

Pattern Matching

Small Enhancement or Major Feature?

Hanno Embregts
Peter Wessels

@hannotify
@PeterWessels



DEVOXX
BELGIUM



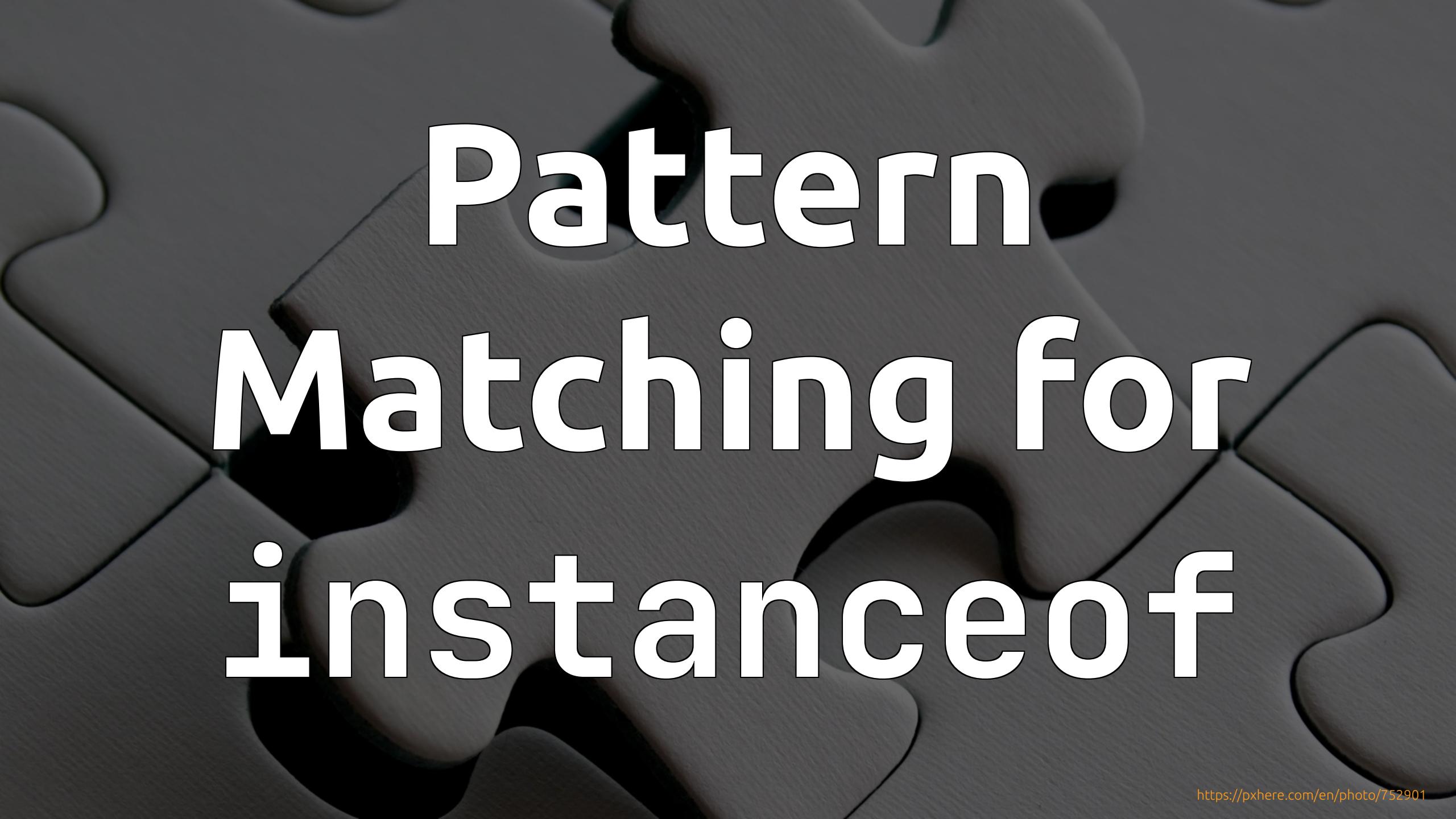


A close-up photograph of a man with short brown hair, wearing a dark suit jacket, a white shirt, and a dark tie. He is looking directly at the camera with a serious expression. His right hand is raised, with his index finger pointing directly at the viewer. The background is dark and out of focus.

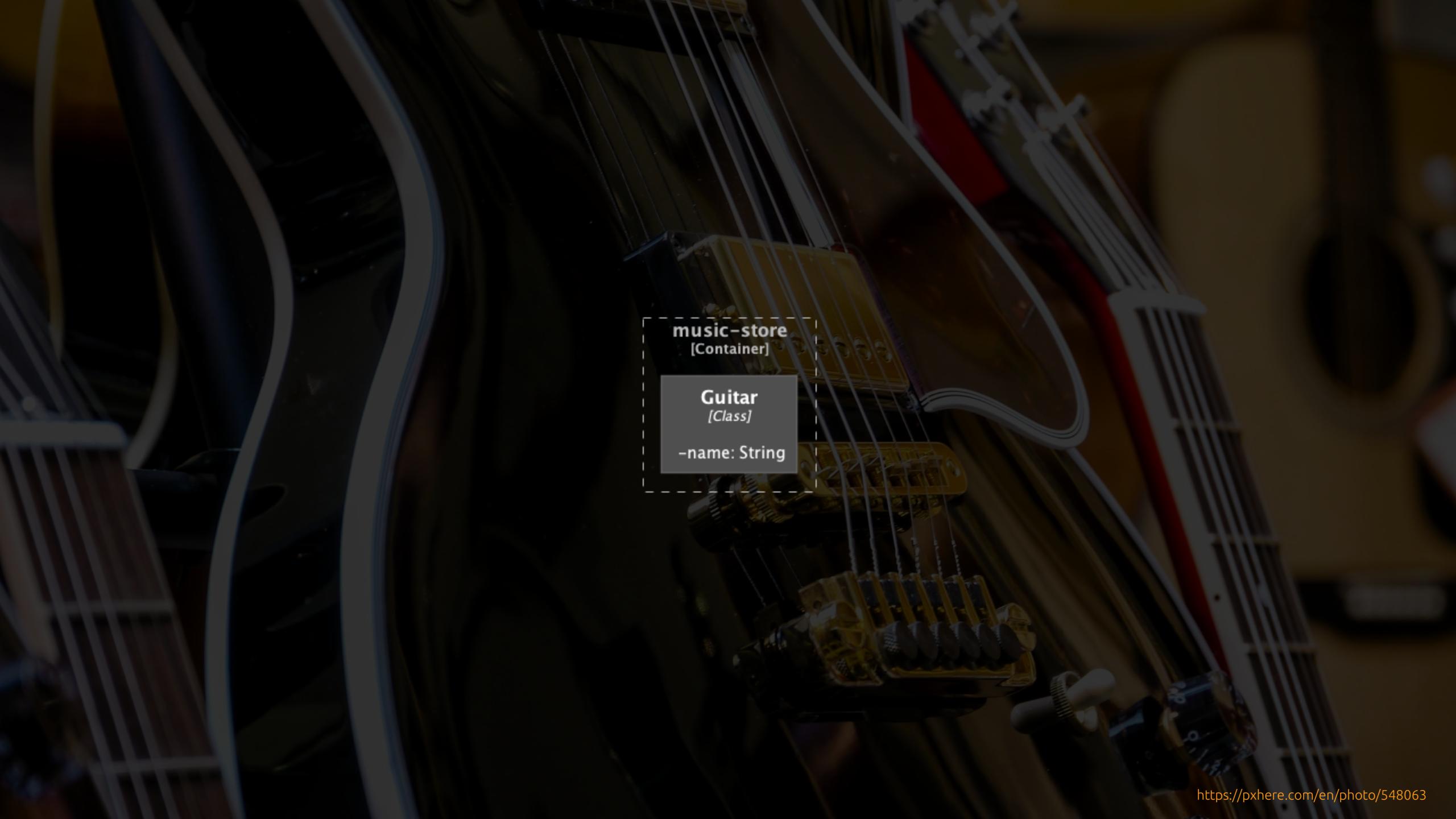
Small Enhancement

A photograph of a group of people in what appears to be a bar or social gathering. In the foreground, a woman with long dark hair, wearing a black top with a colorful floral pattern, is looking towards the camera while holding a drink. Behind her, a man with dark hair and a beard, wearing a dark t-shirt, is also looking towards the camera. In the background, there are other people, including a person in a yellow shirt. The lighting is warm and the overall atmosphere is casual.

Major Feature



Pattern
Matching for
instanceof

The background of the image is a dark, slightly blurred photograph of several guitars, including electric and acoustic models, displayed in a music store.

music-store
[Container]

Guitar
[Class]

-name: String

Instanceof-and-cast

```
1 if (product instanceof Guitar) {  
2     Guitar lesPaul =  
3         (Guitar) product;  
4     // use lesPaul  
5 }
```

Instanceof-and-cast

```
1 if (product instanceof Guitar) { // 1. is product a Guitar?  
2     Guitar lesPaul =  
3         (Guitar) product;  
4     // use lesPaul  
5 }
```

Instanceof-and-cast

```
1 if (product instanceof Guitar) { // 1. is product a Guitar?  
2     Guitar lesPaul =  
3         (Guitar) product; // 2. perform conversion  
4     // use lesPaul  
5 }
```

Instanceof-and-cast

```
1 if (product instanceof Guitar) { // 1. is product a Guitar?
2     Guitar lesPaul = // 3. declare variable, bind value
3             (Guitar) product; // 2. perform conversion
4     // use lesPaul
5 }
```

Improve the situation

```
1 if (product instanceof Guitar) { // 1. is product a Guitar?  
2     Guitar lesPaul = // 3. declare variable, bind value  
3             (Guitar) product; // 2. perform conversion  
4     // use lesPaul  
5 }
```

Improve the situation

```
1 if (product instanceof Guitar lesPaul) {  
2     // use lesPaul  
3 }
```

Type pattern

Consists of a predicate that specifies a type,
along with a single binding variable.

Pattern matching

Allows the conditional extraction of components from objects to be expressed more concisely and safely.

Demo

- Simplify implementation of equals

Declaring 'in the middle'

```
1 if (product instanceof Guitar lesPaul) {  
2     // use lesPaul  
3 }
```

Scoping

Pattern binding variable ('flow scoping')

- The set of places where it would definitely be assigned.

```
1 if (product instanceof Guitar lesPaul && lesPaul.isInTune()) {  
2     // can use lesPaul here  
3 } else {  
4     // can't use lesPaul here  
5 }
```

Benefits

- Nearly 100% of casts will just disappear!
- More concise

It's a kind of Pattern

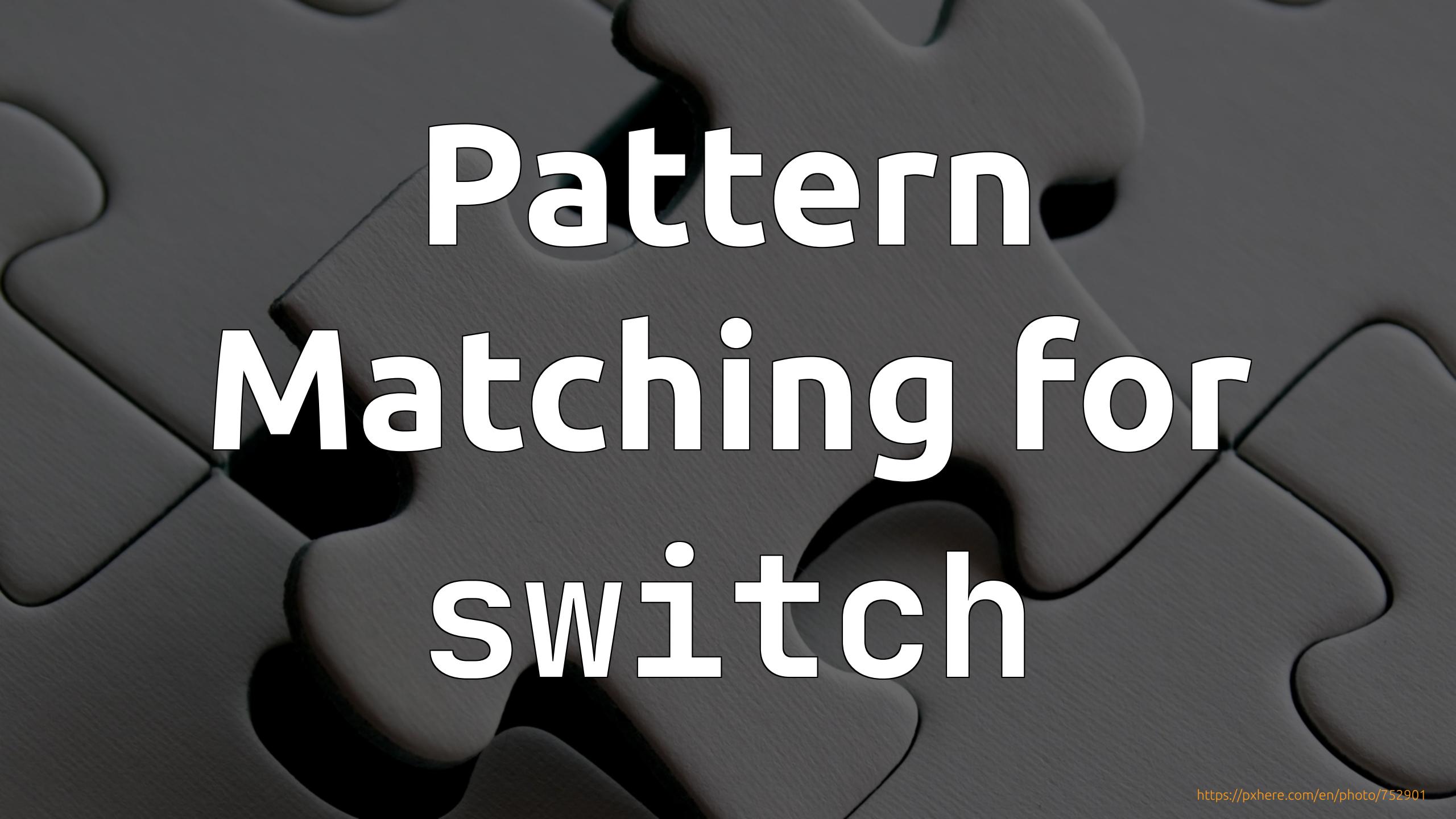
pattern

example

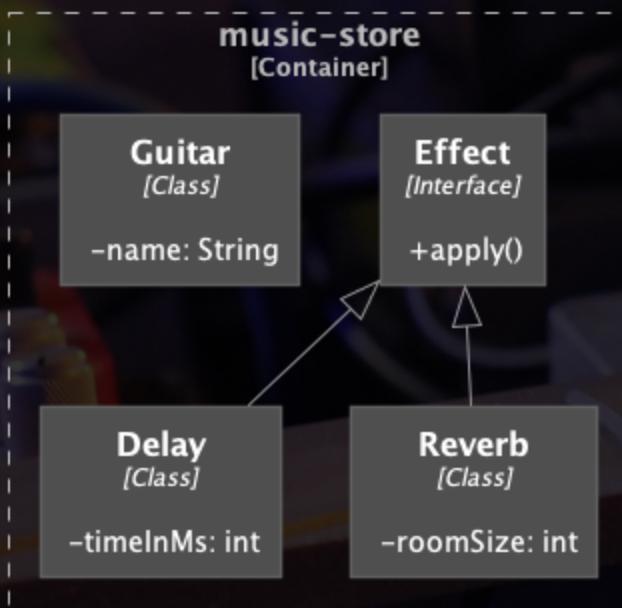
type pattern Guitar lesPaul

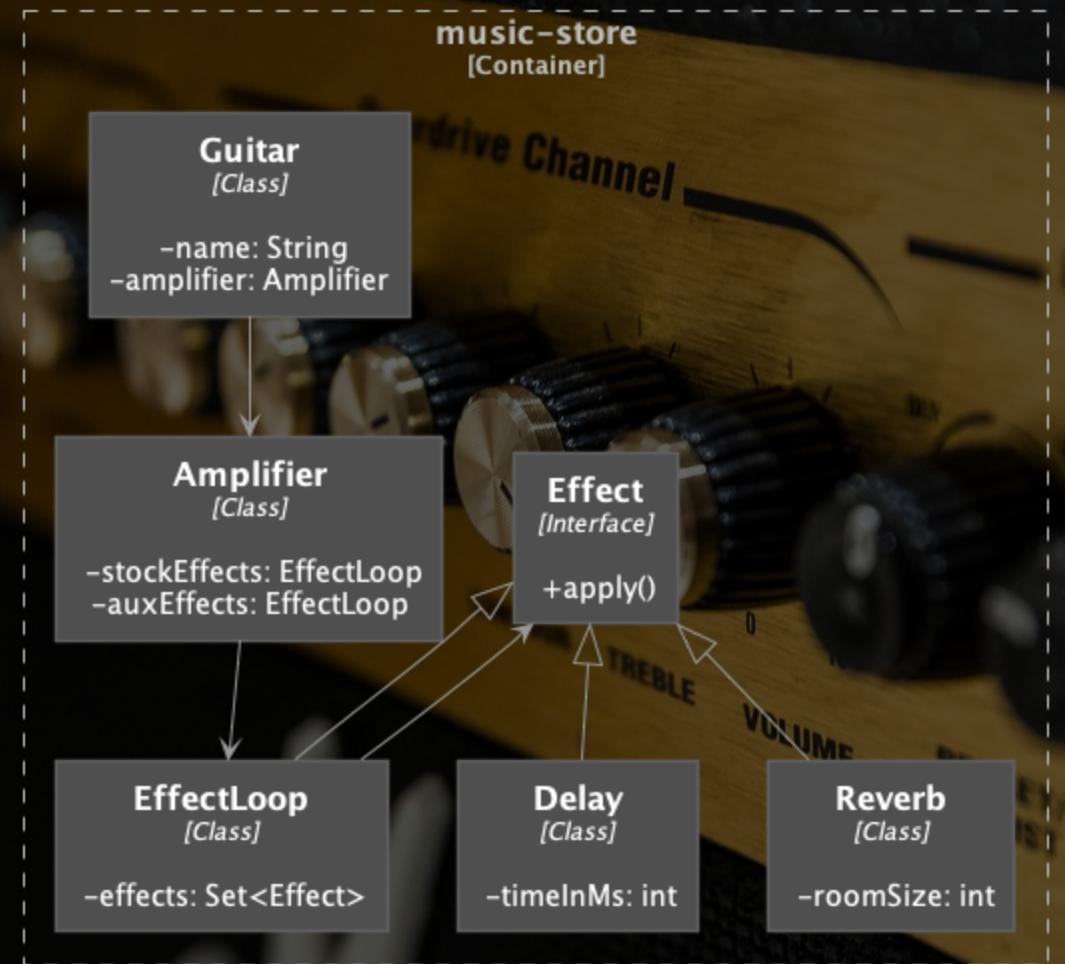
Feature Status

Java version	Feature status	JEP
14	Preview	JEP 305
15	Second preview	JEP 375
16	Final	JEP 394



Pattern Matching for switch







```
1 String apply(Effect effect) {  
2     String formatted = "";  
3     if (effect instanceof Delay) {  
4         Delay de = (Delay) effect;  
5         formatted = String.format("Delay active of %d ms.", de.getTimeInMs());  
6     } else if (effect instanceof Reverb) {  
7         Reverb re = (Reverb) effect;  
8         formatted = String.format("Reverb active of type %s and roomSize %d.", re.getName(), re.getRoomSize());  
9     } else if (effect instanceof Overdrive) {  
10        Overdrive ov = (Overdrive) effect;  
11        formatted = String.format("Overdrive active with gain %d.", ov.getGain());  
12    } else if (effect instanceof Tremolo) {  
13        Tremolo tr = (Tremolo) effect;  
14        formatted = String.format("Tremolo active with depth %d and rate %d.", tr.getDepth(), tr.getRate());  
15    } else if (effect instanceof Tuner) {  
16        Tuner tu = (Tuner) effect;  
17        formatted = String.format("Tuner active with pitch %f and volume %f.", tu.getPitch(), tu.getVolume());  
18    }  
19    return formatted;  
20}
```

```
9 } else if (effect instanceof Overdrive) {  
10     Overdrive ov = (Overdrive) effect;  
11     formatted = String.format("Overdrive active with gain %d.", ov.getGain());  
12 } else if (effect instanceof Tremolo) {  
13     Tremolo tr = (Tremolo) effect;  
14     formatted = String.format("Tremolo active with depth %d and rate %d.", tr.getDepth(), tr.getRate());  
15 } else if (effect instanceof Tuner) {  
16     Tuner tu = (Tuner) effect;  
17     formatted = String.format("Tuner active with pitch %d. Muting all signal!", tu.getPitch());  
18 } else if (effect instanceof EffectLoop) {  
19     EffectLoop el = (EffectLoop) effect;  
20     formatted = el.getEffects().stream().map(this::apply).collect(Collectors.joining(", "));  
21 } else {  
22     formatted = String.format("Unknown effect active: %s.", effect);  
23 }
```

```
1 String apply(Effect effect) {  
2     String formatted = "";  
3     if (effect instanceof Delay de) {  
4         formatted = String.format("Delay active of %d ms.", de.getTimeInMs());  
5     } else if (effect instanceof Reverb re) {  
6         formatted = String.format("Reverb active of type %s and roomSize %d.", re.getName(),  
7             re.getRoomSize());  
7     } else if (effect instanceof Overdrive ov) {  
8         formatted = String.format("Overdrive active with gain %d.", ov.getGain());  
9     } else if (effect instanceof Tremolo tr) {  
10        formatted = String.format("Tremolo active with depth %d and rate %d.", tr.getDepth(),  
11            tr.getRate());  
11    } else if (effect instanceof Tuner tu) {  
12        formatted = String.format("Tuner active with pitch %d. Muting all signal!", tu.getPitch());  
13    } else if (effect instanceof EffectLoop el) {  
14        formatted = el.getEffects().stream().map(this::apply).collect(Collectors.joining(", "));  
15    } else {  
16        formatted = String.format("Unknown effect active: %s " + effect);  
17    }  
18    return formatted;  
19}
```

```
4     formatted = String.format("Delay active on note %s. , ue.getRoomSize()),  
5 } else if (effect instanceof Reverb re) {  
6     formatted = String.format("Reverb active of type %s and roomSize %d.", re.getName(),  
7 } else if (effect instanceof Overdrive ov) {  
8     formatted = String.format("Overdrive active with gain %d.", ov.getGain());  
9 } else if (effect instanceof Tremolo tr) {  
10    formatted = String.format("Tremolo active with depth %d and rate %d.", tr.getDepth(),  
11 } else if (effect instanceof Tuner tu) {  
12    formatted = String.format("Tuner active with pitch %d. Muting all signal!", tu.getPitch());  
13 } else if (effect instanceof EffectLoop el) {  
14    formatted = el.getEffects().stream().map(this::apply).collect(Collectors.joining(", "));  
15 } else {  
16     formatted = String.format("Unknown effect active: %s.", effect);  
17 }  
18 return formatted;  
19 }
```

Switch expression

```
1 String apply(Effect effect) {  
2     return switch(effect) {  
3         default          → String.format("Unknown effect active: %s.", effect);  
4     };  
5 }
```

Switch expression

```
1 String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay de      → String.format("Delay active of %d ms.", de.getTimeInMs());  
4         case Reverb re    → String.format("Reverb active of type %s and roomSize %d.", re  
5         default           → String.format("Unknown effect active: %s.", effect);  
6     };  
7 }
```

Switch expression

```
1 String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay de      → String.format("Delay active of %d ms.", de.getTimeInMs());  
4         case Reverb re    → String.format("Reverb active of type %s and roomSize %d.", r  
5         case Overdrive ov → String.format("Overdrive active with gain %d.", ov.getGain()  
6         case Tremolo tr   → String.format("Tremolo active with depth %d and rate %d.", t  
7         case Tuner tu     → String.format("Tuner active with pitch %d. Muting all signal  
8             default       → String.format("Unknown effect active: %s.", effect);  
9     };  
10 }
```

Switch expression

```
1 String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay de      → String.format("Delay active of %d ms.", de.getTimeInMs());  
4         case Reverb re    → String.format("Reverb active of type %s and roomSize %d.", r  
5         case Overdrive ov → String.format("Overdrive active with gain %d.", ov.getGain()  
6         case Tremolo tr   → String.format("Tremolo active with depth %d and rate %d.", t  
7         case Tuner tu     → String.format("Tuner active with pitch %d. Muting all signal  
8         case EffectLoop el → el.getEffects().stream().map(this::apply).collect(Collectors  
9             default           → String.format("Unknown effect active: %s.", effect);  
10    };  
11 }
```

Switch expression

```
1 String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay de      → String.format("Delay active of %d ms.", de.getTimeInMs());  
4         case Reverb re    → String.format("Reverb active of type %s and roomSize %d.", r.  
5         case Overdrive ov → String.format("Overdrive active with gain %d.", ov.getGain()  
6         case Tremolo tr   → String.format("Tremolo active with depth %d and rate %d.", t.  
7         case Tuner tu     → String.format("Tuner active with pitch %d. Muting all signal  
8         case EffectLoop el → el.getEffects().stream().map(this::apply).collect(Collectors.  
9             default           → String.format("Unknown effect active: %s.", effect);  
10    };  
11 }
```



Sensible operations to the effect loop

- apply()
- setVolume(int volume)
- contains(Effect ... effect)

Nonsensical operations to the effect loop

- `isTunerActive()`
- `isDelayTimeEqualToReverbRoomSize()`
- `isToneSuitableToPlayPrideInTheNameOfLove()`

Switch expression

```
1 String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay de      → String.format("Delay active of %d ms.", de.getTimeInMs());  
4         case Reverb re    → String.format("Reverb active of type %s and roomSize %d.", r.  
5         case Overdrive ov → String.format("Overdrive active with gain %d.", ov.getGain()  
6         case Tremolo tr   → String.format("Tremolo active with depth %d and rate %d.", t.  
7         case Tuner tu     → String.format("Tuner active with pitch %d. Muting all signal  
8         case EffectLoop el → el.getEffects().stream().map(this::apply).collect(Collectors.  
9             default           → String.format("Unknown effect active: %s.", effect);  
10    };  
11 }
```

Switch expression

```
1 static String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay de      → String.format("Delay active of %d ms.", de.getTimeInMs());  
4         case Reverb re    → String.format("Reverb active of type %s and roomSize %d.", r.  
5         case Overdrive ov → String.format("Overdrive active with gain %d.", ov.getGain());  
6         case Tremolo tr   → String.format("Tremolo active with depth %d and rate %d.", t.  
7         case Tuner tu     → String.format("Tuner active with pitch %d. Muting all signal  
8         case EffectLoop el → el.getEffects().stream().map(Effect::apply).collect(Collectors.  
9             default           → String.format("Unknown effect active: %s.", effect);  
10    };  
11 }
```

Benefits of pattern matching

- No need for the Visitor pattern or a common supertype
- A single expression instead of many assignments
- Less error-prone (in adding cases)
- More concise
- Safer - the compiler can check for missing cases

But what if effect is null?

```
1 static String apply(Effect effect) {  
2     return switch(effect) { // throws NullPointerException!  
3         case Delay de      → String.format("Delay active of %d ms.", de.getTimeInMs());  
4         case Reverb re    → String.format("Reverb active of type %s and roomSize %d.", r  
5         case Overdrive ov → String.format("Overdrive active with gain %d.", ov.getGain());  
6         case Tremolo tr   → String.format("Tremolo active with depth %d and rate %d.", t  
7         case Tuner tu     → String.format("Tuner active with pitch %d. Muting all signal  
8         case EffectLoop el → el.getEffects().stream().map(Effect::apply).collect(Collectors.  
9             default           → String.format("Unknown effect active: %s.", effect);  
10    };  
11 }
```

Combining case labels

```
1 static String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay de      → String.format("Delay active of %d ms.", de.getTimeInMs());  
4         case Reverb re    → String.format("Reverb active of type %s and roomSize %d.", r.  
5         case Overdrive ov → String.format("Overdrive active with gain %d.", ov.getGain());  
6         case Tremolo tr   → String.format("Tremolo active with depth %d and rate %d.", t.  
7         case Tuner tu     → String.format("Tuner active with pitch %d. Muting all signal  
8         case EffectLoop el → el.getEffects().stream().map(Effect::apply).collect(Collectors.  
9         case null, default → String.format("Unknown or malfunctioning effect active: %s.");  
10    };  
11 }
```

Demo

- Guarded patterns

Guarded patterns

```
1 String apply(Effect effect, Guitar guitar) {  
2     return switch(effect) {  
3         // (...)  
4         case Tremolo tr → String.format("Tremolo active with depth %d and rate %d.", tr.ge  
5         case Tuner tu → String.format("Tuner active with pitch %d. Muting all signal!", tu  
6         case EffectLoop el → el.getEffects().stream().map(this::apply).collect(Collectors.  
7         default → String.format("Unknown effect active: %s.", effect);  
8     };  
9 }
```

Guarded patterns

```
1 String apply(Effect effect, Guitar guitar) {  
2     return switch(effect) {  
3         // (...)  
4         case Tremolo tr → String.format("Tremolo active with depth %d and rate %d.", tr.ge  
5         case Tuner tu && !tu.isInTune(guitar) → String.format("Guitar is in need of tuning.  
6         case EffectLoop el → el.getEffects().stream().map(this::apply).collect(Collectors.  
7         default → String.format("Unknown effect active: %s.", effect);  
8     };  
9 }
```

Guarded patterns When clauses

```
1 String apply(Effect effect, Guitar guitar) {  
2     return switch(effect) {  
3         // (...)  
4         case Tremolo tr → String.format("Tremolo active with depth %d and rate %d.", tr.ge  
5         case Tuner tu when !tu.isInTune(guitar) → String.format("Guitar is in need of tun  
6         case EffectLoop el → el.getEffects().stream().map(this::apply).collect(Collectors.  
7         default → String.format("Unknown effect active: %s.", effect);  
8     };  
9 }
```

when clauses

```
1 switch(effect) {  
2     // ...  
3     case Tuner tu:  
4         if (!tu.isInTune(guitar)) { // tuning is needed }  
5         else { // no tuning is needed }  
6         break;  
7     // ...  
8 }
```

It's a kind of Pattern



pattern

example

type pattern

Guitar lesPaul

guarded pattern

Tuner tu &&
!tu.isInTune(guitar)

It's a kind of Pattern

pattern

type pattern

guarded pattern

when clause

example

Guitar lesPaul

Tuner tu when
!tu.isInTune(guitar)

Feature Status

Java version	Feature status	JEP
17	Preview	JEP 406
18	Second preview	JEP 420
19	Third preview	JEP 427

Deconstruction Patterns

Disclaimer

We can't tell you when the following features
are coming to Java. Also: syntax and
implementation specifics may still change.

Deconstruction patterns

```
1 String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay de      → String.format("Delay active of %d ms.", de.getTimeInMs());  
4         case Reverb re    → String.format("Reverb active of type %s and roomSize %d.", r  
5         case Overdrive ov → String.format("Overdrive active with gain %d.", ov.getGain()  
6         case Tremolo tr   → String.format("Tremolo active with depth %d and rate %d.", t  
7         case Tuner tu     → String.format("Tuner active with pitch %d. Muting all signal  
8         case EffectLoop el → el.getEffects().stream().map(this::apply).collect(Collectors  
9             default           → String.format("Unknown effect active: %s.", effect);  
10    };  
11 }
```

Deconstruction patterns

```
1 String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay de      → String.format("Delay active of %d ms.", de.getTimeInMs());  
4         case Reverb re    → String.format("Reverb active of type %s and roomSize %d.", r  
5         case Overdrive(int gain) → String.format("Overdrive active with gain %d.", gain);  
6         case Tremolo tr   → String.format("Tremolo active with depth %d and rate %d.", t  
7         case Tuner tu     → String.format("Tuner active with pitch %d. Muting all signal  
8         case EffectLoop el → el.getEffects().stream().map(this::apply).collect(Collectors.  
9             default           → String.format("Unknown effect active: %s.", effect);  
10    };  
11 }
```

Pattern definition

```
1 public class Overdrive implements Effect {  
2     private final int gain;  
3  
4     public Overdrive(int gain) {  
5         this.gain = gain;  
6     }  
7 }
```

Pattern definition

```
1 public class Overdrive implements Effect {  
2     private final int gain;  
3  
4     public Overdrive(int gain) {  
5         this.gain = gain;  
6     }  
7  
8     public pattern Overdrive(int gain) {  
9         gain = this.gain;  
10    }  
11 }
```

Deconstruction patterns

```
1 String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay de      → String.format("Delay active of %d ms.", de.getTimeInMs());  
4         case Reverb re    → String.format("Reverb active of type %s and roomSize %d.", r  
5         case Overdrive(int gain) → String.format("Overdrive active with gain %d.", gain);  
6         case Tremolo tr   → String.format("Tremolo active with depth %d and rate %d.", t  
7         case Tuner tu     → String.format("Tuner active with pitch %d. Muting all signal  
8         case EffectLoop el → el.getEffects().stream().map(this::apply).collect(Collectors.  
9             default           → String.format("Unknown effect active: %s.", effect);  
10    };  
11 }
```

Deconstruction patterns

```
1 String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay(int timeInMs) → String.format("Delay active of %d ms.", timeInMs);  
4         case Reverb(String name, int roomSize) → String.format("Reverb active of type %s  
5             with room size %d.", name, roomSize);  
6         case Overdrive(int gain) → String.format("Overdrive active with gain %d.", gain);  
7         case Tremolo(int depth, int rate) → String.format("Tremolo active with depth %d and  
8             rate %d.", depth, rate);  
9         case Tuner(int pitchInHz) → String.format("Tuner active with pitch %d. Muting all  
10            notes below this frequency.", pitchInHz);  
11        case EffectLoop(Set<Effect> effects) → effects.stream().map(this::apply).collect(  
12            Collector.of(() → new StringBuilder(),  
13                (sb, effect) → sb.append(effect).append(" "),  
14                (sb1, sb2) → sb1.append(sb2)),  
15            () → sb.toString());  
16        default → String.format("Unknown effect active: %s.", effect);  
17    };  
18 }
```

Pattern composition

```
1 static boolean isDelayTimeEqualToReverbRoomSize(EffectLoop effectLoop) {  
2  
3 }
```

Pattern composition

```
1 static boolean isDelayTimeEqualToReverbRoomSize(EffectLoop effectLoop) {  
2     if (effectLoop instanceof EffectLoop(Delay(int timeInMs), Reverb(String name, int roomSize))){  
3         return timeInMs == roomSize;  
4     }  
5     return false;  
6 }
```

Pattern composition

```
1 static boolean isDelayTimeEqualToReverbRoomSize(EffectLoop effectLoop) {  
2     return effectLoop.getEffects().stream()  
3         .filter(e → e instanceof Delay || e instanceof Reverb)  
4         .map(dr → {  
5             if (dr instanceof Delay d) {  
6                 return d.getTimeInMs();  
7             } else {  
8                 Reverb r = (Reverb) dr;  
9                 return r.getRoomSize();  
10            }  
11        })  
12        .distinct().count() = 1;  
13 }
```

It's a kind of Pattern

pattern

type pattern

guarded pattern

when clause

deconstruction pattern

example

Guitar lesPaul

Tuner tu when
!tu.isInTune(guitar)

Delay(int timeInMs)

Var and any patterns

```
1 // Pre-Java 10
2 Guitar telecaster = new Guitar("Fender Telecaster Baritone Blacktop", GuitarType.TELECASTER);
3
4 // Java 10
5 var telecaster = new Guitar("Fender Telecaster Baritone Blacktop", GuitarType.TELECASTER);
```

<https://openjdk.java.net/jeps/286>

Var and any patterns

```
1 static boolean isDelayTimeEqualToReverbRoomSize(EffectLoop effectLoop) {  
2     if (effectLoop instanceof EffectLoop(Delay(int timeInMs), Reverb(String name, int roomSize))){  
3         return timeInMs == roomSize;  
4     }  
5     return false;  
6 }
```

Var and any patterns

```
1 static boolean isDelayTimeEqualToReverbRoomSize(EffectLoop effectLoop) {  
2     if (effectLoop instanceof EffectLoop(Delay(var timeInMs), Reverb(var name, var roomSize))){  
3         return timeInMs == roomSize;  
4     }  
5     return false;  
6 }
```



"To start, press any key."
Where's the "any" key?

Var and any patterns

```
1 static boolean isDelayTimeEqualToReverbRoomSize(EffectLoop effectLoop) {  
2     if (effectLoop instanceof EffectLoop(Delay(var timeInMs), Reverb(_, var roomSize))) {  
3         return timeInMs == roomSize;  
4     }  
5     return false;  
6 }
```

Optimization

```
1 static String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay(int timeInMs) → String.format("Delay active of %d ms.", timeInMs);  
4         case Reverb(String name, int roomSize) → String.format("Reverb active of type %s  
5             with room size %d.", name, roomSize);  
6         case Overdrive(int gain) → String.format("Overdrive active with gain %d.", gain);  
7         case Tremolo(int depth, int rate) → String.format("Tremolo active with depth %d and  
8             rate %d.", depth, rate);  
9         case Tuner(int pitchInHz) → String.format("Tuner active with pitch %d. Muting all  
10            notes below this frequency.", pitchInHz);  
11         case EffectLoop(var effects) → effects.stream().map(Effect::apply).collect(Collectors.  
12             toList());  
13         default → String.format("Unknown effect active: %s.", effect);  
14     };  
15 }
```

Optimization

```
1 static String apply(Effect effect) {  
2     return switch(effect) {  
3         // ...  
4         case EffectLoop(var effects) → effects.stream().map(Effect::apply).collect(Collectors.toList());  
5         default → String.format("Unknown effect active: %s.", effect);  
6     };  
7 }
```

Optimization

```
1 static String apply(Effect effect) {  
2     return switch(effect) {  
3         // ...  
4         case EffectLoop(Tuner(int pitchInHz), _) → String.format("The EffectLoop contains  
5             case EffectLoop(var effects) → effects.stream().map(Effect::apply).collect(Collectors.  
6                 default → String.format("Unknown effect active: %s.", effect);  
7             };  
8 }
```

Benefits

- Better encapsulation
a case branch only receives data that it actually references.
- More elegant logic
by using pattern composition
- Optimization
through the use of any patterns

It's a kind of Pattern

pattern

type pattern

~~guarded pattern~~

when clause

deconstruction pattern

var pattern

example

Guitar lesPaul

Tuner tu when
!tu.isInTune(guitar)

Delay(int timeInMs)

var timeInMs

It's a kind of Pattern

pattern

type pattern

~~guarded pattern~~

when clause

deconstruction pattern

var pattern

any pattern

example

Guitar lesPaul

Tuner tu when
!tu.isInTune(guitar)

Delay(int timeInMs)

var timeInMs

-

Feature Status

Java version	Feature status	JEP
n/a	Exploratory document	Pattern Matching for Java

<https://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html>



Pattern Matching Plays Nice
with
Sealed Types
and Records

Demo

- Make Effect a sealed type

Sealed types yield completeness

```
1 static String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay(int timeInMs) → String.format("Delay active of %d ms.", timeInMs);  
4         case Reverb(String name, int roomSize) → String.format("Reverb active of type %s  
5             with room size %d.", name, roomSize);  
6         case Overdrive(int gain) → String.format("Overdrive active with gain %d.", gain);  
7         case Tremolo(int depth, int rate) → String.format("Tremolo active with depth %d and  
8             rate %d.", depth, rate);  
9         case Tuner(int pitchInHz) → String.format("Tuner active with pitch %d. Muting all  
10            notes below %d Hz.", pitchInHz, pitchInHz / 2);  
11         case EffectLoop(Tuner(int pitchInHz), _) → String.format("The EffectLoop contains  
12             a tuner with pitch %d Hz.", pitchInHz);  
13         case EffectLoop(var effects) → effects.stream().map(Effect::apply).collect(Collectors.  
14             toList());  
15         default → String.format("Unknown effect active: %s.", effect);  
16     };  
17 }
```

Sealed types yield completeness

```
1 static String apply(Effect effect) {  
2     return switch(effect) {  
3         case Delay(int timeInMs) → String.format("Delay active of %d ms.", timeInMs);  
4         case Reverb(String name, int roomSize) → String.format("Reverb active of type %s",  
5             name);  
6         case Overdrive(int gain) → String.format("Overdrive active with gain %d.", gain);  
7         case Tremolo(int depth, int rate) → String.format("Tremolo active with depth %d and  
8             rate %d.", depth, rate);  
9         case Tuner(int pitchInHz) → String.format("Tuner active with pitch %d. Muting all  
10            notes below %d Hz.", pitchInHz);  
11         case EffectLoop(Tuner(int pitchInHz), _) → String.format("The EffectLoop contains  
12            a tuner with pitch %d Hz.", pitchInHz);  
13         case EffectLoop(var effects) → effects.stream().map(Effect::apply).collect(Collectors.  
14             toList());  
15     };  
16 }
```

<https://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html>

Records

Input:

- Commit to the class being a transparent carrier for its data.

Output:

- constructors
- accessor methods
- `equals()`-implementation
- `hashCode()`-implementation
- `toString()`-implementation
- deconstruction pattern

Array patterns

```
1 record EffectLoop(String name, int volume, Effect... effects) { }
```

```
1 static String apply(EffectLoop effectLoop) {}  
2     return switch(effectLoop) {  
3         case EffectLoop(var name, var volume) → "Effect loop contains no effects."  
4     }  
5 }
```

Array patterns

```
1 record EffectLoop(String name, int volume, Effect... effects) { }
```

```
1 static String apply(EffectLoop effectLoop) {}  
2     return switch(effectLoop) {  
3         case EffectLoop(var name, var volume) → "Effect loop contains no effects."  
4         case EffectLoop(_, _, var effect) → "Effect loop contains exactly one effect."  
5     }  
6 }
```

Array patterns

```
1 record EffectLoop(String name, int volume, Effect... effects) { }
```

```
1 static String apply(EffectLoop effectLoop) {}  
2     return switch(effectLoop) {  
3         case EffectLoop(var name, var volume) → "Effect loop contains no effects."  
4         case EffectLoop(_, _, var effect) → "Effect loop contains exactly one effect."  
5         case EffectLoop(_, _, var effect, ...) → "Effect loop contains more than one effect."  
6     }  
7 }
```

Array patterns

```
1 record EffectLoop(String name, int volume, Effect... effects) { }
```

```
1 static String apply(EffectLoop effectLoop) {}  
2     return switch(effectLoop) {  
3         case EffectLoop(var name, var volume) → "Effect loop contains no effects."  
4         case EffectLoop(_, _, var effect) → "Effect loop contains exactly one effect."  
5         case EffectLoop(_, _, var effect, ...) → "Effect loop contains more than one effect."  
6         case EffectLoop(_, _, var effect1, var effect2) → "Effect loop contains exactly two effects."  
7         case EffectLoop(_, _, var effect1, var effect2, ...) → "Effect loop contains more than two effects."  
8     }  
9 }
```

It's a kind of Pattern

pattern example

..

var pattern

any pattern

array pattern

var timeInMs

-

EffectLoop(var name, var effect, ...)

Feature Status

Sealed Types

Java version	Feature status	JEP
15	Preview	JEP 360
16	Second preview	JEP 397
17	Final	JEP 409

Feature Status

Completeness

Java version	Feature status	JEP
17	Preview	JEP 406
18	Second preview	JEP 420
19	Third preview	JEP 427

Feature Status

Record Patterns

Java version	Feature status	JEP
n/a	Preview	JEP 405

A Better Serialization?

Here be dragons!

We can't be sure **at all** that the following features will appear in Java as depicted. They can change a **lot** in the meantime.

Opposites

Deconstruction pattern

- transforms an object into a set of typed fields

Constructor

- transforms a set of typed fields into an object

Serialization

- very important feature
- but many people hate its current implementation

Drawbacks

- it undermines the accessibility model
- serialization logic is not 'readable code'
- it bypasses constructors and data validation

Serialization

```
1 public class EffectLoop implements Effect {  
2     private String name;  
3     private Set<Effect> effects;  
4  
5     public EffectLoop(String name) {  
6         this.name = name;  
7         this.effects = new HashSet<>();  
8     }  
9 }
```

Serialization

```
1 public class EffectLoop implements Effect {  
2     private String name;  
3     private Set<Effect> effects;  
4  
5     public EffectLoop(String name) {  
6         this.name = name;  
7         this.effects = new HashSet<>();  
8     }  
9  
10    public pattern EffectLoop(String name, Effect[] effects) {  
11        name = this.name;  
12        effects = this.effects.toArray();  
13    }  
14 }
```

Serialization

```
6     this.name = name;
7     this.effects = new HashSet<Effect>();
8 }
9
10    public EffectLoop(String name, Effect[] effects) {
11        this(name);
12        for (Effect effect : effects) {
13            this.effects.add(effect);
14        }
15    }
16
17    public pattern EffectLoop(String name, Effect[] effects) {
18        name = this.name;
19        effects = this.effects.toArray();
20    }
21 }
```

Serialization

```
1 public class EffectLoop implements Effect {  
2     private String name;  
3     private Set<Effect> effects;  
4  
5     public EffectLoop(String name) {  
6         this.name = name;  
7         this.effects = new HashSet<>();  
8     }  
9  
10    @Deserializer  
11    public EffectLoop(String name, Effect[] effects) {  
12        this(name);  
13        for (Effect effect : effects) {  
14            this.effects.add(effect);  
15        }  
16    }
```

Some challenges remain

Q: How to support multiple versions of one class?

A: `@Serializer` and `@Deserializer` annotations could get a `property version` in the future.

Feature Status

Java
version

n/a

Feature status

Exploratory
document

JEP

Towards Better
Serialization

<https://cr.openjdk.java.net/~briangoetz/amber/serialization.html>

Future Expansions

Here be super dragons!

We can't be sure that the following features will appear in Java as depicted, **if at all**.
Proceed with caution!

Pattern bind statements

```
1 var reverb = new Reverb("ChamberReverb", 2);
2
3 __let Reverb(String name, int roomSize) = reverb;
4
5 // do something with name & roomSize
```

<https://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html>

Pattern bind statements

```
1 var reverb = new Reverb("ChamberReverb", 2);
2
3 let Reverb(String name, int roomSize) = reverb;
4 else throw new IllegalArgumentException("not a Reverb!");
5
6 // do something with name & roomSize
```

Other ideas

- AND patterns:
`PatternOne&PatternTwo`
- Patterns in **catch** clauses:
(will most likely complement multi-catch blocks)
- Collection patterns

<https://mail.openjdk.java.net/pipermail/amber-spec-experts/2021-January/002758.html>

Pattern Contexts

Pattern Contexts

Pattern context	Example	Purpose
<i>instanceof predicate</i>	product instanceof Guitar guitar	Test if target matches the indicated pattern.
<i>switch statement or expression</i>	switch (effect) { case Delay d → }	Test if target matches one (or more) of the indicated patterns.
<i>bind statement</i>	_let Reverb(var name, var roomSize) = reverb;	Destructure a target using a pattern.



Wrap-up

Pattern matching...

- is a rich feature arc that will play out over several versions.
- allows us to use type patterns in instanceof.
- improves switch expressions.
- makes destructuring objects as easy as (and more similar to) constructing them.
- holds the potential to simplify and streamline much of the code we write today.

A photograph of a group of people sitting in a theater, watching a movie. In the foreground, a woman with long dark hair, wearing a striped shirt, looks towards the screen. Behind her, a man with curly hair, wearing a black t-shirt, also looks towards the screen. To the right, another person's shoulder and back are visible, wearing a yellow shirt. The theater has red seats and a dark interior.

Major Feature

Thank you! 😊

github.com/hannotify/pattern-matching-music-store

hanno.codes peterwessels.nl

[@hannotify](https://twitter.com/hannotify) [@PeterWessels](https://twitter.com/PeterWessels)