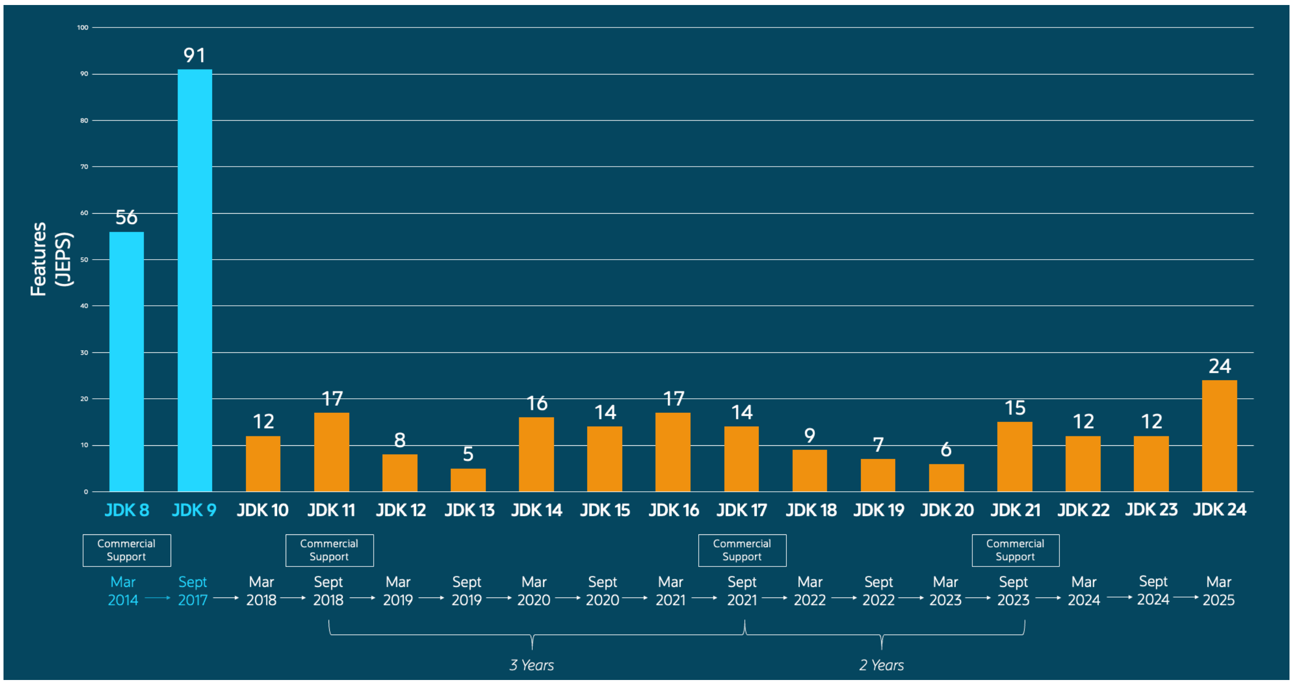
Overview of new features in Java 22 & 23

Guide Java New Java Features About 3984 words About 13 minutes

JDK 23, like JDK 22, is a non-LTS (Long Term Support) version, and Oracle will only provide support for six months. The next long-term support version is JDK 25, which is expected to be released in September next year.

The following figure shows the number of new features and update time brought by each version update from JDK8 to JDK 24:



Because JDK 22 and JDK 23 have many overlapping new features, this article mainly introduces JDK 23 and supplements some features unique to JDK 22.

JDK 23 has 12 new features:

- JEP 455: Primitive Types, instanceof, and Switch in Patterns (Preview)
- JEP 456: Class File API (Second Preview)
- JEP 467: Markdown Documentation Comments
- JEP 469: Vector API (8th incubation)
- JEP 473: Stream Collectors (Second Preview)
- JEP 471: Deprecate memory access methods in sun.misc.Unsafe
- JEP 474: ZGC: Generational Mode by Default
- JEP 476: Module Import Declarations (Preview)

- JEP 477: Unnamed Class and Instance Main Methods (Preview 3)
- JEP 480: Structured Concurrency (Third Preview)
- JEP 481: Scoped Values (Preview 3)
- JEP 482: Flexible Constructor Bodies (Second Preview)

The new features of JDK 22 are as follows:

423: Region Pinning for G1
447: Statements before super(...) (Preview)
454: Foreign Function & Memory API
456: Unnamed Variables & Patterns
457: Class-File API (Preview)
458: Launch Multi-File Source-Code Programs
459: String Templates (Second Preview)
460: Vector API (Seventh Incubator)
461: Stream Gatherers (Preview)
462: Structured Concurrency (Second Preview)
463: Implicitly Declared Classes and Instance Main Methods (Second Preview)
464: Scoped Values (Second Preview)

Among them, I will introduce the following three new features in detail:

- JEP 423: G1 Garbage Collector Region Pinning
- JEP 454: Foreign Function and Memory API
- JEP 456: Unnamed Patterns and Variables
- JEP 458: Launching Multi-File Source Code Programs

JDK 23

JEP 455: Primitive Types, instanceof, and Switch in Patterns (Preview)

Prior to JEP 455, `instanceof` only reference types were supported, and `switch` expressions and statement `case` labels could only use integer literals, enumeration constants, and string literals.

As a preview feature of JEP 455, `instanceof` and `switch` fully support all primitive types, including `byte` , `short` , `char` , `int` , `long` , `float` , `double` , `boolean` .

```java
// 传统写法
if (i >= -128 && i <= 127) {
    byte b = (byte)i;
    ... b ...
}

// 使用 instanceof 改进
if (i instanceof byte b) {
    ... b ...
}

long v = ...;
// 传统写法
if (v == 1L) {
    // ...
} else if (v == 2L) {
    // ...
} else if (v == 10_000_000_000L) {
    // ...
}

// 使用 long 类型的 case 标签
switch (v) {
    case 1L:
        // ...
        break;
    case 2L:
        // ...
        break;
    case 10_000_000_000L:
        // ...
        break;
    default:
        // ...
}
```

JEP 456: Class File API (Second Preview)

The class file API was first previewed in JDK 22, proposed by JEP 457 .

The goal of the class file API is to provide a standardized API for parsing, generating, and converting Java class files, replacing past reliance on third-party libraries (such as ASM) for class file processing.

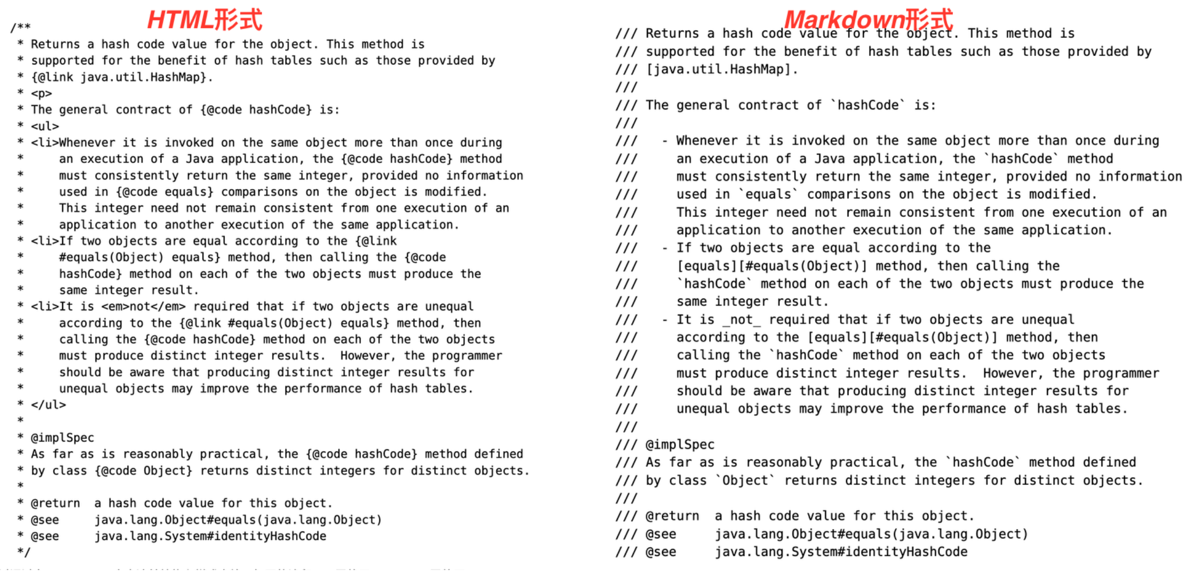
```java
// 创建一个 ClassFile 对象，这是操作类文件的入口。
ClassFile cf = ClassFile.of();
// 解析字节数组为 ClassModel
ClassModel classModel = cf.parse(bytes);

// 构建新的类文件，移除以 "debug" 开头的所有方法
byte[] newBytes = cf.build(classModel.thisClass().asSymbol(),
        classBuilder -> {
            // 遍历所有类元素
            for (ClassElement ce : classModel) {
                // 判断是否为方法 且 方法名以 "debug" 开头
                if (!(ce instanceof MethodModel mm
                        &&
mm.methodName().stringValue().startsWith("debug"))) {
                    // 添加到新的类文件中
                    classBuilder.with(ce);
                }
            }
        });
```

JEP 467: Markdown Documentation Comments

In JavaDoc documentation comments, you can use Markdown syntax instead of using only HTML and JavaDoc tags.

Markdown is more concise and readable, reducing the tediousness of manually writing HTML while retaining support for HTML elements and JavaDoc tags. This enhancement is designed to make writing and reading API documentation comments easier without affecting the interpretation of existing comments. Markdown provides simplified representations of common documentation elements (such as paragraphs, lists, and links), improving the maintainability of documentation comments and the developer experience.



```
/**
 * Returns a hash code value for the object. This method is
 * supported for the benefit of hash tables such as those provided by
 * {@link java.util.HashMap}.
 * <p>
 * The general contract of {@code hashCode} is:
 * <ul>
 * <li>Whenever it is invoked on the same object more than once during
 *     an execution of a Java application, the {@code hashCode} method
 *     must consistently return the same integer, provided no information
 *     used in {@code equals} comparisons on the object is modified.
 *     This integer need not remain consistent from one execution of an
 *     application to another execution of the same application.
 * <li>If two objects are equal according to the {@link
 *     #equals(Object) equals} method, then calling the {@code
 *     hashCode} method on each of the two objects must produce the
 *     same integer result.
 * <li>It is <em>not</em> required that if two objects are unequal
 *     according to the {@link #equals(Object) equals} method, then
 *     calling the {@code hashCode} method on each of the two objects
 *     must produce distinct integer results.  However, the programmer
 *     should be aware that producing distinct integer results for
 *     unequal objects may improve the performance of hash tables.
 * </ul>
 *
 * @implSpec
 * As far as is reasonably practical, the {@code hashCode} method defined
 * by class {@code Object} returns distinct integers for distinct objects.
 *
 * @return  a hash code value for this object.
 * @see     java.lang.Object#equals(java.lang.Object)
 * @see     java.lang.System#identityHashCode
 */
```

```
/// Returns a hash code value for the object. This method is
/// supported for the benefit of hash tables such as those provided by
/// [java.util.HashMap].
///
/// The general contract of `hashCode` is:
///
///   - Whenever it is invoked on the same object more than once during
///     an execution of a Java application, the `hashCode` method
///     must consistently return the same integer, provided no information
///     used in `equals` comparisons on the object is modified.
///     This integer need not remain consistent from one execution of an
///     application to another execution of the same application.
///   - If two objects are equal according to the
///     [equals][#equals(Object)] method, then calling the
///     `hashCode` method on each of the two objects must produce the
///     same integer result.
///   - It is _not_ required that if two objects are unequal
///     according to the [equals][#equals(Object)] method, then
///     calling the `hashCode` method on each of the two objects
///     must produce distinct integer results.  However, the programmer
///     should be aware that producing distinct integer results for
///     unequal objects may improve the performance of hash tables.
///
/// @implSpec
/// As far as is reasonably practical, the `hashCode` method defined
/// by class `Object` returns distinct integers for distinct objects.
///
/// @return  a hash code value for this object.
/// @see     java.lang.Object#equals(java.lang.Object)
/// @see     java.lang.System#identityHashCode
```

JEP 469: Vector API (8th incubation)

Vector computations consist of a series of operations on vectors. The Vector API is used to express vector computations that can be reliably compiled into optimal vector instructions on the supported CPU architecture at runtime, achieving performance superior to equivalent scalar computations.

The goal of the Vector API is to provide users with a wide range of vector computations that are concise, easy to use, and platform-independent.

This is a simple scalar calculation on the elements of an array:

```java
void scalarComputation(float[] a, float[] b, float[] c) {
   for (int i = 0; i < a.length; i++) {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
   }
}
```

Here is the equivalent vector calculation using the Vector API:

```java
static final VectorSpecies<Float> SPECIES =
FloatVector.SPECIES_PREFERRED;

void vectorComputation(float[] a, float[] b, float[] c) {
    int i = 0;
    int upperBound = SPECIES.loopBound(a.length);
    for (; i < upperBound; i += SPECIES.length()) {
        // FloatVector va, vb, vc;
```

```java
            var va = FloatVector.fromArray(SPECIES, a, i);
            var vb = FloatVector.fromArray(SPECIES, b, i);
            var vc = va.mul(va)
                       .add(vb.mul(vb))
                       .neg();
            vc.intoArray(c, i);
        }
        for (; i < a.length; i++) {
            c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
        }
    }
```

JEP 473: Stream Collectors (Second Preview)

The stream collector was first previewed in JDK 22, proposed by JEP 461 .

This improvement enables the Stream API to support custom intermediate operations.

```java
source.gather(a).gather(b).gather(c).collect(...)
```

JEP 471: Deprecate memory access methods in sun.misc.Unsafe

JEP 471 proposes to deprecate `sun.misc.Unsafe` memory access methods in , which will be removed in a future version.

These unsafe methods have safe and effective alternatives:

- `java.lang.invoke.VarHandle` : Introduced in JDK 9 (JEP 193), it provides a safe and efficient way to operate heap memory, including object fields, class static fields, and array elements.
- `java.lang.foreign.MemorySegment` : Introduced in JDK 22 (JEP 454), it provides a way to safely and efficiently access off-heap memory, and sometimes `VarHandle` works in conjunction with .

These two classes are core components of the Foreign Function & Memory API, used to manage and manipulate off-heap memory, respectively. The Foreign Function & Memory API was officially made a standard feature in JDK 22.

```java
import jdk.incubator.foreign.*;
import java.lang.invoke.VarHandle;

// 管理堆外整数数组的类
class OffHeapIntBuffer {

    // 用于访问整数元素的VarHandle
    private static final VarHandle ELEM_VH =
ValueLayout.JAVA_INT.arrayElementVarHandle();

    // 内存管理器
    private final Arena arena;

    // 堆外内存段
    private final MemorySegment buffer;

    // 构造函数，分配指定数量的整数空间
    public OffHeapIntBuffer(long size) {
        this.arena  = Arena.ofShared();
        this.buffer = arena.allocate(ValueLayout.JAVA_INT, size);
    }

    // 释放内存
    public void deallocate() {
        arena.close();
    }

    // 以volatile方式设置指定索引的值
    public void setVolatile(long index, int value) {
        ELEM_VH.setVolatile(buffer, 0L, index, value);
    }

    // 初始化指定范围的元素为0
    public void initialize(long start, long n) {
        buffer.asSlice(ValueLayout.JAVA_INT.byteSize() * start,
                       ValueLayout.JAVA_INT.byteSize() * n)
              .fill((byte) 0);
    }

    // 将指定范围的元素复制到新数组
    public int[] copyToNewArray(long start, int n) {
        return buffer.asSlice(ValueLayout.JAVA_INT.byteSize() *
start,
                                      ValueLayout.JAVA_INT.byteSize() * n)
                        .toArray(ValueLayout.JAVA_INT);
    }
}
```

JEP 474: ZGC: Generational Mode by Default

The Z Garbage Collector (ZGC) has switched to generational mode by default, deprecating non-generational mode and planning to remove it in a future release. This is because generational ZGC is a better choice in most scenarios.

JEP 476: Module Import Declarations (Preview)

Module import declarations allow you to concisely import all exported packages of an entire module from your Java code, without having to declare package imports individually. This feature simplifies the reuse of modular libraries, especially when working with multiple modules, by avoiding the need for numerous package import declarations and making it easier for developers to access third-party libraries and Java base classes.

This feature is especially useful for beginners and prototype development because it does not require developers to modularize their code while retaining compatibility with traditional import methods, improving development efficiency and code readability.

```java
// 导入整个 java.base 模块，开发者可以直接访问 List、Map、Stream 等类，而无需每次手动导入相关包
import module java.base;

public class Example {
    public static void main(String[] args) {
        String[] fruits = { "apple", "berry", "citrus" };
        Map<String, String> fruitMap = Stream.of(fruits)
            .collect(Collectors.toMap(
                s -> s.toUpperCase().substring(0, 1),
                Function.identity()));

        System.out.println(fruitMap);
    }
}
```

JEP 477: Unnamed Class and Instance Main Methods (Preview 3)

This feature mainly simplifies `main` the method declaration. For Java beginners, this `main` method declaration introduces too many Java syntax concepts, which is not conducive to beginners getting started quickly.

Without using this feature before defining a `main` method:

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

To use this new feature, define a `main` method:

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

Simplifying further (unnamed classes allow us to omit the class name)

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

JEP 480: Structured Concurrency (Third Preview)

Java 19 introduces structured concurrency, a multi-threaded programming method. Its purpose is to simplify multi-threaded programming through structured concurrency APIs, not to replace them `java.util.concurrent` . It is currently in the incubator stage.

Structured concurrency treats multiple tasks running in different threads as a single u work, simplifying error handling, improving reliability, and enhancing observability. In other words, structured concurrency preserves the readability, maintainability, and observability of single-threaded code.

The basic API of structured concurrency is `StructuredTaskScope` to `StructuredTaskScope` support splitting a task into multiple concurrent subtasks, which are executed in their own threads, and the subtasks must complete before the main task continues.

`StructuredTaskScope` The basic usage is as follows:

```java
    try (var scope = new StructuredTaskScope<Object>()) {
        // 使用fork方法派生线程来执行子任务
        Future<Integer> future1 = scope.fork(task1);
        Future<String> future2 = scope.fork(task2);
        // 等待线程完成
        scope.join();
        // 结果的处理可能包括处理或重新抛出异常
        ... process results/exceptions ...
    } // close
```

Structured concurrency is well suited for virtual threads, which are lightweight threads implemented by the JDK. Many virtual threads share the same operating system thread, allowing for a very large number of virtual threads.

JEP 481: Scoped Values (Preview 3)

Scoped Values can share immutable data within and between threads, which is better than thread-local variables, especially when using a large number of virtual threads.

```java
final static ScopedValue<...> V = new ScopedValue<>();

// In some method
ScopedValue.where(V, <value>)
           .run(() -> { ... V.get() ... call methods ... });

// In a method called directly or indirectly from the lambda expression
... V.get() ...
```

Scoped values allow data to be shared safely and efficiently between components in a larger program without resorting to method parameters.

JEP 482: Flexible Constructor Bodies (Second Preview)

This feature was first proposed in JDK 22 by JEP 447: Statements before super(...) (Preview) .

Java requires that in a constructor, `super(...)` or `this(...)` call must appear as the first statement. This means that we cannot initialize fields directly in the subclass constructor before calling the superclass constructor.

Flexible constructor bodies solve this problem by allowing statements to be written within the constructor body before calling `super(..)` or `this(..)` . These statements can initialize fields but cannot reference the instance being constructed. This prevents subclass fields from being incorrectly initialized when calling subclass methods in the parent class constructor, enhancing the reliability of class construction.

This feature solves the problem that Java syntax previously restricted the organization of constructor code, allowing developers to express the behavior of constructors more freely and naturally. For example, they can directly validate, prepare, and share parameters in the constructor without relying on auxiliary methods or constructors, improving the readability and maintainability of the code.

```java
class Person {
    private final String name;
    private int age;

    public Person(String name, int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative.");
        }
        this.name = name; // 在调用父类构造函数之前初始化字段
        this.age = age;
        // ... 其他初始化代码
    }
}

class Employee extends Person {
    private final int employeeId;

    public Employee(String name, int age, int employeeId) {
        this.employeeId = employeeId; // 在调用父类构造函数之前初始化字段
        super(name, age); // 调用父类构造函数
        // ... 其他初始化代码
    }
}
```

JDK 22

JEP 423: G1 Garbage Collector Region Pinning

JEP 423 proposes to implement Region Pinning in the G1 garbage collector, aiming to reduce latency issues caused by Java Native Interface (JNI) critical regions.

Objects within JNI critical regions cannot be moved during garbage collection. G1 previously addressed this by disabling garbage collection, which resulted in blocked threads and significant latency. By introducing a region pinning mechanism in G1's old and young generations, the memory regions containing objects within critical regions can be pinned while continuing to reclaim unpinned regions, eliminating the need to disable garbage collection. This improvement significantly reduces latency, improving system throughput and stability when interacting with the JNI.

JEP 454: Foreign Function and Memory API

This API enables Java programs to interoperate with code and data outside the Java runtime. By efficiently calling external functions (that is, code outside the JVM) and safely accessing external memory (that is, memory not managed by the JVM), this API enables Java programs to call native libraries and process native data without the risks and brittleness of JNI.

The foreign function and memory APIs went through their first incubation phase in Java 17, as proposed by JEP 412. They went through their second incubation phase in Java 18, as proposed by JEP 419. They received their first preview in Java 19, as proposed by JEP 424. They received their second preview in JDK 20, as proposed by JEP 434. And they received their third preview in JDK 21, as proposed by JEP 442 .

Finally, the feature was successfully formalized in JDK 22.

In the Java 19 New Features Overview , I introduced the external function and memory API in detail, so I will not give additional introductions here.

JEP 456: Unnamed Patterns and Variables

Unnamed patterns and variables were previewed in JDK 21 by JEP 443 and formalized in JDK 22.

For a detailed introduction to this new feature, see the introduction in the article Overview of New Features in Java 21 (Important) .

JEP 458: Launching Multi-File Source Code Programs

Java 11 introduces JEP 330: Launching Single-File Source Code Programs , which enhances `java` the launcher's functionality to enable it to directly run a single Java source file. Through commands `java HelloWorld.java` , Java can implicitly compile source code in memory and execute it immediately, without generating files on disk `.class` . This simplifies the developer workflow when writing small utility programs or learning Java, eliminating the extra step of manual compilation.

Suppose the file `Prog.java` declares two classes:

```java
class Prog {
    public static void main(String[] args) { Helper.run(); }
}

class Helper {
    static void run() { System.out.println("Hello!"); }
}
```

`java Prog.java` The command compiles two classes in memory and executes `main` the methods of the first class declared in the file.

This approach has a limitation: all source code for a program must be placed `.java` in one file.

JEP 458: Launching Multi-File Source Code Programs is an extension of the functionality of JEP 330 that allows programs composed of multiple Java source files to be run directly without an explicit compilation step.

Suppose there are two Java source files in a directory , `Prog.java` and `Helper.java` each file declares a class:

```java
// Prog.java
class Prog {
    public static void main(String[] args) { Helper.run(); }
}

// Helper.java
class Helper {
    static void run() { System.out.println("Hello!"); }
}
```

When you run the command , the Java launcher compiles and executes the __init__ method of the __init__ class `java Prog.java` in memory . Because the code in the __init__ class references the __init__ class, the launcher automatically locates the __init__ file on the file system , compiles the __init__ class within it , and executes it in memory. This process is automatic, and developers do not need to explicitly call __init__ to compile all source files. `Prog` `main` `Prog` `Helper` `Helper.java` `Helper` `javac`

This feature makes the transition from small to large projects smoother, allowing developers to freely choose when to introduce build tools and avoid being forced to set up complex project structures for rapid iteration. This feature eliminates the limitation of a single file and further simplifies the development process from a single file to a multi-file program, making it particularly suitable for prototyping, rapid experimentation, and the exploration stage of early projects.

Recently Updated2025/3/20 14:56
Contributors: Guide