Java 16 Features

**Date 27/04/2023**

# Overview of Java 16 (Mar 16, 2021)

1. JEP 394 - Pattern Matching for Instanceof
2. Stream API Enhancements
3. JEP 397 - Sealed Class Enhancement
4. JEP 338 - Vector API
5. JEP 389 - Foreign Linker API
6. Date Period Support

# JEP 394 - Pattern Matching for InstanceOf

**Pattern Matching for instanceof - Scope**

A matched variable is only visible within the if block. That is logical, because only if the if comparison is positive, the variable can be cast to the desired type.

If a field with the same name exists within the class, then this field is "shadowed" by a pattern matching variable. The following example shows what this means,

public class **PatternMatchingScopeTest** {

public static void main(String[] args) {

new PatternMatchingScopeTest().processObject("Happy Coding!");

}

private String s = "Hello, world!";

private void processObject(Object obj) {

System.out.println(s); // **Prints "Hello, world!"**

if (obj instanceof String s) {

System.out.println(s); // **Prints "Happy Coding!"**

System.out.println(this.s); // **Prints "Hello, world!"**

}

}

}

**What does this program print?**

In line 10, the field s defined in line 7 is printed.

Line 12 prints the variable s assigned in the instanceof expression, that is, the object obj, which was passed to the method, cast to a String.

To access the field s within the if block, we use this.s in line 13.

It is not allowed to give a pattern matching variable the same name as a variable already defined in the method, as in the following example:

private void processObject(Object obj) {

String s = "Hello, world";

if (obj instanceof String s) { **// Compiler error**

...

}

}

**Pattern Matching for instanceof - Changes in Java 16**

Compared to the first two previews in Java 14 and Java 15, two refinements have been made for the final release,

1. Pattern variables are no longer implicitly final, i.e., they can be changed. The following code is allowed in Java 16, in Java 15, it led to a "pattern binding may not be assigned" compiler error.

**if (obj instanceof String s && s.length() > 5) {**

**s = s.toUpperCase(); // Compiler error in Java 15, allowed in Java 16**

**System.out.println(s);**

**} else if (obj instanceof Integer i) {**

**i = i \* i; // Compiler error in Java 15, allowed in Java 16**

**System.out.println(i);**

**}**

1. "Pattern Matching for instanceof" expression results in a compiler error when comparing an expression of type S with a pattern of type T, where S is a subtype of T.

**private static void processInteger(Integer i) {**

**if (i instanceof Number n) { // Compiler error in Java 16**

**// ...**

**}**

**}**

The concrete error message in this example is "pattern type Number is a subtype of expression type Integer". What exactly does that mean?

Since Integer inherits from Number, both the instanceof check and the cast to Number are superfluous. The Integer object can be used without a cast in all places where Number is expected.

public class \_01\_InstanceOf\_Demo1 {

public static void main(String[] args) {

new \_01\_InstanceOf\_Demo1().processObject("Happy Coding!");

}

private String str = "Hello, World!";

private void processObject(Object obj) {

System.***out***.println(str);

if (obj instanceof String str) {

System.***out***.println(str);

System.***out***.println(this.str);

}

}

/\*\*

\* It is not allowed to give a pattern matching variable the same name as a

\* variable already defined in the method, as in the following code.

\*/

// private void processObject1(Object obj) {

// String str = "Hello, World!";

// System.out.println(str);

// if (obj instanceof String str) { // Compile Time Error

// System.out.println(str);

// System.out.println(this.str);

// }

// }

}

public class \_02\_InstanceOf\_Demo2 {

public static void main(String[] args) {

Object value = "Java16 Pattern Matching";

if (value instanceof String str && str.length() > 5) {

str = str.toUpperCase(); // Compiler error in Java 15, allowed in Java 16

System.***out***.println(str);

}

}

}

public class \_03\_InstanceOf\_Demo3 {

public static void main(String[] args) {

Integer value = 10;

if (value instanceof Number n) {

System.***out***.println(n);

}

}

}

# Stream API Enhancements

Java 16 introduces the following two new Stream methods,

**Stream.toList() and Stream.mapMulti()**

## Stream.toList()

If you wanted to terminate a stream into a list, you had the following options up to now:

**// ArrayList:**

**Stream.of("foo", "bar", "baz").collect(Collectors.toList());**

**// ImmutableCollectionsList:**

**Stream.of("foo", "bar", "baz").collect(Collectors.toUnmodifiableList());**

**// LinkedList:**

**Stream.of("foo", "bar", "baz").collect(Collectors.toCollection(LinkedList::new));**

The return types of the first two variants are not guaranteed. In fact, for the first variant Collectors.toList(), the list is not even guaranteed to be modifiable. With the second variant Collectors.toUnmodifiableList(), it is at least guaranteed that the return value is an unmodifiable list.

Stream.toList() is a fourth variant that also generates an unmodifiable list:

**// ImmutableCollectionsList:**

**Stream.of("foo", "bar", "baz").toList();**

This method is implemented as a default method in the Stream interface and is overridden by a stream-specific optimization in most stream implementations.

public class \_01\_Stream\_Enhancement\_Demo1 {

public static void main(String[] args) {

List<String> integerAsString = Arrays.*asList*("1", "2", "3", "4", "5");

List<Integer> intConversion = integerAsString.stream().map(Integer::*parseInt*).toList();

System.***out***.println(intConversion);

List<String> names = List.*of*("John", "Alex", "Blake", "Erica");

List<String> immutableList = names.stream().filter(e -> e.contains("a")).toList();

System.***out***.println(immutableList);

}

}

## Stream.mapMulti()

To merge collections contained in a stream into a single collection, we usually use flatMap():

**Stream<List<Integer>> stream =Stream.of(List.of(1, 2, 3), List.of(4, 5, 6), List.of(7, 8, 9));**

**List<Integer> list = stream.flatMap(List::stream).toList();**

As a parameter to flatMap(), we need to specify a mapper function that converts each collection contained in the stream into an intermediate stream.

This example was highly simplified. The stream does not have to contain collections directly. For example, it could also contain Customer objects whose getOrders() method returns a list of orders. We could then use flatMap() to compile a list of all the customer’s orders.

**List<Customer> customers = getCustomers();**

**List<Order> allOrders = customers.stream().flatMap(customer -> customer.getOrders().stream())**

**.toList();**

Both examples have in common that a new stream is generated for each element of the original stream. This is subject to a particular overhead.

**Therefore, in Java 16, Stream.mapMulti() was introduced as a more efficient, imperative alternative to the declarative flatMap(); While with flatMap(), we specify which data we want to merge, with mapMulti() we implement how to merge this data.**

For this, we pass a **BiConsumer** to which the following two elements are given during the mapping process.

The element of the stream, i.e., the collection to be collected (the list in the first example) or the object from which a collection is extracted (the customer in the second example).

A Consumer to which we pass the elements of the collection one by one.

Here is the first example converted to mapMulti(),

**List<Integer> list = stream.mapMulti((List<Integer> numbers, Consumer<Integer> consumer) ->**

**numbers.forEach(number -> consumer.accept(number))).toList();**

We can replace the lambda body with a single method reference,

**List<Integer> list = stream.mapMulti((BiConsumer<List<Integer>, Consumer<Integer>>) Iterable::forEach).toList();**

What we are saying here is, Iterate over each of the elements of the lists contained in the stream and pass all the individual elements to the provided Consumer. The intermediate step of creating a new stream per list is omitted.

And here is the second example,

**List<Order> allOrders = customers.stream().mapMulti((Customer customer, Consumer<Order> consumer) -> customer.getOrders().forEach(consumer)).toList();**

We iterate over each customer's orders and pass them to the provided Consumer.

**Should we now always use mapMulti() instead of flatMap()?** No, mapMulti() is just another tool in our toolbox. We should generally not optimize prematurely and use whichever method is most readable in a given case. In the examples above, I would stick with flatMap().

public class \_02\_Stream\_Enhancement\_Demo2 {

public static void main(String[] args) {

Stream<List<Integer>> stream = Stream.*of*(List.*of*(1, 2, 3), List.*of*(4, 5, 6), List.*of*(7, 8, 9));

// List<Integer> list = stream.flatMap(List::stream).toList();

// System.out.println(list);

List<Object> list = stream.mapMulti((numbers, consumer) -> numbers.forEach(number -> consumer.accept(number)))

.toList();

System.***out***.println(list);

List<String> names = List.*of*("John", "Alex", "Blake", "Erica");

names.stream().mapMulti((name, consumer) -> consumer.accept(name)).forEach(System.***out***::println);

}

}

# JEP 397 - Sealed Class Enhancement

Sealed classes were introduced in Java 15 as a preview. With JDK Enhancement Proposal 397, three small changes have been made for Java 16.

1. In the Java Language Specification (JLS), the concept of "**contextual keywords**" replaces the previous "**restricted identifiers**" and "**restricted keywords**". "Contextual keywords ensure that new keywords such as sealed, permits (or yield from the switch expressions) may continue to be used outside the respective context, e.g., as variable or method names.

public class \_01\_Sealed\_Class\_Demo1 {

public static void main(String[] args) {

*sealed*();

}

public static void sealed() {

int permits = 5;

System.***out***.println(permits);

}

}

2. The permits keyword can be omitted if subclasses derived from a sealed class are defined within the same class file ("compilation unit"). These are then considered "implicitly declared permitted subclasses".

public class \_02\_Sealed\_Class\_Demo2 {

public static void main(String[] args) {

ChildClass obj = new ChildClass("John", 101);

System.***out***.println(obj.getName());

System.***out***.println(obj.getId());

}

}

sealed class SealedClassWithoutPermits {

String name;

public String getName() {

return name;

}

}

non-sealed class ChildClass extends SealedClassWithoutPermits {

int id;

ChildClass(String name, int id) {

this.name = name;

this.id = id;

}

public int getId() {

return id;

}

}

What has been changed in the second preview of Sealed Classes is that local classes (that is, classes defined within methods) are not allowed to extend sealed classes.

public class \_03\_Sealed\_Class\_Demo3 {

public static void main(String[] args) {

}

}

sealed class Car {

public static void display() {

// final class BMW extends Car { // Compile Time Error

//

// }

}

}

final class Audi extends Car {

}

3. For instanceof tests, the compiler checks whether the class hierarchy allows the check ever to return true. Since the second preview of Sealed Classes, the information from sealed class hierarchies is included in this check.

public class \_03\_Sealed\_Class\_Demo3 {

public static void main(String[] args) {

FastProcess fp = new FastProcess();

SlowProcess sp = new SlowProcess();

fp.getFastProcess();

sp.getSlowProcess();

}

}

sealed class Process permits SlowProcess, FastProcess {

}

final class SlowProcess extends Process {

public void getSlowProcess() {

System.***out***.println("Slow Process");

}

}

non-sealed class FastProcess extends Process {

public void getFastProcess() {

System.***out***.println("Fast Process");

}

}

/\*

\* The instanceof expression in the below code tests a reference of type Process

\* to see if the object referred to by that reference implements the Loggable

\* interface. Now, in the typical situation, all types that can ever be

\* assignment-compatible with a sealed class will be known to the compiler, and

\* if none implements the interface, the compiler could reject the

\* code outright.

\*

\* However, in this case, the FastProcess class is marked non-sealed, and

\* because of this, it’s possible for this class to have subclasses that are not

\* known to the compiler at this time, and those subclasses might possibly

\* implement the Loggable interface. Because of this possibility, the compiler

\* accepts the code and compiles the test as written.

\*/

interface Loggable {

static boolean isLoggable(Process p) {

return p instanceof Loggable;

}

}

# JEP 338 - Vector API

The vector API, first introduced as an incubator feature with JDK Enhancement Proposal 338, allows us to implement such operations in Java. The JVM will map them to the most efficient CPU instructions of the underlying hardware architecture.

The Vector API is in its initial incubation phase for Java 16. The idea of this API is to provide a means of vector computations that will ultimately be able to perform more optimally (on supporting CPU architectures) than the traditional scalar method of computations.

Let's look at how we might traditionally multiply two arrays:

int[] a = {1, 2, 3, 4};

int[] b = {5, 6, 7, 8};

var c = new int[a.length];

for (int i = 0; i < a.length; i++) {

c[i] = a[i] \* b[i];

}

This example of a scalar computation will, for an array of length 4, execute in 4 cycles. Now, let's look at the equivalent vector-based computation:

int[] a = {1, 2, 3, 4};

int[] b = {5, 6, 7, 8};

var vectorA = IntVector.fromArray(IntVector.SPECIES\_128, a, 0);

var vectorB = IntVector.fromArray(IntVector.SPECIES\_128, b, 0);

var vectorC = vectorA.mul(vectorB);

vectorC.intoArray(c, 0);

The first thing we do in the vector-based code is to create two IntVectors from our input arrays using the static factory method of this class fromArray. The first parameter is the size of the vector, followed by the array and the offset (here set to 0). The most important thing here is the size of the vector that we're getting to 128 bits. In Java, each int takes 4 bytes to hold.

Since we have an input array of 4 ints, it takes 128 bits to store. Our single Vector can store the whole array.

On certain architectures, the compiler will be able to optimize the byte code to reduce the computation from 4 to only 1 cycle. These optimizations benefit areas such as machine learning and cryptography.

# JEP 389 - Foreign Linker API

Since Java 1.1, the Java Native Interface (JNI) has enabled access to native C code from Java. Anyone who has used JNI knows that it is complex, error-prone, and slow. You have to write a lot of Java and C boilerplate code and keep it in sync, which is complicated even with tool support.

To replace JNI with a more modern API, Project Panama was launched.

The Foreign Linker API (JDK Enhancement Proposal 389), together with the Foreign-Memory Access API, introduced as an incubator feature in Java 14 and further refined in Java 15 and Java 16 (JDK Enhancement Proposal 393), provide this replacement.

The Panama developers have set the following goals:

* The previously time-consuming and error-prone process is simplified (the target is to reduce 90% of the effort).
* The performance is significantly increased compared to JNI (the target is a factor of 4 to 5).

Foreign Linker API and Foreign-Memory Access API will be merged into the "Foreign Function & Memory API" in Java 17. It will remain in incubator status until Java 18 and reach the preview stage in Java 19.

# Date Period Support

A new addition to the DateTimeFormatter is the period-of-day symbol “B“, which provides an alternative to the am/pm format. With the DateTimeFormatter class, you can format date values of the Java Date/Time API, e.g., LocalDate, LocalTime, LocalDateTime, or Instant, Year, and YearMonth.

You can, for example, format the current time as follows:

DateTimeFormatter.ofPattern("EEEE, MMMM d, yyyy, h:mm a", Locale.US)

.format(LocalDateTime.now());

**The result is, Wednesday, December 1, 2021, 9:14 PM**

In Java 16, the list of available format characters has been extended by the letter "B", which stands for a prolonged form of the time of day:

DateTimeFormatter.ofPattern("EEEE, MMMM d, yyyy, h:mm B", Locale.US).format(LocalDateTime.now());

**The generated string is, Wednesday, December 1, 2021, 9:16 at night**

[**https://www.baeldung.com/java-datetimeformatter**](https://www.baeldung.com/java-datetimeformatter)

# References

<https://www.happycoders.eu/java/java-16-features/>

<https://www.tutorialspoint.com/java16/index.htm>

<https://www.baeldung.com/java-16-new-features>