

ORACLE

# Java 26

For the JCP Executive Committee

---

**Alex Buckley**

Java Platform Group, Oracle

March 2026

The Java Community Process(SM) | X

+

— □ X

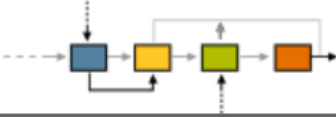
← → ↻ 🏠 🔒

https://jcp.org/en/jsr/detail?id=401

A 🔖 ⚙️ 🎵 ☆ ⌚ 👤 ... 🗨️ Chat



Java Community Process



Community Development of Java Technology Specifications

Press Room | Get Java Here |  

JSRs



» JSRs by Platform

» JSRs by Technology

» JSRs by Stage

» JSRs by Committee

» List of All JSRs

My JCP

Sign-in

Register for Site

Use of JCP site is subject to the [JCP Terms of Use](#) and the [Oracle Privacy Policy](#)

JSRCommunityExpert Group

Summary | Proposal | Detail (Summary & Proposal) | Nominations

JSRs: Java Specification Requests

JSR 401: Java™ SE 26

| Stage                               | Access                        | Start        | Finish       |
|-------------------------------------|-------------------------------|--------------|--------------|
| Final Release                       | <a href="#">Download page</a> | 10 Mar, 2026 |              |
| Public Review Final Approval Ballot | <a href="#">View results</a>  | 24 Feb, 2026 | 02 Mar, 2026 |
| Public Review                       | <a href="#">Download page</a> | 20 Jan, 2026 | 23 Feb, 2026 |
| Expert Group Formation              |                               | 28 May, 2025 | 01 Jul, 2025 |

Status: **Active**

JCP version in use: 2.11

Java Specification Participation Agreement version in use: 2.0

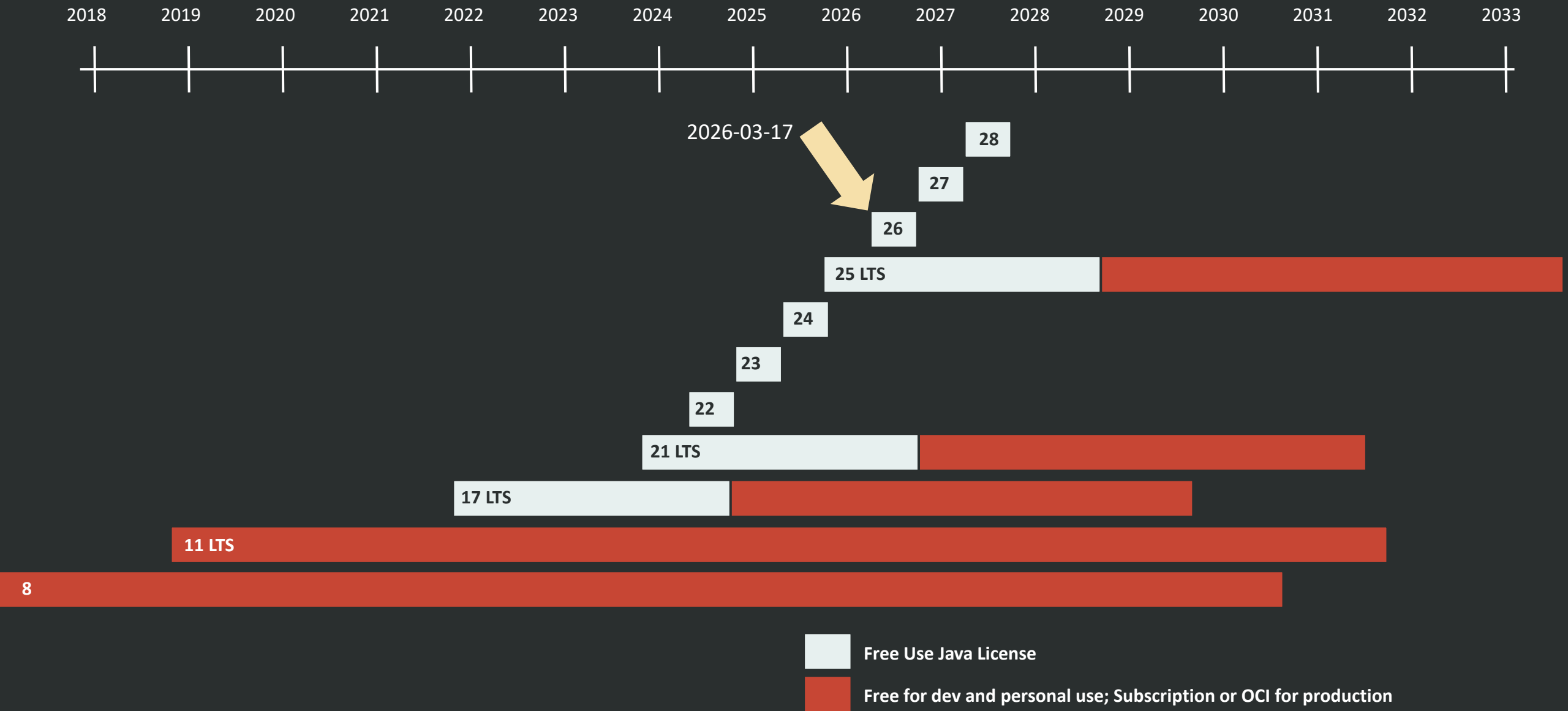
Description:

The JSR for the Java SE 26 Platform. The Reference Implementation of this Specification is the Java Development Kit, version 26.

2

Copyright © 2025, Oracle and/or its affiliates

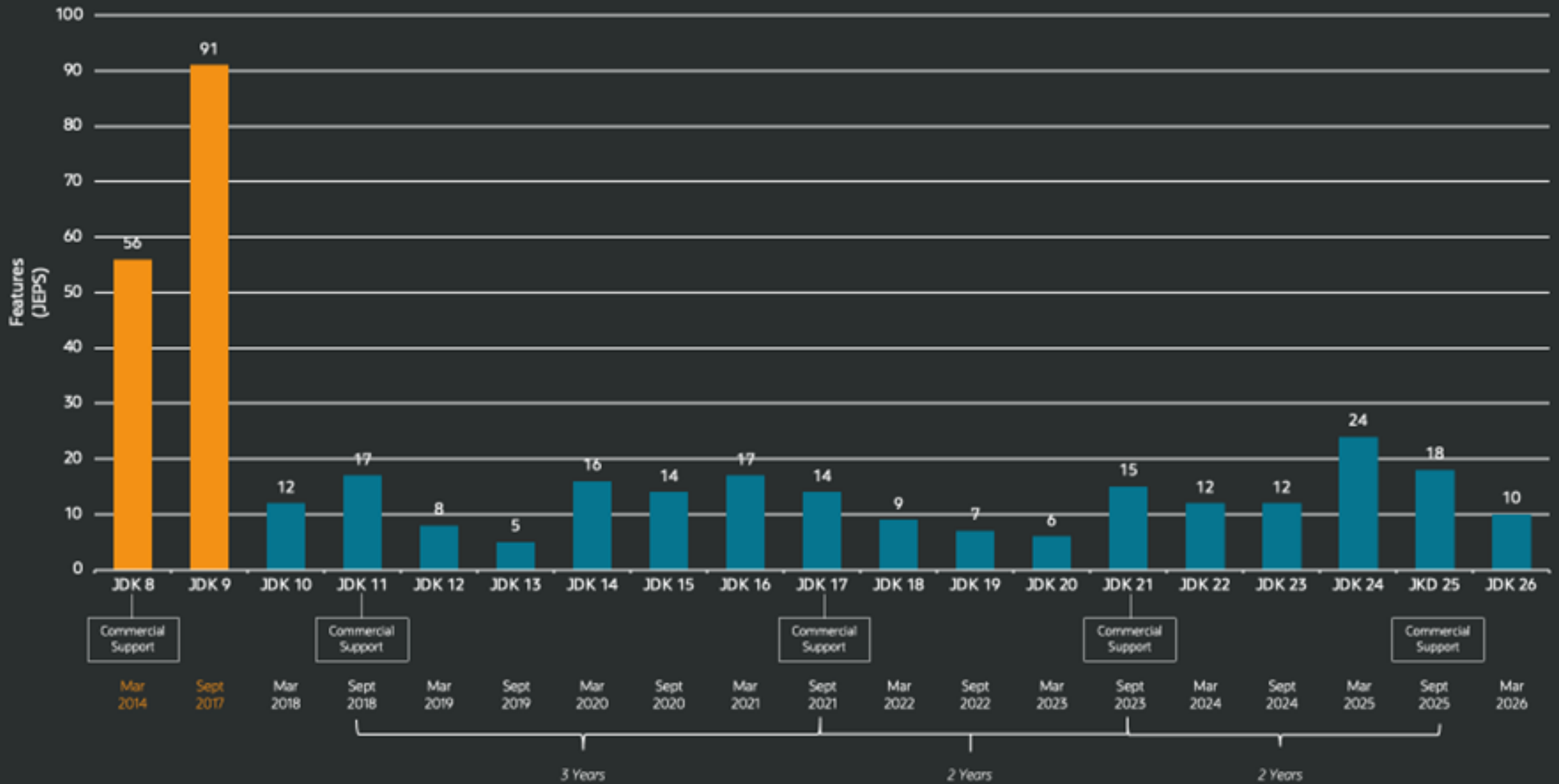




## Issues fixed in JDK 26 per organization

Alibaba Amazon AMD ARM Azul BellSoft Bytedance Canonical Fujitsu Google Huawei IBM Independent Intel ISCAS JetBrains Loongson Microsoft NTT Data Nvidia





# New in Java 26

## Language

Primitive Types in Patterns, instanceof, and switch <sup>4th Preview</sup> [530]

## Libraries

HTTP/3 for the HTTP Client API [517]

Structured Concurrency <sup>6th Preview</sup> [525]

Lazy Constants <sup>2nd Preview</sup> [526]

## Security

PEM Encodings of Cryptographic Objects <sup>2nd Preview</sup> [524]

## Integrity

Prepare to Make Final Mean Final [500]

## Removals

Remove the Applet API [504]

## Libraries

Vector API <sup>11th Incubator</sup> [529]

## Performance

G1 GC: Improve Throughput by Reducing Synchronization [522]

Ahead-of-Time Object Caching with any GC [516]

# New in Java 26

## Language

Primitive Types in Patterns, instanceof, and switch <sup>4th Preview</sup> [530]

## Libraries

HTTP/3 for the HTTP Client API [517]

Structured Concurrency <sup>6th Preview</sup> [525]

Lazy Constants <sup>2nd Preview</sup> [526]

## Security

PEM Encodings of Cryptographic Objects <sup>2nd Preview</sup> [524]

## Integrity

Prepare to Make Final Mean Final [500]

## Removals

Remove the Applet API [504]

## Libraries

Vector API <sup>11th Incubator</sup> [529]

## Performance

G1 GC: Improve Throughput by Reducing Synchronization [522]

Ahead-of-Time Object Caching with any GC [516]

# JEP 530: Primitive Types in Patterns, instanceof, and switch <sup>4th Preview</sup>

## Traditionally:

- Pattern matching for switch does not support primitive type patterns.

 `case int i when i >= 100 -> issueGoldCard();`

- Pattern matching for instanceof does not support primitive type patterns.

 `if (i instanceof byte b) { ... b ... }`

- Pattern matching is inflexible when matching primitive types.

 `record JsonNumber(double d) {}  
if (json instanceof JsonNumber(int age)) { ... age ... }`

- instanceof and switch support some, but not all, primitive types.

 `switch (getMarketCap()) { case 10_000_000_000L -> ... }  
switch (isCompleted()) { case true -> ...; case false -> ...; }`



## Pattern matching works with *all* types

- Enhance instanceof and switch to support primitive type patterns as top level patterns.

 `case int i when i >= 100 -> issueGoldCard();  
if (i instanceof byte b) { ... b ... }`

- Extend pattern matching so that primitive type patterns are applicable to a wider range of match candidate types.

 `record JsonNumber(double d) {}  
if (json instanceof JsonNumber(int age)) { ... age ... }`

- Enhance switch so it works with all primitive types, not just a subset of the integral primitive types

 `switch (getMarketCap()) { case 10_000_000_000L -> ... }  
switch (isCompleted()) { case true -> ...; case false -> ...; }`

User benefit: Uniform data exploration with pattern matching.

Platform benefit: By formalizing *exact* and *inexact* type conversions, we prepare for user-defined conversions of value types, e.g., float16 -> float8.

# Looking forward

## Why four previews?

- Type conversions are a complex and foundational language feature (“We read JLS chapter 5 so you don’t have to”)
- They underpin multiple statements and expressions
- A cross-product of complexity: we need to slide new conversion rules into new and existing constructs
- We want to remove arbitrary legacy restrictions, e.g., no floats or longs in switch

## Going forward

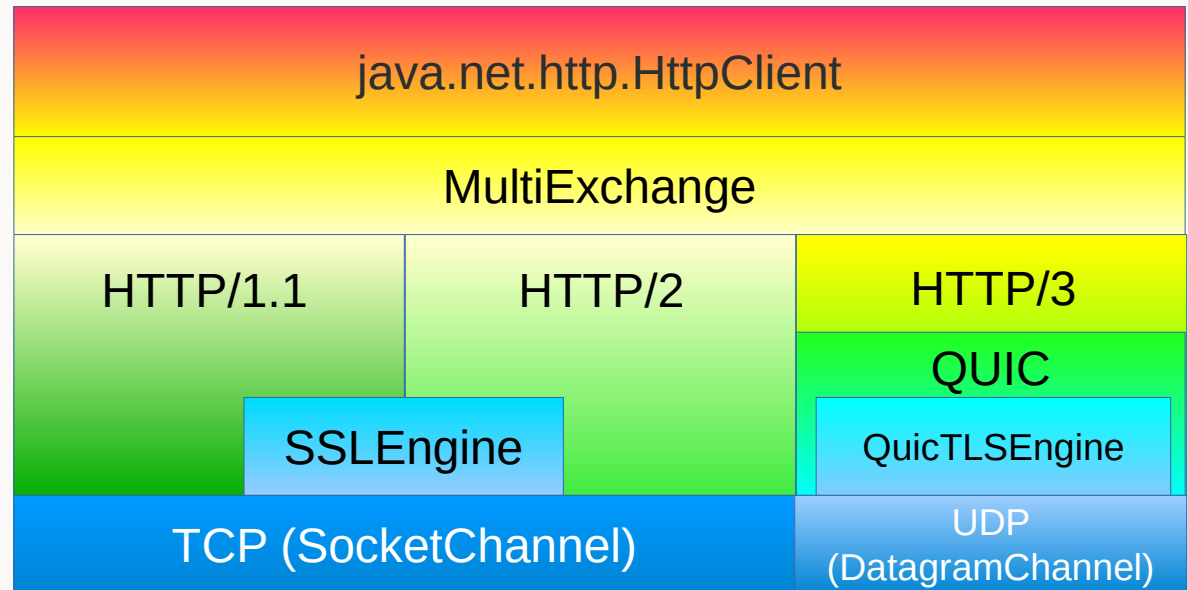
- Likely to re-preview in Java 27 with no changes
- Further JEPs coming for pattern matching in particular and data-oriented programming in general

1. (Readiness) The preview feature has a *high probability* of being 100% finished *within 12 months*. This timeline reflects our experience that two rounds of previewing is the norm, i.e., preview in Java \$N and \$N+1 then final in \$N+2. For APIs that have exceptionally large surface areas or engage deeply with the JVM, and for language features that integrate with other language features as a matter of necessity, we anticipate additional rounds of feedback and revision, as such features will underpin the Java ecosystem for decades to come.

<https://openjdk.org/jeps/12>

# JEP 517: HTTP/3 for the HTTP Client API

- HTTP/3 is the successor to HTTP/2.
- Preserves HTTP semantics while changing the transport for improved latency and reliability.
- Runs over QUIC (UDP), secured with TLS 1.3.
- Benefits: potentially faster handshakes, avoids head-of-line blocking, better performance on lossy networks.
- Because HTTP/3 uses QUIC over UDP, you cannot:
  - upgrade an existing HTTP/1.1 or HTTP/2 (TCP) connection to HTTP/3, or
  - negotiate HTTP/3 over HTTP/1.1 or HTTP/2 during the TCP/TLS handshake.
- Deployment is uneven, so clients must be prepared to fall back to HTTP/2 or HTTP/1.1.



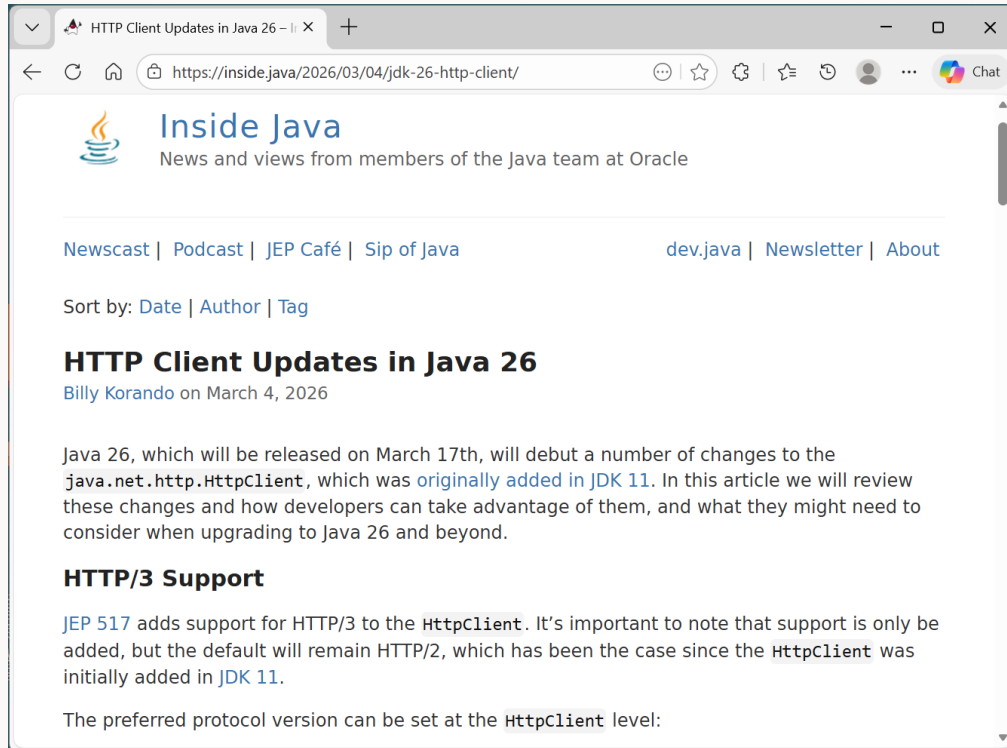
## HTTP/3 support in `java.net.http.HttpClient`

- Opt in at the client level: `HttpClient.newBuilder().version(HttpClient.Version.HTTP_3).build()`
- Opt in at the request level: `HttpRequest.newBuilder(uri).version(HttpClient.Version.HTTP_3).build()`

```
// Example code for a GET, with client-level opt-in
var client    = HttpClient.newBuilder().version(HttpClient.Version.HTTP_3).build();
var request   = HttpRequest.newBuilder(URI.create("https://openjdk.org/")).GET().build();
var response  = client.send(request, HttpResponse.BodyHandlers.ofString());
assert response.statusCode() == 200;
String htmlText = response.body();
```

- Minimal code change for applications; sending requests and handling responses works as usual.
- Multiple discovery strategies are supported, including first-try HTTP/3, parallel racing, `Alt-Svc` header discovery.
- When HTTP/3 is preferred, the client transparently downgrades to HTTP/2 or HTTP/1.1 if the server lacks HTTP/3.

# Other HTTP Client updates in Java 26



The screenshot shows the 'Inside Java' website header with the Oracle logo and navigation links. The article title is 'HTTP Client Updates in Java 26'. The text mentions that Java 26, released on March 17th, includes changes to `java.net.http.HttpClient`, which was originally added in JDK 11. It also highlights 'HTTP/3 Support' via JEP 517, noting that support is optional and the default remains HTTP/2.

## HTTP Client Updates in Java 26

Billy Korando on March 4, 2026

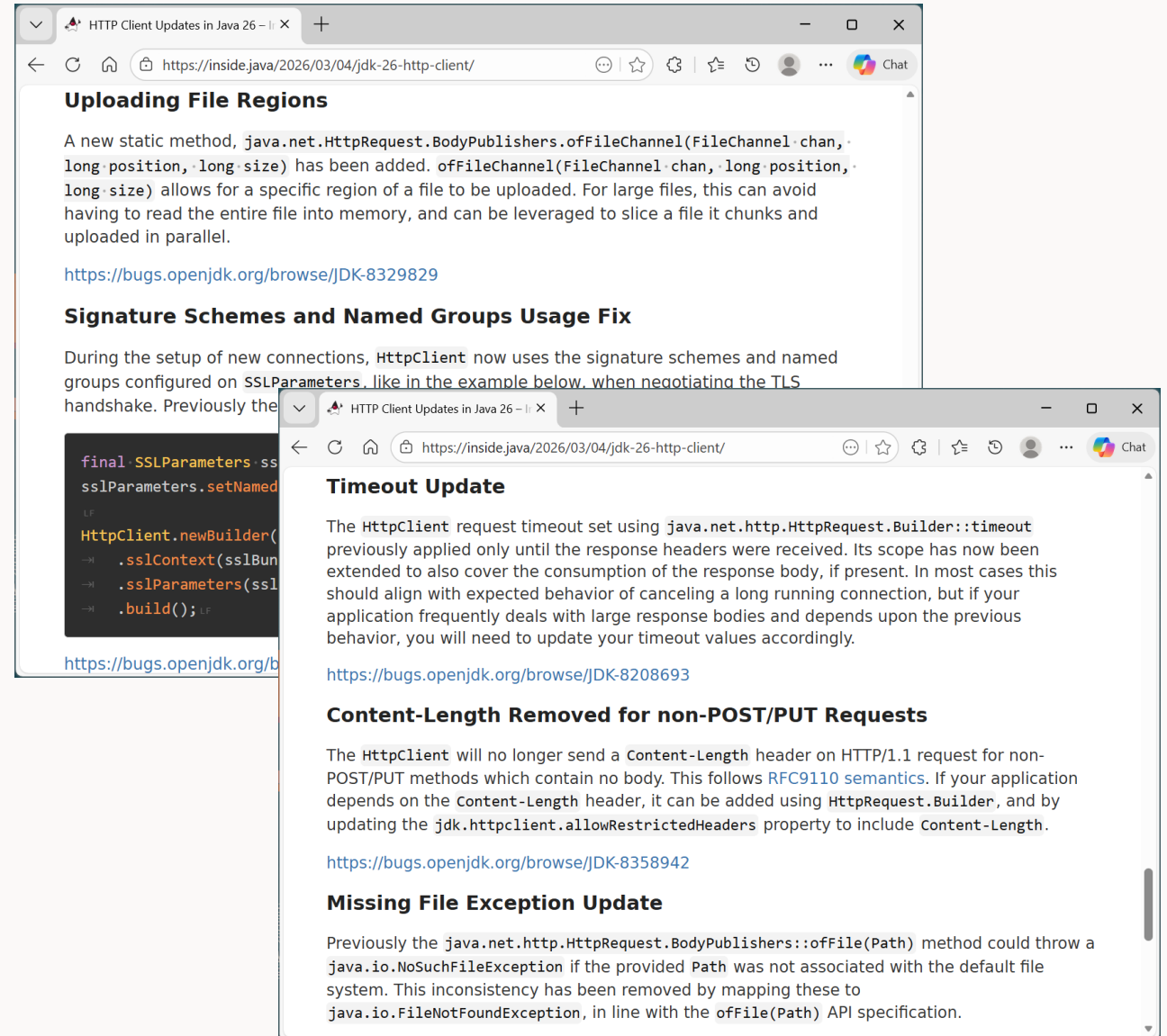
Java 26, which will be released on March 17th, will debut a number of changes to the `java.net.http.HttpClient`, which was originally added in JDK 11. In this article we will review these changes and how developers can take advantage of them, and what they might need to consider when upgrading to Java 26 and beyond.

### HTTP/3 Support

JEP 517 adds support for HTTP/3 to the `HttpClient`. It's important to note that support is only be added, but the default will remain HTTP/2, which has been the case since the `HttpClient` was initially added in JDK 11.

The preferred protocol version can be set at the `HttpClient` level:

<https://inside.java/2026/03/04/jdk-26-http-client/>



The top screenshot shows the 'Uploading File Regions' section, which introduces a new static method `java.net.HttpRequest.BodyPublishers.offFileChannel(FileChannel chan, long position, long size)` for uploading specific file regions in parallel. It also mentions a 'Signature Schemes and Named Groups Usage Fix' where `HttpClient` now uses configured signature schemes and named groups during TLS handshake.

The bottom screenshot shows the 'Timeout Update' section, explaining that the `java.net.http.HttpRequest.Builder::timeout` property now covers the entire response, not just headers. It also mentions 'Content-Length Removed for non-POST/PUT Requests' and a 'Missing File Exception Update' where `java.io.NoSuchFileException` has been replaced with `java.io.FileNotFoundException` for the `offFile(Path)` method.

### Uploading File Regions

A new static method, `java.net.HttpRequest.BodyPublishers.offFileChannel(FileChannel chan, long position, long size)` has been added. `offFileChannel(FileChannel chan, long position, long size)` allows for a specific region of a file to be uploaded. For large files, this can avoid having to read the entire file into memory, and can be leveraged to slice a file it chunks and uploaded in parallel.

<https://bugs.openjdk.org/browse/JDK-8329829>

### Signature Schemes and Named Groups Usage Fix

During the setup of new connections, `HttpClient` now uses the signature schemes and named groups configured on `SSLParameters`, like in the example below, when negotiating the TLS handshake. Previously the

```
final SSLParameters sslParameters = new SSLParameters();
sslParameters.setNamedGroups(Groups.TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256);
HttpClient.newBuilder()
    .sslContext(sslContext)
    .sslParameters(sslParameters)
    .build();
```

<https://bugs.openjdk.org/browse/JDK-8208693>

### Timeout Update

The `HttpClient` request timeout set using `java.net.http.HttpRequest.Builder::timeout` previously applied only until the response headers were received. Its scope has now been extended to also cover the consumption of the response body, if present. In most cases this should align with expected behavior of canceling a long running connection, but if your application frequently deals with large response bodies and depends upon the previous behavior, you will need to update your timeout values accordingly.

<https://bugs.openjdk.org/browse/JDK-8208693>

### Content-Length Removed for non-POST/PUT Requests

The `HttpClient` will no longer send a `Content-Length` header on HTTP/1.1 request for non-POST/PUT methods which contain no body. This follows RFC9110 semantics. If your application depends on the `Content-Length` header, it can be added using `HttpRequest.Builder`, and by updating the `jdk.httpclient.allowRestrictedHeaders` property to include `Content-Length`.

<https://bugs.openjdk.org/browse/JDK-8358942>

### Missing File Exception Update

Previously the `java.net.http.HttpRequest.BodyPublishers.offFile(Path)` method could throw a `java.io.NoSuchFileException` if the provided `Path` was not associated with the default file system. This inconsistency has been removed by mapping these to `java.io.FileNotFoundException`, in line with the `offFile(Path)` API specification.

## Looking forward

- HTTP/3 is not the default due to uneven deployment; this may change in future.
- Potential enhancements
  - Configuration and tuning via system properties.
  - HTTP/3-specific IOException and SSLException subclasses.
  - Current implementation supports only the default SunJSSE provider; third-party providers may be supported.



## JEP 525: Structured Concurrency <sup>6th Preview</sup>

- Simplifies cases where a main task (a unit of work) splits into several concurrent subtasks.
- The API preserves the tasks' natural relationship: it confines the dynamic lifetime of subtasks to the main task's block of code.
- This reduces bugs that arise from cancellation and shutdown.
- A great match for virtual threads!



# Structured Concurrency respects the Java programming model

## Sequential code

```
String stringPart = fetchString(); // throws IOException
int intPart = fetchInt();          // throws IOException

var result = new MyResult(stringPart, intPart);
```

- Useful stack trace if fetchString or fetchInt throws
- Useful thread dump if fetchString or fetchInt hangs
- No additional cleanup/cancellation if fetchString or fetchInt throws

## Structured Concurrency

```
try (var scope = StructuredTaskScope.open()) {

    Subtask<String> subtask1 = scope.fork(this::fetchString);
    Subtask<Integer> subtask2 = scope.fork(this::fetchInt);

    scope.join();

    var result = new MyResult(subtask1.get(), subtask2.get());

}
```

- Useful stack trace for “main” task **and** subtask if fetchString or fetchInt throws
- Useful thread dump if fetchString or fetchInt hangs
- No complicated cleanup/cancellation if fetchString or fetchInt throws



## Example: Fetch results from two remote services

```
try (var scope = StructuredTaskScope.open(Joiner.<String>allSuccessfulOrThrow())) {  
  
    scope.fork(this::fetchString);  
    scope.fork(this::fetchOtherString);  
  
    List<String> results = scope.join(); // in fork order  
  
} catch (FailedException e) {  
    Throwable cause = e.getCause();  
    // ..  
}
```

## Example: Fetch results from *any* of the remote services

```
try (var scope = StructuredTaskScope.open(Joiner.<String>anySuccessfulOrThrow())) {  
  
    scope.fork(this::fetchString);  
    scope.fork(this::fetchOtherString);  
  
    String winner = scope.join();  
  
} catch (FailedException e) {  
    // all failed  
    Throwable cause = e.getCause();  
}
```

## Monitoring with jcmd: Hierarchical thread dumps (Main task)

```
{
  "threadDump": {
    "processId": "87677",
    "time": "2026-03-10T16:14:55.757959Z",
    "runtimeVersion": "27-internal-albatem.open",
    "threadContainers": [
      {
        "container": "<root>",
        "parent": null,
        "owner": null,
        "threads": [
          {
            "tid": "3",
            "time": "2026-03-10T16:14:55.762706Z",
            "name": "main",
            "state": "WAITING",
            "stack": [
              "java.base\\jdk.internal.misc.Unsafe.park(Native Method)",
              "java.base\\java.util.concurrent.locks.LockSupport.park(LockSupport.java:369)",
              "java.base\\jdk.internal.misc.ThreadForker.waitForAll(ThreadForker.java:305)",
              "java.base\\java.util.concurrent.StructuredTaskScopeImpl.join(StructuredTaskScopeImpl.java:240)",
              "Aggregate.main(Aggregate.java:35)"
            ]
          }
        ]
      }
    ]
  }
}
```

## Monitoring with jcmd: Hierarchical thread dumps (Subtasks)

```
{
  "container": "java.util.concurrent.StructuredTaskScopeImpl@1d81eb93",
  "parent": "<root>",
  "owner": "3",
  "threads": [
    {
      "tid": "38",
      "time": "2026-03-10T16:14:55.775456Z",
      "virtual": true,
      "name": "",
      "state": "WAITING",
      "stack": [ 30 elements ]
    },
    {
      "tid": "39",
      "time": "2026-03-10T16:14:55.775889Z",
      "virtual": true,
      "name": "",
      "state": "WAITING",
      "stack": [ 30 elements ]
    },
    {
      "tid": "41",
      "time": "2026-03-10T16:14:55.776622Z",
      "virtual": true,
      "name": "",
      "state": "TIMED_WAITING",
      "stack": [
        "java.base\\java.lang.VirtualThread.parkNanos(VirtualThread.java:897)",
        "java.base\\java.lang.VirtualThread.sleepNanos(VirtualThread.java:994)",
        "java.base\\java.lang.Thread.sleepNanos(Thread.java:560)",
        "java.base\\java.lang.Thread.sleep(Thread.java:593)",
        "Aggregate.sleep(Aggregate.java:41)",
        "Aggregate.Lambda$main$3(Aggregate.java:34)",
        "java.base\\java.util.concurrent.StructuredTaskScopeImpl$SubtaskImpl.run(StructuredTaskScopeImpl.java:347)",
        "java.base\\java.lang.VirtualThread.run(VirtualThread.java:470)"
      ]
    }
  ],
  "threadCount": "3"
}
```



# Looking forward

## Why six previews?

- The problem space and solution spaces are huge.
- The range of possible APIs is very wide.
- The ecosystem is still digesting virtual threads, replacing thread pools, and moving to a world where there is a 1:1 relationship between task and thread.

## Going forward

- The API is powerful but has a few rough edges around exception handling.
- Seventh Preview is proposed for Java 27.
- A future JEP may propose a lower-level API for library developers to create new concurrency APIs.



## JEP 526: Lazy Constants <sup>2nd Preview</sup>

- Variables with constant values are fundamental for human **and** machine understanding.
  - Immutability simplifies reasoning about programs.
  - Immutable objects can be shared freely and safely across threads.
  - When JVM can trust a value won't change, it can fold/hoist/elide, etc.
- Unfortunately, final fields requires **eager initialization** in <init>/<clinit>, and in textual order.
- Lazy Constants deliver as-if-constant speed but with **lazy initialization**.
  - Good for startup: Lazy constants do not inflate startup time and may eliminate work entirely (e.g., loggers).
  - Good for robustness: Ad-hoc patterns (e.g., DCL) introduce concurrency hazards and block constant folding.
- Best of both worlds: Users no longer have to choose between mutable fields and final fields!



## java.lang.LazyConstant & Friends

```
static final Map<String, OrderController> ORDERS =  
    Map.ofLazy(Set.of("Prod", "Test", "Custom"),  
        _ -> new OrderController());  
  
public static OrderController orders() {  
    var curr = Thread.currentThread().getName();  
    return ORDERS.get(curr);  
}
```

- “Computing function” is provided at construction
- Values computed on first get().
- Value is initialized at most once, even with concurrent callers (“winner initializes”).
- Value is treated as true constant by the JVM after initialization (if the lazy constant is stored in a final field).
- Several “computing functions”:
  - LazyConstant.of(() -> value);
  - List.ofLazy(size, index -> element);
  - Map.ofLazy(keys, key -> value)

# Looking forward

- First preview in Java 25 (as “Stable Values”)
  - Mix of function-style methods and low-level mutation methods.
- Second preview in Java 26
  - Function-style methods only.
  - Renamed StableValue to LazyConstant to match the “lazily initialized but treated as constant” use case.
  - Improved discoverability with lazy collection factories available directly in List and Map.
  - Tighter semantics: null values disallowed.
- Third preview in Java 27 (JEP 531)
  - Support lazy sets, not just lazy lists and maps.
  - Improved exception handling.





## JEP 524: Privacy-Enhanced Mail API

- PEM (Privacy-Enhanced Mail) is a widely used text format for transporting and storing cryptographic objects such as public keys, private keys, certificates, and certificate revocation lists.
- It represents cryptographic objects as Base64 text, wrapped with a human-readable header and footer.

```
-----BEGIN PUBLIC KEY-----  
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEi/kRGOL7wCPTN4KJ2ppeSt5UYB6u  
cPjjukDtFTXbguOIFDdZ650/8HTUqS/sVzRF+dg7H3/tkQ/36KdtuADbwQ==  
-----END PUBLIC KEY-----
```

- PEM is common in certificate chains issued by certificate authorities, in tooling such as OpenSSL, and in security-sensitive systems that store or exchange keys.
- PEM processing is needed for applications like Key Management Systems (KMS) and TLS servers and clients.

# Why a Java SE API for Privacy-Enhanced Mail?

## Decoding a public key in PEM format before the PEM API

```
String pemData = "... PEM encoded key ...";

// 1. Manually strip the public key header and footer from the PEM encoded data
String base64Data = pemData.replace("-----BEGIN PUBLIC KEY-----", "")
    .replaceAll(System.lineSeparator(), "")
    .replace("-----END PUBLIC KEY-----", "");

// 2. Pass it to the Java 8 Base64 API to convert it to a DER-encoded byte array
byte[] encoded = Base64.getDecoder().decode(base64Data);

// 3. Developer must have prior knowledge of the key's algorithm to get a KeyFactory in JCA.
KeyFactory keyFactory = KeyFactory.getInstance("RSA");

// 4. Developer must know the key's binary encoding (PKCS#8 or X.509) and
//     instantiate the proper EncodedKeySpec object with the DER-encoded bytes.
X509EncodedKeySpec keySpec = new X509EncodedKeySpec(encoded);

var key = keyFactory.generatePublic(keySpec);
```

## Decoding a public key in PEM format with the PEM API

```
String pemData = "... PEM encoded key ...";
var key = PEMDecoder.of().decode(pemData, PublicKey.class);
```

# The Privacy-Enhanced Mail API

- Eliminates the need for developers to hand-roll PEM parsing and formatting,
- Provides a consistent high-level API to encode and decode between cryptographic objects and PEM text for common artifacts such as keys, certificates, and CRLs,
- More modern (lambdas, streams) than the PEM API in Bouncy Castle.
- Introduces a common marker interface (DEREncodable) for binary-encodable cryptographic objects, so the encoder and decoder can operate over a unified set of types,

## New API

java.security.PEM  
java.security.PEMEncoder  
java.security.PEMDecoder  
java.security.DEREncodable

## Interoperation with existing APIs

javax.crypto.EncryptedPrivateKeyInfo (Seven new methods)  
java.security.PublicKey  
java.security.PrivateKey  
java.security.KeyPair  
java.security.cert.X509CRL  
java.security.cert.X509Certificate  
java.security.spec.PKCS8EncodedKeySpec  
java.security.spec.X509EncodedKeySpec

# Looking forward

- First preview in Java 25
  - Received good feedback on security-dev.
- Second preview in Java 26
  - Naming adjustments for consistency and ease of use, e.g., PEMRecord to PEM.
  - More abstraction due to accepting DEREncodable instead of concrete crypto objects.
  - Expanded support for encrypted key workflows, e.g., reencoding private keys in PKCS8/ASN.1.
- Likely to finalize in Java 27 (JEP draft 8376991)
  - Light renaming and streamlining of types and methods.





# Onward to Java 27!

---

Released September 2026

Hope to see you at JavaOne 2027!