JEP 394: Pattern Matching for instanceof

Owner	Gavin Bierman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	16
Component	specification / language
Discussion	amber dash dev at openjdk dot java dot net
Relates to	JEP 305: Pattern Matching for instanceof (Preview) JEP 375: Pattern Matching for instanceof (Second Preview)
Reviewed by	Alex Buckley, Brian Goetz, Maurizio Cimadamore
Endorsed by	Brian Goetz
Created	2020/07/27 13:05
Updated	2022/06/10 16:12
Issue	8250623

Summary

Enhance the Java programming language with *pattern matching* for the `instanceof` operator. Pattern matching allows common logic in a program, namely the conditional extraction of components from objects, to be expressed more concisely and safely.

History

Pattern matching for `instanceof` was proposed by JEP 305 and delivered in JDK 14 as a preview feature. It was re-proposed by JEP 375 and delivered in JDK 15 for a second round of preview.

This JEP proposes to finalize the feature in JDK 16, with the following refinements:

- Lift the restriction that pattern variables are implicitly final, to reduce asymmetries between local variables and pattern variables.
- Make it a compile-time error for a pattern `instanceof` expression to compare an expression of type *S* against a pattern of type *T*, where *S* is a subtype of *T*. (This `instanceof` expression will always succeed and is then pointless. The opposite case, where a pattern match will always fail, is already a compile-time error.)

Other refinements may be incorporated based on further feedback.

Motivation

Nearly every program includes some sort of logic that combines testing if an expression has a certain type or structure, and then conditionally extracting components of its state for further processing. For example, all Java programmers are familiar with the `instanceof`-and-cast idiom:

```
if (obj instanceof String) {
    String s = (String) obj;    // grr...
    ...
}
```

There are three things going on here: a test (is `obj` a `String`?), a conversion (casting `obj` to `String`), and the declaration of a new local variable (`s`) so that we can use the string value. This pattern is straightforward and understood by all Java programmers, but is suboptimal for several reasons. It is tedious; doing both the type test and cast should be unnecessary (what else would you do after an `instanceof` test?). This boilerplate — in particular, the three occurrences of the type `String` — obfuscates the more significant logic that follows. But most importantly, the repetition provides opportunities for errors to creep unnoticed into programs.

Rather than reach for ad-hoc solutions, we believe it is time for Java to embrace *pattern matching*. Pattern matching allows the desired "shape" of an object to be expressed concisely (the *pattern*), and for various statements and expressions to test that "shape" against their input (the *matching*). Many languages, from Haskell to C#, have embraced pattern matching for its brevity and safety.

Description

A *pattern* is a combination of (1) a *predicate*, or test, that can be applied to a target, and (2) a set of local variables, known as *pattern variables*, that are extracted from the target only if the predicate successfully applies to it.

A *type pattern* consists of a predicate that specifies a type, along with a single pattern variable.

The `instanceof` operator (JLS 15.20.2) is extended to take a type pattern instead of just a type.

This allows us to refactor the tedious code above to the following:

```
if (obj instanceof String s) {
    // Let pattern matching do the work!
    ...
}
```

(In this code, the phrase `String s` is the type pattern.) The meaning is intuitive. The `instanceof` operator matches the target `obj` to the type pattern as follows: If `obj` is an instance of `String`, then it is cast to `String` and the value is assigned to the variable `s`.

The conditionality of pattern matching — if a value does not match a pattern, then the pattern variable is not assigned a value — means that we have to consider carefully the scope of the pattern variable. We could do something simple and say that the scope of the pattern variable is the containing statement and all subsequent statements in the enclosing block. But this has unfortunate *poisoning* consequences, for example:

```
if (a instanceof Point p) {
   ...
}
if (b instanceof Point p) {         // ERROR - p is in scope
   ...
}
```

In other words, by the second statement the pattern variable p would be in a poisoned state — it is in scope, but it should not be accessible since it may not be assigned a value. But even though it shouldn't be accessed, since it is in scope, we can't just declare it again. This means that a pattern variable can become poisoned after it is declared, so programmers would have to think of lots of distinct names for their pattern variables.

Rather than using a coarse approximation for the scope of pattern variables, pattern variables instead use the concept of *flow scoping*. A pattern variable is only in scope where the compiler can deduce that the pattern has definitely matched and the variable will have been assigned a value. This analysis is flow sensitive and works in a similar way to existing flow analyses such as definite assignment. Returning to our example:

```
if (a instanceof Point p) {
    // p is in scope
    ...
}
// p not in scope here
if (b instanceof Point p) {     // Sure!
        ...
}
```

The motto is: "A pattern variable is in scope where it has definitely matched". This allows for the safe reuse of pattern variables and is both intuitive and familiar, since Java developers are already used to flow sensitive analyses.

When the conditional expression of the `if` statement grows more complicated than a single `instanceof`, the scope of the pattern variable grows accordingly. For example, in this code:

```
if (obj instanceof String s && s.length() > 5) {
    flag = s.contains("jdk");
}
```

the pattern variable `s` is in scope on the right hand side of the && operator, as well as in the true block. (The right hand side of the && operator is only evaluated if the pattern match succeeded and assigned a value to `s`.) On the other hand, the following code does not compile:

```
if (obj instanceof String s || s.length() > 5) {    // Error!
    ...
}
```

Because of the semantics of the || operator, the pattern variable `s` might not have been assigned and so the flow analysis dictates that the variable `s` is not in scope on the right hand side of the || operator.

The use of pattern matching in `instanceof` should significantly reduce the overall number of explicit casts in Java programs. Type test patterns are particularly useful when writing equality methods. Consider the following equality method taken from Item 10 of *Effective Java*:

```
public final boolean equals(Object o) {
    return (o instanceof CaseInsensitiveString) &&
        ((CaseInsensitiveString) o).s.equalsIgnoreCase(s);
}
```

Using a type pattern means it can be rewritten to the clearer:

```
public final boolean equals(Object o) {
    return (o instanceof CaseInsensitiveString cis) &&
        cis.s.equalsIgnoreCase(s);
}
```

Other `equals` methods are even more dramatically improved. Consider the class `Point` from above, where we might write an `equals` method as follows:

```
public final boolean equals(Object o) {
    if (!(o instanceof Point))
        return false;
    Point other = (Point) o;
    return x == other.x
```

```
            && y == other.y;
    }
```

Using pattern matching instead, we can combine these multiple statements into a single expression, eliminating the repetition and simplifying the control flow:

```
    public final boolean equals(Object o) {
        return (o instanceof Point other)
            && x == other.x
            && y == other.y;
    }
```

The flow scoping analysis for pattern variables is sensitive to the notion of whether a statement can complete normally. For example, consider the following method:

```
    public void onlyForStrings(Object o) throws MyException {
        if (!(o instanceof String s))
            throw new MyException();
        // s is in scope
        System.out.println(s);
        ...
    }
```

This method tests whether its parameter `o` is a `String`, and throws an exception if not. It is only possible to reach the `println` statement if the conditional statement has completed normally. Because the contained statement of the conditional statement can never complete normally, this can only occur if the conditional expression has evaluated to the value `false`, which, in turn, means that the pattern matching has succeeded. Accordingly, the scope of the pattern variable `s` safely includes the statements following the conditional statement in the method block.

Pattern variables are just a special case of local variables, and aside from the definition of their scope, in all other respects pattern variables are treated as local variables. In particular, this means that (1) they can be assigned to, and (2) they can shadow a field declaration. For example:

```
    class Example1 {
        String s;

        void test1(Object o) {
            if (o instanceof String s) {
                System.out.println(s);      // Field s is shadowed
                s = s + "\n";               // Assignment to pattern variable
                ...
            }
            System.out.println(s);          // Refers to field s
            ...
        }
    }
```

However, the flow scoping nature of pattern variables means that some care must be taken to determine whether a name refers to a pattern variable declaration shadowing a field declaration or to the field declaration itself.

```
    class Example2 {
        Point p;

        void test2(Object o) {
            if (o instanceof Point p) {
                // p refers to the pattern variable
                ...
            } else {
                // p refers to the field
                ...
            }
        }
    }
```

The `instanceof` grammar is extended accordingly:

RelationalExpression:
    ...
    RelationalExpression `instanceof` *ReferenceType*
    RelationalExpression `instanceof` *Pattern*

Pattern:
    ReferenceType Identifier

Future Work

Future JEPs will enhance the Java programming language with richer forms of patterns, such as deconstruction patterns for record classes, and pattern matching for other language constructs, such as `switch` expressions and statements.

Alternatives

The benefits of type patterns could be obtained by *flow typing* in `if` statements, or by a *type switch* construct. Pattern matching generalizes both of these constructs.