Morning Session:

1. 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

1. List (ArrayList)

```
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
    }
}
```

2. Set (HashSet)

```
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]
    }
}
```

3. 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
    }
}
```

2. 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;
    }
}
```

```
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
    }
}
```

3. 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();
    }
}
```

4. 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
    }
}
```

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");

        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)

```
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)

```
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());
        }
    }
}
```