JEP 432: Record Patterns (Second Preview)

Owner	Gavin Bierman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	20
Component	specification / language
Discussion	amber dash dev at openjdk dot org
Relates to	JEP 405: Record Patterns (Preview) JEP 433: Pattern Matching for switch (Fourth Preview) JEP 440: Record Patterns
Reviewed by	Alex Buckley, Brian Goetz
Endorsed by	Brian Goetz
Created	2022/09/20 22:08
Updated	2023/05/12 15:34
Issue	8294078

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. This is a preview language feature.

History

Record patterns were proposed as a preview feature by JEP 405 and delivered in JDK 19. This JEP proposes a second preview with further refinements based upon continued experience and feedback.

The main changes since the first preview are to:

- Add support for inference of type arguments of generic record patterns,
- Add support for record patterns to appear in the header of an enhanced `for` statement, and
- Remove support for named record patterns.

Goals

- Extend pattern matching to express more sophisticated, composable data queries.
- Do not change the syntax or semantics of type patterns.

Motivation

In JDK 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:

```
// Old code
if (obj instanceof String) {
    String s = (String)obj;
    ... use s ...
}

// New code
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.

In JDK 17, JDK 18, and JDK 19 we extended the use of type patterns to `switch` case labels as well, via JEP 406, JEP 420, and JEP 427.

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 record classes

Record classes (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*. 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:

```
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:

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

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.

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

```
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:

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

But our `ColoredPoint` is itself a record, 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:

```
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:

```
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:

```
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:

```
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 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 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.

The `instanceof` expression and `switch` are not the only places where the disaggregating behavior of a record pattern is convenient. Allowing a record pattern in an enhanced `for` statement would make it easy to loop over a collection of record values and swiftly extract the components of each record. For example:

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

static void dump(Point[] pointArray) {
    for (Point(var x, var y) : pointArray) {        // Record Pattern in header!
        System.out.println("(" + x + ", " + y + ")");
    }
}
```

The meaning is intuitive: On each iteration of the loop, each successive element of the array or `Iterable` is pattern matched against the record pattern in the header.

The record pattern in the enhanced `for` statement can have nested patterns, for example:

```
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) {}
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}

static void printUpperLeftColors(Rectangle[] r) {
    for (Rectangle(ColoredPoint(Point p, Color c), ColoredPoint lr): r) {
         System.out.println(c);
    }
}
```

Description

We extend the Java programming language with nestable record patterns.

The grammar for patterns will become:

```
Pattern:
  TypePattern
  ParenthesizedPattern
  RecordPattern

TypePattern:
  LocalVariableDeclaration

ParenthesizedPattern:
  ( Pattern )

RecordPattern:
  ReferenceType RecordStructurePattern

RecordStructurePattern:
  ( [ RecordComponentPatternList ] )

RecordComponentPatternList :
  Pattern { , Pattern }
```

Record patterns

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

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 record component 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 class is generic then it can be used in a record pattern as either a parameterized type or as a raw type. For example:

```
record Box<T>(T t) {}

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

Here the record class type in the record pattern is a parameterized type. It could be written equivalently as follows, in which case the type argument is inferred:

```
static void test2(Box<String> bo) {
    if (bo instanceof Box(var s)) {    // Inferred to be Box<String>(var s)
        System.out.println("String " + s);
    }
}
```

Inference applies to nested record patterns. For example:

```
static void test3(Box<Box<String>> bo) {
    if (bo 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. It would be even more concise to drop the type arguments in the outer record pattern as well:

```
static void test4(Box<Box<String>> bo) {
    if (bo instanceof Box(Box(var s))) {
        System.out.println("String " + s);
    }
}
```

Type patterns do not support 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 420 enhanced both `switch` expressions and `switch` statements to support labels that include patterns, including record patterns. 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`:

```
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:

```
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`:

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

Record patterns and enhanced `for` statements

If R is a record pattern then an enhanced `for` statement of form

```
for (R : e) S
```

is equivalent to the following enhanced `for` statement, which has no record pattern in the header:

```
for (var tmp : e) {
    switch(tmp) {
        case null -> throw new MatchException(new NullPointerException());
        case R -> S;
    }
}
```

This translation has the following consequences:

- The record pattern R must be *applicable* to the element type of the array or `Iterable`.
- The record pattern R must be *exhaustive* for the element type of the array or `Iterable`.
- Should any element of e be `null` then the execution of the enhanced `for` statement results in `MatchException` being thrown.

For example:

```
record Pair(Object fst, Object snd){}

static void notApplicable(String[] arg) {
    for (Pair(var fst, var snd): arg) {   // Compile-time error, pattern not applicable
        System.out.println("An element");
    }
}

static void notExhaustive(Pair[] arg) {
    for (Pair(String s, String t): arg) { // Compile-time error, pattern not exhaustive
        System.out.println(s+", "+t);
    }
}

static void exceptionTest() {
    Pair[] ps = new Pair[]{
        new Pair(1,2),
        null,
        new Pair("hello","world")
    };
    for (Pair(var f, var s): ps) {  // Run-time MatchException
        System.out.println(f);
    }
}
```

Future Work

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

- Array patterns, whose subpatterns match individual array elements;
- Varargs patterns, when the record is a varargs record;
- Do-not-care patterns, which can appear as an element in a record component pattern list but do not declare a pattern variable; and
- Patterns based upon arbitrary classes rather than only record classes.

We may consider some of these in future JEPs.

Dependencies

This JEP builds on JEP 394 (Pattern Matching for `instanceof`), delivered in JDK 16.