Java 21 new features samples

1. Pattern Matching for switch

This feature extends the capabilities of switch expressions and statements, allowing you to express more complex and sophisticated data pattern matching

```
// Define the Shape interface
interface Shape {}

// Circle class implementing Shape interface
class Circle implements Shape {
    private double radius;

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

    public double getRadius() {
        return radius;
    }
}

// Rectangle class implementing Shape interface
class Rectangle implements Shape {
    private double width;
    private double height;

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

    public double getWidth() {
        return width;
    }

    public double getHeight() {
        return height;
    }
}

// Main class
public class ShapeDescriber {

    // describeShape method
    public static String describeShape(Object shape) {
        return switch (shape) {
```

```
            case Circle c -> "Circle with radius " + c.getRadius();
            case Rectangle r -> "Rectangle with width " + r.getWidth() + " and height " + r.ge
            case null -> "It's null";
            default -> "Some other shape";
        };
    }

    // Main method to test the describeShape function
    public static void main(String[] args) {
        Circle circle = new Circle(5.0);
        Rectangle rectangle = new Rectangle(4.0, 6.0);

        System.out.println(describeShape(circle)); // Prints description of the circle
        System.out.println(describeShape(rectangle)); // Prints description of the rectangle
        System.out.println(describeShape(null)); // Prints null case
        System.out.println(describeShape("String")); // Example for default case
    }
}
```

Pattern matching in switch can be combined with '**guards**' (additional boolean conditions) to refine matching criteria

```
enum TrafficLight { RED, YELLOW, GREEN }

String handleMessage(TrafficLight light) {
    return switch (light) {
        case RED     -> "Stop!";
        case YELLOW  -> "Prepare to stop";
        case GREEN   -> "Go!";
        default      -> "Unknown signal"; // Should rarely happen
    };
}
```

Pattern matching offers powerful data extraction mechanisms in switch expressions and statements

```
record Rectangle(int width, int height) {}
record Circle(int radius) {}
record Triangle(int base, int height) {}

String calculateArea(Shape shape) {
    return switch (shape) {
        case Rectangle(int w, int h) -> "Rectangle area: " + (w * h);
        case Circle(int r)           -> "Circle area: " + (Math.PI * r * r);
        case Triangle(int b, int h)  -> "Triangle area: " + (0.5 * b * h);
        default                      -> "Unsupported shape";
    };
}
```

Let's craft a **more complex example** to demonstrate the expressive power of **pattern matching in switch within Java 21**

Scenario: Parsing Geometric Shapes

Suppose we have geometric shapes represented with a hierarchy of sealed classes and records:

```
sealed interface Shape permits Circle, Rectangle, Triangle, ComplexShape {}

record Circle(double radius) implements Shape {}
record Rectangle(double width, double height) implements Shape {}
record Triangle(double base, double height) implements Shape {}
record ComplexShape(List<Shape> subShapes) implements Shape {}
```

Goal: Calculate the combined area of shapes with intricate structures:

```
public static double calculateTotalArea(Shape shape) {
    return switch (shape) {
        case Circle(double radius) -> Math.PI * radius * radius;
        case Rectangle(double width, double height) -> width * height;
        case Triangle(double base, double height) -> 0.5 * base * height;
        case ComplexShape(List<Shape> subShapes) ->
            subShapes.stream()
                     .mapToDouble(ShapeUtils::calculateTotalArea) // Recursive call
                     .sum();
        default -> throw new IllegalArgumentException("Unknown shape type");
    };
}
```

Key Points

Sealed Hierarchy: The sealed keyword ensures all possible shape types are known (enhancing pattern matching safety).

Type Patterns: We match against specific types (Circle, Rectangle, etc.)

Record Patterns: Deconstruct records instantly to extract components (radius, width, height).

Nested Patterns: The ComplexShape case shows how patterns can be nested to handle recursive structures.

Guarded Patterns: We could add guards to patterns (e.g., case Triangle(double base, double height) when base > 0) for even finer-grained matching.

2. Record Patterns

Record patterns streamline working with record types, providing concise syntax for deconstructing them

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

void movePoint(Object obj) {
    if (obj instanceof Point(int x, int y)) {
        // x and y are in scope here
        System.out.println("Coordinates: (" + x + ", " + y + ")");
    }
}
```

Record patterns can be **nested** to extract data from more complex record hierarchies:

```
record Customer(String name, Address address) {}
record Address(String street, String city, String zipCode) {}

void processCustomer(Customer customer) {
    if (customer instanceof Customer(String name, Address(String _, String city, _))) {
        if (city.equals("New York")) {
            System.out.println("Customer " + name + " is from New York.");
        }
    }
}
```

Let's dive into more **advanced and complex** scenarios demonstrating the power of **Record Patterns in Java 21**

Scenario 1: Deeply Nested Data Structures

Imagine we're working with a user information structure nested several levels deep:

```
record Address(String street, String city, String zipCode) {}
record Customer(String name, Address address) {}
record Order(int orderId, Customer customer, List<LineItem> items) {}
record LineItem(String product, int quantity) {}
```

Get Cities of Customers With Large Orders

```
void processOrders(List<Order> orders) {
    Set<String> citiesWithLargeOrders = orders.stream()
                                              .filter(order -> largeOrder(order))
                                              .flatMap(order -> extractCities(order).stream())
                                              .collect(Collectors.toSet());

    System.out.println(citiesWithLargeOrders);
}
```

```
private boolean largeOrder(Order order) {
    // ... logic to determine if an order is large
    return order.items().stream().mapToInt(LineItem::quantity).sum() > 100;
}

private List<String> extractCities(Order order) {
    if (order instanceof Order(_, Customer(_, Address(_, String city, _)), _)) {
        return List.of(city);
    } else {
        return List.of();
    }
}
```

Scenario 2: Polymorphism & Record Patterns

Consider a scenario where we have an abstract base class, with derived classes implemented as records:

```
abstract class ApiResponse {}

record SuccessResponse(String message, Object data) extends ApiResponse {}
record ErrorResponse(int code, String message) extends ApiResponse {}
```

Selective Data Extraction

```
void handleApiResponse(ApiResponse response) {
    if (response instanceof SuccessResponse(String msg, Object data)) {
        // Handle successful data extraction (knowing the type of 'data')
        System.out.println("Success: " + msg);
        // Process the 'data' object based on its expected type
    } else if (response instanceof ErrorResponse(int code, String errMsg)) {
        // Handle error scenario based on error code and message
        System.out.println("Error (" + code + "): " + errMsg);
    }
}
```

Key Points:

Deep Nesting: Record patterns elegantly deconstruct even several layers deep

Type Specialization: When 'data' in SuccessResponse has a specific type, add it to the pattern for direct access (e.g., SuccessResponse(..., Customer data))

Polymorphism: Combine the power of inheritance and record patterns for specific handling based on concrete types

3. String Templates

String templates enhance Java's string **formatting** capabilities, enabling you to embed expressions directly within strings:

```
int width = 5;
int height = 8;
String message = """
                 Shape dimensions:
                 Width:  %d
                 Height: %d
                 """.formatted(width, height);
System.out.println(message);
```

String templates can handle **rich formatting**, including number formatting and alignment:

```
double price = 129.95;
String formatted = """
                   Invoice
                   Item: Widget XYZ
                   Price: $%,10.2f
                   """.formatted(price);
System.out.println(formatted);
```

Let's explore some advanced string formatting scenarios using features introduced in Java 21.

String Templates

Java 21 features String Templates, a powerful way to **embed expressions** directly into string literals. Let's illustrate:

```
int age = 35;
String name = "Alice";
double temperature = 23.6;

String formattedMessage = `Hello, my name is ${name} and I am ${age} years old.
                           The current temperature is ${temperature} degrees Celsius.`;

System.out.println(formattedMessage);
```

Output:

```
Hello, my name is Alice and I am 35 years old. The current temperature is 23.6 degrees Celsius
Notice how expressions within ${ } are automatically evaluated and embedded within the string.
```

Sequenced Collections

Sequenced Collections provide greater control over the processing of collections in a sequential manner. Let's combine this with string formatting:

```
import java.util.stream.Collectors;
import java.util.stream.Stream;

List<String> fruits = List.of("Apple", "Banana", "Mango", "Orange");

String formattedList = fruits.stream()
        .map(String::toUpperCase)
        .collect(Collectors.joining(", ", "[", "]"));

System.out.println(formattedList);
```

Output:

```
[APPLE, BANANA, MANGO, ORANGE]
```

Explanation:

We create a list of fruits

Using a stream, we convert each fruit to uppercase

The Collectors.joining() method in Sequenced Collections helps us neatly format the elements, adding the prefix "[", suffix "]", and a comma separator

Complex Example: Formatting a Report

Let's craft a more elaborate example:

```
class Product {
    String name;
    double price;
    int quantity;

    // Constructor, getters, setters
}

// ... (Assume Product class is defined)

List<Product> order = new ArrayList<>();
order.add(new Product("Laptop", 999.99, 2));
order.add(new Product("Keyboard", 45.50, 1));

String report = `--- Order Summary ---\n` +
```

```
                order.stream()
                     .map(p -> String.format("%-20s $%8.2f  x%2d\n", p.name, p.price, p.quanti
                     .collect(Collectors.joining());

System.out.println(report);
```

Output:

```
--- Order Summary ---
Laptop               $999.99  x2
Keyboard             $ 45.50  x1
```

Notes:

String Templates make the report structure very readable

String.format is used for detailed control over numeric and columnar formatting

4. Scoped Values

Scoped values introduce a safer way to use resources that need to be automatically closed

Scoped values ensure correct closure, similar to try-with-resources but more versatile

```
// Imagine a hypothetical 'AutoCloseableFile' here
try (var file = new AutoCloseableFile("data.txt")) {
    String content = file.readContent();
    // ... do something with content
}  // 'file' is automatically closed here
```

Scoped values go beyond standard try-with-resources to enable more flexible resource handling

```
void writeToFileAndDatabase(String data) throws IOException {
   try (var fileWriter = new FileWriter("log.txt");
        var dbConnection = database.getConnection()) {

       fileWriter.write(data);
       dbConnection.execute("INSERT INTO logs VALUES (?)", data);
   }
}
```

Let's delve into additional examples demonstrating the capabilities of Scoped Values in Java 21

Scenario 1: Implicit Database Connection Scope

Let's simulate a scenario where we establish a database connection and make it implicitly available to various data access methods

DatabaseConnection.java

```
import java.sql.SQLException;

// Simulated class (you would normally interact with a real database library)
public class DatabaseConnection implements AutoCloseable {

    public DatabaseConnection() {
        System.out.println("Opening database connection");
    }

    // Placeholder for simulating a database query
    public void executeQuery(String query) {
        System.out.println("Executing query: " + query);
    }

    @Override
    public void close() throws SQLException {
        System.out.println("Closing database connection");
    }
}
```

File 2: ScopedDatabase.java

```
public class ScopedDatabase {
    public static void main(String[] args) {
        ScopedValue.run(() -> new DatabaseConnection())
                   .thenRun(ScopedDatabase::fetchData)
                   .thenRun(ScopedDatabase::processData);
    }

    static void fetchData() {
        DatabaseConnection conn = ScopedValue.currentOrNull(DatabaseConnection.class);
        if (conn != null) {
            conn.executeQuery("SELECT * FROM some_table");
        } else {
            System.out.println("Error: No database connection in scope");
        }
    }

    static void processData() {
        DatabaseConnection conn = ScopedValue.currentOrNull(DatabaseConnection.class);
        if(conn != null) {
            // ... Process fetched data (assuming results are available)
            System.out.println("Processing fetched data...");
        } else {
            System.out.println("Error: No database connection in scope");
```

```
            }
        }
    }
```

Explanation

DatabaseConnection.java: A simple simulation of a database connection to demonstrate auto-closing

ScopedDatabase.java: main method launches the execution, establishing a Scoped Value holding the DatabaseConnection

fetchData and processData access the shared database connection via ScopedValue.currentOrNull()

How to Run the application

Save: Create the two files above as DatabaseConnection.java and ScopedDatabase.java

Compile: Ensure you have a suitable Java Development Kit (JDK) version 21 or above. Compile both files from your terminal:

```
javac DatabaseConnection.java ScopedDatabase.java
```

Run: Execute the ScopedDatabase class:

```
java ScopedDatabase
```

Expected Output (Illustrative):

Opening database connection

Executing query: SELECT * FROM some_table

Processing fetched data...

Closing database connection

Scenario 2: Hierarchical Request Data

We can use Scoped Values to model **nested** request information or hierarchical contexts

```
class User {
    String name;
    // ...
}

class HttpRequest {
    String ipAddress;
```

```
        // ...
    }

    public class ScopedRequestData {
        public static void main(String[] args) {
            User user = new User("Alice");
            HttpRequest request = new HttpRequest("10.0.0.1");

            ScopedValue.run(() -> user)
                       .thenCombine( () -> request, ScopedRequestData::handleNestedRequest);
        }

        static void handleNestedRequest(User user, HttpRequest request) {
            System.out.println("User: " + user.name);
            System.out.println("Request IP: " + request.ipAddress);
            // ... other nested processing
        }
    }
```

Notes:

ScopedValue.run() initializes a scope with the User object

thenCombine() creates a nested scope including HttpRequest, combining data for the handleNestedRequest method

Key Points

Versatility: Scoped Values elegantly pass contextual information, manage resources, and represent layered structures

Composability: Scoped Values integrate well with methods like thenRun and thenCombine for complex use cases

5. Structured Concurrency

Structured concurrency simplifies the **management of multiple threads**

Instead of dealing with threads directly, it **treats multiple tasks as a single unit of work** for improved reliability and error handling

```
    ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();

    try (executor) {
        Future<String> task1 = executor.submit(() -> "Result of Task 1");
        Future<Integer> task2 = executor.submit(() -> 42);

        System.out.println(task1.get());
```

```
        System.out.println(task2.get());
    }
```

Structured concurrency offers elegant **error handling for concurrent tasks**

```
    ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();

    try (executor) {
        var results = executor.invokeAll(List.of(
                () -> possiblyFailingTask1(),
                () -> possiblyFailingTask2()
        ));

        // Process results, potentially handle exceptions if any task failed
        for (Future<String> result : results) {
            try {
                System.out.println(result.get());
            } catch (ExecutionException ex) {
                System.out.println("Task failed: " + ex.getCause().getMessage());
            }
        }
    }
```

Structured concurrency radically simplifies working with multiple concurrent tasks within a single unit

Coordinating and Handling Errors Among Tasks:

```
    ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();

    try (executor) {
        Scope scope = new StructuredTaskScope.ShutdownOnFailure();
        Future<String> task1 = scope.fork(() -> performNetworkOperation());
        Future<Integer> task2 = scope.fork(() -> computeExpensiveValue());

        scope.join();           // Wait for both tasks to complete
        scope.throwIfFailed();  // Propagate any exceptions

        System.out.println("Network result: " + task1.resultNow());
        System.out.println("Computed value: " + task2.resultNow());
    } catch (ExecutionException | InterruptedException e) {
        System.out.println("Task failed: " + e.getMessage());
    }
```

Virtual threads introduce lightweight threading for massively scalable applications

Handling Many Concurrent Clients:

```
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();

try (var serverSocket = new ServerSocket(8080)) {
    while (true) {
        try (Socket clientSocket = serverSocket.accept()) {
             executor.submit(() -> handleClientRequest(clientSocket));
        }
    }
}

private void handleClientRequest(Socket clientSocket) {
    // Read request, process, send response...
}
```

6. SequencedCollection

Let's explore how you can leverage Java 21's SequencedCollection interface with a few practical examples

Understanding SequencedCollection

The SequencedCollection interface provides a way to work with collections that have a defined order of elements. Key features include:

Getting First/Last: Directly get the first (getFirst()) and last (getLast()) elements

Adding at Start/End: Add elements to the beginning (addFirst()) or end (addLast()) easily

Removing First/Last: Conveniently remove the first (removeFirst()) or last (removeLast()) elements

Reversing: Create a reversed view of the collection (reversed())

Example 1: Basic Operations with ArrayList

```
import java.util.ArrayList;
import java.util.List;

public class SequencedCollectionDemo {
    public static void main(String[] args) {
        List<String> myList = new ArrayList<>();
        myList.add("Alice");
        myList.add("Bob");
        myList.add("Charlie");

        // SequencedCollection features
        myList.addFirst("Zara");
        myList.addLast("Eve");

        System.out.println("First: " + myList.getFirst());
```

```
        System.out.println("Last: " + myList.getLast());

        // Removal
        myList.removeFirst();
        myList.removeLast();

        System.out.println("Modified List: " + myList);
    }
}
```

Example 2: SequencedSet with LinkedHashSet

```
import java.util.LinkedHashSet;
import java.util.Set;

public class SequencedSetDemo {
    public static void main(String[] args) {
        Set<Integer> mySet = new LinkedHashSet<>();
        mySet.add(5);
        mySet.add(1);
        mySet.add(3);

        System.out.println("First element: " + mySet.getFirst());

        // Reverse the order
        SequencedSet<Integer> reversedSet = mySet.reversed();
        System.out.println("Reversed set: " + reversedSet);
    }
}
```

Important Considerations

Compatibility: Existing collections like ArrayList and LinkedList automatically get these features because they now implement SequencedCollection

Sets: LinkedHashSet implements SequencedSet (a sub-interface), providing ordering in sets

Flexibility: Sequenced collections make tasks like working with the ends of a collection or reversing much more intuitive

7. Unnamed Classes and Instance Main Methods

Since Java 21, we can use unnamed classes and instance main methods that allow us to bootstrap a class with minimal syntax

No class declaration is required

Java keywords public and static are no longer required

Main method argument args is optional

Prior to Java 21, we had to write the following class to execute a simple “Hello, World!” message:

HelloWorld.Java before Java 21

```
package com.company.app;

public class HelloWorld {

  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

Since Java 21, the following code is a fully functional class that will print the output “Hello, World!” to the console

We can store this class in any Java file such as HelloWorld.Java and then we can run it

HelloWorld.Java in Java 21

```
void main() {
  System.out.println("Hello, World!");
}
```

This is a preview language feature in Java 21, disabled by default. Use --enable-preview flag to use this feature

```
$ java.exe --enable-preview --source 21 HelloWorld.Java


// Prints
Hello, World!
```

8. Unnamed Patterns and Variables

These Java 21 features help reduce verbosity and streamline code, mainly in pattern matching scenarios

Unnamed Patterns: In pattern matching (with instanceof or record patterns), sometimes you only care about whether a value matches a particular type but don't need to capture that value in a variable

Unnamed patterns let you write a simple underscore (_) where you would normally place a variable name

Unnamed Variables: The underscore (_) can also be used as a variable name

This signifies that you intentionally discard the value, making it clear you're not using it later in the code

Unnamed Patterns with instanceof:

```
Object obj = "Hello";
if (obj instanceof String _) { // The '_' indicates we don't need a variable
    System.out.println("The object is a String");
}
```

Unnamed Patterns in Record Patterns:

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

void processPoint(Object obj) {
    if (obj instanceof Point(_, int y)) {
        System.out.println("Y-coordinate is: " + y);
    }
}
```

Here, we ignore the x-coordinate of a Point object

Unnamed Variables to Eliminate Warnings:

```
for (var _ : myList) {
    // Do something, but don't use the loop variable
}
```

Using _ prevents compiler warnings about an unused variable

Key Benefits

Cleaner Code: Unnamed patterns and variables reduce visual clutter and shift focus to relevant code sections

Compiler Assistance: The compiler ensures you're not accidentally losing value by not naming a pattern or variable

Single Responsibility: This feature nicely complements record patterns and the instanceof operator to encourage the Single Responsibility Principle (making every part of your code have a clear purpose).

Points to Note

Naming Restrictions: Using _ as a variable or pattern name was technically invalid in older Java versions. Java 21 leverages this to introduce the feature seamlessly

Feature State: Consider these features as "preview" in Java 21. This means they might change in future versions based on feedback

9. Virtual Threads

What are Virtual Threads?

Lightweight Threads: Virtual threads are a fundamental change in how Java handles concurrency

They are threads managed primarily by the Java runtime rather than directly by the operating system

This makes them significantly cheaper to create, sustain, and switch between compared to traditional OS-level threads

Simplified Concurrency: Virtual threads aim to make writing highly concurrent applications in Java less complicated and resource-intensive

You gain the benefits of multi-threading without worrying as much about thread pool management and resource limitations

Why They Matter?

Scalability: Virtual threads empower you to write applications that can handle massive numbers of concurrent operations (especially I/O bound operations) without running out of memory or overloading the system

Simplified Code: Code written with virtual threads often looks synchronous even though it's asynchronous – fewer callbacks and complex threading logic make the code easier to write and reason about

Key Concepts

Creation: You start them using Thread.startVirtualThread()

Schedulers: The Java runtime automatically schedules virtual threads onto a small pool of operating system threads (often called carrier threads)

Blocking: When a virtual thread performs a blocking operation (like file or network I/O), the runtime can "park" the virtual thread and run others, rather than blocking the entire carrier thread

Important Notes

Not for Everything: Virtual threads are mainly beneficial for I/O-bound workloads. CPU-intensive tasks still might be better suited for traditional threads

Benefits in a Nutshell

Virtual threads enable you to write Java applications that:

Handle many more concurrent connections/requests with a smaller resource footprint

Can maintain cleaner, more synchronous-looking code, especially in scenarios with network or file I/O

Basic Execution:

```
Thread.startVirtualThread(() -> {
    System.out.println("Hello from a virtual thread!");
}).join();
```

Blocking I/O:

```
public class SimpleServer {
    public static void main(String[] args) {
        try (var serverSocket = new ServerSocket(8080)) {
            while (true) {
                try (var clientSocket = serverSocket.accept()) {
                    // Create a virtual thread to handle each client request
                    Thread.startVirtualThread(() -> {
                        System.out.println("New client connected!");
                        // ... (code to process the client's request)
                    });
                }
            }
        } catch (IOException e) {
            System.out.println("Error handling client: " + e.getMessage());
        }
    }
}
```

Explanation:

This code creates a very basic server

The main loop waits for incoming client connections

For each new connection, a virtual thread is launched to handle the client's request, allowing the server to manage more clients simultaneously