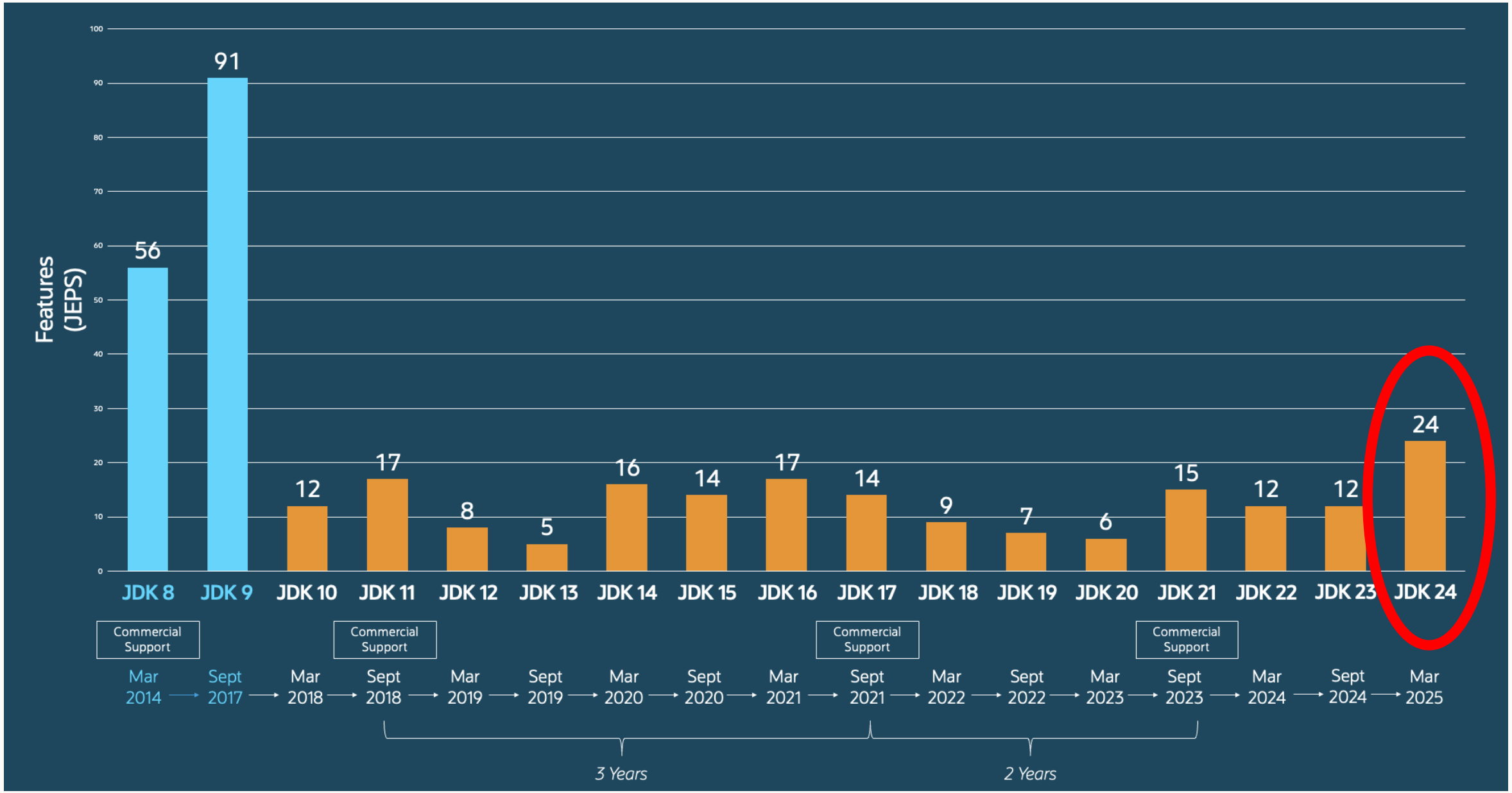
ORACLE

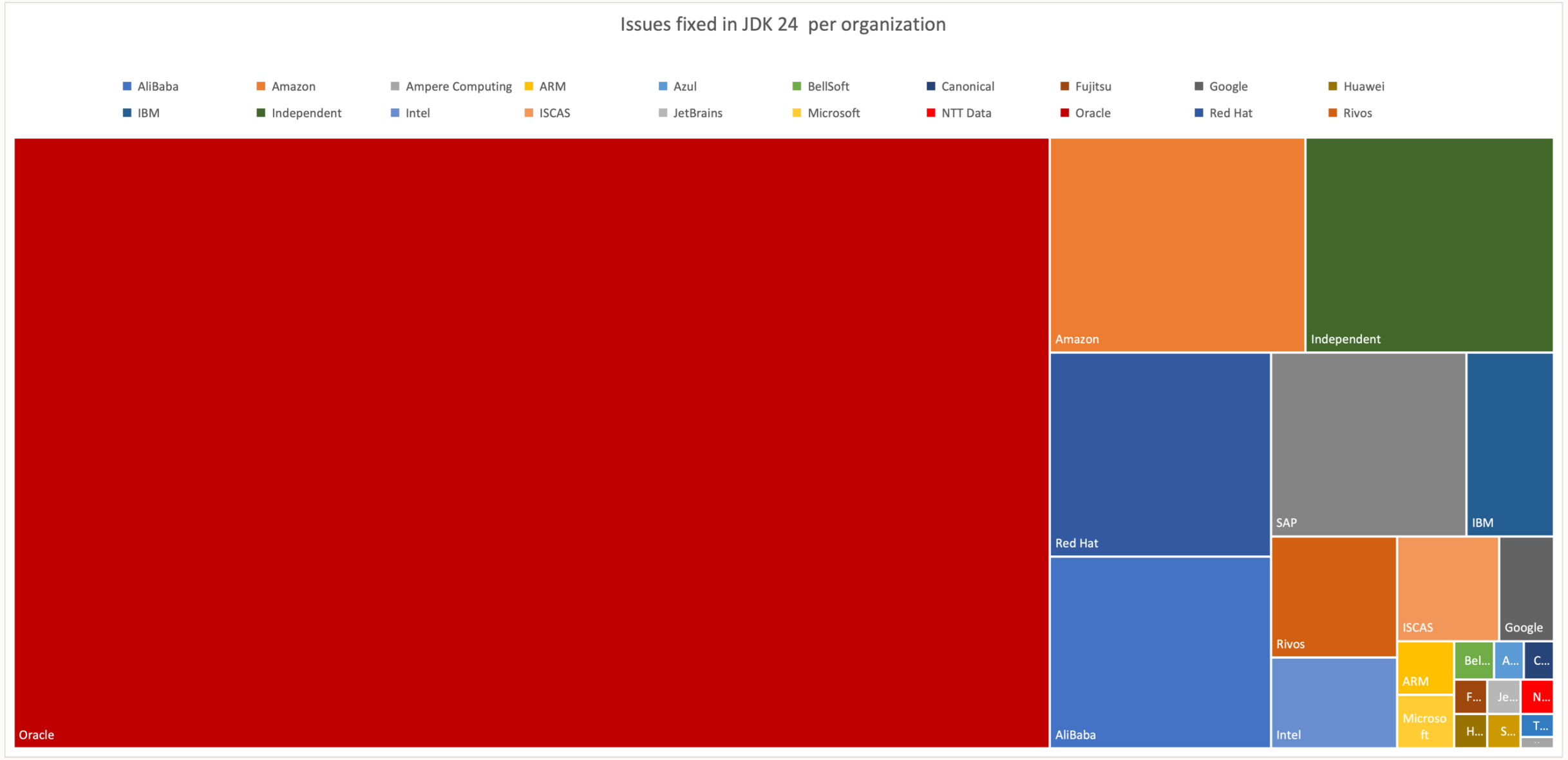
Java™

Java 24

Alex Buckley
Java Platform Group

March 2025





JDK 24 (March 2025)

Language Previews

- Primitive Types in Patterns, `instanceof`, `switch` (488)
- Flexible Constructor Bodies (492)
- Simple Source Files + Instance Main Methods (495)
- Module Import Declarations (494)

Libraries

- Class-File API (484)
- Stream Gatherers (485)
- Structured Concurrency 4th Preview (499)
- Scoped Values 4th Preview (487)
- Vector API 9th Incubator (489)

Security Libraries

- Key Derivation Function API Preview (478)
- Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism (496)
- Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm (497)

Performance and Runtime

- Synchronize Virtual Threads without Pinning (491)
- Compact Object Headers Experimental (450)
- Late Barrier Expansion for G1 (475)
- Ahead-of-Time Class Loading & Linking (483)
- Generational Shenandoah Experimental (404)

Removals and Warnings of Future Changes

- Prepare to Restrict the Use of JNI (472)
- Permanently Disable the Security Manager (486)
- Warn upon Use of Memory-Access Methods in `sun.misc.Unsafe` (498)
- ZGC: Remove the Non-Generational Mode (490)

Ports and Tools

- Remove the Windows 32-bit x86 Port (479)
- Deprecate the (Linux) 32-bit x86 Port for Removal (501)
- Linking Run-Time Images without JMODs (493)

Java SE 24 (in blue)

Language Previews
- Primitive Types in Patterns, `instanceof`, `switch` (488)
- Flexible Constructor Bodies (492)
- Simple Source Files + Instance Main Methods (495)
- Module Import Declarations (494)

Libraries
- Class-File API (484)
- Stream Gatherers (485)
- Structured Concurrency 4th Preview (499)
- Scoped Values 4th Preview (487)
- Vector API 9th Incubator (489)

Security Libraries
- Key Derivation Function API Preview (478)
- Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism (496)
- Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm (497)

Performance and Runtime
- Synchronize Virtual Threads without Pinning (491)
- Compact Object Headers Experimental (450)
- Late Barrier Expansion for G1 (475)
- Ahead-of-Time Class Loading & Linking (483)
- Generational Shenandoah Experimental (404)

Removals and Warnings of Future Changes
- Prepare to Restrict the Use of JNI (472)
- Permanently Disable the Security Manager (486)
- Warn upon Use of Memory-Access Methods in `sun.misc.Unsafe` (498)
- ZGC: Remove the Non-Generational Mode (490)

Ports and Tools
- Remove the Windows 32-bit x86 Port (479)
- Deprecate the (Linux) 32-bit x86 Port for Removal (501)
- Linking Run-Time Images without JMODs (493)

Language Previews

Primitive Types in Patterns, instanceof, and switch (Second Preview)

```
switch (x.getYearlyFlights()) {
    case 0 -> ...;
    case 1 -> ...;
    case 2 -> issueDiscount();
    case int i when i >= 100 -> issueGoldCard();
    case int i -> ... appropriate action when i > 2 && i < 100 ...
}
```

```
boolean v = ...
switch (v) {
    case true -> ...
    case false -> ...
}
```

```
long v = ...;
switch (v) {
    case 1L              -> ...;
    case 2L              -> ...;
    case 10_000_000_000L -> ...;
    case 20_000_000_000L -> ...;
    case long x          -> ... x ...;
}
```

Primitive Types in Patterns, instanceof, and switch (Second Preview)

```
// 1. A sealed hierarchy modeling JSON values
sealed interface JsonValue {
    record JsonString(String s) implements JsonValue { }
    record JsonNumber(double d) implements JsonValue { }
    record JsonObject(Map<String, JsonValue> map) implements JsonValue { }
}
// 2. Construct a JsonValue
var json = new JsonObject(Map.of("name", new JsonString("John"),
                                 "age",  new JsonNumber(30)));  // pass an int to a double parameter
// 3. Deconstruct a JsonValue
if (json instanceof JsonObject(var map)
     && map.get("name") instanceof JsonString(String name)
     && map.get("age")  instanceof JsonNumber(double age)) {
        int age2 = (int)age;  // unavoidable (and potentially lossy!) cast
        ... name ... age2 ... // argh
}
```

Primitive Types in Patterns, instanceof, and switch (Second Preview)

```
// 1. A sealed hierarchy modeling JSON values
sealed interface JsonValue {
    record JsonString(String s) implements JsonValue { }
    record JsonNumber(double d) implements JsonValue { }
    record JsonObject(Map<String, JsonValue> map) implements JsonValue { }
}
// 2. Construct a JsonValue
var json = new JsonObject(Map.of("name", new JsonString("John"),
                                 "age",  new JsonNumber(30)));  // pass an int to a double parameter
// 3. Deconstruct a JsonValue
if (json instanceof JsonObject(var map)
     && map.get("name") instanceof JsonString(String name)
     && map.get("age")  instanceof JsonNumber(double int age)) {
        int age2 = (int)age;  // unavoidable (and potentially lossy!) cast
        ... name ... age ...
}
```

Flexible Constructor Bodies (Third Preview)

```
public class PositiveBigInteger extends BigInteger {
    public PositiveBigInteger(long value) {
        super(value);    // Potentially unnecessary work
        if (value <= 0) throw new IllegalArgumentException(..);
    }
}
```

```
public class PositiveBigInteger extends BigInteger {
    public PositiveBigInteger(long value) {
        super(verifyPositive(value));    // Annoying workaround
    }
    private static long verifyPositive(long value) {
        if (value <= 0) throw new IllegalArgumentException(..);
        return value;
    }
}
```

Flexible Constructor Bodies (Third Preview)

```
public class PositiveBigInteger extends BigInteger {
    public PositiveBigInteger(long value) {
        if (value <= 0) throw new IllegalArgumentException(..);
        super(value);
    }
}
```

Flexible Constructor Bodies (Third Preview)

```
class Employee extends Person {
    String officeID;

    Employee(..., int age, String officeID) {
        if (age < 18  || age > 67)
            throw new IllegalArgumentException("invalid working age");
        this.officeID = officeID;  // Initialize our fields
        super(..., age);
    }
}
```

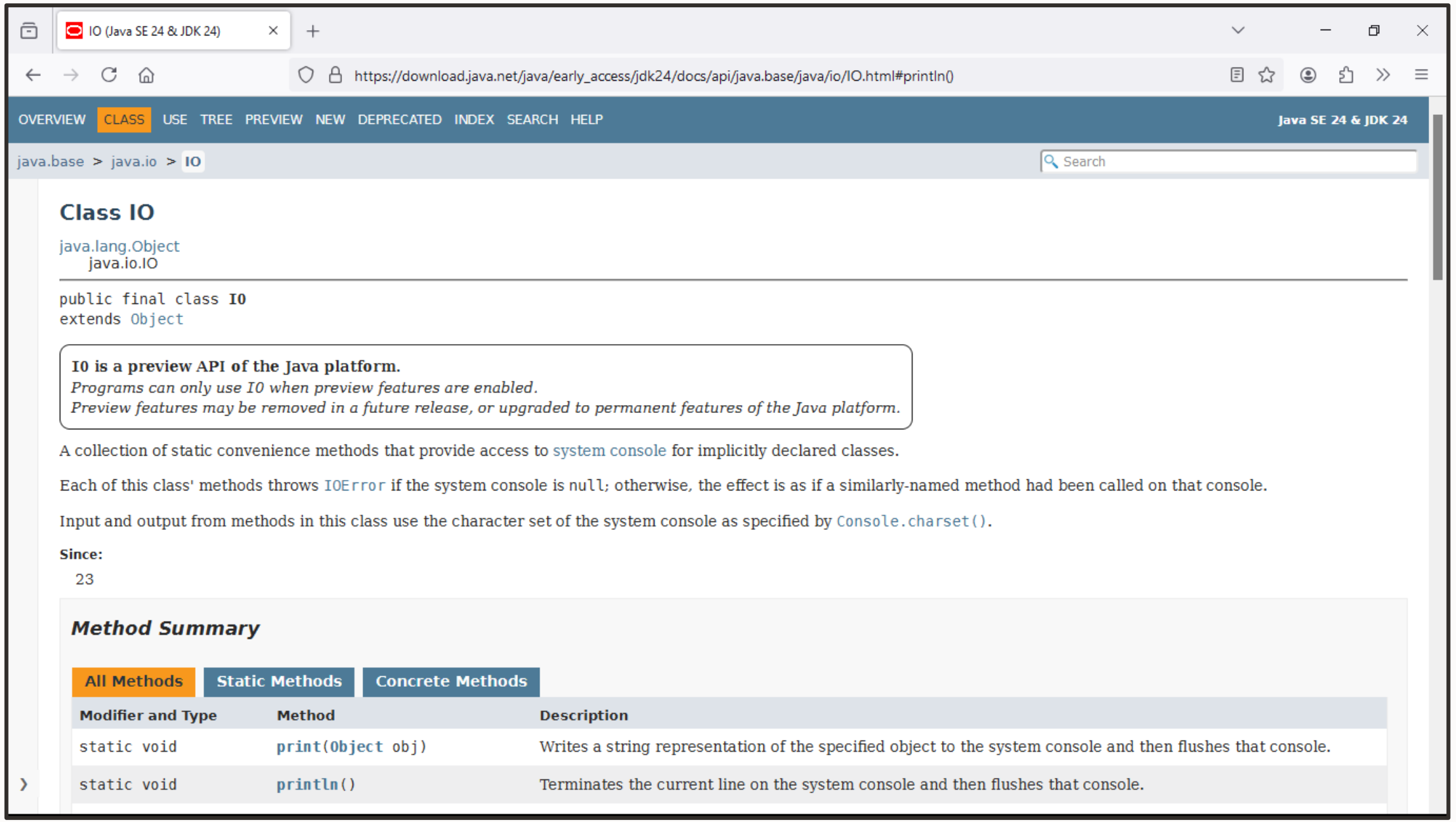
Simple Source Files and Instance Main Methods (Fourth Preview)

Day1.java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Day1.java

```
void main() {
    System.out.println("Hello, World!");
}
```



Simple Source Files and Instance Main Methods (Fourth Preview)

```
import static java.io.IO.*;
import module java.base;
```

Day1.java

```
void main() {
    println("Hello, World!");
}
```

import module java.base;

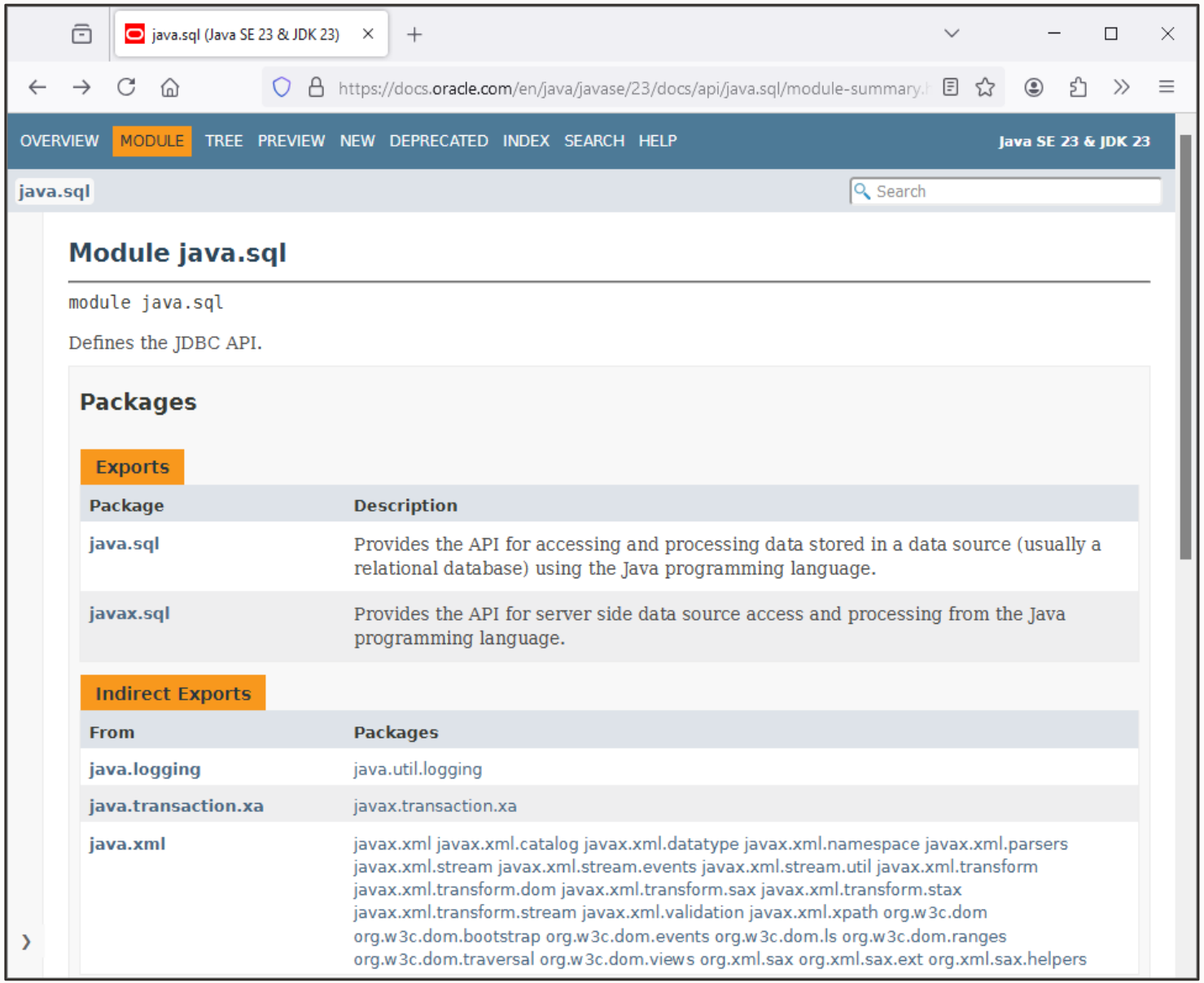
```
import java.io.*;
import java.lang.*;
import java.lang.annotation.*;
import java.lang.classfile.*;
import java.lang.classfile.attribute.*;
import java.lang.classfile.components.*;
import java.lang.classfile.constantpool.*;
import java.lang.classfile.instruction.*;
import java.lang.constant.*;
import java.lang.foreign.*;
import java.lang.invoke.*;
import java.lang.module.*;
import java.lang.ref.*;
import java.lang.reflect.*;
import java.lang.runtime.*;
import java.math.*;
import java.net.*;
import java.net.spi.*;
```

```
import java.nio.*;
import java.nio.channels.*;
import java.nio.channels.spi.*;
import java.nio.charset.*;
import java.nio.charset.spi.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.nio.file.spi.*;
import java.security.*;
import java.security.cert.*;
import java.security.interfaces.*;
import java.security.spec.*;
import java.text.*;
import java.text.spi.*;
import java.time.*;
import java.time.chrono.*;
import java.time.format.*;
import java.time.temporal.*;
import java.time.zone.*;
```

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.concurrent.locks.*;
import java.util.function.*;
import java.util.jar.*;
import java.util.random.*;
import java.util.regex.*;
import java.util.spi.*;
import java.util.stream.*;
import java.util.zip.*;
import javax.crypto.*;
import javax.crypto.interfaces.*;
import javax.crypto.spec.*;
import javax.net.*;
import javax.net.ssl.*;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;
import javax.security.auth.x500.*;
import javax.security.cert.*;
```


import module java.sql;

```
import java.sql.*;
import javax.sql.*;
import module java.logging;
import module java.transaction.xa;
import module java.xml;
```



Libraries

Stream Gatherers

```
long numberOfWords =
    Stream.of("the", "", "fox", "jumps", "over", "the", "", "dog")
          .filter(Predicate.not(String::isEmpty))
          .collect(Collectors.counting());
```


Stream Gatherers

```
List<List<Integer>> windows = Stream.of(1,2,3,4,5,6,7,8).gather(Gatherers.windowFixed(3)).toList();

// [[1, 2, 3], [4, 5, 6], [7, 8]]
```

```
List<List<Integer>> windows2 = Stream.of(1,2,3,4,5,6,7,8).gather(Gatherers.windowSliding(2)).toList();

// [[1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7], [7, 8]]
```

```
List<List<Integer>> windows6 = Stream.of(1,2,3,4,5,6,7,8).gather(Gatherers.windowSliding(6)).toList();

// [[1, 2, 3, 4, 5, 6], [2, 3, 4, 5, 6, 7], [3, 4, 5, 6, 7, 8]]
```

Stream Gatherers

```
Optional<String> numberString =
    Stream.of(1,2,3,4,5,6,7,8,9)
          .gather(
              Gatherers.fold(() -> "", (string, number) -> string + number)
           )
          .findFirst();

// Optional["123456789"]

List<String> numberStrings =
    Stream.of(1,2,3,4,5,6,7,8,9)
          .gather(
              Gatherers.scan(() -> "", (string, number) -> string + number)
           )
          .toList();

// ["1", "12", "123", "1234", "12345", "123456", "1234567", "12345678", "123456789"]
```

Class-File API

Provide a standard API for parsing, generating, and transforming Java class files.

This will:

- Provide an API for processing class files that tracks the class file format defined by the Java Virtual Machine Specification.
- Enable JDK components to migrate to the standard API, and eventually remove the JDK's internal copy of the third-party ASM library.

Class-File API

```
CodeTransform rewriteSystemExit =
    (codeBuilder, codeElement) -> {
        if (codeElement instanceof InvokeInstruction i
                && i.opcode() == Opcode.INVOKESTATIC
                && "java/lang/System".equals(i.owner().asInternalName())
                && "exit".equals(i.name().stringValue())
                && "(I)V".equals(i.type().stringValue())) {
            var runtimeException = ClassDesc.of("java.lang.RuntimeException");
            codeBuilder.new_(runtimeException)
                       .dup()
                       .ldc("System.exit not allowed")
                       .invokespecial(runtimeException, "<init>", MethodTypeDesc.ofDescriptor("(Ljava/lang/String;)V"), false)
                       .athrow();
        } else {
            codeBuilder.with(codeElement);
        }
    };
```


Peter Shor's Algorithm

A quantum computing algorithm published in 1994

- Can efficiently factor large integers and solve the discrete logarithm problem in polynomial time
- This will break all mainstream asymmetric cryptographic algorithms today, including RSA and all flavors of Diffie-Hellman
- Experts have different opinions, but they predict that a Cryptographically Relevant Quantum Computer (CRQC) could be created within the next 10 to 30 years

The solution to this threat is called Post-Quantum Cryptography (PQC).

Harvest Now, Decrypt Later

The attack on key exchange is more serious than digital signatures

- Attackers can gather key exchange message and encrypted communication data now
- And decrypt them in the future when a CRQC is available
- Thus, getting access to all past data communications

Therefore, *early* adoption of PQC key exchange algorithms is *crucial*

PQC Support in the Java Platform

Support for 3 core quantum-resistant algorithms

- HSS/LMS
- ML-KEM
- ML-DSA

Two new important building block APIs

- `KEM`
- `KDF`

Future JDK releases will incorporate more PQC support (ex: TLS Hybrid Key Exchange)

PQC Support: ML-KEM

Quantum-resistant, lattice-based key encapsulation mechanism specified in FIPS 203

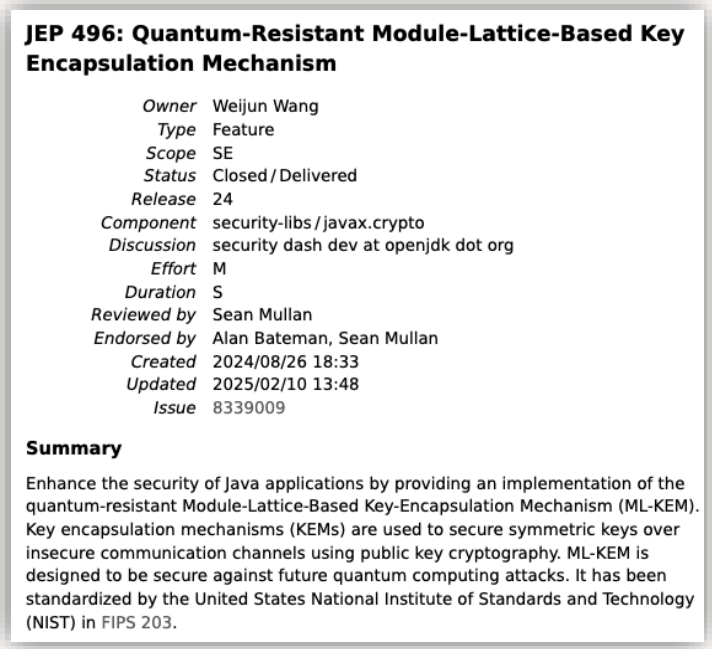
- Defined in JEP 496 and delivered in Java 24

New `KeyPairGenerator`, `KeyFactory`, and `KEM` implementations

- Based on the `KEM` API that was introduced in Java 21
  - And also backported to Java 17
- 3 parameter sets: ML-KEM-512, ML-KEM-768, ML-KEM-1024

Tool support

- `keytool` supports generating ML-KEM key pairs and certificates
  - Certificates must be signed with a different algorithm (ex: RSA or ML-DSA)



JEP 496: Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism

Owner	Weijun Wang
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	24
Component	security-libs / javax.crypto
Discussion	security dash dev at openjdk dot org
Effort	M
Duration	S
Reviewed by	Sean Mullan
Endorsed by	Alan Bateman, Sean Mullan
Created	2024/08/26 18:33
Updated	2025/02/10 13:48
Issue	8339009

Summary

Enhance the security of Java applications by providing an implementation of the quantum-resistant Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM). Key encapsulation mechanisms (KEMs) are used to secure symmetric keys over insecure communication channels using public key cryptography. ML-KEM is designed to be secure against future quantum computing attacks. It has been standardized by the United States National Institute of Standards and Technology (NIST) in FIPS 203.

PQC Support: ML-DSA

Quantum-resistant, lattice-based signature algorithm specified in FIPS 204

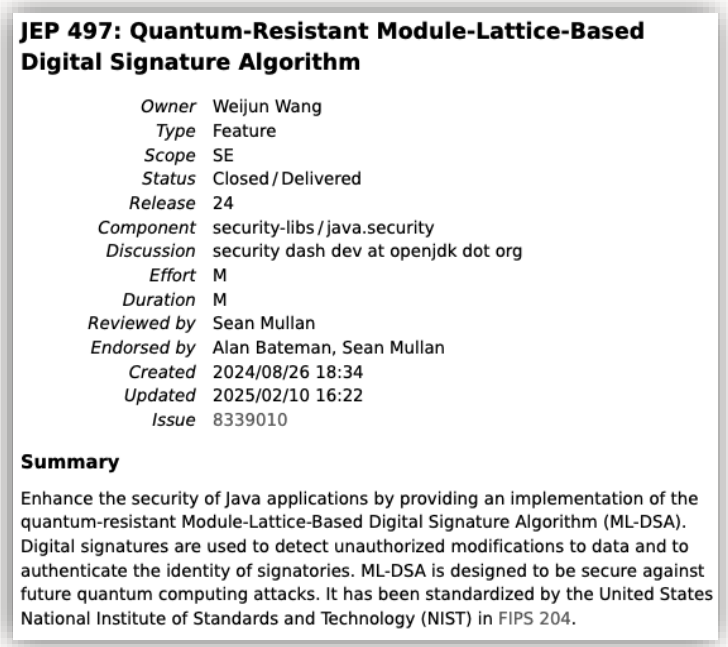
- Defined in JEP 497 and delivered in Java 24

New `KeyPairGenerator`, `KeyFactory`, and `Signature` implementations

- 3 parameter sets: ML-DSA-44, ML-DSA-65, ML-DSA-87

Tool support

- `keytool` supports generating ML-DSA key pairs and certificates



JEP 497: Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm

Owner	Weijun Wang
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	24
Component	security-libs / java.security
Discussion	security dash dev at openjdk dot org
Effort	M
Duration	M
Reviewed by	Sean Mullan
Endorsed by	Alan Bateman, Sean Mullan
Created	2024/08/26 18:34
Updated	2025/02/10 16:22
Issue	8339010

Summary

Enhance the security of Java applications by providing an implementation of the quantum-resistant Module-Lattice-Based Digital Signature Algorithm (ML-DSA). Digital signatures are used to detect unauthorized modifications to data and to authenticate the identity of signatories. ML-DSA is designed to be secure against future quantum computing attacks. It has been standardized by the United States National Institute of Standards and Technology (NIST) in FIPS 204.

Removals and Warnings

Permanently Disable the Security Manager

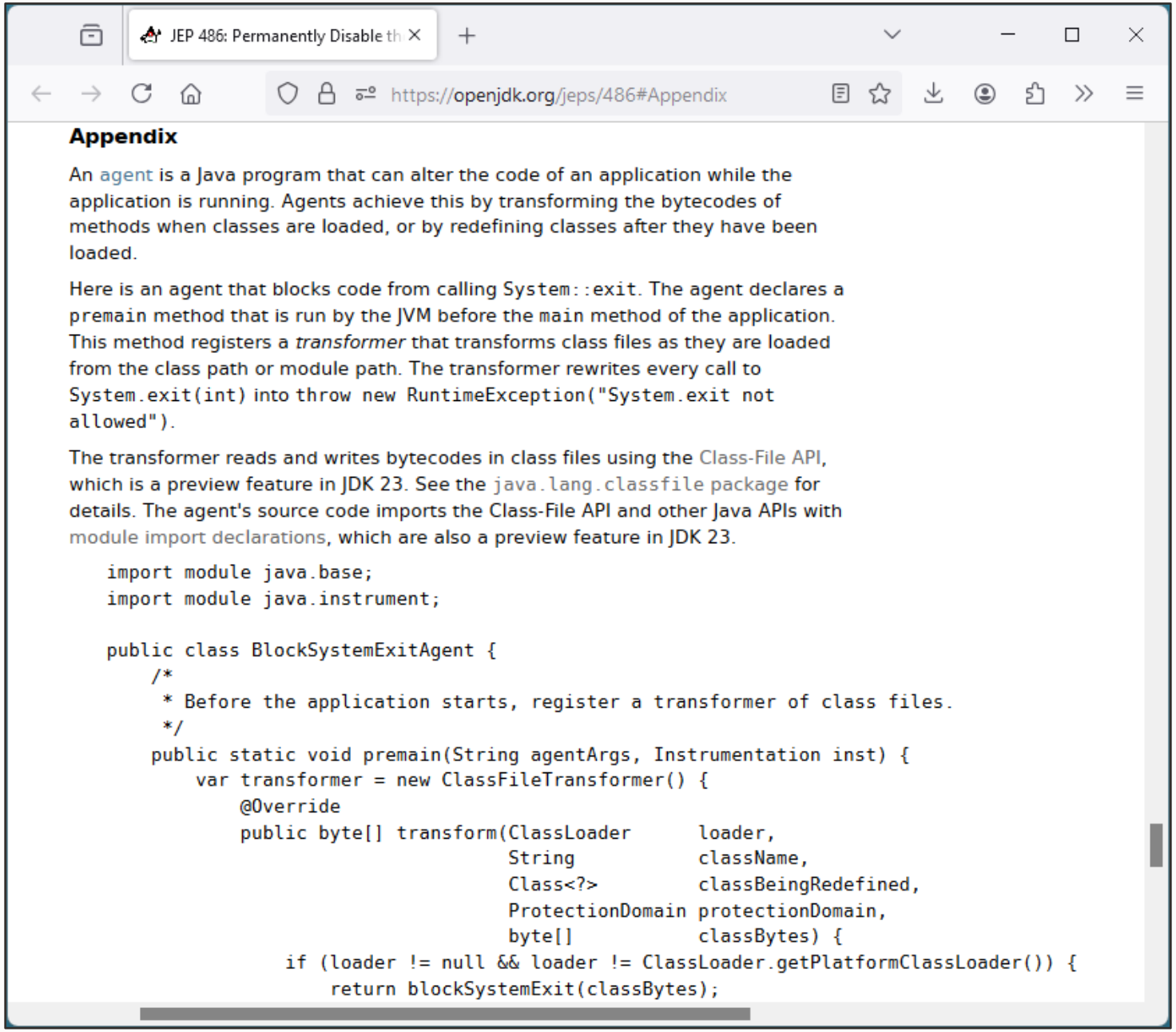
The Security Manager has not been the primary means of securing client-side Java code for many years, it has rarely been used to secure server-side code, and it is costly to maintain. We therefore deprecated it for removal in Java 17 via JEP 411 (2021).

As the next step toward removing the Security Manager, we will revise the Java Platform specification so that developers cannot enable it and other Platform classes do not refer to it. This change will have no impact on the vast majority of applications, libraries, and tools.

We will remove the Security Manager API in a future release.

This will:

- Remove the ability to enable the Security Manager when starting the Java runtime.
- Remove the ability to install a Security Manager while an application is running.
- Retain the Security Manager API in this release, so that maintainers of existing code that depends upon it have time to migrate away.



Appendix

An agent is a Java program that can alter the code of an application while the application is running. Agents achieve this by transforming the bytecodes of methods when classes are loaded, or by redefining classes after they have been loaded.

Here is an agent that blocks code from calling `System::exit`. The agent declares a `premain` method that is run by the JVM before the `main` method of the application. This method registers a *transformer* that transforms class files as they are loaded from the class path or module path. The transformer rewrites every call to `System.exit(int)` into `throw new RuntimeException("System.exit not allowed")`.

The transformer reads and writes bytecodes in class files using the Class-File API, which is a preview feature in JDK 23. See the `java.lang.classfile` package for details. The agent's source code imports the Class-File API and other Java APIs with module import declarations, which are also a preview feature in JDK 23.

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

public class BlockSystemExitAgent {
    /*
     * Before the application starts, register a transformer of class files.
     */
    public static void premain(String agentArgs, Instrumentation inst) {
        var transformer = new ClassFileTransformer() {
            @Override
            public byte[] transform(ClassLoader      loader,
                                    String           className,
                                    Class<?>         classBeingRedefined,
                                    ProtectionDomain protectionDomain,
                                    byte[]           classBytes) {
                if (loader != null && loader != ClassLoader.getPlatformClassLoader()) {
                    return blockSystemExit(classBytes);
```

Prepare to Restrict the Use of JNI

Issue warnings about uses of the Java Native Interface (JNI) and adjust the Foreign Function & Memory (FFM) API to issue warnings in a consistent manner.

All such warnings aim to prepare developers for a future release that ensures integrity by default by uniformly restricting JNI and the FFM API.

Application developers can avoid both current warnings and future restrictions by selectively enabling these interfaces where essential.

This will:

- Preserve the status of JNI as a standard way to interoperate with native code.
- Prepare the Java ecosystem for a future release that disallows interoperation with native code by default, whether via JNI or the FFM API. As of that release, application developers will have to explicitly enable the use of JNI and the FFM API at startup.
- Align the use of JNI and the FFM API so that library maintainers can migrate from one to the other without requiring application developers to change any command-line options.

Warn upon Use of Memory-Access Methods in `sun.misc.Unsafe`

- 90% of the methods in `sun.misc.Unsafe` provide access to on-heap and off-heap memory.
- These methods have supported replacements in the FFM API and other standard APIs.
- JDK 23 deprecated the methods for removal, causing warnings at compile time.
- JDK 24 gives warnings at run time.
- JDK 26+ will throw exceptions.
- JDK 26++ will remove the methods.

```
--sun-misc-unsafe-memory-access={allow, warn, deny}
```

Default in: 23 24 26+

Thank You