MODERN JAVA FEATURES - JAVA 7 TO JAVA 25

🎯 COMPLETE GUIDE TO JAVA EVOLUTION

📋 TỔNG QUAN

Thông tin	Chi tiết
Scope	Java 7 (2011) → Java 25 (2025)
Thời lượng	10-12 buổi × 2 giờ = 20-24 giờ
Format	Có thể standalone hoặc integrate vào khóa học
Yêu cầu đầu vào	Java basics (biết Java 6 fundamentals)
Đầu ra	Modern Java mastery + LTS version expertise

🗺 JAVA VERSION TIMELINE

```
2011: Java 7 (Diamond operator, try-with-resources)
       ↓
2014: Java 8 ⭐ LTS (Lambda, Stream API, Optional) - MAJOR RELEASE
       ↓
2017: Java 9 (Modules, JShell)
       ↓
2018: Java 10 (var keyword)
       ↓
2018: Java 11 ⭐ LTS (HTTP Client, String methods)
       ↓
2019-2021: Java 12-16 (Preview features, Records, Pattern Matching)
       ↓
2021: Java 17 ⭐ LTS (Sealed Classes, Pattern Matching)
       ↓
2022-2023: Java 18-20 (Virtual Threads preview)
       ↓
2023: Java 21 ⭐ LTS (Virtual Threads, Sequenced Collections)
       ↓
2024-2025: Java 22-25 (Stream Gatherers, Flexible Constructor Bodies)
```

⭐ **LTS Versions:** 8, 11, 17, 21 (recommended for production)

📚 CẤU TRÚC KHÓA HỌC (12 BUỔI)

Buổi	Phiên Bản	Chủ Đề	Thời Lượng

Buổi	Phiên Bản	Chủ Đề	Thời Lượng
1	Java 7	Small Enhancements + NIO.2	2h
2-3	Java 8 (Part 1)	Lambda & Functional Interfaces	4h
4	Java 8 (Part 2)	Stream API Mastery	2h
5	Java 8 (Part 3)	DateTime API & Optional	2h
6	Java 9-10	Modules, var, Collections Factory	2h
7	Java 11	HTTP Client, String API, Performance	2h
8	Java 12-16	Switch Expressions, Text Blocks, Records	2h
9	Java 17	Sealed Classes, Pattern Matching	2h
10	Java 18-20	Preview Features Overview	2h
11	Java 21	Virtual Threads, Sequenced Collections	2h
12	Java 22-25	Latest Features & Future	2h

TỔNG: 24 giờ

CHI TIẾT TỪNG BUỔI

▧ BUỔI 1: JAVA 7 (2011)

Mục tiêu:

Hiểu những cải tiến nhỏ nhưng hữu ích của Java 7

Nội dung:

1. Diamond Operator (<>)

Before Java 7:

```java
List<String> list = new ArrayList<String>();
Map<String, List<Integer>> map = new HashMap<String, List<Integer>>();
```

Java 7:

```java
List<String> list = new ArrayList<>();
Map<String, List<Integer>> map = new HashMap<>();
```

---

2. Try-with-Resources

Before Java 7:

```
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("file.txt"));
    String line = br.readLine();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (br != null) {
        try {
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Java 7:

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    String line = br.readLine();
} catch (IOException e) {
    e.printStackTrace();
}
// br is auto-closed
```

Multiple resources:

```
try (FileInputStream fis = new FileInputStream("input.txt");
     FileOutputStream fos = new FileOutputStream("output.txt")) {
    // Use resources
} // Both auto-closed in reverse order
```

---

3. Multi-catch Exception

Before Java 7:

```
try {
    // code
} catch (IOException e) {
    logger.error("Error", e);
} catch (SQLException e) {
    logger.error("Error", e);
}
```

Java 7:

```
try {
    // code
} catch (IOException | SQLException e) {
    logger.error("Error", e);
}
```

---

4. String in Switch

Before Java 7: Only int, byte, short, char, enum

Java 7:

```
String day = "MONDAY";

switch (day) {
    case "MONDAY":
        System.out.println("Start of week");
        break;
    case "FRIDAY":
        System.out.println("End of week");
        break;
    default:
        System.out.println("Midweek");
}
```

---

5. Numeric Literals with Underscores

```
// Before: Hard to read
int million = 1000000;
long creditCard = 1234567890123456L;

// Java 7: Readable
int million = 1_000_000;
long creditCard = 1234_5678_9012_3456L;
double pi = 3.14_15_92_65;

// Binary literals
int binary = 0b1010_1010;
```

---

6. NIO.2 (New I/O 2) - File System API

```
// Path instead of File
Path path = Paths.get("example.txt");

// Files utility class
List<String> lines = Files.readAllLines(path);
Files.write(path, lines);

// Copy, move, delete
Files.copy(source, target);
Files.move(source, target);
Files.delete(path);

// Directory operations
Files.createDirectory(Paths.get("newDir"));
Files.createDirectories(Paths.get("parent/child"));

// Walk file tree
Files.walk(Paths.get("."))
     .filter(Files::isRegularFile)
     .forEach(System.out::println);

// Watch Service
WatchService watchService = FileSystems.getDefault().newWatchService();
path.register(watchService, StandardWatchEventKinds.ENTRY_CREATE);
```

---

7. Fork/Join Framework (Preview)

```
ForkJoinPool pool = new ForkJoinPool();
RecursiveTask<Long> task = new SumTask(array, 0, array.length);
long result = pool.invoke(task);
```

Thực hành:

1. Refactor code cũ sang diamond operator
2. Convert traditional try-finally → try-with-resources
3. File operations với NIO.2
4. String switch cases
5. Numeric literals formatting

▧ BUỔI 2-3: JAVA 8 (PART 1) - LAMBDA & FUNCTIONAL (4h)

Mục tiêu:

Master Lambda expressions và Functional Programming

Nội dung:

1. Lambda Expressions Syntax

```java
// Anonymous class - OLD WAY
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello");
    }
};

// Lambda - NEW WAY
Runnable r2 = () -> System.out.println("Hello");

// With parameters
Comparator<Integer> comp1 = (a, b) -> a.compareTo(b);
Comparator<Integer> comp2 = Integer::compareTo; // Method reference

// Multiple statements
Runnable r3 = () -> {
    System.out.println("Line 1");
    System.out.println("Line 2");
};
```

---

2. Functional Interfaces

```java
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}

// Usage
Calculator add = (a, b) -> a + b;
Calculator multiply = (a, b) -> a * b;

System.out.println(add.calculate(5, 3));      // 8
System.out.println(multiply.calculate(5, 3)); // 15
```

Built-in Functional Interfaces:

```java
// Predicate<T> - test() returns boolean
Predicate<Integer> isEven = n -> n % 2 == 0;
System.out.println(isEven.test(4)); // true
```

```java
// Function<T, R> - apply() transforms T to R
Function<String, Integer> length = s -> s.length();
System.out.println(length.apply("Hello")); // 5

// Consumer<T> - accept() consumes T
Consumer<String> print = s -> System.out.println(s);
print.accept("Hello");

// Supplier<T> - get() supplies T
Supplier<Double> random = () -> Math.random();
System.out.println(random.get());

// BiFunction<T, U, R> - apply() with 2 params
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
System.out.println(add.apply(2, 3)); // 5
```

---

3. Method References

```java
// Static method reference
Function<String, Integer> parseInt1 = s -> Integer.parseInt(s);
Function<String, Integer> parseInt2 = Integer::parseInt;

// Instance method reference
String str = "Hello";
Supplier<Integer> getLength1 = () -> str.length();
Supplier<Integer> getLength2 = str::length;

// Instance method of arbitrary object
Function<String, String> toUpper1 = s -> s.toUpperCase();
Function<String, String> toUpper2 = String::toUpperCase;

// Constructor reference
Supplier<List<String>> listSupplier1 = () -> new ArrayList<>();
Supplier<List<String>> listSupplier2 = ArrayList::new;

Function<Integer, List<String>> listWithSize = ArrayList::new;
```

---

4. Default Methods in Interfaces

```java
interface Vehicle {
    void start();

    // Default method
    default void stop() {
        System.out.println("Vehicle stopped");
    }

    // Static method
    static void honk() {
        System.out.println("Beep beep!");
    }
}

class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car started");
    }
    // stop() inherited
}

// Usage
Car car = new Car();
car.start();
car.stop();             // Uses default
Vehicle.honk();         // Static method call
```

---

5. forEach with Lambda

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Old way
for (String name : names) {
    System.out.println(name);
}

// Lambda way
names.forEach(name -> System.out.println(name));

// Method reference
names.forEach(System.out::println);

// Map forEach
Map<String, Integer> scores = new HashMap<>();
scores.put("Alice", 90);
scores.put("Bob", 85);

scores.forEach((name, score) ->
    System.out.println(name + ": " + score));
```

Thực hành:

1. Convert anonymous classes → lambdas

2. Implement custom functional interfaces
3. Method reference exercises (all 4 types)
4. Lambda with collections (forEach, removeIf, replaceAll)
5. Comparator với lambdas & method references

▧ BUỔI 4: JAVA 8 (PART 2) - STREAM API (2h)

Mục tiêu:

Stream API mastery

Nội dung:

1. Stream Creation

```
// From collection
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> stream1 = list.stream();

// From array
String[] array = {"a", "b", "c"};
Stream<String> stream2 = Arrays.stream(array);

// Stream.of()
Stream<String> stream3 = Stream.of("a", "b", "c");

// Stream.generate()
Stream<Double> randoms = Stream.generate(Math::random).limit(5);

// Stream.iterate()
Stream<Integer> numbers = Stream.iterate(0, n -> n + 2).limit(10);

// IntStream, LongStream, DoubleStream
IntStream intStream = IntStream.range(1, 10);
IntStream intStream2 = IntStream.rangeClosed(1, 10);
```

---

2. Intermediate Operations (Lazy)

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eve");

// filter
names.stream()
     .filter(name -> name.startsWith("A"))
     .forEach(System.out::println); // Alice

// map
names.stream()
```

```
      .map(String::toUpperCase)
      .forEach(System.out::println); // ALICE, BOB, ...

// flatMap
List<List<Integer>> listOfLists = Arrays.asList(
    Arrays.asList(1, 2),
    Arrays.asList(3, 4),
    Arrays.asList(5, 6)
);

listOfLists.stream()
           .flatMap(List::stream)
           .forEach(System.out::println); // 1, 2, 3, 4, 5, 6

// distinct
Stream.of(1, 2, 2, 3, 3, 3)
      .distinct()
      .forEach(System.out::println); // 1, 2, 3

// sorted
names.stream()
     .sorted()
     .forEach(System.out::println);

// sorted with comparator
names.stream()
     .sorted(Comparator.reverseOrder())
     .forEach(System.out::println);

// peek (for debugging)
names.stream()
     .filter(name -> name.length() > 3)
     .peek(name -> System.out.println("Filtered: " + name))
     .map(String::toUpperCase)
     .peek(name -> System.out.println("Mapped: " + name))
     .collect(Collectors.toList());

// limit & skip
names.stream()
     .skip(2)    // Skip first 2
     .limit(2)   // Take next 2
     .forEach(System.out::println);
```

---

3. Terminal Operations (Eager)

```
// forEach
names.forEach(System.out::println);

// count
long count = names.stream()
```

```java
                     .filter(name -> name.length() > 3)
                     .count();

// collect
List<String> filtered = names.stream()
                              .filter(name -> name.startsWith("A"))
                              .collect(Collectors.toList());

// reduce
int sum = IntStream.range(1, 6)
                    .reduce(0, (a, b) -> a + b); // 15

Optional<Integer> max = Stream.of(1, 5, 3, 9, 2)
                               .reduce(Integer::max); // 9

// min, max
Optional<String> shortest = names.stream()
                                  .min(Comparator.comparing(String::length));

Optional<String> longest = names.stream()
                                 .max(Comparator.comparing(String::length));

// anyMatch, allMatch, noneMatch
boolean hasLongName = names.stream()
                           .anyMatch(name -> name.length() > 5); // true

boolean allShort = names.stream()
                        .allMatch(name -> name.length() < 10); // true

boolean noZ = names.stream()
                   .noneMatch(name -> name.contains("Z")); // true

// findFirst, findAny
Optional<String> first = names.stream()
                               .filter(name -> name.startsWith("C"))
                               .findFirst();

Optional<String> any = names.parallelStream()
                             .filter(name -> name.startsWith("C"))
                             .findAny();

// toArray
String[] array = names.stream()
                      .toArray(String[]::new);
```

---

4. Collectors

```java
// toList, toSet
List<String> list = names.stream().collect(Collectors.toList());
Set<String> set = names.stream().collect(Collectors.toSet());
```

```
// joining
String joined = names.stream()
                     .collect(Collectors.joining(", ")); // Alice, Bob, Charlie,
...

String withPrefixSuffix = names.stream()
                               .collect(Collectors.joining(", ", "[", "]"));
// [Alice, Bob, Charlie, ...]

// counting
long count = names.stream()
                  .collect(Collectors.counting());

// summing, averaging
int totalLength = names.stream()
                       .collect(Collectors.summingInt(String::length));

double avgLength = names.stream()
                        .collect(Collectors.averagingInt(String::length));

// summarizing
IntSummaryStatistics stats = names.stream()

.collect(Collectors.summarizingInt(String::length));
System.out.println("Max: " + stats.getMax());
System.out.println("Min: " + stats.getMin());
System.out.println("Avg: " + stats.getAverage());

// groupingBy
Map<Integer, List<String>> byLength = names.stream()
    .collect(Collectors.groupingBy(String::length));
// {3=[Bob, Eve], 5=[Alice, David], 7=[Charlie]}

// groupingBy with downstream collector
Map<Integer, Long> countByLength = names.stream()
    .collect(Collectors.groupingBy(String::length, Collectors.counting()));
// {3=2, 5=2, 7=1}

// partitioningBy
Map<Boolean, List<String>> partitioned = names.stream()
    .collect(Collectors.partitioningBy(name -> name.length() > 4));
// {false=[Bob, Eve], true=[Alice, Charlie, David]}

// toMap
Map<String, Integer> nameToLength = names.stream()
    .collect(Collectors.toMap(
        name -> name,
        String::length
    ));
```

5. Parallel Streams

```
// Sequential
long count1 = IntStream.range(1, 1_000_000)
                       .filter(n -> n % 2 == 0)
                       .count();

// Parallel
long count2 = IntStream.range(1, 1_000_000)
                       .parallel()
                       .filter(n -> n % 2 == 0)
                       .count();

// From collection
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.parallelStream()
       .forEach(System.out::println);
```

Thực hành:

1. Filter, map, collect exercises (30 bài)
2. Grouping & partitioning (10 bài)
3. Reduce operations (10 bài)
4. Complex stream pipelines (15 bài)
5. Performance comparison: sequential vs parallel

▧ BUỔI 5: JAVA 8 (PART 3) - DATETIME API & OPTIONAL (2h)

Mục tiêu:

Modern date/time handling & null safety

Nội dung:

1. DateTime API (java.time)

```
// LocalDate
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1990, Month.JANUARY, 15);
LocalDate parsed = LocalDate.parse("2025-01-15");

// Operations
LocalDate tomorrow = today.plusDays(1);
LocalDate nextWeek = today.plusWeeks(1);
LocalDate nextMonth = today.plusMonths(1);

// Comparisons
```

```java
boolean isBefore = birthday.isBefore(today);
boolean isAfter = birthday.isAfter(today);

// LocalTime
LocalTime now = LocalTime.now();
LocalTime specific = LocalTime.of(14, 30, 0);

// LocalDateTime
LocalDateTime dateTime = LocalDateTime.now();
LocalDateTime specific2 = LocalDateTime.of(2025, 1, 15, 14, 30);

// ZonedDateTime
ZonedDateTime zonedNow = ZonedDateTime.now();
ZonedDateTime tokyo = ZonedDateTime.now(ZoneId.of("Asia/Tokyo"));

// Period (date-based)
Period period = Period.between(birthday, today);
System.out.println(period.getYears() + " years old");

// Duration (time-based)
Duration duration = Duration.between(
    LocalTime.of(9, 0),
    LocalTime.of(17, 30)
);
System.out.println(duration.toHours() + " hours");

// Formatting
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
String formatted = today.format(formatter);
LocalDate parsed2 = LocalDate.parse("15/01/2025", formatter);
```

---

2. Optional

```java
// Creating Optional
Optional<String> empty = Optional.empty();
Optional<String> nonEmpty = Optional.of("Hello");
Optional<String> nullable = Optional.ofNullable(getValue()); // may be null

// Checking value
if (optional.isPresent()) {
    System.out.println(optional.get());
}

// Java 11+
if (optional.isEmpty()) {
    // handle empty
}

// ifPresent
optional.ifPresent(value -> System.out.println(value));
```

```java
// orElse
String value = optional.orElse("default");

// orElseGet (lazy)
String value2 = optional.orElseGet(() -> computeDefault());

// orElseThrow
String value3 = optional.orElseThrow(() -> new RuntimeException("No value"));

// map
Optional<Integer> length = optional.map(String::length);

// flatMap
Optional<String> result = optional.flatMap(this::findRelated);

// filter
Optional<String> filtered = optional.filter(s -> s.length() > 5);

// ifPresentOrElse (Java 9+)
optional.ifPresentOrElse(
    value -> System.out.println("Found: " + value),
    () -> System.out.println("Not found")
);

// or (Java 9+)
Optional<String> result = optional.or(() -> Optional.of("alternative"));
```

Thực hành:

1. DateTime calculations (age, days between, etc.)
2. Timezone conversions
3. Formatting & parsing
4. Replace null checks with Optional
5. Optional chaining exercises

▧ BUỔI 6: JAVA 9-10 (2h)

Mục tiêu:

Modules, var, và collection improvements

Nội dung:

JAVA 9:

1. Module System (Project Jigsaw)

```java
// module-info.java
module com.example.myapp {
    requires java.sql;
    requires java.logging;

    exports com.example.myapp.api;

    opens com.example.myapp.model to com.fasterxml.jackson.databind;
}
```

2. JShell (REPL)

```
$ jshell
jshell> int x = 10
x ==> 10
jshell> x + 5
$2 ==> 15
jshell> /exit
```

3. Collection Factory Methods

```java
// Before Java 9
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");
List<String> immutable = Collections.unmodifiableList(list);

// Java 9
List<String> list = List.of("A", "B", "C"); // Immutable
Set<String> set = Set.of("A", "B", "C");
Map<String, Integer> map = Map.of(
    "A", 1,
    "B", 2,
    "C", 3
);

// Map.ofEntries for more than 10 entries
Map<String, Integer> largeMap = Map.ofEntries(
    Map.entry("A", 1),
    Map.entry("B", 2),
    Map.entry("C", 3)
);
```

4. Stream API Enhancements

```java
// takeWhile
Stream.of(1, 2, 3, 4, 5, 6)
      .takeWhile(n -> n < 4)
      .forEach(System.out::println); // 1, 2, 3

// dropWhile
Stream.of(1, 2, 3, 4, 5, 6)
      .dropWhile(n -> n < 4)
      .forEach(System.out::println); // 4, 5, 6

// ofNullable
Stream<String> stream = Stream.ofNullable(getValue()); // Won't throw NPE

// iterate with predicate
Stream.iterate(1, n -> n <= 10, n -> n + 1)
      .forEach(System.out::println); // 1 to 10
```

5. Optional Enhancements

```java
// stream()
Optional.of("Hello")
        .stream()
        .forEach(System.out::println);

// or()
Optional<String> result = optional.or(() -> Optional.of("alternative"));

// ifPresentOrElse()
optional.ifPresentOrElse(
    value -> System.out.println(value),
    () -> System.out.println("Empty")
);
```

6. Private Methods in Interfaces

```java
interface MyInterface {
    default void method1() {
        commonLogic();
    }

    default void method2() {
        commonLogic();
    }

    private void commonLogic() {
        System.out.println("Common");
    }
}
```

JAVA 10:

1. Local Variable Type Inference (var)

```java
// Before Java 10
ArrayList<String> list = new ArrayList<>();
Map<String, List<Integer>> map = new HashMap<>();

// Java 10
var list = new ArrayList<String>(); // Type inferred
var map = new HashMap<String, List<Integer>>();

// Works with
var number = 10;
var text = "Hello";
var stream = list.stream();

// Limitations - CANNOT use:
// - Fields
// - Method parameters
// - Method return types
// - Without initializer
// var x; // ERROR
```

2. Unmodifiable Collections

```java
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");

// copyOf - creates unmodifiable copy
List<String> copy = List.copyOf(list);

Set<String> set = Set.copyOf(list);
```

Thực hành:

1. Create simple module project
2. JShell interactive coding
3. Collection factory methods exercises
4. Stream takeWhile/dropWhile

5. var usage patterns & limitations

▧ BUỔI 7: JAVA 11 (LTS) (2h)

Mục tiêu:

HTTP Client, String API, và performance improvements

Nội dung:

1. HTTP Client API (Standard)

```java
// Create client
HttpClient client = HttpClient.newBuilder()
    .version(HttpClient.Version.HTTP_2)
    .connectTimeout(Duration.ofSeconds(10))
    .build();

// Synchronous GET
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://api.example.com/users"))
    .GET()
    .build();

HttpResponse<String> response = client.send(
    request,
    HttpResponse.BodyHandlers.ofString()
);

System.out.println(response.statusCode());
System.out.println(response.body());

// Asynchronous GET
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
      .thenApply(HttpResponse::body)
      .thenAccept(System.out::println)
      .join();

// POST with JSON
String json = "{\"name\":\"John\",\"age\":30}";
HttpRequest postRequest = HttpRequest.newBuilder()
    .uri(URI.create("https://api.example.com/users"))
    .header("Content-Type", "application/json")
    .POST(HttpRequest.BodyPublishers.ofString(json))
    .build();

// Authentication
HttpRequest authRequest = HttpRequest.newBuilder()
    .uri(URI.create("https://api.example.com/data"))
    .header("Authorization", "Bearer " + token)
```

```
       .GET()
       .build();
```

---

2. String API Enhancements

```java
// isBlank() - checks if empty or only whitespace
"   ".isBlank();  // true
"".isBlank();     // true
"a".isBlank();    // false

// lines() - stream of lines
String multiline = "Line 1\nLine 2\nLine 3";
multiline.lines()
         .forEach(System.out::println);

// strip(), stripLeading(), stripTrailing() - better than trim()
"  Hello  ".strip();         // "Hello"
"  Hello  ".stripLeading();  // "Hello  "
"  Hello  ".stripTrailing(); // "  Hello"

// repeat()
"Ha".repeat(3);  // "HaHaHa"
"-".repeat(20);  // "--------------------"
```

---

3. Files Methods

```java
// readString(), writeString()
String content = Files.readString(Path.of("file.txt"));
Files.writeString(Path.of("output.txt"), "Hello World");

// isSameFile()
boolean same = Files.isSameFile(path1, path2);
```

---

4. Collection toArray()

```java
List<String> list = List.of("A", "B", "C");

// Before Java 11
String[] array1 = list.toArray(new String[0]);

// Java 11 - simpler
String[] array2 = list.toArray(String[]::new);
```

---

5. Optional isEmpty()

```java
Optional<String> optional = Optional.ofNullable(getValue());

// Java 11
if (optional.isEmpty()) {
    System.out.println("No value");
}

// Before Java 11
if (!optional.isPresent()) {
    System.out.println("No value");
}
```

---

6. Predicate not()

```java
List<String> names = List.of("Alice", "", "Bob", "", "Charlie");

// Before
names.stream()
     .filter(s -> !s.isBlank())
     .forEach(System.out::println);

// Java 11
names.stream()
     .filter(Predicate.not(String::isBlank))
     .forEach(System.out::println);
```

---

7. Lambda Parameter var

```java
// Can use var in lambda parameters
BiFunction<Integer, Integer, Integer> add = (var a, var b) -> a + b;

// Useful with annotations
(@NonNull var x, @Nullable var y) -> x + y
```

Thực hành:

1. HTTP Client - GET/POST requests
2. Parse JSON response
3. String API exercises

4. File read/write operations
5. Stream with Predicate.not()

📓 BUỔI 8: JAVA 12-16 (2h)

Mục tiêu:

Switch expressions, text blocks, records

Nội dung:

JAVA 12-13: Switch Expressions

Old Switch (Statement):

```java
int numLetters;
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numLetters = 6;
        break;
    case TUESDAY:
        numLetters = 7;
        break;
    default:
        numLetters = -1;
}
```

Java 12-13: Switch Expression

```java
// Arrow syntax (no fall-through)
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY -> 7;
    case THURSDAY, SATURDAY -> 8;
    case WEDNESDAY -> 9;
    default -> -1;
};

// With blocks
int result = switch (day) {
    case MONDAY -> {
        System.out.println("Start of week");
        yield 1;
    }
    case FRIDAY -> {
        System.out.println("End of week");
```

```
            yield 5;
        }
        default -> 0;
    };

    // Traditional switch with yield (Java 13)
    int value = switch (day) {
        case MONDAY:
        case FRIDAY:
            yield 6;
        case TUESDAY:
            yield 7;
        default:
            yield -1;
    };
```

JAVA 13-15: Text Blocks

```
// Before - ugly
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello</p>\n" +
              "    </body>\n" +
              "</html>\n";

// Java 13-15 - beautiful
String html = """
              <html>
                  <body>
                      <p>Hello</p>
                  </body>
              </html>
              """;

// JSON example
String json = """
              {
                  "name": "John",
                  "age": 30,
                  "city": "New York"
              }
              """;

// SQL example
String sql = """
             SELECT id, name, email
             FROM users
             WHERE age > 18
             ORDER BY name
             """;
```

```
// Escape sequences
String text = """
              Line 1 \
              Line 2
              """; // Line 1 Line 2 (backslash escapes newline)
```

JAVA 14-16: Records

```
// Before - verbose POJO
public class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public int getAge() { return age; }

    @Override
    public boolean equals(Object o) { /* ... */ }
    @Override
    public int hashCode() { /* ... */ }
    @Override
    public String toString() { /* ... */ }
}

// Java 14-16 - concise record
public record Person(String name, int age) {}

// Usage
Person person = new Person("Alice", 30);
System.out.println(person.name());  // Alice
System.out.println(person.age());   // 30
System.out.println(person);         // Person[name=Alice, age=30]

// Records can have:
// - Methods
// - Static fields
// - Static methods
// - Compact constructor

public record Person(String name, int age) {
    // Compact constructor
    public Person {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
```

```java
            }
        }

        // Instance method
        public boolean isAdult() {
            return age >= 18;
        }

        // Static method
        public static Person of(String name, int age) {
            return new Person(name, age);
        }
    }
```

JAVA 14-16: Pattern Matching for instanceof

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

// Java 14-16
if (obj instanceof String str) {
    System.out.println(str.length());
}

// More complex
if (obj instanceof String str && str.length() > 5) {
    System.out.println("Long string: " + str);
}
```

JAVA 16: Stream.toList()

```java
List<String> names = List.of("Alice", "Bob", "Charlie");

// Before
List<String> filtered = names.stream()
                             .filter(name -> name.startsWith("A"))
                             .collect(Collectors.toList());

// Java 16
List<String> filtered = names.stream()
                             .filter(name -> name.startsWith("A"))
                             .toList(); // Returns unmodifiable list
```

Thực hành:

1. Convert old switch → switch expressions
2. Refactor string concatenation → text blocks
3. Convert POJOs → records
4. Pattern matching instanceof exercises
5. Stream.toList() usage

▧ BUỔI 9: JAVA 17 (LTS) (2h)

Mục tiêu:

Sealed classes và pattern matching

Nội dung:

1. Sealed Classes

```java
// Define sealed class hierarchy
public sealed class Shape
    permits Circle, Rectangle, Triangle {}

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

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

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

public final class Rectangle extends Shape {
    private final double width, height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double area() {
        return width * height;
    }
}

public non-sealed class Triangle extends Shape {
    // Triangle can be extended
}

// Usage - compiler knows all possible subclasses
public double calculateArea(Shape shape) {
    return switch (shape) {
        case Circle c -> Math.PI * c.radius() * c.radius();
        case Rectangle r -> r.width() * r.height();
        case Triangle t -> /* calculate */;
        // No default needed - compiler knows all cases
    };
}
```

Sealed Interfaces:

```java
public sealed interface Payment
    permits CreditCardPayment, CashPayment, CryptoPayment {}

public final class CreditCardPayment implements Payment { }
public final class CashPayment implements Payment { }
public record CryptoPayment(String walletAddress) implements Payment { }
```

---

2. Pattern Matching for switch (Preview)

```java
// Type patterns
public 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);
        default -> obj.toString();
    };
}

// Guarded patterns
public String classify(int number) {
    return switch (number) {
        case int n when n < 0 -> "negative";
        case int n when n == 0 -> "zero";
        case int n when n > 0 -> "positive";
        default -> "unknown";
    };
}

// Null handling
public String handleNull(String str) {
    return switch (str) {
        case null -> "null value";
        case String s when s.isEmpty() -> "empty";
        case String s -> s.toUpperCase();
    };
}
```

---

3. Helpful NullPointerExceptions

```
// Before Java 14: Cryptic message
// Exception: NullPointerException at line 10

// Java 17: Detailed message
String name = person.getAddress().getCity().getName();
// Exception: Cannot invoke "City.getName()" because the return value
// of "Address.getCity()" is null
```

---

4. Random Generator Enhancements

```
// New RandomGenerator interface
RandomGenerator random = RandomGenerator.of("L64X128MixRandom");

// Thread-safe random
random.ints(10).forEach(System.out::println);

// Different algorithms
RandomGenerator legacy = RandomGenerator.of("Random");
RandomGenerator secure = RandomGenerator.of("SecureRandom");
```

Thực hành:

1. Create sealed class hierarchies (3 examples)
2. Pattern matching switch exercises
3. Refactor instanceof chains → switch
4. Sealed interfaces implementation
5. Complex pattern matching scenarios

▧ BUỔI 10: JAVA 18-20 (2h)

Mục tiêu:

Preview features overview

Nội dung:

JAVA 18:

1. Simple Web Server (jwebserver)

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

2. Code Snippets in JavaDoc

```
/**
 * Example usage:
 * {@snippet :
 *   List<String> list = List.of("A", "B", "C");
 *   list.forEach(System.out::println);
 * }
 */
```

JAVA 19:

1. Virtual Threads (Preview)

```
// Traditional thread
Thread thread = new Thread(() -> {
    System.out.println("Hello from thread");
});
thread.start();

// Virtual thread (Java 19 Preview)
Thread vThread = Thread.startVirtualThread(() -> {
    System.out.println("Hello from virtual thread");
});

// ExecutorService with virtual threads
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    for (int i = 0; i < 10_000; i++) {
        executor.submit(() -> {
            // Task
        });
    }
} // Auto-close waits for all tasks
```

2. Record Patterns (Preview)

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

// Pattern matching with records
public void process(Object obj) {
    if (obj instanceof Point(int x, int y)) {
        System.out.println("Point: " + x + ", " + y);
    }
}

// Switch with record patterns
public int sum(Object obj) {
    return switch (obj) {
        case Point(int x, int y) -> x + y;
        default -> 0;
    };
}
```

JAVA 20:

1. Scoped Values (Preview)

```java
// Alternative to ThreadLocal
final static ScopedValue<User> CURRENT_USER = ScopedValue.newInstance();

public void process() {
    User user = loadUser();
    ScopedValue.where(CURRENT_USER, user).run(() -> {
        // CURRENT_USER available in this scope
        businessLogic();
    });
}
```

Thực hành:

1. Explore Simple Web Server
2. Virtual threads examples
3. Record patterns exercises
4. Compare ThreadLocal vs ScopedValues

▧ BUỔI 11: JAVA 21 (LTS) (2h)

Mục tiêu:

Virtual threads và sequenced collections

Nội dung:

1. Virtual Threads (Final)

```java
// Create virtual thread
Thread vThread = Thread.ofVirtual()
                       .name("worker")
                       .start(() -> {
                           System.out.println("Task");
                       });

// Virtual thread executor
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    // Submit 1 million tasks easily!
    for (int i = 0; i < 1_000_000; i++) {
        executor.submit(() -> {
            // I/O bound task
            fetchDataFromAPI();
        });
    }
}

// Virtual thread factory
ThreadFactory factory = Thread.ofVirtual().factory();

// Structured Concurrency
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Future<String> user = scope.fork(() -> fetchUser());
    Future<Integer> order = scope.fork(() -> fetchOrder());

    scope.join();           // Wait for all
    scope.throwIfFailed();  // Throw if any failed

    // Both completed successfully
    processResults(user.resultNow(), order.resultNow());
}
```

---

2. Sequenced Collections

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

// Usage with List
List<String> list = new ArrayList<>(List.of("A", "B", "C"));
list.addFirst("Z");        // [Z, A, B, C]
list.addLast("D");         // [Z, A, B, C, D]
String first = list.getFirst();  // Z
String last = list.getLast();    // D

List<String> reversed = list.reversed(); // [D, C, B, A, Z]

// LinkedHashSet now has ordering
LinkedHashSet<String> set = new LinkedHashSet<>();
set.add("First");
set.add("Second");
set.addFirst("Zero");   // [Zero, First, Second]

// LinkedHashMap
LinkedHashMap<String, Integer> map = new LinkedHashMap<>();
map.put("A", 1);
map.put("B", 2);
map.putFirst("Z", 0);  // {Z=0, A=1, B=2}
```

---

3. Record Patterns (Final)

```java
record Point(int x, int y) {}
record ColoredPoint(Point point, String color) {}

// Nested patterns
public void printColoredPoint(Object obj) {
    if (obj instanceof ColoredPoint(Point(int x, int y), String color)) {
        System.out.println("Point(" + x + ", " + y + ") in " + color);
    }
}

// Switch with patterns
public String describe(Object obj) {
    return switch (obj) {
        case Point(int x, int y) -> "Point at " + x + ", " + y;
        case ColoredPoint(Point(int x, int y), String color) ->
            color + " point at " + x + ", " + y;
        case String s -> "String: " + s;
        case null -> "null";
        default -> "Unknown";
    };
}
```

---

4. Pattern Matching for switch (Final)

```java
// Comprehensive switch
public String processShape(Shape shape) {
    return switch (shape) {
        case null -> "No shape";
        case Circle c when c.radius() > 10 -> "Large circle";
        case Circle c -> "Small circle: " + c.radius();
        case Rectangle r when r.width() == r.height() -> "Square";
        case Rectangle r -> "Rectangle: " + r.width() + "x" + r.height();
        case Triangle t -> "Triangle";
    };
}
```

---

5. String Templates (Preview)

```java
// String interpolation
String name = "Alice";
int age = 30;

// Traditional
String msg1 = "Hello " + name + ", you are " + age + " years old";

// String template (Preview)
String msg2 = STR."Hello \{name}, you are \{age} years old";

// With expressions
String msg3 = STR."Next year you'll be \{age + 1}";

// Multi-line
String html = STR."""
    <html>
        <body>
            <h1>Welcome \{name}</h1>
            <p>Age: \{age}</p>
        </body>
    </html>
    """;
```

Thực hành:

1. Virtual threads - create millions of threads
2. Sequenced collections operations
3. Record patterns - nested matching
4. Pattern switch - complex scenarios
5. Performance comparison: virtual vs platform threads

🔯 BUỔI 12: JAVA 22-25 & FUTURE (2h)

Mục tiêu:

Latest features và future direction

Nội dung:

JAVA 22:

1. Unnamed Variables & Patterns

```java
// Before - unused variables
try {
    int result = compute();
} catch (Exception e) {  // e unused
    System.out.println("Error occurred");
}

// Java 22
try {
    int result = compute();
} catch (Exception _) {  // _ = don't care
    System.out.println("Error occurred");
}

// Pattern matching
if (obj instanceof Point(int x, int _)) {
    // Only care about x, not y
    System.out.println("x = " + x);
}

// Lambda
list.forEach(_ -> System.out.println("Item"));
```

2. Statements before super() (Preview)

```java
public class Child extends Parent {
    private final int value;

    public Child(int input) {
        // Can have statements before super()!
        if (input < 0) {
            throw new IllegalArgumentException();
        }
        this.value = input * 2;
        super(input);
    }
}
```

JAVA 23:

1. Markdown Documentation

```java
/// This is a **markdown** comment!
///
/// # Features
/// - Bullet points
/// - `Code formatting`
/// - **Bold** and *italic*
///
/// ```java
/// Example code = new Example();
/// ```
public class Example {
}
```

2. Module Import

```java
// Before
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.ArrayList;
import java.util.HashMap;

// Java 23 (Preview)
import module java.base;
// Imports everything from java.base module
```

JAVA 24:

1. Stream Gatherers (Preview)

```java
// Custom stream operations
List<String> result = Stream.of("a", "b", "c", "d", "e")
    .gather(Gatherers.windowFixed(2))  // Sliding window
    .toList();
// [[a, b], [c, d], [e]]
```

```java
// Custom gatherer
List<Integer> result = Stream.of(1, 2, 3, 4, 5)
    .gather(Gatherers.scan(() -> 0, (a, b) -> a + b))  // Running sum
    .toList();
// [1, 3, 6, 10, 15]
```

2. Structured Concurrency (Final)

```java
// Better than CompletableFuture for structured tasks
try (var scope = new StructuredTaskScope.ShutdownOnSuccess<String>()) {
    Subtask<String> task1 = scope.fork(() -> fetchFromAPI1());
    Subtask<String> task2 = scope.fork(() -> fetchFromAPI2());

    scope.join();  // Wait for first success

    String result = scope.result();  // Get successful result
}
```

JAVA 25 (Current):

1. Flexible Constructor Bodies (Preview)

```java
public record Range(int start, int end) {
    // Can now initialize fields before canonical constructor
    public Range {
        if (start > end) {
            // Swap
            int temp = start;
            start = end;
            end = temp;
        }
        // start and end assigned automatically
    }
}
```

2. Primitive Patterns (Preview)

```java
// Pattern matching for primitives
int value = 42;

String result = switch (value) {
    case int i when i < 0 -> "Negative";
    case int i when i == 0 -> "Zero";
    case int i when i > 0 -> "Positive";
};
```

FUTURE ROADMAP:

Project Valhalla (Value Types)

```
// Inline classes - no object overhead
public inline class Point {
    private final int x;
    private final int y;
    // Stored directly, no heap allocation
}
```

Project Loom Enhancements

- Even lighter virtual threads
- Better structured concurrency

Pattern Matching Evolution

- Pattern matching for arrays
- Pattern matching for primitives
- Destructuring assignments

Foreign Function & Memory API

```
// Native code without JNI
SymbolLookup stdlib = Linker.nativeLinker().defaultLookup();
MemorySegment strlen = stdlib.find("strlen").get();
// Call C functions directly
```

Thực hành:

1. Unnamed patterns usage
2. Stream gatherers examples
3. Structured concurrency scenarios
4. Compare old vs new APIs

📊 TỔNG KẾT & LỘ TRÌNH ÁP DỤNG

Phiên bản nên dùng Production:

Version	Status	Use When
Java 8	Legacy LTS	Legacy projects, wide compatibility
Java 11	LTS (Older)	Stable, well-tested, conservative choice
Java 17	LTS (Recommended)	Modern features + stability
Java 21	LTS (Latest)	Virtual threads, latest features
Java 25	Non-LTS	Cutting edge, preview features

Migration Path:

```
Java 8
  ↓ (Add modules, var, HTTP Client)
Java 11
  ↓ (Add records, text blocks, switch expressions)
Java 17
  ↓ (Add virtual threads, sequenced collections)
Java 21
  ↓ (Add stream gatherers, flexible constructors)
Java 25
```

Feature Adoption Priority:

HIGH PRIORITY (Use immediately):

1. Lambda & Stream API (Java 8)
2. Optional (Java 8)
3. DateTime API (Java 8)
4. var keyword (Java 10)
5. Collection factory methods (Java 9)
6. Text blocks (Java 15)
7. Records (Java 16)
8. Pattern matching instanceof (Java 16)
9. Switch expressions (Java 14)
10. HTTP Client (Java 11)

MEDIUM PRIORITY (Gradual adoption):

1. Sealed classes (Java 17)
2. Pattern matching switch (Java 21)
3. Record patterns (Java 21)
4. Sequenced collections (Java 21)
5. Virtual threads (Java 21)

LOW PRIORITY (Advanced scenarios):

1. Modules (Java 9)

2. Structured concurrency (Java 24)
3. Stream gatherers (Java 24)
4. Primitive patterns (Java 25)

📚 TÀI LIỆU THAM KHẢO

Official:

- OpenJDK Release Notes
- Oracle Java Documentation
- JEP (JDK Enhancement Proposals) Index

Books:

- "Modern Java in Action" - Raoul-Gabriel Urma
- "Java: The Complete Reference" - Herbert Schildt (Latest Edition)

Online:

- Inside Java (inside.java)
- Baeldung Java Tutorials
- Java Magazine (Oracle)

🎯 BÀI TẬP THỰC HÀNH (200 bài)

Module	Số Bài
Java 7	10
Java 8 (Lambda, Stream, DateTime)	80
Java 9-10	15
Java 11	20
Java 12-16	30
Java 17	20
Java 18-20	10
Java 21	15
TỔNG	**200**

GOOD LUCK! Stay modern with Java! ☕ 🚀