Petrucci

Java 23

Key Concepts in Brief

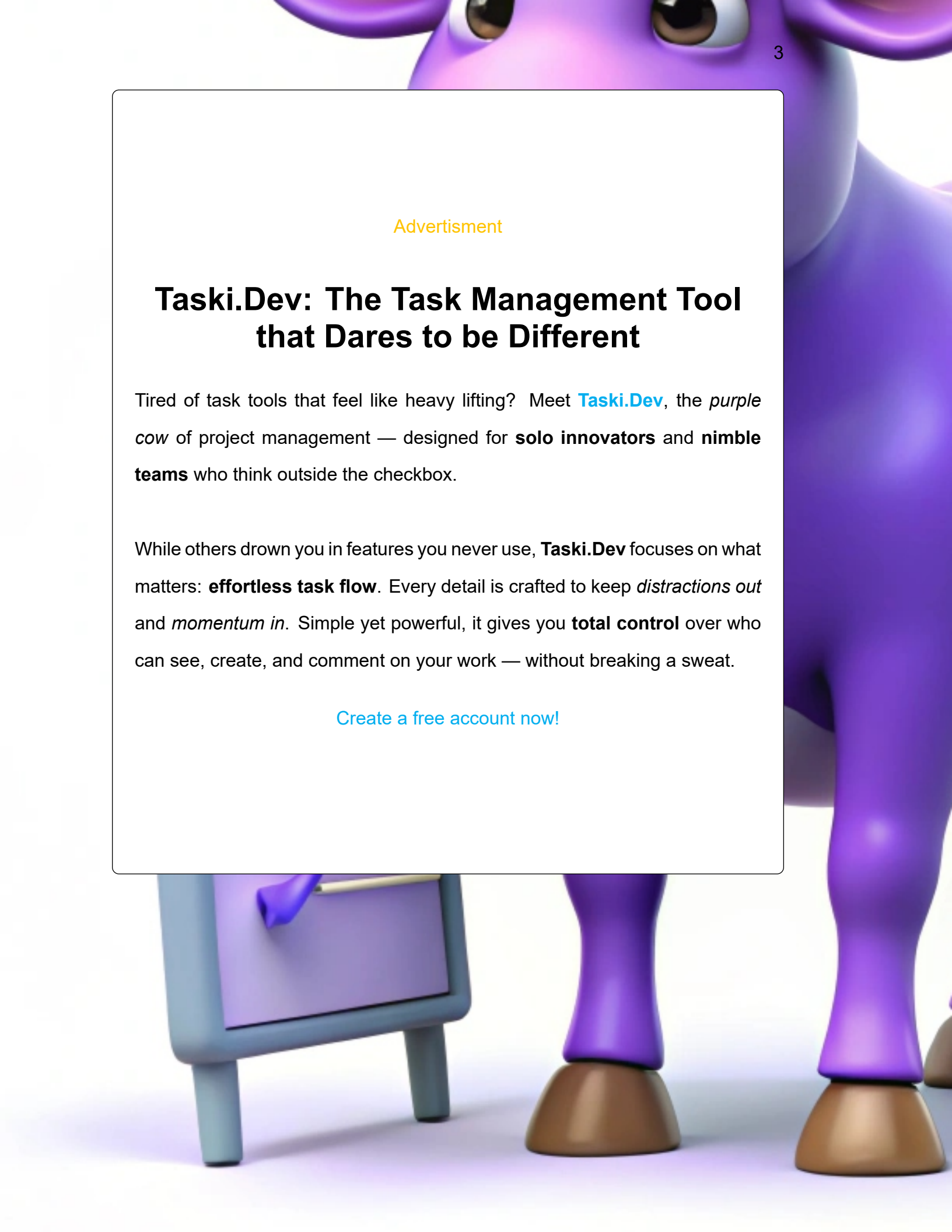
2024

OCTOBER 2, 2024

FIRST EDITION

CONTACT INFORMATION

sergio@petrucci.dev
https://petrucci.dev



Contents

Chapter 1

What's New in Java 23

In September 2024, Oracle announced that JDK 23 is now generally available.


```
Java 23 is a short-term JDK release and will be succeeded by
Java 24 in March 2025. As Java 23 is not an LTS (Long-Term Support)
version, it will receive premier support for only six months.
```

This release introduces twelve notable enhancements, each supported by its own **JDK Enhancement Proposals (JEPs)**, which include eight preview features and one incubator feature. These enhancements encompass improvements to the

Java Language, APIs, performance, and the tools included in the JDK.

- **Simplified Module Imports (Preview)**

JEP 476 introduces a streamlined way to import all packages exported by a module, making it easier to reuse modular libraries.

- **Improved Beginner Friendliness (Preview)**

JEP 477 aims to make Java more beginner-friendly by allowing simplified declarations for single-class programs, allowing beginners to gradually learn advanced features as their skills progress.

- **Enhanced Concurrent Programming (Preview)**

JEP 480 introduces structured concurrency, which treats related tasks running in different threads as a single unit, simplifying error handling, cancellation, and improving reliability and observability.

- **Scoped Values for Efficient Data Sharing (Preview)**

JEP 481 introduces scoped values, allowing methods to share immutable data with callees and child threads more efficiently than thread-local variables.

- **More Flexible Constructors (Preview)**

JEP 482 allows statements before explicit constructor invocations (`super()` or `this()`), enabling field initialization before another constructor is called, which improves reliability when methods are overridden.

- **Markdown Support in JavaDoc Comments**

JEP 467 enables writing JavaDoc documentation comments in Markdown, offering

a more readable and user-friendly alternative to HTML and JavaDoc tags.

- **ZGC Generational Mode Now Default**

Z Garbage Collector (ZGC) now defaults to generational mode for improved performance, with the non-generational mode being deprecated.

- **Annotation Processing Changes**

Annotation processing in `javac` is no longer enabled by default and requires explicit configuration.

- **Security Enhancements**

JMX Subject Delegation and the JMX Management Applet (m-let) have been removed to prepare for the removal of the Security Manager in a future release. Additionally, keychain support is expanded in the Apple provider for Java Security to include system root certificates.

- **Performance Optimizations**

Several performance optimizations have been implemented, including a new Parallel GC Full GC algorithm, support for duration until another instant in `java.time.Instant`, and a change in the default maximum fraction digits for `java.text.DecimalFormat`.

- **Legacy Features Removed**

Several legacy features have been removed, including:

`ThreadGroup.stop`, `Thread.suspend/resume`, `ThreadGroup.suspend/resume`, and the `jdk.random module`.

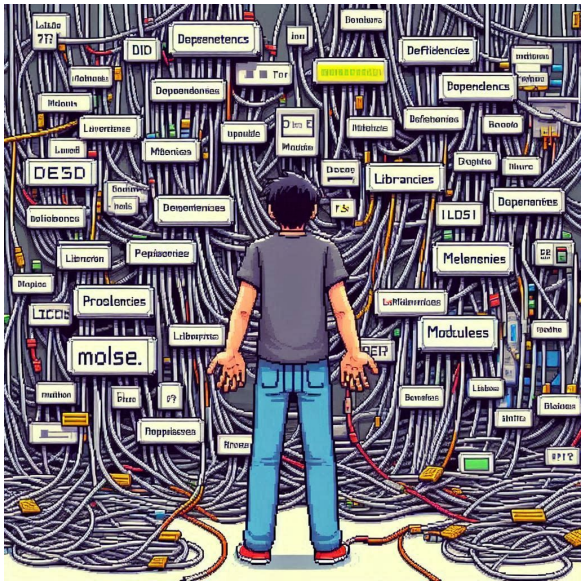
Chapter 2

Simplified Module Imports

Java 23 introduces a preview feature designed to simplify how developers import packages from modules: **Simplified Module Imports**. This chapter explores this new feature, its benefits, potential challenges, and how it integrates with other Java language features.

2.1 The Need for Simplified Imports

Before Java 9, managing dependencies and code organization often led to challenges like ”classpath hell.” Java 9 introduced the module system (Project Jigsaw) to address these issues. Modules group related packages, offering a more structured approach to application development. However, the adoption of the module system hasn’t been as widespread as initially anticipated.



2.2 Understanding Java 9 Modules

2.2.1 What is a Module?

In essence, a module is a grouping of related packages, resources (like images and XML files), and a module descriptor. This descriptor outlines:

- The module's name.
- The module's dependencies on other modules.
- The packages it explicitly makes available to other modules.
- The services it offers and consumes.
- The modules it grants reflective access to.

Modules provide a higher level of aggregation than packages, enhancing code organisation and maintainability.

2.2.2 Why Modules?

Before Java 9, the Java platform was essentially a monolithic entity. While there were attempts to modularise Java, none were widely adopted, and none successfully modularised the platform itself.

The lack of a formal module system led to issues like:

- Difficulties managing dependencies and version conflicts between JAR files.
- Limited control over the accessibility of internal APIs.
- Challenges in securing critical code and preventing access to internal APIs.

These issues, prevalent in larger applications, were often difficult to resolve.

2.2.3 Addressing the Challenges

The introduction of JPMS in Java 9 aimed to tackle these challenges. By dividing the JDK itself into modules, Java 9 facilitates various configurations. This modular JDK offers advantages like:

- **Reliable Configuration** Modules declare dependencies explicitly, enabling the JPMS to ensure all required modules are present during compilation and runtime. This mitigates the risk of runtime errors due to missing dependencies.
- **Strong Encapsulation** Modules control which packages are accessible to others. This fine-grained access control enhances security by restricting access to internal APIs, promoting better code maintainability and reducing the risk of unintended dependencies.
- **Improved Security** The module system enforces stricter access control to internal APIs. While this necessitated workarounds for some projects relying heavily on internal APIs, it ultimately enhances the platform's security.

2.2.4 The Unfulfilled Promise of JPMS

One of the most significant barriers to JPMS adoption has been its complexity. Developers accustomed to the classpath model found the transition to a modular system challenging. The introduction of new concepts like module descriptors (`module-info.java`) and the need to explicitly declare module dependencies added a layer of complexity to Java development.

```
module com.example.mymodule {
  requires java.base;
  requires com.example.anothermodule;
  exports com.example.mymodule.api;
}
```

This additional complexity was often seen as unnecessary for smaller projects, where the benefits of modularity were less apparent.

JPMS introduced strict encapsulation rules that broke compatibility with many existing libraries and frameworks. Many popular Java libraries were not initially designed with JPMS in mind, leading to conflicts when attempting to modularize applications that depended on these libraries.

For many existing Java projects, especially those with established architectures, the perceived benefits of adopting JPMS did not outweigh the costs of migration. The effort required to modularize a large, existing codebase was often deemed too high compared to the potential gains in maintainability or performance.

The Java ecosystem already had established solutions for achieving modularity, such as OSGi and Maven modules. Many developers and organizations had invested in these technologies and saw little reason to switch to JPMS, which offered similar benefits but required significant changes to their existing systems.

The Java community is now split between those who have adopted JPMS and those who continue to use the classpath model or alternative modularity solutions.

With fewer developers and projects using JPMS, the development of tools, best practices, and patterns for modular Java has been slower than initially anticipated.

Many projects continue to rely on the classpath model, potentially missing out on the benefits of stronger encapsulation and improved runtime optimizations.

Despite its challenges, JPMS remains an integral part of the Java platform. Its adoption may increase gradually as:

- Tools and IDEs improve their support for modular development
- More libraries and frameworks become JPMS-compatible
- Developers gain experience and become more comfortable with modular concepts
- New projects are started with JPMS in mind from the beginning

The Java Platform Module System represented a significant effort to modernize the Java platform and address long-standing issues with the classpath model. However, its limited adoption highlights the challenges of introducing fundamental changes to a mature ecosystem. As the Java community continues to evolve, the role of JPMS in shaping the future of Java development remains to be seen. Whether it will eventually gain widespread adoption or remain a niche feature used primarily in specific scenarios is a question that only time will answer.

2.3 Simplified Module Imports to the Rescue

One contributing factor to the slow adoption is the verbosity associated with importing multiple packages from modules. While modules aim to streamline dependency management, importing numerous packages from a module can lead to cluttered code with numerous import statements. This is particularly noticeable when working with modules that export many packages, such as `java.base`.

Simplified Module Imports aim to address this verbosity and simplify the use of modules. This feature enables developers to import all public packages exported by a module using a single statement. For instance, instead of writing multiple `import` statements for packages within the `java.base` module, developers can now use a single statement:

```
import module java.base;
```

This concise syntax offers several advantages:

- **Improved Code Readability:** By reducing the number of import statements, the code becomes cleaner and easier to read, especially when working with modules that export many packages.
- **Enhanced Developer Productivity:** Writing and managing imports become less tedious, allowing developers to focus more on application logic.
- **Simplified Adoption of Modules:** The simplified syntax lowers the entry barrier for developers hesitant to adopt modules due to the verbosity of traditional import statements.

2.4 Understanding the Mechanics

Let's examine how the `import module` statement functions within the context of module dependencies:

- **Transitive Dependencies:** When a module (`M1`) requires another module (`M2`) transitively, and a source file imports `M1` using `import module`, it automatically gains access to the public packages exported by both `M1` and `M2`.

- **Resolving Ambiguous Imports:** Importing multiple modules can lead to situations where classes with the same name exist in different imported packages, causing name ambiguity. The Java compiler identifies these ambiguities during compilation. Developers can resolve these ambiguities by using a single-type-import declaration to explicitly specify the desired class.

- **No Forced Modularization:** A key advantage of Simplified Module Imports is that it doesn't force developers to modularize their entire codebase. Developers can use this feature to import modules even if their code resides in the unnamed module (class path).

2.5 Impact on Different Developer Groups

This new feature caters to a wide range of Java developers:

- Simplifies the learning process for new Java developers. They no longer need to grasp the complexities of package hierarchies early on. They can directly access common classes from modules like `java.base` without numerous import statements.

- For seasoned developers working on large projects, Simplified Module Imports streamlines code, improves readability, and potentially reduces errors associated with managing multiple import statements.

- While large frameworks like Spring initially faced challenges in fully adopting the module system due to their reliance on reflection, the increasing modularization of libraries and the continuous development efforts, such as planned module system support in Spring, signal a potential rise in adoption, further benefiting from features like Simplified Module Imports.

2.6 Conclusion

As a preview feature, Simplified Module Imports is still under development. It's crucial to remember that preview features might change in future Java releases based on feedback and further development. However, given its potential to streamline Java development and promote the adoption of modules, Simplified Module Imports represents a significant step towards a more concise and developer-friendly Java language.

Chapter 3

Implicit Classes and Simplified Imports

Java, known for its robustness and scalability, has often been considered verbose, particularly for beginners. To address this and make the language more approachable, Java 23 introduces two key preview features working in tandem: **Implicitly Declared Classes** (JEP 477) and **Module Import Declarations** (JEP 476). This chapter explores these features, highlighting how they simplify Java development, especially for newcomers.

3.1 Implicitly Declared Classes: A Gentle Introduction

Traditionally, even a simple ”Hello, World!” program in Java required understanding classes, methods, and the static keyword, often leaving beginners puzzled. Implicitly Declared Classes aim to streamline this initial learning curve by allowing developers to omit explicit class declarations for simple programs.

Consider the classic ”Hello, World!” example:

```
public class HelloWorld {

  public static void main(String[] args) {

    System.out.println("Hello, World!");

  }
}
```

With **Implicitly Declared Classes**, this simplifies to:

```
void main() {

  println("Hello, World!");

}
```

The compiler now infers the existence of a class, removing the need for explicit class and `public static void main(String[] args)` declarations. This significantly reduces the cognitive load on beginners, allowing them to focus on core concepts like printing output without getting bogged down by syntax.

3.2 Seamless Console Interaction: The java.io.IO Class

To further simplify console input and output, JEP 477 introduces the ‘java.io.IO‘ class, providing convenient static methods like `println`, `print`, and `readln`. These methods are automatically imported into implicitly declared classes, making interaction with the console intuitive and concise.

Here’s an example demonstrating the use of `readln` for user input:

```
void main() {

  String name = readln("Enter your name: ");

  println("Welcome, " + name + "!");

}
```

Beginners can now easily write interactive programs without grappling with the complexities of `System.out` or `System.in`, fostering a smoother learning experience.

3.3 Streamlining Imports

As programs grow, they often require functionalities provided by different classes within the Java API. JEP 476 introduces the `import module` statement, allowing developers to import all public types from a module with a single line. This is particularly beneficial when working with modules containing numerous packages, like `java.base`.

For instance, to use the `List` interface from `java.util`, a traditional approach would involve:

```
import java.util.List;
class MyProgram {
  // ... code using the List interface
}
```

With JEP 476, this becomes:

```
import module java.base;
class MyProgram {
  // ... code using the List interface
}
```

The `import module java.base;` statement provides access to all public types within the ‘java.base‘ module, including `java.util.List`, eliminating the need for individual import statements.

3.4 A Synergistic Relationship for Simplicity

The true power of these features lies in their synergy. JEP 477 leverages JEP 476 to provide implicitly declared classes with automatic access to all public types within `java.base`. This means beginners can start using classes like `List`, `ArrayList`, and others from `java.base` without any explicit import statements, further reducing the learning curve.

Imagine a beginner wants to work with a list of names:

```
void main() {
  var names = List.of("Alice", "Bob", "Charlie");
  for (var name : names) {
    println("Hello, " + name + "!");
  }
}
```

This code works seamlessly within an implicitly declared class, showcasing how JEP 476 and JEP 477 work together to create a streamlined and beginner-friendly experience.

3.5 Conclusion: A More Approachable Java

Implicitly Declared Classes and **Simplified Module Imports** mark a significant step towards a more approachable Java for beginners. By minimizing boilerplate code and simplifying package access, these features allow new programmers to focus on core programming concepts, fostering a more engaging and enjoyable learning journey. As Java continues to evolve, these enhancements contribute to its relevance and appeal, ensuring it remains a powerful and accessible language for generations of developers to come.

Chapter 4

Structured Concurrency in Java

Concurrent programming, the art of making different parts of a program run seemingly simultaneously, is essential for building responsive and efficient applications. However, traditional approaches to concurrency, often relying on threads and locks, can be error-prone and difficult to debug. Java 23 introduces a powerful preview feature, **Structured Concurrency** (JEP 480), designed to simplify concurrent programming and improve the reliability of concurrent applications.

4.1 The Challenges of Unstructured Concurrency

Before exploring structured concurrency, let's understand the challenges posed by traditional or "unstructured" concurrency. Imagine a server application handling user requests. Each request might involve multiple tasks, such as fetching data from a database, processing that data, and sending a response back to the user. With unstructured concurrency using thread pools (`ExecutorService`), these tasks might be executed in separate threads. While this allows for parallelism, it introduces complexities:

- If one task encounters an error and fails to complete, other tasks might continue running in their threads, potentially leading to resource leaks if not managed properly.
- Interrupting or cancelling a group of related tasks running in different threads can be cumbersome. Ensuring that all tasks are properly stopped and resources are cleaned up adds to the complexity.
- Debugging and understanding the flow of execution in an application with multiple threads running independently can be challenging. Traditional debugging tools often don't provide a clear view of the relationships between tasks, making troubleshooting difficult.

4.2 Introducing Structured Concurrency

Structured concurrency addresses these challenges by enforcing a simple yet powerful principle:

> The lifetime of a concurrent subtask should be confined to the lexical scope of its parent task.

This means that if a task spawns subtasks to be executed concurrently, all those subtasks must complete before the parent task can finish. This principle brings structure to concurrency, making it easier to reason about, manage, and debug.

4.3 StructuredTaskScope: The Core of Structured Concurrency

The core of structured concurrency in Java is the `StructuredTaskScope` class. It acts like a container for a group of related tasks, ensuring their lifetimes are correctly managed. Let's illustrate with an example.

Assume we're building a weather application that fetches weather data from three different sources for a given location. We want to retrieve the data concurrently to improve responsiveness. Here's how we can do it using `StructuredTaskScope`:

```
package dev.petrucci.java23;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.StructuredTaskScope;

public class Main {

  static String fetchWeather(String source,
                             String location) {
    // Simulate network call
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }

    return " Weather data from "
                    + source + " for " + location;
  }

  public static void main(String[] args)
          throws ExecutionException, InterruptedException {
```

```
    try (var scope =
          new StructuredTaskScope.ShutdownOnFailure()) {

      var weatherFromSourceA = scope.fork(() ->
              fetchWeather(" Source A", " London "));

      var weatherFromSourceB = scope.fork(() ->
              fetchWeather(" Source B", " Tokyo "));

      var weatherFromSourceC = scope.fork(() ->
              fetchWeather(" Source C", " New York "));

      // Propagate if any task failed
      scope.join().throwIfFailed();

      System.out.println(weatherFromSourceA.get());
      System.out.println(weatherFromSourceB.get());
      System.out.println(weatherFromSourceC.get());
    }
  }
}
```

Let's break down this example:

1. **Creating the Scope:** We create a `StructuredTaskScope` using a try-with-resources block. This ensures that the scope is automatically closed, and any running tasks are cancelled when the block exits, even if exceptions occur.

2. **Forking Subtasks:** The `fork()` method submits a task (represented by a `Callable` or `Runnable`) to the scope for execution in a separate thread. Each `fork()` call returns a `Subtask` object, which represents the running subtask.

3. **Joining Subtasks:** The `join()` method waits for all subtasks to complete. This ensures that the main thread doesn't proceed until all weather data has been fetched. The `throwIfFailed()` method will rethrow the first exception encountered from any of the subtasks. This is a convenience method offered by the `ShutdownOnFailure` policy.

4. **Retrieving Results:** Once `join()` returns successfully, we can retrieve the results of each subtask using `Subtask.get()`.

4.4 Benefits of Structured Concurrency

The structured approach exemplified above provides several advantages:

- **Implicit Cancellation:** If any task within the `StructuredTaskScope` throws an exception, the other tasks are automatically cancelled. This prevents thread leaks and ensures a clean shutdown.
- **Error Handling:** Exceptions thrown within subtasks are propagated and can be handled by the parent task. This allows for centralised error handling and improves the robustness of the code.
- **Enhanced Observability:** The `StructuredTaskScope` maintains a hierarchical relationship between the parent task and its subtasks. This relationship can be observed through thread dumps and other debugging tools, making it easier to understand the flow of execution in concurrent code.

4.5 Conclusion

Structured concurrency in Java, represented by `StructuredTaskScope`, introduces a powerful mechanism for managing concurrency within well-defined lexical scopes. This approach simplifies error handling, cancellation, and debugging, making concurrent programming more manageable and less error-prone. As Java continues to evolve, structured concurrency is likely to become a cornerstone for building robust and efficient concurrent applications.

Chapter 5

Sharing Data Smartly in Java

Scoped Values (JEP 481), a feature introduced in Java 23, provides a safer and more efficient way to share data across different parts of a program, especially when working with concurrent threads. Scoped values are designed to address the limitations of traditional methods like thread-local variables, making data sharing more manageable, robust, and performant.

5.1 The Need for Efficient Data Sharing

Java applications and libraries are structured as collections of classes and methods. Methods communicate by passing data through method calls, which often involves passing the data as parameters. However, sometimes it becomes impractical to pass all the necessary data as parameters, especially when dealing with complex call chains or when the data is meant for internal framework use and irrelevant to application code.

For instance, imagine a web framework handling an HTTP request. The framework might need to access internal context data, like the authenticated user ID or transaction ID, across different methods in the call chain. Passing this context data as a

parameter to every method would be cumbersome and could introduce potential errors.

5.2 Limitations of Thread-Local Variables

Traditionally, developers have relied on **thread-local variables** to share data without resorting to method parameters. Thread-local variables maintain a separate value for each thread, avoiding the need to pass data explicitly. However, thread-local variables come with inherent limitations:

- Thread-local variables can be modified by any code that can access them, making data flow difficult to track and control.
- The values stored in thread-local variables persist until the thread exits or the value is explicitly removed, potentially leading to resource leaks and security vulnerabilities if not managed carefully.
- When a thread creates child threads, the child threads inherit the thread-local variables of the parent thread, potentially leading to significant memory overhead, especially with large numbers of virtual threads.

5.3 Scoped Values: A More Efficient Solution

Scoped values address the drawbacks of thread-local variables by providing a more controlled and efficient mechanism for sharing data. They are essentially containers that allow a method to share data with its direct and indirect callees within the same thread and with child threads, without relying on method parameters.

Scoped values offer these advantages:

- **Bounded Lifetime:** The data associated with a scoped value is only accessible within the scope of the method that bound it. This ensures that the data is not accessible outside its intended context, preventing unintended modifications or leaks.
- **Immutability:** The values stored in scoped values are typically immutable, ensuring that the data shared is consistent and reliable.
- **Efficient Inheritance:** Scoped values can be inherited by child threads without copying the data from the parent thread, minimizing memory overhead.
- **Easy to Use:** The `ScopedValue` API provides methods for creating and managing scoped values, with options for binding multiple values at a call site.

5.4 Practical Example

Let's illustrate the use of scoped values with an example:

```
package dev.petrucci.java23;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.StructuredTaskScope;

public class WeatherApp {

  private static final ScopedValue<String> Location =
                                ScopedValue.newInstance();

  static String fetchWeather(String source) {

    // Simulate network call
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }

    return "Weather data from " + source + " for " +
                                          Location.get();
```

```
  }

  public static void main(String[] args) throws
  ExecutionException, InterruptedException {

    ScopedValue.runWhere(Location, "London", () -> {

      try (var scope = new StructuredTaskScope
      .ShutdownOnFailure()) {

        var weatherFromSourceA = scope.fork(() ->
                fetchWeather("Source A"));

        var weatherFromSourceB = scope.fork(() ->
                fetchWeather("Source B"));

        var weatherFromSourceC = scope.fork(() ->
                fetchWeather("Source C"));

        try {
          scope.join().throwIfFailed();
        } catch (Exception e) {
          System.err.println("Failed to fetch weather data");
          throw new RuntimeException(e);
        }

        System.out.println(weatherFromSourceA.get());
        System.out.println(weatherFromSourceB.get());
        System.out.println(weatherFromSourceC.get());


      }
    });
  }
}
```

In this example:

1. **Declaring a Scoped Value:** We declare a scoped value named `Location` using `ScopedValue.newInstance()`.

2. **Binding the Scoped Value:** We use `ScopedValue.runWhere()` to bind the `Location` scoped value to the ”London” string. This creates a scope where all methods called within this scope can access the bound value of ”London”

using `Location.get()`.

3. **Using the Scoped Value:** The `fetchWeather()` method retrieves weather data from different sources. Each call to `fetchWeather()` uses `Location.get()` to access the value bound to the `Location` scoped value, which is ”London” in this case.

4. **Structured Concurrency:** We use a `StructuredTaskScope` to execute the `fetchWeather()` calls concurrently.

5. **Automatic Cleanup:** Once the `runWhere()` block finishes, the binding for the `Location` scoped value is automatically removed.

5.5 Conclusion

Scoped values provide a valuable addition to Java’s concurrency features. They offer a safer, more efficient, and easier-to-use mechanism for sharing data across threads and different parts of a program. As Java continues to evolve, scoped values are likely to become a crucial tool for building robust, efficient, and reliable concurrent applications.

Chapter 6

Flexible Constructor Bodies

Java 23 introduces a significant shift in how constructors are structured, offering developers more flexibility in writing code within a constructor body. This chapter explores the concept of **Flexible Constructor Bodies**, explaining its purpose, benefits, and the specific rules governing its use.

6.1 The Motivation for Change

Prior to Java 23, the language imposed a strict requirement: if a constructor explicitly invoked another constructor (using `this(...)` or `super(...)`), that invocation had to be the very first statement within the constructor body. This constraint, while ensuring a predictable constructor execution order, often led to less readable and maintainable code. Developers were forced to employ workarounds like auxiliary methods or constructors to perform tasks that logically should reside within the constructor itself.

6.2 Introducing Flexible Constructor Bodies

JEP 482, **Flexible Constructor Bodies**, directly addresses this limitation. It allows developers to include statements **before** an explicit constructor invocation (`super(...)` or `this(...)`) within a constructor body. These statements, referred to as the ”prologue” of the constructor, expand the possibilities for what can be achieved before control is passed to another constructor.

6.3 Benefits of Flexibility

This newfound flexibility brings several advantages:

- Code that logically belongs within a constructor can now be placed there directly, improving readability and maintainability.
- Validation, preparation, and sharing of arguments passed to other constructors can be done more cleanly within the constructor itself, reducing the need for auxiliary methods or constructors.
- Subclasses gain the ability to initialize their fields before invoking superclass constructors, mitigating potential issues where superclass code might encounter uninitialized subclass fields.

6.4 Rules and Restrictions

While offering greater flexibility, JEP 482 introduces a new concept – the **early construction context.** This context encompasses both the argument list of an explicit constructor invocation and any statements within the prologue.

Here are the key rules governing code within an early construction context:

- Code in the prologue cannot use `this` to refer to the current instance, access its fields, or invoke its methods. This restriction ensures that superclass constructors operate on a consistently initialized object.

- A key exception to the previous rule is that fields declared within the same class as the constructor, and lacking explicit initializers, can be assigned values within the prologue. This enables subclasses to initialize their fields before a superclass constructor might access them.

- Calling instance methods (including those inherited from superclasses) is prohibited within the prologue. This restriction prevents scenarios where instance methods might rely on object state that hasn't been fully initialized.

- In the case of nested classes, code within the prologue of an inner class constructor can access the enclosing instance and its members. This is permitted because the enclosing instance is created before the inner class instance.

6.5 Impact on Records and Enums

The changes introduced by JEP 482 also benefit constructors of record classes and enum classes. Non-canonical record constructors, which use `this(...)` to invoke alternative constructors, can now contain statements in their prologue. Similarly, enum constructors, which might use `this(...)` to call other enum constructors, gain the same flexibility.

6.6 Examples of Flexible Constructor Bodies

6.6.1 Validating Superclass Constructor Arguments

Before Java 23, validating an argument passed to a superclass constructor had to be done after invoking `super(...)`, potentially leading to unnecessary work if the validation failed.

```
public class PositiveBigInteger extends BigInteger {

  public PositiveBigInteger(long value) {

    // Could be unnecessary if validation fails
    super(value);

    if (value <= 0) {
      throw new IllegalArgumentException
                              ("Value must be positive");
    }
  }
}
```

With **Flexible Constructor Bodies**, validation can be performed in the prologue, leading to a more efficient ”fail-fast” behaviour:

```
public class PositiveBigInteger extends BigInteger {

  public PositiveBigInteger(long value) {

    if (value <= 0) {
      throw new IllegalArgumentException
                              ("Value must be positive");
    }

    super(value);

  }
}
```

6.6.2 Preparing Superclass Constructor Arguments

Previously, complex calculations for superclass constructor arguments often involved auxiliary methods. Flexible Constructor Bodies allow for this preparation within the constructor itself.

```
public class Sub extends Super {

  public Sub(Certificate certificate) {

    var publicKey = certificate.getPublicKey();

    if (publicKey == null) {
      throw new IllegalArgumentException
                    ("Certificate missing public key");
    }

    byte[] certBytes = switch (publicKey) {
      case RSAKey rsaKey -> ...
      case DSAPublicKey dsaKey -> ...
      default -> ...
    };

    super(certBytes);

  }
}
```

6.6.3 Initializing Subclass Fields to Avoid Issues with Overridden Methods

Although discouraged, superclass constructors might call methods that are overridden in subclasses. This can lead to unexpected behaviour if the overridden method in the subclass relies on fields not yet initialized at the time of the superclass constructor call.

```
class Super {

  Super() { overriddenMethod(); }
```

```
  void overriddenMethod() {
    System.out.println("Super: hello");
  }

}


class Sub extends Super {

  final int x;

  Sub(int x) {
    // Initialize before superclass constructor
    this.x = x;
    super();
  }

  @Override
  void overriddenMethod() {
    System.out.println("Sub: " + x);
  }

}
```

In this example, initializing `this.x` before `super()` ensures that the overridden `overriddenMethod` in the subclass accesses the correct value of `x`.

These examples showcase how **Flexible Constructor Bodies** enhance code clarity and maintainability, allowing for more intuitive and efficient constructor implementations.

6.7 Conclusion

Flexible constructor bodies in Java 23 mark a significant evolution in how constructors are designed. The ability to write code before explicit constructor invocations empowers developers to write clearer, more maintainable, and robust code, particularly in scenarios involving subclassing and complex constructor logic. Understanding the concept of the ”early construction context” and its associated rules is crucial to harnessing the full potential of this new feature.

Chapter 7

Markdown Documentation Comments

Java 23 introduces a significant enhancement to how developers write documentation comments within their code: the ability to use Markdown syntax. JEP 467, **Markdown Documentation Comments**, brings the simplicity and readability of Markdown to Java documentation, making it easier for developers to create and maintain high-quality API documentation.

7.1 The Need for Modernization

Since the inception of Java, documentation comments have relied on a combination of HTML tags and custom JavaDoc tags (e.g., `@param`, `@return`). While HTML was a reasonable choice in 1995, its complexity as a manually written markup language has become increasingly apparent over time.

Modern developers often find HTML tedious to write and read, hindering the creation of well-formatted documentation. Moreover, the extensive use of JavaDoc tags, while powerful, can be cumbersome and less intuitive, especially for those new to the language.

7.2 Markdown: A More Human-Friendly Approach

Markdown has emerged as a popular alternative for creating simple, yet well-structured documents. Its lightweight syntax focuses on readability and ease of writing, making it a natural fit for documentation comments, which are typically less complex than full-fledged web pages.

The key benefit of Markdown is its ability to express common formatting elements, like headings, lists, and links, using simple, intuitive characters. This simplicity reduces the visual clutter of HTML tags, making the documentation easier to read directly in the source code.

7.3 Best of Both Worlds: Markdown and JavaDoc

JEP 467 is designed to leverage the strengths of both Markdown and the existing JavaDoc system. It introduces a new documentation comment style using `///` at the beginning of each line, signaling the use of Markdown within the comment block.

The goal is not to replace HTML and JavaDoc tags entirely, but to provide a more streamlined approach for common formatting tasks. Developers can still use HTML tags for elements not directly supported by Markdown, and JavaDoc tags remain essential for generating API documentation structure and linking.

7.4 Impact and Future Directions

JEP 467 represents a significant step towards making Java documentation more approachable and maintainable. By adopting Markdown, Java embraces a widely used standard, potentially encouraging more developers to contribute to documentation efforts.

While the initial release focuses on incorporating basic Markdown elements, there are opportunities for future enhancements. These might include support for more advanced Markdown features or tools to help migrate existing JavaDoc comments to the Markdown format.

It's important to note that JEP 467 doesn't mandate the conversion of existing documentation; it provides an alternative that developers can adopt at their own pace.

7.5 Examples of Markdown Documentation Comments

Let's provide concrete examples to solidify our understanding.

Basic Formatting with Markdown

Imagine a simple utility class with a method for calculating the factorial of a number:

```
/// This class provides utility methods for
/// mathematical operations.
```

```
public class MathUtils {

  /// Calculates the factorial of a non-negative integer.
  ///
  /// For example:
  /// - `factorial(0)` returns `1`
  /// - `factorial(5)` returns `120`
  ///
  /// @param n The non-negative integer for which to
  ///     calculate the factorial.
  /// @return The factorial of `n`.
  /// @throws IllegalArgumentException if `n` is negative.
  public static long factorial(int n) {

    if (n < 0) {
      throw new IllegalArgumentException
                          ("Input must be non-negative.");
    }

    if (n == 0) {
      return 1;
    } else {
      return n * factorial(n - 1);
    }

  }
}
```

In this example:

- We use '///' to denote Markdown documentation comments.
- Paragraphs are separated by a blank line.
- A code snippet (`factorial(0)`) is enclosed in backticks.
- An unordered list is created using the '-' character.
- JavaDoc tags like `@param`, `@return`, and `@throws` are used alongside Markdown.

Linking to Other Classes and Methods

Consider a scenario where you want to reference another class or method within your documentation:

```
/// Represents a connection to a remote server.
///
/// This class utilizes the [NetworkManager]
/// for establishing and managing network connections.
///
/// See also: [NetworkManager#connect(String)]
public class ServerConnection {
  // ... class implementation ...
}
```

Here:

- `NetworkManager` creates a link to the `NetworkManager` class documentation.
- `NetworkManager.connect(String)` creates a link to the `connect(String)` method within the `NetworkManager` class.

These examples highlight how JEP 467 integrates Markdown's simplicity with the existing JavaDoc infrastructure, resulting in cleaner, more readable documentation comments directly within the Java source code.

Chapter 8

ZGC: Generational Mode by Default

Java 23 solidifies the role of the **Z Garbage Collector (ZGC)** as a high-performance, low-latency option by making its generational mode the default. This chapter explores JEP 474, which ushers in this change, and discusses its implications for Java developers.

8.1 Understanding ZGC and Generational Garbage Collection

ZGC, introduced in earlier Java versions, addresses the increasing demand for predictable low latency in garbage collection, particularly for applications with large heaps. Its design aims to minimize pauses caused by garbage collection, making it suitable for latency-sensitive workloads.

Generational garbage collection, a concept well-established in Java, is based on the observation that most objects in a program have a short lifespan. This ”generational hypothesis” is exploited by dividing the heap into generations:

- Where new objects are allocated (**Young Generation**).
- Where long-lived objects that survive young generation collections are promoted (**Old Generation**).

By focusing collection efforts primarily on the young generation, where most garbage is generated, generational garbage collectors can achieve higher efficiency and reduce pause times.

8.2 Shifting the Default

Prior to Java 23, ZGC operated in a non-generational mode by default. While this mode provided the core benefits of ZGC’s low-latency design, it didn’t fully leverage the advantages of generational garbage collection. JEP 474 changes this by making generational ZGC the default mode.

This shift signals a clear direction for ZGC’s future development. By prioritizing the generational mode, Oracle aims to streamline development efforts and optimize ZGC’s performance for a broader range of applications.

8.3 Benefits of Generational ZGC

The transition to generational ZGC as the default offers several benefits.

Focusing collection efforts on the young generation typically results in less overall work for the garbage collector, leading to lower CPU usage. With reduced CPU overhead and shorter pause times, applications can process more requests, leading to increased throughput. Generational ZGC relies more heavily on automatic tuning mechanisms, reducing the need for manual configuration and making it easier for developers to adopt.

8.4 Illustrating the Impact

We can illustrate the potential impact of this feature through a few scenarios:

Example 1: A Caching Service

Imagine a caching service that frequently creates short-lived objects to store cached data. In a non-generational ZGC environment, these objects, even with their short lifespan, would be subject to the full garbage collection cycle.

With generational ZGC, these short-lived objects would be allocated in the young generation and quickly reclaimed, minimizing their impact on garbage collection pause times and overall CPU usage.

Example 2: A High-Frequency Trading Application

In a high-frequency trading application, where microsecond latency can be critical, minimizing garbage collection pauses is paramount. Generational ZGC, by efficiently collecting short-lived objects, can contribute to maintaining consistently low latency for trade processing.

8.5 Considerations and Potential Challenges

While generational ZGC is expected to be beneficial for most applications, there are a few considerations:

- **Workload Characteristics:** Applications with unusual object lifecycles, where objects don't strictly adhere to the generational hypothesis, might require careful tuning.
- **Legacy Configurations:** Existing applications with ZGC configurations tuned for the non-generational mode might need adjustments to fully leverage the generational mode.

8.6 Conclusion

JEP 474 marks a significant milestone in ZGC's evolution, establishing its generational mode as the default and paving the way for future optimizations. By embracing generational garbage collection, ZGC becomes an even more compelling option for a wider range of Java applications, particularly those demanding high performance and low latency.

Chapter 9

Removed Features and Options

This chapter details the features and options removed in Java 23.

9.1 Aligned Access Modes for MethodHandles

The atomic access modes for `MethodHandles::byteArrayViewVarHandle` and `MethodHandles::byteBufferViewVarHandle` (when accessing heap buffers) have been removed. This change is based on the fact that the removed functionality relied on an implementation detail in the reference JVM implementation that wasn't part of the JVM specification.

The following methods are affected by this change:

- `MethodHandles::byteArrayViewVarHandle`
- `MethodHandles::byteBufferViewVarHandle`
- `ByteBuffer::alignedSlice`
- `ByteBuffer::alignmentOffset`

The `ByteBuffer::alignedSlice` and `ByteBuffer::alignmentOffset` methods

will now throw an `UnsupportedOperationException` for heap byte buffers when `unitSize` is greater than 1.

Recommended Alternatives:

- Utilise direct (off-heap) byte buffers for reliable aligned access.
- Store data in a `long[]` array, which offers stronger alignment guarantees compared to `byte[]`.
- Use a `MemorySegment` backed by a `long[]` array for atomic access with any primitive type via the Foreign Function and Memory API:

```
long[] arr = new long;

MemorySegment arrSeg = MemorySegment.ofArray(arr);

// accessing aligned ints
VarHandle vh = ValueLayout.JAVA_INT.varHandle();

// 0L is offset in bytes
vh.setVolatile(arrSeg, 0L, 42);

long result = vh.getVolatile(arrSeg, 0L); // 42
```

9.2 Thread Management Methods

Several thread management methods, known for potential deadlocks, have been removed in this release:

- `java.lang.ThreadGroup.stop()`
- `java.lang.Thread.suspend()`
- `java.lang.Thread.resume()`
- `java.lang.ThreadGroup.suspend()`

- `java.lang.ThreadGroup.resume()`

These methods were deprecated in earlier Java versions and now result in a `NoSuchMethodError` if code compiled against older releases is executed on JDK 23 or newer.

9.3 Module `jdk.random`

The `jdk.random` module, containing implementations of `java.util.random.RandomGenerator` algorithms, has been removed. These implementations are now part of the `java.base` module.

Applications referencing `jdk.random` in build scripts or module dependencies should remove these references.

9.4 Legacy Locale Data

The legacy JRE locale data (including its alias, COMPAT) has been removed. This legacy data was superseded by CLDR (Common Locale Data Registry) locale data, which became the default in JDK 9.

Applications using JRE/COMPAT locale data should migrate to CLDR locale data or consider workarounds described in JEP 252.

9.5 JMX Subject Delegation

In preparation for the removal of the Security Manager, JMX Subject Delegation has been removed. Calling `javax.management.remote.JMXConnector.getMBeanServerConnection (Subject delegationSubject)` with a non-null delegation subject will throw an `UnsupportedOperationException`.

To operate with multiple identities, client applications now need multiple calls to `JMXConnectorFactory.connect()` and the `getMBeanServerConnection()` method on the returned `JMXConnector`.

Refer to the Security in Java Management Extensions Guide for more information.

9.6 JMX Management Applet (m-let)

The m-let feature, part of the effort to prepare for the removal of the Security Manager, has been removed. This removal doesn't affect the JMX agent for local/remote monitoring, JVM instrumentation, or JMX-based tooling.

The following API classes are removed:

- `javax.management.loading.MLet`
- `javax.management.loading.MLetContent`
- `javax.management.loading.PrivateMLet`
- `javax.management.loading.MLetMBean`

9.7 RegisterFinalizersAtInit Option

The HotSpot VM option `-XX:[+-]RegisterFinalizersAtInit`, deprecated in JDK 22, is now obsolete.

9.8 -Xnoagent Option for the `java` Launcher

The `-Xnoagent` option for the `java` launcher, deprecated in a previous release, has been removed. Using this option will now result in an error, preventing the process launch. Applications using this option should remove it.

9.9 Obsolete Desktop Integration from Linux Installers

Linux installers will no longer deposit files in `/usr/share/icons`, `/usr/share/mime`, and `/usr/share/applications` subtrees.

Chapter 10

Deprecated Features and Options

This chapter details the features and options deprecated in Java 23.

10.1 java.beans.beancontext Package

The `java.beans.beancontext.*` package, introduced in JDK 1.2, has been deprecated for removal in a future release. This package, based on concepts from Apple Computer's OpenDoc, aimed to provide mechanisms for assembling JavaBeans components into hierarchies, enabling them to interact through services expressed as interfaces.

However, with advancements in the Java language, including annotations, lambdas, modules, and programming paradigms like "Declarative Configuration", "Dependency Injection", and "Inversion of Control", the APIs within the `java.beans.beancontext.*` package have become obsolete and represent an outdated approach to component assembly.

Developers are strongly advised against using these APIs and should plan to migrate existing code reliant on this package to alternative solutions in anticipation of its removal.

10.2 JVM TI GetObjectMonitorUsage Function

The JVM Tool Interface (JVM TI) function `GetObjectMonitorUsage` has been modified and no longer returns monitor information for virtual threads. This function now only provides monitor owner details when the monitor is owned by a platform thread.

The arrays returned by this function, representing threads waiting to own the monitor and threads waiting for notification, are now also restricted to platform threads.

This change also affects the corresponding JDWP command `ObjectReference.MonitorInfo`, and the methods `owningThread()`, `waitingThreads()`, and `entryCount()` defined in the `com.sun.jdi.ObjectReference` class.

10.3 DontYieldALot Flag

The undocumented `DontYieldALot` product flag, initially introduced to address a scheduling issue specific to the Solaris operating system, has been deprecated. This flag has been unnecessary for an extended period and hasn't functioned as intended for many years.

Marked as deprecated, the flag is scheduled for obsolescence and subsequent removal in future releases.

10.4 -XX:+UseEmptySlotsInSupers

The `-XX:+UseEmptySlotsInSupers` option, which controls the allocation of fields in superclasses during field layout, has been deprecated in JDK 23 and is set to become obsolete in JDK 24. The default value remains "true," signifying that the HotSpot JVM will consistently allocate fields in a superclass if aligned space is available.

Developers are advised to note that code relying on the specific positioning of instance fields should consider this detail of instance field layout. It is essential to remember that the JVM field layout format is not part of the JVM Specification and may be subject to change.

10.5 PreserveAllAnnotations VM Option

The VM option `PreserveAllAnnotations` has been deprecated and will produce a warning message when used. Introduced primarily for testing Java Annotation code, this option, always disabled by default, is slated for obsolescence and eventual removal in future releases.

10.6 UseNotificationThread VM Option

The VM option `UseNotificationThread`, responsible for switching debug notification delivery from the hidden ”Service Thread” to the non-hidden ”Notification Thread”, has been deprecated.

With this option (defaulting to `true`) intended to address potential issues arising from the use of the ”Notification Thread,” the absence of reported problems has led to the decision to make the ”Notification Thread” the sole method for sending notifications in the future. Consequently, the `UseNotificationThread` option will be obsoleted and eventually removed.

Chapter 11

Notable Issues Fixed

This chapter details notable issues addressed in Java 23, enhancing the platform's stability, security, and performance.

11.1 Packaging and Deployment

11.1.1 Accurate Package Listing with jpackage on Debian

Previous versions of `jpackage` encountered difficulties when determining the complete set of required packages on Debian-based Linux distributions. Specifically, issues arose when shared libraries referenced through symbolic links were involved. This led to incomplete package lists during the creation of installation packages, ultimately causing installations to fail due to missing shared library dependencies.

Java 23 rectifies this problem, ensuring that `jpackage` now accurately identifies and includes all necessary packages, even when symbolic links are part of the shared library paths. This enhancement improves the reliability of `jpackage` on Debian-based systems.

11.2 Core Libraries and APIs

11.2.1 Enhanced HTTP Server Responsiveness

The behaviour of the built-in HTTP server within Java's core libraries has been adjusted to improve response times. Previously, when chunked transfer encoding was used or when a response body was present, the HTTP server would immediately send response headers. This behaviour, however, could lead to delays in certain operating systems due to delayed acknowledgments.

In Java 23, the HTTP server now buffers the response headers and transmits them along with the response body if one is expected. This modification is designed to bring about performance improvements, particularly for responses with bodies or those employing chunked transfer encoding. It's important to note that it is now recommended to consistently close the HTTP exchange or response body stream to explicitly trigger the sending of response headers, especially in cases where no response body is present.

11.2.2 DecimalFormat Pattern String Memory Usage

Addressing a potential memory exhaustion issue, Java 23 modifies the behaviour of `java.text.DecimalFormat` when dealing with empty pattern strings. Prior to this change, invoking `DecimalFormat.getMaximumFractionDigits()` on a `DecimalFormat` instance initialised with an empty pattern string would return `Integer.MAX_VALUE`. This could lead to an `OutOfMemoryError` when `DecimalFormat.toPattern()` was subsequently called.

To mitigate this risk, Java 23 adjusts the return value of `DecimalFormat.getMaximumFractionDigits()` in this specific scenario to 340. This alteration prevents excessive memory allocation while still accommodating a wide

range of practical use cases. For situations demanding a higher maximum fraction digit count exceeding 340, developers are advised to explicitly set this value using the `DecimalFormat.setMaximumFractionDigits()` method.

11.2.3 MessageFormat Pattern String Quoting

This release fixes an issue in `java.text.MessageFormat` concerning the accurate representation of quoted curly braces within nested subformat patterns. Previously, the `MessageFormat.toPattern()` method, responsible for generating a pattern string equivalent to the original, might have incorrectly omitted quoting for opening or closing curly braces intended as literal text within nested subformats.

This omission could lead to parsing errors when creating a new `MessageFormat` from the generated pattern string, potentially throwing an exception or resulting in a different interpretation of the pattern. Java 23 rectifies this behaviour, ensuring that quoted curly braces in nested subformat patterns are correctly preserved during the pattern string generation by `MessageFormat.toPattern()`. This correction maintains the consistency and predictability of `MessageFormat` patterns.

11.2.4 Lenient Date/Time Parsing and Space Separators

Java 23 brings enhancements to the parsing of date and time strings, particularly in addressing compatibility concerns stemming from changes introduced in JDK 20 with the adoption of CLDR version 42. The update in CLDR replaced standard ASCII spaces (U+0020) with Narrow No-Break Spaces (NNBSP, U+202F) in specific locales, causing potential parsing discrepancies.

To accommodate this, Java 23 introduces a ”loose matching” behaviour for space characters during date and time string parsing. This lenient approach treats different space characters, including ASCII spaces and NNBSP, interchangeably.

It's important to note that this "loose matching" is active only when using the "lenient" parsing style, available in both `java.time.format` and `java.text` packages.

In `java.time.format`, developers need to explicitly enable lenient parsing using `DateTimeFormatterBuilder.parseLenient()` because the default mode remains strict. Conversely, in `java.text`, lenient parsing is the default; developers requiring strict parsing need to explicitly disable leniency by setting `DateFormat.setLenient(false)`. This enhancement provides flexibility in handling space characters during date and time parsing, improving compatibility and reducing potential parsing errors.

11.3 HotSpot Virtual Machine

11.3.1 Standardised Naming for Filler Array Objects

In previous releases, the HotSpot JVM used an internal class named `jdk.vm.internal.FillerArray` to represent areas of unreachable memory within the heap. This naming convention, however, posed problems for external tools like `jmap -histo` that parsed heap histograms. The non-array-like name contradicted the object's actual behaviour, which represented a range of memory, leading to confusion when calculating instance sizes.

Java 23 addresses this by renaming the class to `[Ljdk/internal/vm/FillerElement;`, aligning it with the standard Java class naming convention for arrays. This change makes the output of tools like `jmap -histo` more consistent and easier to parse for external applications analysing heap information.

11.3.2 Enhanced G1 Garbage Collector: Marking Stack Expansion

The G1 garbage collector in Java 23 benefits from an enhancement allowing for the expansion of the marking stack during the Reference Processing phase of garbage

collection. Previously, the marking stack size could only grow during the Concurrent Mark phase. This limitation could lead to premature `StackOverflowError` exceptions during Reference Processing if the stack reached its limit before fully processing all references.

With this enhancement, G1 can now expand the marking stack during Reference Processing, preventing these premature overflows and allowing the garbage collector to fully complete its tasks. This improvement contributes to the robustness and efficiency of the G1 garbage collector, especially in scenarios with a large number of object references.

11.3.3 Improved Startup Performance with JFR

Java 23 addresses a performance regression observed in earlier versions when using the Java Flight Recorder (JFR) with the `-XX:StartFlightRecording` option. The regression, noticeable in smaller applications, was attributed to technical debt in the JFR bytecode instrumentation, leading to increased startup times compared to JDK 21.

This release resolves the underlying issues, bringing startup times back to levels comparable to JDK 21. This improvement is particularly beneficial for short-lived applications where startup time is a significant performance factor.

11.4 JVM Tool Interface (JVMTI)

11.4.1 Clarification of Contended Monitor Status

Java 23 brings important clarifications and consistency improvements to how ”contended monitor” status is reported through the JVM Tool Interface (JVMTI), the Java Debug Wire Protocol (JDWP), and the Java Debug Interface (JDI).

Previously, inconsistencies existed in the specification and implementation of `GetCurrentContendedMonitor` in JVMTI and its corresponding counterparts in JDWP and JDI. These inconsistencies centred around whether a thread waiting in `java.lang.Object.wait()` should be considered as contending for the monitor.

Java 23 aligns the behaviour across JVMTI, JDWP, and JDI, adhering more strictly to the specification. A monitor is now considered contended only if a thread is actively waiting to enter or re-enter a monitor (for example, waiting on a synchronized block or method). A thread in `Object.wait()`, while technically associated with the monitor, is not considered actively contending for it.

This clarification ensures consistent reporting of monitor contention status across different debugging and monitoring tools, providing developers with a more accurate view of thread synchronization and potential bottlenecks.

11.4.2 Accurate Reporting of Waiting Threads

This release corrects an inaccuracy in the JVMTI function `GetObjectMonitorUsage`, specifically related to the reporting of threads waiting on a monitor. Previously, this function, designed to provide information about a monitor's usage, incorrectly included threads waiting in `java.lang.Object.wait()` within the list of threads waiting to acquire the monitor.

Java 23 addresses this issue, ensuring that `GetObjectMonitorUsage` now accurately reports only those threads that are actively waiting to enter or re-enter the monitor, excluding threads in the `Object.wait()` state.

A similar clarification has been applied to the corresponding JDWP command, `ObjectReference.MonitorInfo`. Although the command's behaviour remains unchanged (still including waiting threads from `Object.wait()`), the specification now explicitly clarifies what constitutes "waiting threads," making it clear that this count includes threads both actively contending for the monitor and those parked in `Object.wait()`.

This change promotes consistency between the specification and behaviour of these diagnostic tools.

11.5 Compiler and Tools

11.5.1 Enclosing Instances and Local Classes

Java 23 enforces stricter adherence to the *Java Language Specification (JLS)* regarding local classes declared within specific contexts. According to JLS 21 §15.9.2, local and anonymous classes defined within a static context, which includes parameters of superclass or this-class constructor invocations, are not permitted to have immediately enclosing istances. However, there was an inconsistency where the compiler allowed local classes, but not anonymous classes, to reference the enclosing instance within superclass/this-class constructor parameters, even though such references would invariably lead to errors during instantiation.

In Java 23, the compiler correctly prohibits local classes from containing references to the enclosing instance when declared within superclass or this-class constructor parameters, aligning the behaviour with anonymous classes and adhering to the JLS specification. While this scenario might not be common, the change eliminates a source of potential confusion and ensures consistent behaviour across different class declaration types.

11.5.2 Javadoc Member Reference Validation

The javadoc tool in this release includes a fix to enhance the validation of member references within documentation comments. Previously, `@see` and `{@link...}` tags permitted using a nested class to qualify the name of a member belonging to its enclosing class. This behaviour, however, was incorrect and could lead to

inaccurate or misleading documentation links.

Java 23 corrects this behaviour, ensuring that `@see` and `{@link...}` tags now require the correct class name when referencing members. Using a nested class to qualify a member of its enclosing class will now result in a warning or error during javadoc processing. This change improves the accuracy and reliability of javadoc-generated documentation.

Index