Java 16 New Features Overview

Guide Java New Java Features About 2997 words About 10 minutes

Java 16 was officially released on March 16, 2021, and is not a long-term support (LTS) version.

Related reading: OpenJDK Java 16 documentation .

JEP 338: Vector API (First Incubation)

The Vector API was originally proposed by JEP 338 and integrated into Java 16 as an incubating API . The second round of incubation was proposed by JEP 414 and integrated into Java 17, the third round was proposed by JEP 417 and integrated into Java 18, and the fourth round was proposed by JEP 426 and integrated into Java 19.

This incubator API provides an initial iteration of an API for expressing vector computations that reliably compile at runtime to the optimal vector hardware instructions for supported CPU architectures, resulting in superior performance over equivalent scalar computations, leveraging Single Instruction Multiple Data (SIMD) technology, a type of instruction available on most modern CPUs. While HotSpot supports automatic vectorization, the set of scalar operations that can be converted is limited and susceptible to code changes. This API will enable developers to easily write portable, high-performance vector algorithms in Java.

I have already introduced the vector API in detail in the Java 18 new features overview , so I will not give an additional introduction here.

JEP 347: Enabling C++14 Language Features

Java 16 enables the use of C++14 language features in C++ source code in the JDK and provides specific guidance on which features can be used in HotSpot code.

In Java 15, the language features used by C++ code in the JDK are limited to the C++98/03 language standard. This requires updating the minimum acceptable versions of the compilers on various platforms.

JEP 376: ZGC Concurrent Thread Stack Handling

Java 16 moves ZGC thread stack handling from safepoints to a concurrent phase, allowing GC safepoint pauses in milliseconds even on large heaps. Eliminating this last source of latency in the ZGC garbage collector can significantly improve application performance and efficiency.

JEP 387: Elastic Metaspace

Since the introduction of Metaspace, feedback has shown that it often consumes excessive off-heap memory, leading to memory waste. The Elastic Metaspace feature returns unused HotSpot class metadata (i.e., metaspace) memory to the operating system more quickly, reducing the metaspace footprint.

In addition, this proposal simplifies the Metaspace code to reduce maintenance costs.

JEP 390: Warnings for Value-Based Classes

> The following introduction is excerpted from: Practice | Analysis of Java16 New Syntax Features , the original text is very well written and recommended for reading.

As early as Java 9, Java designers `@Deprecated` upgraded annotations, adding two new elements: since `since` and . The since element specifies the version in which the annotated API was deprecated, while further clarifies the semantics of an API annotated with the @Deprecated annotation. If , it indicates that the API will definitely be removed in a future version, and developers should use a new API instead, eliminating ambiguity. (Before Java 9, APIs annotated with the @Deprecated annotation had various semantic possibilities, including: risky to use, potential for future compatibility issues, possible removal in a future version, and the need for a better alternative.) `forRemoval` `@Deprecated` `forRemoval` `forRemoval=true`

A closer look at the primitive wrapper classes (e.g., `java.lang.Integer` and `java.lang.Double`) reveals that their constructors are annotated `@Deprecated(since="9", forRemoval = true)` , indicating that they will be removed in the future. This coding style should not be used in your program `new Integer();` (it's recommended to use the _integer_ (int)) `Integer a = 10;` or `Integer.valueOf()` _integer_ (int)). If you continue to use it, the compiler will generate a warning: "'Integer(int)' is deprecated and marked for removal." Also, note that these wrapper types have been assigned the same value types as _integer_ (int) `java.util.Optional` and `java.time.LocalDateTime` _integer_ (int)).

Secondly, if you continue `synchronized` to use value types in synchronized blocks, warnings or even exceptions will be generated at compile time and runtime. It is important to note that even if no warnings or exceptions are generated at compile time and runtime, it is not recommended `synchronized` to use value types in synchronized blocks. For example, let's take auto-increment as an example. Example 1-5:

```java
public void inc(Integer count) {
    for (int i = 0; i < 10; i++) {
        new Thread(() -> {
            synchronized (count) {
                count++;
            }
        }).start();
    }
}
```

When executing the program example above, the final output will inevitably differ from your expectations. This is a common mistake made by many new developers. In a concurrent environment, `Integer` objects cannot be `synchronized` guaranteed thread-safe using . This is because each `count++` operation generates `hashcode` a different . In short, each lock is applied to a different object. Therefore, if you want to ensure atomicity during actual development, you should use `AtomicInteger` .

JEP 392: Bundling Tools

In Java 14, JEP 343 introduced the bundler command `jpackage` . It continued incubation in Java 15, and now in Java 16, it has finally become an official feature.

This packaging tool allows packaging of self-contained Java applications. It supports native packaging formats, providing a natural installation experience for end users, including msi and exe on Windows, pkg and dmg on macOS, and deb and rpm on Linux. It also allc startup parameters to be specified when packaging and can be called directly from the command line or programmatically through the ToolProvider API. Note that the jpackage module name has changed from jdk.incubator.jpackage to jdk.jpackage. This will improve the end user experience when installing applications and simplify deployment in the "app store" model.

For an example of using this packaging tool, see the video Playing with Java 16 jpackage (a web browser is required).

JEP 393: Foreign Memory Access API (Third Incubation)

The foreign memory access API was introduced to allow Java programs to safely and efficiently access foreign memory outside the Java heap.

The foreign memory access API was first incubated in Java 14 (JEP 370), resurrected a second time in Java 15 (JEP 383), and incubated a third time in Java 16.

The purpose of introducing the external memory access API is as follows:

- Universal: A single API should be able to operate on various external memories (such as native memory, persistent memory, heap memory, etc.).
- Security: Regardless of the memory operations, the API should not violate the security of the JVM.
- Control: You can freely choose how to release memory (explicitly, implicitly, etc.).
- Available: If access to external memory is required, the API should be `sun.misc.Unsafe` .

JEP 394: instanceof Pattern Matching (Formalization)

JDK version	Update Type	JEP	Update Content
Java SE 14	preview	JEP 305	First introduction of instanceof pattern matching.
Java SE 15	Second Preview	JEP 375	No changes compared to the previous version. We will continue to collect more feedback.

JDK version	Update Type	JEP	Update Content
Java SE 16	Permanent Release	JEP 394	Pattern variables are no longer implicitly final.

Starting from Java 16, you can `instanceof` modify the value of variables in .

```java
// Old code
if (o instanceof String) {
    String s = (String)o;
    ... use s ...
}

// New code
if (o instanceof String s) {
    ... use s ...
}
```

JEP 395: Record Types (Formalization)

Record type change history:

JDK version	Update Type	JEP	Update Content
Java SE 14	Preview	JEP 359	Introduces `record` the keyword, `record` providing a compact syntax for defining immutable data in a class.
Java SE 15	Second Preview	JEP 384	Supported for use in local methods and interfaces `record` .
Java SE 16	Permanent Release	JEP 395	Non-static inner classes can define non-constant static members.

Starting with Java SE 16, non-static inner classes can define non-constant static members.

```java
public class Outer {
  class Inner {
    static int age;
  }
}
```

> Before JDK 16, if you write the code above, the IDE will prompt you that the static field age cannot be declared static in a non-static inner type, unless initialized with a constant expression.

JEP 396: Strong Encapsulation of JDK Internal Elements by Default

This feature enables strong encapsulation by default for all internal elements of the JDK, `sun.misc.Unsafe` with the exception of critical internal APIs (for example). Code that accesses JDK internal APIs that compiled successfully with earlier versions may no longer work by default. Developers are encouraged to migrate from using internal elements to methods that use the standard API so that they and their users can seamlessly upgrade to future Java versions. Strong encapsulation was controlled by the JDK 9 launcher option --illegal-access, which defaulted to warning in JDK 15 and deny in JDK 16. It is still possible (for now) to relax encapsulation for all packages using a single command-line option, but in the future this will only be possible by turning on specific packages using --add-opens.

JEP 397: Sealed Classes (Preview)

Sealed classes were previewed by JEP 360 and integrated into Java 15. In JDK 16, sealed classes were improved (stricter reference checking and inheritance relationships of sealed classes) and previewed again by JEP 397 .

I covered sealed classes in detail in the Java 14 & 15 New Features Overview, so I won't go into further detail here .

Other optimizations and improvements

- **JEP 380: Unix-Domain Socket Channels** : Unix-domain sockets have been a feature of most Unix platforms and are now supported on Windows 10 and Windows Server 2019. This feature adds Unix-domain (AF_UNIX) socket support to the socket

channel and server socket channel APIs of the java.nio.channels package. It extends the inherited channel mechanism to support Unix-domain socket channels and server socket channels. Unix-domain sockets are used for inter-process communication (IPC) on the same host. They are largely similar to TCP/IP, except that sockets are addressed by file system path names instead of Internet Protocol (IP) addresses and port numbers. For local inter-process communication, Unix-domain sockets are more secure and efficient than TCP/IP loopback connections.
- **JEP 389: Foreign Linker API (Incubating):** This incubating API provides statically typed, pure Java access to native code. This API will greatly simplify the previously complex and error-prone process of binding native libraries. Java 1.1 has supported native method calls through the Java Native Interface (JNI), but it has been difficult to use. Java developers should be able to bind specific native libraries for specific tasks. It also provides support for foreign functions without any intermediate JNI glue code.
- **JEP 357: Migrate from Mercurial to Git** : Previously, the OpenJDK source code was managed using the version management tool Mercurial, and now it has been migrated to Git.
- **JEP 369: Migrate to GitHub** : Following the migration from Mercurial to Git in JEP 357, the OpenJDK community Git repository was hosted on GitHub. However, this migration was only performed for JDK 11 and later.
- **JEP 386: Porting Alpine Linux** : Alpine Linux is an independent, non-commercial Linux distribution. It is very small, requiring no more than 8MB of space for a container and approximately 130MB of disk space for a minimal installation. It is also very simple and secure. This proposal ports the JDK to Alpine Linux. Since Alpine Linux is a lightweight Linux distribution based on the musl library, it is also applicable to other Linux distributions that use the musl library for x64 and AArch64 architectures.
- **JEP 388: Windows/AArch64 Porting** : These JEPs focus less on porting efforts themselves and more on integrating them into the JDK mainline repository. JEP 386 ports the JDK to Alpine Linux and other distributions that use musl as the primary C library on x64. Additionally, JEP 388 ports the JDK to Windows AArch64 (ARM64).

References

- Java Language Changes
- Consolidated JDK 16 Release Notes
- Java 16 officially released, new features analyzed one by one
- Practical | Analyzing the New Syntax Features of Java 16 (Excellent)

Recently Updated2023/10/27 05:49
Contributors: sam , guide , xtexChooser , Guide , Erzbir , Mr.Hope , paigeman , yuanrui