Java16

Java 16, released on the 16th of March 2021, is the latest short-term incremental release building on Java 15. This release comes with some interesting features, such as records and sealed classes.

1.. Invoke Default Methods From Proxy Instances (JDK-8159746)

As an enhancement to the default method in Interfaces, with the release of Java 16, support has been added to **java.lang.reflect.InvocationHandler** invoke default methods of an interface via a dynamic proxy using reflection. To illustrate this, let’s look at a simple default method example:

```
interface HelloWorld {
    default String hello() {
        return "world";
    }
}
```

With this enhancement, we can invoke the default method on a proxy of that interface using reflection:

```
Object proxy = Proxy.newProxyInstance(getSystemClassLoader(), new Class<?>[] { HelloWorld.class },
    (prox, method, args) -> {
        if (method.isDefault()) {
            return InvocationHandler.invokeDefault(prox, method, args);
        }
        // ...
    }
);
Method method = proxy.getClass().getMethod("hello");
assertThat(method.invoke(proxy)).isEqualTo("world");
```

2.. Day Period Support (JDK-8247781)

A new addition to the **DateTimeFormatter** is the period-of-day symbol “B“, which provides an alternative to the am/pm format:

```
LocalTime date = LocalTime.parse("15:25:08.690791");
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("h B");
assertThat(date.format(formatter)).isEqualTo("3 in the afternoon");
```

Instead of something like “3pm“, we get an output of “3 in the afternoon“. We can also use the “B“, “BBBB“, or “BBBBB” DateTimeFormatter pattern for short, full, and narrow styles respectively.

3.. Add Stream.toList Method (JDK-8180352)

The aim is to reduce the boilerplate with some commonly used Stream collectors, such as Collectors.toList and Collectors.toSet:

```
List<String> integersAsString = Arrays.asList("1", "2", "3");
List<Integer> ints = integersAsString.stream().map(Integer::parseInt).collect(Collectors.toList());
List<Integer> intsEquivalent = integersAsString.stream().map(Integer::parseInt).toList();
```

Our ints example works the old way, but the ints Equivalent has the same result and is more concise.

4.. Vector API Incubator (JEP-338)

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.

We should note that being in the incubation stage means this Vector API is subject to change with newer releases.

5.. Records (JEP-395)

Records were introduced in Java 14. Java 16 brings some incremental changes. Records are similar to enum in the fact that they are a restricted form of class. Defining a record is a concise way of defining an immutable data holding object.

5.1 Example Without Records

First, let's define a Book class:

```java
public final class Book {
    private final String title;
    private final String author;
    private final String isbn;

    public Book(String title, String author, String isbn) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public String getIsbn() {
        return isbn;
    }

    @Override
    public boolean equals(Object o) {
        // ...
    }

    @Override
    public int hashCode() {
        return Objects.hash(title, author, isbn);
    }
}
```

Creating simple data holding classes in Java requires a lot of boilerplate code. This can be cumbersome and lead to bugs where developers don't provide all the necessary methods, such as equals and hashCode. Similarly, sometimes developers skip the necessary steps for creating proper immutable classes. Sometimes we end up reusing a general-purpose class rather than defining a specialist one for each different use case. Most modern IDEs provide an ability to auto-generate code (such as setters, getters, constructors, etc.) that helps mitigate these issues and reduces the overhead on a developer writing the code. However, Records provide an inbuilt mechanism to reduce the boilerplate code and create the same result.

5.2 Example with Records

Here is Book re-written as a Record:

```java
public record Book(String title, String author, String isbn) {
}
```

By using the record keyword, we have reduced the Book class to two lines. This makes it a lot easier and less error-prone.

5.3 New Additions to Records in Java 16

With the release of Java 16, we can now define records as class members of inner classes. This is due to relaxing restrictions that were missed as part of the incremental release of Java 15 under JEP-384:

```
class OuterClass {
    class InnerClass {
        Book book = new Book("Title", "author", "isbn");
    }
}
```

6.. Pattern Matching for instanceof (JEP-394)

Pattern matching for the instanceof keyword has been added as of Java 16. Previously we might write code like this:

```
Object obj = "TEST";

if (obj instanceof String) {
    String t = (String) obj;
    // do some logic...
}
```

Instead of purely focusing on the logic needed for the application, this code must first check the instance of obj, then cast the object to a String and assign it to a new variable t. With the introduction of pattern matching, we can re-write this code:

```
Object obj = "TEST";

if (obj instanceof String t) {
    // do some logic
}
```

We can now declare a variable – in this instance t – as part of the instanceof check.

7.. Sealed Classes (JEP-397)

Sealed classes, first introduced in Java 15, provide a mechanism to determine which sub-classes are allowed to extend or implement a parent class or interface.

Let's illustrate this by defining an interface and two implementing classes:

```
public sealed interface JungleAnimal permits Monkey, Snake  {
}

public final class Monkey implements JungleAnimal {
}

public non-sealed class Snake implements JungleAnimal {
}
```

The sealed keyword is used in conjunction with the permits keyword to determine exactly which classes are allowed to implement this interface. In our example, this is Monkey and Snake. All inheriting classes of a sealed class must be marked with one of the following:

- sealed – meaning they must define what classes are permitted to inherit from it using the permits keyword.
- final – preventing any further subclasses
- non-sealed – allowing any class to be able to inherit from it.

A significant benefit of sealed classes is that they allow for exhaustive pattern matching checking without the need for a catch for all non-covered cases. For example, using our defined classes, we can have logic to cover all possible subclasses of JungleAnimal:

```java
JungleAnimal j = // some JungleAnimal instance

if (j instanceof Monkey m) {
    // do logic
} else if (j instanceof Snake s) {
    // do logic
}
```

We don’t need an else block as the sealed classes only allow the two possible subtypes of Monkey and Snake.

7.1 New Additions to Sealed Classes in Java 16

There are a few additions to sealed classes in Java 16. These are the changes that Java 16 introduces to the sealed class: The Java language recognizes sealed, non-sealed, and permits as contextual keywords (similar to abstract and extends) Restrict the ability to create local classes that are subclasses of a sealed class (similar to the inability to create anonymous classes of sealed classes). Stricter checks when casting sealed classes and classes derived from sealed classes

8.. Other Changes

Continuing from JEP-383 in the Java 15 release, the foreign linker API provides a flexible way to access native code on the host machine. Initially, for C language interoperability, in the future, it may be adaptable to other languages such as C++ or Fortran. The goal of this feature is to eventually replace the Java Native Interface. Another important change is that JDK internals is now strongly encapsulated by default. These have been accessible since Java 9. However, now the JVM requires the argument –illegal-access=permit. This will affect all libraries and apps (particularly when it comes to testing) that are currently using JDK internals directly and simply ignoring the warning messages.