DAY 1-2(03/10/2025)

Java 21 Features & Enhancements – Records

Definition:

- Records are **special classes** for **immutable data carriers**.
- Auto-generate constructor, getters, equals(), hashCode(), and toString().

Syntax:

public record WalletTransaction(String transactionId, double amount, String type) {}

Key Features:

- Immutable fields (cannot be modified after creation)
- Minimal boilerplate code
- Supports **compact constructor** for validation

Compact Constructor Example:

```
public record WalletTransaction(String transactionId, double amount, String type) {
   public WalletTransaction {
      if(transactionId == null || transactionId.isBlank())
         throw new IllegalArgumentException("transactionId cannot be null");
      if(amount <= 0)
         throw new IllegalArgumentException("amount must be positive");
      if(!type.equals("CREDIT") && !type.equals("DEBIT"))
         throw new IllegalArgumentException("type must be CREDIT or DEBIT");
   }
}
```

Example Usage:

```
WalletTransaction txn = new WalletTransaction("TXN001", 100, "CREDIT");
System.out.println(txn.transactionId());  // TXN001
System.out.println(txn.amount());         // 100.0
System.out.println(txn.type());           // CREDIT
System.out.println(txn);                  // WalletTransaction[transactionId=TXN001,
amount=100.0, type=CREDIT]
```

Advantages:

- Ensures data integrity
- Reduces boilerplate code
- Easy to read and maintain

Java 21 Features & Enhancements – Sealed Classes

Definition:

- Sealed classes restrict which other classes can extend or implement them.
- Helps control inheritance hierarchies and improves type safety.

Syntax Example:

```
public sealed class Transaction permits CreditTransaction, DebitTransaction { }

public final class CreditTransaction extends Transaction { }

public final class DebitTransaction extends Transaction { }
```

Key Points:

- sealed → marks a class as sealed
- permits → lists classes allowed to extend it
- Subclasses must be final, sealed, or non-sealed
- Prevents unexpected extensions by other developers

Use Cases:

- Fixed hierarchy modeling (e.g., transaction types, shapes, events)
- Improves pattern matching reliability
- Ensures compile-time safety for type checking

Advantages:

- Controlled inheritance
- Improves readability and maintainability
- Works well with switch expressions and pattern matching

Example Usage with Pattern Matching:

```
Transaction txn = new CreditTransaction();

if (txn instanceof CreditTransaction credit) {

   System.out.println("Processing credit transaction");

} else if (txn instanceof DebitTransaction debit) {

   System.out.println("Processing debit transaction");

}
```

Java 21 Features & Enhancements – Pattern Matching

Definition:

- Pattern Matching simplifies **type checking and casting**.
- Reduces boilerplate instanceof + cast statements.
- Introduced improvements in **Java 21** for cleaner conditional logic.

Syntax Example:

```
Object obj = new WalletTransaction("TXN001", 100, "CREDIT");

if (obj instanceof WalletTransaction wt) {

   System.out.println(wt.amount());  // Auto-cast to WalletTransaction

}
```

Old Way (pre-pattern matching):

```
if(obj instanceof WalletTransaction) {

   WalletTransaction wt = (WalletTransaction) obj;  // manual cast

   System.out.println(wt.amount());

}
```

Key Points:

- Combines type check + casting in one step
- Works with instanceof and switch expressions
- Supports records and sealed classes efficiently

Example with Sealed Classes:

```
Transaction txn = new CreditTransaction();

switch(txn) {

   case CreditTransaction c -> System.out.println("Processing credit");

   case DebitTransaction d -> System.out.println("Processing debit");

}
```

Java 21 Features & Enhancements – Virtual Threads

Definition:

- Virtual Threads are **lightweight threads** introduced in Project Loom (Java 21).
- They allow handling **concurrency at massive scale** without the memory and scheduling cost of platform (OS) threads.

Key Points:

- A **virtual thread** is still a Thread in Java, but **much cheaper** than platform threads.
- Thousands (even millions) can be created without exhausting system resources.
- Great for apps with **many concurrent tasks** (e.g., transaction processing, web servers).

Creating Virtual Threads:

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

executor.submit(() -> {

   System.out.println("Running task on " + Thread.currentThread());

});

executor.shutdown();
```

Output Example:

```
Running task on VirtualThread[#22]/runnable@ForkJoinPool-1-worker-2
```

Benefits:

- High scalability → can handle thousands of tasks simultaneously.
- Simple coding model → same APIs as normal threads, but lighter.
- Better resource usage → reduces CPU + memory overhead.
- Works seamlessly with existing APIs (Executors, CompletableFuture, etc.).

Collections Framework and Advanced Operations

1. What is the Collections Framework?

- A **unified architecture** for storing and manipulating groups of objects in Java.
- Provides **interfaces**, **implementations (classes)**, and **algorithms (methods)**.
- Saves effort: no need to build custom data structures.

2. Core Interfaces

- **Collection** (root interface) → List, Set, Queue extend it.
- **List** → Ordered, allows duplicates (e.g., ArrayList, LinkedList).
- **Set** → No duplicates (e.g., HashSet, TreeSet).
- **Queue / Deque** → FIFO or double-ended (e.g., PriorityQueue, ArrayDeque).
- **Map** (not a child of Collection) → Key-value pairs (e.g., HashMap, TreeMap).

3. Advanced Operations

a) Streams API (Functional Style)

- Works on collections to process data **declaratively**.

Examples:

```
 List<String> names = List.of("Alice", "Bob", "Charlie");
names.stream()
    .filter(n -> n.startsWith("A"))
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

  - **filter** → select elements
  - **map** → transform elements
  - **forEach** → consume/output elements

b) Bulk Operations (on collections directly)

- removeIf() → removes elements based on condition
- forEach() → iterate with lambda
- replaceAll() → apply transformation

Example:

```
List<Integer> nums = new ArrayList<>(List.of(1, 2, 3, 4));
nums.removeIf(n -> n % 2 == 0);  // Removes even numbers
nums.replaceAll(n -> n * n);     // Squares remaining numbers
```

c) Map Enhancements (Java 8+)

- forEach() → loop with key & value
- computeIfAbsent() → add only if key missing
- merge() → combine values for duplicate keys

Example:

```
Map<String, Integer> scores = new HashMap<>();
scores.put("Alice", 50);
scores.merge("Alice", 20, Integer::sum); // Alice = 70
```

d) Parallel Streams

- Use multiple threads internally for performance.

Example:

```
long count = names.parallelStream()
            .filter(n -> n.length() > 3)
            .count();
```

e) Immutable and Unmodifiable Collections

- List.of(...), Set.of(...), Map.of(...) → immutable collections.

- Collections.unmodifiableList(list) → prevents modification.

4. Why Important?

- Provides **ready-made, optimized data structures**.
- Advanced operations (Streams, bulk ops, Map APIs) allow **concise and powerful code**.
- Supports **functional programming** → cleaner, less error-prone.

Exception Handling Strategies and Best Practices

1. What is an Exception?

- An **unexpected event** that disrupts program execution.

- **Types:**
  - **Checked Exceptions** → must be handled at compile-time (e.g., IOException).
  - **Unchecked Exceptions** → runtime errors (e.g., NullPointerException).
  - **Errors** → serious issues not meant to be handled (e.g., OutOfMemoryError).

2. Exception Handling Keywords

- **try** → code that may throw exception.
- **catch** → handles the exception.
- **finally** → cleanup block (always executes).
- **throw** → explicitly throw exception.
- **throws** → declare exceptions in method signature.

3. Strategies

- Use **specific exceptions** (not generic Exception).
- Apply **centralized exception handling** (e.g., @ControllerAdvice in Spring).
- **Don't swallow exceptions** → always log or rethrow.
- Use **finally** or **try-with-resources** for cleanup.
- **Wrap low-level exceptions** into meaningful custom exceptions.
- Create **custom exceptions** for domain-specific errors.
- Follow **fail-fast principle** → validate inputs early.

4. Best Practices

- Handle exceptions **at the right level** in code.
- **Never use exceptions for normal flow control** (they're costly).
- Always **log exceptions** with proper details (stack trace, message).
- **Meaningful messages** → user-friendly + developer-friendly.
- Prefer **unchecked exceptions** for programming errors.

Functional Programming in Java – Streams & Lambda Expressions

1. Lambda Expressions

Definition:

- Anonymous functions that can be treated as **objects**.
- Used to provide **implementation for functional interfaces** (interfaces with single abstract method).

Syntax:

(parameters) -> expression

(parameters) -> { statements }

Examples:

// Simple lambda for Runnable

Runnable r = () -> System.out.println("Hello Lambda");

// Lambda for Comparator

Comparator<Integer> cmp = (a, b) -> a - b;

Key Points:

- Reduces boilerplate code.
- Can be passed as **parameters** or returned from methods.
- Often used with **collections and streams**.

2. Streams API

Definition:

- Streams provide a **functional way to process sequences of elements**.
- Supports operations like **filter, map, reduce, collect**.

Example:

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

// Filter + Map + ForEach

```
names.stream()
    .filter(n -> n.startsWith("A"))    // keeps names starting with A
    .map(String::toUpperCase)          // converts to uppercase
    .forEach(System.out::println);     // prints result
```

Key Points:

- Streams are **lazy** → operations execute only when terminal operation is called.
- Can be **sequential** (stream()) or **parallel** (parallelStream()).

Benefits:

- Concise, readable, declarative code.
- Works well with **immutable data** (like records).
- Easy to **parallelize** for concurrency.