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

JEP 441: Pattern Matching for switch

Owner	Gavin Bierman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	21
Component	specification / language
Discussion	amber dash dev at openjdk dot org
Relates to	JEP 433: Pattern Matching for switch (Fourth Preview)
Reviewed by	Brian Goetz
Endorsed by	Brian Goetz
Created	2023/01/18 14:43
Updated	2023/09/19 13:38
Issue	8300542

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.

History

This feature was originally proposed by JEP 406 (JDK 17) and subsequently refined by JEPs 420 (JDK 18), 427 (JDK 19), and 433 (JDK 20). It has co-evolved with the *Record Patterns* feature (JEP 440), with which it has considerable interaction. This JEP proposes to finalize the feature with further small refinements based upon continued experience and feedback.

Apart from various editorial changes, the main changes from the previous JEP are to:

- Remove parenthesized patterns, since they did not have sufficient value, and
- Allow qualified enum constants as `case` constants in `switch` expressions and statements.

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, making it both more concise and less error-prone:

```
// Prior to Java 16
if (obj instanceof String) {
    String s = (String)obj;
    ... use s ...
}

// As of Java 16
if (obj instanceof String s) {
    ... use s ...
}
```

In the new code, `obj` matches the type pattern `String s` if, at run time, the value of `obj` is an instance of `String`. If the pattern matches then the `instanceof` expression is `true` and the pattern variable `s` is initialized to the value of `obj` cast to `String`, which can then be used in the contained block.

We often want to compare a variable such as `obj` 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. We can only switch on values of a few types — integral primitive types (excluding `long`), their corresponding boxed forms, `enum` types, and `String` — and we 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:

```
// Prior to Java 21
static String formatter(Object obj) {
    String formatted = "unknown";
    if (obj instanceof Integer i) {
        formatted = String.format("int %d", i);
    } else if (obj instanceof Long l) {
        formatted = String.format("long %d", l);
    } else if (obj instanceof Double d) {
        formatted = String.format("double %f", d);
    } else if (obj 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 enforce this invariant. If some "then" 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:

```
// As of Java 21
static String formatterPatternSwitch(Object obj) {
    return switch (obj) {
        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        -> obj.toString();
    };
}
```

The semantics of this `switch` are clear: A `case` label with a pattern applies if the value of the selector expression `obj` 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 `obj` matches at most one of the following conditions, figure it out and evaluate the corresponding arm." As a bonus, it is more optimizable; in this case we are more likely to be able to perform the dispatch in $O(1)$ time.

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`:

```
// Prior to Java 21
```

```
static void testFooBarOld(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 reference type, and `case` labels can have type patterns, then the standalone `null` test feels like an arbitrary distinction which invites needless boilerplate and opportunities for error. It would be better to integrate the `null` test into the `switch` by allowing a new `null` case label:

```
// As of Java 21
static void testFooBarNew(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.)

Case refinement

In contrast to case labels with constants, a pattern case label can apply to many values. This can often lead to conditional code on the right-hand side of a switch rule. For example, consider the following code:

```
// As of Java 21
static void testStringOld(String response) {
    switch (response) {
        case null -> { }
        case String s -> {
            if (s.equalsIgnoreCase("YES"))
                System.out.println("You got it");
            else if (s.equalsIgnoreCase("NO"))
                System.out.println("Shame");
            else
                System.out.println("Sorry?");
        }
    }
}
```

The problem here is that using a single pattern to discriminate among cases does not scale beyond a single condition. We would prefer to write multiple patterns but we then need some way to express a refinement to a pattern. We therefore allow `when` clauses in switch blocks to specify guards to pattern case labels, e.g., `case String s when s.equalsIgnoreCase("YES")`. We refer to such a case label as a *guarded* case label, and to the boolean expression as the *guard*.

With this approach, we can rewrite the above code using guards:

```
// As of Java 21
static void testStringNew(String response) {
    switch (response) {
        case null -> { }
        case String s
        when s.equalsIgnoreCase("YES") -> {
            System.out.println("You got it");
        }
        case String s
```

```
            when s.equalsIgnoreCase("NO") -> {
                System.out.println("Shame");
            }
            case String s -> {
                System.out.println("Sorry?");
            }
        }
    }
```

This leads to a more readable style of `switch` programming where the complexity of the test appears on the left of a switch rule, and the logic that applies if that test is satisfied is on the right of a switch rule.

We can further enhance this example with extra rules for other known constant strings:

```
// As of Java 21
static void testStringEnhanced(String response) {
    switch (response) {
        case null -> { }
        case "y", "Y" -> {
            System.out.println("You got it");
        }
        case "n", "N" -> {
            System.out.println("Shame");
        }
        case String s
        when s.equalsIgnoreCase("YES") -> {
            System.out.println("You got it");
        }
        case String s
        when s.equalsIgnoreCase("NO") -> {
            System.out.println("Shame");
        }
        case String s -> {
            System.out.println("Sorry?");
        }
    }
}
```

These examples shows how case constants, case patterns, and the `null` label combine to showcase the new power of `switch` programming: We can simplify complicated conditional logic that was formerly mixed with business logic into a readable, sequential list of switch labels with the business logic to the right of the switch rules.

Switches and enum constants

The use of enum constants in case labels is, at present, highly constrained: The selector expression of the `switch` must be of the enum type, and the labels must be simple names of the enum's constants. For example:

```
// Prior to Java 21
public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

static void testforHearts(Suit s) {
    switch (s) {
        case HEARTS -> System.out.println("It's a heart!");
        default -> System.out.println("Some other suit");
    }
}
```

Even after adding pattern labels, this constraint leads to unnecessarily verbose code. For example:

```
// As of Java 21
sealed interface CardClassification permits Suit, Tarot {}
public enum Suit implements CardClassification { CLUBS, DIAMONDS, HEARTS, SPADES }
final class Tarot implements CardClassification {}

static void exhaustiveSwitchWithoutEnumSupport(CardClassification c) {
    switch (c) {
        case Suit s when s == Suit.CLUBS -> {
            System.out.println("It's clubs");
        }
        case Suit s when s == Suit.DIAMONDS -> {
            System.out.println("It's diamonds");
        }
        case Suit s when s == Suit.HEARTS -> {
            System.out.println("It's hearts");
        }
        case Suit s -> {
            System.out.println("It's spades");
        }
        case Tarot t -> {
            System.out.println("It's a tarot");
        }
    }
}
```

This code would be more readable if we could have a separate case for each enum constant rather than lots of guarded patterns. We therefore relax the requirement that the selector expression be of the enum type and we allow case constants to use qualified names of enum constants. This allows the above code to be rewritten as:

```
    // As of Java 21
    static void exhaustiveSwitchWithBetterEnumSupport(CardClassification c) {
        switch (c) {
            case Suit.CLUBS -> {
                System.out.println("It's clubs");
            }
            case Suit.DIAMONDS -> {
                System.out.println("It's diamonds");
            }
            case Suit.HEARTS -> {
                System.out.println("It's hearts");
            }
            case Suit.SPADES -> {
                System.out.println("It's spades");
            }
            case Tarot t -> {
                System.out.println("It's a tarot");
            }
        }
    }
```

Now we have a direct case for each enum constant without using guarded type patterns, which were previously used simply to work around the present constraint of the type system.

Description

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

- Improve enum constant case labels,
- 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 (along with the required richer analysis of exhaustiveness of switch blocks), and
- Allow optional when clauses to follow case labels.

Improved enum constant case labels

It has long been a requirement that, when switching over an enum type, the only valid case constants were enum constants. But this is a strong requirement that becomes burdensome with the new, richer forms of switch.

To maintain compatibility with existing Java code, when switching over an enum type a case constant can still use the simple name of a constant of the enum type being switched over.

For new code, we extend the treatment of enums. First, we allow qualified names of enum constants to appear as case constants. These qualified names can be used when switching over an enum type.

Second, we drop the requirement that the selector expression be of an enum type when the name of one of that enum's constants is used as a case constant. In that situation we require the name to be qualified and its value to be assignment compatible with the type of the selector expression. (This aligns the treatment of enum case constants with the numerical case constants.)

For example, the following two methods are allowed:

```
// As of Java 21
sealed interface Currency permits Coin {}
enum Coin implements Currency { HEADS, TAILS }

static void goodEnumSwitch1(Currency c) {
    switch (c) {
        case Coin.HEADS -> {    // Qualified name of enum constant as a label
            System.out.println("Heads");
        }
        case Coin.TAILS -> {
            System.out.println("Tails");
        }
    }
}

static void goodEnumSwitch2(Coin c) {
    switch (c) {
        case HEADS -> {
            System.out.println("Heads");
        }
        case Coin.TAILS -> {    // Unnecessary qualification but allowed
            System.out.println("Tails");
        }
    }
}
```

The following example is not allowed:

```
// As of Java 21
static void badEnumSwitch(Currency c) {
    switch (c) {
        case Coin.HEADS -> {
            System.out.println("Heads");
        }
        case TAILS -> {         // Error - TAILS must be qualified
            System.out.println("Tails");
        }
        default -> {
            System.out.println("Some currency");
        }
    }
}
```

Patterns in switch labels

We revise the grammar for switch labels in a switch block to read (compare JLS §14.11.1):

```
SwitchLabel:
  case CaseConstant { , CaseConstant }
  case null [, default]
  case Pattern [ Guard ]
  default
```

The main enhancement is to introduce a new case label, `case p`, 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 `obj` matches the pattern `Long l`, and the expression associated with the label `case Long l` is evaluated:

```
// As of Java 21
static void patternSwitchTest(Object obj) {
    String formatted = switch (obj) {
        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        -> obj.toString();
    };
}
```

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

```
// As of Java 21
static void testOld(Object obj) {
    switch (obj) {
        case String s:
            if (s.length() == 1) { ... }
            else { ... }
            break;
        ...
    }
}
```

The desired test — that `obj` 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 case labels* by allowing an optional *guard*, which is a boolean expression, to follow the pattern label. This permits the above code to be rewritten so that all the conditional logic is lifted into the switch label:

```
// As of Java 21
static void testNew(Object obj) {
    switch (obj) {
        case String s when s.length() == 1 -> ...
        case String s                      -> ...
        ...
    }
}
```

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

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

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

- Enhanced type checking
- Exhaustiveness of `switch` expressions and statements
- Scope of pattern variable declarations
- Dealing with `null`
- Errors

Enhanced type checking

Selector expression typing

Supporting patterns in `switch` means that we can relax the 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 `obj` 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`:

```
// As of Java 21
```

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

static void typeTester(Object obj) {
    switch (obj) {
        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).

Dominance of case labels

Supporting pattern case labels means that for a given value of the selector expression it is now possible for more than one case label to apply, whereas previously at most one case label could apply. For example, if the selector expression evaluates to a `String` then both the case labels `case String s` and `case CharSequence cs` would apply.

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 case label appearing in a switch block that applies to a value is chosen.

```
// As of Java 21
static void first(Object obj) {
    switch (obj) {
        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 `obj` is of type `String` then the first case 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 case labels?

```
// As of Java 21
static void error(Object obj) {
    switch (obj) {
        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 `obj` is of type `String` the `CharSequence` case label applies, since it appears first in the switch block. The `String` case 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 case label `case CharSequence cs` *dominates* the second case 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 case label dominates a guarded pattern case label that has the same pattern. For example, the (unguarded) pattern case label case String s dominates the guarded pattern case label case String s when s.length() > 0, since every value that matches the case label case String s when s.length() > 0 must match the case label case String s.

A guarded pattern case label dominates another pattern case 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 case label case String s when true dominates the pattern case 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 case label can dominate a constant case label. For example, the pattern case label case Integer i dominates the constant case label case 42, and the pattern case label case E e dominates the constant case label case A when A is a member of enum class type E. A guarded pattern case label dominates a constant case label if the same pattern case label without the guard does. In other words, we do not check the guard, since this is undecidable in general. For example, the pattern case label case String s when s.length() > 1 dominates the constant case label case "hello", as expected; but case Integer i when i != 0 dominates the case label case 0.

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

```
// As of Java 21
Integer i = ...
switch (i) {
    case -1, 1 -> ...                   // Special cases
    case Integer j when j > 0 -> ...    // Positive integer cases
    case Integer j -> ...               // All the remaining integers
}
```

The compiler checks all case labels. It is a compile-time error for a case label in a switch block to be dominated by any preceding case 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 case 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:

```
// As of Java 21
static void matchAll(String s) {
    switch(s) {
        case String t:
            System.out.println(t);
            break;
        default:
            System.out.println("Something else");  // Error - dominated!
    }
}

static void matchAll2(String s) {
    switch(s) {
        case Object o:
            System.out.println("An Object");
            break;
        default:
            System.out.println("Something else");  // Error - dominated!
    }
}
```

Exhaustiveness of `switch` expressions and statements

Type coverage

A `switch` expression requires that all possible values of the selector expression be handled in the switch block; in other words, it must be *exhaustive*. This maintains the property that successful evaluation of a `switch` expression always yields a value.

For normal `switch` expressions, this property is enforced by a 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:

```
// As of Java 21
static int coverage(Object obj) {
    return switch (obj) {           // Error - not exhaustive
        case String s -> s.length();
    };
}
```

The switch block has only one switch label, `case String s`. This matches any value of `obj` 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:

```
// As of Java 21
static int coverage(Object obj) {
    return switch (obj) {           // 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 every type, so this example is (at last!) legal:

```
// As of Java 21
static int coverage(Object obj) {
    return switch (obj) {
        case String s  -> s.length();
        case Integer i -> i;
        default -> 0;
    };
}
```

Exhaustiveness in practice

The notion of type coverage already exists in non-pattern `switch` expressions. For example:

```
// As of Java 20
enum Color { RED, YELLOW, GREEN }

int numLetters = switch (color) {   // Error - not exhaustive!
    case RED -> 3;
    case GREEN -> 5;
}
```

This switch expression over an enum class is not exhaustive because the anticipated input YELLOW is not covered. As expected, adding a case label to handle the YELLOW enum constant is sufficient to make the switch exhaustive:

```
// As of Java 20
int numLetters = switch (color) {   // Exhaustive!
    case RED -> 3;
    case GREEN -> 5;
    case YELLOW -> 6;
}
```

That a switch written this way is exhaustive has two important benefits.

First, it would be cumbersome to have to write a default clause, which probably just throws an exception, since we have already handled all the cases:

```
int numLetters = switch (color) {
    case RED -> 3;
    case GREEN -> 5;
    case YELLOW -> 6;
    default -> throw new ArghThisIsIrritatingException(color.toString());
}
```

Manually writing a default clause in this situation is not only irritating but actually pernicious, since the compiler can do a better job of checking exhaustiveness without one. (The same is true of any other match-all clause such as default, case null, default, or an unconditional type pattern.) If we omit the default clause then we will discover at compile time if we have forgotten a case label, rather than finding out at run time — and maybe not even then.

More importantly, what happens if someone later adds another constant to the Color enum? If we have an explicit match-all clause then we will only discover the new constant value if it shows up at run time. But if we code the switch to cover all the constants known at compile time, and omit the match-all clause, then we will find out about this change the next time we recompile the class containing the switch. A match-all clause risks sweeping exhaustiveness errors under the rug.

In conclusion: An exhaustive switch without a match-all clause is better than an exhaustive switch with one, when possible.

Looking to run time, what happens if a new Color constant is added, and the class containing the switch is not recompiled? There is a risk that the new constant will be exposed to our switch. Because this risk is always present with enums, if an exhaustive enum switch does not have a match-all clause then the compiler will synthesize a default clause that throws an exception. This guarantees that the switch cannot complete normally without selecting one of the clauses.

The notion of exhaustiveness is designed to strike a balance between covering all reasonable cases while not forcing you to write possibly many rare corner cases that will pollute or even dominate your code for little actual value. Put another way: Exhaustiveness is a compile-time approximation of true run-time exhaustiveness.

Exhaustiveness and sealed classes

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, which as argued above is good practice. Consider the following example of a sealed interface S with three permitted subclasses A, B, and C:

```
// As of Java 21
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:

```
// As of Java 21
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>` and no parameterization of `A` is a subtype of `I<Integer>`.

Again, the notion of exhaustiveness is an approximation. Because of separate compilation, it is possible for a novel implementation of the interface `I` to show up at runtime, so the compiler will in this case insert a synthetic `default` clause that throws.

The notion of exhaustiveness is made more complicated by record patterns (JEP 440) since record patterns can be nested. Accordingly, the notion of exhaustiveness must reflect this potentially recursive structure.

Exhaustiveness and compatibility

The requirement 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 described 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:

```
// As of Java 21
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;
    };
}
```

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`:

```
// As of Java 21
```

```
static void testFlowScoping(Object obj) {
    if ((obj 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 parts of the code where the pattern variable s will have been initialized. In this example, that is in the right-hand operand of the && expression and in the "then" block. However, s 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 the pattern of a guarded case label includes the guard, i.e., the when expression.
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:

```
// As of Java 21
static void testScope1(Object obj) {
    switch (obj) {
        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 guard, i.e., the expression `c.charValue() == 7`.

This variant shows the second rule in action:

```
// As of Java 21
static void testScope2(Object obj) {
    switch (obj) {
        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:

```
// As of Java 21
static void testScope3(Object obj) {
    switch (obj) {
        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:

```
    // As of Java 21
    static void testScopeError(Object obj) {
        switch (obj) {
            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 `obj` 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 `obj` 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:

```
    // As of Java 21
    void testScope4(Object obj) {
        switch (obj) {
            case String s:
                System.out.println("A string: " + s);  // s in scope here!
            default:
                System.out.println("Done");            // s not in scope here
        }
    }
```

Dealing with `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. There are, however, reasonable and non-exception-raising semantics for pattern matching and null values, so in pattern switch blocks we can treat null in a more regular fashion whilst remaining compatible with existing switch semantics.

First, we introduce a new null case label. 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 nullMatch(null) will print null! rather than throw NullPointerException:

```
// As of Java 21
static void nullMatch(Object obj) {
    switch (obj) {
        case null     -> System.out.println("null!");
        case String s -> System.out.println("String");
        default       -> System.out.println("Something else");
    }
}
```

A switch block without a case null label is treated as if it has a case null rule whose body throws NullPointerException. In other words, this code:

```
// As of Java 21
static void nullMatch2(Object obj) {
    switch (obj) {
        case String s  -> System.out.println("String: " + s);
        case Integer i -> System.out.println("Integer");
        default        -> System.out.println("default");
    }
}
```

is equivalent to:

```
// As of Java 21
static void nullMatch2(Object obj) {
    switch (obj) {
        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 nullMatch(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 can directly handle this case inside the switch. If you see a null label in a switch block then that 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. The treatment of null values in switch blocks is thus regularized.

It is meaningful, and not uncommon, to want to combine a null case with a default. To that end we allow null case labels to have an optional default; for example:

```
// As of Java 21
Object obj = ...
switch (obj) {
    ...
    case null, default ->
        System.out.println("The rest (including null)");
    }
```

The value of `obj` matches this label if either it is the null reference value, or none of the other case labels match.

It is a compile-time error for a switch block to have both a `null` case label with a `default` and a `default` label.

Errors

Pattern matching can complete abruptly. For example, when matching a value against a record pattern, the record's accessor method can complete abruptly. In this case, pattern matching is defined to complete abruptly by throwing a `MatchException`. If such a pattern appears as a label in a `switch` then the `switch` will also complete abruptly by throwing a `MatchException`.

If a `case` pattern has a guard, and evaluating the guard completes abruptly, then the `switch` completes abruptly for the same reason.

If no label in a pattern `switch` matches the value of the selector expression then the `switch` completes abruptly by throwing a `MatchException`, since pattern switches must be exhaustive.

For example:

```
    // As of Java 21
    record R(int i) {
        public int i() {    // bad (but legal) accessor method for i
            return i / 0;
        }
    }

    static void exampleAnR(R r) {
        switch(r) {
            case R(var i): System.out.println(i);
        }
    }
```

The invocation `exampleAnR(new R(42))` causes a `MatchException` to be thrown. (A record accessor method which always throws an exception is highly irregular, and an exhaustive pattern `switch` which throws a `MatchException` is highly unusual.)

By contrast:

```
    // As of Java 21
    static void example(Object obj) {
        switch (obj) {
            case R r when (r.i / 0 == 1): System.out.println("It's an R!");
            default: break;
        }
    }
```

The invocation `example(new R(42))` causes an `ArithmeticException` to be thrown.

To align with pattern `switch` semantics, `switch` expressions over `enum` classes now throw `MatchException` rather than `IncompatibleClassChangeError` when no switch label applies at run time. This is a minor incompatible change to the language. (An exhaustive `switch` over an enum fails to match only if the `enum` class is changed after the `switch` has been compiled, which is highly unusual.)

Future work

- At the moment, pattern `switch` does not support the primitive types `boolean`, `long`, `float`, and `double`. Allowing these primitive types would also mean allowing them in `instanceof` expressions, and aligning primitive type patterns with reference type patterns, which would require considerable additional work. This is left for a possible future JEP.
- 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 `Expr`s), and `NegExpr` (containing a single `Expr`), we can match against

an `Expr` and act on the specific subtypes all in one step:

```
// Some future Java
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 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 earlier previews, the resulting ambiguity with boolean expressions led us to prefer guarded `case` labels over guarded patterns.

Dependencies

This JEP builds on *Pattern Matching for `instanceof`* (JEP 394), delivered in JDK 16, and also the enhancements offered by *Switch Expressions* (JEP 361). It has co-evolved with *Record Patterns* (JEP 440).