Recods for immutable data carriers

1. What is a Record in Java?

- Introduced in **Java 14 (preview)** and standardized in **Java 16**.
- A **record** is a special kind of class meant to **store immutable data**.
- Automatically provides:
    - `private final` fields
    - `getter` methods
    - `equals()`, `hashCode()`, `toString()`
- Cannot have mutable state (fields are final by default).

```
// Traditional class
public class User {
    private final String name;
    private final int age;

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

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

    @Override
    public String toString() {
        return "User[name=" + name + ", age=" + age + "]";
    }
}

// Record (immutable data carrier)
public record UserRecord(String name, int age) {}
```

The **record automatically provides**:

```
UserRecord user = new UserRecord("Alice", 30);
System.out.println(user.name()); // Alice
System.out.println(user.age());  // 30
System.out.println(user);        // UserRecord[name=Alice, age=30]
```

Sealed class

A **sealed class** is a class that **restricts which other classes can extend it**.

- Introduced in **Java 17**.
- It's useful when you want **controlled inheritance**, i.e., you know all possible subclasses at compile-time.
- Helps **model fixed hierarchies** (like enums but with richer data).

Syntax

```
// Sealed class
public sealed class Shape permits Circle, Rectangle {
    // Common fields or methods
}

// Allowed subclasses
public final class Circle extends Shape {
    private double radius;
    public Circle(double radius) { this.radius = radius; }
}

public final class Rectangle extends Shape {
    private double length, width;
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
}
```

2️⃣ Rules

1. **`permits`** keyword lists all allowed subclasses.
2. Subclasses must be one of:
    - `final` → cannot be extended further
    - `non-sealed` → can be extended further
    - `sealed` → continues the sealed chain
3. Helps the compiler **know all possible subtypes**, which is useful for `switch` statements.

3️⃣ Example usage

```
public class Main {
    public static void main(String[] args) {
        Shape s1 = new Circle(5);
        Shape s2 = new Rectangle(3, 4);

        printShape(s1);
        printShape(s2);
    }

    static void printShape(Shape shape) {
        // Compiler knows all subclasses → no default needed
        switch (shape) {
            case Circle c -> System.out.println("Circle with radius "
+ c.radius);
            case Rectangle r -> System.out.println("Rectangle " +
r.length + "x" + r.width);
        }
    }
}
```

✅ **Benefits:**

- Safer and controlled inheritance
- Better for pattern matching (`switch` expressions)
- Makes your design **explicit and maintainable**

What is Pattern Matching?

Pattern Matching allows you to **test the type of an object and extract its contents in a single step**.

- Introduced gradually in Java 16+ (with `instanceof`) and enhanced in Java 17–21 with **switch expressions**.
- It reduces **boilerplate type checks and casts**.

2️⃣ Traditional way without pattern matching

```
Object obj = "Hello";

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

- You need **manual casting**, which is verbose and error-prone.

3️⃣ Pattern Matching with `instanceof` (Java 16+)

```
Object obj = "Hello";
```

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

✅ What happens:

- `obj instanceof String s` → checks type and **declares a new variable `s`** of type `String`.
- No manual cast needed.

4️⃣ Pattern Matching in `switch` (Java 17–21)

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

final class Circle implements Shape {
    double radius;
    public Circle(double radius) { this.radius = radius; }
}

final class Rectangle implements Shape {
    double length, width;
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
}

public class Main {
    static void printArea(Shape shape) {
        switch (shape) {
            case Circle c -> System.out.println("Circle area: " +
Math.PI * c.radius * c.radius);
            case Rectangle r -> System.out.println("Rectangle area: "
+ r.length * r.width);
        }
```

```
    }

    public static void main(String[] args) {
        Shape s1 = new Circle(5);
        Shape s2 = new Rectangle(3, 4);

        printArea(s1);
        printArea(s2);
    }
}
```

- `switch` **matches the object type** and binds it to a variable (`c` or `r`).
- No need for explicit `instanceof` + cast.
- Cleaner, less boilerplate.

Benefits of Pattern Matching

1. **Less boilerplate**: fewer `instanceof` + cast statements.
2. **Safer**: compiler knows the variable type within the branch.
3. **Works well with Sealed Classes**: compiler knows all possible subtypes → no default needed in `switch`.
4. **Readable**: logic becomes cleaner and easier to maintain.

Java Collections Framework (JCF) Overview

The **Collections Framework** is a set of interfaces and classes in `java.util` that lets you **store, retrieve, and manipulate groups of objects efficiently**.

Main Interfaces

Interface	Description	Common Implementations
`List`	Ordered collection, allows duplicates	`ArrayList`, `LinkedList`
`Set`	Unordered collection, no duplicates	`HashSet`, `LinkedHashSet`, `TreeSet`
`Queue`	FIFO or priority ordering	`LinkedList`, `PriorityQueue`
`Deque`	Double-ended queue	`ArrayDeque`, `LinkedList`
`Map`	Key-value pairs	`HashMap`, `LinkedHashMap`, `TreeMap`

2️⃣ Advanced Operations on Collections

Java 8+ introduced **Streams API** and **enhanced operations**, which make it easy to process collections.

Example Collection

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

---

a) Filtering

```
List<String> uniqueNames = names.stream()
```

```
                                .filter(name -> name.length() <= 4)
                                .toList();
System.out.println(uniqueNames); // [Bob]
```

---

b) Mapping / Transformation

```
List<String> upperNames = names.stream()
                               .map(String::toUpperCase)
                               .toList();
System.out.println(upperNames); // [ALICE, BOB, CHARLIE, ALICE, DAVID]
```

---

c) Sorting

```
List<String> sortedNames = names.stream()
                                .sorted()
                                .toList();
System.out.println(sortedNames); // [Alice, Alice, Bob, Charlie,
David]
```

---

d) Aggregation / Reduction

```
String concatenated = names.stream()
                           .distinct()
```

```
                            .reduce("", (a, b) -> a + b + ", ");

System.out.println(concatenated); // Alice, Bob, Charlie, David,
```

---

e) Grouping and Counting

```
Map<String, Long> frequency = names.stream()

                                    .collect(Collectors.groupingBy(n ->
n, Collectors.counting()));

System.out.println(frequency); // {Alice=2, Bob=1, Charlie=1, David=1}
```

---

f) Advanced Set Operations

```
Set<Integer> set1 = Set.of(1, 2, 3, 4);

Set<Integer> set2 = Set.of(3, 4, 5, 6);


// Intersection

Set<Integer> intersection =
set1.stream().filter(set2::contains).collect(Collectors.toSet());

System.out.println(intersection); // [3, 4]


// Union

Set<Integer> union = Stream.concat(set1.stream(),
set2.stream()).collect(Collectors.toSet());

System.out.println(union); // [1, 2, 3, 4, 5, 6]
```

---

g) Parallel Operations

Collections can be **processed in parallel** for better performance on multi-core CPUs:

```
List<Integer> numbers = IntStream.rangeClosed(1,
1000).boxed().toList();

int sum = numbers.parallelStream().reduce(0, Integer::sum);

System.out.println(sum); // 500500
```

3️⃣ Key Takeaways

1. **Streams + Lambdas** make operations on collections **concise and readable**.
2. **Collectors** allow grouping, counting, partitioning, etc.
3. **Parallel streams** help with **high-performance concurrent processing**.
4. Always choose the **right collection type** for your problem (e.g., `ArrayList` for random access, `LinkedList` for insertion/deletion heavy).