JEP 420: Pattern Matching for switch (Second Preview)

Owner	Gavin Bierman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	18
Component	specification / language
Discussion	amber dash dev at openjdk dot java dot net
Relates to	JEP 406: Pattern Matching for switch (Preview) JEP 427: Pattern Matching for switch (Third Preview)
Reviewed by	Brian Goetz, Maurizio Cimadamore
Created	2021/09/03 13:00
Updated	2023/05/12 15:34
Issue	8273326

Summary

Enhance the Java programming language with pattern matching for `switch` expressions and statements, along with extensions to the language of patterns. Extending pattern matching to `switch` allows an expression to be tested against a number of patterns, each with a specific action, so that complex data-oriented queries can be expressed concisely and safely. This is a preview language feature in JDK 18.

History

Pattern Matching for `switch` was proposed by JEP 406 and delivered in JDK 17 as a preview feature. This JEP proposes a second preview of the feature in JDK 18 with minor refinements based upon experience and feedback.

The enhancements since the first preview are:

- Dominance checking now forces a constant case label to appear before a guarded pattern of the same type, for readability (see 1b, below); and
- Exhaustiveness checking of switch blocks is now more precise with sealed hierarchies where the permitted direct subclass only extends an instantiation of the (generic) `sealed` superclass (see 2, below).

Goals

- Expand the expressiveness and applicability of `switch` expressions and statements by allowing patterns to appear in `case` labels.
- Allow the historical null-hostility of `switch` to be relaxed when desired.
- Introduce two new kinds of patterns: *guarded patterns*, to allow pattern matching logic to be refined with arbitrary boolean expressions, and *parenthesized patterns*, to resolve some parsing ambiguities.
- Ensure that all existing `switch` expressions and statements continue to compile with no changes and execute with identical semantics.
- Do not introduce a new `switch`-like expression or statement with pattern-matching semantics that is separate from the traditional `switch` construct.
- Do not make the `switch` expression or statement behave differently when case labels are patterns versus when case labels are traditional constants.

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:

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

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

We often want to compare a variable such as `o` against multiple alternatives. Java supports multi-way comparisons with `switch` statements and, since Java 14, `switch` expressions (JEP 361), but unfortunately `switch` is very limited. You can only switch on values of a few types — numeric types, enum types, and `String` — and you can only test for exact equality against constants. We might like to use patterns to test the same variable against a number of possibilities, taking a specific action on each, but since the existing `switch` does not support that, we end up with a chain of `if...else` tests such as:

```
static String formatter(Object o) {
    String formatted = "unknown";
    if (o instanceof Integer i) {
        formatted = String.format("int %d", i);
    } else if (o instanceof Long l) {
```

```
            formatted = String.format("long %d", l);
        } else if (o instanceof Double d) {
            formatted = String.format("double %f", d);
        } else if (o instanceof String s) {
            formatted = String.format("String %s", s);
        }
        return formatted;
    }
```

This code benefits from using pattern `instanceof` expressions, but it is far from perfect. First and foremost, this approach allows coding errors to remain hidden because we have used an overly general control construct. The intent is to assign something to `formatted` in each arm of the `if...else` chain, but there is nothing that enables the compiler to identify and verify this invariant. If some block — perhaps one that is executed rarely — does not assign to `formatted`, we have a bug. (Declaring `formatted` as a blank local would at least enlist the compiler's definite-assignment analysis in this effort, but such declarations are not always written.) In addition, the above code is not optimizable; absent compiler heroics it will have $O(n)$ time complexity, even though the underlying problem is often $O(1)$.

But `switch` is a perfect match for pattern matching! If we extend `switch` statements and expressions to work on any type, and allow `case` labels with patterns rather than just constants, then we could rewrite the above code more clearly and reliably:

```
    static String formatterPatternSwitch(Object o) {
        return switch (o) {
            case Integer i -> String.format("int %d", i);
            case Long l    -> String.format("long %d", l);
            case Double d  -> String.format("double %f", d);
            case String s  -> String.format("String %s", s);
            default        -> o.toString();
        };
    }
```

The semantics of this `switch` are clear: A `case` label with a pattern matches the value of the selector expression `o` if the value matches the pattern. (We have shown a `switch` expression for brevity but could instead have shown a `switch` statement; the switch block, including the `case` labels, would be unchanged.)

The intent of this code is clearer because we are using the right control construct: We are saying, "the parameter `o` matches at most one of the following conditions, figure it out and evaluate the corresponding arm." As a bonus, it is optimizable; in this case we are more likely to be able to perform the dispatch in $O(1)$ time.

Pattern matching and `null`

Traditionally, `switch` statements and expressions throw `NullPointerException` if the selector expression evaluates to `null`, so testing for `null` must be done outside of the `switch`:

```
    static void testFooBar(String s) {
        if (s == null) {
            System.out.println("oops!");
            return;
        }
        switch (s) {
            case "Foo", "Bar" -> System.out.println("Great");
            default           -> System.out.println("Ok");
        }
    }
```

This was reasonable when `switch` supported only a few reference types. However, if `switch` allows a selector expression of any type, and `case` labels can have type patterns, then the standalone `null` test feels like an arbitrary distinction, and invites needless boilerplate and opportunity for error. It would be better to integrate the `null` test into the `switch`:

```
    static void testFooBar(String s) {
        switch (s) {
            case null         -> System.out.println("Oops");
            case "Foo", "Bar" -> System.out.println("Great");
            default           -> System.out.println("Ok");
        }
    }
```

The behavior of the `switch` when the value of the selector expression is `null` is always determined by its `case` labels. With a `case null` (or a total type pattern; see 4a, below) the `switch` executes the code associated with that label; without a `case null`, the `switch` throws `NullPointerException`, just as before. (To maintain backward compatibility with the current semantics of `switch`, the `default` label does not match a `null` selector.)

We may wish to handle `null` in the same way as another `case` label. For example, in the following code, `case null, String s` would match both the `null` value and all `String` values:

```
    static void testStringOrNull(Object o) {
        switch (o) {
            case null, String s -> System.out.println("String: " + s);
```

```
        }
    }
```

Refining patterns in `switch`

Experimentation with patterns in `switch` suggests it is common to want to refine patterns. Consider the following code that switches over a Shape value:

```
class Shape {}
class Rectangle extends Shape {}
class Triangle  extends Shape { int calculateArea() { ... } }

static void testTriangle(Shape s) {
    switch (s) {
        case null:
            break;
        case Triangle t:
            if (t.calculateArea() > 100) {
                System.out.println("Large triangle");
                break;
            }
        default:
            System.out.println("A shape, possibly a small triangle");
    }
}
```

The intent of this code is to have a special case for large triangles (with area over 100), and a default case for everything else (including small triangles). However, we cannot express this directly with a single pattern. We first have to write a case label that matches all triangles, and then place the test of the area of the triangle rather uncomfortably within the corresponding statement group. Then we have to use fall-through to get the correct behavior when the triangle has an area less than 100. (Note the careful placement of `break;` inside the `if` block.)

The problem here is that using a single pattern to discriminate among cases does not scale beyond a single condition. We need some way to express a *refinement* to a pattern. One approach might be to allow `case` labels to be refined; such a refinement is called a *guard* in other programming languages. For example, we could introduce a new keyword `where` to appear at the end of a `case` label and be followed by a boolean expression, e.g., `case Triangle t where t.calculateArea() > 100`.

However, there is another approach: Rather than extend the functionality of `case` labels, we can extend the language of patterns themselves. We can add a new kind of pattern called a *guarded pattern*, written `p && b`, that allows a pattern `p` to be refined by an arbitrary boolean expression `b`.

With this approach, we can revisit the `testTriangle` code to express the special case for large triangles directly. This eliminates the use of fall-through in the `switch` statement, which in turn means we can enjoy concise arrow-style (`->`) rules:

```
static void testTriangle(Shape s) {
    switch (s) {
        case Triangle t && (t.calculateArea() > 100) ->
            System.out.println("Large triangle");
        default ->
            System.out.println("A shape, possibly a small triangle");
    }
}
```

The value of `s` matches the pattern `Triangle t && (t.calculateArea() > 100)` if, first, it matches the type pattern `Triangle t` and, if so, the expression `t.calculateArea() > 100` evaluates to `true`.

Using `switch` makes it easy to understand and change case labels when application requirements change. For example, we might want to split triangles out of the default path; we can do that by using both a refined pattern and a non-refined pattern:

```
static void testTriangle(Shape s) {
    switch (s) {
        case Triangle t && (t.calculateArea() > 100) ->
            System.out.println("Large triangle");
        case Triangle t ->
            System.out.println("Small triangle");
        default ->
            System.out.println("Non-triangle");
    }
}
```

Description

We enhance `switch` statements and expressions in two ways:

- Extend `case` labels to include patterns in addition to constants, and
- Introduce two new kinds of patterns: *guarded patterns* and *parenthesized patterns*.

Patterns in switch labels

The heart of the proposal is to introduce a new `case p` switch label, where `p` is a pattern. The essence of a `switch` is unchanged: The value of the selector expression is compared to the switch labels, one of the labels is selected, and the code associated with that label is executed. The difference is now that for `case` labels with patterns, that selection is determined by pattern matching rather than by an equality test. For example, in the following code, the value of `o` matches the pattern `Long l`, and the code associated with `case Long l` will be executed:

```
Object o = 123L;
String formatted = switch (o) {
    case Integer i -> String.format("int %d", i);
    case Long l    -> String.format("long %d", l);
    case Double d  -> String.format("double %f", d);
    case String s  -> String.format("String %s", s);
    default        -> o.toString();
};
```

There are four major design issues when `case` labels can have patterns:

1. Enhanced type checking
2. Exhaustiveness of `switch` expressions and statements
3. Scope of pattern variable declarations
4. Dealing with `null`

1. Enhanced type checking

1a. Selector expression typing

Supporting patterns in `switch` means that we can relax the current restrictions on the type of the selector expression. Currently the type of the selector expression of a normal `switch` must be either an integral primitive type (`char`, `byte`, `short`, or `int`), the corresponding boxed form (`Character`, `Byte`, `Short`, or `Integer`), `String`, or an enum type. We extend this and require that the type of the selector expression be either an integral primitive type or any reference type.

For example, in the following pattern `switch` the selector expression `o` is matched with type patterns involving a class type, an enum type, a record type, and an array type (along with a `null` case label and a `default`):

```
record Point(int i, int j) {}
enum Color { RED, GREEN, BLUE; }

static void typeTester(Object o) {
    switch (o) {
        case null     -> System.out.println("null");
        case String s -> System.out.println("String");
        case Color c  -> System.out.println("Color with " + c.values().length + " values");
        case Point p  -> System.out.println("Record class: " + p.toString());
        case int[] ia -> System.out.println("Array of ints of length" + ia.length);
        default       -> System.out.println("Something else");
    }
}
```

Every case label in the switch block must be compatible with the selector expression. For a case label with a pattern, known as a *pattern label*, we use the existing notion of *compatibility of an expression with a pattern* (JLS §14.30.1).

1b. Dominance of pattern labels

It is possible for the selector expression to match multiple labels in a switch block. Consider this problematic example:

```
static void error(Object o) {
    switch(o) {
        case CharSequence cs ->
            System.out.println("A sequence of length " + cs.length());
        case String s ->    // Error - pattern is dominated by previous pattern
            System.out.println("A string: " + s);
        default -> {
            break;
        }
    }
}
```

The first pattern label `case CharSequence cs` *dominates* the second pattern label `case String s` because every value that matches the pattern `String s` also matches the pattern `CharSequence cs`, but not vice versa. This is because the type of the second pattern, `String`, is a subtype of the type of the first pattern, `CharSequence`.

A pattern label of the form `case p` where `p` is a total pattern for the type of the selector expression dominates a label `case null`. This is because a total pattern matches all values, including `null`.

A pattern label of the form `case p` dominates a pattern label of the form `case p && e`, i.e., where the pattern is a guarded version of the original pattern. For example, the pattern label `case String s` dominates the pattern label `case String s && s.length() > 0`, since every value that matches the guarded pattern `String s && s.length() > 0` also matches the pattern `String s`.

A pattern label can dominate a constant label. For example, the pattern label `case Integer i` dominates the constant label `case 42`, and the pattern label `case E e` dominates the constant label `case A` when `A` is an enum constant of enum class

type E. A guarded pattern label dominates a constant label if its contained label does; we do not check the guarding expression, since this is undecidable in general. Thus a guarded pattern label `case String s && s.length()>1` dominates the constant label `case "hello"`, as expected; but also `case Integer i && i <> 0` dominates the label `case 0`. This leads to a simple and readable ordering of case labels, where the constant labels should appear before the guarded pattern labels, and those should appear before the non-guarded type pattern labels:

```
switch(o) {
    case -1, 1 -> ...               // Special cases
    case Integer i && i > 0 -> ...  // Positive integer cases
    case Integer i -> ...           // All the remaining integers
    default ->
}
```

The compiler checks all labels. It is a compile-time error if a label in a switch block is dominated by an earlier label in that switch block. This dominance requirement ensures that if a switch block contains only type pattern case labels, they will appear in subtype order.

(The notion of dominance is analogous to conditions on the `catch` clauses of a `try` statement, where it is an error if a `catch` clause that catches an exception class E is preceded by a `catch` clause that can catch E or a superclass of E (JLS §11.2.3). Logically, the preceding `catch` clause dominates the subsequent `catch` clause.)

It is also a compile-time error if a switch block has more than one match-all switch label. The two *match-all* labels are `default` and total type patterns (see 4a, below).

2. Exhaustiveness of `switch` expressions and statements

A `switch` expression requires that all possible values of the selector expression are handled in the switch block; in other words, it is *exhaustive*. This maintains the property that successful evaluation of a `switch` expression will always yield a value. For normal `switch` expressions, this is enforced by a fairly straightforward set of extra conditions on the switch block. For pattern `switch` expressions, we achieve this by defining a notion of *type coverage* of switch labels in a switch block. The type coverage of all the switch labels in the switch block is then combined to determine if the switch block exhausts all the possibilities of the selector expression.

Consider this (erroneous) pattern `switch` expression:

```
static int coverage(Object o) {
    return switch (o) {         // Error - not exhaustive
        case String s -> s.length();
    };
}
```

The switch block has only one `case` label, `case String s`. This matches any value of the selector expression whose type is a subtype of `String`. We therefore say that the type coverage of this switch label is every subtype of `String`. This pattern `switch` expression is not exhaustive because the type coverage of its switch block (all subtypes of `String`) does not include the type of the selector expression (`Object`).

Consider this (still erroneous) example:

```
static int coverage(Object o) {
    return switch (o) {         // Error - not exhaustive
        case String s  -> s.length();
        case Integer i -> i;
    };
}
```

The type coverage of this switch block is the union of the coverage of its two switch labels. In other words, the type coverage is the set of all subtypes of `String` and the set of all subtypes of `Integer`. But, again, the type coverage still does not include the type of the selector expression, so this pattern `switch` expression is also not exhaustive and causes a compile-time error.

The type coverage of a `default` label is all types, so this example is (at last!) legal:

```
static int coverage(Object o) {
    return switch (o) {
        case String s  -> s.length();
        case Integer i -> i;
        default -> 0;
    };
}
```

If the type of the selector expression is a sealed class (JEP 409), then the type coverage check can take into account the `permits` clause of the sealed class to determine whether a switch block is exhaustive. This can sometimes remove the need for a `default` clause. Consider the following example of a `sealed` interface `S` with three permitted subclasses `A`, `B`, and `C`:

```
sealed interface S permits A, B, C {}
final class A implements S {}
final class B implements S {}
record C(int i) implements S {}  // Implicitly final

static int testSealedExhaustive(S s) {
```

```
        return switch (s) {
            case A a -> 1;
            case B b -> 2;
            case C c -> 3;
        };
    }
```

The compiler can determine that the type coverage of the switch block is the types `A`, `B`, and `C`. Since the type of the selector expression, `S`, is a sealed interface whose permitted subclasses are exactly `A`, `B`, and `C`, this switch block is exhaustive. As a result, no `default` label is needed.

Some extra care is needed when a permitted direct subclass only implements an instantiation of a (generic) `sealed` superclass. For example:

```
sealed interface I<T> permits A, B {}
final class A<X> implements I<String> {}
final class B<Y> implements I<Y> {}

static int testGenericSealedExhaustive(I<Integer> i) {
    return switch (i) {
        // Exhaustive as no A case possible!
        case B<Integer> bi -> 42;
    }
}
```

The only permitted subclasses of `I` are `A` and `B`, but the compiler can detect that the switch block need only cover the class `B` to be exhaustive since the selector expression is of type `I<Integer>`.

To defend against incompatible separate compilation, the compiler automatically adds a `default` label whose code throws an `IncompatibleClassChangeError`. This label will only be reached if the `sealed` interface is changed and the `switch` code is not recompiled. In effect, the compiler hardens your code for you.

(The requirement for a pattern `switch` expression to be exhaustive is analogous to the treatment of a `switch` expression whose selector expression is an enum class, where a `default` clause is not required if there is a clause for every constant of the enum class.)

The usefulness of having the compiler verify that `switch` expressions are exhaustive is extremely useful. Rather than keep this check solely for `switch` *expressions*, we extend this to `switch` statements also. To ensure backward compatibility, all existing `switch` statements will compile unchanged. But if a `switch` statement uses any of the new features detailed in this JEP, then the compiler will check that it is exhaustive.

More precisely, exhaustiveness is required of any `switch` statement that uses pattern or `null` labels or whose selector expression is not one of the legacy types (`char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, `Integer`, `String`, or an enum type).

This means that now both `switch` expressions *and* `switch` statements get the benefits of stricter type checking. For example:

```
sealed interface S permits A, B, C {}
final class A implements S {}
final class B implements S {}
record C(int i) implements S {}  // Implicitly final

static void switchStatementExhaustive(S s) {
    switch (s) {    // Error - not exhaustive; missing clause for permitted class B!
        case A a :
            System.out.println("A");
            break;
        case C c :
            System.out.println("C");
            break;
    };
}
```

Making most `switch` statements exhaustive is simply a matter of adding a simple `default` clause at the end of the switch block. This leads to clearer and easier to verify code. For example, the following `switch` statement is not exhaustive and is erroneous:

```
Object o = ...
switch (o) {    // Error - not exhaustive!
    case String s:
        System.out.println(s);
        break;
    case Integer i:
        System.out.println("Integer");
        break;
}
```

It can be made exhaustive trivially:

```
Object o = ...
switch (o) {
    case String s:
```

```
            System.out.println(s);
            break;
        case Integer i:
            System.out.println("Integer");
            break;
        default:    // Now exhaustive!
            break;
    }
```

(Future compilers of the Java language may emit warnings for legacy `switch` statements that are not exhaustive.)

3. Scope of pattern variable declarations

Pattern variables (JEP 394) are local variables that are declared by patterns. Pattern variable declarations are unusual in that their scope is *flow-sensitive*. As a recap consider the following example, where the type pattern `String s` declares the pattern variable `s`:

```
    static void test(Object o) {
        if ((o instanceof String s) && s.length() > 3) {
            System.out.println(s);
        } else {
            System.out.println("Not a string");
        }
    }
```

The declaration of `s` is in scope in the right-hand operand of the && expression, as well as in the "then" block. However, it is not in scope in the "else" block; in order for control to transfer to the "else" block the pattern match must fail, in which case the pattern variable will not have been initialized.

We extend this flow-sensitive notion of scope for pattern variable declarations to encompass pattern declarations occurring in `case` labels with two new rules:

1. The scope of a pattern variable declaration which occurs in a `case` label of a `switch` rule includes the expression, block, or `throw` statement that appears to the right of the arrow.
2. The scope of a pattern variable declaration which occurs in a `case` label of a `switch` labeled statement group includes the block statements of the statement group. It should not be possible to fall through a `case` label that declares a pattern variable.

This example shows the first rule in action:

```
    static void test(Object o) {
        switch (o) {
            case Character c -> {
                if (c.charValue() == 7) {
                    System.out.println("Ding!");
                }
                System.out.println("Character");
            }
            case Integer i ->
                throw new IllegalStateException("Invalid Integer argument of value " + i.intValue());
            default -> {
                break;
            }
        }
    }
```

The scope of the declaration of the pattern variable `c` is the block to the right of the first arrow.

The scope of the declaration of the pattern variable `i` is the `throw` statement to the right of the second arrow.

The second rule is more complicated. Let us first consider an example where there is only one `case` label for a `switch` labeled statement group:

```
    static void test(Object o) {
        switch (o) {
            case Character c:
                if (c.charValue() == 7) {
                    System.out.print("Ding ");
                }
                if (c.charValue() == 9) {
                    System.out.print("Tab ");
                }
                System.out.println("character");
            default:
                System.out.println();
        }
    }
```

The scope of the declaration of the pattern variable `c` includes all the statements of the statement group, namely the two `if` statements and the `println` statement. The scope does not include the statements of the `default` statement group, even though the execution of the first statement group can fall through the `default` switch label and execute these statements.

The possibility of falling through a case label that declares a pattern variable must be excluded as a compile-time error. Consider this erroneous example:

```
static void test(Object o) {
    switch (o) {
        case Character c:
            if (c.charValue() == 7) {
                System.out.print("Ding ");
            }
            if (c.charValue() == 9) {
                System.out.print("Tab ");
            }
            System.out.println("character");
        case Integer i:                 // Compile-time error
            System.out.println("An integer " + i);
        default:
            break;
    }
}
```

If this were allowed and the value of the selector expression o was a `Character`, then execution of the switch block could fall through the second statement group (after `case Integer i:`) where the pattern variable `i` would not have been initialized. Allowing execution to fall through a case label that declares a pattern variable is therefore a compile-time error.

This is why `case Character c: case Integer i: ...` is not permitted. Similar reasoning applies to the prohibition of multiple patterns in a case label: Neither `case Character c, Integer i: ...` nor `case Character c, Integer i -> ...` is allowed. If such case labels were allowed then both `c` and `i` would be in scope after the colon or arrow, yet only one of them would have been initialized depending on whether the value of o was a `Character` or an `Integer`.

On the other hand, falling through a label that does not declare a pattern variable is safe, as this example shows:

```
void test(Object o) {
    switch (o) {
        case String s:
            System.out.println("A string");
        default:
            System.out.println("Done");
    }
}
```

4. Dealing with `null`

4a. Matching `null`

Traditionally, a `switch` throws `NullPointerException` if the selector expression evaluates to `null`. This is well-understood behavior and we do not propose to change it for any existing `switch` code.

However, given that there is a reasonable and non-exception-bearing semantics for pattern matching and `null` values, there is an opportunity to make pattern `switch` more `null`-friendly while remaining compatible with existing `switch` semantics.

First, we introduce a new `null` label for a case, which matches when the value of the selector expression is `null`.

Second, we observe that if a pattern that is *total* for the type of the selector expression appears a pattern case label, then that label will also match when the value of the selector expression is `null`. (A type pattern p of type *U* is *total* for a type *T*, if *T* is a subtype of *U*. For example, the type pattern `Object o` is total for the type `String`.)

We lift the blanket rule that a `switch` immediately throws `NullPointerException` if the value of the selector expression is `null`. Instead, we inspect the case labels to determine the behavior of a `switch`:

- If the selector expression evaluates to `null` then any `null` case label or a total pattern case label is said to match. If there is no such label associated with the switch block then the `switch` throws `NullPointerException`, as before.
- If the selector expression evaluates to a non-`null` value then we select a matching case label, as normal. If no case label matches then any match-all label is considered to match.

For example, given the declaration below, evaluating `test(null)` will print `null!` rather than throw `NullPointerException`:

```
static void test(Object o) {
    switch (o) {
        case null     -> System.out.println("null!");
        case String s -> System.out.println("String");
        default       -> System.out.println("Something else");
    }
}
```

This new behavior around `null` is as if the compiler automatically enriches the switch block with a `case null` whose body throws `NullPointerException`. In other words, this code:

```
static void test(Object o) {
    switch (o) {
        case String s  -> System.out.println("String: " + s);
        case Integer i -> System.out.println("Integer");
        default        -> System.out.println("default");
    }
}
```

is equivalent to:

```
static void test(Object o) {
    switch (o) {
        case null      -> throw new NullPointerException();
        case String s  -> System.out.println("String: "+s);
        case Integer i -> System.out.println("Integer");
        default        -> System.out.println("default");
    }
}
```

In both examples, evaluating `test(null)` will cause `NullPointerException` to be thrown.

We preserve the intuition from the existing `switch` construct that performing a switch over `null` is an exceptional thing to do. The difference in a pattern `switch` is that you have a mechanism to directly handle this case inside the `switch` rather than outside. If you choose not to have a null-matching case label in a switch block then switching over a `null` value will throw `NullPointerException`, as before.

4b. New label forms arising from `null` labels

Switch blocks in Java 16 support two styles: one based on labeled groups of statements (the `:` form) where fall-through is possible, and one based on single-consequent form (the `->` form) where fall-through is not possible. In the former style, multiple labels are typically written `case l1: case l2:` whereas in the latter style, multiple labels are written `case l1, l2 ->`.

Supporting `null` labels means that a number of special cases can be expressed in the `:` form. For example:

```
Object o = ...
switch(o) {
    case null: case String s:
        System.out.println("String, including null");
        break;
    ...
}
```

Developers reasonably expect the `:` and `->` forms to be equally expressive, and that if `case A: case B:` is supported in the former style, then `case A, B ->` should be supported in the latter style. Consequently, the previous example suggests that we should support a `case null, String s ->` label, as follows:

```
Object o = ...
switch(o) {
    case null, String s -> System.out.println("String, including null");
    ...
}
```

The value of `o` matches this label when either it is the null reference, or it is a `String`. In both cases, the pattern variable `s` is initialized with the value of `o`.

(The reverse form, `case String s, null` is also allowed, and behaves identically.)

It is also meaningful, and not uncommon, to combine a `null` case with a `default` label, i.e.,

```
Object o = ...
switch(o) {
    ...
    case null: default:
        System.out.println("The rest (including null)");
}
```

Again, this should be supported in the `->` form. To do so we introduce a new `default` case label:

```
Object o = ...
switch(o) {
    ...
    case null, default ->
        System.out.println("The rest (including null)");
}
```

The value of `o` matches this label if either it is the null reference value, or no other labels match.

Guarded and parenthesized patterns

After a successful pattern match we often further test the result of the match. This can lead to cumbersome code, such as:

```
static void test(Object o) {
    switch (o) {
        case String s:
            if (s.length() == 1) { ... }
```

```
                else { ... }
                break;
            ...
        }
    }
```

The desired test — that `o` is a `String` of length 1 — is unfortunately split between the `case` label and the ensuing `if` statement. We could improve readability if a pattern `switch` supported the combination of a pattern and a boolean expression in a `case` label.

Rather than add another special `case` label, we enhance the pattern language by adding *guarded patterns*, written `p && e`. This allows the above code to be rewritten so that all the conditional logic is lifted into the `case` label:

```
static void test(Object o) {
    switch (o) {
        case String s && (s.length() == 1) -> ...
        case String s                      -> ...
        ...
    }
}
```

The first case matches if `o` is both a `String` *and* of length 1. The second case matches if `o` is a `String` of some other length.

Sometimes we need to parenthesize patterns to avoid parsing ambiguities. We therefore extend the language of patterns to support parenthesized patterns written `(p)`, where `p` is a pattern.

More precisely, we change the grammar of patterns. Assuming that the record patterns and array patterns of JEP 405 are added, the grammar for patterns will become:

```
Pattern:
  PrimaryPattern
  GuardedPattern

GuardedPattern:
  PrimaryPattern && ConditionalAndExpression

PrimaryPattern:
  TypePattern
  RecordPattern
  ArrayPattern
  ( Pattern )
```

A *guarded pattern* is of the form `p && e`, where `p` is a pattern and `e` is a boolean expression. In a guarded pattern any local variable, formal parameter, or exceptional parameter that is used but not declared in the subexpression must either be `final` or effectively final.

A guarded pattern `p && e` introduces the union of the pattern variables introduced by pattern `p` and expression `e`. The scope of any pattern variable declaration in `p` includes the expression `e`. This allows for patterns such as `String s && (s.length() > 1)`, which matches a value that can be cast to a `String` such that the string has a length greater than one.

A value matches a guarded pattern `p && e` if, first, it matches the pattern `p` and, second, the expression `e` evaluates to `true`. If the value does not match `p` then no attempt is made to evaluate the expression `e`.

A *parenthesized pattern* is of the form `(p)`, where `p` is a pattern. A parenthesized pattern `(p)` introduces the pattern variables that are introduced by the subpattern `p`. A value matches a parenthesized pattern `(p)` if it matches the pattern `p`.

We also change the grammar for `instanceof` expressions to:

```
InstanceofExpression:
  RelationalExpression instanceof ReferenceType
  RelationalExpression instanceof PrimaryPattern
```

This change, and the non-terminal `ConditionalAndExpression` in the grammar rule for a guarded pattern, ensure that, for example, the expression `e instanceof String s && s.length() > 1` continues to unambiguously parse as the expression `(e instanceof String s) && (s.length() > 1)`. If the trailing `&&` is intended to be part of a guarded pattern then the entire pattern should be parenthesized, e.g., `e instanceof (String s && s.length() > 1)`.

The use of the non-terminal `ConditionalAndExpression` in the grammar rule for guarded patterns also removes another potential ambiguity concerning a `case` label with a guarded pattern. For example:

```
boolean b = true;
switch (o) {
    case String s && b -> s -> s;
}
```

If the guard expression of a guarded pattern were allowed to be an arbitrary expression then there would be an ambiguity as to whether the first occurrence of `->` is part of a lambda expression or part of the switch rule, whose body is a lambda expression. Since a lambda expression can never be a valid boolean expression, it is safe to restrict the grammar of the guard expression.

Future work

- At the moment, a pattern `switch` does not support the primitive types `boolean`, `float`, and `double`. Their utility seems minimal, but support for these could be added.
- We expect that, in the future, general classes will be able to declare deconstruction patterns to specify how they can be matched against. Such deconstruction patterns can be used with a pattern `switch` to yield very succinct code. For example, if we have a hierarchy of `Expr` with subtypes for `IntExpr` (containing a single `int`), `AddExpr` and `MulExpr` (containing two `Exprs`), and `NegExpr` (containing a single `Expr`), we can match against an `Expr` and act on the specific subtypes all in one step:

  ```
  int eval(Expr n) {
       return switch(n) {
           case IntExpr(int i) -> i;
           case NegExpr(Expr n) -> -eval(n);
           case AddExpr(Expr left, Expr right) -> eval(left) + eval(right);
           case MulExpr(Expr left, Expr right) -> eval(left) * eval(right);
           default -> throw new IllegalStateException();
       };
  }
  ```

  Without such pattern matching, expressing ad-hoc polymorphic calculations like this requires using the cumbersome visitor pattern. Pattern matching is generally more transparent and straightforward.
- It may also be useful to add AND and OR patterns, to allow more expressivity for `case` labels with patterns.

Alternatives

- Rather than support a pattern `switch` we could instead define a *type `switch`* that just supports switching on the type of the selector expression. This feature is simpler to specify and implement but considerably less expressive.
- There are many other syntactic options for guarded patterns, such as `p where e`, `p when e`, `p if e`, or even `p && e`.
- An alternative to guarded patterns is to support *guards* directly as a special form of `case` label:

  ```
  SwitchLabel:
    case Pattern [ when Expression ]
    ...
  ```

  Supporting guards in `case` labels requires introducing `when` as a new contextual keyword, whereas guarded patterns do not require new contextual keywords or operators. Guarded patterns offer considerably more flexibility, since a guarded pattern can occur near where it applies rather than at the end of the switch label.

Dependencies

This JEP builds on pattern matching for `instanceof` (JEP 394) and also the enhancements offered by `switch` expressions (JEP 361). When JEP 405 (Record Patterns & Array Patterns) appears, the resulting implementation will likely make use of dynamic constants (JEP 309).