CORE JAVA TOPICS (with Employee class)

Features of Java:

1. Simple
   - Easy to learn and use
   - Syntax is similar to C/C++, but with fewer complexities
2. Object-Oriented
   - Everything in Java is treated as an object
   - Follows OOP concepts: Inheritance, Polymorphism, Encapsulation, Abstraction
3. Platform-Independent
   - Write Once, Run Anywhere (WORA)
   - Java code is compiled into bytecode, which runs on any device with a Java Virtual Machine (JVM)
4. Secure
   - No direct memory access like C/C++
   - Uses a sandbox to run code
   - Supports cryptography, access control, and authentication
5. Robust
   - Strong memory management
   - Automatic garbage collection
   - Exception handling and type checking
6. Multithreaded
   - Allows simultaneous execution of two or more parts of a program
   - Useful for multimedia, gaming, real-time apps
7. Architecture-Neutral
   - Java bytecode is not dependent on processor architecture
   - Same bytecode runs on any platform
8. Portable
   - Java programs can move easily from one system to another
   - No platform-specific implementation
9. High Performance
   - Although slower than C++, Java is faster than many interpreted languages
   - Just-In-Time (JIT) compiler improves performance
10. Distributed
    - Java provides built-in networking features
    - Can create distributed applications using RMI, EJB, or Web Services

11. Dynamic

- Can load classes at runtime using Reflection
- Supports dynamic linking of new class libraries

---

1. Java Basics

Definition: Java is an object-oriented, platform-independent language used to develop secure and robust applications.

Example:

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

---

2. Class and Object

Definition:
A *class* is a blueprint. An *object* is an instance of a class.

Code Example:

```
public class Employee {
    int id;
    String name;
    double salary;
}


public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.id = 101;
        emp.name = "Alice";
        emp.salary = 60000;
        System.out.println(emp.name + "'s Salary is ₹" + emp.salary);
    }
}
```

Output:

```
Alice's Salary is ₹60000.0
```

Assignment:

- Create a class Product with id, name, price. Display details.

---

3. Constructor

Definition:
A constructor is a special method invoked at the time of object creation.

Example:

```
public class Employee {
    int id;
    String name;
    double salary;

    Employee(int id, String name, double salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }
}
```

Assignment:

- Add a default constructor and print "Employee Created".

---

4. Encapsulation

Definition:
Encapsulation is the process of wrapping data and methods into a single unit (class), using private access modifiers.

Example:

```
public class Employee {
    private int id;
    private String name;

    public void setId(int id) { this.id = id; }
    public int getId() { return id; }

    public void setName(String name) { this.name = name; }
    public String getName() { return name; }
}
```

Assignment:

- Add getters/setters for salary.

---

5. Inheritance

Definition:
Inheritance allows one class to inherit the fields and methods of another.

Example:

```
public class Manager extends Employee {
    double bonus;

    public void showBonus() {
        System.out.println("Bonus: " + bonus);
    }
}
```

---

6. Polymorphism

Definition:
Polymorphism means one interface, many implementations.

Method Overriding Example:

```
public class Employee {
    void work() {
        System.out.println("Employee works.");
    }
}

public class Developer extends Employee {
    void work() {
        System.out.println("Developer codes.");
    }
}
```

Output:

```
Developer codes.
```

---

7. Abstraction

Definition:
Hiding internal details and showing only necessary features.

Abstract Class Example:

```
abstract class Employee {
    abstract void work();
    void breakTime() {
        System.out.println("Break time!");
    }
}
```

---

8. Array & ArrayList

Definition:
Array is a fixed-size data structure. ArrayList is dynamic.

ArrayList Example:

```
ArrayList<Employee> list = new ArrayList<>();
list.add(new Employee(101, "Ram", 55000));
```

---

9. Exception Handling

Definition:
Managing runtime errors using try, catch, finally.

Example:

```
try {
    int x = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Can't divide by 0");
}
```

---

10. File I/O

Definition:
Java allows reading/writing files using File, FileReader, BufferedWriter.

Example:

```
FileWriter fw = new FileWriter("emp.txt");
fw.write("Name: Ram, Salary: 50000");
fw.close();
```

==

ADVANCED JAVA TOPICS

11. JDBC (Java Database Connectivity)

Definition:
Used to connect Java apps with relational databases like MySQL.

Steps:

1. Load Driver
2. Establish Connection
3. Execute SQL
4. Close Connection

Example:

```
Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root",
"pass");

PreparedStatement ps = con.prepareStatement("INSERT INTO employee VALUES (?, ?,
?)");

ps.setInt(1, 101);

ps.setString(2, "Ram");

ps.setDouble(3, 60000);

ps.executeUpdate();

con.close();
```

---

12. Servlets + JSP (Basics)

Servlet:

```
@WebServlet("/add")

public class AddServlet extends HttpServlet {

    protected void doPost(HttpServletRequest req, HttpServletResponse res) {

        String name = req.getParameter("name");

        // DB logic

    }

}
```

JSP:

```
<form action="add" method="post">

  Name: <input type="text" name="name">

</form>
```

---

13. JPA (Java Persistence API)

Definition: ORM framework used to map Java classes to DB tables.

Example:

```
@Entity
public class Employee {
   @Id
   @GeneratedValue
   private int id;
   private String name;
   private double salary;
}
```

Repository:

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {}
```

---

14. Spring Boot (REST API)

Definition: Spring Boot simplifies building production-ready Java apps with minimal setup.

Controller Example:

```
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

   @Autowired
   private EmployeeService service;

   @PostMapping
   public Employee add(@RequestBody Employee emp) {
      return service.save(emp);
   }

   @GetMapping
   public List<Employee> getAll() {
      return service.getAll();
   }
}
```

Mini Assignment: **PROJECTS**

Project 1: Core Java Console App

- CRUD operations using Scanner + ArrayList
- Save/load from file using FileWriter, BufferedReader

Project 2: JDBC CLI App

- Connect to MySQL
- Perform CRUD on employee table

Project 3: Spring Boot REST API

- Use JPA + Repository + Controller + Service
- Expose REST endpoints for Employee

==

15. Multithreading

Definition:
Multithreading allows concurrent execution of two or more threads (lightweight processes) to make applications faster and responsive.

- Thread class vs Runnable
- Synchronization
- Thread Lifecycle
- Example: Background Salary Processing for Employees

Example 1 – Thread using extends Thread

```
class SalaryProcessor extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println("Processing salary for employee " + i);
        }
    }
}


public class Main {
    public static void main(String[] args) {
        SalaryProcessor t1 = new SalaryProcessor();
        t1.start();
    }
}
```

Example 2 – Thread using implements Runnable

```
class Task implements Runnable {
    public void run() {
        System.out.println("Running task in separate thread");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new Task());
        t.start();
    }
}
```

Assignment:

- Create a thread to print the list of all employees with salary above ₹50,000.

---

16. Collections Framework

Definition:
Collections are data structures used to store and manipulate groups of data dynamically. Java provides List, Set, Map etc.

- List, Set, Map interfaces
- ArrayList, HashMap, TreeSet usage with Employee
- Sorting with Comparator & Comparable

Example 1 – ArrayList<Employee>

```
ArrayList<Employee> list = new ArrayList<>();
list.add(new Employee(101, "Alice", 50000));
list.add(new Employee(102, "Bob", 60000));
```

Example 2 – Sort using Comparator

```
Collections.sort(list, (e1, e2) -> e1.salary > e2.salary ? -1 : 1);
```

Example 3 – HashMap<Integer, Employee>

```
Map<Integer, Employee> empMap = new HashMap<>();
empMap.put(101, new Employee(101, "John", 50000));
System.out.println(empMap.get(101).name);
```

Assignment:

- Sort employees by name and print.

- Use TreeSet<Employee> with Comparator for sorted view.

---

17. Enum

Definition:
Enum is a special class that represents a group of constants.

- Define constant values (like DEPARTMENT, DESIGNATION)
- Use Enums in Employee class

Example – Enum with Employee

```
enum Department {
   HR, DEV, SALES
}


class Employee {
   int id;
   String name;
   Department dept;


   public Employee(int id, String name, Department dept) {
      this.id = id;
      this.name = name;
      this.dept = dept;
   }
}
```

Assignment:

- Create enum Level {JUNIOR, MID, SENIOR} and use in Employee class.

---

18. Wrapper Classes

Definition:
Wrapper classes convert primitive types to objects — used in collections and generics.

- Convert primitives to objects (int → Integer)
- Useful for Collections and Generics

Example:

```
int id = 101;
Integer empId = Integer.valueOf(id); // Boxing
int realId = empId.intValue();       // Unboxing
```

Assignment:

- Accept employee data from console using wrapper classes and store in ArrayList<Employee>.

---

19. Lambda Expressions

Definition:
Lambda expressions provide a clear and concise way to represent one method interface using an expression.

- Write compact code for iteration, filtering, sorting
- Example: Sort List<Employee> using lambda

Example – Sorting with Lambda

```
List<Employee> list = new ArrayList<>();

list.add(new Employee(101, "A", 45000));

list.add(new Employee(102, "B", 60000));


list.sort((e1, e2) -> Double.compare(e2.salary, e1.salary));
```

Filter Example:

```
list.stream()

   .filter(emp -> emp.salary > 50000)

   .forEach(e -> System.out.println(e.name));
```

Assignment:

- Use lambda to print employees whose names start with “A”.

---

20. Swagger + Validation in Spring Boot

- Document REST APIs using Swagger UI
- Use @Valid, @NotNull, @Min, @Email for field validation

Swagger Setup:

- Add dependency:

```
xml

<dependency>

   <groupId>org.springdoc</groupId>

   <artifactId>springdoc-openapi-ui</artifactId>

   <version>1.6.14</version>

</dependency>
```

- Access Swagger UI at: http://localhost:8080/swagger-ui.html

Validation Annotations:

```
@Entity
public class Employee {
   @Id
   @GeneratedValue
   private int id;

   @NotBlank(message = "Name is required")
   private String name;

   @Min(value = 10000, message = "Minimum salary is 10000")
   private double salary;
}
```

Enable validation in controller:

```
@PostMapping("/add")
public ResponseEntity<Employee> add(@Valid @RequestBody Employee emp) {
   return ResponseEntity.ok(service.save(emp));
}
```

Assignment:

- Add @Email field in Employee for official email.
- Try sending invalid input via Swagger and verify validation.

===

21. Design Patterns (OOP Best Practices)

Definition:
Design Patterns are standard solutions to common software design problems.

Example – Singleton Pattern (One instance)

```
public class EmployeeService {
   private static EmployeeService instance = new EmployeeService();

   private EmployeeService() {}
```

```
    public static EmployeeService getInstance() {
        return instance;
    }
}
```

Example – Factory Pattern

```
interface Employee {
    void showRole();
}

class Developer implements Employee {
    public void showRole() {
        System.out.println("Developer role.");
    }
}

class EmployeeFactory {
    public static Employee getEmployee(String type) {
        if (type.equals("DEV")) return new Developer();
        return null;
    }
}
```

Assignment:

- Implement a Factory for Manager, Intern, HR.

---

22. Deployment – WAR & Docker

WAR Deployment (Apache Tomcat)

- Package Spring Boot using:

```
mvn clean package
```

- Copy .war to /webapps of Apache Tomcat
- Start server and access: http://localhost:8080/app

Dockerize Spring Boot App

Step 1: Create Dockerfile

Dockerfile

```
FROM openjdk:17

COPY target/app.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]
```

Step 2: Build and Run

```
docker build -t employee-api .

docker run -p 8080:8080 employee-api
```

Assignment:

- Deploy your Spring Boot Employee API using Docker

---

23. JUnit Testing (Unit Testing in Java)

Definition:
JUnit is a framework used to write repeatable unit tests.

JUnit 5 Test Example

```
public class EmployeeService {
    public double annualSalary(double monthlySalary) {
        return monthlySalary * 12;
    }
}


import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;


public class EmployeeServiceTest {
    @Test
    void testAnnualSalary() {
        EmployeeService service = new EmployeeService();
        assertEquals(120000, service.annualSalary(10000));
    }
}
```

Assignment:

- Write tests for Employee name not null, salary > 10000.

==

Spring Boot MVC – CRUD Using Annotations

Tech Stack:

- Spring Boot
- Spring Data JPA
- H2/MySQL
- Spring Web (REST API)
- Lombok (optional)

Project Structure:

```
com.example.employeeapi
│
├── controller
│   └── EmployeeController.java
├── service
│   └── EmployeeService.java
│   └── EmployeeServiceImpl.java
├── repository
│   └── EmployeeRepository.java
├── model
│   └── Employee.java
├── EmployeeApiApplication.java
```

---

1. Employee Model

```
package com.example.employeeapi.model;


import jakarta.persistence.*;
import jakarta.validation.constraints.*;


@Entity
public class Employee {


    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @NotBlank(message = "Name is mandatory")
    private String name;

    @Min(value = 10000, message = "Salary must be at least ₹10000")
    private double salary;

    @Email
    private String email;

    // Getters and Setters (or use Lombok @Data)
}
```

---

2. EmployeeRepository

```
package com.example.employeeapi.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.employeeapi.model.Employee;

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
}
```

---

3. EmployeeService Interface

```
package com.example.employeeapi.service;

import com.example.employeeapi.model.Employee;
import java.util.List;

public interface EmployeeService {
    Employee save(Employee emp);
    Employee update(Employee emp, int id);
    void delete(int id);
    List<Employee> getAll();
    Employee getById(int id);
}
```

---

4. EmployeeServiceImpl

```
package com.example.employeeapi.service;

import com.example.employeeapi.model.Employee;
import com.example.employeeapi.repository.EmployeeRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeRepository repo;

    @Override
    public Employee save(Employee emp) {
        return repo.save(emp);
    }

    @Override
    public Employee update(Employee emp, int id) {
        Employee existing = repo.findById(id).orElseThrow();
        existing.setName(emp.getName());
        existing.setSalary(emp.getSalary());
        existing.setEmail(emp.getEmail());
        return repo.save(existing);
    }
    @Override
    public void delete(int id) {
        repo.deleteById(id);
    }

    @Override
    public List<Employee> getAll() {
        return repo.findAll();
    }

    @Override
    public Employee getById(int id) {
        return repo.findById(id).orElseThrow();
    }
}
```

---

5. EmployeeController

```
package com.example.employeeapi.controller;

import com.example.employeeapi.model.Employee;
import com.example.employeeapi.service.EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.*;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    @Autowired
    private EmployeeService service;
```

```
  @PostMapping
  public ResponseEntity<Employee> create(@Validated @RequestBody Employee emp) {
    return new ResponseEntity<>(service.save(emp), HttpStatus.CREATED);
  }

  @GetMapping
  public List<Employee> getAll() {
    return service.getAll();
  }

  @GetMapping("/{id}")
  public ResponseEntity<Employee> getOne(@PathVariable int id) {
    return ResponseEntity.ok(service.getById(id));
  }

  @PutMapping("/{id}")
  public ResponseEntity<Employee> update(@PathVariable int id, @RequestBody
Employee emp) {
    return ResponseEntity.ok(service.update(emp, id));
  }

  @DeleteMapping("/{id}")
  public ResponseEntity<Void> delete(@PathVariable int id) {
    service.delete(id);
    return ResponseEntity.noContent().build();
  }
}
```

Annotations Used:

- @Entity, @Id, @GeneratedValue, @Min, @Email – JPA + Validation
- @RestController, @RequestMapping, @PostMapping, etc. – MVC Controller
- @Autowired – Dependency Injection
- @Validated, @RequestBody, @PathVariable – Request binding and validation

---

Testing with Swagger UI:

Add this to pom.xml:

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.0.4</version>
</dependency>
```

Run app and access:

http://localhost:8080/swagger-ui/index.html

Java 8 Features with Employee Examples

1. Lambda Expressions

Definition: Enables writing anonymous methods in a short way.

```
List<Employee> list = Arrays.asList(
    new Employee(101, "Ram", 50000),
    new Employee(102, "Sam", 60000)
);

// Print names using lambda
list.forEach(e -> System.out.println(e.getName()));
```

---

2. Functional Interface

Definition: An interface with only one abstract method.

```
@FunctionalInterface
interface BonusCalculator {
    double calculate(double salary);
}

BonusCalculator b = s -> s * 0.10;
System.out.println("Bonus: " + b.calculate(50000));
```

---

3. Default & Static Methods in Interfaces

```
interface EmployeeService {
    default void printWelcome() {
        System.out.println("Welcome Employee");
    }
    static void companyName() {
        System.out.println("SATz Corp");
    }
}
```

---

4. Method References

```
list.forEach(System.out::println); // Reference to println
list.forEach(Employee::printName); // Custom method reference
```

---

5. Streams API

- Used to process collections in a functional style

```
// Filter employees with salary > 55000
list.stream()
    .filter(e -> e.getSalary() > 55000)
    .forEach(e -> System.out.println(e.getName()));

// Get list of employee names
List<String> names = list.stream()
    .map(Employee::getName)
    .collect(Collectors.toList());
```

---

6. Optional Class

```
Optional<Employee> emp = Optional.of(new Employee(1, "John", 50000));
emp.ifPresent(e -> System.out.println(e.getName()));

Optional<Employee> emp2 = Optional.empty();
System.out.println(emp2.orElse(new Employee(0, "Default", 0)).getName());
```

---

7. Collectors

```
Map<String, Double> nameSalaryMap = list.stream()
    .collect(Collectors.toMap(Employee::getName, Employee::getSalary));

Double averageSalary = list.stream()
```

```
   .collect(Collectors.averagingDouble(Employee::getSalary));
```

---

8. Date and Time API

```
LocalDate joinDate = LocalDate.now();
System.out.println("Join Date: " + joinDate);

LocalDate dob = LocalDate.of(1995, Month.JUNE, 25);
Period age = Period.between(dob, LocalDate.now());
System.out.println("Age: " + age.getYears());
```

---

9. Predicate, Function, Consumer

```
Predicate<Employee> highSalary = e -> e.getSalary() > 50000;
Function<Employee, String> empName = e -> e.getName();
Consumer<Employee> printEmp = e -> System.out.println(e.getName());

list.stream().filter(highSalary).forEach(printEmp);
```

==

Java 17 Features with Employee Class Examples

1. Sealed Classes

Definition: Restrict which classes can extend a superclass.

```
public sealed class Employee permits Manager, Developer {}

final class Manager extends Employee {}
final class Developer extends Employee {}
```

Use this to control your class hierarchy and prevent unintended extensions.

---

2. Records (Immutable Data Classes)

Definition: Concise syntax to declare a class whose main purpose is to store data.

```
public record EmployeeRecord(int id, String name, double salary) {}

public class Main {
   public static void main(String[] args) {
      EmployeeRecord emp = new EmployeeRecord(101, "John", 55000);
      System.out.println(emp.name() + " - ₹" + emp.salary());
```

```
    }
}
```

No need to write constructors, getters, equals, hashCode, or toString.

---

3. Pattern Matching for instanceof

```
Object obj = new Employee(1, "Sam", 50000);


if (obj instanceof Employee e) {
    System.out.println("Name: " + e.getName());
}
```

📌 Cleaner code without explicit casting.

---

4. Text Blocks (Java 15+)

Definition: Multi-line string literal.

```
String json = """
    {
      "id": 101,
      "name": "Alice",
      "salary": 50000
    }
    """;
System.out.println(json);
```

---

5. Switch Expressions (Preview in Java 17)

```
String role = "DEV";
String message = switch (role) {
    case "HR" -> "Welcome HR!";
    case "DEV" -> "Welcome Developer!";
    default -> "Welcome User!";
};
System.out.println(message);
```

---

6. Enhanced NullPointerException (NPE) Messages

Now Java tells you **which variable was null** in a chain call:

```
emp.getDepartment().getManager().getName();  // Now traces exact null
```

🔹 7. JEP 409 – Pattern Matching for switch (Preview)

Combines switch and pattern matching:

```
static String process(Object obj) {
    return switch (obj) {
        case String s -> "It's a string: " + s;
        case Integer i -> "It's an integer: " + i;
        case Employee e -> "Employee: " + e.getName();
        default -> "Unknown type";
    };
}
```

🔹 8. Helpful JVM Features

- **Improved performance**
- **Sealed interfaces**
- **JEP 356: Enhanced Pseudo-Random Number Generators**

==

Java Developer Resume Template

Career Objective

A passionate and results-driven Java Developer with *X years* of experience in building scalable backend systems using Java, Spring Boot, REST APIs, and MySQL. Looking to leverage my skills to develop enterprise-grade applications in a dynamic team environment.

Technical Skills

Core Java	**Collections, Multithreading, OOP, JDBC, Exception Handling**
Frameworks	Spring Boot, Spring MVC, JPA, Hibernate
Web & API	RESTful APIs, JSON, Swagger, Postman
Database	MySQL, MongoDB, H2
Tools	Git, Maven, Docker, JUnit
Deployment	Tomcat, Docker, WAR packaging
IDEs	IntelliJ IDEA, Eclipse

Projects

1. Employee Management System – Spring Boot + JPA

- Developed a RESTful CRUD API for managing employee records using Spring Boot and Spring Data JPA.
- Implemented validations with Hibernate Validator and tested APIs using Postman.
- Added Swagger UI for API documentation.
- Tech Used: Java, Spring Boot, JPA, MySQL, Swagger, Maven

2. Core Java Payroll Console App

- Built a console-based payroll system using Java and OOP principles.
- Features: Add, update, delete employee, calculate salary, and file I/O persistence.

3. Dockerzed Spring Boot App

- Dockerized a Spring Boot application and deployed it in a containerized environment.
- Used Dockerfile, Docker Compose, and exposed the app to the host system.

****