Getting The Most From Modern Java

Simon Ritter, Deputy CTO | Azul

Introduction

Java has changed…

…a lot

- Six-month release cadence
- Eight releases since JDK 9
- More features being delivered faster than ever before

- This session will explore new features added since JDK 11
- Helping you to be ready for JDK 17, the next LTS release

Incubator Modules

- Defined by JEP 11
- Non-final APIs and non-final tools
  - Deliver to developers to solicit feedback
  - Can result in changes or even removal
  - First example: HTTP/2 API (Introduced in JDK 9, final in JDK 11)

Preview Features

- Defined by JEP 12
- New feature of the Java language, JVM or Java SE APIs
  - Fully specified, fully implemented but not permanent
  - Solicit developer real-world use and experience
  - May lead to becoming a permanent feature in future release
- Must be explicitly enabled
  - `javac --release 17 --enable-preview ...`
  - `java --enable-preview ...`
- Preview APIs
  - May be required for a preview language feature
  - Part of the Java SE API (java or javax namespace)
- All language features from JDK 12 onwards are initially included as preview features

Foojay.io

- Friends of OpenJDK
- Lots of information

azul

JDK 12

Switch Expressions

- Switch construct was a statement
  - No concept of generating a result that could be assigned
- Rather clunky syntax
  - Every case statement needs to be separated
  - Must remember break (default is to fall through)
  - Scope of local variables is not intuitive

Old-Style Switch Statement

```
int numberOfLetters;
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numberOfLetters = 6;
        break;
    case TUESDAY:
        numberOfLetters = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        numberOfLetters = 8;
        break;
    case WEDNESDAY:
        numberOfLetters = 9;
        break;
    default:
        throw new IllegalStateException("Huh?: " + day); };
```

New-Style Switch Expression

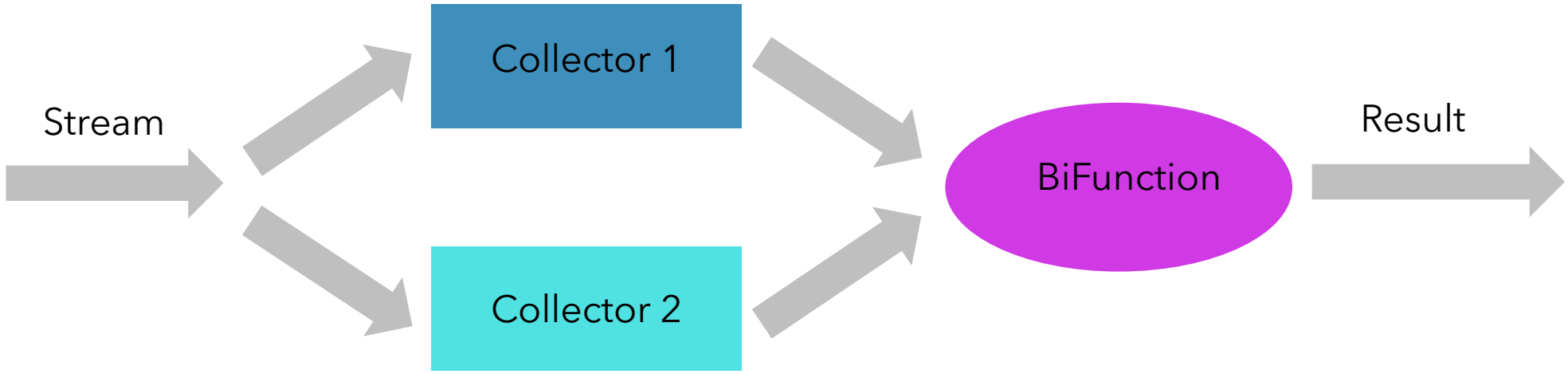
```
int numberOfLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY -> 7;
    case THURSDAY, SATURDAY -> 8;
    case WEDNESDAY -> 9;
    default -> throw new IllegalStateException("Huh?: " + day);
};
```

New Old-Style Switch Expression

```
int numberOfLetters = switch (day) {
  case MONDAY:
  case FRIDAY:
  case SUNDAY:
    break 6;
  case TUESDAY:
    break 7;
  case THURSDAY:
  case SATURDAY:
    break 8;
  case WEDNESDAY:
    break 9;
  default:
    throw new IllegalStateException("Huh?: " + day);
};
```

Streams

- New collector, teeing
  - `teeing(Collector, Collector, BiFunction)`
- Collect a stream using two collectors
- Use a BiFunction to merge the two collections



Streams

```
// Averaging
Double average = Stream.of(1, 4, 5, 2, 1, 7)
  .collect(teeing(summingDouble(i -> i), counting(),
                  (sum, n) -> sum / n));
```

azul

JDK 13

Text Blocks

```
String webPage = """
                 <html>
                   <body>
                     <p>My web page</p>
incidental white space   </body>
                 </html>     """;
System.out.println(webPage);
```

Must be followed by newline

Any trailing whitespace is stripped

```
$ java WebPage
<html>
  <body>
    <p>My web page</p>
  </body>
</html>
$
```

Text Blocks

```
String webPage = """
                 <html>
                   <body>
                     <p>My web page</p>
                   </body>
                 </html>
                 """;
System.out.println(webPage);

$ java WebPage
    <html>
      <body>
        <p>My web page</p>
      </body>
    </html>

$
```

Intentional indentation

incidental white space

Additional blank line

Switch Expression

```
int numberOfLetters = switch (day) {
  case MONDAY:
  case FRIDAY:
  case SUNDAY:
    break 6;
  case TUESDAY:
    break 7;
  case THURSDAY:
  case SATURDAY:
    break 8;
  case WEDNESDAY:
    break 9;
  default:
    throw new IllegalStateException("Huh?: " + day);
};
```

Switch Expression

```
int numberOfLetters = switch (day) {
  case MONDAY:
  case FRIDAY:
  case SUNDAY:
    yield 6;
  case TUESDAY:
    yield 7;
  case THURSDAY:
  case SATURDAY:
    yield 8;
  case WEDNESDAY:
    yield 9;
  default:
    throw new IllegalStateException("Huh?: " + day);
};
```

azul

JDK 14

Simple Java Data Class

```
class Point {
  private final double x;
  private final double y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double x() {
    return x;
  }

  public double y() {
    return y;
  }
}
```

Records

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


record Anything<T>(T t) { }   // Generic Record


public record Circle(double radius) {
  private static final double PI = 3.142;  // Static instance fields are allowed

  public double area() {
    return PI * radius * radius;
  }
}
```

Record Additional Details

- The base class of all records is `java.lang.Record`
  - Records cannot sub-class (but may implement interfaces)
- Object methods `equals()`, `hashCode()` and `toString()` can be overridden
- Records are implicitly final (although you may add the modifier)
- Records do not follow the Java bean pattern
  - `x()` not `getX()` in Point example
  - `record Point(getX, getY)  // If you must`

Record Constructors

```
record Trex(int x, int y) {
  public Trex(int x, int y) {    // Canonical constructor
    if (x < y)
      System.out.println("inverted values");
    this.x = x;   // This line needed
    this.y = y;   // This line needed
  }
}

record Range(int low, int high) {
  public Range {    // Compact constructor
    if (low > high)
      throw new IllegalArgumentException("Bad values");
  }
}
```

Compact constructor can only throw unchecked exception

Record Constructors


```
record Trex(int x, int y) {
  public Trex(int x, int y, int z) throws TrexException {  // Standard constructor
    this(x, y);   // This line must be present

    if (x < y)
      throw new TrexException();  //  Checked Exception
  }
}
```

Record Default Constructor

```
record Trex(int x, int y) {
  public Trex() {  // Default constructor
    this(2, 3);   // This line must be present
  }
}
```

Using instanceof

```
if (obj instanceof String) {
  String s = (String)obj;
  System.out.println(s.length());
}
```

Pattern Matching instanceof

```
if (obj instanceof String s)
  System.out.println(s.length());
else
  // Use of s not allowed here
```

```
if (obj instanceof String s && s.length() > 0)
  System.out.println(s.length());
```

```
// Compiler error
if (obj instanceof String s || s.length() > 0)
  System.out.println(s.length());
```

Pattern Matching instanceof

- Uses flow scoping

```
if (!(o instanceof String s))
  return;

System.out.println(s.length());
```

Pattern Matching instanceof Puzzler

- Will this work?

```
Object s = new Object();

if (s instanceof String s)
  System.out.println("String of length " + s.length());
else
  System.out.println("No string");
```

Text Blocks

- Second preview
- Two new escape sequences

```
String continuous = """
                      This line will not \
                      contain a newline in the middle
                      and solves the extra blank line issue \
                    """;


String endSpace = """
                    This line will not    \s
                    lose the trailing spaces    \s""";
```

Foreign-Memory Access API (JEP 393)

- API for safe and efficient access to memory outside of the Java heap
- `MemorySegment`
  - Models a contiguous area of memory
- `MemoryAddress`
  - Models an individual memory address (on or off heap)
- `MemoryLayout`
  - Programmatic description of a `MemorySegment`

```
try (MemorySegment segment = MemorySegment.allocateNative(100)) {
  for (int i = 0; i < 25; i++)
    MemoryAccess.setIntAtOffset(segment, i * 4, i);
}
```

Foreign-Memory Access API (JEP 393)

- Example using MemoryLayout and VarHandle
  - Simpler access of structured data

```
SequenceLayout intArrayLayout
  = MemoryLayout.ofSequence(25,
      MemoryLayout.ofValueBits(32,
        ByteOrder.nativeOrder()));

VarHandle indexedElementHandle
  = intArrayLayout.varHandle(int.class,
      PathElement.sequenceElement());

try (MemorySegment segment = MemorySegment.allocateNative(intArrayLayout)) {
   for (int i = 0; i < intArrayLayout.elementCount().getAsLong(); i++)
       indexedElementHandle.set(segment, (long) i, i);
}
```

Helpful NullPointerException

- Who's never had an `NullPointerException`?

```
a.b.c.i = 99;
```

```
Exception in thread "main" java.lang.NullPointerException
    at Prog.main(Prog.java:5)
```

```
Exception in thread "main" java.lang.NullPointerException:
        Cannot read field "c" because "a.b" is null
    at Prog.main(Prog.java:5)
```

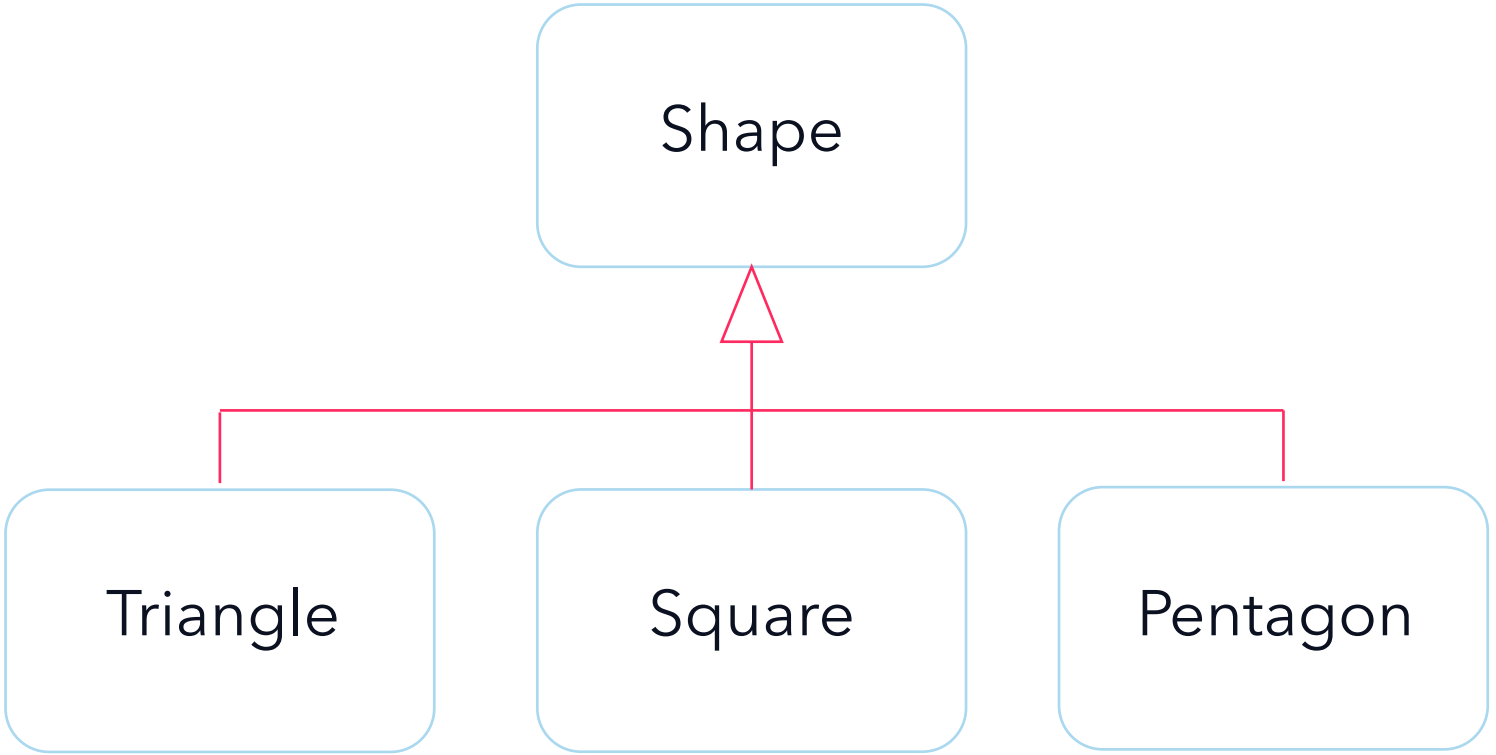
- Enabled with `-XX:+ShowCodeDetailsInExceptionMessages`

azul

JDK 15

Java Inheritance

- A class (or interface) in Java can be sub-classed by any class
  - Unless it is marked as final



Sealed Classes (JEP 360)

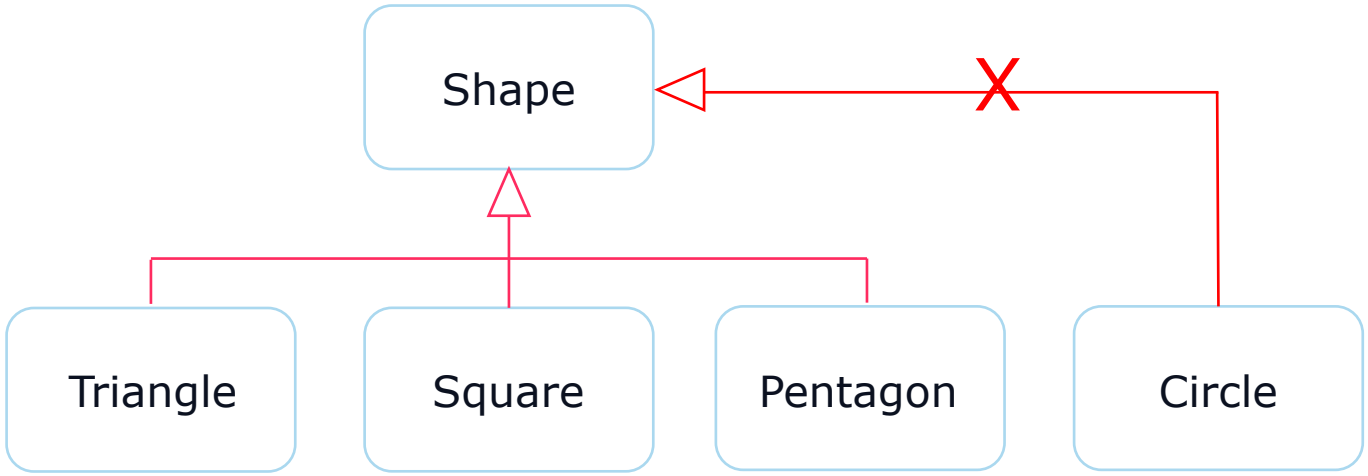
- Preview feature
- Sealed classes allow control over which classes can sub-class a class
  - Think of final as the ultimate sealed class

- Although called sealed classes, this also applies to interfaces

Sealed Classes (JEP 360)

- Classes must all be in the same package or module

```
public sealed class Shape permits Triangle, Square, Pentagon { ... }
```



Sealed Classes (JEP 360)

- All sub-classes must have inheritance capabilities explicitly specified

```
// Restrict sub-classes to defined set
public sealed class Triangle permits Equilateral, Isosoles extends Shape { ... }


// Prevent any further sub-classing
public final class Square extends Shape { ... }


// Allow any classes to sub-class this one (open)
public non-sealed class Pentagon extends Shape { ... }
```

Records (Second Preview)

- Record fields are now (really) final
  - Cannot be changed via reflection (will throw `IllegalAccessException`)
- Native methods now explicitly prohibited
  - Could introduce behaviour dependent on external state

Records (Second Preview)

- Local records
  - Like a local class
  - Implicitly static (also now applies to enums and interfaces)

```
List<Seller> findTopSellers(List<Seller> sellers, int month) {
  // Local record
  record Sales(Seller seller, double sales) {}

  return sellers.stream()
                .map(seller -> new Sales(seller, salesInMonth(seller, month)))
                .sorted((s1, s2) -> Double.compare(s2.sales(), s1.sales()))
                .map(Sales::seller)
                .collect(toList());
}
```

Records (Second Preview)

- Records work with sealed classes (interfaces)

```
public sealed interface Car permits RedCar, BlueCar { ... }

public record RedCar(int w) implements Car { ... }
public record BlueCar(long w, int c) implements Car { ... }
```

azul

JDK 16

Pattern Matching instanceof

- Now a final feature (as are Records in JDK 16)
- Two minor changes to previous iterations
  - Pattern variables are no longer explicitly final
  - Compile-time error to compare an expression of type S against a pattern of type T where S is a sub-type of T

```
static void printColoredPoint(Rectangle r) {
  if (r instanceof Rectangle rect) {
    System.out.println(rect);
  }
}
```

```
|  Error:
|  pattern type Rectangle is a subtype of expression type Rectangle
|      if (r instanceof Rectangle rect) {
|          ^-------------------------^
```

Stream toList()

- Simplified terminal operation that avoids explicit use of collect()

```
List l = Stream.of(1, 2, 3)
  .collect(Collectors.toList());
```

```
List l = Stream.of(1, 2, 3)
  .toList();
```

Period of Day

- More variation than simple A.M. or P.M.
- More descriptive

```
jshell> DateTimeFormatter.ofPattern("B").format(LocalTime.now())

$3 ==> "in the afternoon"
```

azul

JDK 17

Pattern Matching for switch

- Switch is limited on what types you can use (Integral values, Strings, enumerations)
- This is now expanded to allow type patterns to be matched
  - Like pattern matching for instanceof

```
void typeTester(Object o) {
  switch (o) {
    case null -> System.out.println("Null type");
    case String s -> System.out.println("String: " + s);
    case Color c -> System.out.println("Color with RGB: " + c.getRGB());
    case int[] ia -> System.out.println("Array of ints, length" + ia.length);
    default -> System.out.println(o.toString());
  }
}
```

Pattern Matching for switch (Completeness)

```
void typeTester(Object o) {
  switch (o) {
    case String s -> System.out.println("String: " + s);
    case Integer i -> System.out.println("Integer with value " + i.getInteger());
  } default -> System.out.println("Some other type");
} }
}
```

```
void typeTester(Shape shape) {   // Using previous sealed class example
  switch (shape) {
    case Triangle t -> System.out.println("It's a triangle");
    case Square s -> System.out.println("It's a square");
    case Pentagon p -> System.out.println("It's a pentagon");
    case Shape s -> System.out.println("It's a shape");
  }
}
```

Guarded Patterns

```
void shapeTester(Shape shape) {   // Using previous sealed class example
  switch (shape) {
    case Triangle t && t.area() > 25 -> System.out.println("It's a big triangle");
    case Triangle t -> System.out.println("It's a small triangle");
    case Square s -> System.out.println("It's a square");
    case Pentagon p -> System.out.println("It's a pentagon");
    case Shape s -> System.out.println("It's a shape");
  }
}
```

GuardedPattern:
PrimaryPattern && ConditionalAndExpression

azul

Compatability Issues

Removed From The JDK

- JDK 14: CMS Garbage Collector
  - You should really be using G1 (or Azul Prime)
- JDK 15: Nashorn scripting engine
  - JavaScript from Java?
- JDK 17: Experimental AOT and JIT compilers
  - Didn't shown much appeal
- JDK 17: Deprecate the Security Manager for removal
  - No, it doesn't make Java less secure

Internal JDK APIs

- JDK 9 introduced encapsulation of internal JDK APIs
  - Never intended for general developer use
  - Too difficult for backwards compatibility
    - Off by default
    - Controlled by `--illegal-access` flag
- JDK 16 took this one step further
  - Default became deny access
  - Access could still be turned back on
- JDK 17 completes strong encapsulation (almost)
  - The `--illegal-access` flag now has no effect (just a warning)
  - Critical APIs (like `sun.misc.Unsafe`) are still accessible

azul

Summary

Azul Platform Core / Azul Zulu Builds of OpenJDK

- Enhanced build of OpenJDK source code
  - Fully TCK tested
  - JDK 6, 7, 8, 11, 13, 15 and 17 supported with updates
- Wide platform support:
  - Intel 64-bit Windows, Mac, Linux
  - Intel 32-bit Windows and Linux
- Real drop-in replacement for Oracle JDK
  - Many enterprise customers
  - No reports of any compatibility issues

Conclusions

- The six-month release cycle is working well
- The language is developing to address some developer pain-points
- There are some other new features we have not been able to cover
  - JVM specific things
- Use Azul Platform Core, with Azul Zulu builds of OpenJDK, if you want to deploy to production

- Remember to check out foojay.io for additional Java information

azul

Questions?