Java 16

***Invoke Default Methods From Proxy Instances*** *-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");

***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.*

***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();**

***Vector API Incubator (JEP-338) -*** *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.*

**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.*

***New Additions to Records -*** *Now define records as class members of inner classes****.***

**class** **OuterClass** {

**class** **InnerClass** {

**Book** book = **new** **Book**("Title", "author", "isbn");

}

}

***Pattern Matching for instanceof*** *–*

**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.*

***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.*

***Other Changes*** *- 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.*