Object Oriented Programming in Java

Pattern Matching in Java

Suhel Hammoud

Pattern Matching

- Pattern matching, Key Concepts
- Pattern Matching for `instanceof`
- Pattern Matching for `switch`
- Guarded Pattern Matching in `switch`
- Record Patterns
- Nested Record Patterns in `switch`

Pattern Matching: Key Concepts

Pattern matching in Java is broader than regular expressions.

Pattern Matching includes three main concepts:

- **Matched Target**: What you want to match (e.g., an object or a string).
- **Pattern**: The condition or structure to match (e.g., regex or type).
- **Result**: The outcome of a successful match (e.g., group, start/end indexes).

These apply across all kinds of pattern matching.

Pattern Matching for instanceof

Before Java 16

```java
public void print(Object o) {
    if (o instanceof String){
      String s = (String) o;
      System.out.println("This is a String of length " + s.length());
    } else {
      System.out.println("This is not a String");
    }
}
```

Pattern Matching for instanceof

Introduced in **Java SE 16**.

```java
public void print(Object o) {
    if (o instanceof String s){
        System.out.println("This is a String of length " + s.length());
    } else {
        System.out.println("This is not a String");
    }
}
```

- o is machted target
- String s is a **type pattern**.
- If the check passes, s is result usable in the scope.

Extended instanceof Usage

Pattern variables within boolean conditions:

```
if (o instanceof String s && !s.isEmpty()) {
    System.out.println("Non-empty string: " + s.length());
}
```

Pattern matching to simplify early returns:

```
if (!(o instanceof String s)) {
  return;
}
System.out.println("Length: " + s.length());
```

Compiler-Aware Matching

Some type checks are statically invalid:

```
Double pi = Math.PI;
if (pi instanceof String s) {
    // Compile-time error
}
```

The compiler rejects it because String is final and cannot match Double.

Cleaner Code with Pattern Matching

Example

```java
public class Point {
    private int x;
    private int y;

    public boolean equals(Object o) {
        if (!(o instanceof Point)) {
            return false;
        }
        Point point = (Point) o;
        return x == point.x && y == point.y;
    }

    // constructor, hashCode method and accessors have been omitted
}
```

Cleaner Code with Pattern Matching

Rewrite `equals()` using pattern matching:

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

It's more concise and readable than the traditional instanceof + cast pattern.

Pattern Matching for switch

Example:

```
Object o = ...; // any object
String formatted = null;
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 {
    formatted = String.format("Object %s", o.toString());
}
```

Pattern Matching for switch

Introduced in **JDK 21**.

Example using switch:

```java
Object o = ...; // any object
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);
    default        -> String.format("Object %s", o.toString());
};
```

Switch-based matching is **O(1)**, vs **O(n)** for if-else chains.

Guarded Pattern Matching in switch

Can we check condition in switch case similar to instanceof?

```
if (object instanceof String s && !s.isEmpty()) { }
```

Yes we can! Switch cases can now use guard clauses:

```
String result = switch(o) {
    case String s when !s.isEmpty() -> "Non-empty string: " + s;
    default                          -> "Other: " + o;
};
```

This combines type matching with boolean checks.

Record Patterns

```java
record Point(int x, int y) {}
```

```java
if (o instanceof Point p) {
    int a = p.x();
    int b =p.y();
    //Use a and b
}
```

Better way: Deconstruct it.

```java
if (o instanceof Point(int a, int b)) {
    // Use a and b
}
```

Record Patterns: Details

- Pattern matches **accessors** of the record.
- Based on the **canonical constructor**.
- Supports **type inference** using var.

```
record Point(double x, double y) {}
if (o instanceof Point(var x, var y)) {
    // x, y are doubles
}
```

Record Patterns in switch

Record patterns also work in switch:

```
record Box(Object o) {}

switch (o) {
    case Box(String s) -> ...
    case Box(Integer i) -> ...
    default -> ...
}
```

The compiler ensures type safety.

Record Pattern: Restrictions

Invalid matches:

```
record Box(CharSequence o) {}
// Will not compile:
case Box(Integer i) -> ...
```

No boxing/unboxing support:

```
record Point(Integer x, Integer y) {}
// Invalid:
if (o instanceof Point(int x, int y)) {}
```

Nested Record Patterns

You can nest record patterns:

```java
record Point(double x, double y) {}

record Circle(Point center, double radius) {}

if (o instanceof Circle(Point(var x, var y), var r)) {
    // Use x, y, and r
}
```

Great for matching complex structures.

Nested Record Patterns, using _ character

You can extract certain fields from record patterns:

```java
record Point(double x, double y) {}

record Circle(Point center, double radius) {}

if (o instanceof Circle(Point(_, m), _)) {
  // Use m
}

if (o instanceof Circle(_, var r)) {
  // Use r
}
```

Summary

Pattern matching now supported in:

- `instanceof` keyword
- `switch` statement/expression

Supported pattern types:

- **Type patterns**
- **Record patterns**

References:

- docs.oracle.com/en/java/javase/17/language/pattern-matching-instanceof.html
- dev.java/learn/pattern-matching
- https://github.com/suhelhammoud/java-course/