JEP 440: Record Patterns

Owner	Gavin Bierman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	21
Component	specification / language
Discussion	amber dash dev at openjdk dot org
Relates to	JEP 432: Record Patterns (Second Preview)
Reviewed by	Brian Goetz
Endorsed by	Brian Goetz
Created	2023/01/18 14:38
Updated	2023/08/28 16:51
Issue	8300541

Summary

Enhance the Java programming language with *record patterns* to deconstruct record values. Record patterns and type patterns can be nested to enable a powerful, declarative, and composable form of data navigation and processing.

History

Record patterns were proposed as a preview feature by JEP 405 and delivered in JDK 19, and previewed a second time by JEP 432 and delivered in JDK 20. This feature has co-evolved with *Pattern Matching for `switch`* (JEP 441), with which it has considerable interaction. This JEP proposes to finalize the feature with further refinements based upon continued experience and feedback.

Apart from some minor editorial changes, the main change since the second preview is to remove support for record patterns appearing in the header of an enhanced `for` statement. This feature may be re-proposed in a future JEP.

Goals

- Extend pattern matching to destructure instances of `record` classes, enabling more sophisticated data queries.
- Add nested patterns, enabling more composable data queries.

Motivation

In Java 16, JEP 394 extended the `instanceof` operator to take a *type pattern* and perform *pattern matching*. This modest extension allows the familiar instanceof-and-cast idiom to be simplified, making it both more concise and less error-prone:

```
// Prior to Java 16
if (obj instanceof String) {
    String s = (String)obj;
    ... use s ...
}

// As of Java 16
if (obj instanceof String s) {
    ... use s ...
}
```

In the new code, `obj` matches the type pattern `String s` if, at run time, the value of `obj` is an instance of `String`. If the pattern matches then the `instanceof` expression is `true` and the pattern variable `s` is initialized to the value of `obj` cast to `String`, which can then be used in the contained block.

Type patterns remove many occurrences of casting at a stroke. However, they are only the first step towards a more declarative, data-focused style of programming. As Java supports new and more expressive ways of modeling data, pattern matching can streamline the use of such data by enabling developers to express the semantic intent of their models.

Pattern matching and records

Records (JEP 395) are transparent carriers for data. Code that receives an instance of a record class will typically extract the data, known as the *components*, using the built-in component accessor methods. For example, we can use a type pattern to test whether a value is an instance of the record class `Point` and, if so, extract the x and y components from the value:

```
// As of Java 16
record Point(int x, int y) {}

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

The pattern variable `p` is used here solely to invoke the accessor methods `x()` and `y()`, which return the values of the components x and y. (In every record class there is a one-to-one correspondence between its accessor methods and its components.) It would be better if the pattern could not only test whether a value is an instance of `Point` but also extract the x and y components from the value directly, invoking the accessor methods on our behalf. In other words:

```
// As of Java 21
static void printSum(Object obj) {
    if (obj instanceof Point(int x, int y)) {
        System.out.println(x+y);
    }
}
```

`Point(int x, int y)` is a *record pattern*. It lifts the declaration of local variables for extracted components into the pattern itself, and initializes those variables by invoking the accessor methods when a value is matched against the pattern. In effect, a record pattern disaggregates an instance of a record into its components.

Nested record patterns

The true power of pattern matching is that it scales elegantly to match more complicated object graphs. For example, consider the following declarations:

```
// As of Java 16
record Point(int x, int y) {}
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) {}
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
```

We have already seen that we can extract the components of an object with a record pattern. If we want to extract the color from the upper-left point, we could write:

```
// As of Java 21
static void printUpperLeftColoredPoint(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint ul, ColoredPoint lr)) {
         System.out.println(ul.c());
    }
}
```

But the `ColoredPoint` value `ul` is itself a record value, which we might want to decompose further. Record patterns therefore support *nesting*, which allows the record component to be further matched against, and decomposed by, a nested pattern. We can nest another pattern inside the record pattern and decompose both the outer and inner records at once:

```
// As of Java 21
static void printColorOfUpperLeftPoint(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint(Point p, Color c),
                               ColoredPoint lr)) {
        System.out.println(c);
    }
```

```
}
```

Nested patterns allow us, further, to take apart an aggregate with code that is as clear and concise as the code that puts it together. If we were creating a rectangle, for example, we would likely nest the constructors in a single expression:

```
// As of Java 16
Rectangle r = new Rectangle(new ColoredPoint(new Point(x1, y1), c1),
                            new ColoredPoint(new Point(x2, y2), c2));
```

With nested patterns we can deconstruct such a rectangle with code that echoes the structure of the nested constructors:

```
// As of Java 21
static void printXCoordOfUpperLeftPointWithPatterns(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint(Point(var x, var y), var c),
                               var lr)) {
        System.out.println("Upper-left corner: " + x);
    }
}
```

Nested patterns can, of course, fail to match:

```
// As of Java 21
record Pair(Object x, Object y) {}

Pair p = new Pair(42, 42);

if (p instanceof Pair(String s, String t)) {
    System.out.println(s + ", " + t);
} else {
    System.out.println("Not a pair of strings");
}
```

Here the record pattern `Pair(String s, String t)` contains two nested type patterns, namely `String s` and `String t`. A value matches the pattern `Pair(String s, String t)` if it is a `Pair` and, recursively, its component values match the type patterns `String s` and `String t`. In our example code above these recursive pattern matches fail since neither of the record component values are strings, and thus the `else` block is executed.

In summary, nested patterns elide the accidental complexity of navigating objects so that we can focus on the data expressed by those objects. They also give us the power to centralize error handling, since a value fails to match a nested pattern *P(Q)* if either, or both, of the subpatterns fail to match. We need not check and handle each individual subpattern matching failure — either the entire pattern matches, or not.

Description

We extend the Java programming language with nestable record patterns.

The grammar for patterns becomes:

```
Pattern:
  TypePattern
  RecordPattern

TypePattern:
  LocalVariableDeclaration

RecordPattern:
  ReferenceType ( [ PatternList ] )

PatternList :
  Pattern { , Pattern }
```

Record patterns

A *record pattern* consists of a record class type and a (possibly empty) pattern list which is used to match against the corresponding record component values.

For example, given the declaration

```
record Point(int i, int j) {}
```

a value v matches the record pattern `Point(int i, int j)` if it is an instance of the record type `Point`; if so, the pattern variable `i` is initialized with the result of invoking the accessor method corresponding to `i` on the value v, and the pattern variable `j` is initialized to the result of invoking the accessor method corresponding to `j` on the value v. (The names of the pattern variables do not need to be the same as the names of the record components; i.e., the record pattern `Point(int x, int y)` acts identically except that the pattern variables x and y are initialized.)

The `null` value does not match any record pattern.

A record pattern can use `var` to match against a record component without stating the type of the component. In that case the compiler infers the type of the pattern variable introduced by the `var` pattern. For example, the pattern `Point(var a, var b)` is shorthand for the pattern `Point(int a, int b)`.

The set of pattern variables declared by a record pattern includes all of the pattern variables declared in the pattern list.

An expression is compatible with a record pattern if it could be cast to the record type in the pattern without requiring an unchecked conversion.

If a record pattern names a generic record class but gives no type arguments (i.e., the record pattern uses a raw type) then the type arguments are always inferred. For example:

```
// As of Java 21
record MyPair<S,T>(S fst, T snd){};

static void recordInference(MyPair<String, Integer> pair){
    switch (pair) {
        case MyPair(var f, var s) ->
            ... // Inferred record pattern MyPair<String,Integer>(var f, var s)
        ...
    }
}
```

Inference of type arguments for record patterns is supported in all constructs that support record patterns, namely `instanceof` expressions and `switch` statements and expressions.

Inference works with nested record patterns; for example:

```
// As of Java 21
record Box<T>(T t) {}

static void test1(Box<Box<String>> bbs) {
    if (bbs instanceof Box<Box<String>>(Box(var s))) {
        System.out.println("String " + s);
    }
}
```

Here the type argument for the nested pattern `Box(var s)` is inferred to be `String`, so the pattern itself is inferred to be `Box<String>(var s)`.

In fact it is possible to drop the type arguments in the outer record pattern as well, leading to the concise code:

```
// As of Java 21
static void test2(Box<Box<String>> bbs) {
    if (bbs instanceof Box(Box(var s))) {
        System.out.println("String " + s);
    }
}
```

Here the compiler will infer that the entire `instanceof` pattern is `Box<Box<String>>(Box<String>(var s))`,

For compatibility, type patterns do not support the implicit inference of type arguments; e.g., the type pattern `List l` is always treated as a raw type pattern.

Record patterns and exhaustive `switch`

JEP 441 enhances both `switch` expressions and `switch` statements to support

pattern labels. Both `switch` expressions and pattern `switch` statements must be *exhaustive:* The switch block must have clauses that deal with all possible values of the selector expression. For pattern labels this is determined by analysis of the types of the patterns; for example, the case label `case Bar b` matches values of type `Bar` and all possible subtypes of `Bar`.

With pattern labels involving record patterns, the analysis is more complex since we must consider the types of the component patterns and make allowances for `sealed` hierarchies. For example, consider the declarations:

```
class A {}
class B extends A {}
sealed interface I permits C, D {}
final class C implements I {}
final class D implements I {}
record Pair<T>(T x, T y) {}

Pair<A> p1;
Pair<I> p2;
```

The following `switch` is not exhaustive, since there is no match for a pair containing two values both of type `A`:

```
// As of Java 21
switch (p1) {                 // Error!
    case Pair<A>(A a, B b) -> ...
    case Pair<A>(B b, A a) -> ...
}
```

These two switches are exhaustive, since the interface `I` is `sealed` and so the types `C` and `D` cover all possible instances:

```
// As of Java 21
switch (p2) {
    case Pair<I>(I i, C c) -> ...
    case Pair<I>(I i, D d) -> ...
}

switch (p2) {
    case Pair<I>(C c, I i) -> ...
    case Pair<I>(D d, C c) -> ...
    case Pair<I>(D d1, D d2) -> ...
}
```

In contrast, this `switch` is not exhaustive since there is no match for a pair containing two values both of type `D`:

```
// As of Java 21
switch (p2) {                         // Error!
    case Pair<I>(C fst, D snd) -> ...
    case Pair<I>(D fst, C snd) -> ...
    case Pair<I>(I fst, C snd) -> ...
}
```

Future Work

There are many directions in which the record patterns described here could be extended:

- Varargs patterns, for records of variable arity;
- Unnamed patterns, which can appear in record-pattern pattern lists and which match any value but do not declare pattern variables; and
- Patterns that can apply to values of arbitrary classes rather than only record classes.

We may consider some of these in future JEPs.

Dependencies

This JEP builds on *Pattern Matching for `instanceof`* (JEP 394), delivered in JDK 16. It has co-evolved with *Pattern Matching for `switch`* (JEP 441).