Pattern Matching For Switch

1.) Using a *switch* expression or a pattern over an enum class now throws a *MatchException*. Earlier, we used to get an *IncompatibleClassChangeError* if no *switch* label was applied at run time.
2.) They have added support for type-inference of arguments for generic record patterns in *switch* expressions and statements, along with the other constructs that support patterns.

Code:

```
/**
 * To run: `java --enable-preview --source 20
PatternMatchingForSwitchFourthPreviewTest.java`
 */
public class PatternMatchingForSwitchFourthPreviewTest {
 public static void main(String[] args) {
 recordErrorInSwitchPatternMatching();

 genericRecordInSwitch();
 }

 static void recordErrorInSwitchPatternMatching() {
 var dot = new OneDimensionalPoint(10);

 switch (dot) {
 // will cause MatchException with wrapped exception (the record pattern
completes abruptly with the ArithmeticException)
 case OneDimensionalPoint(var x): System.out.println("1D point");
 // the occurring in guarded clause, it just rethrows the exception
 // will cause ArithmeticException
 // case OneDimensionalPoint p when (p / 0 == 1): System.out.println("Non
sense");
 }
 }

 static void genericRecordInSwitch() {
 var w = new Wrapper<String>("some text");
 switch (w) {
 // will infer Wrapper<String>
 case Wrapper(var v): System.out.println("Wrapped value: " + v);
 }
 }
}

record OneDimensionalPoint(int x) {
 public int x() {
 return x / 0;
 }
}

record Wrapper<T>(T t) {}
```

<u>Output:</u>

```
Exception in thread "main" java.lang.MatchException: java.lang.ArithmeticException: / by zero

at
PatternMatchingForSwitchFourthPreviewTest.recordErrorInSwitchPatternMatching(PatternMatc
hingForSwitchFourthPreviewTest.java:16)

at
PatternMatchingForSwitchFourthPreviewTest.main(PatternMatchingForSwitchFourthPreviewTe
st.java:6)

Caused by: java.lang.ArithmeticException: / by zero

at OneDimensionalPoint.x(PatternMatchingForSwitchFourthPreviewTest.java:35)

at
PatternMatchingForSwitchFourthPreviewTest.recordErrorInSwitchPatternMatching(PatternMatc
hingForSwitchFourthPreviewTest.java:1)
```

Record Patterns Second Preview

1.) Added support for type inference of arguments of generic record patterns.

2.) Added support for record patterns to be usable in the header of an enhanced *for* loop.
3.) Removed support for named record patterns, where we could provide an optional identifier to the record patterns that we can use to refer to the record pattern.

Code:

```
import java.util.List;
import java.util.ArrayList;

/**
 * To run: `java --enable-preview --source 20
RecordsPatternSecondPreviewTest.java`
 */
public class RecordPatternsSecondPreviewTest {
 public static void main(String[] args) {
 enhancedForLoop();

 genericInferrenceTest();

 recordPatternInEnhancedForLoopHeader();
 }

 public static void enhancedForLoop() {
 var points = new Point[] {
 new Point(10, 10),
 new Point(20, 20),
 new Point(30, 30),
 new Point(20, 50),
 new Point(10, 60)
 };

 // we can now deconstruct a record type in the enhanced for loop
 for (Point(int x, int y) : points) {
 System.out.printf("Drawing at x=%d and y=%d%n", x, y);
 }
 }
 public static void genericInferrenceTest() {
 var point = new Point(42, 42);
 var decoratedPoint = new Decorator(new ColoredPoint(point, "RED"));
 var anotherDecorated = new Decorator(decoratedPoint);

 // here we don't need to use
`Decorator<Decorator<ColoredPoint>>(Decorator<ColoredPoint>(ColoredPoi
nt cp))` like in JDK 19
 if (anotherDecorated instanceof Decorator(Decorator(ColoredPoint(Point(int x, int
y), String color)))) {
 System.out.println("\nAren't you using too much decorator?");
 System.out.printf("x=%d, y=%d; color=%s%n%n", x, y, color);
 }
 }

 static void recordPatternInEnhancedForLoopHeader() {
 var items = new ColoredPoint[] { new ColoredPoint(new Point(42, 42), "red") };

 for (ColoredPoint(Point(var x, var y), String color) : items) {
 System.out.printf("Point [%d, %d] has color %s", x, y, color);
 }
 }
}

record Point(int x, int y) {}

record ColoredPoint(Point p, String color) {}

record Decorator<T>(T t) {}
```

Output:

```
Drawing at x=10 and y=10

Drawing at x=20 and y=20

Drawing at x=30 and y=30

Drawing at x=20 and y=50

Drawing at x=10 and y=60
```

```
Aren't you using too much decorator?
x=42, y=42; color=RED
Point [42, 42] has color red
```

Structured Concurrency With Scoped Value

Scoped values provide a simple, immutable, and inheritable data-sharing option, specifically in situations where we're working with a large number of threads.

A *ScopedValue* is an immutable value that is available for reading for a bounded period of execution by a thread. Since it's immutable, it allows safe and easy data-sharing for a limited period of thread execution. Also, we need not pass the values as method arguments.

Code:

```
 import jdk.incubator.concurrent.*;

/**
 * Run: `java --source 20 --enable-preview --add-modules jdk.incubator.concurrent
ScopedValueUsageWithReturnValueExample.java`
 */
public class ScopedValueUsageWithReturnValueExample {
 final static ScopedValue<Integer> MAIN_SCOPE = ScopedValue.newInstance();

 public static void main(String[] args) throws Exception {
 // we use `call` to run a scope and get it returned value
 var result = ScopedValue.where(MAIN_SCOPE, 42)
 .call(() -> { // throws Exception
 var calculator = new Calculator();
 return calculator.calculate();
 });
 System.out.println("Result from calculation: " + result);
 }
}
```

```
class Calculator {
 public int calculate() {
 var seed = ScopedValueUsageWithReturnValueExample.MAIN_SCOPE.get();
 return seed + 42;
 }
}
```

Output:

```
Result from calculation: 42
```