JEP 405: Record Classes to Extend Pattern Matching in Java

This item in japanese

Lire ce contenu en français

JEP 405, Record Patterns (Preview), has been promoted from **Proposed to Target** to **Targeted** for JDK 19. Under the umbrella of Project Amber, this JEP proposes to enhance the language with record patterns to deconstruct record values. Record patterns may be used in conjunction with type patterns to "enable a robust, declarative, and composable form of data navigation and processing." This is still a preview feature.

JEP 394, Pattern Matching for instanceof, delivered in JDK 16, extended the **`instanceof`** operator to take a type pattern and perform pattern matching. Consider the following example:

```
public void print(Object o) {
    if (o instanceof Double) {
        Double d = (Double) o;
        System.out.println("d = " + d);
    }
}
```

The above code could be written using the pattern matching as follows:

```
public void print(Object o) {
    if (o instanceof Double d) {
        System.out.println("d = " + d);
```

In the above code, **o** matches the type pattern **Double d** if, at run time, the value of **o** is an instance of **Double**. This reduces explicit typecast and makes the code shorter and more manageable.

JEP 395, Records, introduced Record classes, a transparent carrier of data that made it easy for developers to write immutable objects. Consider the following example:

```
record Point(int x, int y) { }
```

With this, developers are no longer required to explicitly write a constructor, accessor methods, and other methods such as **toString()**, and **hashCode()**. Thus, code becomes clean and less verbose.

If an instance of a record class is used within a code block, developers usually extract the data using its accessor methods. For instance:

```
public void printSum(Object o) {
    if (o instanceof Point p) {
        int x = p.x();
        int y = p.y();
        System.out.println(x + y);
    }
}
```

In the above code, the pattern variable **p** is used to invoke the accessor methods **x()** and **y()** to get the value of **x** and **y**. There is no other use of **p** here. In the case of the record pattern, the variable **p** is no longer required.

Now the above code can be rewritten:

```
public void printSum(Object o) {
   if (o instanceof Point(int x int y)) {
       System.out.println(x + y);
```

Similarly, this allows developers to deconstruct more complicated object graphs. Consider the following code example:

```
enum Color {RED, GREEN, BLUE}
record ColoredPoint(Point p, Color color) {}
record Point(int x, int y) {}
record Square(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
```

If developers need to print the upper left **`ColoredPoint`** in a pattern matching scenario using the Record pattern, it can be deconstructed as follows:

```
public void printUpperLeftColoredPoint(Square s) {
    if (s instanceof Square(ColoredPoint(Point(var x, var y), var co
    }
}
```

On the other hand, the alternative to the above deconstructed code is much more verbose.

Furthermore, the type patterns were extended for use in **`switch`** case labels via JEP 406, Pattern Matching for switch (Preview) (delivered in JDK 17), and JEP 420, Pattern Matching for switch (Second Preview) (delivered in JDK 18). With these, a similar deconstruction can be used in the **`switch`** statement. However, work on this JEP is still ongoing, and there are many directions in which it could evolve and expand. Enthusiastic developers can watch the mailing list and join this discussion.

About the Author