New Features

What to expect

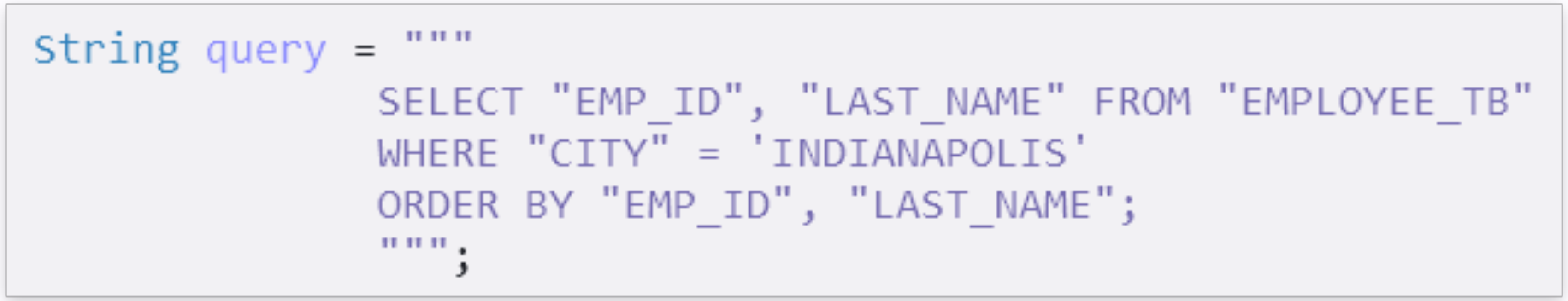
- new Java features
  - with use cases for regular developers
  - focus on refactoring
  - hands on showcases
- Text Blocks, Records, Pattern Matching, Switch Expressions, ...

Text Blocks: JEP 378 - History

- succeeding Raw String Literals: JEP 326
  - intended for JDK 12, but withdrawn
- Preview: JDK 13, 14
- Release: JDK 15
- two new escape sequences added in JDK 14

Text Blocks: JEP 378 - Summary

- making it easy to express strings that span several lines of source code
- enhance the readability of strings
- Example:



```
String query = """
               SELECT "EMP_ID", "LAST_NAME" FROM "EMPLOYEE_TB"
               WHERE "CITY" = 'INDIANAPOLIS'
               ORDER BY "EMP_ID", "LAST_NAME";
               """;
```

- Showcase: TextBlocks.java

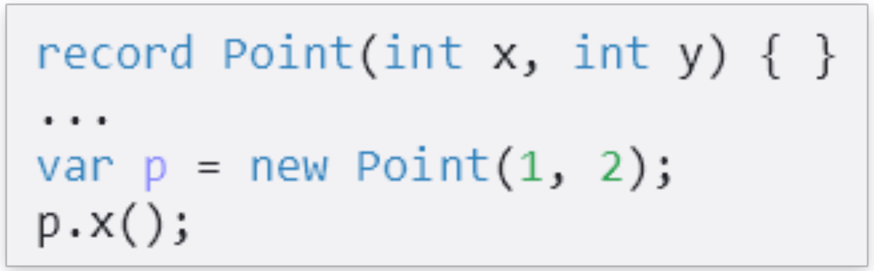
Records: JEP 395 - History

- Preview: JDK 14, 15
- Release: JDK 16

Records: JEP 395 - Summary

- transparent carriers for immutable data
- object-oriented construct that expresses a simple aggregation of values
- focus on modeling immutable data rather than extensible behavior
- automatically implement data-driven methods such as equals and accessors
- help to model simple data aggregates with less code

Records: JEP 395 - Example



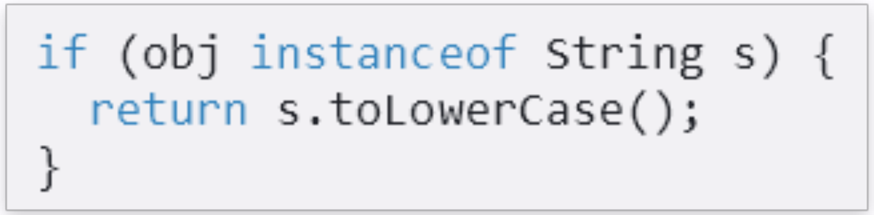
```java
record Point(int x, int y) { }
...
var p = new Point(1, 2);
p.x();
```

- Showcase: Records.java

PatternMatching for instanceof: JEP 394

- History:
  - Preview: JDK 14, 15
  - Release: JDK 16
- Summary:
  - allows conditional extraction of components from objects
  - expressed more concisely and safely
  - reduce boilerplate and unnecessary repetition

Pattern Matching for instanceof: JEP 394 - Example



```java
if (obj instanceof String s) {
  return s.toLowerCase();
}
```

- Showcase: PatternMatchingInstanceOf.java

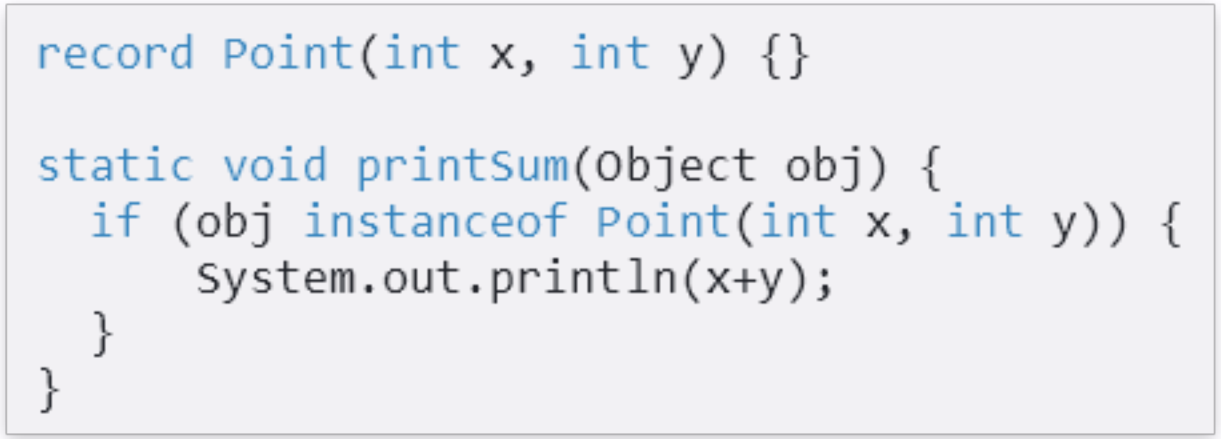
Record Patterns: JEP 440 - History

- Preview: JDK 19, 20
- Release: JDK 21
- co-evolved with Pattern Matching for switch
- enhance Pattern Matching for instanceof

Record Patterns: JEP 440 - Summary

- deconstruct record values
- record patterns and type patterns can be nested
  - powerful, declarative, and composable form of data navigation and processing
- extend pattern matching to destructure instances of record classes
  - more sophisticated data queries.

Record Patterns: JEP 440 - Example



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

static void printSum(Object obj) {
  if (obj instanceof Point(int x, int y)) {
      System.out.println(x+y);
  }
}
```

- Showcase: RecordPatterns.java

Switch Expressions: JEP 361 - History

- History:
  - Preview: JDK 12, 13
  - Release: JDK 14

Switch Expressions: JEP 361 - Summary

- switch as a statement or an expression,
- both forms can use
  - traditional case ... : labels (with fall through)
  - new case ... -> labels (with no fall through)
    - new statement for yielding a value from a switch expression.
- simplify everyday coding
- prepare pattern matching in switch

Switch Expressions: JEP 361 - Example

```
switch (day) {
  case MONDAY, FRIDAY, SUNDAY -> System.out.println(6);
  case TUESDAY                -> System.out.println(7);
  case THURSDAY, SATURDAY     -> System.out.println(8);
  case WEDNESDAY              -> System.out.println(9);
}
```

- Showcase: SwitchExpressions.java

Pattern Matching for switch: JEP 441

- History:
  - Preview: JDK 17, 18, 19, 20
  - Release: JDK 21
  - co-evolved with the Record Patterns

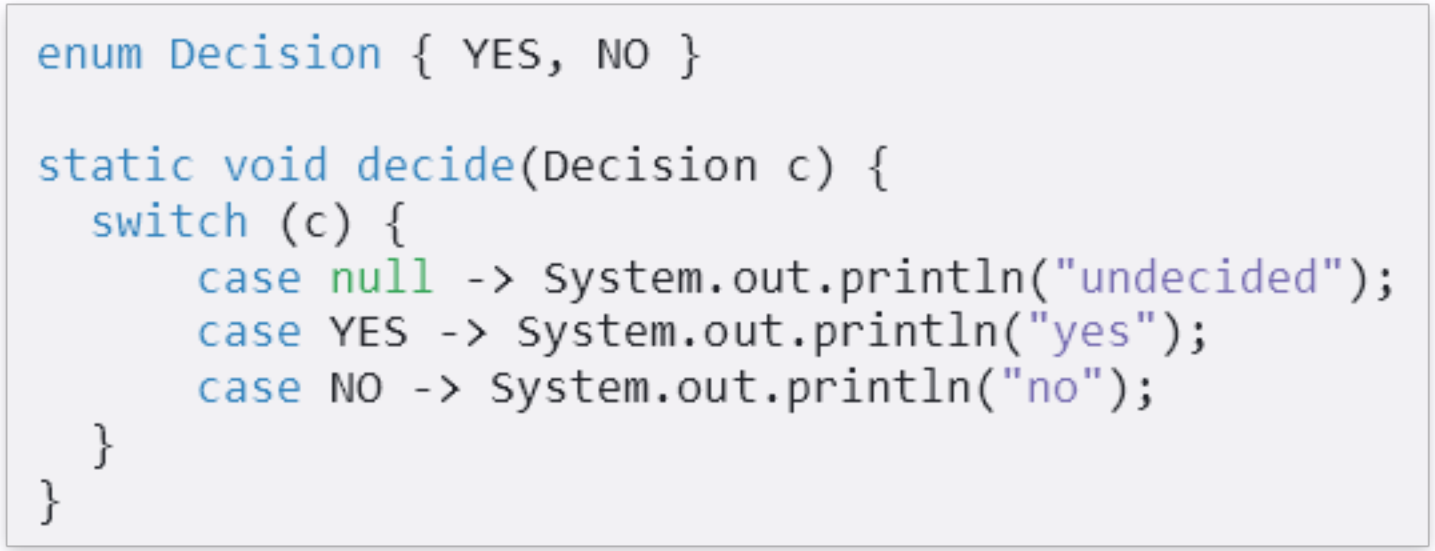
Pattern Matching for switch: JEP 441 - Summary

- extending pattern matching to switch
  - test an expression against a number of patterns, each with a specific action
  - complex data-oriented queries can be expressed concisely and safely

Pattern Matching for switch: JEP 441 - Summary

- allow patterns to appear in case labels
- combine with conditions with `when`
- checking for null now possible
- pattern switch statements have to cover all possible input values, e.g. through `default`
- backwards compatibility for "old" switch statements ensured

Pattern Matching for switch: JEP 441



```java
enum Decision { YES, NO }

static void decide(Decision c) {
  switch (c) {
      case null -> System.out.println("undecided");
      case YES -> System.out.println("yes");
      case NO -> System.out.println("no");
  }
}
```

- Showcase: PatternMatchingSwitch.java

Sealed Classes: JEP 409 - JDK 17

- restrict which other classes or interfaces may extend or implement them
- control which code is responsible for implementing it
- more declarative than access modifiers
- declare a class hierarchy that is not open for extension
- useful in API design
- classes with (non-)sealed superclass must be final, sealed or non-sealed

Sealed Classes: JEP 409 - Example

```java
public sealed interface Shape permits Circle, Rectangle {}

public final class Circle implements Shape {}
public non-sealed class Rectangle implements Shape {}

public final class Square extends Rectangle {}
```

Virtual Threads: JEP 444 - History

- Preview: JDK 19, 20
- Release: JDK 21

Virtual Threads: JEP 444 - Summary

- lightweight threads
- reduce effort of writing concurrent applications
- compatible to java.lang.Thread API
  - adoption with minimal changes
- cheap and plentiful, should never be pooled.
- short-lived and have shallow call stacks
  - e.g. a single HTTP client call or a JDBC query

Virtual Threads: JEP 444 - Summary

- number of platform threads is limited
  - JDK threads as wrappers around operating system (OS) threads
  - heavyweight and expensive
  - pooling necessary
  - long-lived, deep call stacks, and shared among many tasks
- enable server applications written in thread-per-request style to scale with near-optimal hardware utilization

Helpful NullPointerExceptions: JEP 358

- History:
  - Preview: -
  - Release: JDK 14
  - default activated in JDK 15: Ticket
- Summary:
  - improve the usability of NullPointerExceptions
  - describe which variable was null
- Showcase: HelpfulNullPointerExceptions.java

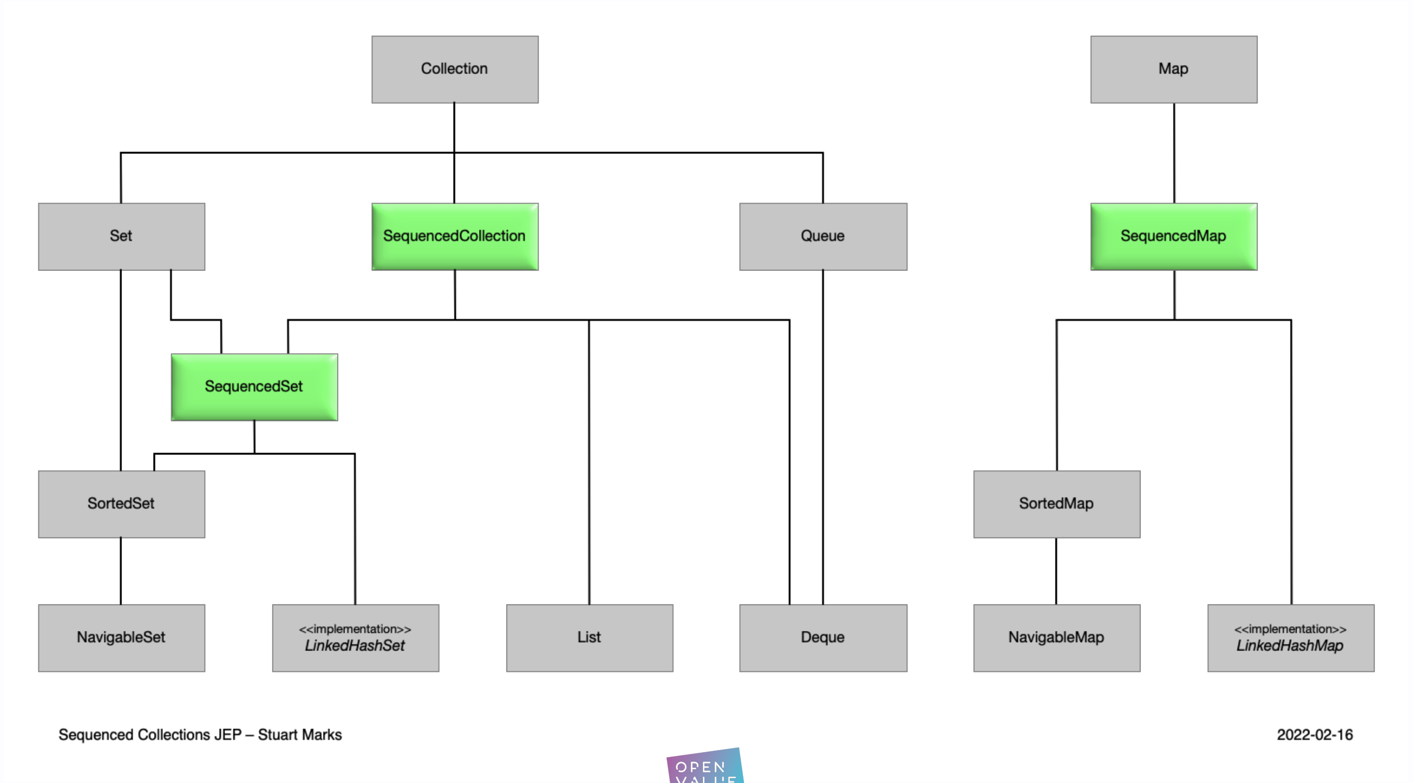
Sequenced Collections: JEP 431 - JDK 21

- unify access methods for all datastructures where order might be relevant
- methods for reversing and accessing, adding or removing the first or last element
- getting the first or last element is not always simple or even standardized

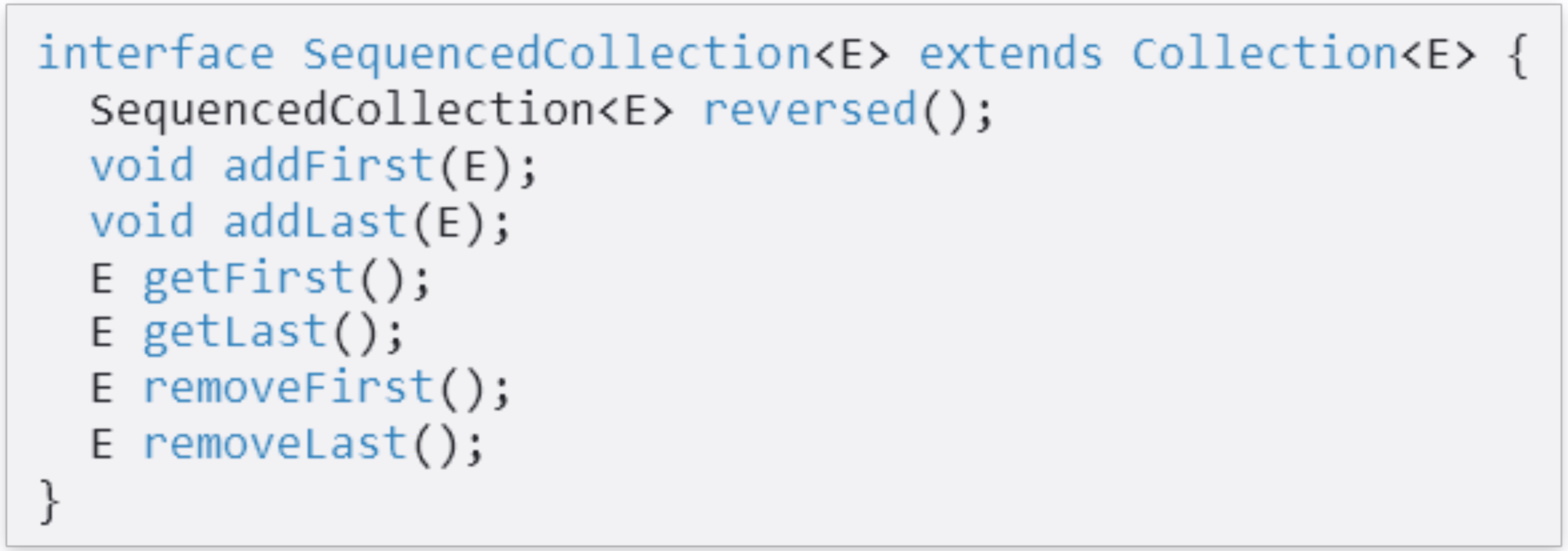
Sequenced Collections: JEP 431 - Before

	First element	**Last element**
List	list.get(0)	list.get(list.size() - 1)
Deque	deque.getFirst()	deque.getLast()
SortedSet	sortedSet.first()	sortedSet.last()
LinkedHashSet	lsh.iterator().next()	// missing

Sequenced Collections: JEP 431 - Overview



Sequenced Collections: JEP 431 - java.util.SequencedCollection

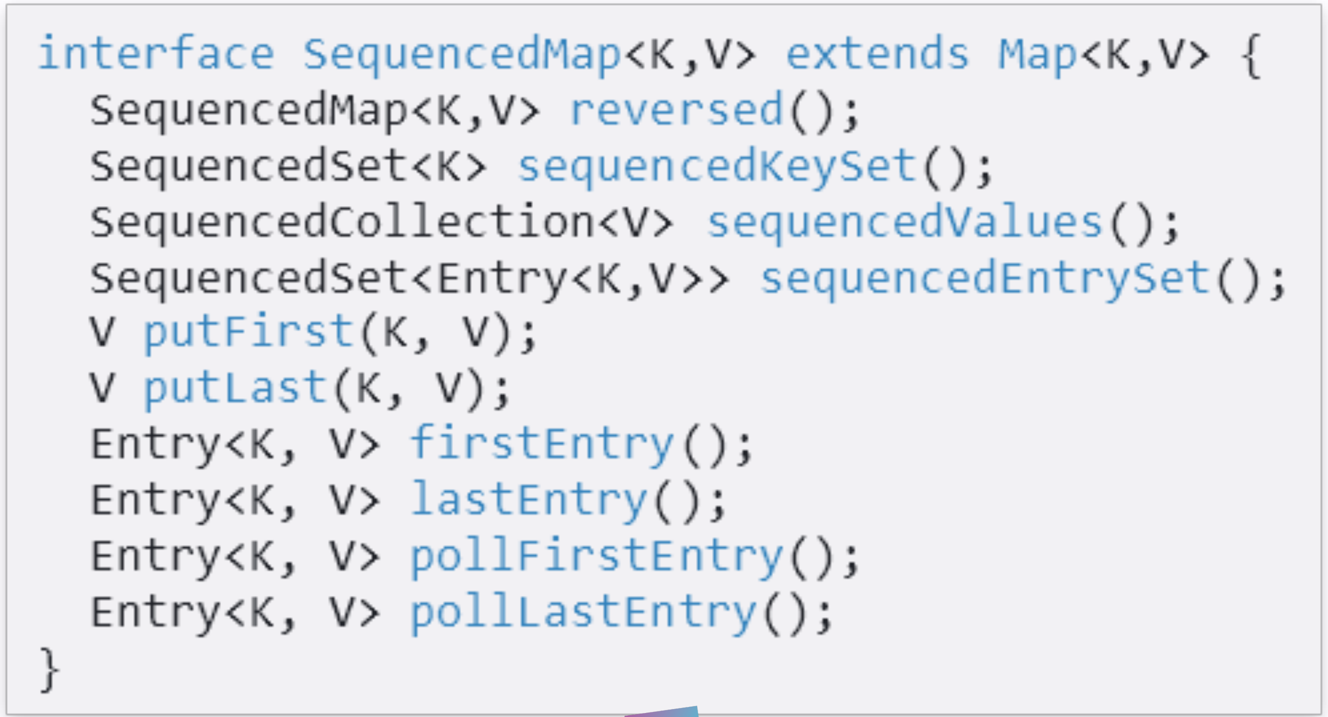


```
interface SequencedCollection<E> extends Collection<E> {
  SequencedCollection<E> reversed();
  void addFirst(E);
  void addLast(E);
  E getFirst();
  E getLast();
  E removeFirst();
  E removeLast();
}
```

Sequenced Collections: JEP 431 - java.util.SequencedSet

```
public interface SequencedSet<E> extends SequencedCollection<E>, Set<E> {
  SequencedSet<E> reversed();
}
```

Sequenced Collections: JEP 431 - java.util.SequencedMap



```java
interface SequencedMap<K,V> extends Map<K,V> {
  SequencedMap<K,V> reversed();
  SequencedSet<K> sequencedKeySet();
  SequencedCollection<V> sequencedValues();
  SequencedSet<Entry<K,V>> sequencedEntrySet();
  V putFirst(K, V);
  V putLast(K, V);
  Entry<K, V> firstEntry();
  Entry<K, V> lastEntry();
  Entry<K, V> pollFirstEntry();
  Entry<K, V> pollLastEntry();
}
```