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:

```
import java.util.*;

/**
 * To run: `java PatternMatchingForSwitchTest.java`
 */
public class PatternMatchingForSwitchTest {
 public static void main(String[] args) {
 switchExhaustivenessAndCompatibility();
 switchExpressionWithPatternMatching();
 switchScopeWithPatternMatching();
 switchEnhancedTypeChecking();
 switchWithGuardedCaseLabel();
 switchWithEnumConstants();
 }

 static void switchExhaustivenessAndCompatibility() {
 /**
 * compiler won't require exhaustiveness if it doesn't use any pattern or null label,
or if selector
 * expression is from a legacy type (char, byte, short, int, Character, Byte, Short,
Integer, String or a enum)
 */
 int i = 42;
 switch (i) {
 case 0:
 System.out.println("Zero");
 break;
 case 42:
 System.out.println("42!");
 break;
 }

 var shape = Shape.TRIANGLE;
 // default case way
 switch (shape) {
 case TRIANGLE:
 System.out.println("Is a triangle");
 break;
 case CIRCLE:
 System.out.println("Is a circle");
 break;
 }

 // lambda expression way
 switch (shape) {
 case TRIANGLE -> System.out.println("Is a triangle");
 case CIRCLE -> System.out.println("Is a circle");
 }
 }

 static void switchExpressionWithPatternMatching() {
 Object obj = 42;

 // if a switch use any feature from the JEP, the compiler will check for
exhaustiveness
 String message = switch (obj) {
 case Integer i -> String.format("int %d", i);
 case String s -> String.format("string %s", s);
 case Double d -> String.format("double %d", d);
 // JDK 17: we can test null here, without check before the switch =)
 // switch case without null will throw a NullPointerException (compatibility)
 case null -> "null =s";
 // required to be exhaustive
 default -> obj.toString();
 // we can only combine null with default in a the same case
 // null, default -> obj.toString();
 };
 System.out.println(message);
 }

 static void switchScopeWithPatternMatching() {
 Number number = 42;

 String message = null;

 switch (number) {
 case Integer i when i == 0:
 System.out.println("zero =0");
 // doesn't allow fall-through (must have the break or yield when using `:` in a
switch statement or expression)
 // error: illegal fall-through to a pattern
 break;

 case Integer x when x < 0:
 message = "zero or lower";
 break;
 case Integer n when n == 21:
 message = "half of the answer";
 break;
 case Integer n when n == 42:
 message = "answer";
 break;
 default:
 message = "unhandled";
 }
 System.out.println("switch statement: " + message);

 message = switch (number) {
 case Integer i when i == 0:
 System.out.println("zero =0");
 yield "zero =0";
 case Integer x when x < 0:
 yield "zero or lower";
```

```
case Integer n when n == 21:
yield "half of the answer";
case Integer n when n == 42:
yield "answer";
default:
yield "unhandled";
};
System.out.println("switch expression: " + message);
}

static void switchEnhancedTypeChecking() {
Object value = 42;
var message = switch (value) {
case null -> "The value is `null`";
case String s -> "Is String: " + s;
case Integer n -> "is an integer: " + n;
case Number n -> "Is a Number: " + n;
case int[] ar -> "Is an array of number: " + ar;
case List list -> "Is a list of some type: " + list;
// can infer the record generic type
case Wrapper(var v) -> "Wrapped value: " + v;
default -> "Is untested type =(: " + value.toString();
};
System.out.println(message);
}

static void switchWithGuardedCaseLabel() {
Object obj = 42;

// JDK 20 changed `&&` to `when`
String message = switch (obj) {
// JDK 21 removes the ()
// case Integer i when (i < 0) -> "negative number";
case Integer i when i < 0 -> "negative number";
case Integer i when i == 0 -> "zero";
case Integer i when i > 0 && i <= 100 -> "positive number between 0 and 100";
// must be after the guarded case as it is dominant
```

```
 case Integer i -> "number";
 // required to be exhaustive
 default -> obj.toString();
 };
 System.out.println(message);
 }

 static void switchWithEnumConstants() {
 Object shape = Shape.TRIANGLE;

 // before
 var shapeName = switch (shape) {
 case Shape s when s == Shape.CIRCLE -> "Circle";
 case Shape s when s == Shape.RECTANGLE -> "Rectangle";
 case Shape s when s == Shape.TRIANGLE -> "Triangle";
 default -> "none";
 };

 System.out.println(shapeName);
 }
}

enum Shape { CIRCLE, RECTANGLE, TRIANGLE }

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

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

<u>Output:</u>

```
 42!
Is a triangle
Is a triangle
```

```
int 42
switch statement: answer
switch expression: answer
is an integer: 42
positive number between 0 and 100
Triangle
```

Record Pattern

Enhance the Java programming language with *record patterns* to deconstruct record values. Record patterns and type patterns can be nested to enable a powerful, declarative, and composable form of data navigation and processing.

Code:

```
import java.util.*;

/**
 * To run: `java RecordPatternTest.java`
 */
public class RecordPatternTest {
 public static void main(String[] args) {
 instanceofExample();
 switchExample();
 genericInferenceTest();
 }

 public static void instanceofExample() {
 System.out.println("=== instanceof example ===");
 var point = new Point(10, 10);
 var coloredPoint = new ColoredPoint(point, "blue");
 Object obj = coloredPoint;

 // JDK 16
 if (obj instanceof ColoredPoint cp) {
```

```
System.out.println("obj is a ColoredPoint: " + cp);
}

if (obj instanceof Point p) {
System.out.println("obj is a point: " + p);
} else {
System.out.println("obj is not a point");
}

// JDK 20
if (obj instanceof ColoredPoint(Point(int x, var y), String color)) {
System.out.printf("Point [%d,%d] has color %s%n", x, y, color);
}
}

public static void switchExample() {
System.out.println("=== switch example ===");
var list = new Decorator[] {
new Decorator(new Point(21, 21)),
new Decorator(new ColoredPoint(new Point(42, 42), "red")),
new Decorator("test")
};

for (Decorator d : list) {
switch (d) {
case Decorator(Point(var x, var y)) -> {
System.out.printf("Point [%d,%d]%n", x, y);
}
case Decorator(ColoredPoint(Point(var x, var y), String color)) -> {
System.out.printf("ColoredPoint [%d,%d] with color %s%n", x, y, color);
}
case Decorator(String s) -> {
System.out.println("Decorated string: " + s);
}
// required to be exhaustive
default -> System.out.println("None matched");
}
}
```

```
 }

 public static void genericInferenceTest() {
 System.out.println("=== generic type inference example ===");
 var point = new Point(42, 42);
 var decoratedPoint = new Decorator(new ColoredPoint(point, "RED"));
 var anotherDecorated = new Decorator(decoratedPoint);

 // JDK 20: type inference in Decorator<T>(Decorator<T>(T))
 if (anotherDecorated instanceof Decorator(Decorator(ColoredPoint(Point(int x, int
y), String color)))) {
 System.out.printf("x=%d, y=%d; color=%s%n%n", x, y, color);
 }
 }

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

 // support removed in JDK 21
 /*
 // only works in JDK 20
 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:

```
 === instanceof example ===

obj is a ColoredPoint: ColoredPoint[p=Point[x=10, y=10], color=blue]
obj is not a point
Point [10,10] has color blue
=== switch example ===
Point [21,21]
ColoredPoint [42,42] with color red
Decorated string: test
=== generic type inference example ===
x=42, y=42; color=RED
```

String Templates

Aim to simplify the process of string formatting and manipulation in Java. This feature enables developers to incorporate expressions directly within string literals, thus facilitating the creation and formatting of intricate strings

Code:

```
import static java.lang.StringTemplate.STR;

public class StringTemplate {
 public static void main(String[] args) {
 String name= "Alex";
 String message = STR."Greetings \{ name }!";
 System.out.println(message);
/*//concatenation
message = "Greetings " + name + "!";

//String.format()
message = String.format("Greetings %s!", name); //concatenation

//MessageFormat
message = new MessageFormat("Greetings {0}!").format(name);

//StringBuilder
message = new StringBuilder().append("Greetings ").append(name).append("!").toString();*/
 }
}
```

Output:

```
Greetings Alex!
```

Sequenced Collections

This enhancement aims to address common scenarios where accessing the first and last elements of various collection types in Java required non-uniform and sometimes cumbersome approaches

Code:

```
import java.util.*;

/**
 * Java Doc:
 * https://cr.openjdk.org/~smarks/collections/specdiff-sequenced-20230327/java.base/java/util/SequencedCollection.html
 * To run: `java SequencedCollectionExample.java`
 */
public class SequencedCollectionExample {
 public static void main(String[] args) {
 testCollections();
 testSets();
 testMaps();
 }

 public static void testCollections() {
 System.out.println("=== List ===");

 List<String> list = new ArrayList<>();
 SequencedCollection<String> sequenced = new ArrayList<>();

 // add operations (from Collection)
 list.add("elem");
 sequenced.add("elem");
 print(list, sequenced);

 // add first (head)
 list.set(0, "First"); // replace :s
 sequenced.addFirst("First"); // push
 print(list, sequenced);

 // add last (tail)
 list.add("last");
 sequenced.addLast("last");
 print(list, sequenced);

 // get first
 list.get(0);
 sequenced.getFirst();

 // get last
 list.get(list.size() - 1);
 sequenced.getLast();

 // remove first
 list.remove(0);
 sequenced.removeFirst();
 print(list, sequenced);

 // remove last
 list.remove(list.size() - 1);
 sequenced.removeLast();
 print(list, sequenced);

 // descending iterator
 ListIterator<String> reversed = list.listIterator(list.size());
 while (reversed.hasPrevious()) {
 reversed.previous();
 }
 Iterator<String> reversedSequence = sequenced.reversed().iterator();
 }
```

```
public static void testSets() {
System.out.println("=== Set ===");

Set<String> set = new LinkedHashSet<>();
SequencedSet<String> sequenced = new LinkedHashSet<>();
// SortedSet won't support `addFirst` and `addLast`
// SequencedSet<String> sortedSequenced = new SortedSet<>();

// add element (from Set)
set.add("elem");
sequenced.add("elem");
print(set, sequenced);

// get first
set.iterator().next();
sequenced.getFirst();

// get last
Iterator<String> iterator = set.iterator();
String last;
while (iterator.hasNext())
last = iterator.next();
last = sequenced.getLast();

// descending iterator
Iterator<String> descendingIterator = new
TreeSet<>(set).descendingSet().iterator();
descendingIterator = sequenced.reversed().iterator();
}

public static void testMaps() {
System.out.println("=== Map ===");
Map<String, String> map = new LinkedHashMap<>();
SequencedMap<String, String> sequenced = new LinkedHashMap<>();

map.put("elem", "v1");
sequenced.put("elem", "v1");
print(map, sequenced);
```

```
 // put first
 map.put("first", "v2");
 sequenced.putFirst("first", "v2");
 print(map, sequenced);

 // get first
 var entry = map.entrySet().iterator().next();
 entry = sequenced.firstEntry();

 // put last
 map.put("last", "v2");
 sequenced.putLast("last", "v2");
 print(map, sequenced);

 // get last
 var iterator = map.entrySet().iterator();
 while (iterator.hasNext())
 entry = iterator.next();
 entry = sequenced.lastEntry();

 // reverse
 var reversedIterator = sequenced.reversed().entrySet().iterator();
 }

 public static void print(Collection<?> collection, SequencedCollection<?>
newCollection) {
 System.out.printf("Collection: %s%n", collection);
 System.out.printf("Sequenced: %s%n", newCollection);
 }

 public static void print(Map<?,?> collection, SequencedMap<?,?> newCollection)
{
 System.out.printf("Map: %s%n", collection);
 System.out.printf("SequencedMap: %s%n", newCollection);
 }
}
```

Output:

```
=== List ===
Collection: [elem]
Sequenced: [elem]
Collection: [First]
Sequenced: [First, elem]
Collection: [First, last]
Sequenced: [First, elem, last]
Collection: [last]
Sequenced: [elem, last]
Collection: []
Sequenced: [elem]
=== Set ===
Collection: [elem]
Sequenced: [elem]
=== Map ===
Map: {elem=v1}
SequencedMap: {elem=v1}
Map: {elem=v1, first=v2}
SequencedMap: {first=v2, elem=v1}
Map: {elem=v1, first=v2, last=v2}
SequencedMap: {first=v2, elem=v1, last=v2}
```