JEP 427: Pattern Matching for switch (Third 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 406: Pattern Matching for switch (Preview) JEP 420: Pattern Matching for switch (Second Preview) JEP 405: Record Patterns (Preview) JEP 433: Pattern Matching for switch (Fourth Preview)
Reviewed by	Brian Goetz
Endorsed by	Brian Goetz
Created	2022/02/22 18:01
Updated	2023/05/12 15:34
Issue	8282272

Summary

Enhance the Java programming language with pattern matching for `switch` expressions and statements. 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.

History

Pattern Matching for `switch` was proposed as a preview feature by JEP 406 and delivered in JDK 17, and proposed for a second preview by JEP 420 and delivered in JDK 18. This JEP proposes a third preview with further refinements based upon continued experience and feedback.

The main changes since the second preview are:

- Guarded patterns are replaced with `when` clauses in switch blocks.
- The runtime semantics of a pattern switch when the value of the selector expression is `null` are more closely aligned with legacy switch semantics.

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.
- Increase the safety of `switch` statements by requiring that pattern `switch` statements cover all possible input values.
- Ensure that all existing `switch` expressions and statements continue to compile with no changes and execute with identical semantics.

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 — integral primitive types (excluding `long`), their corresponding boxed forms, 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 developers do not always write such declarations.) 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 can 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 applies if the value of the selector expression `o` 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 switches 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` by allowing a new `null` case label:

```
    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`, 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 combine `null` with another `case` label. For example, in the following code, the label `case null, String s` matches both the `null` value and all `String` values:

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

Case refinement

Experimentation with patterns in `switch` suggests it is common to want to refine the test embodied by a pattern label. For example, 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 (those whose area is 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 the `break` statement 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 is to introduce *guarded patterns*, written `p && b`, that allow a pattern `p` to be refined by an arbitrary boolean expression `b`.

We implemented guarded patterns in the predecessors of this JEP. Based upon experience and feedback we propose instead to allow `when` clauses in switch blocks to specify guards to pattern labels, e.g., `case Triangle t when t.calculateArea() > 100`. We refer to such a pattern label as a *guarded* pattern label, and to the boolean expression as the *guard*.

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 null ->
            { break; }
        case Triangle t
        when t.calculateArea() > 100 ->
            System.out.println("Large triangle");
        default ->
            System.out.println("A shape, possibly a small triangle");
    }
}
```

The first clause is taken if the value of `s` matches the pattern `Triangle t` *and* subsequently the guard `t.calculateArea() > 100` evaluates to `true`. (The guard is able to use any pattern variables that are declared by the pattern in the `case` label.)

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 two case clauses, one with a guard and one without:

```
static void testTriangle(Shape s) {
    switch (s) {
        case null ->
            { break; }
        case Triangle t
        when 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 three ways:

- Extend `case` labels to include patterns and `null` in addition to constants,

- Broaden the range of types permitted for the selector expressions of both `switch` statements and `switch` expressions, and
- Allow optional `when` clauses to follow `case` labels.

For convenience, we also introduce *parenthesized patterns*.

Patterns in switch labels

The main enhancement is to introduce a new `case p` switch label, where `p` is a pattern. The essence of `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 or evaluated. The difference now is that for `case` labels with patterns, the label selected is determined by the result of 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 expression associated with the label `case Long l` is evaluated:

```
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();
};
```

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 pattern `case` label and the following `if` statement.

To address this we introduce *guarded pattern labels* by supporting an optional guard after the pattern label. This allows the above code to be rewritten so that all the conditional logic is lifted into the switch label:

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

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

Only pattern labels can have a guard. For example, it is not valid to write a label with a `case` constant and a guard, e.g. `case "Hello" when RandomBooleanExpression()`.

Sometimes we need to parenthesize patterns to improve readability. We therefore extend the language of patterns to support *parenthesized patterns* written `(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`.

There are four major language design areas to consider when supporting patterns in `switch`:

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 (excluding `long`), the corresponding boxed form (i.e., `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 (excluding `long`) 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: " + c.toString());
            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

Supporting pattern labels means that it is now possible for multiple labels to apply to the value of a selector expression (previously, all case labels were disjoint). For example, both the labels `case String s` and `case CharSequence cs` could apply to a value of type `String`.

The first issue to resolve is deciding exactly which label should apply in this circumstance. Rather than attempt a complicated best-fit approach, we adopt a simpler semantics: The first pattern label appearing in a switch block that applies to a value is chosen.

```
    static void first(Object o) {
        switch (o) {
            case String s ->
                System.out.println("A string: " + s);
            case CharSequence cs ->
                System.out.println("A sequence of length " + cs.length());
            default -> {
                break;
            }
        }
    }
```

In this example, if the value of `o` is of type `String` then the first pattern label will apply; if it is of type `CharSequence` but not of type `String` then the second pattern label will apply.

But what happens if we swap the order of these two pattern labels?

```
    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;
            }
        }
    }
```

Now if the value of `o` is of type `String` the `CharSequence` pattern label applies, since it appears first in the switch block. The `String` pattern label is unreachable in the sense that there is no value of the selector expression that would cause it to be chosen. By analogy to unreachable code, this is treated as a programmer error and results in a compile-time error.

More precisely, we say that 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`.

An unguarded pattern label dominates a guarded pattern label that has the same pattern. For example, the (unguarded) pattern label `case String s` dominates the guarded pattern label `case String s when s.length() > 0`, since every value that matches the pattern label `case String s when s.length() > 0` must match the pattern label `case String s`.

A guarded pattern label dominates another pattern label (guarded or unguarded) only when both the former's pattern dominates the latter's pattern *and* when its guard is a constant expression of value `true`. For example, the guarded pattern label `case String s when true` dominates the pattern label `case String s`. We do not analyze the guarding expression any further in order to determine more precisely which values match the pattern label (a problem which is undecidable in general).

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 the same pattern label without the `when` clause does. In other words, we do not check the guard, since this is undecidable in general. For example, the pattern label `case String s when s.length() > 1` dominates the constant label `case "hello"`, as expected; but `case Integer i when i != 0` dominates the label `case 0`.

All this suggests a simple, predictable, and readable ordering of case labels in which the constant labels should appear before the guarded pattern labels, and those should appear before the unguarded pattern labels:

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

The compiler checks all labels. It is a compile-time error for a label in a switch block to be 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 for a switch block of a `switch` expression or `switch` statement to have more than one match-all switch label. The match-all labels are `default` and pattern labels where the pattern unconditionally matches the selector expression. For example, the type pattern `String s` unconditionally matches a selector expression of type `String`, and the type pattern `Object o` unconditionally matches a selector expression of any reference type.

2. Exhaustiveness of `switch` expressions and statements

A `switch` expression requires that all possible values of the selector expression be 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 and statements, 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 switch 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 - still 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 a specific parameterization 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, in this case of a `switch` over a sealed class where the switch block is exhaustive and there is no match-all case, 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.

This condition of exhaustiveness applies to both pattern `switch` expressions and pattern `switch` *statements*. To ensure backward compatibility, all existing `switch` statements will compile unchanged. But if a `switch` statement uses any of the `switch` enhancements detailed in this JEP, then the compiler will check that it is exhaustive. (Future compilers of the Java language may emit warnings for legacy `switch` statements that are not 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). 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 switches exhaustive is just 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 pattern `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;
    }
```

The notion of exhaustiveness is made more complicated by *record patterns* (JEP 405) since these patterns support the nesting of other patterns inside them. Accordingly, the notion of exhaustiveness has to reflect this potentially recursive structure.

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 three new rules:

1. The scope of a pattern variable declaration which occurs in a switch label includes any `when` clause of that label.
2. 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.
3. 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. Falling through a `case` label that declares a pattern variable is forbidden.

This example shows the first rule in action:

```
static void test(Object o) {
    switch (o) {
        case Character c
        when c.charValue() == 7:
            System.out.println("Ding!");
            break;
        default:
            break;
        }
    }
}
```

The scope of the declaration of the pattern variable `c` includes the `when` expression of the switch label.

This variant shows the second 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: "
                                            + i.intValue());
        default -> {
            break;
        }
    }
}
```

Here 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 third 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.

We forbid the possibility of falling through a `case` label that declares a pattern variable. 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` were 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 a switch label consisting of multiple pattern labels, e.g. `case Character c: case Integer i: ...`, is not permitted. Similar reasoning applies to the prohibition of multiple patterns within a single `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, we have 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`. We then 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 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 `default` 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 see a `null` label in a switch block, then this label will match a `null` value. If you do not see a `null` label in a switch block, then switching over a `null` value will throw `NullPointerException`, as before.

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

As of Java 16, `switch` blocks 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.

Future work

- At the moment, a pattern `switch` does not support the primitive types `boolean`, `long`, `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 pattern labels, such as `p where e`, `p if e`, or even `p &&& e`.
- An alternative to guarded pattern labels is to support *guarded patterns* directly as a special pattern form, e.g. `p && e`. Having experimented with this in previous previews, the resulting ambiguity with boolean expressions have lead us to prefer `when` clauses in pattern switches.

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) appears, the resulting implementation will likely make use of dynamic constants (JEP 309).