Pattern matching for Switch

Pattern Matching for Switch (JEP 441)

📦 Finalized in Java 21

👤 Gavin Bierman

Pattern matching for Switch - The Problem ❌

Pre-Java 21:

- `switch` works only with **int, String, enums**
- Case labels must be **constants**

Complex branching = chain of `if/else` with `instanceof`

❌ Boilerplate casts

❌ Error-prone

❌ Hard to ensure **exhaustiveness**

Pattern matching for Switch - The Goal 🎯

Make `switch` more **expressive & type-safe**

Allow **patterns** as case labels

Integrate **null** into `switch` safely

Ensure **exhaustive handling** (esp. sealed types)

Keep **backward compatibility**

Pattern matching for Switch - From instanceof to Pattern Switch

Before (Java 16+ instanceof):

```
if (s instanceof Rectangle r)
 return 2*r.length() + 2*r.width();
else if (s instanceof Circle c)
 return 2*c.radius()*Math.PI;
```

Now (Java 21 switch):

```
return switch (s) {
 case Rectangle r -> 2*r.length() + 2*r.width();
 case Circle c    -> 2*c.radius()*Math.PI;
 default          -> throw new IllegalArgumentException();
};
```

✅ No casts, clearer, safer

`switch` selector =

- Any **reference type**
- `int` (other primitives excluded for now)

Case labels can be:

- Constants
- **Patterns** (class, record, array)
- `null`

Pattern matching for Switch - Example: Type Patterns

```
record Point(int x, int y) {}
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);
   case Point p  -> System.out.println("Point " + p);
   case int[] ia -> System.out.println("int[] length " + ia.length);
   default        -> System.out.println("Other");
 }
}
```

Pattern matching for Switch - Guarded Patterns (when)

```
static void test(Object obj) {
 switch (obj) {
   case String s when s.length() == 1 ->
           System.out.println("Short: " + s);
   case String s ->
           System.out.println(s);
   default ->
           System.out.println("Not a string");
 }
}
```

✅ Cleaner than splitting into if/else inside case

Pattern matching for Switch - Enums

Before (verbose, guarded patterns):

```
case Suit s when s == Suit.CLUBS -> ...
```

Now (qualified constants allowed):

```
case Suit.CLUBS -> System.out.println("Clubs");
case Suit.HEARTS -> System.out.println("Hearts");
```

✅ Direct & readable, even across sealed hierarchies

Pattern matching for Switch - Null in Switch

- Old behavior: `switch` on `null` → NPE
- New: explicit `null` label supported

```
switch (obj) {
 case null      -> System.out.println("null!");
 case String s -> System.out.println("String");
 default        -> System.out.println("Other");
}
```

✅ You decide whether `null` is handled

Pattern matching for Switch - Exhaustiveness ✅

- Switch must cover **all possible values**
- Ways to ensure:
  - `default` label
  - Cover all **enum constants**
  - Cover all **sealed type subtypes**

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

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

✅ No `default` needed → compiler ensures completeness

Pattern matching for Switch - Dominance Rules ⚖️

- First matching case wins
- Compiler error if a case is **dominated** by a previous one

```
switch (obj) {
 case CharSequence cs -> ...
 case String s -> ...  // ❌ unreachable
}
```

Pattern matching for Switch - Key Takeaways 💡

`switch` now works with **patterns + null**

Safer & more expressive branching

Exhaustiveness guaranteed by compiler

Enum + sealed hierarchies supported directly

Much less boilerplate