OpenJDK
Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds
Mailing lists
Wiki · IRC
Bylaws · Census
Legal
Workshop
JEP Process
Source code
Mercurial
GitHub
Tools
Git
jtreg harness
Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web
Projects
(overview, archive)
Amber
Babylon
CRaC
Caciocavallo
Closures
Code Tools
Coin
Common VM Interface
Compiler Grammar
Detroit
Developers' Guide
Device I/O
Duke
Galahad
Graal
IcedTea
JDK 7
JDK 8
JDK 8 Updates
JDK 9
JDK (..., 21, 22, 23)
JDK Updates
JavaDoc.Next
Jigsaw
Kona
Kulla
Lambda
Lanai
Leyden
Lilliput
Locale Enhancement
Loom
Memory Model Update
Metropolis
Mission Control
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch32
Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
Portola
SCTP
Shenandoah
Skara
Sumatra
Tiered Attribution
Tsan
Type Annotations
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC
ORACLE

JEP 405: Record Patterns (Preview)

Owner	Gavin Bierman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	19
Component	specification / language
Discussion	amber dash dev at openjdk dot java dot net
Relates to	JEP 427: Pattern Matching for switch (Third Preview) JEP 432: Record Patterns (Second Preview)
Reviewed by	Alex Buckley, Brian Goetz
Endorsed by	Brian Goetz
Created	2021/01/21 16:44
Updated	2023/05/12 15:34
Issue	8260244

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.

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 (o instanceof String) {
    String s = (String)o;
    ... use s ...
}

// New code
if (o instanceof String s) {
    ... use s ...
}
```

In the new code, o matches the type pattern `String s` if, at run time, the value of o 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 o cast to `String`, which can then be used in the contained block.

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

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 o) {
    if (o 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 o) {
    if (o 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);
    }
}
```

In summary, nested patterns elide the accidental complexity of navigating objects so that we can focus on the data expressed by those objects.

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 [ Identifier ]

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, and an optional identifier. A record pattern with an identifier is called a *named* record pattern, and the variable is referred to as the *record pattern variable*.

For example, given the declaration

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

a value v matches the record pattern `Point(int i, int j) p` 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, and the pattern variable `j` is initialized to the result of invoking the accessor method corresponding to `j` on the value. (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 record pattern variable p is initialized to the value of v cast to `Point`.

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 and, if the record pattern is a named record pattern, the record pattern variable.

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 any record pattern that names this record class must use a generic type. For example, given the declaration:

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

The following methods are correct:

```
static void test1(Box<Object> bo) {
    if (bo instanceof Box<Object>(String s)) {
        System.out.println("String " + s);
    }
}
static void test2(Box<Object> bo) {
    if (bo instanceof Box<String>(var s)) {
        System.out.println("String " + s);
    }
}
```

whereas both of the following result in compile-time errors:

```
static void erroneousTest1(Box<Object> bo) {
    if (bo instanceof Box(var s)) {                 // Error
        System.out.println("I'm a box");
    }
}
static void erroneousTest2(Box b) {
    if (b instanceof Box(var t)) {                  // Error
        System.out.println("I'm a box");
    }
}
```

In the future we may extend inference to infer the type arguments of generic record patterns.

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, as 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 as 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) -> ...
}
```

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;
- Inference for type arguments in generic record patterns, possibly using a diamond form (<>);
- 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.