

- A single instance variable
- A one argument constructor
- Mutator/setter method
- Accessor/getter method
- Method to print the object

Here is the code implementing the preceding listed constructor and methods:

```
public class PlanetCollection
{
    // Instance Variable
    private String planetName;

    // constructor
    public PlanetCollection(String name)
    {
        setPlanetName(name);
    }

    // mutator
    public void setPlanetName(String name)
    {
        this.planetName = name;
    }

    // accessor
    public String getPlanetName()
    {
        return this.planetName;
    }

    public void print()
    {
        System.out.println(getPlanetName());
    }
}
```

Now, let's look at the driver class that populates the collection:

```
import java.util.ArrayList;

public class OldSchool
{
    private static ArrayList<PlanetCollection> myPlanets =
        new ArrayList<>();
```

```

public static void main(String[] args)
{
    add("Earth");
    add("Jupiter");
    add("Mars");
    add("Venus");
    add("Saturn");
    add("Mercury");
    add("Neptune");
    add("Uranus");
    add("Dagobah");
    add("Kobol");

    for (PlanetCollection orb : myPlanets)
    {
        orb.print();
    }
}

public static void add(String name)
{
    PlanetCollection newPlanet = new PlanetCollection(name);
    myPlanets.add(newPlanet);
}
}

```

Here is the output from this application:

```

Problems @ Javadoc Declaration Console ✎
<terminated> OldSchool [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (Aug 9, 2017, 7:02:41 PM)
Earth
Jupiter
Mars
Venus
Saturn
Mercury
Neptune
Uranus
Dagobah
Kobol

```

This code is, unfortunately, very verbose. We populated our collection in static initializer blocks instead of using a field initializer. There are other methods of populating our list, but they are all more verbose than they should have to be. These other methods have additional problems, such as the need to create extra classes, the use of obscure code, and hidden references.

Let's now take a look at the solution to this problem, provided by the new Java 9 platform. We will look at what is new in the next section.

Using new collection literals

In order to rectify the currently required code verbosity in creating collections, we need library APIs for creating collection instances. Look at our pre-Java 9 code snippet in the previous section and then consider this possible refactoring:

```
PlanetCollection<String> myPlanets = Set.of(  
    "Earth",  
    "Jupiter",  
    "Mars",  
    "Venus",  
    "Saturn",  
    "Mercury",  
    "Neptune",  
    "Uranus",  
    "Dagobah",  
    "Kobol");
```

This code is highly human-readable and not verbose.

The new implementation will include static factory methods on the following interfaces:

- `List`
- `Map`
- `Set`

So, we are now able to create unmodifiable instances of `List` collections, `Map` collections, and `Set` collections. They can be instantiated with the following syntax:

- `List.of(a, b, c, d, e);`
- `Set.of(a, b, c, d, e);`
- `Map.of();`

The `Map` collections will have a set of fixed arguments.

Platform-specific desktop features [JEP-272]

The exciting JEP-272 was to create a new public API so that we can write applications with access to platform-specific desktop features. These features include interacting with task bars/docks and listening for application and system events.

The macOS X `com.apple.eawt` package was an internal API and, starting with Java 9, is no longer accessible. In support of Java 9's new embedded platform-specific desktop features, `apple.applescript` classes are being removed from the Java platform without replacement.

This effort had several objectives:

- Create a public API to replace the functionality in `com.apple.{east,eio}`
- Ensure OS X developers do not lose functionality. To this end, the Java 9 platform has replacements for the following packages:
 - `com.apple.eawt`
 - `com.apple.eio`
- Provide developers with a near-common set of features for platforms (that is, Windows and Linux) in addition to OS X. The common features include:
 - Login/logout handler with event listeners
 - Screen lock handler with event listeners
 - Task bar / dock actions to include:
 - Requesting user attention
 - Indicating task progress
 - Action shortcuts

The new API will be added to the `java.awt.Desktop` class.

Enhanced method handles [JEP-274]

The **Enhanced Method Handles** JEP-274 was to improve the following listed classes, to make common usage easier with improved optimizations:

- `MethodHandle` class
- `MethodHandles` class
- `MethodHandles.Lookup` class

The listed classes are all part of the `java.lang.invoke` package, which has been updated as part of the Java 9 platform. The improvements were made possible through the use of lookup refinement and `MethodHandle` combinations for loops and `try...finally` blocks.

In this section, we will look at the following regarding JEP-274:

- Reason for the enhancement
- Lookup functions
- Argument handling
- Additional combinations

Reason for the enhancement

This enhancement stemmed from developer feedback and the desire to make the `MethodHandle`, `MethodHandles`, and `MethodHandles.Lookup` classes much easier to use. There was also the call to add additional use cases.

The changes resulted in the following benefits:

- Enabled precision in the usage of the `MethodHandle` API
- Instantiation reduction
- Increased JVM compiler optimizations

Lookup functions

Changes regarding lookup functions, for the Java 9 platform, include the following:

- `MethodHandles` can now be bound to non-abstract methods in interfaces
- The lookup API allows class lookups from different contexts

The `MethodHandles.Lookup.findSpecial(Class<?> refs, String name, MethodType type, Class<?> specialCaller)` class has been modified to permit locating super-callable methods on interfaces.

In addition, the following methods have been added to the `MethodHandles.Lookup` class:

- `Class<?> findClass(String targetName)`
- `Class<?> accessClass(Class<?> targetClass)`

Argument handling

Three updates were made to improve `MethodHandle` argument handling for the Java 9 platform. These changes are highlighted as follows:

- Argument folding using `foldArguments(MethodHandle target, MethodHandle combinator)` did not previously have a position argument.
 - Argument collection using the `MethodHandle.asCollector(Class<?> arrayType, int arrayLength)` method did not previously support collecting the arguments into an array except for the trailing element. This has been changed, and there is now an additional `asCollector` method to support that functionality in Java 9.
- Argument spreading using the `MethodHandle.asSpreader(Class<?> arrayType, int arrayLength)` method spreads the contents of the trailing array to a number of arguments, in a reverse method of argument collection. Argument spreading has been modified to support the expansion of an array anywhere in the method signature.



The new method definitions for the updated `asCollector` and `asSpreader` methods are provided in the next section.

Additional combinations

The following addition combinations have been added to support the ease of use and optimizations for the `MethodHandle`, `MethodHandles`, and `MethodHandles.Lookup` classes of the `java.lang.invoke` package in the Java 9 platform:

- Generic loop abstraction:
 - `MethodHandle loop(MethodHandle[] . . . clauses)`
- While loops:
 - `MethodHandle whileLoop(MethodHandle init, MethodHandle pred, MethodHandle body)`
- Do...while loops:
 - `MethodHandle doWhileLoop(MethodHandle init, MethodHandle body, MethodHandle pred)`
- Counting loops:
 - `MethodHandle countedLoop(MethodHandle iterations, MethodHandle init, MethodHandle body)`
- Data structure iteration:
 - `MethodHandle iteratedLoop(MethodHandle iterator, MethodHandle init, MethodHandle body)`
- Try...finally blocks:
 - `MethodHandle tryFinally(MethodHandle target, MethodHandle cleanup)`
- Argument handling:
 - Argument spreading:
 - `MethodHandle asSpreader(int pos, Class<?> arrayType, int arrayLength)`
 - Argument collection:
 - `MethodHandle asCollector(int pos, Class<?> arrayType, int arrayLength)`
 - Argument folding:
 - `MethodHandle foldArguments(MethodHandle target, int pos, MethodHandle combiner)`

Enhanced deprecation [JEP-277]

There are two facilities for expressing deprecation:

- `@Deprecated` annotation
- `@deprecated` javadoc tag

These facilities were introduced in Java SE 5 and JDK 1.1, respectively. The `@Deprecated` annotation was intended to annotate program components that should not be used because they were deemed dangerous and/or there was a better option. That was the intended use. Actual use varied including and because warnings were only provided at compile time; there was little reason to ignore the annotated code.

The **Enhanced Deprecation** JEP-277 was taken on to provide developers with clearer information regarding the intended disposition of the APIs in the specification documentation. Work on this JEP also resulted in a tool for analyzing a program's use of deprecated APIs.

To support this fidelity in information, the following components have been added to the `java.lang.Deprecated` annotation type:

- `forRemoval()`:
 - Returns Boolean `true` if the API element has been slated for future removal
 - Returns Boolean `false` if the API element has not been slated for future removal but is deprecated
 - Default is `false`
- `since()`:
 - Returns a string containing the release or version number, at which point the specified API was marked as deprecated

What the @Deprecated annotation really means

When an API, or methods within an API, has/have been marked with the `@Deprecated` annotation, one or more of the following conditions typically exists:

- There are errors in the API for which there is no plan to fix them
- Using the API is likely to result in errors
- The API has been replaced by another API
- The API is experimental

Summary

In this chapter, we covered 16 JEPs that were incorporated into the Java 9 platform. These JEPs cover a wide range of tools and updates to APIs to make developing with Java easier, with greater optimization possibilities for our resulting programs. Our review included a look at the new HTTP client, changes to Javadoc and the Doclet API, the new JavaScript parser, JAR and JRE changes, the new Java-level JVM compiler interface, the new support for TIFF images, platform logging, XML Catalog support, collections, and the new platform-specific desktop features. We also looked at enhancements to method handling and the deprecation annotation.

In the next chapter, we will cover concurrency enhancements introduced with the Java 9 platform. Our primary focus will be the support for reactive programming that is provided by the flow class API. We will also explore additional concurrency enhancements introduced in Java 9.

Concurrency and Reactive Programming

In the previous chapter, we covered several **Java Enhancement Proposals (JEPs)** that were incorporated into the Java 9 platform. These JEPs represented a wide range of tools and updates to APIs to make developing with Java easier, with greater optimization possibilities for our Java applications. We looked at the new HTTP client, changes to Javadoc and the Doclet API, the new JavaScript parser, JAR and JRE changes, the new Java-level JVM compiler interface, the new support for TIFF images, platform logging, XML catalog support, collections, and the new platform-specific desktop features. We also looked at enhancements to method handling and the deprecation annotation.

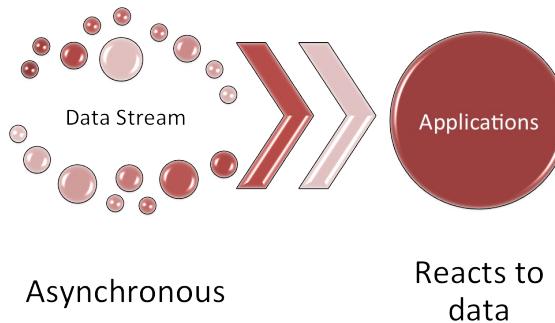
In this chapter we will cover concurrency enhancements introduced with the Java 9 platform. Our primary focus will be the support for reactive programming, a concurrency enhancement that is provided by the `Flow` class API. Reactive programming is a new concept for Java 9, so we will take an exploratory approach to the topic. We will also explore additional concurrency enhancements introduced in Java 9.

Specifically, we will cover the following topics:

- Reactive programming
- The new `Flow` API
- Additional concurrency updates
- Spin-wait hints

Reactive Programming

Reactive programming is when applications react to an asynchronous data stream as it occurs. The following image illustrates the flow:



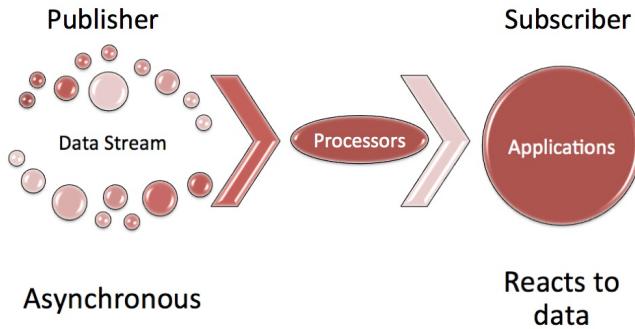
Reactive programming is not a fancy software engineering term only used by academics. It is, in fact, a programming model that can result in much greater efficiencies as opposed to the more common method of having applications iterate over data that is in memory.

There is more to reactive programming. First, let's consider that the data stream is provided by a publisher in an asynchronous manner to the subscriber.



Data streams are a binary input/output of strings and primitive data types. The `DataInput` interface is used for an input stream and the `DataOutput` interface is used for output streams.

Processors, or a chain of processors, can be used to transform the data stream without the publisher or subscriber being impacted. In the following example, the **Processors** work on the stream of data without **Publisher** or **Subscriber** involvement, or even awareness:



In addition to greater efficiency, reactive programming represents several additional benefits, which are highlighted here:

- The code base can be less verbose, making it:
 - Easier to code
 - Easier to maintain
 - Easier to read
- Stream processing results in memory efficiencies
- This is a solution for a variety of programming applications
- Less boiler-plate code needs to be written, so development time can be focused on programming core functionalities
- The following types of programming require less time and code:
 - Concurrency
 - Low-level threading
 - Synchronization

Reactive programming standardization

There are standards in many aspects of software development, and reactive programming has not escaped this. There is a **Reactive Streams** initiative to standardize asynchronous stream processing. The specific focus, in the context of Java, is with the JVM and JavaScript.

The Reactive Streams initiative aims at tackling the issue of governing how the data stream is exchanged between threads. As you will recall from the previous section, the idea of processors is predicated on there being no impact on the publisher or receiver. This no-impact mandate stipulates that the following are not required:

- Data buffering
- Data translation
- Conversion

The basic semantics of the standard define the regulation of data stream element transmission. This standard was specifically established for delivery with the Java 9 platform. Reactive Streams includes a library that will help developers convert from `org.reactivestreams` and `java.util.concurrent.Flow` namespaces.

The key to being successful with reactive programming and the Reactive Streams standardization is understanding the relevant terminology:

Term	Description
Demand	Demand refers to the subscriber's request for more elements as well as referring to the total number of elements requested that have not been fulfilled by the publisher yet.
Demand	Demand also refers to the total number of elements requested that have not been fulfilled by the publisher yet.
External synchronization	External access coordination for thread safety.
	Methods are said to be non-obstructing if they rapidly execute

Non-obstructing	without the requirement for heavy computations. Non-obstructing methods do not delay a subscriber's thread execution.
NOP	NOP execution is execution that can be called repeatedly without impact to the calling thread.
Responsivity	This term refers to a component's ability to respond.
Return normally	Return normally refers to when there are no errors--the normal condition. The <code>onError</code> method is the only way permitted by the standard to inform the subscriber of a failure.
Signal	<p>One of the following methods:</p> <ul style="list-style-type: none"> • <code>cancel</code> • <code>onComplete</code> • <code>onError</code> • <code>onNext</code> • <code>onSubscribe</code> • <code>request</code>

You can obtain the standard on Maven Central (<https://search.maven.org>). Here is the standard from Maven Central as of the publication date of this book:

```

<dependency>
  <groupId>org.reactivestreams</groupId>
  <artifactId>reactive-streams</artifactId>
  <version>1.0.1</version>
</dependency>

<dependency>
  <groupId>org.reactivestreams</groupId>
  <artifactId>reactive-streams-tck</artifactId>
  <version>1.0.0</version>
  <scope>test</scope>
</dependency>

```

In the next section, we will look at the Flow APIs in the Java 9 platform, as they correspond to the Reactive Streams specification.

The New Flow API

The `Flow` class is part of the `java.util.concurrent` package. It helps developers incorporate reactive programming in their applications. The class has one method, `defaultBufferSize()`, and four interfaces.

The `defaultBufferSize()` is a static method that returns the default buffer size for publishing and subscribing buffering. This default value is `256` and it is returned as an `int`. Let's look at the four interfaces.

The Flow.Publisher interface

The `Flow.Publisher` interface is a functional interface. A `Publisher` is a producer of data sent to subscribers:

```
| @FunctionalInterface  
| public static interface Flow.Publisher<T>
```

This functional interface can serve as a lambda expression assignment target. It only takes one argument--the subscribed item type `<T>`. It has one method:

- `void onSubscribe(Flow.Subscription subscription)`

The Flow.Subscriber interface

The `Flow.Subscriber` interface is used to receive messages and its implementation is shown here:

```
|     public static interface Flow.Subscriber<T>
```

This interface is set up to receive messages. It only takes one argument--the subscribed item type `<T>`. It has the following methods:

- `void onComplete()`
- `void onError(Throwable throwable)`
- `void onNext(T item)`
- `void onSubscribe(Flow.Subscription subscription)`

The Flow.Subscription interface

The `Flow.Subscription` interface ensures that only subscribers receive what is requested. Also, as you will see here, a subscription can be cancelled at anytime:

```
|     public static interface Flow.Subscription
```

This interface does not take any arguments and is the linkage that controls the messages between instances of `Flow.Publisher` and `Flow.Subscriber`. It has the following methods:

- `void cancel()`
- `void request(long n)`

The Flow.Processor interface

The `Flow.Processor` interface can serve as both a `Subscriber` and a `Publisher`. The implementation is provided here:

```
| static interface Flow.Processor<T,R> extends Flow.Subscriber<T>,
|   Flow.Publisher<R>
```

This interface takes two arguments--the subscribed item type `<T>` and the published item type `<R>`. It does not have its own methods, but does inherit the following method from `java.util.concurrent.Flow.Publisher`:

- `void subscribe(Flow.Subscriber<? super T> subscriber)`

`Flow.Processor` also inherits the following methods from the `java.util.concurrent.Flow.Subscriber` interface:

- `void onComplete()`
- `void onError(Throwable throwable)`
- `void onNext(T item)`
- `void onSubscribe(Flow.Subscription subscription)`

Sample implementation

In any given implementation of reactive programming, we will have a `Subscriber` that requests data and a `Publisher` that provides the data. Let's first look at a sample `Subscriber` implementation:

```
import java.util.concurrent.Flow.*;

public class packtSubscriber<T> implements Subscriber<T>
{
    private Subscription theSubscription;

    // We will override the four Subscriber interface methods

    @Override
    public void onComplete()
    {
        System.out.println("Data stream ended");
    }

    @Override
    public void onError(Throwable theError)
    {
        theError.printStackTrace();
    }

    @Override
    public void onNext(T theItem)
    {
        System.out.println("Next item received: " + theItem);
        theSubscription.request(19); // arbitrary number for
        example purposes
    }

    @Override
    public void onSubscribe(Subscription theSubscription)
    {
        this.theSubscription = theSubscription;
        theSubscription.request(19);
    }
}
```

As you can see, implementing the `Subscriber` is not difficult. The heavy work is done with the processors in-between the `Subscriber` and `Publisher`. Let's look at a sample implementation where the `Publisher` publishes a data stream to subscribers:

```
import java.util.concurrent.SubsmissionPublisher;

. . .

// First, let's create a Publisher instance
```

```
SubmissionPublisher<String> packtPublisher = new
SubmissionPublisher<>();

// Next, we will register a Subscriber
PacktSubscriber<String> currentSubscriber = new
PacktSubscriber<>();
packtPublisher.subscribe(currentSubscriber);

// Finally, we will publish data to the Subscriber and
// close the publishing effort
System.out.println("||---- Publishing Data Stream ----||");
. . .
packtPublisher.close();
System.out.println("||---- End of Data Stream Reached ----||");
```

Additional Concurrency Updates

The **More Concurrency Updates** Java Enhancement Proposal, JEP 266, aimed to improve the use of concurrency in Java. In this section, we will briefly explore the concept of Java concurrency and look at related enhancements to the Java 9 platform:

- Java concurrency
- Supporting Reactive Streams
- `CompletableFuture` API enhancements

Java concurrency

In this section, we will start with a brief explanation of concurrency, then look at system configurations, cover Java threads, and then look at the concurrency improvements.

Concurrency explained

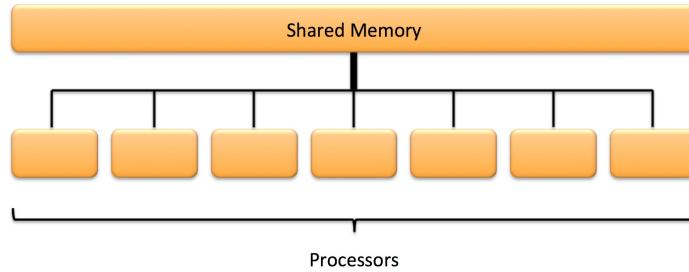
Concurrent processing has been around since the 1960s. In those formative years, we already had systems that permitted multiple processes to share a single processor. These systems are more clearly defined as pseudo-parallel systems because it only appeared that multiple processes were being simultaneously executed. Our computers today still operate in this manner. The difference between the 1960s and current day is that our computers can have multiple CPUs, each with multiple cores, which better supports concurrency.



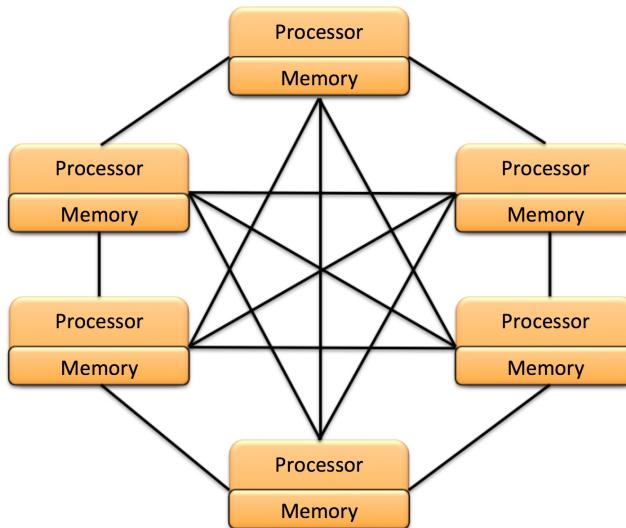
Concurrency and parallelism are often used as interchangeable terms. Concurrency is when multiple processes overlap, although the start and stop times could be different. Parallelism occurs when tasks start, run, and stop at the same time.

System configurations

There are several different processor configurations that need to be considered. This section features two common configurations. The first configuration is that of shared memory and is illustrated here:



As you can see, the shared memory system configuration has multiple processors that all share a common system memory. The second featured system configuration is a distributed memory system:



With the distributed memory system, each processor has its own memory and each individual processor is fully linked with the other processors, making for a distributed system that is fully linked.

Java threads

A thread in Java is a program execution and is built into the JVM. The `Thread` class is part of the `java.lang` package (`java.lang.Thread`). Threads have priorities that control in what order the JVM executes them. While the concept is simple, implementation is not. Let's start by taking a close look at the `Thread` class.

The `Thread` class has two nested classes:

- `public static enum Thread.State`
- `public static interface Thread.UncaughtExceptionHandler`

There are three instance variables for managing thread priorities:

- `public static final int MAX_PRIORITY`
- `public static final int MIN_PRIORITY`
- `public static final int NORM_PRIORITY`

The `Thread` class has eight constructors, all of which allocate a new `Thread` object. Here are the constructor signatures:

- `public Thread()`
- `public Thread(Runnable target)`
- `public Thread(Runnable target, String name)`
- `public Thread(String name)`
- `public Thread(ThreadGroup group, Runnable target)`
- `public Thread(ThreadGroup group, Runnable target, String name)`
- `public Thread(ThreadGroup group, Runnable target, String name, long stackSize)`
- `public Thread(ThreadGroup group, String name)`

The `Thread` class also has 43 methods, six of which have been deprecated. The remaining methods are listed here, save for the accessors and mutators which are listed separately. You can consult the documentation for details about each of these methods:

- `public static int activeCount()`
- `public final void checkAccess()`
- `protected Object clone() throws CloneNotSupportedException`

- public static Thread currentThread()
- public static void dumpStack()
- public static int enumerate(Thread[] array)
- public static boolean holdsLock(Object obj)
- public void interrupt()
- public static boolean interrupted()
- public final boolean isAlive()
- public final boolean isDaemon()
- public boolean isInterrupted()
- **join methods:**
 - public final void join() throws InterruptedException
 - public final void join(long millis) throws InterruptedException
 - public final void join(long millis, int nano) throws InterruptedException
- public void run()
- **sleep methods:**
 - public static void sleep(long mills) throws InterruptedException
 - public static void sleep(long mills, int nano) throws InterruptedException
- public void start()
- public String toString()
- public static void yield()

Here is the list of accessors/getters and mutators/setters for the `Thread` class:

- **accessors/getters:**
 - public static Map<Thread, StackTraceElement[]> getAllStacktraces()
 - public ClassLoader getContextClassLoader()
 - public static Thread.UncaughtExceptionHandler
getDefaultUncaughtExceptionHandler()
 - public long getId()
 - public final String getName()
 - public final int getPriority()
 - public StackTraceElement[] getStackTrace()
 - public Thread.State getState()
 - public final ThreadGroup getThreadGroup()
 - public Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()
- **mutators/setters:**
 - public void setContextClassLoader(ClassLoader cl)
 - public final void setDaemon(boolean on)
 - public static void

```
    setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)
• public final void setName(String name)
• public final void setPriority(int newPriority)
• public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)
```

In Java, concurrency is commonly referred to as multithreading. As indicated earlier, managing threads, and especially multithreads, requires great fidelity in control. Java employs a couple of techniques including the use of locks. Code segments can be locked to ensure that only a single thread can execute that code at any given time. We can lock classes and method with the use of the `synchronized` keyword. Here is an example of how to lock an entire method:

```
public synchronized void protectedMethod()
{
    . . .
}
```

The next code snippet demonstrates how to use the `synchronized` keyword to lock blocks of code within a method:

```
. . .
public class unprotectedMethod()
{
    . . .
    public int doSomething(int tValue)
    {
        synchronized (this)
        {
            if (tValue != 0)
            {
                // do something to change tValue
                return tValue;
            }
        }
    }
}
```

Concurrency improvements

The ability to employ multiple threads in our Java applications stands to greatly improve efficiency and leverage the increasing processing capabilities of modern computers. The use of threads in Java gives us great granularity in our concurrency controls.

Threads are at the core of Java's concurrency functionality. We can create a thread in Java by defining a `run` method and instantiating a `Thread` object. There are two methods of accomplishing this set of tasks. Our first option is to extend the `Thread` class and override the `Thread.run` method. Here is an example of that approach:

```
. . .
class PacktThread extends Thread
{
    . . .
    public void run()
    {
        . . .
    }
}

. . .

Thread varT = new PacktThread();

. . .

// This next line is start the Thread by executing
// the run() method.
varT.start();

. . .
```

A second approach is to create a class that implements the `Runnable` interface and passing an instance of the class to the constructor of the `Thread`. Here is an example:

```
. . .
class PacktRunner implements Runnable
{
    . . .
    public void run()
    {
        . . .
    }
}

. . .
```

```
PacktRunner varR = new PacktRunner();
Thread varT = new Thread(varR);

. . .

// This next line is start the Thread by executing the
// run() method.
varT.start();

. . .
```

Both of these methods work equally well, and which one you use is considered to be the developer's choice. Of course, if you are looking for additional flexibility, the second approach is probably a better one to use. You can experiment with both methods to help you make your determination.

CompletableFuture API enhancements

The `CompletableFuture<T>` class is part of the `java.util.concurrent` package. The class extends the `Object` class and implements the `Future<T>` and `CompletionStage<T>` interfaces. This class is used to annotate threads that can be completed. We can use the `CompletableFuture` class to represent a future result. When the `complete` method is used, that future result can be completed.

It is important to realize that if multiple threads attempt to simultaneously complete (finish or cancel), all but one will fail. Let's look at the class and then look at the enhancements.

Class details

The `CompletableFuture<T>` class has one internal class that marks asynchronous tasks:

```
| public static interface  
|   CompletableFuture.AynchronousCompletionTask
```

The constructor for the `CompletableFuture<T>` class has to be in sync with the provided constructor signature, and it takes no arguments. The class has the following methods organized by what they return.

Returns a `CompletionStage`:

- `public CompletableFuture<Void> acceptEither(CompletionStage<? extends T> other, Consumer<? super T> action)`
- `public CompletableFuture<Void> acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action)`
- `public CompletableFuture<Void> acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action, Executor executor)`
- `public <U> CompletableFuture<U> applyToEither(CompletionStage<? extends T> other, Function<? super T, U> fn)`
- `public <U> CompletableFuture<U> applyToEitherAsync(CompletionStage<? extends T> other, Function<? super T, U> fn)`
- `public <U> CompletableFuture<U> applyToEitherAsync(CompletionStage<? extends T> other, Function<? super T, U> fn, Executor executor)`
- `public static <U> CompletedStage<U> completedStage(U value)`
- `public static <U> CompletionStage<U> failedStage(Throwable ex)`
- `public <U> CompletableFuture<U> handle(BiFunction<? super T, Throwable, ? extends U> fn)`
- `public <U> CompletableFuture<U> handleAsync(BiFunction<? super T, Throwable, ? extends U> fn)`
- `public <U> CompletableFuture<U> handleAsync(BiFunction<? super T, Throwable, ? extends U> fn, Executor executor)`
- `public CompletionStage<T> minimalCompletionStage()`
- `public CompletableFuture<Void> runAfterBoth(CompletionStage<?> other, Runnable action)`
- `public CompletableFuture<Void> runAfterBothAsync(CompletionStage<?> other, Runnable action)`

- public CompletableFuture<Void> runAfterBothAsync(CompletionStage<?> other, Runnable action, Executor executor)
- public CompletableFuture<Void> runAfterEither(CompletionStage<?> other, Runnable action)
- public CompletableFuture<Void> runAfterEitherAsync(CompletionStage<?> other, Runnable action)
- public CompletableFuture<Void> runAfterEitherAsync(CompletionStage<?> other, Runnable action, Executor executor)
- public CompletableFuture<T> whenComplete(BiConsumer<? super T, ? super Throwable> action)
- public CompletableFuture<T> whenCompleteAsync(BiConsumer<? super T, ? super Throwable> action)
- public CompletableFuture<T> whenCompleteAsync(BiConsumer<? super T, ? super Throwable> action, Executor executor)

These methods return a CompletionStage:

- public CompletableFuture<Void> thenAccept(Consumer<? super T> action)
- public CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action)
- public CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action, Executor executor)
- public <U> CompletableFuture<Void> thenAcceptBoth(CompletionStage<? extends U> other, BiConsumer<? super T, ? super U> action)
- public <U> CompletableFuture<Void> thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T, ? super U> action)
- public <U> CompletableFuture<Void> thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T, ? super U> action, Executor executor)
- public <U> CompletableFuture<U> thenApply(Function<? super T, ? extends U> fn)
- public <U> CompletableFuture<U> thenApplyAsync(Function<? super T, ? extends U> fn)
- public <U> CompletableFuture<U> thenApplyAsync(Function<? super T, ? extends U> fn, Executor executor)
- public <U, V> CompletableFuture<V> thenCombine(CompletionStage<? extends U> other, BiFunction<? super T, ? super U, ? extends V> fn)
- public <U, V> CompletableFuture<V> thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super T, ? super U, ? extends V> fn)
- public <U, V> CompletableFuture<V> thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super T, ? super U, ? extends V> fn, Executor executor)
- public <U> CompletableFuture<U> thenCompose(Function<? super T, ? extends CompletionStage<U>> fn)

- public <U> CompletableFuture<U> thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn)
- public <U> CompletableFuture<U> thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn, Executor executor)
- public CompletableFuture<Void> thenRun(Runnable action)
- public CompletableFuture<Void> thenRunAsync(Runnable action)
- public CompletableFuture<Void> thenRunAsync(Runnable action, Executor executor)

These methods return a CompletableFuture:

- public static CompletableFuture<Void> allOf(CompletableFuture<?>...cfs)
- public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)
- public CompletableFuture<T> completeAsync(Supplier<? extends T> supplier, Executor executor)
- public CompletableFuture<T> completeAsync(Supplier<? extends T> supplier)
- public static <U> CompletableFuture<U> completedFuture(U value)
- public CompletableFuture<T> completeOnTimeout(T value, long timeout, TimeUnit unit)
- public CompletableFuture<T> copy()
- public CompletableFuture<T> exceptionally(Function<Throwable, ? extends T> fn)
- public static <U> CompletableFuture<U> failedFuture(Throwable ex)
- public <U> CompletableFuture<U> newIncompleteFuture()
- public CompletableFuture<T> orTimeout(long timeout, TimeUnit unit)
- public static CompletableFuture<Void> runAsync(Runnable runnable)
- public static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)
- public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
- public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)
- public CompletableFuture<T> toCompletableFuture()

These methods return a Executor:

- public Executor defaultExecutor()
- public static Executor delayedExecutor(long delay, TimeUnit unit, Executor executor)
- public static Executor delayedExecutor(long delay, TimeUnit unit)

These methods return a boolean:

- public boolean cancel(boolean mayInterruptIfRunning)
- public boolean complete(T value)
- public boolean completeExceptionally(Throwable ex)
- public boolean isCancelled()
- public boolean isCompletedExceptionally()
- public boolean isDone()

No return type:

- public void obtrudeException(Throwable ex)
- public void obtrudeValue(T value)

Additional methods:

- public T get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException
- public T get() throws InterruptedException, ExecutionException
- public T getNow(T valueIfAbsent)
- public int getNumberOfDependents()
- public T join()
- public String toString()

Enhancements

The `CompletableFuture<T>` class received the following enhancements as part of the Java 9 platform:

- Added time-based enhancements:
 - This enables completions based on lapsed time
 - Delayed executions are now also supported
- Significant enhancement to subclasses:
 - Extending `CompletableFuture` is easier
 - Subclasses support alternative default executors

Specifically, the following methods were added in Java 9:

- `newIncompleteFuture()`
- `defaultExecutor()`
- `copy()`
- `minimalCompletionStage()`
- `completeAsync()`
- `orTimeout()`
- `completeOnTimeout()`
- `delayedExecutor()`
- `completedStage()`
- `failedFuture()`
- `failedStage()`

Spin-Wait Hints

With concurrency, we need to ensure that threads waiting to be executed actually get executed. The concept of spin-wait is a process that continually checks for a true condition. The aim of Java Enhancement Proposal 285 was to create an API that permits Java code to issue hints that a spin loop is currently being executed.

While this is not a feature that every Java developer will use, it can be useful for low-level programming. The hint system simply issues hints--indications, and performs no other actions. Justifications for adding these hints include the following assumptions:

- A spin loop's action time can be improved when using a spin hint
- Use of spin hints will reduce thread-to-thread latency
- CPU power consumption will be reduced
- Hardware threads will execute faster

This hint functionality will be contained in a new `onSpinWait()` method as part of the `java.lang.Thread` class. Here is an example of implementing the `onSpinWait()` method:

```
    . . .
    volatile boolean notInReceiptOfEventNotification;

    . . .

    while ( notInReceiptOfEventNotification );
    {
        java.lang.Thread.onSpinWait();
    }

    // Add functionality here to read and process the event
    . . .
```

Summary

In this chapter, we covered concurrency enhancements introduced with the Java 9 platform. We took a deep look at concurrency both as a core Java concept and with an eye to what Java 9 is delivering. We also explored the `Flow` class API that supports reactive programming, a new concept in Java 9. In addition, we explored concurrency enhancements and the new spin-wait hints introduced in Java 9.

In the next chapter, we will highlight the security enhancements introduced in Java 9 along with practical examples.

Security Enhancements

In the last chapter, we covered concurrency enhancements introduced with the Java 9 platform. We took an in-depth look at concurrency both as a core Java concept and as a series of enhancements for Java 9. We also explored the `Flow` class API that supports Reactive Programming, a new concept in Java 9. In addition, we explored concurrency enhancements and the new Spin-Wait hints introduced in Java 9.

In this chapter, we will look at several small changes made to the JDK that involve security. The size of these changes does not reflect their significance. The security enhancements introduced with the Java 9 platform provide developers with a greater ability to write and maintain applications that are more secure than previously possible.

Specifically, we will review the following content areas in this chapter:

- Datagram Transport Layer Security
- Creating PKCS12 keystores
- Improving security application performance
- TLS application-layer protocol negotiation extension
- Leveraging CPU instructions for GHASH and RSA
- OCSP stapling for TLS
- DRBG-based `SecureRandom` implementations

Datagram Transport Layer Security

Datagram Transport Layer Security (DTLS), is a communications protocol. The protocol provides a layer of security for datagram-based applications. DTLS permits secure communications and is based on the **Transport Layer Security (TLS)** protocol. Embedded security helps ensure messages are not forged, tampered with, or eavesdropped.

Let's review the relevant terminology:

- **Communication protocol:** A set of rules that govern how information is transmitted.
- **Datagram:** A structured transfer unit.
- **Eavesdropping:** Undetected listening to in-transit data packets.
- **Forgery:** Transmission of a packet with falsified sender.
- **Network packet:** A formatted unit of data for transmission.
- **Tampering:** The altering of data packets after the sender transmits them and before the intended receiver receives them.
- **TLS protocol:** The most common network security protocol. Uses, as an example, IMPA and POP for email.

The DTLS Java Enhancement Proposal 219 is aimed at creating an API for the DTLS versions 1.0 and 1.2.

In the sections that follow, we will look at each of the DTLS versions, 1.0 and 1.2, and then review the changes to the Java 9 platform.

DTLS protocol version 1.0

DTLS protocol version 1.0 was established in 2006 and provides communications security for datagram protocols. Here are the basic characteristics:

- Permits client/server applications to communicate without permitting:
 - Eavesdropping
 - Tampering
 - Message forgery
- Based on the TLS protocol
- Provides security guarantees
- The DLS protocol's datagram semantics are preserved

The following diagram illustrates where the **Transport Layer** fits into the overall schema of SSL/TLS protocol layers and protocols for each layer:



DTLS protocol version 1.0 provides detailed specifications with the major areas of coverage listed as follows:

- Ciphers:
 - Anti-replay block cipher
 - New cipher suites
 - Standard (or null) stream cipher
- Denial of service countermeasures
- Handshake:
 - Message format
 - Protocol
 - Reliability
- Messages:
 - Fragmentation and reassembly

- Loss-insensitive messaging
- Size
- Timeout and retransmission
- Packet loss

- **Path Maximum Transition Unit (PMTU) discovery**

- Record layer
- Record payload protection
- Reordering
- Replay detection
- Transport layer mapping

DTLS protocol version 1.2

DTLS protocol version 1.2 was published in January 2012 and is copyrighted by the **Internet Engineering Task Force (IETF)**. This section shares code samples that illustrate the changes in version 1.2.

The following code illustrates the TLS 1.2 handshake message header. This format supports:

- Message fragmentation
- Message loss
- Reordering

```
// Copyright (c) 2012 IETF Trust and the persons identified as
// authors of the code. All rights reserved.

struct
{
    HandshakeType msg_type;
    uint24 length;
    uint16 message_seq;                                // New field
    uint24 fragment_offset;                            // New field
    uint24 fragment_length;                           // New field
    select (HandshakeType)
    {
        case hello_request: HelloRequest;
        case client_hello: ClientHello;
        case hello_verify_request: HelloVerifyRequest; // New type
        case server_hello: ServerHello;
        case certificate:Certificate;
        case server_key_exchange: ServerKeyExchange;
        case certificate_request: CertificateRequest;
        case server_hello_done:ServerHelloDone;
        case certificate_verify: CertificateVerify;
        case client_key_exchange: ClientKeyExchange;
        case finished: Finished;
    } body;
} Handshake;
```



The code presented in this section is from the DTLS protocol documentation and is republished here in accordance with IETF's Legal Provisions Relating to IETF Documents.

The record layer contains the information that we intend to send into records. The information starts off inside a `DTLSPlaintext` structure and then, after the handshake takes place, the records are encrypted and are eligible to be sent by the

communication stream. The record layer format follows with new fields in version 1.2 annotated with the // New field in-code comments as follows:

```
// Copyright (c) 2012 IETF Trust and the persons identified
// as authors of the code. All rights reserved.

struct
{
    ContentType type;
    ProtocolVersion version;
    uint16 epoch;                                // New field
    uint48 sequence_number;                        // New field
    uint16 length;
    opaque fragment[DTLSPlaintext.length];
} DTLSPlaintext;

struct
{
    ContentType type;
    ProtocolVersion version;
    uint16 epoch;                                // New field
    uint48 sequence_number;                        // New field
    uint16 length;
    opaque fragment[DTLSCompressed.length];
} DTLSCompressed;

struct
{
    ContentType type;
    ProtocolVersion version;
    uint16 epoch;                                // New field
    uint48 sequence_number;                        // New field
    uint16 length;
    select (CipherSpec.cipher_type)
    {
        case block: GenericBlockCipher;
        case aead: GenericAEADCipher;           // New field
    } fragment;
} DTLSCiphertext;
```

Finally, here is the updated handshake protocol:

```
// Copyright (c) 2012 IETF Trust and the persons identified
// as authors of the code. All rights reserved.

enum {
    hello_request(0), client_hello(1),
    server_hello(2),
    hello_verify_request(3),                                // New field
    certificate(11), server_key_exchange (12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    finished(20), (255) } HandshakeType;

struct {
    HandshakeType msg_type;
    uint24 length;
```

```
uint16 message_seq;                                // New field
uint24 fragment_offset;                            // New field
uint24 fragment_length;                           // New field
select (HandshakeType) {
    case hello_request: HelloRequest;
    case client_hello: ClientHello;
    case server_hello: ServerHello;
    case hello_verify_request: HelloVerifyRequest; // New field
    case certificate: Certificate;
    case server_key_exchange: ServerKeyExchange;
    case certificate_request: CertificateRequest;
    case server_hello_done: ServerHelloDone;
    case certificate_verify: CertificateVerify;
    case client_key_exchange: ClientKeyExchange;
    case finished: Finished;
} body; } Handshake;

struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    opaque cookie<0..2^8-1>;                      // New field
    CipherSuite cipher_suites<2..2^16-1>;
    CompressionMethod compression_methods<1..2^8-1>; } ClientHello;

struct {
    ProtocolVersion server_version;
    opaque cookie<0..2^8-1>; } HelloVerifyRequest;
```

DTLS support in Java 9

Java 9's implementation of the DTLS API is transport-independent and light-weight. The design considerations for the API were as follows:

- Read timeouts will not be managed
- The implementation will use a single TLS record for each wrap/unwrap operation
- The application, not the API, will be required to:
 - Determine timeout values
 - Assemble out-of-order application data

The DTLS is a protocol used to secure data from the application layer before that data is passed to a transport layer protocol. DTLS is a good solution for encrypting and transmitting real-time data. Caution should be exercised so that we do not introduce vulnerabilities in our application's implementation. Here are security considerations specific to implementing DTLS in your Java 9 applications:

- Implement DTLS v1.2, since that is the latest version supported by Java 9.
- Avoid **Rivest-Shamir-Adleman (RSA)** encryption. If RSA must be used, add additional security to your private keys since this is a weak point for RSA.
- Use 192 bits or more when using the **Elliptic Curve Diffie-Hellman (ECDH)** anonymous key agreement protocol. The 192-bit value is based on a **National Institute of Standards and Technology (NIST)** recommendation.
- The use of **Authenticated Encryption with Associated Data (AEAD)**, a form of encryption, is highly recommended. AEAD provides authenticity, confidentiality, and integrity assurances on the data being encrypted and decrypted.
- Always implement the `renegotiation_info` extension when implementing handshake renegotiation.
- Establish a **Forward Secrecy (FS)** capability in all Java applications using a communication protocol. Implementing FS ensures past session encryption keys are not compromised when long-term encryption keys are compromised. Ideally a **Perfect Forward Secrecy (PFS)**, where each key is only valid for a single session, would be used in the Java applications that call for the greatest security of transmitted data.

Creating PKCS12 keystores

The Java 9 platform provides increased security for keystores. In order to appreciate the changes ushered in by Java Enhancement Proposal 229, create PKCS12 keystores by default, we will first review the concept of keystores, look at the `KeyStore` class, and then look at the changes.

Keystore primer

The concept of a `KeyStore` is relatively simple. It is essentially a database file, or data repository file, that stores public key certificates and private keys. The `Keystore` will be stored in the `/jre/lib/security/cacerts` folder. As you will see in the next section, this database is managed by Java's `java.security.KeyStore` class methods.

`KeyStore` features include:

- Contains one of the following entry types:
 - Private keys
 - Public key certificates
- Unique alias string names for every entry
- Password protection for each key

Java Keystore (JKS)

The `java.security.KeyStore` class is the storage facility for cryptographic keys and certificates. This class extends `java.lang.Object`, see as follows:

```
| public class KeyStore extends Object
```

There are three types of entries managed by a `KeyStore`, each implements the `KeyStore.Entry` interface, one of the three interfaces provided by the `KeyStore` class. The Entry implementations are defined in the following table:

Implementation	Description
<code>KeyStore.PrivateKeyEntry</code>	<ul style="list-style-type: none">Contains the <code>PrivateKey</code> and can store it in a protected formatContains the certificate chain for the public key
<code>KeyStore.SecretKeyEntry</code>	<ul style="list-style-type: none">Contains the <code>SecretKey</code> and can store it in a protected format
<code>KeyStore.TrustedCertificateEntry</code>	<ul style="list-style-type: none">Contains a single public key <code>Certificate</code> from an external source

This class has been part of the Java platform since version 1.2. It has one constructor, three interfaces, six sub-classes, and several methods. The constructor definition is:

```
| protected KeyStore(KeyStoreSpi keyStoresSpi,  
| Provider provider, String type)
```

The `Keystore` class contains the following interfaces:

- `public static interface KeyStore.Entry`:

- This interface serves as a marker for `KeyStore` entry types and contains no methods.
- `public static interface KeyStore.LoadStoreParameter:`
 - This interface serves as a marker for load and store parameters and has the following method that returns null or the parameter used to protect the `KeyStore` data:
 - `getProtectionParameter()`
- `public static interface KeyStore.ProtectionParameter:`
 - This interface serves as a marker for `KeyStore` protection parameters and contains no methods.

The `java.security.KeyStore` class also contains the six nested classes listed as follows.

Builder

The `KeyStore.Builder` class is used when you want to defer the instantiation of a `KeyStore`:

```
|     public abstract static class KeyStore.Builder extends Object
```

This class provides the necessary information for instantiating a `KeyStore` object. The class has the following methods:

- `public abstract KeyStore getKeyStore() throws KeyStoreException`
- `public abstractKeyStore.ProtectionParameter getProjectionParameter(String alias) throws KeyStoreException`
- Three options for `newInstance`:
 - `public static KeyStore.Builder newInstance(KeyStore keyStore, KeyStore.ProtectionParameter protectionParameter)`
 - `public static KeyStore.Builder newInstance(String type, Provider provider, File file, KeyStore.ProtectionParameter protection)`
 - `public static KeyStore.Builder newInstance(String type, Provider provider, KeyStore.ProtectionParameter protection)`

The CallbackHandlerProtection class

The `KeyStore.CallbackHandlerProtection` class definition is as follows:

```
| public static class KeyStore.CallbackHandlerProtection extends  
|   Object implements KeyStore.ProtectionParameter
```

This class provides a `ProtectionParameter` to encapsulate a `CallbackHandler` and has the following method:

```
|   public CallbackHandler getCallbackHandler()
```

The PasswordProtection class

The `KeyStore.PasswordProtection` class definition is as follows:

```
| public static class KeyStore.PasswordProtection extends Object  
|     implements KeyStore.ProtectionParameter, Destroyable
```

This call provides an implementation of `ProtectionParameter` that is password-based. The class has the following methods:

- `public void destroy() throws DestroyFailedException:`
 - This method clears the password
- `public char[] getPassword():`
 - Returns a reference to the password
- `public boolean isDestroyed():`
 - Returns true if the password was cleared

The PrivateKeyEntry class

The `KeyStore.PrivateKeyEntry` class definition is as follows:

```
| public static final class KeyStore.PrivateKeyEntry extends  
|   Object implements KeyStore.Entry
```

This creates an entry to hold a `PrivateKey` and the corresponding `Certificate` chain. This class has the following methods:

- `public Certificate getCertificate():`
 - Returns the **end entity** `Certificate` from the `Certificate` **chain**
- `public Certificate[] getCertificateChain():`
 - Returns the `Certificate` **chain** as an array of `Certificates`
- `public PrivateKey getPrivateKey():`
 - Returns the `PrivateKey` from the current entry
- `public String toString():`
 - Returns the `PrivateKeyEntry` as a `String`

The SecretKeyEntry class

The `KeyStore.SecretKeyEntry` class definition is as follows:

```
| public static final class KeyStore.SecretKeyEntry extends  
|   Object implements KeyStore.Entry
```

This class holds a `SecretKey` and has the following methods:

- `public SecretKey getSecretKey():`
 - Returns the entry's `SecretKey`
- `public String toString():`
 - Returns the `SecretKeyEntry` as a `String`.

The TrustedCertificateEntry class

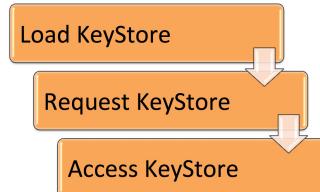
The `KeyStore.TrustedCertificateEntry` class definition is as follows:

```
| public static final class KeyStore.TrustedCertificateEntry extends  
|   Object implements KeyStore.Entry
```

This class holds a trusted `Certificate` and has the following methods:

- `public Certificate getTrustedCertificate():`
 - Returns the entry's trusted `Certificate`
- `public String toString():`
 - Returns the entry's trusted `Certificate` as a `String`

The key to using this class is understanding the flow. First, we must load the `KeyStore`, using the `getInstance` method. Next we request access to the `KeyStore` instance. Then, we have access to read and write to the `Object`:



The following code snippet shows the load-request-access implementation:

```
. . .  
try {  
    // KeyStore implementation will be returned for the default type  
    KeyStore myKS = KeyStore.getInstance(KeyStore.getDefaultType());  
  
    // Load  
    myKS.load(null, null);  
  
    // Instantiate a KeyStore that holds a trusted certificate  
    TrustedCertificateEntry myCertEntry =  
        new TrustedCertificateEntry(generateCertificate());  
  
    // Assigns the trusted certificate to the "pack.pub" alias  
    myKS.setCertificateEntry("packt.pub",  
        myCertEntry.getTrustedCertificate());  
  
    return myKS;  
}  
catch (Exception e) {
```

```
        throw new AssertionError(e);
    }
    . . .
```

PKCS12 default in Java 9

Prior to Java 9, the default `KeyStore` type was **Java KeyStore (JKS)**. The Java 9 platform now uses PKCS as the default `KeyStore` type, more specifically, PKCS12.



*PKCS is the acronym for **Public Key Cryptography Standards**.*

This change to PKCS provides stronger cryptographic algorithms as compared to JKS. As you would expect, JDK 9 will still be compatible with JKS to support previously developed systems.

Improving security application performance

Java Enhancement Proposal 232, titled *Improving Security Application Performance*, was focused on performance improvements when running applications with a security manager installed. Security managers can result in processing overhead and less than ideal application performance.

This is an impressive undertaking as current CPU overhead when running security managers is estimated to result in 10-15% performance degradation. It is not feasible to completely remove the CPU overhead as some CPU processing is required to run the security manager. That being said, the intention of this proposal (JEP-232) was to decrease the overhead percentage as much as possible.

This effort resulted in the following optimizations, each detailed in subsequent sections:

- Security policy enforcement
- Permission evaluation
- Hash code
- Package checking algorithm

Security policy enforcement

JDK 9 uses `ConcurrentHashMap` for mapping `ProtectionDomain` to `PermissionCollection`. `ConcurrentHashMap` is typically used for high concurrency in applications. It has the following characteristics:

- Thread safe
- Enter map does not need to be synchronized
- Fast reads
- Writes use locks
- No object-level locking
- Locking at a very granular level

The `ConcurrentHashMap` class definition follows:

```
| public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>
|   implements ConcurrentMap<K, V>, Serializable
```

In the preceding class definition, `K` refers to the type of keys maintained by the hash map and `V` indicates the type of mapped values. There is a `KeySetView` sub-class and several methods.

There are three additional classes related to enforcing security policy--

`ProtectionDomain`, `PermissionCollection`, and `SecureClassLoader`:

- The `ProtectionDomain` class is used to encapsulate a group of classes so that permissions can be granted to the domain.
- The `PermissionCollection` class represents a collection of permission objects.
- The `SecureClassLoader` class, which extends the `ClassLoader` class, provides additional functionality for defining classes with permissions for retrieval by the system policy. In Java 9, this class uses `ConcurrentHashMap` for increased security.

Permission evaluation

Under the category of permission evaluation, three optimizations were made:

- The `identifyPolicyEntries` list previously had policy provider code for synchronization. This code has been removed in JDK 9.
- `PermissionCollection` entries are now stored in a `ConcurrentHashMap`. They were previously stored as a `HashMap` in a `Permission` class.
- Permissions are now stored in concurrent collections in subclasses of `PermissionCollection`.

The `java.Security.CodeSource` package

A hash code is an object-generated number that is stored in a hash table for rapid storage and retrieval. Every object in Java has a hash code. Here are some characteristics and rules for hash codes:

- Hash codes are the same for equal objects within a running process
- Hash codes can change between execution cycles
- Hash codes should not be used as keys

The Java 9 platform includes a modified `hashCode` method of `java.security.CodeSource` to optimize DNS lookups. These can be processor intensive, so a String version of the code source URL is used to compute hash codes.

The `CodeSource` class definition follows:

```
| public class CodeSource extends Object implements Serializable
```

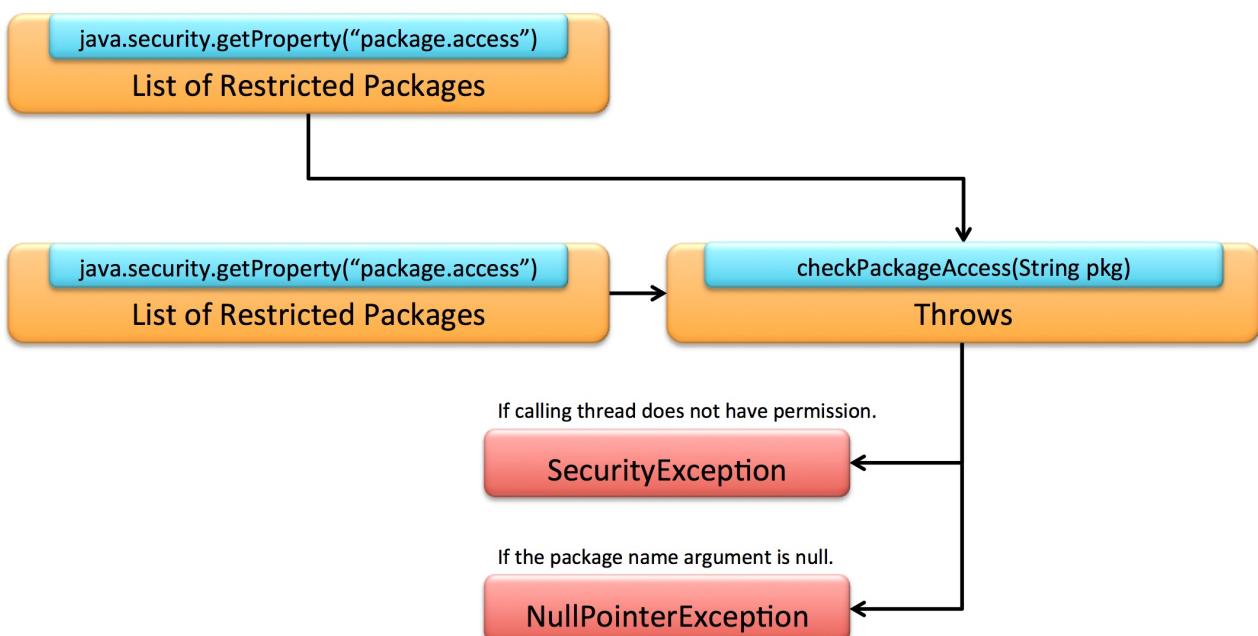
This class has the following methods:

- `public boolean equals(Object obj)`: Returns `true` if the objects are equal. This overrides the `equals` method in the `Object` class.
- `public final Certificate[] getCertificates()`: Returns an array of certificates.
- `public final CodeSigner[] getCodeSigners()`: Returns an array of the code signers associated with the `CodeSource`.
- `public final URL getLocation()`: Returns the URL.
- `public int hashCode()`: Returns the hash code value for the current object.
- `public boolean implies(CodeSource codesource)`: Returns true if the given code source meets the following criteria:
 - is not null
 - object's certificates are not null
 - object's location is not null
- `public String toString()`: Returns a `String` with information about the `CodeSource` to include the location and certificates.

Package checking algorithm

Java 9's final performance improvement when running applications with a security manager installed, came in the form of the `java.lang.SecurityManager` package enhancements. Specifically, the `checkPackageAccess` method's package checking algorithm was modified.

The `java.lang.SecurityManager` class allows applications to implement security policy on specific operations. The `public void checkPackageAccess(String pkg)` method, of this class receives a comma-delimited list of restricted packages from the `getProperty` method. As illustrated here, depending on the evaluation, the `checkPackageAccess` method can throw one of two exceptions:

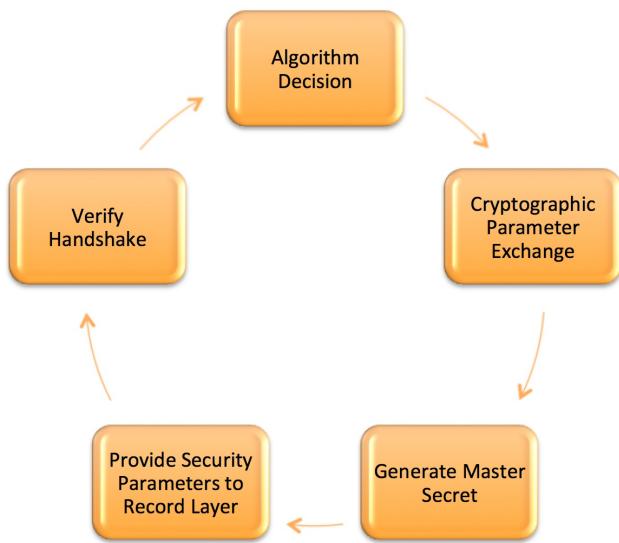


TLS application-layer protocol negotiation extension

Java Enhancement Proposal 244, simply enhanced the `javax.net.ssl` package so that it supports the **Transport Layer Security (TLS) ALPN (Application Layer Protocol Negotiation)** extension. This extension permits application protocol negotiation for TLS connections.

TLS ALPN extension

The ALPN is a TLS extension and can be used to negotiate which protocol to implement when using a secure connection. ALPN represents an efficient means of negotiating protocols. As indicated in the following diagram, there are five basic steps to TLS handshakes:



The javax.net.ssl package

The `java.net.ssl` package contains classes relating to secure socket packages. This permits us to use SSL as an example, for the reliable detection of errors introduced to the network byte stream. It also provides the ability to encrypt the data as well as provide authentication of client and server.

This package includes the following interfaces:

- public interface HandshakeCompletedListener extends EventListener
- public interface HostnameVerifier
- public interface KeyManager
- public interface ManagerFactoryParameters
- public interface SSLSession
- public interface SSLSessionBindingListener extends EventListener
- public interface SSLSessionContext
- public interface TrustManager
- public interface X509KeyManager extends KeyManager
- public interface X509TrustManager extends TrustManager

The `java.net.ssl` package also has the following sub-classes:

- public class CertPathTrustManagerParameters extends Object implements ManagerFactoryParameters
- public abstract class ExtendedSSLSession extends Object implements SSLSession
- public class HandshakeCompleteEvent extends EventObject
- public abstract classHttpsURLConnection extends HttpURLConnection
- public class KeyManagerFactory extends Object
- public abstract class KeyManagerFactorySpi
- public class KeyStoreBuilderParameters extends Object implements ManagerFactoryParameters
- public class SSLContext extends Object
- public abstract class SSLContextSpi extends Object
- public abstract class SSLEngine extends Object
- public class SSLEngineResult extends Object
- public class SSLParameters extends Object
- public final class SSLPermission extends BasicPermission
- public abstract class SSLServerSocket extends ServerSocket

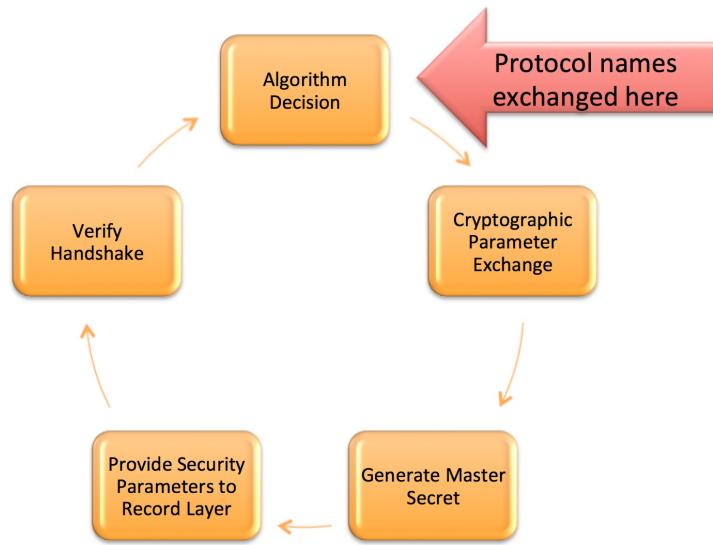
- public abstract class SSLSocketFactory extends ServerSocketFactory
- public class SSLSessionBindingEvent extends EventObject
- public abstract class SSLSocket extends Socket
- public abstract class SSLSocketFactory extends SocketFactory
- public class TrustManagerFactory extends Object
- public abstract class TrustManagerFactorySpi extends Object
- public abstract class X509ExtendedKeyManager extends Object implements X509KeyManager
- public abstract class X509ExtendedTrustManager extends Object implements X509TrustManager

The `java.net.ssl` package extension

The change to the `java.net.ssl` package in the Java 9 platform is that it now supports the TLS ALPN extension. Key benefits of this change are:

- TLS clients and servers can now use multiple application-layer protocols, which may or may not use the same transport-layer port
- The ALPN extension permits clients to prioritize application-layer protocols it supports
- Servers can select a client protocol and for the TLS connection
- Supports HTTP/2

The following illustration was previously presented as the five basic steps to TLS handshakes. Updated for Java 9 and presented here, the illustration indicates where the protocol names are shared between the client and server:



Once the client's list of application layer protocols is received, the server can select the server's preferred intersection value and externally scan initial plain text `clientHello`s and select an ALPN protocol. An application server will do one of the following:

- Select any of the supported protocols
- Decide that the ALPN values (remotely offered and locally supported) are mutually exclusive

- Ignore the ALPN extension

Other key behaviors with regards to the ALPN extension:

- The server can alter connection parameters
- After the SSL/TLS handshake starts, the application can query to see if an ALPN value has been selected yet
- After the SSL/TLS handshake ends, the application can review which protocol was used

A `ClientHello` is the first message in the TLS handshake. It has the following structure:

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-1>;
    CompressionMethod compression_methods<1..2^8-1>;
    Extension extensions<0..2^16-1>;
} ClientHello;
```

Leveraging CPU instructions for GHASH and RSA

The self-descriptive title of Java Enhancement Proposal (JEP) 246, **Leverage CPU Instructions for GHASH and RSA**, provides great insight into its goal. The point of this JEP was to improve the performance of cryptographic operations, specifically GHASH and RSA. The performance improvement has been achieved in Java 9 by leveraging the newest SPARC and Intel x64 CPU instructions.

This enhancement did not require new or modified APIs as part of the Java 9 platform.

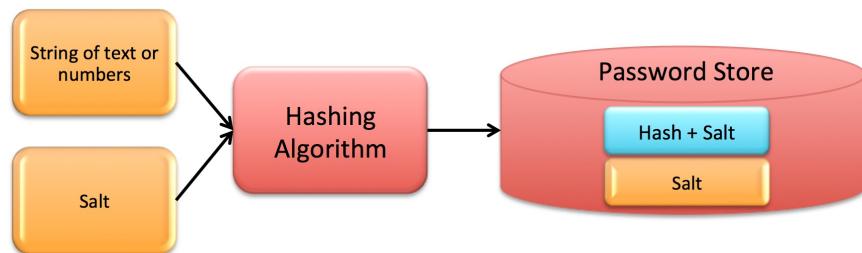
Hashing

Galois HASH (GHASH) and **Rivest-Shamir-Adleman (RSA)** are crypto systems hashing algorithms. Hashes are a fixed length string or number generated from a string of text. Algorithms, specifically hashing algorithms, are devised so that the resultant hashes cannot be reverse engineered. We use hashing to store passwords that are generated with a salt.

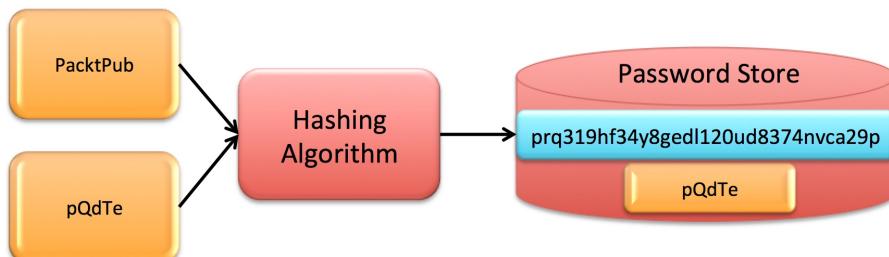


Salts, in cryptology, are random data used as an input to a hashing function to generate a password. Salts help protect against rainbow table attacks and dictionary attacks.

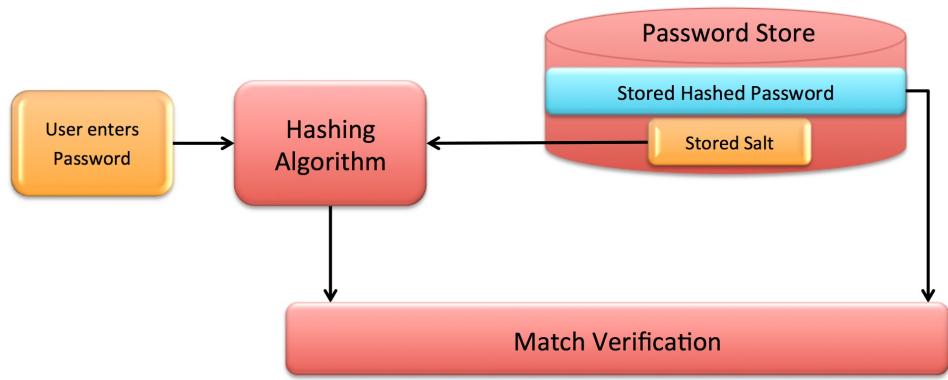
The following graphic illustrates the basics of how hashing works:



As you can see, the hashing algorithm is fed plain text and a salt resulting in a new hashed password and the salt being stored. Here is the same graphic with sample input/output to demonstrate the functionality:



The validation process, the following diagram starts with the user entering their plain text password. The hashing algorithm takes that plain text and rehashes it with the stored salt. Then the resulting hashed password is compared to the stored one:



OCSP stapling for TLS

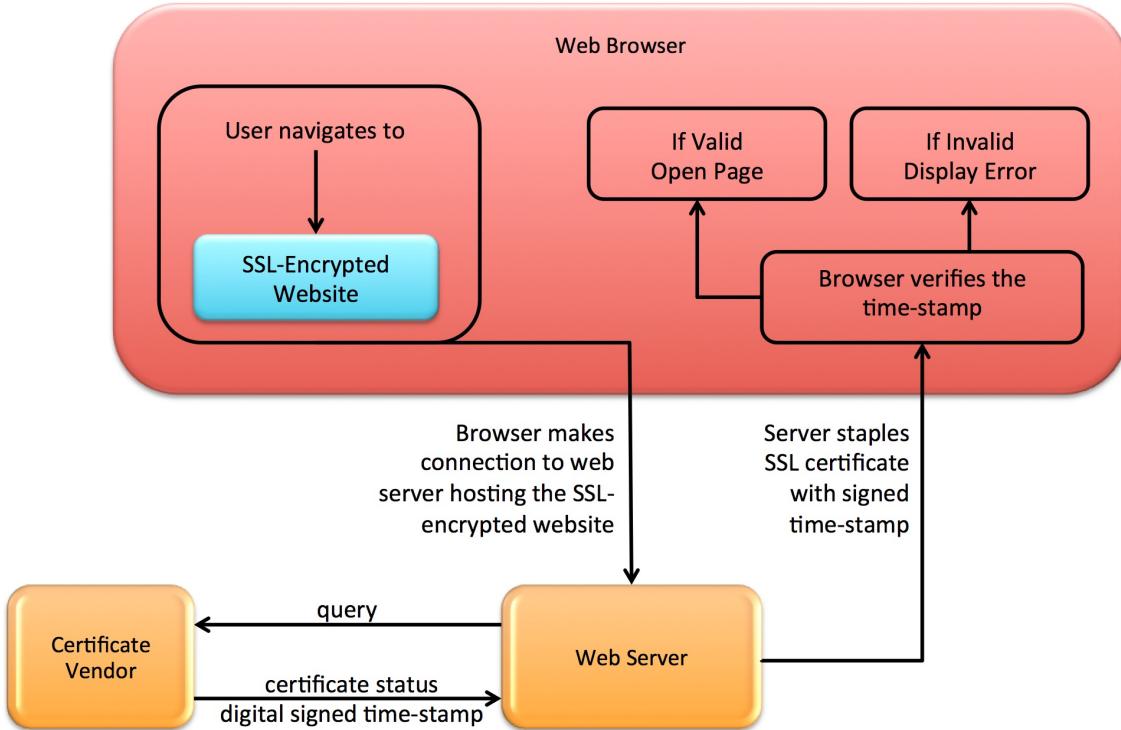
Online Certificate Status Protocol (OCSP) stapling is a method of checking the revocation status of digital certificates. The OCSP stapling approach for determining an SSL certificate's validity is assessed as being both safe and quick. The determination speed is achieved by permitting web servers to provide the validity information on its organic certificates instead of the lengthier process of requesting validating information from the certificate's issuing vendor.



Online Certificate Status Protocol (OCSP) stapling was previously referred to as the Transport Layer Security (TLS) certificate status request extension.

OCSP stapling primer

THE OCSP stapling process involves several components and validity checks. The following graphic illustrates the OCSP stapling process:



As you can see, the process starts when the user attempts to open an SSL-encrypted website via their browser. The browser queries the web server to ensure the SSL-encrypted website has a valid certificate. The web server queries the certificate's vendor and is provided with both the certificate status and the digital signed time-stamp. The web server takes those two components (certificate status and digital signed time-stamp), staples them together, and returns the stapled set to the requesting browser. The browser can then check the validity of the time-stamp and decide whether to display the SSL-encrypted website or to display an error.

Changes for the Java 9 platform

Java Enhancement Proposal 249, **OCSP Stapling for TLS**, implements OCSP stapling via the TLS certificate status request extension. OCSP stapling checks the validity of X.509 certificates.



X.509 certificates are digital certificates that use the X509 Public Key Infrastructure (PKI).

Prior to Java 9, the certificate validity check (really, the check to see if the certificate has been revoked) can be enabled on the client side and has the following inefficiencies:

- OCSP responder performance bottlenecks
- Performance degradation based on multiple passes
- Additional performance degradation if OCSP checking is performed client side
- False **fails** when browsers do not connect to an OCSP responder
- Susceptibility of denial of service attacks on OCSP responders

The new OCSP stapling for TLS includes the following system property changes for the Java 9 platform:

- `jdk.tls.client.enableStatusRequestExtension`:
 - Default setting: true
 - Enables `status_request` extension
 - Enables `status_request_v2` extension
 - Enables processing `CertificateStatus` messages from server
- `jdk.tls.server.enableStatusRequestExtension`:
 - Default setting: false
 - Enables OCSP stapling support server-side
- `jdk.tls.stapling.responseTimeout`:
 - Default setting: 5000 milliseconds
 - Controls maximum time allocated by server to obtain OCSP responses
- `jdk.tls.stapling.cacheSize`:
 - Default setting: 256
 - Controls maximum number of cache entries

- Can set maximum to zero eliminates ceiling
- `jdk.tls.stapling.cacheLifetime`:
 - Default setting: 3600 seconds (1 hour)
 - Controls maximum lifetime of a cached response
 - Can set value to zero in order to disable cache lifetime
- `jdk.tls.stapling.responderURI`:
 - Default setting: none
 - Can set a default URI for certificates without the **Authority Info Access (AIA)** extension
 - Does not override the AIA extension unless `jdk.tls.stapling.Override` property is set
- `jdk.tls.stapling.responderOverride`:
 - Default setting: false
 - Allows a `jdk.tls.stapling.responderURI` provided property to override AIA extension values
- `jdk.tls.stapling.ignoreExtensions`:
 - Default setting: false
 - Disables OCSP extension forwarding as specified in `status_request` or `status_request_v2` TLS extensions.

The `status_request` and `status_request_v2` TLS hello extensions are now supported by both client and server-side Java implementations.

DRBG-based SecureRandom implementations

Prior to Java 9, the JDK had two approaches to generating secure random numbers. One method, written in Java, used SHA1-based random number generation and was not terribly strong. The other method was platform-dependent and used preconfigured libraries.

Deterministic Random Bit Generator (DRBG) is a method for generating random numbers. It has been approved by the **National Institute of Standards and Technology (NIST)**, a branch of the U.S. Department of Commerce. DRBG methodologies include modern and stronger algorithms for generating secure random numbers.

Java Enhancement Proposal 273, **DRBG-Based SecureRandom Implementations** aimed to implement three specific DRBG mechanisms. These mechanisms are listed as follows:

- Hash_DRBG
- HMAC_DRBG
- CTR_DRBG



You can learn specifics about each of the DRBG mechanisms at <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf>

Here are the three new APIs:

- `SecureRandom`: New methods allowing the configuration of `SecureRandom` objects with the below listed configurable properties:
 - seeding
 - reseeding
 - random-bit-generation
- `SecureRandomSpi`: new methods to implement the `SecureRandom` methods
- `SecureRandomParameter`: new interface so input can be passed to the new `SecureRandom` methods

Summary

In this chapter, we looked at several small, but significant changes to the JDK that involve security. The featured security enhancements that are part of the Java 9 platform provide developers with the distinct ability to write and maintain applications that implement security. Specifically, we covered DTLS, keystores, improving security application performance, the TLS application-layer protocol negotiation extension, leveraging CPU instructions for GHASH and RSA, OCSP stapling for TLS, and DRBG-based `SecureRandom` implementations.

In the next chapter we will explore the new command-line flags used in Java 9 as well as changes to various command-line tools. Our coverage will include managing the Java JVM run-time and compiler using the new command-line options and flags.

Command Line Flags

In the previous chapter, we looked at several security changes to the JDK. Java 9's security enhancements provide developers with the ability to write and maintain applications that implement security. Specifically, we covered datagram transport layer security, Keystores, improving security application performance, the TLS application-layer protocol negotiation extension, leveraging CPU instructions for GHASH and RSA, OCSP stapling for TLS, and DRBG-based `SecureRandom` implementations.

In this chapter, we will explore several changes to the Java 9 platform with the common theme of command-line flags. Specifically, we will cover the following concepts:

- Unified JVM logging
- Compiler control
- Diagnostic commands
- Heap profiling agent
- Removing your JHAT
- Command-line flag argument validation
- Compiling for older platform versions

Unified JVM Logging [JEP 158]

Creating a unified logging schema for the JVM was the central goal of JEP-158. Here is a comprehensive list of the goals of the JEP:

- Create a JVM-wide set of command-line options for all logging operations
- Use categorized tags for logging
- Permit messages to have multiple tags, also referred to as tag sets
- Provide six levels of logging:
 - Error
 - Warning
 - Information
 - Debug
 - Trace
 - Develop
- Select which messages are logged based on levels
- Optionally direct logging to the console or a file
 - Print one line at a time and do not support interleaving within the same line
- Permit output of multiple line logs (non-interleaved)
- Format all logging messages so that they are easily human-read
- Add decorations such as uptime, level, and tags
- Like levels, select which messages are logged based on decorations
- Convert pre-Java 9 `tty>print` logging to use unified logging as the output
- Permit dynamic message configuration using `jcmd` and `MBeans`
- Permit the ability to enable and disable individual log messages
- Add ability to determine the order in which decorations are printed

The unified logging changes to the JVM can be grouped into the five categories listed here:

- Command-line options
- Decorations
- Levels
- Output
- Tags

Let's briefly look at each of these categories.

Command-line options

The new command-line option, `-Xlog`, was added to the logging framework in Java 9. This command-line option has an extensive array of parameters and possibilities. The basic syntax is simply `-Xlog` followed by an option. Here is the formal basic syntax:

```
| -Xlog[:option]
```

Here is a basic example with the `all` option:

```
| -Xlog:all
```

Here is the extensive command-line syntax used to configure the new unified logging:

```
-Xlog[:option]

option          := [<what>] [: [<output>] [: [<decorators>] [:<output-options>]]]
                  'help'
                  'disable'

what           := <selector>[,...]

selector        := <tag-set>[*] [=level]
tag-set         := <tag>[+..]
                  'all'

tag             := name of tag

level           := trace
                  debug
                  info
                  warning
                  error

output          := 'stderr'
                  'stdout'
                  [file=>]<filename>

decorators       := <decorator>[,...]
                  'none'

decorator        := time
                  uptime
                  timemillis
                  uptimemillis
                  timenanos
                  uptimenanos
```

```
pid
tid
level
tags

output-options  := <output_option>[ , . . . ]

output-option   := filecount=<file count>
                  filesize=<file size in kb>
                  parameter=value
```

The following `-Xlog` examples are followed by a description:

```
| -Xlog:all
```

In the preceding example, we are telling the JVM to take the following actions:

- Log all messages
- Use the `info` level
- Provide output to `stdout`

With this example, all `warning` messages will still be output to `stderr`.



The next example, shown here, logs messages at the `debug` level:

```
| -Xlog:gc+rt*=debug
```

In the preceding example, we are telling the JVM to take the following actions:

- Log all messages tagged with, at a minimum, the `gc` and `rt` tags
- Use the `debug` level
- Provide output to `stdout`

The next example pushes the output to an external file:

```
| -Xlog:disable - Xlog:rt=debug:rtdebug.txt
```

In the preceding example, we are telling the JVM to take the following actions:

- Disable all messages except those tagged with `rt` tags
- Use the `debug` level
- Provide output to a file named `rtdebug.txt`

Decorations

In the context of Java 9's logging framework, decorations are metadata about the log message. Here is the alphabetic list of decorations that are available:

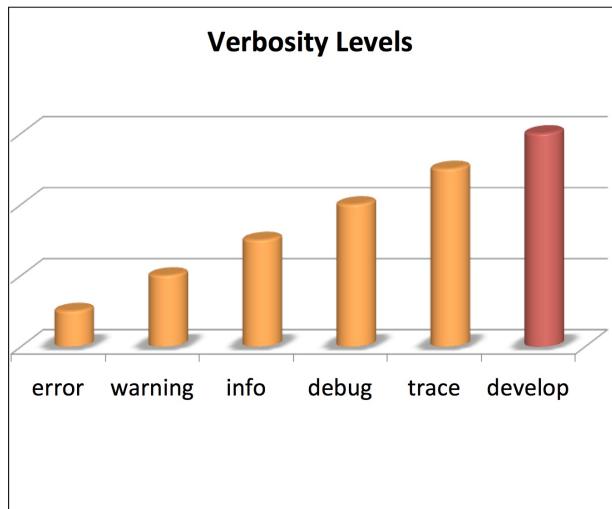
- **level**: The level associated with the logged message
- **pid**: PID = Processor IDentifier
- **tags**: The tag-set associated with the logged message
- **tid**: TID = Thread IDentifier
- **time**: Refers to current date and time using ISO-8601 format
- **timemillis**: Current time in milliseconds
- **timenanos**: Current time in nanoseconds
- **uptime**: Time, in seconds and milliseconds, since the JVM started
- **uptimemillis**: Time, in milliseconds, since the JVM started
- **uptimenanos**: Time, in nanoseconds, since the JVM started

Decorations can be surpassed or included in unified logging output. Regardless of which decorations are used, they will appear in the output in the following order:

1. time
2. uptime
3. timemillis
4. uptimemillis
5. timenanos
6. uptimenanos
7. pid
8. tid
9. level
10. tags

Levels

Logged messages are individually associated with a verbosity level. As previously listed, the levels are **error**, **warning**, **information**, **debug**, **trace**, and **develop**. The following chart shows how the levels have an increasing level of verbosity in respect to how much information is logged. The **develop** level is for development purposes only and is not available in on-product application builds:



Output

The Java 9 logging framework supports three types of output with examples of direct use with the `-Xlog` command-line syntax:

In the following example, we provide output to `stderr`:

```
| -Xlog:all=warning:stderr:none
```

The following example provides output to `stdout`:

```
| -Xlog:all=warning:stdout:none
```

The following example writes the output to a text file:

```
| -Xlog:all=warning:file=logmessages.txt:none
```

Tags

The new logging framework consists of a set of tags identified in the JVM. These tags can be changed in source code if needed. The tags should be self-identifying, such as `gc` for garbage collection.

When more than one tag is grouped together, they form a tag-set. When we add our own tags via source code, each tag should be associated with a tag-set. This will help ensure the tags stay organized and easily human-readable.

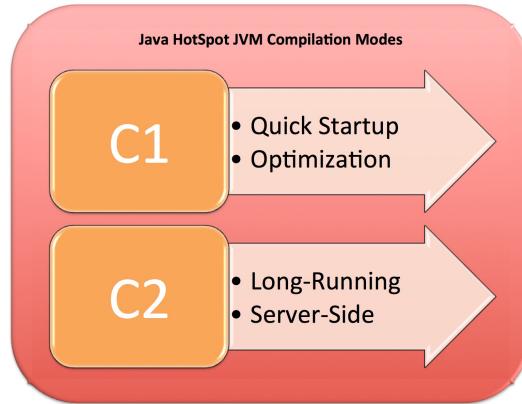
Compiler control [JEP 165]

Controlling Java Virtual Machine compilers might seem like an unnecessary task, but for many developers, this is an important aspect of testing. Java Enhancement Proposal 165 detailed a plan to implement runtime management of JVM compilers. This is accomplished with method-dependent compiler flags.

In this section, we will start with a look at JVM compilation modes, then look at the compiler that can be controlled using the Java 9 platform.

Compilation modes

The changes in the Java 9 platform include granular control of both the c1 and c2 JVM compliers. As you can see in the following illustration, the Java HotSpot JVM has two **Just-in-Time (JIT)** compilation modes--**c1** and **c2**:



The **C1** and **C2** compilation modes use different compilation techniques and, if used on the same code base, can produce different sets of machine code.

C1 compilation mode

The C1 compilation mode inside Java HotSpot VM is typically used for applications that have the following characteristics:

- Quick startup
- Increased optimization
- Client-side

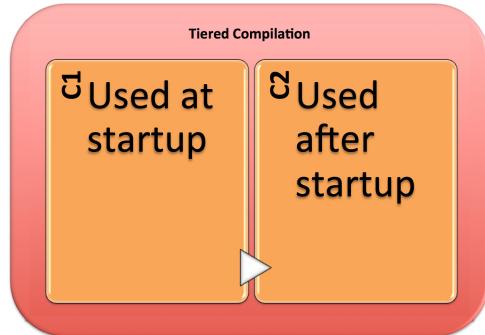
C2 compilation mode

The second compilation mode, C2, is used by applications with the following listed characteristics:

- Long runtimes
- Server-side

Tiered compilation

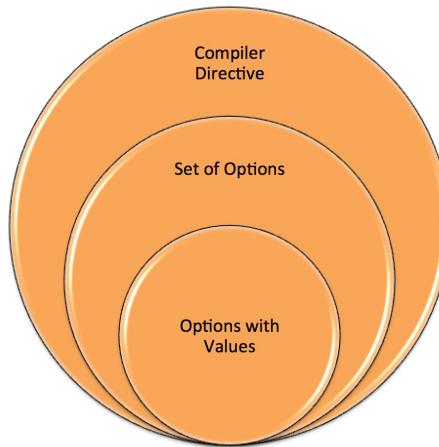
Tiered compilation allows us to use both **c1** and **c2** compilation modes. Starting with Java 8, tiered compilation is the default process. As illustrated here, the **c1** mode is used at startup to help provide greater optimization. Then, once the app has sufficiently warmed up, the **c2** mode is employed:



Compiler control in Java 9

Java 9 comes with the promise of the ability to have finite control over JVM compilers and to make changes at runtime. These additional abilities do not degrade performance. This permits greater fidelity of testing and testing optimization as we can run small compiler tests without having to relaunch the entire JVM.

To control compiler operations, we need to create a directives file. These files contain compiler directives which consist of a set of options with values. Directive files essentially use a subset of JSON:



The **JavaScript Object Notation (JSON)** format is used for data-interchange. The directive files have the following formatting differences from JSON:

- `int` and `doubles` are the only supported number formats
- Double forward slash (`//`) can be used for comment lines
- Trailing commas (,) can be used in arrays and objects
- Escape characters are not supported
- Option names are formatted as strings and do not have to be quoted

You can learn more about JSON at <http://www.json.org>.



We can add our directive file using the following syntax at the command line:

```
| -XX:CompilerDirectivesFile=<file>
```

Here is a shell example of a directives file:

```
[ // Open square bracket marks the start of the directives file

{ // Open curly brace marks the start of a directive block

    // A directives block that applies specifically to the C1 mode
    c1: {
        // directives go here
    },
    // A directives block that applies specifically to the C2 mode
    c2: {
        // directives go here
    },
    // Here we can put a directives that do not apply to
    // a specific compiler mode
},
{
    // can have multiple directive blocks
    c1: {
        // directives go here
    }
    c2: {
        // directives go here
    }
}

] // Close square bracket marks the start of the directives file
```

Diagnostic commands [JEP 228]

The Java Enhancement Proposal 228, **Add More Diagnostic Commands**, defined seven additional diagnostic commands to enhance the ability to diagnose the JDK and the JVM. The new diagnostic commands are detailed here.

The `print_codegenlist` command prints methods that are currently queued for compilation. Since c1 and c2 compilation modes are on separate queues, this command would need to be issued to a specific queue.

The `dump_codelist` diagnostic command will print the following listed information for the compiled methods:

- Full signature
- Address range
- State
 - Alive
 - Nonentrant
 - Zombie

In addition, the `dump_codelist` diagnostic command allows the output to be directed to `stdout` or to a specified file. Output can be in XML form or standard text.

The `print_codeblocks` command allows us to print:

- Code cache size
- Code cache list
- List of blocks in the code cache
- Addresses for code blocks

The `datadump_request` diagnostic command sends a dump request to the **Java Virtual Machine Tool Interface (JVMTI)**. This replaces the **Java Virtual Machine Debug Interface (JVMDI)** and the **Java Virtual Machine Profiling Interface (JVMPI)** interfaces.

With the `set_vmflag` command, we can set a command-line flag or option in the JVM or the libraries.

The `print_class_summary` diagnostic command prints a list of all loaded classes as well as the structure of their inheritance.

The `print_utf8pool` command prints all UTF-8 string constants.

Heap profiling agent [JEP 240]

Java Enhancement Proposal 240 is titled *Remove the JVM TI hprof Agent*. Here are the key terms associated with this JEP and referenced in the title that might be new to you:

- **Tool Interface (TI)**: This is a native programming interface that allows tools to control the execution of applications that are being run inside the Java Virtual Machine. The interface also permits state inquiries. The full nomenclature for this tool is the Java Virtual Machine Tool Interface, or JVM TI.
- **Heap Profiling (HPROF)**: This is an internal JDK tool used for profiling a JVM's use of CPUs and the heap. The most common exposure developers have to `hprof` is the file that is generated when following a crash. The generated file contains a heap dump.

The Java 9 JDK does not contain the `hprof` agent. It was removed largely because there are superior alternatives available. Here is a table of the related functionality:

HPROF Functionality	Alternative
Allocation Profiler (<code>heap=sites</code>)	Java VisualVM
CPU Profiler (<code>cpu=samples</code>) (<code>cpu=times</code>)	Java VisualVM Java Flight Recorder
Heap Dumps (<code>heap=dump</code>)	Internal JVM functionality: <ul style="list-style-type: none">• <code>GC.heap_dump (icmd <pid> GC.heap_dump)</code>• <code>jmap -dump</code>

Interestingly, when HPROF was originally created, it was not intended to be used in production. In fact, it was only meant to test code for the JVM Tool Interface. So, with the advent of the Java 9 platform, the HPROF library (`libhprof.so`) will no longer be part of the JDK.

Removing your JHAT [JEP 241]

The **Java Heap Analysis Tool (JHAT)** is used to parse Java heap dump files. The syntax for this heap dump file parsing tool is as follows:

```
jhat
  [-stack <bool>]
  [-refs <bool>]
  [-port <port>]
  [-baseline <file>]
  [-debug <int>]
  [-version]
  [-h|-help]
  <file>
```

Here is a quick look at the options associated with the JHAT command:

Option	Description	Default
-J<flag>	This passes <flag> to the runtime system.	N/A
-stack<bool>	Toggles tracking of object allocation call stack.	true
-refs<bool>	Toggles tracking of references to objects.	true
-port<port>	Indicates the port for the JHAT HTTP server.	7000
-exclude<exclude-filename>	Exclude indicated file from reachable objects query.	N/A
-baseline<filename>	Specifies the baseline heap dump for use in comparisons.	N/A
-debug<int>	Sets verbosity of output.	N/A
-version	Simply outputs the JHAT release number.	N/A
-h -help	Provides help text.	N/A

JHAT has been part of the Java platform since JDK-6 in an experimental form. It

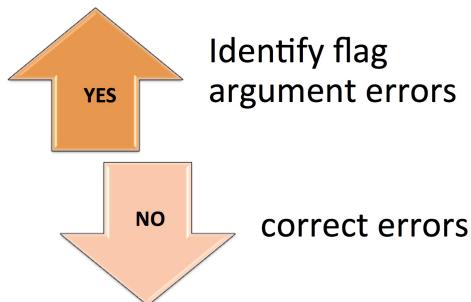
was not supported and has been deemed to be outdated. Starting with Java 9, this tool will no longer be part of the JDK.

JVM command-line flag argument validation [JEP 245]

In this chapter, you have gained exposure to much of the command-line flag usage with the Java 9 platform. Java Enhancement Proposal 245, titled *Validate JVM Command-Line Flag Arguments*, was created to ensure all JVM command-line flags with arguments are validated. The primary goals of this effort were:

- Avoid JVM crashes
- Provide error messages to inform of invalid flag arguments

As you can see from the following graphic, there was no attempt to auto-correct the flag argument errors; rather, just to identify the errors and prevent the JVM from crashing:



A sample error message is provided here and indicates that the flag argument was out of range. This error would be displayed during the flag argument range check performed during the JVM's initialization:

```
| exampleFlag UnguardOnExecutionViolation = 4 is outside the allowed range [ 0 . . . 3 ]
```

Here are some specifics regarding this change to the Java platform:

- Expand on the current `globals.hpp` source file to ensure complete flag default values and permissible ranges are documented
- Define a framework to support adding new JVM command-line flags in the future:
 - This will include value ranges and value sets
 - This will ensure the validity checking will apply to all newly added

command-line flags

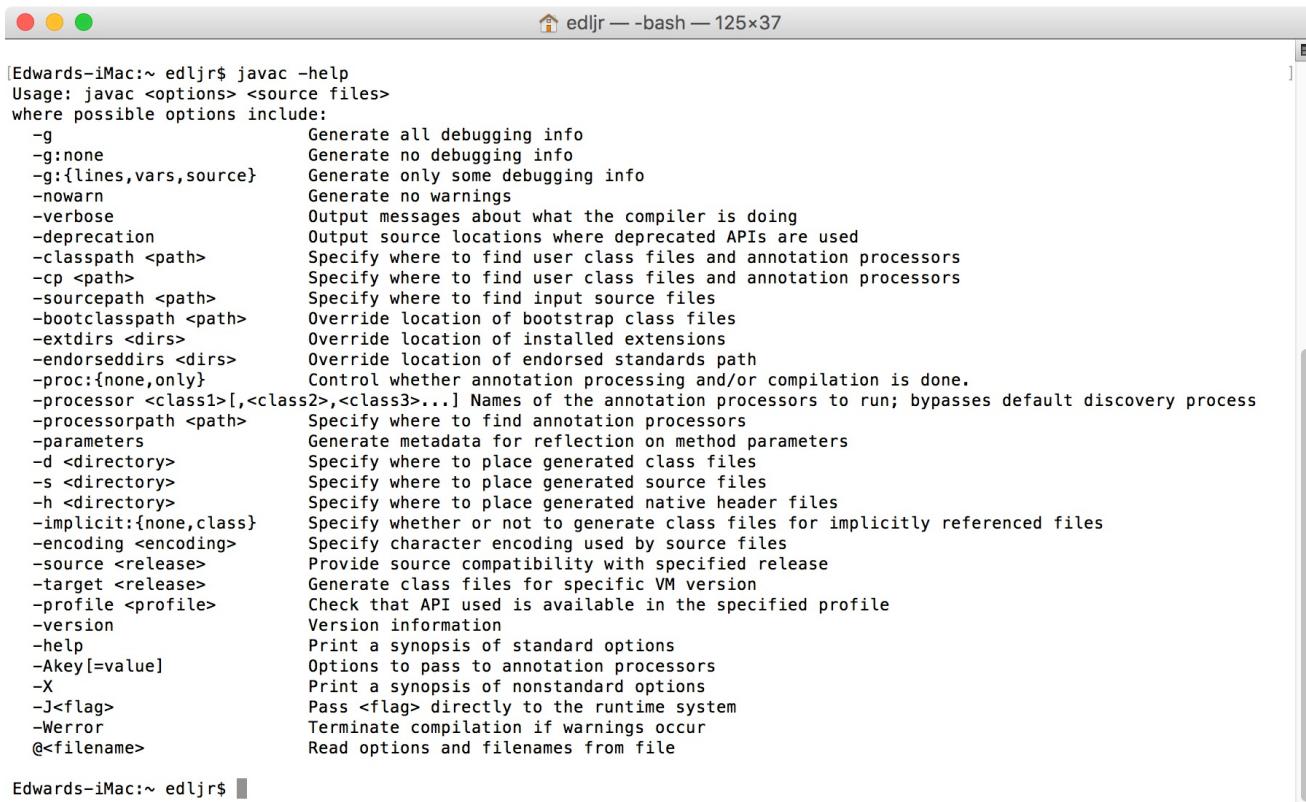
- Modify macro tables:

- Add min/max for optional range
- Add constraint entries for the following:
 - Ensure constraint checks are performed each time a flag changes
 - All manageable flags will continue to be checked while the JVM is running

Compile for older platform versions [JEP 247]

The Java Compiler, `javac`, has been updated for Java 9 to ensure it can be used to compile Java programs to run on user-selected older versions of the Java platform. This was the focus of Java Enhancement Proposal 247, **Compile for Older Platform Versions**.

As you can see in the following screenshot, `javac` has several options including `-source` and `-target`. The `javac` presented in the following screenshot is from Java 8:



```
[Edwards-iMac:~ edljr$ javac -help
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                     Generate no debugging info
  -g:{lines,vars,source}       Generate only some debugging info
  -nowarn                     Generate no warnings
  -verbose                    Output messages about what the compiler is doing
  -deprecation                Output source locations where deprecated APIs are used
  -classpath <path>          Specify where to find user class files and annotation processors
  -cp <path>                  Specify where to find user class files and annotation processors
  -sourcepath <path>          Specify where to find input source files
  -bootclasspath <path>       Override location of bootstrap class files
  -extdirs <dirs>             Override location of installed extensions
  -endorseddirs <dirs>        Override location of endorsed standards path
  -proc:{none,only}            Control whether annotation processing and/or compilation is done.
  -processor <class1>[,<class2>,<class3>...] Names of the annotation processors to run; bypasses default discovery process
  -processorpath <path>        Specify where to find annotation processors
  -parameters                 Generate metadata for reflection on method parameters
  -d <directory>              Specify where to place generated class files
  -s <directory>              Specify where to place generated source files
  -h <directory>              Specify where to place generated native header files
  -implicit:{none,class}      Specify whether or not to generate class files for implicitly referenced files
  -encoding <encoding>        Specify character encoding used by source files
  -source <release>           Provide source compatibility with specified release
  -target <release>           Generate class files for specific VM version
  -profile <profile>          Check that API used is available in the specified profile
  -version                    Version information
  -help                      Print a synopsis of standard options
  -Akey[=value]               Options to pass to annotation processors
  -X                         Print a synopsis of nonstandard options
  -J<flag>                   Pass <flag> directly to the runtime system
  -Werror                     Terminate compilation if warnings occur
  @<filename>                Read options and filenames from file
Edwards-iMac:~ edljr$ ]
```

The `-source` option is used to dictate the Java version accepted by the compiler. The `-target` option informs which version of class files `javac` will produce. By default, `javac` generates class files in the most recent java version and that of the platform APIs. This can cause a problem when the compiled application uses APIs that are only available in the most recent platform version. This would render the application ineligible to run on older platform versions, despite what is dictated with the `-source` and `-target` options.

To address the aforementioned problem, a new command-line option is introduced with the Java 9 platform. This option is the `--release` option and, when used, will automatically configure javac to generate class files that link against a specific platform version. The following screenshot shows the `javac` options with the Java 9 platform. As you can see, the new `--release` option is included:



```
C:\Users\elavi>javac -help
Usage: javac <options> <source files>
where possible options include:
  @<filename>           Read options and filenames from file
  -Akey[=value]          Options to pass to annotation processors
  --add-modules <module>(<module>)*
    Root modules to resolve in addition to the initial modules, or all modules
    on the module path if <module> is ALL-MODULE-PATH.
  --boot-class-path <path>, -bootclasspath <path>
    Override location of bootstrap class files
  --class-path <path>, -classpath <path>, -cp <path>
    Specify where to find user class files and annotation processors
  -d <directory>         Specify where to place generated class files
  -deprecation
    Output source locations where deprecated APIs are used
  -encoding <encoding>   Specify character encoding used by source files
  -endorseddirs <dirs>   Override location of endorsed standards path
  -extdirs <dirs>        Override location of installed extensions
  -g
    Generate all debugging info
  -g:{lines,vars,source}  Generate only some debugging info
  -g:none                Generate no debugging info
  -h <directory>
    Specify where to place generated native header files
  --help, -help            Print this help message
  --help-extra, -X        Print help on extra options
  -imlicit:{none,class}
    Specify whether or not to generate class files for implicitly referenced files
  -J<flag>               Pass <flag> directly to the runtime system
  --limit-modules <module>(<module>)*
    Limit the universe of observable modules
  --module <module-name>, -m <module-name>
    Compile only the specified module, check timestamps
  --module-path <path>, -p <path>
    Specify where to find application modules
  --module-source-path <module-source-path>
    Specify where to find input source files for multiple modules
  --module-version <version>
    Specify version of modules that are being compiled
  -nowarn                 Generate no warnings
  -parameters
    Generate metadata for reflection on method parameters
  -proc:{none,only}
    Control whether annotation processing and/or compilation is done.
  -processor <class1>[,<class2>,<class3>...]
    Names of the annotation processors to run; bypasses default discovery process
  --processor-module-path <path>
    Specify a module path where to find annotation processors
  --processor-path <path>, -processorpath <path>
    Specify where to find annotation processors
  -profile <profile>
    Check that API used is available in the specified profile
  --release <release>
    Compile for a specific VM version. Supported targets: 6, 7, 8, 9
  -s <directory>          Specify where to place generated source files
  -source <release>
    Provide source compatibility with specified release
  --source-path <path>, -sourcepath <path>
    Specify where to find input source files
  --system <jdk>|none     Override location of system modules
  -target <release>
    Generate class files for specific VM version
  --upgrade-module-path <path>
    Override location of upgradeable modules
  -verbose                Output messages about what the compiler is doing
  --version, -version      Version information
  -Werror                 Terminate compilation if warnings occur

C:\Users\elavi>
```

Here is the syntax for the new option:

```
| javac --release <release> <source files>
```

Summary

In this chapter we explored several changes to the Java 9 platform with the common theme of command-line flags. Specifically, we covered unified JVM logging, compiler control, new diagnostic commands, removal of the HPROF heap profiling agent, the removal of the JHAT, command-line flag argument validation, and the ability to compile for older platform versions.

In the next chapter, we will focus on best practices with additional utilities provided with the Java 9 platform. These will include UTF-8, Unicode 7.0, Linux, and more.

Best Practices In Java 9

In the last chapter, we explored several changes regarding command-line flags in Java 9. Specifically, we covered unified JVM logging, compiler control, new diagnostic commands, removal of the HPROF heap profiling agent, the removal of the **Java Heap Analysis Tool (JHAT)**, command-line flag argument validation, and the ability to compile for older platform versions.

In this chapter, we will focus on best practices with additional utilities provided with the Java 9 platform. Specifically, we will cover:

- Support for UTF-8
- Unicode 7.0.0
- Linux/AArch64 port
- Multi-resolution images
- Common Locale Data Repository

Support for UTF-8

Unicode Transformation Format-8 (UTF-8) is a character set that encapsulates all Unicode characters using one to four 8-bit bytes. It is the byte-oriented encoded form of Unicode. UTF-8 is and has been the predominant character set for encoding web pages since 2009. Here are some characteristics of UTF-8:

- Can encode all 1,112,064 Unicode code points
- Uses one to four 8-bit bytes
- Accounts for nearly 90% of all web pages
- Is backward compatible with ASCII
- Is reversible

The pervasive use of UTF-8 underscores the importance of ensuring the Java platform fully supports UTF-8. This mindset led to the Java Enhancement Proposal 226, **UTF-8 property resource bundles**. With Java 9 applications, we have the ability to specify property files that have UTF-8 encoding. The Java 9 platform includes changes to the `ResourceBundle` API to support UTF-8.

Let's take a look at the pre-Java 9 `ResourceBundle` class, followed by what changes were made to this class in the Java 9 platform.

The ResourceBundle class

The following class provides developers with the ability to isolate locale-specific resources from a resource bundle. This class significantly simplifies localization and translation:

```
|     public abstract class ResourceBundle extends Object
```

Creating resource bundles needs a purposeful approach. For example, let's imagine that we are creating a resource bundle that will support multiple languages for a business application. Our button labels, among other things, will be displayed differently depending on the current locale. So, for our example, we can create a resource bundle for our buttons. We can call it `buttonResources`. Then, for each locale, we can create a `buttonResource_<identifier>`. Here are some examples:

- `buttonResource_ja`: for Japanese
- `buttonResource_uk`: for UK English
- `buttonResource_it`: for Italian
- `buttonResource_lh`: for Lithuanian

We can use a resource bundle with the same name as the base name for our default bundle. So, `buttonResource` would contain our default bundle.

To obtain a locale-specific object, we make a call to the `getBundle` method. An example follows:

```
    . . .
ResourceBundle = buttonResource =
    ResourceBundle.getBundle("buttonResource", currentLocale);
    . . .
```

In the next sections we will examine the `ResourceBundle` class by looking at its nested class, field and constructor, and included methods.

The nested class

There is one nested class associated with the `ResourceBundle` class, that is the `ResourceBundle.Control` class. It provides callback methods that are used when the `ResourceBundle.getBundle` method is used:

```
public static class ResourceBundle.Control extends Object
```

The `ResourceBundle.Control` class has the following fields:

- public static final List<String> FORMAT_CLASS
 - public static final List<String> FORMAT_DEFAULT
 - public static final List<String> FORMAT_PROPERTIES
 - public static final long TTL_DONT_CACHE
 - public static final long TTL_NO_EXPIRATION_CONTROL

The class has a single, empty constructor and the following methods:

- `getCandidateLocales()`:

```
public List<Locale> getCandidateLocales(String baseName,  
    Locale locale)
```

Component	Details
Throws	NullPointerException (if <code>baseName</code> or <code>locale</code> is null)
Parameters	<code>baseName</code> : a fully qualified class name <code>locale</code> : the desired locale
Returns	List of candidate locales

- `getControl()`:

```
public static final ResourceBundle.Control getControl(  
    List<String> formats)
```

Component	Details
Throws	<code>IllegalArgumentException (if formats is unknown)</code> <code>NullPointerException (if formats is null)</code>
Parameters	<code>formats</code> : These are the formats that will be returned by the <code>ResourceBundle.Control.getFormats</code> method
Returns	A <code>ResourceBundle.Control</code> that supports the formats specified

- `getFallbackLocale()`:

```
|     public Locale getFallbackLocale(String baseName, Locale locale)
```

Component	Details
Throws	<code>NullPointerException (if baseName or locale is null)</code>
Parameters	<code>baseName</code> : a fully qualified class name <code>locale</code> : the desired locale that could not be found with the <code>ResourceBundle.getBundle</code> method
Returns	The fallback locale

- `getFormats()`:

```
|     public List<String> getFormats(String baseName)
```

Component	Details
Throws	<code>NullPointerException (if baseName is null)</code>
Parameters	<code>baseName</code> : a fully qualified class name
Returns	A list of Strings with their formats so the resource bundles can be loaded

-
- `getNoFallbackControl():`

```
|     public static final ResourceBundle.Control
|         getNoFallbackControl(List<String> formats)
```

Component	Details
Throws	<code>IllegalArgumentException</code> (if formats is unknown) <code>NullPointerException</code> (if formats is null)
Parameters	<code>formats</code> : these are the formats that will be returned by the <code>ResourceBundle.Control.getFormats</code> method
Returns	A <code>ResourceBundle.Control</code> that supports the formats specified without a fallback locale.

- `getTimeToLive():`

```
|     public long getTimeToLive(String baseName, Locale locale)
```

Component	Details
Throws	<code>NullPointerException</code> (if baseName is null)
Parameters	<code>baseName</code> : a fully qualified class name <code>locale</code> : the desired locale
Returns	Zero or a positive millisecond that is offset from the cached time

- `needsReload():`

```

public boolean needsReload(String baseName, Locale locale,
    String format, ClassLoader loader, ResourceBundle bundle,
    long loadTime)

```

Component	Details
Throws	<p><code>NullPointerException</code> (if any of the following listed parameters are <code>null</code>):</p> <ul style="list-style-type: none"> • <code>baseName</code> • <code>locale</code> • <code>format</code> • <code>loader</code> • <code>bundle</code>
Parameters	<p><code>baseName</code>: a fully qualified class name</p> <p><code>locale</code>: the desired locale</p> <p><code>format</code>: the resource bundle format</p> <p><code>loader</code>: the <code>ClassLoader</code> that should be used to load the bundle</p> <p><code>bundle</code>: the expired bundle</p> <p><code>loadTime</code>: a time bundle was added to the cache</p>
Returns	<code>true/false</code> to indicate if the expired bundle needs to be reloaded

- `newBundle()`:

```

public ResourceBundle newBundle(String baseName, Locale locale,
    String format, ClassLoader loader, boolean reload)

```

Component	Details
	<p><code>ClassCastException</code> (if the loaded class cannot be cast to <code>ResourceBundle</code>)</p> <p><code>ExceptionInInitializerError</code> (if initialization fails)</p>

Throws	<p><code>IllegalAccessException</code> (if the class or constructor is not accessible)</p> <p><code>IllegalArgumentException</code> (if the format is unknown)</p> <p><code>InstantiationException</code> (if the class instantiation fails)</p> <p><code>IOException</code> (resource reading error)</p> <p><code>NullPointerException</code> (if any of the following listed parameters are null):</p> <ul style="list-style-type: none"> • <code>baseName</code> • <code>locale</code> • <code>format</code> • <code>loader</code> <p><code>SecurityException</code> (if access to new instances is denied)</p>
Parameters	<p><code>baseName</code>: a fully qualified class name</p> <p><code>locale</code>: the desired locale</p> <p><code>format</code>: the resource bundle format</p> <p><code>loader</code>: the <code>ClassLoader</code> that should be used to load the bundle</p> <p><code>reload</code>: <code>true/false</code> flag indicating if the resource bundle has expired</p>
Returns	Instance of the resource bundle

- `toBundleName()`:

```
|     public String toBundleName(String baseName, Locale locale)
```

Component	Details
Throws	<code>NullPointerException</code> (if <code>baseName</code> or <code>locale</code> is null)

Parameters	<p><code>baseName</code>: a fully qualified class name <code>locale</code>: the desired locale</p>
Returns	<p>The bundle name</p>

- `toResourceName()`:

```
public final String toResourceName(String bundleName,
String suffix)
```

Component	Details
Throws	<p><code>NullPointerException</code> (if <code>bundleName</code> or <code>suffix</code> is null)</p>
Parameters	<p><code>bundleName</code>: the name of the bundle <code>suffix</code>: the suffix for the file name</p>
Returns	<p>The converted resource name</p>

Fields and constructors

The `ResourceBundle` class has one field as described here:

```
|     protected ResourceBundle parent
```

The parent bundle is searched by the `getObject` method when a specified resource is not found.

The constructor for the `ResourceBundle` class is as shown here:

```
|     public ResourceBundle()
|     {
|     }
```

Methods

The `ResourceBundle` class has 18 methods, each described here:

- `clearCache()`:

```
|     public static final void clearCache()
```

Component	Details
Throws	None
Parameters	None
Returns	None

```
|     public static final void clearCache(ClassLoader loader)
```

Component	Details
Throws	<code>NullPointerException</code> (if <code>loader</code> is null)
Parameters	<code>loader</code> : the class loader
Returns	None

- `containsKey()`:

```
|     public boolean containsKey(String key)
```

Component	Details
Throws	<code>NullPointerException</code> (if <code>key</code> is null)
Parameters	<code>key</code> : resource key
Returns	<code>true/false</code> depending on if the key is in the <code> ResourceBundle</code> or parent

bundles

- `getBundle()`:

```
|     public static final ResourceBundle getBundle(String basename)
```

Component	Details
Throws	<code>MissingResourceException</code> (if the resource bundle for the provided <code>basename</code> is not found) <code>NullPointerException</code> (if <code>basename</code> is null)
Parameters	<code>basename</code> : fully qualified class name
Returns	Resource bundle based on the given <code>basename</code> and the default locale

```
|     public static final ResourceBundle getBundle(String basename,
          ResourceBundle.Control control)
```

Component	Details
Throws	<code>IllegalArgumentException</code> (if the passed control performs improperly) <code>MissingResourceException</code> (if the resource bundle for the provided <code>basename</code> is not found) <code>NullPointerException</code> (if <code>basename</code> is null)
Parameters	<code>basename</code> : fully qualified class name <code>control</code> : the control provides information so the resource bundle can be loaded
Returns	Resource bundle based on the given <code>basename</code> and the default locale

```
|     public static final ResourceBundle getBundle(String basename,
          Locale locale)
```

Component	Details

Throws	<p><code>MissingResourceException</code> (if the resource bundle for the provided <code>baseName</code> is not found)</p> <p><code>NullPointerException</code> (if <code>baseName</code> or <code>locale</code> is null)</p>
Parameters	<p><code>baseName</code>: fully qualified class name</p> <p><code>locale</code>: desired locale</p>
Returns	<p>Resource bundle based on the given <code>baseName</code> and <code>locale</code></p>

```
public static final ResourceBundle getBundle(String baseName,
                                         Locale targetLocale, ResourceBundle.Control control)
```

Component	Details
Throws	<p><code>IllegalArgumentException</code> (if the passed control performs improperly)</p> <p><code>MissingResourceException</code> (if the resource bundle for the provided <code>baseName</code> is not found in any of the locales)</p> <p><code>NullPointerException</code> (if <code>baseName</code>, <code>control</code>, or <code>locale</code> is null)</p>
Parameters	<p><code>baseName</code>: fully qualified class name</p> <p><code>control</code>: the control provides information so the resource bundle can be loaded</p> <p><code>targetLocale</code>: desired locale</p>
Returns	<p>Resource bundle based on the given <code>baseName</code> and <code>locale</code></p>

```
public static final ResourceBundle getBundle(String baseName,
                                         Locale locale, ClassLoader loader)
```

Component	Details

Throws	<p><code>MissingResourceException</code> (if the resource bundle for the provided <code>baseName</code> is not found in any of the locales)</p> <p><code>NullPointerException</code> (if <code>baseName</code>, <code>loader</code>, or <code>locale</code> is null)</p>
Parameters	<p><code>baseName</code>: fully qualified class name</p> <p><code>locale</code>: desired locale</p> <p><code>loader</code>: class loader</p>
Returns	<p>Resource bundle based on the given <code>baseName</code> and <code>locale</code></p>

```
public static final ResourceBundle getBundle(String baseName,
    Locale targetLocale, ClassLoader loader,
    ResourceBundle.Control control)
```

Component	Details
Throws	<p><code>IllegalArgumentException</code> (if the passed control performs improperly)</p> <p><code>MissingResourceException</code> (if the resource bundle for the provided <code>baseName</code> is not found in any of the locales)</p> <p><code>NullPointerException</code> (if <code>baseName</code>, <code>control</code>, <code>loader</code>, or <code>targetLocale</code> is null)</p>
Parameters	<p><code>baseName</code>: fully qualified class name</p> <p><code>control</code>: the control providing information so the resource bundle can be loaded</p> <p><code>loader</code>: class loader</p> <p><code>targetLocale</code>: desired locale</p>
Returns	<p>Resource bundle based on the given <code>baseName</code> and <code>locale</code></p>

- `getKeys()`:

```
|     public abstract Enumeration<String> getKeys()
```

Component	Details
Throws	None
Parameters	None
Returns	Enumeration of keys in the <code>ResourceBundle</code> and parent bundles

- `getLocale()`:

```
|     public Locale getLocale()
```

Component	Details
Throws	None
Parameters	None
Returns	the <code>locale</code> of the current resource bundle

- `getObject()`:

```
|     public final Object getObject(String key)
```

Component	Details
Throws	<code>MissingResourceException</code> (if the resource for the provided key is not found) <code>NullPointerException</code> (if <code>key</code> is null)
Parameters	<code>key</code> : this is the key for the desired object
Returns	The object for the key provided

- `getString()`:

```
|     public final String getString(String key)
```

Component	Details
Throws	<code>ClassCastException</code> (if the found object is not a key) <code>MissingResourceException</code> (if the resource for the provided key is not found) <code>NullPointerException</code> (if <code>key</code> is null)
Parameters	<code>key</code> : this is the key for the desired <code>String</code>
Returns	The <code>String</code> for the key provided

- `getStringArray()`:

```
|     public final String[] getStringArray(String key)
```

Component	Details
Throws	<code>ClassCastException</code> (if the found object is not a <code>String</code> array) <code>MissingResourceException</code> (if the resource for the provided key is not found) <code>NullPointerException</code> (if <code>key</code> is null)
Parameters	<code>key</code> : this is the key for the desired <code>String</code> array
Returns	The <code>String</code> array for the key provided

- `handleGetObject()`:

```
|     protected abstract Object handleGetObject(String key)
```

Component	Details
Throws	<code>NullPointerException</code> (if <code>key</code> is null)
Parameters	<code>key</code> : key for the desired <code>Object</code>

Returns	The object for the given key
---------	------------------------------

- handleKeySet():

```
|   protected Set<String> handleKeySet()
```

Component	Details
Throws	None
Parameters	None
Returns	Set of keys in <code>ResourceBundle</code>

- keySet():

```
|   public Set<String> keySet()
```

Component	Details
Throws	None
Parameters	None
Returns	Set of keys in <code>ResourceBundle</code> and its parent bundles

- setParent():

```
|   protected void setParent(ResourceBundle parent)
```

Component	Details
Throws	None
Parameters	<code>parent</code> : the parent bundle for the current bundle
Returns	None

Changes in Java 9

The properties file format, based on ISO-8859-1, was previously supported by the Java platform. That format does not easily support escape characters, although it does provide an appropriate escape mechanism. The use of ISO-8859-1 requires conversion between the text characters and their escaped form.

The Java 9 platform includes a modified `ResourceBundle` class with the default file encoding set to UTF-8 vice ISO-8859-1. This saves applications the time it takes to make the aforementioned escape mechanism conversions.

Unicode 7.0.0

Java Enhancement Proposal 227, titled Unicode 7.0, was created to indicate the need to update the appropriate APIs to support Unicode version 7.0. That version of Unicode was released on June 16, 2014. Previous to Java 9, Unicode version 6.2 was the latest version supported.



*You can learn more about **Unicode version 7.0.0** at the official specification page at: <http://unicode.org/versions/Unicode7.0.0/>.*

At the time this book was published, the most recent Unicode standard was version 10.0.0, released on June 20, 2017. Interestingly, the Java 9 platform will support Unicode version 7.0.0, but not the more recent version 10.0.0 of the Unicode standard. In addition to the two Unicode specifications listed here, from version 7.0.0, will not be implemented by the Java 9 platform:

- **Unicode Technical Standard #10 (UTS #10)**
 - Unicode collation algorithm: details how to compare Unicode strings
- **Unicode Technical Standard #46 (UTS #46)**
 - **Unicode Internationalizing Domain Names for Applications (IDNA) Compatibility processing:** comprehensive mapping for text case and domain name variants

The core of the Java 9 platform changes, specific to Unicode 7.0.0 support, includes the following Java classes:

- `java.lang` package
 - `Character`
 - `String`
- `java.text`.package
 - `Bidi`
 - `BreakIterator`
 - `Normalizer`

Let's take a quick look at each of those classes to help solidify our comprehension of the broad impact that support for Unicode 7.0.0 has on the Java 9 platform.

The `java.lang` package

The `java.lang` package provides fundamental classes used in nearly every Java application. In this section, we will look at the `Character` and `String` classes.

The `Character` class:

```
| public final class Character extends Object implements  
|   Serializable, Comparable<Character>
```

This is one of the many core classes that has been around since the first version of Java. An object of the `Character` class consists of a single field of type `char`.

The `String` class:

```
| public final class String extends Object implements  
|   Serializable, Comparable<String>, CharSequence
```

Strings, another core originating class, are immutable character strings.

Modifying the `Character` and `String` classes to support a newer version of Unicode, version 7.0 for Java 9, is an important step to help keep Java relevant as the premier programming language.

The `java.text` package

The `Bidi`, `BreakIterator`, and `Normalizer` classes are not as widely used as the `Character` and `String` classes. Here is a brief overview of those classes.

The `Bidi` class:

```
| public final class Bidi extends Object
```

This class is used to implement Unicode's bidirectional algorithm. This is used to support Arabic or Hebrew.



For specific information on the Unicode Bidirectional Algorithm, visit <http://unicode.org/reports/tr9/>.

The `BreakIterator` class:

```
| public abstract class BreakIterator extends Object  
|     implements Cloneable
```

This class is used for finding text boundaries.

The `Normalizer` class:

```
| public final class Normalizer extends Object
```

This method contains two methods:

- `isNormalized`: used to determine if `char` values of a given sequence are normalized
- `normalize`: normalizes a sequence of `char` values

Additional significance

As previously stated, JDK 8 supports Unicode 6.2. Version 6.3 was released on September 30, 2013 with the following listed highlights:

- Bidirectional behavior improvements
- Improved Unihan data
- Better support for Hebrew

Version 7.0.0, released on June 16, 2014, introduced the following changes:

- Added 2,834 characters
 - Increased support for Azerbaijan, Russian, and high German dialects
 - Pictographic symbols
 - Historic scripts for several countries and regions
- Updates to the Unicode bidirectional algorithm
- Nearly 3,000 new Cantonese pronunciation entries
- Major enhancements to the Indic script properties

The vast changes to Unicode with version 6.3 and 7.0.0 underscores the importance of the Java 9 platform supporting 7.0.0 as opposed to 6.2, as with Java 8.

The Linux/AArch64 port

Java Enhancement Proposal 237 (JEP 237) had a single goal of porting JDK 9 to Linux/AArch64. In order to understand what this means to us as Java 9 developers, let's talk a bit about hardware.

ARM is a British company that has been creating computing cores and architectures for over three decades. Their original name was **Acorn RISC Machine (ARM)**, with **RISC** standing for **Reduced Instruction Set Computing**. Somewhere along the way, they changed their name to **Advanced RISC Machine (ARM)**, and finally, to ARM Holdings or just ARM. They license their architectures to other companies. ARM reports that there have been over 100 billion ARM processors manufactured.

In late 2011, ARM came out with a new ARM architecture called ARMv8. This architecture included a 64-bit optional architecture called AArch64, which, as you would expect, came with a new instruction set. Here is an abbreviated list of AArch64 features:

- A64 instruction set:
 - 31 general purpose 64-bit registers
 - Dedicated zero or stack pointer registers
 - The ability to take 32-bit or 64-bit arguments
- Advanced SIMD (NEON) - enhanced:
 - 32x 128-bit registers
 - Supports double-precision floating points
 - AES encrypt/decrypt and SHA-1/SHA-2 hashing
- New exception system

Oracle did a great job of identifying this architecture as something that needs to be supported in the new Java 9 platform. The new AArch64 architecture is said to essentially be an entirely new design. JDK 9 has been successfully ported to Linux/AArch64 with the following implementations:

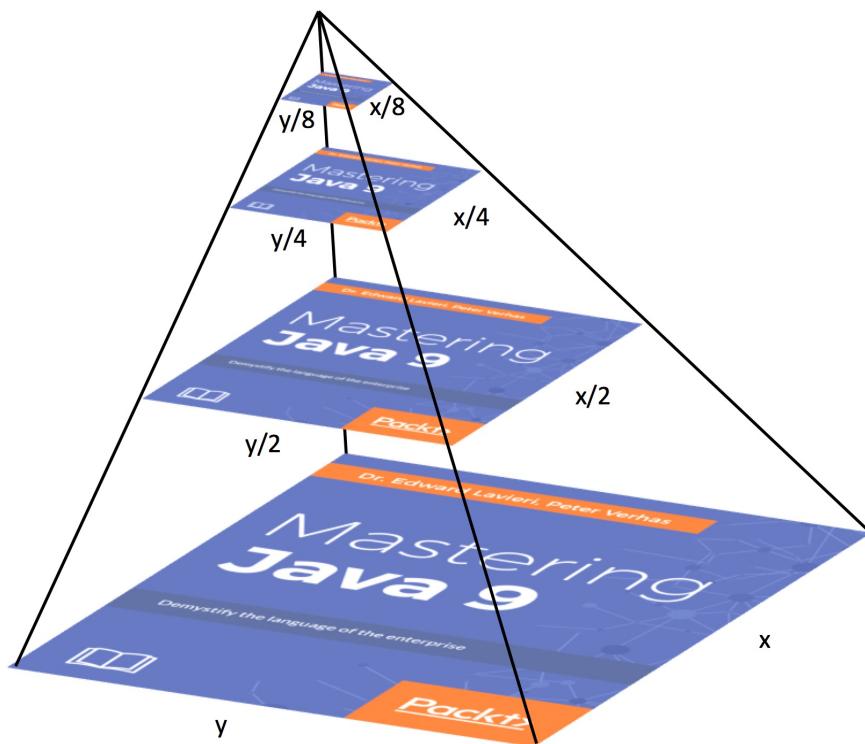
- Template interpreter
- C1 JIT compiler
- C2 JIT compiler



For information about the C1 and C2 JIT compilers, refer to [Chapter 14](#), Command Line Flags.

Multi-resolution Images

The purpose of Java Enhancement Proposal 251 was to create a new API that supports multi-resolution images. Specifically, to allow a multi-resolution image to encapsulate several resolution variants of the same image. This new API will be located in the `java.awt.image` package. The following diagram shows how multi-resolution can encapsulate a set of images, with different resolutions, into a single image:



This new API will give developers the ability to retrieve all image variants or retrieve a resolution-specific image. This is a powerful set of capabilities. The `java.awt.Graphics` class will be used to retrieve the desired variant from the multi-resolution image.

Here is a quick look at the API:

```
package java.awt.image;

public interface MultiResolutionImage
{
    Image getResolutionVariant(float destinationImageWidth,
        float destinationImageHeight);
```

```
|     public List <Image> getResolutionVariants();  
| }
```

As you can see in the preceding code example, the API contains the `getResolutionVariant` and `getResolutionVariants` that return an `Image` and a list of images respectively. Since `MultiResolutionImage` is an interface, we will need an abstract class to implement it.

Common Locale Data Repository (CLDR)

Java Enhancement Proposal 252, uses CLDR Locale Data by default, implements the decision to use locale data from the Unicode Common Locale Data Repository by default. CLDR is a key component of many software applications that supports multiple languages. It is touted as the largest locale data repository and is used by a plethora of large software providers to include Apple, Google, IBM, and Microsoft. The widespread use of CLDR has made it the unofficial industry standard repository for locale data. Making this the default repository in the Java 9 platform further solidifies it as the software industry standard.

Interestingly, CLDR was already part of JDK 8, but was not the default library. In Java 8, we had to enable CLDR by setting a system property as shown here:

```
| java.locale.providers=JRE,CLDR
```

So, in Java 9, we no longer have to enable CLDR as it will be the default repository.

There are additional locale data repositories in the Java 9 platform. They are listed here in their default lookup order:

1. **Common Locale Data Repository (CLDR).**
2. COMPAT - previously JRE.
3. **Service Provider Interface (SPI).**

To change the lookup order, we can change the `java.locale.providers` setting as illustrated:

```
| java.locale.providers=SPI,COMPAT,CLDR
```

In the preceding example, `SPI` would be first, followed by `COMPAT`, and then `CLDR`.

Summary

In this chapter, we focused on best practices with additional utilities provided by the Java 9 platform. Specifically, we covered UTF-8 property files, Unicode 7.0.0, Linux/AArch64 port, multi-resolution images, and Common Locale Data Repository.

In the next chapter, our final chapter, we will look at the future direction for the Java platform by looking ahead to what we can expect in Java 10.

Future Directions

In the last chapter, we focused on best practices with some exciting utilities provided by the Java 9 platform. Specifically, we covered UTF-8 Property Files, Unicode 7.0.0, Linux/AArch64 port, multi-resolution images, and common locale data repository.

This chapter provides an overview of the future development of the Java platform, beyond Java 9. We will look at what is planned for Java 10 and what further changes we are likely to see in the future. Each potential change to the Java platform will be characterized as targeted, submitted, or drafted. Targeted refers to changes that have been earmarked for Java 10. Submitted refers to a change that has been submitted but does not target a specific version of the Java platform. Changes that are drafted are still on the drawing board and are not ready to be submitted or designated as targeted.

Specifically, this chapter covers future changes to the Java platform grouped in the following categories:

- JDK changes
- Java Compiler
- Java Virtual Machine
- JavaX
- Special projects

Future Changes to the JDK

The Java Development Kit is at core of the Java platform and is continually being updated with new capabilities and efficiencies realized with each release. Looking beyond Java 9, we see a multitude of possible changes to the JDK. Many of these changes will be implemented in Java 10 and others might be saved for later releases.

The changes to the JDK in Java 10 and beyond are presented in the following proposal categories:

- Targeted for Java 10
- Submitted Proposals
- Drafted Proposals

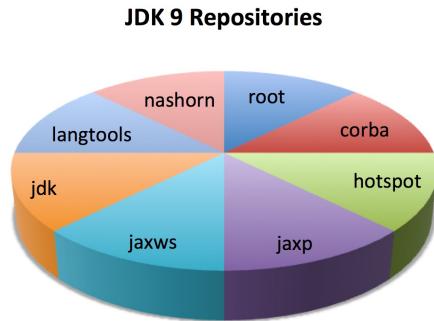
JDK changes targeted for Java 10

At the time of this book's publication, the following listed **Java Development Kit (JDK)**-related changes were slated for inclusion in the Java 10 platform:

- Repository consolidation
- Native-Header Tool Removal

Repository consolidation

The Java 9 platform consists of eight distinct repositories as depicted in the following diagram. In Java 10, we should see all of these repositories combined into a single repository:



Repository consolidation should help streamline development. Moreover, it should increase the ease of maintaining and updating the Java platform.

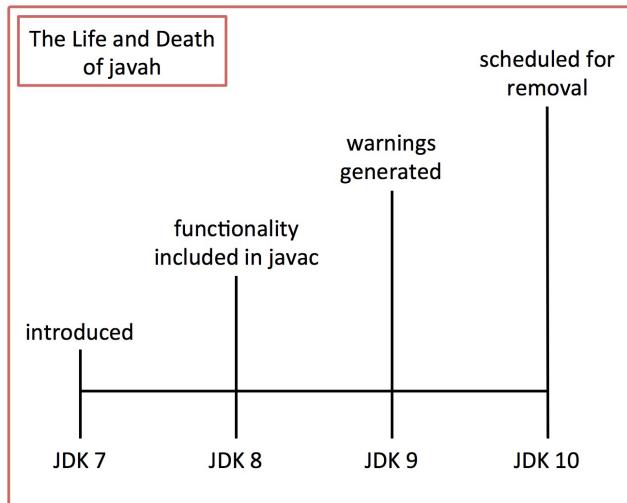


You can get an early look at this repository at <http://hg.openjdk.java.net/jdk10/consol-proto/>.

Native-header tool removal

The javah tool is used to generate C header files and C source files from Java classes. C programs can reference the generated header files and source files.

Here is a look at the life and death of the javah tool:



As illustrated earlier, the javah tool was introduced with Java 7 and its functionality was included in the javac that came with JDK8. This functionality was reportedly superior compared to that of the original tool. In JDK 9, developers received warnings each time the javah tool was used, informing them of its pending removal from the JDK. The tool is slated for removal in JDK 10.

JDK-related submitted proposals

The following Java Enhancement Proposals have been submitted, but have not yet been committed for delivery as part of the Java 10 platform. Oracle has set a two-year release plan, so it is safe to assume that, many if not all, of the proposals listed in this section and beyond have a chance of being part of the Java 10 platform:

- Parallelize the Full GC Phase in CMS
- REST APIs for JMX
- Support Heap Allocation

Parallelize the Full GC Phase in CMS

In [Chapter 7](#), *Leveraging the New Default G1 Garbage Collector*, we reviewed the changes to the **Concurrent Mark Sweep (CMS)** garbage collector. CMS garbage collection involves scanning heap memory, marking objects for removal and then making a sweep to actually remove those objects. The CMS method of garbage collection is essentially an upgraded "Mark and Sweep" method; which you can refer to [Chapter 7](#), *Leveraging the New Default G1 Garbage Collector*, for additional information.

The current downside to CMS garbage collection is that the serial mark and sweep is implemented using a single thread. This results in unwanted pause times. Currently, full garbage collection takes place in four phases:

- **Marking phase:** Mark objects for collection
- **Forwarding phase:** Determine where live objects will be relocated
- **Adjust pointer phase:** Updates points based on new locations of live objects
- **Compaction phase:** Moves objects to designated locations

The future plan for CMS is to implement the mark and sweep so they can be performed in parallel. The change is not to the garbage collection algorithm. Instead, each of the above listed phases will be parallelized. This will result in greater efficiencies for CMS garbage collection and hopefully eliminate, or significantly reduce, pause times.

REST APIs for JMX

Representational State Transfer (REST), RESTful programming, and RESTful API use a client/server cacheable communications protocol, usually HTTP. REST is a common software architecture for developing networked applications.

One of the future changes to the Java platform is to provide RESTful web interfaces to MBeans.

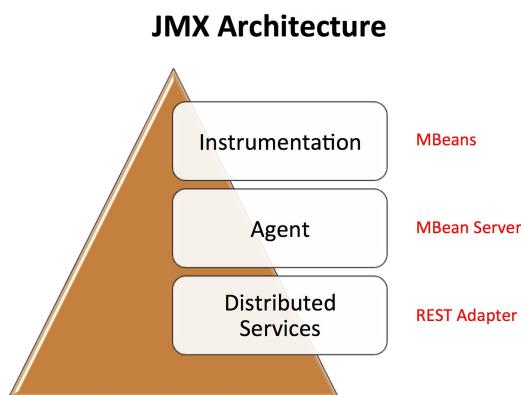


Managed Bean (MBean) is an object in Java that represents a resource to be managed. These resources could include a specific hardware device, an application, a service, or other component.

The interfaces will permit MBeans to use the following HTTP methods:

- CONNECT
- DELETE
- GET
- HEAD
- OPTIONS
- POST
- PUT
- TRACE

MBeans are managed using **Java Management Extensions (JMX)**. The JMX architecture has three levels, as depicted in the following diagram:



As you can see, the REST adapter is part of the **Distributed Services** level. That

level contains both connectors and adapters. The connectors provide mirroring of agent level interfaces to remote clients. The adapters, on the other hand, convert the interfaces using a different protocol. The future change will be to transform the services at the **Agent** level to REST APIs.

Support heap allocation

A proposed future change is to allow developers to designate alternate memory devices for the Java heap. Specifically, the proposal is to permit developers to designate non-DRAM memory for the Java heap. This change takes advantage of the decreasing cost of memory and memory devices.

Implementation is likely to use an `AllocateHeapAt` flag.

JDK-related drafted proposals

This section covers several JDK-related proposals that, at the time of this book's publication, were in the draft phase. That suggests they might not be fully analyzed or might even be cancelled. That being said, it is likely that each of these will move from drafted, to submitted, and then to targeted for the Java 10 platform.

Draft proposals covered in this section are as follows:

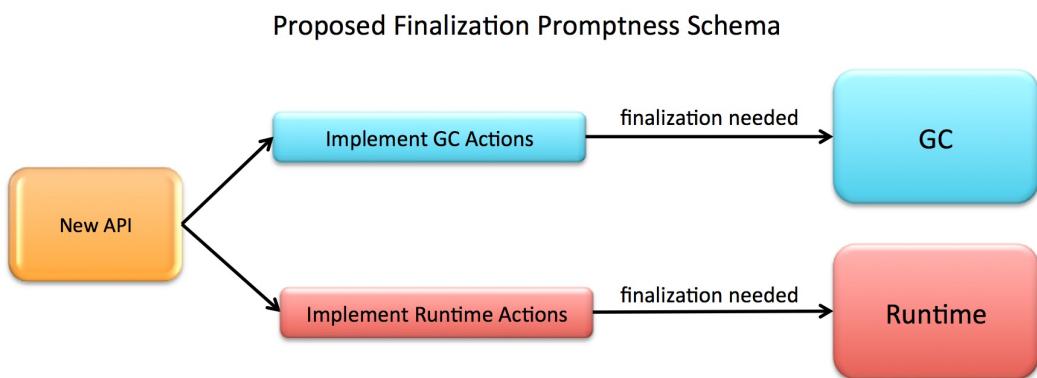
- Finalization promptness
- Java memory model
- Foreign function interface
- Isolated methods
- Reduce metaspace waste
- Improve IPv6 support
- Unboxed argument lists for method handles
- Enhanced MandelblotSet Demo Using Value Types
- Efficient Array Comparison Intrinsics

Finalization promptness

The Java language includes finalization to clean up objects that were unreachable by garbage collection. The proposed change is to make this process quicker and will require modifications to the following:

- ReferenceHandleThread
- FinalizerThread
- java.lang.ref.Reference

Additional changes related to increasing the promptness of finalization includes the creation of a new API. The following graphic details how the API will be able to implement GC and runtime actions, and then inform that finalization needs to take place. This certainly should result in faster processing:



Java memory model

There is a continuing effort to keep **Java's memory model (JMM)** updated. Current efforts are focused on several areas to include:

- Shared memory concurrency
- JVM concurrency support
- JDK components
- Tools

Expected results of JMM-related engineering efforts are as follows:

- Improved formalization
- JVM coverage
- Extended scope
- C11/C++11 compatibility
- Implementation guidance
- Testing support
- Tool support

Foreign Function Interfaces

Foreign Function Interfaces (FFI) are software APIs that permits programs to call methods/functions from a program written in a different language. In an upcoming version of the JDK, we are apt to see an FFI that allows developers to call upon shared libraries and operating-system kernels directly from java methods. The proposed FFI will reportedly also enable developers to manage native memory blocks.

The new FFI will be similar to **Java Native Access (JNA)** and **Java Native Runtime (JNR)**. JNA is a library that permits access to native shared libraries without having to use the **Java Native Interface (JNI)**. JNR is a Java API that is used for calling native code. The proposed FFI will permit and optimize native method calls as well as optimized native memory management.

Isolated methods

The `MethodHandles.Lookup` class is part of the `java.lang.invoke` package. We use lookup objects to create method handles and a lookup class to access them. Here is the header for the lookup class:

```
| public static final class MethodHandles.Lookup extends Object
```

Future changes to the `MethodHandles.Lookup` class will support the loading of method byte codes without the need for an attached class. Furthermore, these methods will be referenced using method handles. The class will have a new `loadCode` method.

Reducing metaspace waste

Currently, when metaspace chunks are freed, they cannot be used as different sized chunks. So, if metaspace chunk A was freed and was of size X, that space cannot be reused for a metaspace chunk greater than or less than size X. This results in a tremendous amount of unusable metaspace waste. This can also lead to out-of-memory errors.

A future change to the JDK will resolve this issue by increasing the reuse of metaspace chunks. The change will support the following cases:

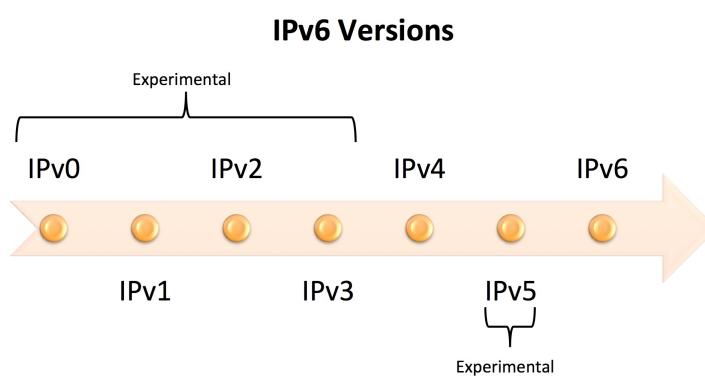
- Allow neighboring chunks to form a larger chunk
- Allow larger chunks to be divided into smaller chunks

This proposed change resolves the issue by ensuring smaller chunks can be reused and that larger chunks are not wasted as they can be split to support the reuse of smaller chunks.

Improving IPv6 support

Internet Protocol version 6 (IPv6) is the current version of the Internet Protocol. The Internet Protocol provides the identification and location schema that enables Internet traffic routing. IPv6 is considered an Internet layer protocol that is used by packet-switched networking.

The following diagram shows the history of the Internet Protocol:



IPv6 is the replacement for IPv4 and has several changes that the Java platform should support. Key IPv6 changes from IPv4 are categorized as follows:

- Jumbograms
- Larger address space
- Mobility
- Multicasting
- Network-layer security
- Options extensibility
- Privacy
- Simplified router processing
- Stateless address auto-configuration

As the Internet continues to transition from IPv4 to IPv6, the following cases are possible and should all be supported in the Java 10 platform:

- Multiple versions of IPv4 exist
- One version of IPv6 exists
- Multiple versions of IPv6 exist

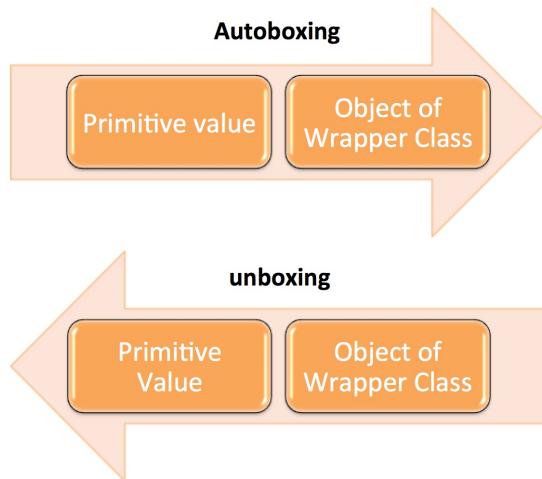
- Multiple versions of IPv4 and one version of IPv6 exist
- Multiple versions of IPv4 and IPv6 exist

Unboxed argument lists for method handles

The way in which unboxed argument lists are currently handled can lead to processing inefficiencies. This is especially true when we use `Object[]` or `List<Object>` as variable-length argument lists. Java uses `java.lang.invoke` to transform the method calls using boxing. In Java, autoboxing is when the compiler automatically converts primitive types and their corresponding object wrapper classes. Here is the list of wrapper classes along with the corresponding primitive type:

Wrapper class	Primitive type
Boolean	<code>boolean</code>
Byte	<code>byte</code>
Character	<code>char</code>
Double	<code>double</code>
Float	<code>float</code>
Integer	<code>int</code>
Long	<code>long</code>
Short	<code>short</code>

As you can see from the following illustration autoboxing occurs when we go from primitive values to an object of the associated wrapper class and, when we go from an object of a wrapper class to primitive values it is called unboxing:



The inefficiencies are due to mismatches between the argument's list actual types and the array or list encasing them. In a future Java release, these inefficiencies will be removed. A new `ArgumentList` class will be added to the Java platform that polymorphically boxes valid arguments lists into a heap node.

Enhanced MandelblotSet demo using value types

This low priority Java Enhancement Proposal is likely to be implemented in Java 10 as its scope is limited. The plan is to develop a sample Java application that demonstrates improvements in memory and performance specific to using Valhalla project components, value types, and generics instead of primitive types.



Valhalla project components refer to user-defined custom immutable primitive types as value types.

You can read more about value types in the Java Virtual Machine section of this chapter.

A Mandelbrot set is a specific example of fractal mathematics used in chaos theory. The sample MandelbrotSet that accompanied JDK 8 provides a comparison of parallel and sequential data streams. In Java 10 or beyond, the sample MandelbrotSet will be updated to show performance and memory efficiencies between using Valhalla project components, value types and generics as opposed to primitive types.

Efficient array comparison intrinsics

A future change to the Java platform will be to include a method for comparing arrays. Currently, this is something developers have to code on their own. The change will be incorporated by adding something similar to the `compareTo` method in `java.util.Arrays`.

Although specifics are not available, the prospect of being able to compare arrays using native functionality is exciting. This is a component that will save many developers time. This is likely to be realized in the Java 10 platform release.

Future changes to the Java Compiler

There are two notable draft changes to the Java platform, specifically the Java Compiler. These Java Enhancement Proposals are listed as follows and detailed in this section:

- Policy for retiring `javac -source` and `-target` options
- Pluggable static analyzers

Policy for retiring javac -source and -target options

A formal draft proposal has been submitted to define a policy for retiring `-source` and `-target` options. This effort is to help reduce maintenance costs of the compiler. The `-source` and `-target` options were provided to ease development efforts, but not formally required by any standards. Starting with the Java 9 platform, these target options are not recognized.

The new policy is called "one plus three back" which means that the current version will be supported as well as the three previous releases. This policy will persist with JDK 10.

Pluggable static analyzers

An ongoing research Java Enhancement Proposal was initiated in the summer of 2013 as an exploratory measure and future support for a full Java Enhancement Proposal to empower developers to define extensions that can, at compile time, conduct the arbitrary static analysis. The research is to see how a pluggable static type analyzer framework can be implemented for the Java Compiler.

The goals of the research are as follows:

- Collect static analyzer requirements
- Analyze static analyzers
- Determine the requirements for a framework that supports static analyzers
- Implement and test

The final outcome of the ongoing research will be either to submit a feature Java Enhancement Proposal or to make the recommendation that pursuit of the feature cease.

Future Changes to the Java Virtual Machine

Several new features and enhancements to the Java Virtual Machine (JVM) and core libraries have been submitted and drafted. It is likely that at least some of these features and enhancements will be realized in the Java 10 platform, and others will be saved for later releases.

JVM-related submitted proposals

There are three Java Enhancement Proposals that have been submitted. While, not currently earmarked for Java 10, it is likely that we will see the changes when Java 10 is released. The three proposals are listed as follows:

- Container aware Java
- Enable execution of Java methods on GPU
- Epsilon GC: The arbitrarily low overhead gGarbage (non-) collector

Container aware Java

An effort is being made so the JVM and core libraries are aware when they are running in a container. Moreover, to be adaptive in the use of available system resources. This feature is especially relevant with the ubiquitous nature of cloud computing.

There are two major components of the proposed feature:

- Detection:
 - Determine if Java is running inside a container
- Container resource exposure:
 - Expose container resources limits
 - Expose container resource configuration

Several configuration status points have been initially identified:

General	CPU-related	Memory-related
isContainerized	CPU Period	Block I/O Device Weight
	CPU Quota	Block I/O Weight
	CPU Set Memory Nodes	Current Memory Usage
	CPU Sets	Device I/O Read Rate
	CPU Usage	Device I/O Write Rate
	CPU Usage Per CPU	Max Memory Usage
	Number of CPUs	Maximum Kernal Memory
		Memory Swappiness
		OOM Kill Enabled
		OOM Score Adjustment
		Shared Memory Size
		Soft Memory Limit
		Total Memory Limit

Initially, this feature is scheduled to support Docker on Linux-64. A likely scenario is that this feature be released with Java 10 with sole support for Docker on Linux-64. Then, feature support will be expanded in subsequent releases of the Java platform.

Enable execution of Java methods on GPU

Enabling the seamless ability for Java applications to take advantage of GPUs is the subject of project Sumatra. The goal is to use Java's Stream API in parallel and the lambda programming model. It makes great sense for us to exploit the processing power and efficiency of GPUs.

The overarching goal is to make this feature easy to use for developers. The feature will be implemented with the following characteristics:

- Do not change the syntax of the Java parallel stream API
- Hardware and software stacks should be automatically detected
- Automatic detection and analysis to determine if using the GPU makes sense from a performance standard
- Provide CPU execution when offloading processing to a GPU fails
- There will be no performance degradation
- There will be no new security risks introduced by this feature
- There will be memory persistence between the CPU and GPU

The key benefit of this Java Enhancement Proposal will be performance improvements for our Java applications.

Epsilon GC - The arbitrarily low overhead garbage (non-) collector

In [Chapter 7](#), *Leveraging the New Default G1 Garbage Collector*, we detailed the enhancements to Java's Garbage Collection with the release of the Java 9 platform. In the spirit of continuous improvement, a Java Enhancement Proposal has been submitted to develop a garbage collection to specifically handle memory allocation. This garbage collector will signal the JVM to shutdown when no more memory is available on the Java heap.

The goal is for this garbage collector to be passive and use very limited overhead. The introduction of this garbage collection is not intended to degrade performance.

This change will not impact current garbage collectors.

JVM-related drafted proposals

The following Java Enhancement Proposals have been drafted for a future version of the Java platform and are detailed in this section:

- Provide stable USDT probe points on JVM compiled methods
- Concurrent Monitor Deflation
- Low-overhead way of sampling Java heap allocations
- Diagnostic Command Framework
- Enhanced Class Redefinition
- Enable NUMA mode by default when appropriate
- Value objects
- Align JVM Access Checks

Provide stable USDT probe points on JVM compiled methods

User-level Statistically Defined Tracing (USDT) is used to insert probe points to mark the entry and exit of methods. Compilers then permit a handshake with tracing tools so that those tools can discover the probe points and manipulate them.



Common tracing tools are Dtrace and Berkeley Packet Filters (BPF).

The Java Virtual Machine, even with JVM 9, does not support this technology set. The current lack of support stems from how the JVM generates compiled code; it does this dynamically without any static **Executable Linkable Files (ELFs)**.

Tracing tools need the ELFs to work. An additional mitigating factor is that the JVM dynamically patches its own generated code which does not support external patching.

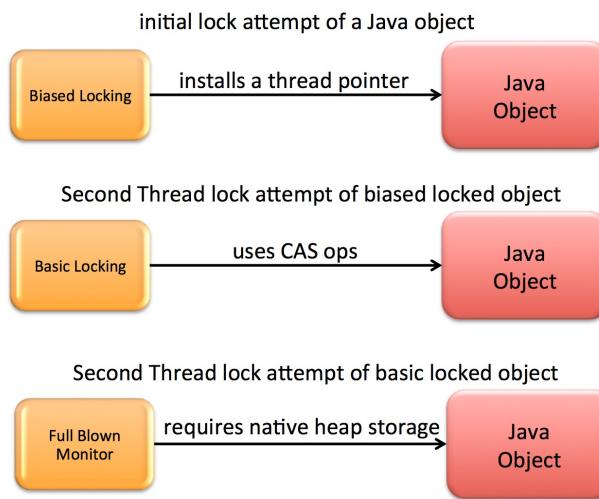
In a future Java release, likely Java 10, the **JVMTI (JVM Tools Interface)** will be modified to support probe tools to perform their standard operations on the JVM's dynamically compiled code. Provisionally identified changes to JVMTI APIs include:

- Adding patch points or method entry and exit
- The enumeration of the compiled methods
- State change notifications on compiled method load
- Query support
- Toggle trace points on/off
- Making chunks of compiled methods inspectable

The good news is that there will not need to be any changes to how Java code is compiled. It can already be patched, so the required functionality will be created by modifying the USDT API as well as a few changes to the JVM.

Concurrent monitor deflation

A monitor, in our context, is a synchronized mechanism that controls concurrent access to an object. Monitors help prevent multiple threads from accessing a monitored object at the same time. The JVM automatically switches between three monitor implementation methods. The three implementation methods are illustrated as follows:



The initial lock of a Java object uses biased locking. That method ensures only the locking thread can lock the object. With this approach, the JVM installs a thread pointer in the Java object. When a second thread attempts to lock the Java object, the JVM switches to the basic locking monitor implementation method. This second method uses **compare-and-swap (CAS)** operations. When a CAS operation fails, such as when a second thread attempt to lock the Java object, the JVM switches to the third monitor implementation method. That method is a full-blown monitor. This method requires native heap storage which is referred to as the monitor being inflated.

The purpose of the Concurrent Monitor Deflation Java Enhancement Proposal is to perform monitor deflation while the threads are running. This will decrease the JVM-induced pause times.

Provide a low-overhead way of sampling Java heap allocations

Mismanagement of Java heaps can result in heap exhaustion, and insufficient memory due to memory fragmentation (GC thrashing). In a future release of Java, most likely Java 10, we will have a means of sampling Java heap allocations. This will be implemented by enhancing the **Java Virtual Machine Tools Interface (JVMTI)**. The resulting functionality will provide an extremely low-overhead solution.

Diagnostic Command Framework

Java Enhancement Proposal 137, Diagnostic Command Framework, proposes a framework be created for sending diagnostic commands to the Java Virtual Machine.

The framework will include a **Java Management Extension** (JMX) interface, which will permit remote issuing of diagnostic commands via a JMX connection.

The JRocket Mission Control tools already have this feature successfully implemented. This served as proof of concept and it is therefore extremely likely that this enhancement will be part of the Java 10 platform.

Enhanced Class Redefinition

Java Enhancement Proposal 159, Enhanced Class Redefinition, calls for enhanced JVM capabilities in regards to class redefinition at runtime. Specifically the proposal includes the following class redefinition operations:

- Adding super types
- Adding methods
- Adding static fields
- Adding instance fields
- Removing methods
- Removing static fields
- Removing instance fields

Current JVM class redefinition capabilities are limited to method swapping. This is viewed as extremely restrictive. With the new proposed enhancement, developers will not have to restart their applications after changes. This is especially beneficial when dealing with large and distributed systems.

Enable NUMA mode by default when appropriate

Java Enhancement Proposal 163, enable NUMA mode by default when appropriate. This proposal is only applicable to NUMA hardware. The intent is to have the JVM enable the following flag when it detects NUMA hardware:

| XX:+UseNUMA

This flag can currently be evoked manually. With the proposed enhancement, it will be evoked automatically by the JVM when it has detected that it is running on a NUMA piece of hardware.



Non-Uniform Memory Access (NUMA) is a memory model used in computer multiprocesssing. With this memory model, access time is dependent on the memory location relative to that of the processor.

This will be an easy enhancement to implement and is likely to be part of the Java 10 platform release.

Value objects

Java Enhancement Proposal 169, value objects, intended to provide the necessary JVM infrastructure to permit working with objects that are immutable as well as objects that are without reference. This new infrastructure will allow for the efficient by-value computation with non-primitive data types.

The set of goals for this proposal include the following:

- More closely align `java.lang.Integer` and `int` semantics.
- Make Java data structures more portable
- Support abstract data types with a performance similar to that of Java primitive data types:
 - User-defined
 - Library-defined
- Optimize parallel computations by enabling function-style computation with pure data
- Improve support for:
 - Complex numbers
 - Vector values
 - Tuples
- Increase safety and security
- Decrease "defensive copying"

One of the stated implementation strategies is to add a `lockPermanently` operation. It will get passed an Object and then mark that Object as both immutable and unaliasable. The concept of a permanently locked object stipulates that:

- Fields cannot be changed
- Elements of an array cannot be changed
- No synchronization is possible
- 'Waiting' methods cannot be evoked
- 'Notifying' methods cannot be evoked
- Identity hash codes inquiries are not permitted
- Pointer equality checks cannot be performed

This is likely to be one of the more popular additions to the Java 10 platform.

Align JVM Access Checks

Java Enhancement Proposal 181, Align JVM Checks with Java Language Rules for Nested Classes, focuses on the need to align JVM access checking rules with Java language rules, specifically for constructors, fields, and methods in nested classes. This will be accomplished by partitioning related classes in nests. Class files will be able to access private names of other class files in the same nest.

Nests will share an access control context. With the advent of nests, access bridges will not be required. The bulk of the change will be to the JVM's access rules.

Future Changes to JavaX

The `Javax.*` packages are the subject of two specific Java Enhancement Proposals that have been submitted for a future Java platform release. Those proposals are as follows:

- JMX specific annotations for registration of managed resources
- Modernize the GTK3 Look and Feel implementation

JMX specific annotations for registration of managed resources

The draft Java Enhancement Proposal titled, JMX specific annotations for registration of managed resources, will provide a set of annotations for registration and configuration of **MBeans (Managed Bean)**.



An MBean is a Java Object representing a manageable resource (app, service, component, or device).

The goal of this proposal is to lessen the burden on developers in the registration and configuring of MBeans. In addition, the source code readability will increase by ensuring all MBean declaration components are co-located.

The JMX specific annotations will be located in the `javax.management.annotations` package.

This Java Enhancement Proposal has been specifically planned for Java 11. Although, there is a possibility that it could be redesigned for Java 10.

Modernize the GTK3 Look and Feel Implementation

GTK3 is a widget toolkit used for creating graphical user interfaces, formally known as the GIMP toolkit. The draft Java Enhancement Proposal titled, Modernize the GTK3 Look and Feel implementation, calls for the rewriting of the current GTK2 Look and Feel so that it uses GTK3 instead.

GTK3 implementation will not replace GTK2. It is important to note that one or the other, not both of these can be used at runtime.



You can access the GTK3 reference manual at <https://developer.gnome.org/gtk3/stable/>.

Ongoing Special Projects

Java Enhancement Proposals present design and implementation changes to the Java platform. The criteria for a JEP being drafted is that the work must meet at least one of the following:

- At least two weeks of engineering work
- Signifies a significant change to the JDK
- Represents a high demand issue for developers or customers

Projects, on the other hand, represent collaborative efforts that are sponsored by one of the following groups:

- 2D Graphics
- Adoption
- AWT
- Build
- Compatibility and specification review
- Compiler
- Conformance
- Core Libraries
- Governing Board
- HotSpot
- Internationalization
- JMX
- Members
- Networking
- NetBeans Projects
- Porters
- Quality
- Security
- Serviceability
- Sound
- Swing
- Web

Groups are formal and new ones can be proposed.

The following listed active projects represent possible future enhancement areas to the Java platform. Brief information about each project is provided later in this section and provides insight into general areas of future changes:

- Annotations pipeline 2.0
- Audio Synthesis Engine
- Caciocavallo
- Common VM Interface
- Compiler Grammar
- Da Vinci Machine
- Device I/O
- Graal
- HarfBuzz Integration
- Kona
- OpenJFX
- Panama
- Shenandoah

Annotations pipeline 2.0

This project explores improvements to how annotations are handled within the Java compiler pipeline. There is no intention to propose changing specifications; rather, the focus is on performance enhancements.

Audio Synthesis Engine

This project is looking at the creation of a new midi synthesizer for the JDK. The current midi synthesizer belongs to a licensed library. The working group would like to see the new midi synthesizer as an open source JDK asset.

Caciocavallo

The Caciocavallo project aims to improve the OpenJDK **Abstract Windows Toolkit (AWT)** internal interfaces. This extends to 2D subsystems. The proposed improvement stands to ease the way AWT is ported to new platforms.

Common VM Interface

The Common VM Interface project has the goal of documenting the VM interface for OpenJDK. This should make it easier for Classpath VMs and other VMs to use OpenJDK.

Compiler Grammar

The Compiler Grammar project is working on an experimental Java Compiler that is based on ANTLR grammar. **ANTLR, Another Tool for Language Recognition**, is a parser that reads, processes, and executes structured text or binary files. The project team hopes this Java Compiler will replace the current one as it uses a hand-written parser, **LALR (Look-Ahead Left to Right)**. The LALR parser has been identified by the project group as fragile and difficult to extend.

Da Vinci Machine

The Da Vinci Machine Project, represents the effort to extend the JVM with support for non-Java languages. Current efforts are focused on allowing the new languages to exist alongside Java in the JVM. Performance and efficiency are key characteristics of the effort.

Device I/O

This project intends to provide access to generic peripheral devices via a Java-level API. The initial list of peripheral devices the project team wants to support include:

- **GPIO (General Purpose Input/Output)**
- **I2C (Inter-Integrated Circuit Bus)**
- **SPI (Serial Peripheral Interface)**
- **UART (Universal Asynchronous Receiver/Transmitter)**

Graal

The Graal project has the goal of exposing VM functionality via Java APIs. This exposure will permit developers to write, in Java, dynamic compilers for a given language runtime. This effort includes the development of a multi-language interpreter framework.

HarfBuzz Integration

The HarfBuzz Integration project hopes to integrate the HarfBuzz layout engine into the Java Development Kit. This is intended to replace the ICU layout engine with the HarfBuzz layout engine. The ICU layout engine has been deprecated, solidifying the importance of this project's future success.

Kona

The Kona project, is working to define and implement Java APIs to support the **Internet of Things (IoT)** domain. This includes networking technologies and protocols. Although not stated, safety and security will be paramount to this effort's implementation success.

OpenJFX

There are not many details available regarding the OpenJFX project. The stated goal of this project is to create the next-generation Java client toolkit. Based on the project title, it can be assumed that the group wants to create an OpenJFX version of JavaFX, which is a set of packages used to create rich internet applications.

Panama

Project panama is focused on enhancing the connections between JVM and non-Java APIs. The project includes the following selected components:

- Native function calls
- Native data access from JVM
- Native data access inside JVM heap
- New data layouts in JVM heap
- API extraction tools for header files

The project team has generated a repository tree that matches JDK 9's structure. This significantly increases the likelihood of the project's success.

Shenandoah

Project Shenandoah has the goal of significantly reducing the pause times with garbage collection operations. The approach is to have more garbage collection operations run concurrently with the Java application. In [Chapter 7, Leveraging the New Default G1 Garbage Collector](#) you read about CMS and G1. The Shenandoah project intends to add concurrent compaction to the possible garbage collection approaches.

Summary

In this chapter we provided an overview of the future developments of the Java platform, beyond Java 9. We looked at what is planned for Java 10 and what further changes we are likely to see beyond Java 10. Each potential change to the Java platform was characterized as targeted, submitted, or drafted. Specifically, we covered future changes to the Java platform grouped in the following categories: JDK Changes, Java Compiler, Java Virtual Machine, JavaX, and special projects.