Java 15

Java 15 reached general availability in September 2020 and is the next short-term release for the JDK platform. It builds on several features from earlier releases and also provides some new enhancements. In this post, we'll look at some of the new features of Java 15, as well as other changes that are of interest to Java developers.

1.. Records (JEP 384)

The record is a new type of class in Java that makes it easy to create immutable data objects. Originally introduced in Java 14 as an early preview, Java 15 aims to refine a few aspects before becoming an official product feature. Let's look at an example using current Java and how it could change with records.

1.1 Without Records

Prior to records, we would create an immutable data transfer object (DTO) as:

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

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Notice that there's a lot of code here to create an immutable object that really just holds the state. All of our fields are explicitly defined using final, we have a single all-arguments constructor, and we have an accessor method for every field. In some cases, we might even declare the class itself as final to prevent any sub-classing. In many cases, we would also go a step further and override the toString method to provide meaningful logging output. We would probably also want to override the equals and hashCode methods to avoid unexpected consequences when comparing two instances of these objects.

1.2 With Records

Using the new record class, we can define the same immutable data object in a much more compact way:

```
public record Person(String name, int age) {
}
```

A few things have happened here. First and foremost, the class definition has a new syntax that is specific for records. This header is where we provide the details about the fields inside the record.

Using this header, the compiler can infer the internal fields. This means we don't need to define specific member variables and accessors, as they're provided by default. We also don't have to provide a constructor. Additionally, the compiler provides sensible implementations for the toString, equals, and hashCode methods. While records eliminate a lot of boilerplate code, they do allow us to override some of the default behaviors. For example, we could define a canonical constructor that does some validation:

```
public record Person(String name, int age) {
    public Person {
        if(age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
    }
}
```

It's worth mentioning that records do have some restrictions. Among other things, they are always final, they cannot be declared abstract, and they can't use native methods.

2.. Sealed Classes (JEP 360)

Currently, Java provides no fine-grained control over the inheritance. Access modifiers such as public, protected, private, as well as the default package-private, provide very coarse-grained control. To that end, the goal of sealed classes is to allow individual classes to declare which types may be used as sub-types. This also applies to interfaces and determining which types can implement them. Sealed classes involve two new keywords — sealed and permits.

```
public abstract sealed class Person
    permits Employee, Manager {

    //...
}
```

In this example, we've declared an abstract class named Person. We've also specified that the only classes that can extend it are Employee and Manager. Extending the sealed class is done just as it is today in Java, using the extends keyword:

```
public final class Employee extends Person {
}

public non-sealed class Manager extends Person {
}
```

It's important to note that any class that extends a sealed class must itself be declared sealed, non-sealed, or final. This ensures the class hierarchy remains finite and known by the compiler. This finite and exhaustive hierarchy is one of the great benefits of using sealed classes. Let's see an example of this in action:

```
if (person instanceof Employee) {
    return ((Employee) person).getEmployeeId();
}
else if (person instanceof Manager) {
    return ((Manager) person).getSupervisorId();
}
```

Without a sealed class, the compiler can't reasonably determine that all possible sub-classes are covered with our if-else statements. Without an else clause at the end, the compiler would likely issue a warning indicating our logic doesn't cover every case.

3.. Hidden Classes (JEP 371)

A new feature being introduced in Java 15 is known as hidden classes. While most developers won't find a direct benefit from them, 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. Note that current versions of Java do allow for the creation of anonymous classes similar to hidden classes. However, they rely on the Unsafe API. Hidden classes have no such dependency.

4.. Pattern Matching Type Checks (JEP 375)

The pattern matching feature was previewed in Java 14, and Java 15 aims to continue its preview status with no new enhancements. As a review, the goal of this feature is to remove a lot of boilerplate code that typically comes with the instanceof operator:

```
if (person instanceof Employee) {
    Employee employee = (Employee) person;
    Date hireDate = employee.getHireDate();
    //...
}
```

This is a very common pattern in Java. Whenever we check if a variable is a certain type, we almost always follow it with a cast to that type. The pattern matching feature simplifies this by introducing a new binding variable:

```
if (person instanceof Employee employee) {
    Date hireDate = employee.getHireDate();
    //...
}
```

Notice how we provide a new variable name, employee, as part of the type check. If the type check is true, then the JVM automatically casts the variable for us and assigns the result to the new binding variable. We can also combine the new binding variable with conditional statements:

```
if (person instanceof Employee employee && employee.getYearsOfService() > 5) {
    //...
}
```

In future Java versions, the goal is to expand pattern matching to other language features such as switch statements.

5.. Foreign Memory API (JEP 383)

Foreign memory access is already an incubating feature of Java 14. In Java 15, the goal is to continue its incubation status while adding several new features:

- A new VarHandle API, to customize memory access var handles
- Support for parallel processing of a memory segment using the Spliterator interface
- Enhanced support for mapped memory segments
- Ability to manipulate and dereference addresses coming from things like native calls

Foreign memory generally refers to memory that lives outside the managed JVM heap. Because of this, it's not subject to garbage collection and can typically handle incredibly large memory segments. While these new APIs likely won't impact most developers directly, they will provide a lot of value to third-party libraries that deal with foreign memory. This includes distributed caches, denormalized document stores, large arbitrary byte buffers, memory-mapped files, and more.

6.. 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. Both were previously available using experimental feature flags. This approach allows developers to test the new garbage collectors and submit feedback without downloading a separate JDK or add-on. One note on Shenandoah: it isn't available from all vendor JDKs — most notably, Oracle JDK doesn't include it.

7.. Other Changes

There are several other noteworthy changes in Java 15.

- After multiple rounds of previews in Java 13 and 14, text blocks will be a fully supported product feature in Java 15.
- Helpful null pointer exceptions, originally delivered in Java 14 under JEP 358, are now enabled by default.
- The legacy DatagramSocket API has been rewritten. This is a follow-on to a rewrite in Java 14 of the Socket API. While it won't impact most developers, it is interesting as it's a prerequisite for Project Loom.
- Also of note, Java 15 includes cryptographic support for Edwards-Curve Digital Signature Algorithm. EdDSA is a modern elliptic curve signature scheme that has several advantages over the existing signature schemes in the JDK.
- Finally, several things have been deprecated in Java 15. Biased locking, Solaris/SPARC ports, and RMI Activation are all removed or scheduled for removal in a future release.
- Of note, The Nashorn JavaScript engine, originally introduced in Java 8, is now removed. With the introduction of GraalVM and other VM technologies recently, it's clear Nashorn no longer has a place in the JDK ecosystem.