Java 15

***Sealed Classes*** *- Sealed classes let you restrict or permit class hierarchies to only certain types. sealed classes in Java offer a mechanism for controlling class hierarchies, which helps prevent errors and improves code quality. By explicitly specifying which classes can extend a sealed class, you can write more predictable and maintainable code. This is incredibly useful for pattern matching as you have a specific number of classes to switch between.*

*The following syntax defines a sealed class in Java 15.*

public sealed class Vehicle permits Car, Bike, Truck {

...

}

*The above code means, only the classes defined after the keyword permits are allowed to extend the Vehicle sealed class. In case, you’ve classes Car, Bike and Truck in the same file as Vehicle, you can omit the keyword permits.*

sealed class Vehicle {...}

final class Car extends Vehicle {...}

final class Bike extends Vehicle {...}

final class Truck extends Vehicle {...}

*Every permitted class must be set with an explicit modifier. It can either be final or sealed or non-sealed.*

*Here’s how each of the modifiers impact inheritance:*

* *A permitted subclass that’s declared final cannot be extended further.*
* *A permitted subclass that’s declared sealed can be extended further but only by classes that are permitted by the subclass.*
* *A permitted subclass may be declared non-sealed can be extended further by any class. The superclass cannot restrict the subclasses further down this class hierarchy.*

*Java’s Reflection API also gets two new methods for dealing with sealed classes:*

java.lang.constant.ClassDesc[] getPermittedSubclasses();

boolean isSealed()

***Records (Second Preview) –*** *Records are meant to be data carrier classes and you should totally avoid defining native methods in them.*

* *Prior to Java 15, one could declare native methods in records(though it wasn’t a good idea). Now the JEP explicitly prohibits against declaring native methods in records. Understandably, defining a native method steals away the USP of records by bringing an external state dependency.*
* *The implicitly declared fields corresponding to the record components of a record class are final and should not be modified via reflection now as it will throw IllegalAccessException.*
* *You can define a sealed interface and implement them on your records in the following ways.*

sealed interface Car permits BMW, Audi { ... }

record BMW(int price) implements Car { ... }

record Audi(int price, String model) implements Car { ... }

* *Records can also be defined within methods to store intermediate values. Unlike local classes, a local record is implicitly static. This means they cannot access variables and instance members of the enclosing methods which is great since it prevents capturing of values by the record.*

List<Merchant> findTopMerchants(List<Merchant> merchants, int month) {

// Local record

record MerchantSales(Merchant merchant, double sales) {}

return merchants.stream()

.map(merchant -> new MerchantSales(merchant, computeSales(merchant, month)))

.sorted((m1, m2) -> Double.compare(m2.sales(), m1.sales()))

.map(MerchantSales::merchant)

.collect(toList());

}

***Hidden Classes (JEP 371)***

*A new feature being introduced in Java 15 is known as hidden classes. Anyone who works with dynamic bytecode or JVM languages will likely find them useful.*

*The goal of hidden classes is to allow the runtime creation of classes that are not discoverable. This means they cannot be linked by other classes, nor can they be discovered via reflection. Classes such as these typically have a short lifecycle, and thus, hidden classes are designed to be efficient with both loading and unloading.*

***Pattern Matching Type Checks (JEP 375) –***

***Garbage Collectors -*** *In Java 15, both ZGC (JEP 377) and Shenandoah (JEP 379) will be no longer be experimental. Both will be supported configurations that teams can opt to use, while the G1 collector will remain the default.*