Morning Session:

Collections Framework

Concept

- **Collections** are data structures to store, retrieve, and manipulate groups of objects.
- **Key Interfaces**:
  - `List` → Ordered, allows duplicates (`ArrayList`, `LinkedList`)
  - `Set` → No duplicates (`HashSet`, `TreeSet`)
  - `Map` → Key-value pairs (`HashMap`, `TreeMap`)

Examples

List (ArrayList)

```java
import java.util.ArrayList;

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

        System.out.println(names); // [Alice, Bob, Charlie]
        System.out.println(names.get(1)); // Bob
    }
}
```

Set (HashSet)

```java
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        HashSet<Integer> numbers = new HashSet<>();
```

```
        numbers.add(10);
        numbers.add(20);
        numbers.add(10); // Duplicate ignored

        System.out.println(numbers); // [20, 10]
    }
}
```

Map (HashMap)

```
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap<String, Integer> ages = new HashMap<>();
        ages.put("Alice", 25);
        ages.put("Bob", 30);

        System.out.println(ages.get("Alice")); // 25
    }
}
```

Generics in Java

Concept

- **Generics** enforce **type safety** at compile time.
- Eliminates the need for explicit typecasting.

Example

```
class Box<T> {
    private T content;

    public void setContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.setContent("Hello");
        System.out.println(stringBox.getContent()); // Hello

        Box<Integer> intBox = new Box<>();
        intBox.setContent(100);
        System.out.println(intBox.getContent()); // 100
    }
}
```

LinkedList

A doubly-linked list implementation of the List and Deque interfaces.

Characteristics:

- Allows null elements
- Maintains insertion order
- Not synchronized
- Good for frequent add/remove operations
- Slower random access than ArrayList

Example:

```java
LinkedList<String> linkedList = new LinkedList<>();

// Adding elements
linkedList.add("Apple");
linkedList.addFirst("Banana");  // Adds to beginning
linkedList.addLast("Cherry");   // Adds to end
linkedList.add(1, "Mango");     // Adds at specific index

// Accessing elements
String first = linkedList.getFirst();
String last = linkedList.getLast();
String element = linkedList.get(2);
```

```java
// Removing elements
linkedList.removeFirst();
linkedList.removeLast();
linkedList.remove("Mango");

// Iterating
for (String fruit : linkedList) {
    System.out.println(fruit);
}

// As Deque
linkedList.offer("Date");      // Adds to end
linkedList.poll();             // Removes from head
```

LinkedHashSet

Hash table and linked list implementation of the Set interface with predictable iteration order.

Characteristics:

- Maintains insertion order
- No duplicates
- Allows one null element
- Slower than HashSet for add/remove
- Faster iteration than HashSet

Example:

```java
LinkedHashSet<String> linkedHashSet = new LinkedHashSet<>();

linkedHashSet.add("Apple");
linkedHashSet.add("Banana");
linkedHashSet.add("Apple");  // Duplicate, won't be added
linkedHashSet.add(null);     // Allows null
linkedHashSet.add("Cherry");

System.out.println(linkedHashSet); // [Apple, Banana, null, Cherry] - maintains order

// Iteration preserves insertion order
```

```java
for (String item : linkedHashSet) {
    System.out.println(item);
}
```

LinkedHashMap

Hash table and linked list implementation of the Map interface with predictable iteration order.

Characteristics:

- Maintains insertion order by default
- Can be configured to maintain access order (for LRU caches)
- Slower than HashMap for add/remove
- Faster iteration than HashMap
- Allows one null key and multiple null values

Example:

```java
// Insertion order maintained
LinkedHashMap<String, Integer> linkedHashMap = new LinkedHashMap<>();

linkedHashMap.put("Apple", 1);
linkedHashMap.put("Banana", 2);
linkedHashMap.put(null, 3);     // null key
linkedHashMap.put("Cherry", null); // null value
linkedHashMap.put("Banana", 5);  // Updates existing

System.out.println(linkedHashMap); // {Apple=1, Banana=5, null=3, Cherry=null}

// Access order example (LRU cache)
LinkedHashMap<String, Integer> accessOrderMap = new LinkedHashMap<>(16, 0.75f, true);
accessOrderMap.put("A", 1);
accessOrderMap.put("B", 2);
accessOrderMap.put("C", 3);

accessOrderMap.get("A");  // Moves "A" to end
System.out.println(accessOrderMap); // {B=2, C=3, A=1}
```

TreeSet

A NavigableSet implementation based on a TreeMap.

Characteristics:

- Stores elements in sorted order (natural ordering or by Comparator)
- No duplicates
- No null elements (if natural ordering used)
- Basic operations (add, remove, contains) have O(log n) time

Example:

```
TreeSet<Integer> treeSet = new TreeSet<>();

treeSet.add(5);
treeSet.add(2);
treeSet.add(8);
treeSet.add(2);  // Duplicate, not added

System.out.println(treeSet); // [2, 5, 8] - sorted

// Methods
System.out.println(treeSet.first());  // 2
System.out.println(treeSet.last());   // 8
System.out.println(treeSet.higher(4)); // 5
System.out.println(treeSet.lower(5)); // 2

// Subsets
System.out.println(treeSet.subSet(2, true, 5, true)); // [2, 5]
System.out.println(treeSet.headSet(5));  // [2] (elements < 5)
System.out.println(treeSet.tailSet(5));  // [5, 8] (elements >= 5)
```

TreeMap

A Red-Black tree based NavigableMap implementation.

Characteristics:

- Keys are sorted (natural ordering or by Comparator)

- No duplicate keys
- No null keys if natural ordering used (but can have null values)
- Basic operations have O(log n) time

Example:

```
TreeMap<String, Integer> treeMap = new TreeMap<>();

treeMap.put("Apple", 1);
treeMap.put("Banana", 2);
treeMap.put("Cherry", 3);
treeMap.put("Banana", 4);  // Updates existing

System.out.println(treeMap); // {Apple=1, Banana=4, Cherry=3} - sorted keys

// Navigation methods
System.out.println(treeMap.firstKey());  // "Apple"
System.out.println(treeMap.lastKey());   // "Cherry"
System.out.println(treeMap.higherKey("Banana")); // "Cherry"
System.out.println(treeMap.lowerKey("Banana"));  // "Apple"

// Submaps
System.out.println(treeMap.subMap("Apple", true, "Cherry", false)); // {Apple=1, Banana=4}
System.out.println(treeMap.headMap("Cherry"));  // {Apple=1, Banana=4}
System.out.println(treeMap.tailMap("Banana"));  // {Banana=4, Cherry=3}
```

Collections.synchronizedList

Returns a synchronized (thread-safe) list backed by the specified list.

Characteristics:

- All operations are synchronized
- Must manually synchronize on returned list during iteration
- Prefer CopyOnWriteArrayList for better concurrent performance

Example:

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
```

```java
// Add elements - thread-safe
syncList.add("A");
syncList.add("B");

// Must synchronize during iteration
synchronized(syncList) {
    for (String item : syncList) {
        System.out.println(item);
    }
}

// Alternative for thread-safe iteration
List<String> snapshot = new ArrayList<>(syncList);
for (String item : snapshot) {
    System.out.println(item);
}
```

Collections.synchronizedSet

Returns a synchronized (thread-safe) set backed by the specified set.

Characteristics:

- All operations are synchronized
- Must manually synchronize on returned set during iteration
- Prefer CopyOnWriteArraySet for better concurrent performance

Example:

```java
Set<String> syncSet = Collections.synchronizedSet(new HashSet<>());

// Thread-safe operations
syncSet.add("A");
syncSet.remove("B");

// Synchronized iteration
synchronized(syncSet) {
    for (String item : syncSet) {
        System.out.println(item);
    }
}
```

Collections.synchronizedMap

Returns a synchronized (thread-safe) map backed by the specified map.

Characteristics:

- All operations are synchronized
- Must manually synchronize on returned map during iteration
- Prefer ConcurrentHashMap for better concurrent performance

Example:

```
Map<String, Integer> syncMap = Collections.synchronizedMap(new HashMap<>());

// Thread-safe operations
syncMap.put("A", 1);
syncMap.get("B");

// Synchronized iteration
synchronized(syncMap) {
    for (Map.Entry<String, Integer> entry : syncMap.entrySet()) {
        System.out.println(entry.getKey() + ": " + entry.getValue());
    }
}
```

List.of() (Java 9+)

Creates an immutable list containing the specified elements.

Characteristics:

- Immutable (cannot add, remove, or modify elements)
- Null elements not allowed
- Space-efficient implementations for small lists
- Thread-safe

Example:

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

```java
System.out.println(immutableList); // [A, B, C]

// Operations
System.out.println(immutableList.get(1)); // "B"
System.out.println(immutableList.size()); // 3

// Unsupported operations (throw UnsupportedOperationException)
immutableList.add("D");
immutableList.set(0, "X");
immutableList.remove(0);

// Null not allowed
List.of("A", null, "B"); // Throws NullPointerException
```

List.copyOf() (Java 10+)

Creates an immutable list containing the elements of the given Collection.

Characteristics:

- Returns an immutable list
- If input is already immutable, may return the input itself
- Null elements not allowed
- Thread-safe

Example:

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

List<String> immutableCopy = List.copyOf(original);

System.out.println(immutableCopy); // [A, B]

// Changes to original don't affect the copy
original.add("C");
System.out.println(immutableCopy); // Still [A, B]

// If original is already immutable, might return same instance
```

```java
List<String> immutable = List.of("X", "Y");
List<String> copy = List.copyOf(immutable);
System.out.println(immutable == copy); // true (same instance)

// Unsupported operations
immutableCopy.add("D"); // UnsupportedOperationException
```

Comparable Interface

Overview

The `Comparable` interface is used to define the natural ordering of objects. It contains a single method:

java
```java
public interface Comparable<T> {
   int compareTo(T o);
}
```

Characteristics

- Defines the natural ordering of objects
- Implemented by the class whose objects need to be sorted
- Only one natural ordering can be defined per class
- Used by Collections.sort() and Arrays.sort() methods
- Used by sorted collections like TreeSet and TreeMap

Example: Implementing Comparable

```java
class Student implements Comparable<Student> {
   private int rollNo;
   private String name;
   private int age;

   public Student(int rollNo, String name, int age) {
      this.rollNo = rollNo;
      this.name = name;
      this.age = age;
   }
```

```java
    @Override
    public int compareTo(Student other) {
        // Natural ordering by roll number
        return this.rollNo - other.rollNo;
    }

    // Getters
    public int getRollNo() { return rollNo; }
    public String getName() { return name; }
    public int getAge() { return age; }

    @Override
    public String toString() {
        return "Student{rollNo=" + rollNo + ", name='" + name + "', age=" + age + "}";
    }
}

public class ComparableExample {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(103, "Alice", 20));
        students.add(new Student(101, "Bob", 22));
        students.add(new Student(102, "Charlie", 21));

        // Sorting using natural ordering (by rollNo)
        Collections.sort(students);

        System.out.println("Students sorted by roll number:");
        students.forEach(System.out::println);

        /* Output:
        Student{rollNo=101, name='Bob', age=22}
        Student{rollNo=102, name='Charlie', age=21}
        Student{rollNo=103, name='Alice', age=20}
        */
    }
}
```

compareTo() Contract

- Returns negative if this object is less than the specified object

- Returns zero if this object is equal to the specified object
- Returns positive if this object is greater than the specified object
- Must be consistent with equals() (recommended but not required)

Comparator Interface

Overview

The `Comparator` interface is used to define external comparison logic for objects. It contains:

java
```java
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

Characteristics

- Defines multiple comparison strategies for a class
- Implemented as a separate class or using lambda expressions
- More flexible than Comparable as it doesn't require modifying the original class
- Used by Collections.sort() with a Comparator parameter
- Used by sorted collections with custom ordering

Example: Implementing Comparator

```java
// Name comparator
class NameComparator implements Comparator<Student> {
   @Override
   public int compare(Student s1, Student s2) {
      return s1.getName().compareTo(s2.getName());
   }
}

// Age comparator
class AgeComparator implements Comparator<Student> {
   @Override
   public int compare(Student s1, Student s2) {
      return s1.getAge() - s2.getAge();
```

```
    }
}

public class ComparatorExample {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(103, "Alice", 20));
        students.add(new Student(101, "Bob", 22));
        students.add(new Student(102, "Charlie", 21));

        // Sorting by name using Comparator
        Collections.sort(students, new NameComparator());
        System.out.println("Students sorted by name:");
        students.forEach(System.out::println);

        /* Output:
        Student{rollNo=103, name='Alice', age=20}
        Student{rollNo=101, name='Bob', age=22}
        Student{rollNo=102, name='Charlie', age=21}
        */

        // Sorting by age using Comparator
        Collections.sort(students, new AgeComparator());
        System.out.println("\nStudents sorted by age:");
        students.forEach(System.out::println);

        /* Output:
        Student{rollNo=103, name='Alice', age=20}
        Student{rollNo=102, name='Charlie', age=21}
        Student{rollNo=101, name='Bob', age=22}
        */
    }
}
```

Modern Comparator Usage (Java 8+)

With Java 8, we can use lambda expressions and method references to create comparators more concisely.

```
public class ModernComparatorExample {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
```

```java
        students.add(new Student(103, "Alice", 20));
        students.add(new Student(101, "Bob", 22));
        students.add(new Student(102, "Charlie", 21));

        // Using lambda expression
        Comparator<Student> nameComp = (s1, s2) -> s1.getName().compareTo(s2.getName());
        students.sort(nameComp);

        // Using Comparator.comparing()
        students.sort(Comparator.comparing(Student::getName));

        // Reverse order
        students.sort(Comparator.comparing(Student::getName).reversed());

        // Multiple criteria
        students.sort(Comparator.comparing(Student::getAge)
                      .thenComparing(Student::getName));

        // Handling null values
        Comparator<Student> nullSafeNameComp = Comparator.comparing(
            Student::getName, Comparator.nullsLast(Comparator.naturalOrder()));
    }
}
```

Comparing Comparable and Comparator

Feature	Comparable	Comparator
Package	java.lang	java.util
Method	compareTo()	compare()
Sorting logic	Inside the class being sorted	Separate class/lambda
Number of sorts	Single (natural ordering)	Multiple (different comparators)
Modifies class	Yes (implements interface)	No
Usage	Collections.sort(list)	Collections.sort(list, comparator)

Feature	Comparable	Comparator
Null handling	Throws NullPointerException	Can handle nulls with nullsFirst/nullsLast

Practical Example with Both

```
public class StudentSorter {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(103, "Alice", 20));
        students.add(new Student(101, "Bob", 22));
        students.add(new Student(102, "Charlie", 21));
        students.add(new Student(104, "Alice", 19));

        // Natural ordering (by rollNo - from Comparable)
        Collections.sort(students);
        System.out.println("By roll number:");
        students.forEach(System.out::println);

        // By name (using Comparator)
        students.sort(Comparator.comparing(Student::getName));
        System.out.println("\nBy name:");
        students.forEach(System.out::println);

        // By age then name
        students.sort(Comparator.comparing(Student::getAge)
                    .thenComparing(Student::getName));
        System.out.println("\nBy age then name:");
        students.forEach(System.out::println);

        // Reverse order by name
        students.sort(Comparator.comparing(Student::getName).reversed());
        System.out.println("\nBy name (reverse order):");
        students.forEach(System.out::println);
    }
}
```

Key Points to Remember

1. Use `Comparable` when you want to define a natural/default ordering for your objects
2. Use `Comparator` when you need multiple ways to compare objects or when you can't modify the source class
3. Java 8 introduced many helpful static methods in the `Comparator` interface like `comparing()`, `thenComparing()`, `nullsFirst()`, `nullsLast()`, and `naturalOrder()`
4. For complex sorting requirements, you can chain multiple comparators using `thenComparing()`
5. Always ensure your comparison logic is consistent with `equals()` when possible to avoid confusing behavior in sorted collections

Lambda Expressions & Functional Interfaces

Concept

- **Lambda** → Shortcut for anonymous classes.
- **Functional Interface** → Interface with **one abstract method** (`Runnable`, `Comparator`).

Example

```
@FunctionalInterface
interface Greeting {
    void greet(String name);
}

public class Main {
    public static void main(String[] args) {
        // Lambda Expression
        Greeting g = (name) -> System.out.println("Hello, " + name);
        g.greet("Alice"); // Hello, Alice

        // Using Lambda with Runnable
        Runnable r = () -> System.out.println("Thread running");
        new Thread(r).start();
```

```
    }
}
```

Java Date & Time API (java.time)

Key Classes

- `LocalDate` → Date (yyyy-MM-dd)
- `LocalTime` → Time (HH:mm:ss)
- `LocalDateTime` → Date + Time
- `DateTimeFormatter` → Format dates

Example

```
import java.time.*;
import java.time.format.DateTimeFormatter;

public class Main {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        System.out.println(today); // 2023-10-05

        LocalTime now = LocalTime.now();
        System.out.println(now); // 14:30:45

        LocalDateTime current = LocalDateTime.now();
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm");
        System.out.println(current.format(formatter)); // 05-10-2023 14:30
    }
}
```

1. Introduction to Stream API

The Java Stream API (introduced in Java 8) provides a functional approach to process collections of objects. It allows you to perform complex data processing operations like filtering, mapping, reducing, and more in a declarative way.

Key Characteristics:

- **Not a data structure**: Doesn't store data, it operates on source data structures
- **Functional in nature**: Operations don't modify the source
- **Lazy evaluation**: Intermediate operations are only executed when terminal operation is invoked
- **Consumable**: Can only be traversed once

2. Stream Creation

From Collections

```
List<String> names = Arrays.asList("John", "Alice", "Bob");
Stream<String> stream = names.stream();
```

From Arrays

```
String[] namesArray = {"John", "Alice", "Bob"};
Stream<String> stream = Arrays.stream(namesArray);
```

Static Factory Methods

```
Stream<String> stream = Stream.of("John", "Alice", "Bob");
Stream<Integer> numbers = Stream.iterate(0, n -> n + 1).limit(10); // Infinite stream
Stream<String> empty = Stream.empty();
```

From Files

```
try (Stream<String> lines = Files.lines(Paths.get("file.txt"))) {
    lines.forEach(System.out::println);
}
```

Intermediate Operations

filter()

Filters elements based on a predicate.

```
List<String> names = Arrays.asList("John", "Alice", "Bob", "Anna");
names.stream()
    .filter(name -> name.startsWith("A"))
    .forEach(System.out::println); // Alice, Anna
```

map()

Transforms each element using a function.

```
List<String> names = Arrays.asList("John", "Alice", "Bob");
names.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println); // JOHN, ALICE, BOB
```

flatMap()

Flattens nested structures.

```
List<List<String>> nestedNames = Arrays.asList(
    Arrays.asList("John", "Doe"),
    Arrays.asList("Alice", "Smith")
);

nestedNames.stream()
        .flatMap(List::stream)
        .forEach(System.out::println); // John, Doe, Alice, Smith
```

distinct()

Removes duplicates.

```
List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 3, 3);
numbers.stream()
      .distinct()
      .forEach(System.out::println); // 1, 2, 3
```

sorted()

Sorts elements.

```
List<String> names = Arrays.asList("John", "Alice", "Bob");
names.stream()
    .sorted()
    .forEach(System.out::println); // Alice, Bob, John
```

peek()

Debugging helper to inspect elements.

```java
List<String> names = Arrays.asList("John", "Alice", "Bob");
names.stream()
    .peek(System.out::println)
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

limit() and skip()

Limit or skip elements.

```java
Stream.iterate(0, n -> n + 1)
     .skip(5)
     .limit(10)
     .forEach(System.out::println); // Prints 5-14
```

Terminal Operations

forEach()

Iterates through each element.

```java
List<String> names = Arrays.asList("John", "Alice", "Bob");
names.stream().forEach(System.out::println);
```

collect()

Converts stream to a collection or other data structure.

```java
List<String> names = Arrays.asList("John", "Alice", "Bob");
List<String> filtered = names.stream()
                       .filter(name -> name.length() > 3)
                       .collect(Collectors.toList());
```

toArray()

Converts stream to an array.

```java
String[] namesArray = names.stream().toArray(String[]::new);
```

reduce()

Combines elements into a single result.

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream().reduce(0, Integer::sum);
```

min() and max()

Find minimum or maximum element.

```java
Optional<String> shortestName = names.stream()
                                     .min(Comparator.comparingInt(String::length));
```

count()

Counts elements in stream.

```java
long count = names.stream().filter(name -> name.startsWith("A")).count();
```

anyMatch(), allMatch(), noneMatch()

Short-circuiting predicate checks.

```java
boolean anyStartsWithA = names.stream().anyMatch(name -> name.startsWith("A"));
boolean allLongerThan2 = names.stream().allMatch(name -> name.length() > 2);
boolean noneEmpty = names.stream().noneMatch(String::isEmpty);
```

findFirst() and findAny()

Find elements in stream.

```java
Optional<String> first = names.stream().findFirst();
Optional<String> any = names.parallelStream().findAny();
```

Collectors

Powerful terminal operations to transform streams into different forms.

toList(), toSet(), toCollection()

java

```java
List<String> list = names.stream().collect(Collectors.toList());
Set<String> set = names.stream().collect(Collectors.toSet());
TreeSet<String> treeSet = names.stream()
                               .collect(Collectors.toCollection(TreeSet::new));
```

Parallel Streams

Enable parallel processing of streams.

java
```java
List<String> names = Arrays.asList("John", "Alice", "Bob", "Anna", "David");
List<String> result = names.parallelStream()
                         .filter(name -> name.length() > 3)
                         .map(String::toUpperCase)
                         .collect(Collectors.toList());
```

Practical Examples

Example 1: Processing Employees

```java
List<Employee> employees = Arrays.asList(
    new Employee("John", "IT", 75000),
    new Employee("Alice", "HR", 65000),
    new Employee("Bob", "IT", 80000),
    new Employee("Anna", "Finance", 90000)
);

// Average salary by department
Map<String, Double> avgSalaryByDept = employees.stream()
    .collect(Collectors.groupingBy(
        Employee::getDepartment,
        Collectors.averagingDouble(Employee::getSalary)
    ));

// Highest paid employee
Optional<Employee> highestPaid = employees.stream()
    .max(Comparator.comparingDouble(Employee::getSalary));

// All IT employees sorted by name
List<Employee> itEmployees = employees.stream()
    .filter(e -> "IT".equals(e.getDepartment()))
    .sorted(Comparator.comparing(Employee::getName))
    .collect(Collectors.toList());
```

Example 2: Word Count

```java
String sentence = "the quick brown fox jumps over the lazy dog";
Map<String, Long> wordCount = Arrays.stream(sentence.split(" "))
```

```
    .collect(Collectors.groupingBy(
        Function.identity(),
        Collectors.counting()
    ));
```

Example 3: Prime Numbers

```
IntStream.rangeClosed(2, 100)
    .filter(n -> IntStream.rangeClosed(2, (int) Math.sqrt(n))
                    .noneMatch(i -> n % i == 0))
    .forEach(System.out::println);
```

Afternoon Session:

1. JDBC (Java Database Connectivity)

Steps to Connect to a Database

1. **Load Driver** → `Class.forName("com.mysql.cj.jdbc.Driver");`
2. **Create Connection** → `Connection con = DriverManager.getConnection(url, user, pass);`
3. **Execute Query** → `Statement stmt = con.createStatement();`
4. **Process Result** → `ResultSet rs = stmt.executeQuery("SELECT * FROM users");`
5. **Close Connection** → `con.close();`

Example

```
import java.sql.*;

public class Main {
    public static void main(String[] args) throws SQLException {
        String url = "jdbc:mysql://localhost:3306/mydb";
        String user = "root";
        String pass = "password";

        Connection con = DriverManager.getConnection(url, user, pass);
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

```java
        while (rs.next()) {
            System.out.println(rs.getString("name"));
        }

        con.close();
    }
}
```

2. JUnit with Mockito (Unit Testing & Mocking)

JUnit Basics

- **Annotations**:
  - `@Test` → Marks a test method.
  - `@Before` → Runs before each test.
  - `@After` → Runs after each test.

Example (JUnit Test)

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator calc = new Calculator();
        assertEquals(5, calc.add(2, 3));
    }
}
```

Mockito (Mocking Dependencies)

```java
import org.junit.Test;
import static org.mockito.Mockito.*;

public class UserServiceTest {
    @Test
    public void testGetUser() {
        // Create a mock UserRepository
        UserRepository mockRepo = mock(UserRepository.class);
```

```
        when(mockRepo.findById(1)).thenReturn(new User(1, "Alice"));

        UserService service = new UserService(mockRepo);
        User user = service.getUser(1);

        assertEquals("Alice", user.getName());
    }
}
```

3. Error Handling & Debugging Techniques

Common Debugging Techniques

1. **Logging** → `System.out.println()` or `Logger`.
2. **Breakpoints** → Debug mode in IDE.
3. **Stack Traces** → Analyze exceptions.

Example (Debugging with Logs)

```
import java.util.logging.*;

public class Main {
    private static final Logger logger = Logger.getLogger(Main.class.getName());

    public static void main(String[] args) {
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            logger.severe("Division by zero: " + e.getMessage());
        }
    }
}
```