Modern Java: From 8 to 21 and Beyond

A deep dive into Java's evolution →

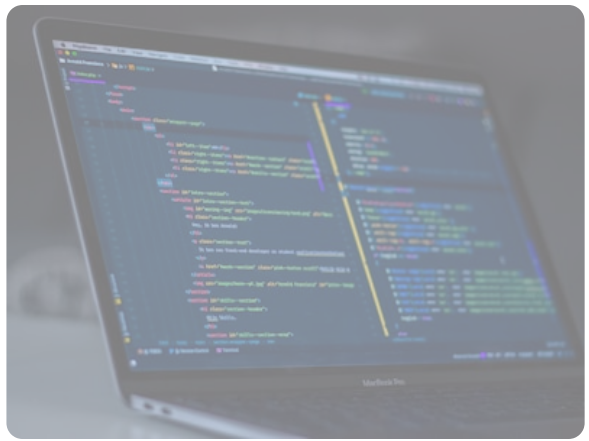
Contact Info

Ken Kousen
Kousen IT, Inc.

- ken.kousen@kousenit.com
- http://www.kousenit.com
- http://kousenit.org (blog)
- Social Media:
  - @kenkousen (Twitter)
  - @kousenit.com (Bluesky)
  - https://www.linkedin.com/in/kenkousen/ (LinkedIn)
- *Tales from the jar side* (free newsletter)
  - https://kenkousen.substack.com
  - https://youtube.com/@talesfromthejarside

Java 8: The Functional Revolution

- Lambda expressions and method references
- Stream API for data processing
- Optional for null safety
- Default and static methods in interfaces



Lambda Expressions

```
// Before Java 8 - comparing by length
Collections.sort(list, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
});

// With Java 8 lambdas
list.sort((s1, s2) -> Integer.compare(s1.length(), s2.length()));

// Best practice: use Comparator static method
list.sort(Comparator.comparingInt(String::length));
```

ProcessDictionaryV2: A Java 8 Showcase

```
private <T> T processFile(Function<Stream<String>, T> processor) {
    try (Stream<String> words = Files.lines(DICTIONARY)) {
        return processor.apply(words);
    } catch (IOException e) {
        throw new UncheckedIOException(e);
    }
}
```

- Higher-order functions with generics
- Try-with-resources for automatic resource management
- Functional programming pattern

Stream Processing Example

```
private int maxLength() {
    return processFile(words ->
            words.mapToInt(String::length)
                    .max()
                    .orElse(0));
}
```

- Method reference: `String::length`
- IntStream for primitive operations
- Optional handling with `orElse()`

Collectors: The Power Tool You Missed

```
public void printWordsOfEachLength() {
    processFile(words -> {
        words.filter(s -> s.length() > maxForFilter)
                .collect(Collectors.groupingBy(String::length))
                .forEach((len, wordList) ->
                    System.out.printf("%d: %s%n", len, wordList));
        return null;
    });
}
```

`groupingBy` creates a Map of words grouped by length

Downstream Collectors

```
Map<Integer, Long> map = words
    .filter(filter::apply)
    .collect(Collectors.groupingBy(
        String::length,
        Collectors.counting()
    ));
```

- Count elements in each group
- Process stream only once
- Combine multiple operations

Teeing Collector (Java 12)

```
WordStats wordStats = words.filter(w -> w.length() >= maxForFilter)
    .collect(Collectors.teeing(
        Collectors.counting(),
        Collectors.teeing(
            Collectors.averagingDouble(String::length),
            Collectors.maxBy(Comparator.comparingInt(String::length)),
            (avg, maxWord) -> new WordStats(0, avg, maxWord.orElse(""))
        ),
        (count, stats) -> new WordStats(
            count, stats.averageLength(), stats.longestWord()
        )
    ));
```

- Combine multiple collectors into one
- Process stream only once
- Perfect for statistics gathering

Static and Default Methods in Interfaces

```
public interface SumNumbers {
    default int addEvens(int... nums) {
        return add(n -> n % 2 == 0, nums);
    }

    default int addOdds(int... nums) {
        return add(n -> n % 2 != 0, nums);
    }

    private int add(IntPredicate predicate, int... nums) {
        return IntStream.of(nums)
                .filter(predicate)
                .sum();
    }
}
```

- Interface evolution without breaking changes
- Private methods (Java 9) for code reuse
- Default methods provide implementations

JShell: Java's REPL (Java 9)

```
jshell> int x = 10
x ==> 10

jshell> x * x
$2 ==> 100

jshell> List.of("Java", "Python", "Kotlin")
$3 ==> [Java, Python, Kotlin]

jshell> /methods
|    int square(int)
```

- Interactive Java shell for experimentation
- Perfect for learning and demos
- No need for main methods or compilation

Optional: Creating Instances

```
// Creating Optionals
Optional<String> empty = Optional.empty();
Optional<String> value = Optional.of("Hello");
Optional<String> nullable = Optional.ofNullable(getString());
```

- `empty()` for known absent values
- `of()` for non-null values (throws NPE if null)
- `ofNullable()` for potentially null values

Optional: Chaining Operations

```
String result = Optional.ofNullable(getName())
    .map(String::toUpperCase)
    .filter(s -> s.length() > 3)
    .orElse("DEFAULT");
```

Functional pipeline for null-safe transformations

Optional: New Methods

```
// Java 9: or() method
Optional<String> alternative = first.or(() -> second)
                                    .or(() -> third);

// Java 11: isEmpty()
if (optional.isEmpty()) {
    // handle empty case
}
```

Local Variable Type Inference (Java 10)

```
// Before Java 10
Map<String, List<Customer>> customersByCity =
    new HashMap<String, List<Customer>>();

// With var
var customersByCity = new HashMap<String, List<Customer>>();
```

Reduces boilerplate while maintaining strong typing

var with Complex Types

```
// Perfect for stream results
var result = customers.stream()
    .filter(c -> c.orders() > 10)
    .collect(Collectors.groupingBy(Customer::city));

// Works with anonymous types
var anonymous = new Object() {
    String name = "Java";
    int version = 21;
};
```

Text Blocks (Java 15)

```
// JSON example
String json = """
    {
        "name": "%s",
        "version": %d,
        "features": [
            "records",
            "pattern matching",
            "virtual threads"
        ]
    }
    """.formatted(name, version);

// SQL example
String query = """
    SELECT c.name, COUNT(o.id) as order_count
    FROM customers c
    LEFT JOIN orders o ON c.id = o.customer_id
    WHERE c.active = true
    GROUP BY c.name
    HAVING COUNT(o.id) > ?
    """;
```

Records: Simple Data Classes (Java 16)

```java
// Simple record
public record Customer(String name, String email) {}

// Automatically provides:
// - Constructor
// - Getters (name(), email())
// - equals(), hashCode(), toString()
```

Records: Validation and Methods

```
public record Person(String first, String last, LocalDate dob) {
    // Compact constructor for validation
    public Person {
        Objects.requireNonNull(first);
        Objects.requireNonNull(last);
        if (dob.isAfter(LocalDate.now())) {
            throw new IllegalArgumentException("Invalid date of birth");
        }
    }

    // Additional methods
    public String fullName() {
        return String.format("%s %s", first, last);
    }
}
```

Pattern Matching: instanceof (Java 14)

```java
// Before
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.toUpperCase());
}

// With pattern matching
if (obj instanceof String s) {
    System.out.println(s.toUpperCase());
}
```

No explicit cast needed!

Switch Expressions (Java 14)

```
String result = switch (day) {
    case MONDAY, FRIDAY -> "Work day";
    case SATURDAY, SUNDAY -> "Weekend";
    default -> "Midweek";
};
```

- Arrow syntax prevents fall-through
- Can return values
- Multiple case labels

Pattern Matching in Switch (Java 21)

```
String format(Object obj) {
    return switch (obj) {
        case Integer i -> String.format("int %d", i);
        case Long l -> String.format("long %d", l);
        case Double d -> String.format("double %f", d);
        case String s -> String.format("String %s", s);
        case null -> "null";
        default -> obj.toString();
    };
}
```

Record Pattern Matching

```
public String getArea(Shape shape) {
    return switch (shape) {
        case Rectangle(double width, double height) ->
            "Rectangle area: " + (width * height);
        case Circle(double radius) ->
            "Circle area: " + (Math.PI * radius * radius);
        case Triangle(double base, double height) ->
            "Triangle area: " + (0.5 * base * height);
    };
}
```

Direct access to record components!

Pattern Matching with Guards

```
String categorize(Shape shape) {
    return switch (shape) {
        case Rectangle r when r.width() == r.height() -> "Square";
        case Rectangle r -> "Rectangle";
        case Circle c when c.radius() > 10 -> "Large circle";
        case Circle c -> "Small circle";
        default -> "Unknown shape";
    };
}
```

Conditional patterns with `when` clauses

Sealed Classes (Java 17)

```
public abstract sealed class Shape
    permits Circle, Rectangle, Square {

    public abstract double area();
}
```

Explicitly control which classes can extend Shape

Sealed Class Implementation

```
public final class Circle extends Shape {
    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

Exhaustive Switch with Sealed Classes

```
double calculateArea(Shape shape) {
    return switch (shape) {
        case Circle c -> Math.PI * c.radius() * c.radius();
        case Rectangle r -> r.width() * r.height();
        case Square s -> s.side() * s.side();
        // No default needed - compiler knows all subtypes!
    };
}
```

HTTP Client (Java 11)

```
public class AstroClient {
    private final HttpClient client = HttpClient.newHttpClient();

    public CompletableFuture<AstroResponse> getAstronautsAsync() {
        var request = HttpRequest.newBuilder()
            .uri(URI.create("http://api.open-notify.org/astros.json"))
            .header("Accept", "application/json")
            .GET()
            .build();

        return client.sendAsync(request,
                HttpResponse.BodyHandlers.ofString())
            .thenApply(HttpResponse::body)
            .thenApply(json -> gson.fromJson(json, AstroResponse.class));
    }
}
```

- Modern API design
- Built-in async support
- HTTP/2 ready
- No external HTTP client dependency needed

String Enhancements (Java 11)

```
String text = "  Hello World  ";

// isBlank() - true if empty or whitespace only
"".isBlank();        // true
"   ".isBlank();     // true
"Hello".isBlank();   // false

// strip() methods - Unicode-aware trimming
text.strip();         // "Hello World"
text.stripLeading();  // "Hello World  "
text.stripTrailing(); // "  Hello World"

// repeat() and lines()
"Java".repeat(3);     // "JavaJavaJava"
"Line1\nLine2".lines().count();  // 2
```

Traditional Threads: Limited Scalability

```
try (var executor = Executors.newFixedThreadPool(10)) {
    for (int i = 0; i < 1000; i++) {
        executor.submit(() -> {
            Thread.sleep(1000);
            doWork();
        });
    }
}
```

Limited by OS threads (expensive resources)

Virtual Threads (Java 21)

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    for (int i = 0; i < 1_000_000; i++) {
        executor.submit(() -> {
            Thread.sleep(1000);
            doWork();
        });
    }
}
```

- Can handle millions of threads!
- Lightweight, JVM-managed
- Makes blocking I/O scalable

Virtual Threads in Practice

```
public class CustomerService {
    public CompletableFuture<Customer>
            findCustomerWithOrdersVirtual(long customerId) {

        return CompletableFuture.supplyAsync(() -> {
            // These block, but it's OK with virtual threads!
            Customer customer = customerRepository
                .findById(customerId);
            List<Order> orders = orderRepository
                .findByCustomerId(customerId);

            return new CustomerWithOrders(customer, orders);
        }, virtualThreadExecutor);
    }
}
```

- Simple blocking code
- No callback hell
- Scales like async
- Best of both worlds

CompletableFuture (Java 8)

```
// Asynchronous composition
CompletableFuture<String> future = CompletableFuture
    .supplyAsync(() -> fetchUserData(userId))
    .thenApply(user -> user.getName())
    .thenApply(String::toUpperCase)
    .thenCompose(name -> fetchUserPreferences(name))
    .exceptionally(throwable -> "Default User");

// Combining multiple futures
CompletableFuture<String> result = CompletableFuture
    .allOf(future1, future2, future3)
    .thenApply(v -> future1.join() + future2.join() + future3.join());
```

Foundation for async programming in Java

Sequenced Collections (Java 21)

```java
// New interfaces for ordered collections
SequencedCollection<String> list = new ArrayList<>();
list.add("first");
list.add("last");

// Direct access to first and last elements
String first = list.getFirst();  // "first"
String last = list.getLast();    // "last"

// Reversed view
SequencedCollection<String> reversed = list.reversed();

// Works with LinkedHashSet, LinkedHashMap too
SequencedSet<String> set = new LinkedHashSet<>();
SequencedMap<String, Integer> map = new LinkedHashMap<>();
```

Stream API: takeWhile (Java 9)

```
Stream.of(2, 4, 6, 8, 9, 10, 12)
    .takeWhile(n -> n % 2 == 0)  // [2, 4, 6, 8]
    .forEach(System.out::println);
```

Takes elements while predicate is true

Stream API: dropWhile (Java 9)

```
Stream.of(2, 4, 6, 8, 9, 10, 12)
    .dropWhile(n -> n % 2 == 0)  // [9, 10, 12]
    .forEach(System.out::println);
```

Drops elements while predicate is true, returns the rest

Stream API: More Java 9 Features

```
// Stream.iterate with predicate
Stream.iterate(1, n -> n < 100, n -> n * 2)
    .forEach(System.out::println);  // 1, 2, 4, 8, 16, 32, 64

// Stream.ofNullable
Stream.ofNullable(getValue())
    .flatMap(Collection::stream)
    .forEach(System.out::println);
```

Stream API: Convenience Methods

```java
// Java 16: Direct toList()
List<String> list = stream.toList();

// Instead of
List<String> oldWay = stream.collect(Collectors.toList());
```

Stream Gatherers (Java 24)

Custom intermediate operations for streams

```
// Built-in gatherers
List<List<Integer>> windows = numbers.stream()
    .gather(Gatherers.windowFixed(3))
    .toList();

List<Integer> runningSum = numbers.stream()
    .gather(Gatherers.scan(() -> 0, Integer::sum))
    .toList();
```

Stream Gatherers: Built-in Operations

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

// Fixed-size windows
List<List<Integer>> fixed = numbers.stream()
    .gather(Gatherers.windowFixed(3))
    .toList(); // [[1,2,3], [4,5,6], [7,8,9]]

// Sliding windows
List<List<Integer>> sliding = numbers.stream()
    .gather(Gatherers.windowSliding(3))
    .toList(); // [[1,2,3], [2,3,4], [3,4,5], ...]
```

Stream Gatherers: Custom Implementation

```
// Custom gatherer for consecutive pairs
Gatherer<Integer, ArrayList<Integer>, Pair<Integer, Integer>> pairwise =
    Gatherer.ofSequential(
        ArrayList::new,
        Gatherer.Integrator.ofGreedy((state, element, downstream) -> {
            state.add(element);
            if (state.size() == 2) {
                boolean result = downstream.push(
                    new Pair<>(state.get(0), state.get(1)));
                state.clear();
                return result;
            }
            return true;
        })
    );
```

Parallel Streams (Java 8)

```
// Sequential processing
long count = bigList.stream()
    .filter(expensiveOperation)
    .count();

// Parallel processing
long count = bigList.parallelStream()
    .filter(expensiveOperation)
    .count();
```

Three requirements for useful parallel streams:

1. **LOT of data** or **time-consuming operations** per element
2. **Data should be easy to partition** (ArrayList > LinkedList)
3. **Operations must be stateless and associative**

Switch with yield

```
int numDays = switch (month) {
    case JANUARY, MARCH, MAY, JULY, AUGUST, OCTOBER, DECEMBER -> 31;
    case APRIL, JUNE, SEPTEMBER, NOVEMBER -> 30;
    case FEBRUARY -> {
        yield Year.of(year).isLeap() ? 29 : 28;
    }
};
```

Exhaustive Switch

```
enum Size { SMALL, MEDIUM, LARGE }

String getPrice(Size size) {
    return switch (size) {
        case SMALL -> "$5";
        case MEDIUM -> "$10";
        case LARGE -> "$15";
        // No default needed - compiler knows all enum values!
    };
}
```

Works with enums, sealed classes, and boolean type

Reading Files with Streams

```
try (Stream<String> lines = Files.lines(path)) {
    lines.filter(line -> line.contains("ERROR"))
         .map(String::toUpperCase)
         .forEach(System.out::println);
}
```

Lazy processing of large files

Simple File Operations (Java 11)

```
// Text files
String content = Files.readString(path);
Files.writeString(path, "Hello, World!");

// Binary files (images, etc.)
byte[] imageData = Files.readAllBytes(imagePath);
Files.write(outputPath, imageData);
```

File Comparison (Java 12)

```
long mismatch = Files.mismatch(path1, path2);
if (mismatch == -1) {
    System.out.println("Files are identical");
} else {
    System.out.println("Files differ at byte: " + mismatch);
}
```

Simple Web Server (Java 18)

```
# Start a web server in current directory
$ jwebserver
Binding to loopback by default. For all interfaces use "-b 0.0.0.0".
Serving /current/directory on 127.0.0.1 port 8000

# Custom port and directory
$ jwebserver -p 3000 -d /path/to/files
```

```
// Programmatic usage
var server = SimpleFileServer.createFileServer(
    new InetSocketAddress(8080),
    Path.of("/www"),
    OutputLevel.VERBOSE
);
server.start();
```

Process API (Java 9)

```
// Current process information
ProcessHandle current = ProcessHandle.current();
System.out.println("PID: " + current.pid());
current.info().command().ifPresent(cmd ->
    System.out.println("Command: " + cmd));

// List all processes (useful for monitoring, MCP servers, etc.)
ProcessHandle.allProcesses()
    .filter(p -> p.info().command()
        .map(cmd -> cmd.contains("java"))
        .orElse(false))
    .forEach(p -> System.out.printf(
        "PID: %d, Command: %s%n",
        p.pid(),
        p.info().command().orElse("unknown")));
```

Immutable Collections (Java 9)

```
List<String> list = List.of("Java", "Python", "Kotlin");
Set<Integer> set = Set.of(1, 2, 3, 4, 5);
Map<String, Integer> map = Map.of(
    "one", 1,
    "two", 2,
    "three", 3
);
```

Convenient and immutable by default

Maps with Many Entries

```
Map<String, String> bigMap = Map.ofEntries(
    Map.entry("key1", "value1"),
    Map.entry("key2", "value2"),
    Map.entry("key3", "value3"),
    // ... more entries
);
```

Copy Factories (Java 10)

```
List<String> copy = List.copyOf(originalList);

// Creates immutable copy
// Null elements not allowed
// Changes to original don't affect copy
```

Base64 API (Java 8)

```
// Encoding
String original = "Hello, World!";
String encoded = Base64.getEncoder().encodeToString(original.getBytes());
System.out.println(encoded); // SGVsbG8sIFdvcmxkIQ==

// Decoding
byte[] decoded = Base64.getDecoder().decode(encoded);
String result = new String(decoded);
System.out.println(result); // Hello, World!

// URL-safe encoding (for web APIs)
String urlSafe = Base64.getUrlEncoder().encodeToString(data);

// Common with AI/ML REST APIs for image data
String imageBase64 = Base64.getEncoder().encodeToString(imageBytes);
```

Helpful NullPointerExceptions (Java 14)

Before Java 14

```
String city = person.getAddress().getCity().toUpperCase();
// Exception in thread "main" java.lang.NullPointerException
```

Java 14+

```
String city = person.getAddress().getCity().toUpperCase();
// Exception in thread "main" java.lang.NullPointerException:
//     Cannot invoke "String.toUpperCase()" because the return value of
//     "Address.getCity()" is null
```

Pinpoints exactly what was null!

CompactNumberFormat (Java 12)

```
public void testCompactNumberFormat() {
    NumberFormat shortFormat = NumberFormat
        .getCompactNumberInstance(Locale.US, NumberFormat.Style.SHORT);
    NumberFormat longFormat = NumberFormat
        .getCompactNumberInstance(Locale.US, NumberFormat.Style.LONG);

    assertEquals("1K", shortFormat.format(1000));
    assertEquals("1M", shortFormat.format(1_000_000));
    assertEquals("1B", shortFormat.format(1_000_000_000));

    assertEquals("1 thousand", longFormat.format(1000));
    assertEquals("1 million", longFormat.format(1_000_000));
}
```

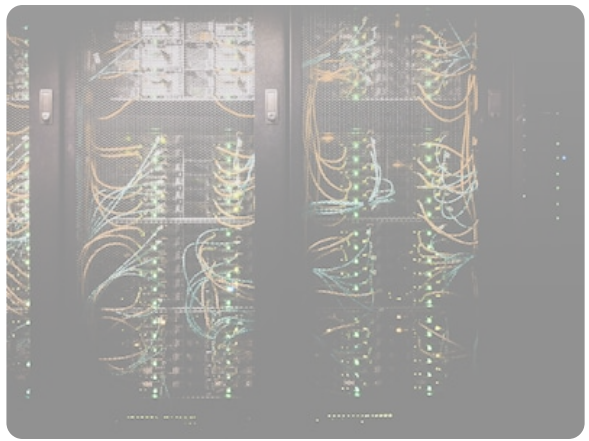
Unnamed Variables (Java 22)

```java
// Ignore values you don't need with _
try {
    int result = someMethod();
} catch (Exception _) {  // Don't care about exception details
    System.out.println("Failed");
}

// In loops - just counting
for (String _ : collection) {
    count++;
}

// Pattern matching - only care about x
switch (obj) {
    case Point(int x, _) -> System.out.println("x = " + x);
}
```

Data-Oriented Programming

Modern Java enables a new programming paradigm combining:

- **Records** for immutable data carriers
- **Sealed classes** for controlled type hierarchies
- **Pattern matching** for data processing



DOP Example: User Hierarchy

```
// Sealed interface with record implementations
public sealed interface User permits Admin, RegularUser, Guest {
    String name();
}

public record Admin(String name, Set<String> permissions) implements User {}
public record RegularUser(String name, LocalDate lastLogin) implements User {}
public record Guest(String name) implements User {}
```

DOP: Pattern Matching Processing

```
public String processUser(User user) {
    return switch (user) {
        case Admin(var name, var permissions) ->
            String.format("Admin %s with %d permissions", name, permissions.size());
        case RegularUser(var name, var lastLogin) ->
            String.format("User %s, last login: %s", name, lastLogin);
        case Guest(var name) ->
            String.format("Guest %s (limited access)", name);
        // No default needed - exhaustive!
    };
}
```

Data processing becomes declarative and type-safe!

DOP: Result Pattern Example

```
// From the astro example in this repository
public sealed interface Result<T> permits Success, Failure {
}

public record Success<T>(T data) implements Result<T> {}
public record Failure<T>(String error) implements Result<T> {}

// Client code knows there are only two possibilities
Result<AstroResponse> result = fetchAstronauts();
String message = switch (result) {
    case Success(var astroData) ->
        "Found " + astroData.number() + " astronauts";
    case Failure(var error) ->
        "Error: " + error;
};
```

See the `astro` package for the complete example!

String Templates (Preview, Withdrawn)

- Direct string interpolation
- Custom template processors
- Withdrawn in Java 23 for redesign

Structured Concurrency (Preview)

- Better concurrent task management
- Automatic cancellation
- Simplifies error handling

Foreign Function & Memory API

- Replace JNI
- Direct memory access
- Better performance than JNI

Pattern Matching Enhancements

- Unnamed patterns with `_`
- Pattern variables in loops
- More expressive code

Java 25 LTS (September 2025)

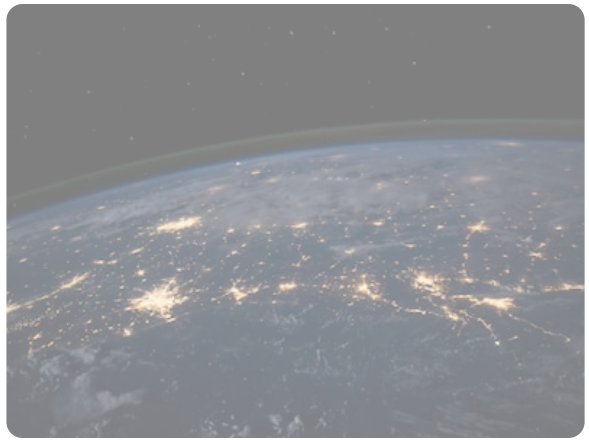
Language Features:

- Scoped Values (finalized) - Thread-safe data sharing
- Primitive Types in Patterns (3rd preview)
- Flexible Constructor Bodies (preview)

Performance & JVM:

- Ahead-of-Time Class Loading & Linking
- Compact Object Headers (production)
- Vector API (incubator)

Key Point: Long-Term Support release for enterprise adoption



Best Practice: Prefer Immutability

- Use records for data carriers
- Use `List.of()` instead of `new ArrayList()`
- Design with immutability first

Best Practice: Embrace Pattern Matching

- Replace visitor pattern with sealed classes
- Use pattern matching for cleaner code
- Leverage exhaustive switches

Best Practice: Virtual Threads for I/O

- Simple blocking code that scales
- Avoid reactive unless necessary
- One thread per request is back!

Best Practice: Use Modern APIs

- HTTP Client over HttpURLConnection
- Modern date/time API over Date/Calendar
- Use `var` wisely for complex types

Official Documentation

- https://dev.java/
- JEP (Java Enhancement Proposals)
- https://openjdk.org/projects/jdk/ (JEPs by release)
- OpenJDK documentation

Recommended Books

- "Modern Java Recipes" (Kousen)
- "Modern Java in Action" (Urma, Fusco, Mycroft)
- "Effective Java, 3rd Edition" (Bloch)
- "Java: The Complete Reference" (Schildt)

Online Resources

- https://www.baeldung.com
- https://kousenit.com
- https://inside.java
- https://javaalmanac.io

This Repository

- https://github.com/kousen/java_latest
- Full examples and tests
- Ready-to-run demonstrations

Thank You!

Questions?

Kenneth Kousen
Author, Speaker, Java Champion

kousenit.com | @kenkousen