WIPRO TRAINIING NOTES

OOPS Concepts:

OOP (Object-Oriented Programming) has 4 major pillars:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

1. Abstraction

- **Definition**: Hiding internal details and showing only the necessary functionality to the user.
- Achieved using **abstract classes** or **interfaces**.

Example:

```
abstract class Vehicle {
   abstract void start(); // abstract method (no body)
   void fuel() {
      System.out.println("Fueling up...");
   }
}

class Car extends Vehicle {
   @Override
   void start() {
      System.out.println("Car starts with a key.");
   }
}

public class Main {
   public static void main(String[] args) {
      Vehicle v = new Car(); // abstraction: reference is Vehicle, object is Car
      v.start(); // prints: Car starts with a key.
      v.fuel();  // prints: Fueling up...
   }
}
```

2. Encapsulation

- Wrapping data (fields) and methods into a single unit (class).
- Protects data using private variables and allows controlled access with getters/setters.

Example:

```
class Employee {
   private String name;   // private: cannot be accessed directly
   private double salary;

   // getter
   public String getName() {
      return name;
   }

   // setter
   public void setName(String name) {
      this.name = name;
   }

   public void setSalary(double salary) {
      if (salary > 0) { // validation
         this.salary = salary;
      }
   }

   public double getSalary() {
      return salary;
   }
}

public class Main {
   public static void main(String[] args) {
      Employee emp = new Employee();
      emp.setName("Ravi");
      emp.setSalary(50000);

      System.out.println(emp.getName() + " earns " + emp.getSalary());
   }
}
```

3. Inheritance

- Allows a child class to reuse code from a parent class.
- Uses the extends keyword.

Example:

```
class Order {
   String orderId;
```

```
    void processOrder() {
        System.out.println("Processing generic order...");
    }
}

class OnlineOrder extends Order {
    void processOrder() {
        System.out.println("Processing online order with payment gateway...");
    }
}

public class Main {
    public static void main(String[] args) {
        OnlineOrder order = new OnlineOrder();
        order.processOrder(); // prints: Processing online order with payment gateway...
    }
}
```

4. Polymorphism

- Same method name, different behavior.
- Two types:
    1. **Compile-time polymorphism (Overloading)**

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println(c.add(5, 10));     // 15
        System.out.println(c.add(3.2, 4.5));  // 7.7
    }
}
```

    2. **Runtime polymorphism (Overriding)**

```
class Animal {
    void speak() {
        System.out.println("Generic sound...");
```

```
    }
}

class Dog extends Animal {
    @Override
    void speak() {
        System.out.println("Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog(); // parent reference, child object
        a.speak(); // prints: Woof!
    }
}
```

Types of Inheritance

1.Single Inheritance

```
class Parent {
    void show() { System.out.println("Parent method"); }
}
class Child extends Parent {
    void display() { System.out.println("Child method"); }
}
public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.show();    // Parent method
        c.display(); // Child method
    }
}
```

2.Multilevel Inheritance

```
class Grandparent {
    void greet() { System.out.println("Hello from Grandparent"); }
}
class Parent extends Grandparent {
    void say() { System.out.println("Hello from Parent"); }
}
class Child extends Parent {
    void speak() { System.out.println("Hello from Child"); }
}
```

```
public class Main {
   public static void main(String[] args) {
      Child c = new Child();
      c.greet();
      c.say();
      c.speak();
   }
}
```

3.Multiple Inheritance

Java doesn't allow multiple inheritance with classes to avoid the diamond problem, but interfaces solve this.

```
interface A {
   void methodA();
}
interface B {
   void methodB();
}
class C implements A, B {
   public void methodA() { System.out.println("From A"); }
   public void methodB() { System.out.println("From B"); }
}
public class Main {
   public static void main(String[] args) {
      C obj = new C();
      obj.methodA();
      obj.methodB();
   }
}
```

4.Hierarchial Inheritance

```
class Parent {
   void greet() { System.out.println("Hello from Parent"); }
}
class Child1 extends Parent {}
class Child2 extends Parent {}

public class Main {
   public static void main(String[] args) {
      Child1 c1 = new Child1();
      Child2 c2 = new Child2();
      c1.greet();
      c2.greet();
```

```
    }
}
```

5.Hybrid Inheritance

```
interface A {
    void methodA();
}
class B {
    void methodB() { System.out.println("From B"); }
}
class C extends B implements A {
    public void methodA() { System.out.println("From A"); }
}
public class Main {
    public static void main(String[] args) {
        C obj = new C();
        obj.methodA();
        obj.methodB();
    }
}
```

Super Keyword:

- The super keyword in Java is a reference variable used to refer to the immediate parent class object.
- super() must be the first statement inside a constructor.

1.To Call Parent Class Constructor

When a child class object is created, its parent constructor is also called.

- If not written, the compiler automatically adds super() as the first statement.
- If you want to call a specific constructor from the parent, you use super(parameters).

Example

```
class Parent {
    Parent() {
        System.out.println("Parent constructor");
    }
    Parent(String msg) {
        System.out.println("Parent constructor with: " + msg);
    }
```

```
}

class Child extends Parent {
   Child() {
      super("Hello from Child"); // calls parameterized parent constructor
      System.out.println("Child constructor");
   }
}

public class Main {
   public static void main(String[] args) {
      Child c = new Child();
   }
}
```

Output
Parent constructor with: Hello from Child
Child constructor

2. To Call Parent Class Method

If a child class overrides a parent class method, we can still call the parent version using super.methodName().

Example

```
class Parent {
   void display() {
      System.out.println("Parent display method");
   }
}

class Child extends Parent {
   void display() {
      super.display(); // calls Parent's display()
      System.out.println("Child display method");
   }
}

public class Main {
   public static void main(String[] args) {
      Child c = new Child();
      c.display();
   }
}
```

Output
Parent display method
Child display method

Inner Classes in Java

- An inner class is a class defined inside another class.
- It helps in logically grouping classes that are only used in one place and increases encapsulation.

Types of Inner Classes

Java provides 4 types of inner classes:

1. Non-static Member Inner Class

- Defined inside a class, but outside methods.
- Always tied to an instance of the outer class.
- Can access all members of the outer class, including private.

Example

```
class Hello {
   private String greeting = "Welcome!";

   // Non-static inner class
   class Message {
      void show() {
         System.out.println("Hello from inner class");
         System.out.println("Greeting: " + greeting); // can access private
      }
   }
}

public class Main {
   public static void main(String[] args) {
      Hello h = new Hello();
      Hello.Message m = h.new Message(); // need outer instance
      m.show();
   }
}
```

Output
Hello from inner class
Greeting: Welcome!

2. Static Nested Class

- Declared with the static keyword.
- Does not need an instance of the outer class.
- Can only access static members of the outer class.

Example

```
class Outer {
   static String appName = "PlayStore";

   // Static nested class
   static class Nested {
      void display() {
         System.out.println("App Name: " + appName);
      }
   }
}

public class Main {
   public static void main(String[] args) {
      Outer.Nested obj = new Outer.Nested(); // directly created
      obj.display();
   }
}
```

Output

App Name: PlayStore

3. Local Inner Class

- Declared inside a method, constructor, or block.
- Scope is limited to that method/block only.
- Can use final or effectively-final variables from the method.

Example

```
class Outer {
   void process() {
      final int x = 10;

      // Local inner class inside method
      class Local {
         void show() {
            System.out.println("Local inner class, value = " + x);
         }
```

```
        }

        Local l = new Local();
        l.show();
    }
}

public class Main {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.process();
    }
}
```

Output
Local inner class, value = 10

4. Anonymous Inner Class

- No name.
- Used to create single-use implementations of abstract classes or interfaces.
- Very common in event listeners and thread creation.

Example

```
interface Greeting {
    void sayHello();
}

public class Main {
    public static void main(String[] args) {
        // Anonymous inner class
        Greeting g = new Greeting() {
            public void sayHello() {
                System.out.println("Hello from Anonymous Inner Class");
            }
        };

        g.sayHello();
    }
}
```

Output
Hello from Anonymous Inner Class

Java Keywords

Final

- Final variable: assigned once. For references, the reference can't be changed but internal state may change.
- Final method: cannot be overridden by subclasses.
- Final class: cannot be subclassed (e.g., String).
- Blank final: assigned in constructor or static block.

Example

```
final class ConstantVariables {
   public static final double gst = 0.18;
   public static final double pi = 3.14;
}

class Salary {
   public final double salary(double basicPay) {          // final method
      return basicPay * ConstantVariables.gst;
   }
}
```

This

- Refers to current object. Used to disambiguate fields vs parameters and to call other constructors: this(...). Cannot be used in static context.

Transient

- Mark field transient to skip it during serialization (e.g., password fields).

Instanceof

- Check type at runtime before casting

```
if (obj instanceof Employee) {
   Employee e = (Employee) obj;
}
```

Collections

A framework is a set of ready-made classes and interfaces that give a structure and common behavior so developers don't reimplement common tasks.

A collection is a group of objects (elements). Collections handle storage, retrieval, update and traversal of groups of objects.

Array vs Collection

- Array -- can hold primitives & objects but is fixed-size and homogeneous (same declared type).
- Collection (java.util) -- dynamic, richer API (add/remove/search/sort). Use List, Set, Map, etc.

Two types of containers in Collections framework

Collection (stores elements)

- Interfaces: Collection, List, Set, Queue
- Implementations: ArrayList, LinkedList, HashSet, TreeSet, PriorityQueue

Map (stores key -- value pairs)

- Interface: Map
- Implementations: HashMap, LinkedHashMap, TreeMap, ConcurrentHashMap

Common choices

- ArrayList<T> — fast random access, good for many reads.
- LinkedList<T> — good for many inserts/deletes in middle.
- HashSet<T> — no duplicates, average constant-time operations.
- TreeSet<T> — sorted set.
- HashMap<K,V> — fast key lookup.

Exception Handling

Root class: Throwable

- Error (serious JVM/system issues): e.g., OutOfMemoryError, StackOverflowError — usually not recoverable.
- Exception (application-level): can be handled with try-catch-finally. Subclassed into checked (compile-time) and unchecked/runtime exceptions.

Checked vs Unchecked

- Checked: IOException, ClassNotFoundException — compiler forces handling or declaring throws.
- Unchecked: NullPointerException, IndexOutOfBoundsException, ArithmeticException. Not required to declare.

Try-Catch-Finally

```
try {
   int x = 10 / divisor;  // could throw ArithmeticException
} catch (ArithmeticException e) {
   System.out.println("Cannot divide by zero: " + e.getMessage());
} finally {
   System.out.println("Runs always (cleanup)");
}
```

Example: File reading (checked exception)

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public void readFile(String path) throws IOException {
   try (BufferedReader br = new BufferedReader(new FileReader(path))) {
      String line;
      while ((line = br.readLine()) != null) {
         System.out.println(line);
      }
   } // try-with-resources automatically closes file
}
```

Common runtime exceptions

- ArithmeticException — divide by zero.
- NullPointerException — using null reference.
- ClassCastException — invalid cast.
- IndexOutOfBoundsException — invalid index access.
- IllegalArgumentException / NumberFormatException — bad method input.

Best practices

- Catch specific exceptions first; avoid broad catch(Exception e) unless you log/handle properly.
- Use finally or try-with-resources to clean up resources.

Multithreading

Thread lifecycle: New → Runnable → Running → Blocked/Waiting → TimedWaiting (sleep/join) → Terminated.

Two ways to create:

1. extends Thread and override run()
2. implements Runnable and pass to new Thread(runnable)

```
class Task implements Runnable {
   public void run() {
      System.out.println("Running in " + Thread.currentThread().getName());
   }
}

Thread t = new Thread(new Task(), "Worker-1");
t.start(); // schedules thread; JVM will call run() on a new thread
```

Design Patterns

Factory Pattern

- Provides objects based on input without exposing creation logic.

```
interface Payment { void pay(double amount); }

class CreditCard implements Payment {
   public void pay(double amount) {
      System.out.println("Payment done using credit card: " + amount);
   }
}

class Upi implements Payment {
   public void pay(double amount) {
      System.out.println("Payment done using UPI: " + amount);
   }
}

class PaymentFactory {
   public static Payment getPayment(String type) {
      switch(type) {
         case "credit": return new CreditCard();
         case "upi":    return new Upi();
         default: throw new IllegalArgumentException("Unexpected: " + type);
      }
   }
```

```
}

// usage
Payment p = PaymentFactory.getPayment("credit");
p.pay(12000);
```

Prototype Pattern

- Cloneable documents to avoid expensive re-creation.

```
abstract class Document implements Cloneable {
   public String content;
   public Document(String content) { this.content = content; }
   public abstract void print();
   @Override
   public Document clone() {
      try { return (Document) super.clone(); } catch (CloneNotSupportedException e) {
return null; }
   }
}

class Invoice extends Document {
   public Invoice(String content) { super(content); }
   public void print() { System.out.println("Invoice: " + content); }
}

// usage
Invoice original = new Invoice("Original invoice 15000");
Invoice copy = (Invoice) original.clone();
copy.content = "Cloned invoice + GST";
copy.print();
```

Builder Pattern (immutable product)

- Build complex object step-by-step with readable API.

```
final class Product {
   private final String name;
   private final double price;
   private final String configuration;

   private Product(ProductBuilder b) {
      this.name = b.name; this.price = b.price; this.configuration = b.configuration;
   }
```

```
    public static class ProductBuilder {
        private String name; private double price; private String configuration;
        public ProductBuilder setName(String n) { name = n; return this; }
        public ProductBuilder setPrice(double p) { price = p; return this; }
        public ProductBuilder setConfiguration(String c) { configuration = c; return this;
}
        public Product build() { return new Product(this); }
    }

    public void show() { System.out.println("Product: " + name + " Price: " + price); }
}
```

Multitasking

- OS feature to run multiple processes concurrently (time-sharing CPU).
- Each process has its own separate memory space.
- **Example:** Running a browser, music player, and Word on one computer.

Multithreading

- A single process is divided into multiple threads (lightweight execution units).
- Threads share the process's memory/resources but have their own stack, state, priority.
- Example: Web server creating a new thread for each client request.
- Synchronous: One thread completes before another starts (e.g., deposit/withdraw).
- Asynchronous: Multiple threads run concurrently to save waiting time (e.g., file copy with progress bar).

Multiprocessing

- Uses multiple processors/CPU cores to run tasks truly in parallel.
- Each process has dedicated memory.
- Example: Gaming, video editing (tasks split across cores for faster execution).

Threads in Java

- Java uses preemptive scheduling → JVM scheduler assigns time slices to threads.
- Each thread has: individual stack, program counter, state, priority, but shares heap memory.

Ways to create a thread:

1. Extend the Thread class.
2. Implement Runnable and pass it to a Thread object.

Thread Lifecycle (States)

1. **New** → new Thread() (created but not started).
2. **Runnable** → start() called (ready to run, but waiting for scheduler).
3. **Running** → actually executing run() method.
4. **Blocked** → waiting for access to synchronized block/method.
5. **Waiting** → waits indefinitely (wait()), resumes with notify()/notifyAll().
6. **Timed Waiting/Paused** → sleeps for time (sleep(ms), wait(ms), join(ms)).
7. **Terminated/Dead** → finished execution (stop() or run completes).

Hibernate Querying Approaches

1. HQL (Hibernate Query Language)

- Hibernate's own object-oriented SQL-like language.
- Works with entity class names and properties instead of table and column names.
- Example:

```
Query q = session.createQuery("from Employee e where e.eid = :id");
q.setParameter("id", 101);
Employee emp = (Employee) q.uniqueResult();
```

2. Native SQL

- Writing plain SQL queries directly to the DB.
- Similar to JDBC queries, but Hibernate can still map the result back to entities.
- Example:

```
Query q = session.createSQLQuery("SELECT * FROM employee WHERE eid =
:id").addEntity(Employee.class);
q.setParameter("id", 101);
Employee emp = (Employee) q.uniqueResult();
```

3. Criteria API (part of JPA, implemented by Hibernate)

- Instead of writing query strings, you build queries using Java objects.
- More type-safe, reusable, dynamic (good for filters & search forms).
- Example:

```
Searching employee by name and joining date range –
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> root = cq.from(Employee.class);
// Example filter: name and date
Predicate nameFilter = cb.equal(root.get("name"), "Ravi");
Predicate dateFilter = cb.between(root.get("joinDate"), startDate, endDate);
// Combine filters
cq.select(root).where(cb.and(nameFilter, dateFilter));
```

```
Query<Employee> q = session.createQuery(cq);
List<Employee> results = q.getResultList();
```

Hibernate Mapping

Mapping tells Hibernate how Java objects relate to database tables.

(a) Pure Hibernate setup

- Uses hibernate.cfg.xml file → SessionFactory (Singleton) → Session (per job/transaction).
- Transactions managed manually.
- Example DBs: H2 (in-memory for testing), MySQL.

(b) Spring + Hibernate (JPA)

- Uses LocalSessionFactoryBean instead of building SessionFactory manually.
- Transactions handled by HibernateTransactionManager.
- Easier and integrates with Spring's dependency injection.

Types of Mapping

1) One-to-One (Unidirectional)

- Example: Student ↔ ReportCard.
- Student is owner, ReportCard is child.

```
@Entity
public class Student {
   @Id
   private int id;
   private String name;

   @OneToOne(cascade = CascadeType.ALL)
   @JoinColumn(name = "report_id") // FK column in Student table
   private ReportCard reportCard;
}

@Entity
public class ReportCard {
   @Id
   private int id;
   private String grade;
}
```

- If we save student → ReportCard is auto-saved because of cascade.
- Without cascade → We must save both manually.

Cascade types:

- PERSIST → Save child when saving parent.
- MERGE → Update child when updating parent.
- REMOVE → Delete child when deleting parent.
- ALL → Apply all above.

2) One-to-One (Bidirectional)

- Both entities reference each other.
- Student is owner, ReportCard is inverse side.

```
@Entity
public class Student {
   @Id
   private int id;
   private String name;

   @OneToOne(cascade = CascadeType.ALL)
   @JoinColumn(name = "report_id")
   private ReportCard reportCard;
}

@Entity
public class ReportCard {
   @Id
   private int id;
   private String grade;

   @OneToOne(mappedBy = "reportCard") // tells Hibernate Student owns it
   private Student student;
}
```

Now you can do:

```
student.getReportCard();
reportCard.getStudent();
```

- Hibernate knows Student is the owner (because of @JoinColumn).
- Only **owner side** updates the foreign key in DB.

b) One-to-Many Mapping

- Unidirectional One-to-Many (Student → Courses)
- Student knows Courses, but Course doesn't know Student.

Student Entity

```
@Entity
public class Student {
   @Id
   private int id;
   private String name;

   @OneToMany(cascade = CascadeType.ALL)
   @JoinColumn(name = "student_id") // FK in Course table
   private List<Course> courses = new ArrayList<>();
}
```

Course Entity

```
@Entity
public class Course {
   @Id
   private int id;
   private String courseName;
}
```

- Hibernate creates a student_id FK column in the Course table.
- Only navigation: student.getCourses()

3.Bidirectional One-to-Many (Student ↔ Courses)

Both Student and Course know each other.

Student Entity

```
@Entity
public class Student {
   @Id
   private int id;
   private String name;

   @OneToMany(mappedBy = "student", cascade = CascadeType.ALL)
   private List<Course> courses = new ArrayList<>();
}
```

Course Entity

```
@Entity
public class Course {
   @Id
```

```
    private int id;
    private String courseName;

    @ManyToOne
    @JoinColumn(name = "student_id")  // FK in Course table
    private Student student;
}
```

Navigation both ways:

- student.getCourses()
- course.getStudent()

4.Many-to-Many Mapping

- Unidirectional Many-to-Many (Student → Courses)
- Students can enroll in multiple Courses, but Course doesn't know Students.

Student Entity

```
@Entity
public class Student {
    @Id
    private int id;
    private String name;

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses = new ArrayList<>();
}
```

Course Entity

```
@Entity
public class Course {
    @Id
    private int id;
    private String courseName;
}
```

- Hibernate creates a **join table** student_course(student_id, course_id).
- Navigation: student.getCourses()

5.Bidirectional Many-to-Many (Student ↔ Courses)

Both Student and Course know each other.

Course Entity

```
@Entity
public class Course {
   @Id
   private int id;
   private String courseName;

   @ManyToMany(mappedBy = "courses")
   private List<Student> students = new ArrayList<>();
}
```

Navigation both ways:

- student.getCourses()
- course.getStudents()

Spring Fundamentals

What is Spring?

- Spring = A Java framework to build enterprise-level applications.
- It manages objects for you (through IoC + Dependency Injection) and adds features like AOP, Security, Data Access, Microservices.

Spring Modules

Module	Purpose
Spring Core	Provides IoC container → manages objects with Dependency Injection.
Spring MVC	Build Web Applications & REST APIs.
Spring Boot	Auto-configuration, starter dependencies, fast development.
Spring AOP	Aspect Oriented Programming → logging, security, transactions.
Spring Security	Authentication & Authorization (login, roles).
Spring Data	Simplifies DB access (JDBC, JPA, Hibernate).
Spring Cloud	Tools for Microservices & distributed systems.

Inversion of Control (IoC)

Normally, developers create objects like this:

```
Student s = new Student();
```

But in Spring, IoC container creates and manages the object for you. You just ask for it → Spring gives it.

Dependency Injection (DI)

A design pattern where **Spring injects dependencies into a class** instead of the class creating them.

Types of DI:

1. **Constructor Injection** → Best when dependency is required.
2. **Setter Injection** → Best when dependency is optional.

Constructor Injection Example:

```
@Component
class Student {
   private Address address;

   @Autowired
   public Student(Address address) {   // injected via constructor
      this.address = address;
   }

   public void show() {
      System.out.println("Student has an address: " + address.getCity());
   }
}

@Component
class Address {
   private String city = "Chennai";
   public String getCity() { return city; }
}
```

Here, Spring creates Student and Address automatically and injects Address into Student.

Setter Injection Example

```
@Component
class Student {
   private Address address;

   @Autowired
   public void setAddress(Address address) {  // injected via setter
      this.address = address;
   }
}
```

IoC Containers in Spring

Container	Use Case
BeanFactory	Core Spring container. Lightweight. Creates beans only when needed (lazy).
ApplicationContext	Advanced container. Supports events, annotations (@Autowired, @Component), internationalization, web support.

Spring MVC

Spring MVC is a framework built on top of the Servlet API to build web applications and RESTAPIs.

It follows the MVC design pattern:

- Model → Business logic, data (POJOs, Service, DAO).
- View → UI layer (JSP, Thymeleaf, HTML, Angular, React).
- Controller → Handles request/response, connects Model and View.

Spring MVC Flow (Step by Step)

1. Client sends a request

- Example: User enters URL → http://localhost:8080/myapp/hello
- Or clicks a button/form submit.

2. DispatcherServlet (Front Controller)

- Every request first goes to DispatcherServlet.
- In Spring Boot, it is auto-configured (no web.xml needed).
- Think of it as a traffic police → decides where the request should go.

3. HandlerMapping

- DispatcherServlet asks: "Which controller method should handle this URL?"
- HandlerMapping looks at annotations like:
  - @RequestMapping("/hello")
  - @GetMapping("/login")
- Finds the right controller method.

4. Controller Executes

- Controller method is executed.
- It may:
  - Call Service layer for business logic.
  - Call DAO/Repository layer to fetch data from DB.

Example:

```
@Controller
public class HelloController {

   @GetMapping("/hello")
   public String sayHello(Model model) {
      model.addAttribute("message", "Hello, Spring MVC!");
      return "welcome";   // logical view name
   }
}
```

Here:

- Model stores data (message).
- "welcome" is a logical view name (not actual JSP yet).

5. Model + View Name Returned

- Controller gives:
  - Model → Data (like message = Hello, Spring MVC!)
  - View name → "welcome"

6. ViewResolver

- DispatcherServlet asks ViewResolver to map "welcome" to an actual file.
- Example:
  "welcome" → /WEB-INF/views/welcome.jsp

In Thymeleaf, "welcome" → templates/welcome.html

7. View is Rendered

- The model data is merged with the view.
- Example welcome.jsp:
  <h1>${message}</h1>
- Output:
  <h1>Hello, Spring MVC!</h1>

8. Response sent back to Client

- Final HTML page is sent to the browser.

Real-Life Analogy (Login Example)

1. User submits Login Form (/login).
2. DispatcherServlet intercepts request.
3. HandlerMapping finds → LoginController.login().
4. Controller calls Service → DAO → Database.
5. Controller returns Model (user info) + "home" (view name).
6. ViewResolver maps → /WEB-INF/views/home.jsp.
7. JSP renders Welcome, John!.
8. Browser shows Home Page.

SpringORM

In Core Java JDBC, the developer had to do everything manually:

1. Jar files → Add JDBC driver jar manually into classpath.
2. Properties file → Manually configure DB connection (URL, username, password).
3. Model/Bean class → Manually map Java properties → DB table fields.
4. Database & Queries → Developer must know DB-specific queries (MySQL, Oracle, MongoDB).
5. PreparedStatement / Statement / CallableStatement → Write queries for CRUD manually.

Hibernate (ORM Framework)

Hibernate solves these problems by mapping Java objects directly to DB tables.

- You work with Objects (POJOs) instead of SQL queries.
- Hibernate internally generates the SQL for you.
- Database-independent (dialects handle MySQL, Oracle, PostgreSQL, etc.).

Hibernate Flow (Step by Step)

1. **Configuration** → Hibernate reads hibernate.cfg.xml (DB URL, user, dialect, mappings).

2. **SessionFactory** → Created once, manages sessions.
3. **Session** → Created per request. Manages entity objects (cache, lifecycle).
4. **Transaction** → Begin → Perform CRUD/queries → Commit/Rollback.
5. **Hibernate Query API** → Generates SQL based on HQL/JPQL/Criteria.
6. **Dialect** → Converts HQL into DB-specific SQL.
7. **Database** → Executes final query.

Web Application

web.xml – Deployment Descriptor

- First file the **Web Container (Tomcat, Jetty, etc.)** reads when deploying your app.
- Defines configuration about:
  - Servlets & Servlet Mappings
  - JSP settings
  - Filters
  - Error handling
  - Security constraints
  - Session settings
  - Welcome pages

Example web.xml

```
<web-app>
  <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>com.example.HelloServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <error-page>
    <error-code>404</error-code>
    <location>/error404.jsp</location>
  </error-page>
</web-app>
```

Servlet Life Cycle

Managed by Web Container:

1. **init()** → called once when servlet is created.
2. **service()** → handles requests (doGet, doPost, doPut, doDelete).
3. **destroy()** → called once when servlet is destroyed.

Example:

```
public class AddToCartServlet extends HttpServlet {
   public void init() {
      System.out.println("Servlet Initialized");
   }
   public void service(HttpServletRequest req, HttpServletResponse res) {
      System.out.println("Service method called");
   }
   public void destroy() {
      System.out.println("Servlet Destroyed");
   }
}
```

Request & Response

- **HttpServletRequest** → carries data from client (browser → server).
- **HttpServletResponse** → sends data from server → client.

RequestDispatcher

Used to forward/include another resource (Servlet/JSP).

- forward() → transfers control to another resource.
- include() → includes output of another resource.

Example:

```
RequestDispatcher rd = request.getRequestDispatcher("welcome.jsp");
rd.forward(request, response);   // Forward request
```

Session Management

Session = time between login & logout of a user.

Server Side

- **HttpSession**

```
HttpSession session = request.getSession();
```

```
session.setAttribute("username", "Rashik");
String user = (String) session.getAttribute("username");
```

Client Side

1. Cookies (created by server)
2. URL Rewriting (index.jsp;jsessionid=12345)
3. Hidden Form Fields

ServletContext vs ServletConfig

ServletContext

- Application-wide scope.
- Defined in web.xml or listener.

Example:

```
ServletContext context = getServletContext();
String shop = context.getInitParameter("storeName");
```

ServletConfig

- Servlet-specific configuration.

Example:

```
@WebServlet(
  urlPatterns="/WelcomeServlet",
  initParams = @WebInitParam(name="welcomeMessage",
value="Welcome!")
)
public class WelcomeServlet extends HttpServlet {
   public void init(ServletConfig config) {
     String msg = config.getInitParameter("welcomeMessage");
     System.out.println(msg);
   }
}
```

Listener

Special class that listens to events (startup, shutdown).

Example:

```
@WebListener
public class MyAppListener implements ServletContextListener {
```

```
  public void contextInitialized(ServletContextEvent sce) {
    ServletContext ctx = sce.getServletContext();
    ctx.setAttribute("discountRate", 0.10);
  }
  public void contextDestroyed(ServletContextEvent sce) {
    System.out.println("Application stopped");
  }
}
```

Filters

Used to modify requests/responses (like authentication, logging).

Example:

```
@WebFilter("/secure/*")
public class AuthFilter implements Filter {
  public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
  throws IOException, ServletException {
    System.out.println("Filter applied");
    chain.doFilter(req, res);  // continue request
  }
}
```

JSP (Java Server Pages)

JSP is used for **view/UI** part.

JSP Tags:

1. **Scriplet** <% code %>
   <% int x=5; %>

2. **Expression** <%= variable %>
   <%= "Hello " + x %>

3. **Declaration** <%! declaration %>
   <%! int counter=0; %>

Scopes in JSP

1. **Application** → available across application (ServletContext).
2. **Session** → available per user session.
3. **Request** → valid for one request.
4. **Page** → valid only within the JSP page.

Error Handling

Handled using web.xml or JSP errorPage attribute.

Example:

```
<error-page>
   <error-code>500</error-code>
   <location>/error500.jsp</location>
</error-page>
```

Security Config

In web.xml, we can restrict access to resources.

Example:

```
<security-constraint>
  <web-resource-collection>
     <web-resource-name>Admin Area</web-resource-name>
     <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
     <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
```

Spring Boot

Spring Boot is a module on top of Spring Framework that helps developers build standalone, production-ready applications with minimum configuration.

- It removes the need for boilerplate XML configuration.
- Provides auto-configuration, embedded servers, starter dependencies.
- Developer can focus more on business logic instead of plumbing code.

Advantages of Spring Boot

1. Auto-Configuration

- Spring Boot automatically configures beans based on classpath dependencies.
- Example: If spring-boot-starter-data-jpa is in the project, it automatically sets up Hibernate, EntityManager, DataSource.

2. Starter Dependencies

- Instead of adding 10 different dependencies manually, you just add a single starter.
- Example:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

This single dependency pulls Spring MVC, Jackson (JSON), Embedded Tomcat automatically.

3. Embedded Servers

- No need to install Tomcat/Jetty separately.
- Spring Boot apps come with embedded server.
- Run with just:

  mvn spring-boot:run

  or run the main() method → app starts at http://localhost:8080.

4. Dependencies Managed by Spring Boot

- No need to search and add versions from Maven Central manually.
- Spring Boot has a parent POM that manages versions.

5. Centralized Configuration

- All app settings (port, DB config, logging) go into:
  - application.properties OR
  - application.yml

Example:

```
server.port=9090
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=root
```

Spring Boot Security

Spring Security is a separate project under the Spring ecosystem (like Spring Boot, Spring Data, etc.), designed to provide authentication, authorization, and protection against common attacks (CSRF, XSS, Session fixation, etc.).
It is highly customizable and can secure:

- Web applications (with login forms, roles, session management)
- REST APIs (via tokens/JWT/OAuth2)
- Method-level access (using @PreAuthorize, @Secured)

Why use Spring Security?

Before Spring Security, developers had to manually implement security like:

- Handling login/logout with sessions
- Checking user roles/permissions
- Preventing SQL injection, CSRF, etc.

Spring Security solves these by providing prebuilt configurations while still allowing customization.

Core Concepts in Spring Security

1. **Authentication**
    - Who are you? (identity check)
    - User provides username/password (or token/JWT) → verified against in-memory DB, JDBC, LDAP, or OAuth provider.
    - Example: logging into Gmail with email + password.
2. **Authorization**
    - What are you allowed to do?
    - After authentication, roles/permissions decide what resources the user can access.
    - Example: An admin can create/delete users, but a guest may only view data.

Default Behavior in Spring Boot with Spring Security

When you add Spring Security dependency in pom.xml, Spring Boot automatically:

1. Protects all endpoints → requires authentication.
2. Creates a default login page at http://localhost:8080/login.
3. Provides a default user with username = user and password = randomly generated in console logs.

SpringBoot Inheritance

Relationships Annotations

1. @OneToOne

- One entity is associated with exactly one other entity.
- Example: A Passport belongs to exactly one Person, and a Person has only one Passport.

2. @ManyToOne

- Many entities relate to one entity.
- Example: Many Employees can belong to One Department.

3. @OneToMany

- Reverse of @ManyToOne.
- Example: One Department has many Employees.

4. @ManyToMany

- Many entities relate to many entities.
- Example: A Student can enroll in many Courses, and a Course can have many Students.
- Needs a Join Table.

@Embeddable and @Embedded

Used when an object doesn't exist by itself but is part of another entity.
Example: Address embedded in Student.

```
@Embeddable
public class Address {
   private String street;
   private String city;
   private String zip;
}

@Entity
public class Student {
   @Id @GeneratedValue
   Long id;
   String name;

   @Embedded
   private Address address;
}
```

Output

id | name | street | city | zip
12 | John | MG Rd | Pune | 411045

Inheritance in JPA

1. Single_Table

- All parent + child entities in one table.
- Discriminator column differentiates child type.
- Pros: Simple, fast queries.
- Cons: Many nulls.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="dtype")
public abstract class Person {
   @Id @GeneratedValue
   Long id;
   String name;
}
@Entity
@DiscriminatorValue("STUDENT")
public class Student extends Person {
   String course;
}
@Entity
@DiscriminatorValue("TEACHER")
public class Teacher extends Person {
   String subject;
}
```

Table

```
id | name | course | subject | dtype
1  | Niti | BTech  | null    | STUDENT
2  | Jiya | null   | AI      | TEACHER
```

2. Joined

- Each entity → its own table.
- Common fields → Parent Table.
- Child tables store only specific fields.
- **Pros**: No nulls.
- **Cons**: Joins required.

Example:

Person: id | name
Student: id | course
Teacher: id | subject

3. Table_Per_Class

- No parent table. Each child has its **own table with all fields**.
- **Pros**: No joins.
- **Cons**: Harder queries (no single parent table).

Example:

Student: id | name | course
Teacher: id | name | subject

@EntityGraph

By default, JPA loads associations **lazily** (not fetched until accessed).

Problem → **LazyInitializationException** if accessed outside transaction.

@EntityGraph lets you **fetch associations eagerly per query**.

```
@Entity
public class Department {
   @Id @GeneratedValue
   Long id;
   String name;

   @OneToMany(mappedBy = "department")
   List<Employee> employees;
}

public interface DepartmentRepo extends JpaRepository<Department, Long> {
   @EntityGraph(attributePaths = {"employees"})
   Department findByName(String name);
}
```

Now, when you fetch Department → it will also fetch Employees in the **same query** (avoiding LazyInitializationException).

Polymorphic Behavior

If you query the **Parent class**, Hibernate automatically fetches all subclasses.

List<Person> people = entityManager.createQuery("from Person", Person.class).getResultList();

This returns both Student and Teacher objects.

Traditional and Reactive Programming in Spring

1. Traditional Programming (Servlet-based Spring MVC)

- Architecture: Built on Servlet API and typically runs on Tomcat.
- Thread-per-request model:
  - Each incoming HTTP request blocks a dedicated server thread until processing completes.
  - Example: If a database query takes 5 seconds, the thread is blocked for 5 seconds.
- Limitations:
  - Not scalable for high concurrency (e.g., thousands of users).
  - Wastes server resources during blocking operations (I/O, DB calls).

2. Reactive Programming (Non-blocking)

- Concept: Code reacts to events/data asynchronously using event-loop model.
- Benefit: Handles thousands of requests concurrently with fewer threads.
- Example (Netflix): Millions of users stream simultaneously → handled efficiently via event-driven streams.

3. Reactive Streams Core Concepts

- Publisher → Produces data
  - Mono<T> → Emits 0 or 1 item.
  - Flux<T> → Emits 0…N items.
  - Example:
  - Mono.just("Happy Teachers Day");
  - Flux.just("Priya", "Sakshi", "Shubham");
- Subscriber → Consumes data from Publisher.
- Backpressure → Subscriber controls rate of data consumption (avoids overload).
- Operators → Transform/react to data streams (map, filter, flatMap, etc.).

4. Spring WebFlux

- Introduced in Spring 5 as an alternative to Spring MVC.
- Runs on Netty (non-blocking server) by default, but can also run on Tomcat, Jetty.
- Supports: Reactive Streams API (Publisher, Subscriber).
- Data types: Works with Mono and Flux.
- Ideal use cases:
  - High concurrency applications.
  - Real-time streaming apps (chat, live dashboards, video streaming).
  - APIs with very high throughput.

5. WebFlux Components

1. **Dependencies:**

   spring-boot-starter-webflux

2. **Reactive Controller:**

   - Uses @RestController, @RequestMapping.
   - Returns Mono<T> or Flux<T>.

3. **Router Functions (alternative to controllers):**

   - Functional style with RouterFunction and HandlerFunction.

4. **Error Handling:**

   - Similar to Spring MVC but with ResponseStatusException, onErrorResume.

5. **Testing:**

   - WebTestClient instead of MockMvc.

6. **Reactive Data Repositories:**

   - Uses R2DBC (Reactive Relational Database Connectivity) for SQL databases.
   - Native support for reactive NoSQL (MongoDB, Cassandra).

7. **Operators:**

   - map, filter, subscribe, flatMap, collectList.

6. Reactive Microservices

- Combine Spring WebFlux + Spring Cloud Gateway + Eureka.
- Services communicate using WebClient (non-blocking) instead of RestTemplate.
- Fully reactive stack → ensures end-to-end non-blocking flow.

SQL

```
/* ============================================================
   SQL - CATEGORIES OF COMMANDS
   ============================================================
   1) DDL (Data Definition Language) → Defines structure
     - CREATE, ALTER, DROP, TRUNCATE
   2) DML (Data Manipulation Language) → Modifies data
     - INSERT, UPDATE, DELETE
   3) DQL (Data Query Language) → Retrieves data
     - SELECT
   4) DCL (Data Control Language) → Permissions
     - GRANT, REVOKE
   5) TCL (Transaction Control Language) → Transaction mgmt
     - COMMIT, ROLLBACK, SAVEPOINT
============================================================ */

/* ============================
   DATABASE CREATION
   ============================ */
create database wiprotraining;
use wiprotraining;

show tables; -- Show all tables in current DB

/* ============================
   DBMS vs RDBMS
   ============================
   - DBMS → File-based, no relations (MS Access, XML, Excel).
   - RDBMS → Relational, uses tables with PK/FK, faster due to indexing.
     Examples: MySQL, Oracle, PostgreSQL, SQL Server.
============================ */

/* ============================
   TABLE CREATION
   ============================ */
drop table if exists todotask;

create table todotask (
  id int primary key,
  title varchar(255),
  start_date date,
  due_date date
);

-- View structure
describe todotask;
```

-- Insert sample data

```
insert into todotask values(101,'Learn Java','2025-08-05','2025-09-20');
insert into todotask values(102,'Coding Challenges','2025-08-05','2025-09-20');

select * from todotask;

/* ============================
   CHILD TABLE with FK + CASCADE
   ============================ */

create table checklist (
  id int,
  task_id int,
  title varchar(255) not null,
  is_completed boolean not null default false,
  primary key(id, task_id),  -- Composite Key
  foreign key(task_id) references todotask(id) on delete cascade
);
```

-- If a task is deleted from todotask → related checklist records also deleted.

```
describe checklist;
```

```
/* ============================
   DROP, TRUNCATE, DELETE
   ============================
   - DROP → Removes structure + data.
   - TRUNCATE → Removes all data but keeps structure.
   - DELETE → Removes specific rows with WHERE condition.
============================ */

truncate table states;
drop table if exists states, cities;

/* ============================
   STATES & CITIES Example
   ============================ */

create table states (
  statecode char(2) primary key,
  name varchar(30)
);

insert into states values("IN", "Delhi");
```

```
create table cities (
  name varchar(30),
  state_co char(2),
  foreign key(state_co) references states(statecode) on delete cascade
);

insert into cities values("Agra", "IN");

select * from states;
select * from cities;

/* ==========================
   ALTER COMMANDS
   ========================== */

create table contacts (
  id int auto_increment unique key,
  name varchar(40) not null,
  email varchar(50) not null
);

-- Add new column
alter table contacts add officialaddress varchar(50);

-- Modify column size
alter table contacts modify officialaddress varchar(100);

-- Rename column
alter table contacts change officialaddress permanentaddress varchar(100);

-- Add default constraint
alter table contacts modify email varchar(55) default 'xyz@gmail.com';

/* ==========================
   CONSTRAINTS
   ==========================
   - NOT NULL → No blank values.
   - PRIMARY KEY (PK) → Unique + Not Null.
   - FOREIGN KEY (FK) → Reference to another table.
   - UNIQUE (UK) → Allows NULLs but unique values.
   - DEFAULT → Assigns default values.
   - CHECK → Validation rule.
========================== */

insert into contacts (name, permanentaddress) values("Niti","Delhi");

/* Add CHECK constraint */
```

```
alter table contacts add constraint ch check(permanentaddress != officialaddress);

/* ============================
   DATA TYPES
   ============================
   int, boolean, decimal, datetime, timestamp, date, char, varchar, text,
   enum(), blob (binary large object → images, files)
============================ */

/* Example: Storing image in BLOB */

create table image (
  id int primary key auto_increment,
  title varchar(255) not null,
  image_data longblob not null
);

-- Load file into blob

insert into image(title, image_data)
values("myphoto", load_file('C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/image.png'));

/* ============================
   TEMPORARY TABLES
   ============================ */

create temporary table tmp2
select * from orders where status='Shipped';

/* ============================
   AGGREGATE FUNCTIONS
   ============================
   - COUNT(), SUM(), AVG(), MIN(), MAX()
   - GROUP BY → Groups rows.
   - HAVING → Filters groups.
   - DISTINCT → Unique values.
============================ */

select distinct status from orders;
select avg(buyprice) from products where productLine='Motorcycles';
select productline, max(buyprice) from products group by productline;

/* ============================
   ORDER BY & LIMIT
   ============================ */
```

```
select customerNumber, amount from payments order by amount limit 5;

/* ============================
   DATE FUNCTIONS
   ============================ */

select datediff('2025-08-06','1975-10-12') as age;
select now();

/* ============================
   JOINS
   ============================
   - INNER JOIN → Common records.
   - LEFT JOIN → All from left + matching right.
   - RIGHT JOIN → All from right + matching left.
============================ */


select pl.productline, p.productVendor
from productlines pl
left join products p using(productline);

/* ============================
   VIEW
   ============================ */

create view vn as
select distinct c.customernumber, o.ordernumber
from customers c left join orders o using(customernumber);

select * from vn where ordernumber is not null;

/* ============================
   SUBQUERIES
   ============================
   - Simple Subquery → Independent.
   - Correlated Subquery → Depends on outer query.
============================ */


/* Employees earning more than dept avg salary */

select employee_id, employee_name
from employee e
where salary > (
  select avg(salary) from employee e2
```

```
  where e2.department_id = e.department_id
);

/* ============================
   STORED PROCEDURES
   ============================ */

create database student_database;
use student_database;

create table student (
  id int auto_increment primary key,
  name varchar(100),
  age int
);
```

-- Procedure Example

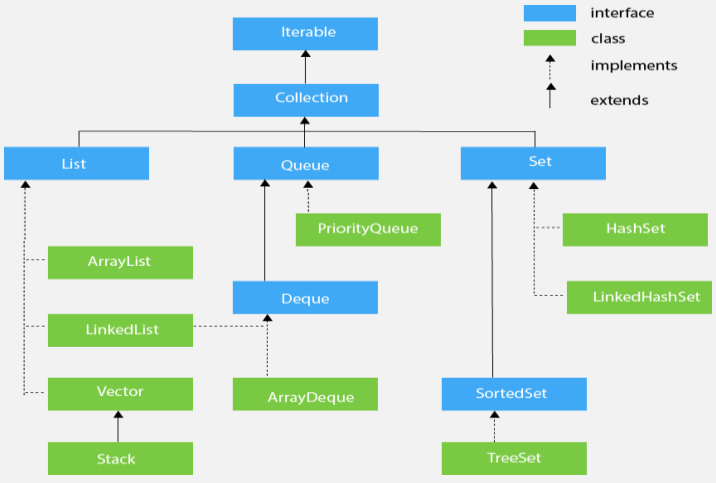
```
DELIMITER $$
CREATE PROCEDURE insert_user(IN sname varchar(20), IN sage int)
BEGIN
  insert into student(name, age) values(sname, sage);
  select * from student;
END $$
DELIMITER ;
```

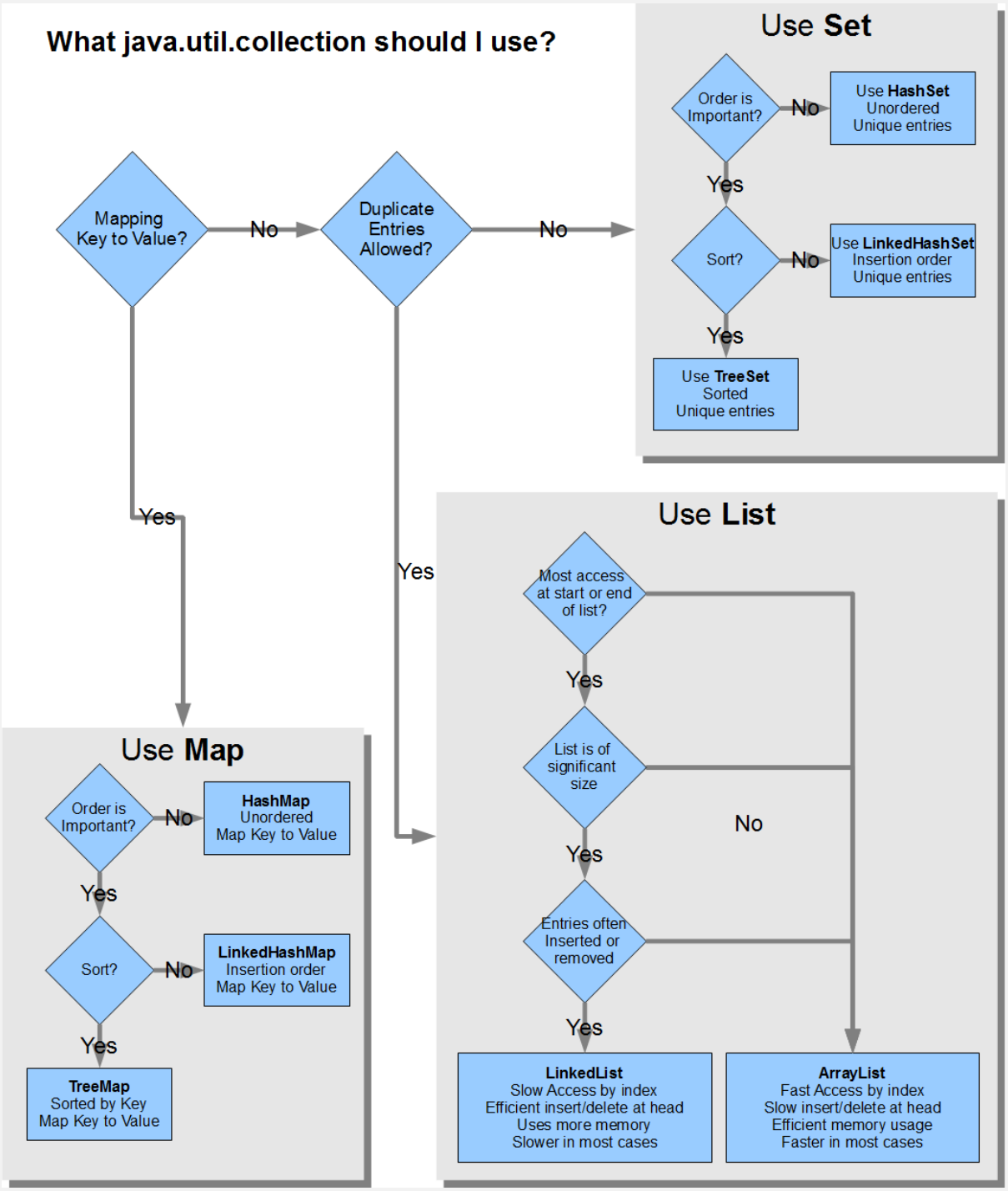
-- Call procedure

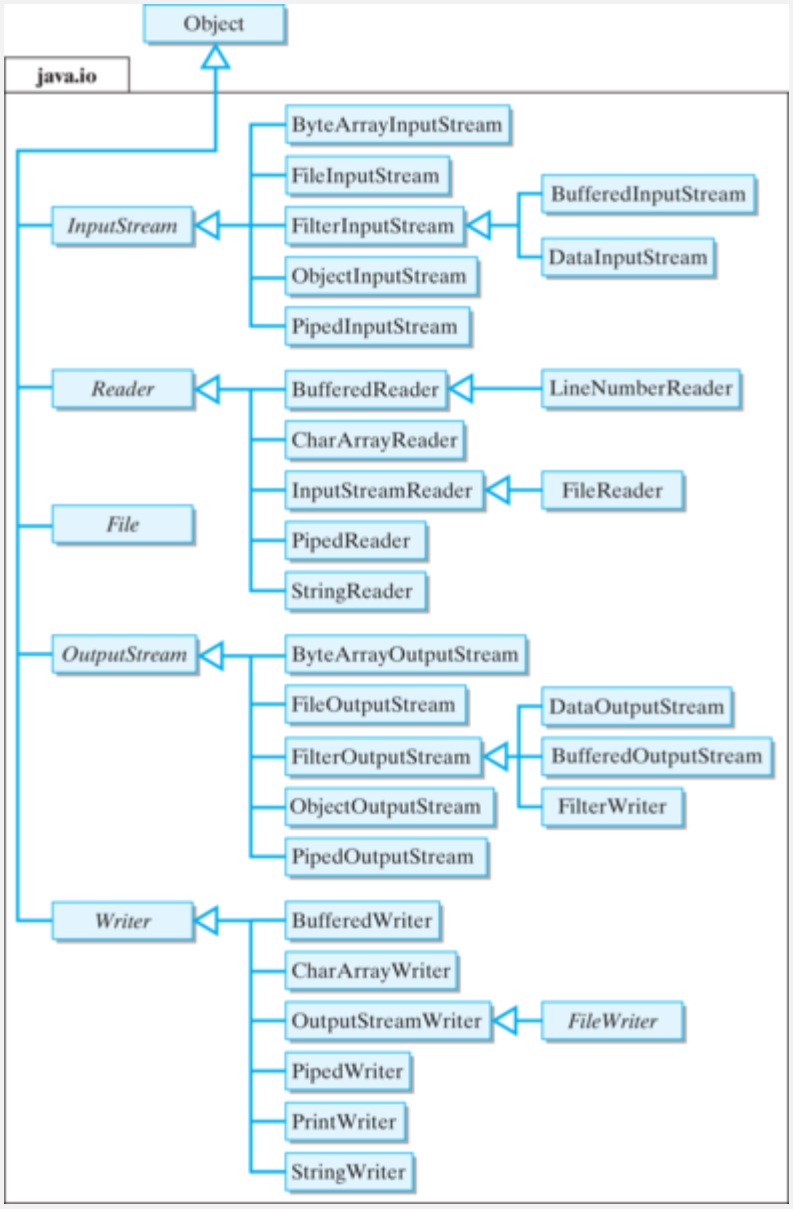
```
call insert_user("Jiya",45);
```

Images/Flowcharts

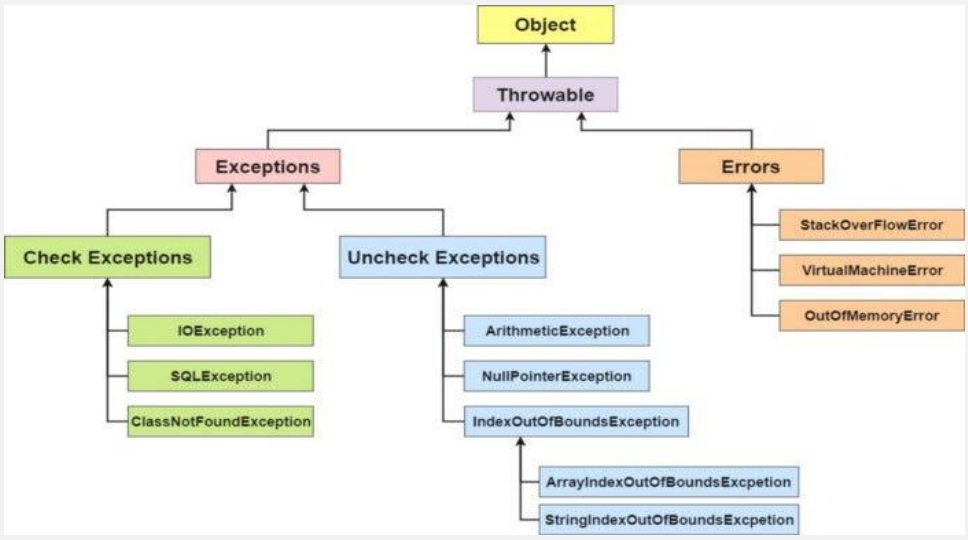
Collection

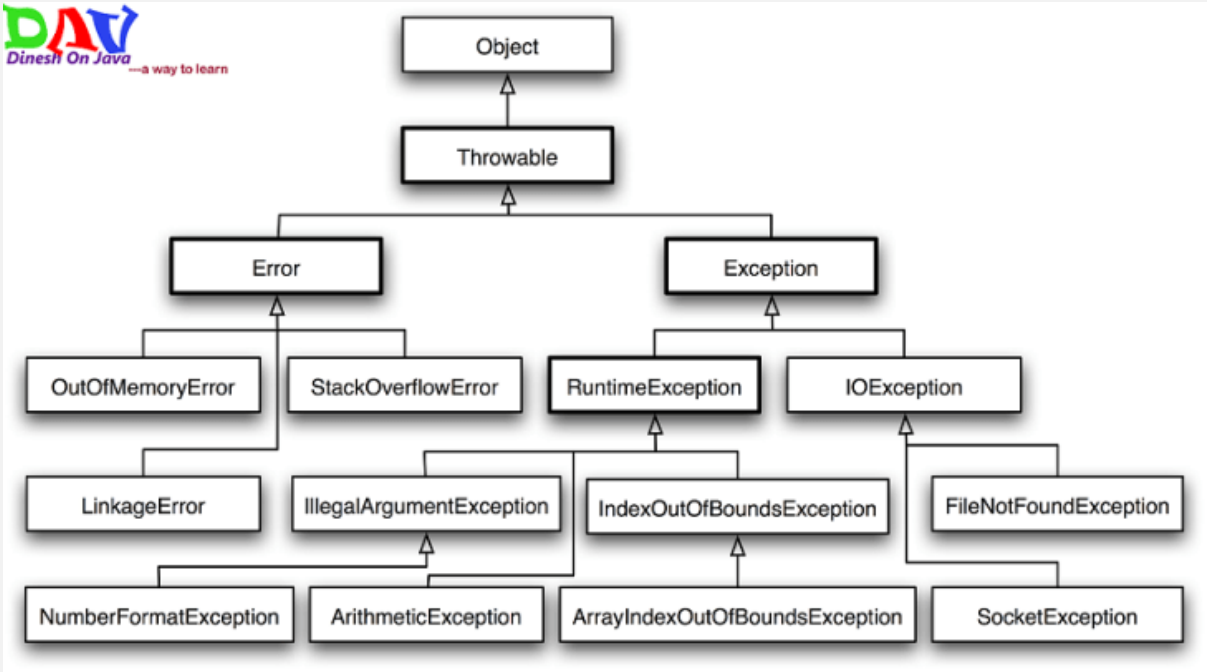


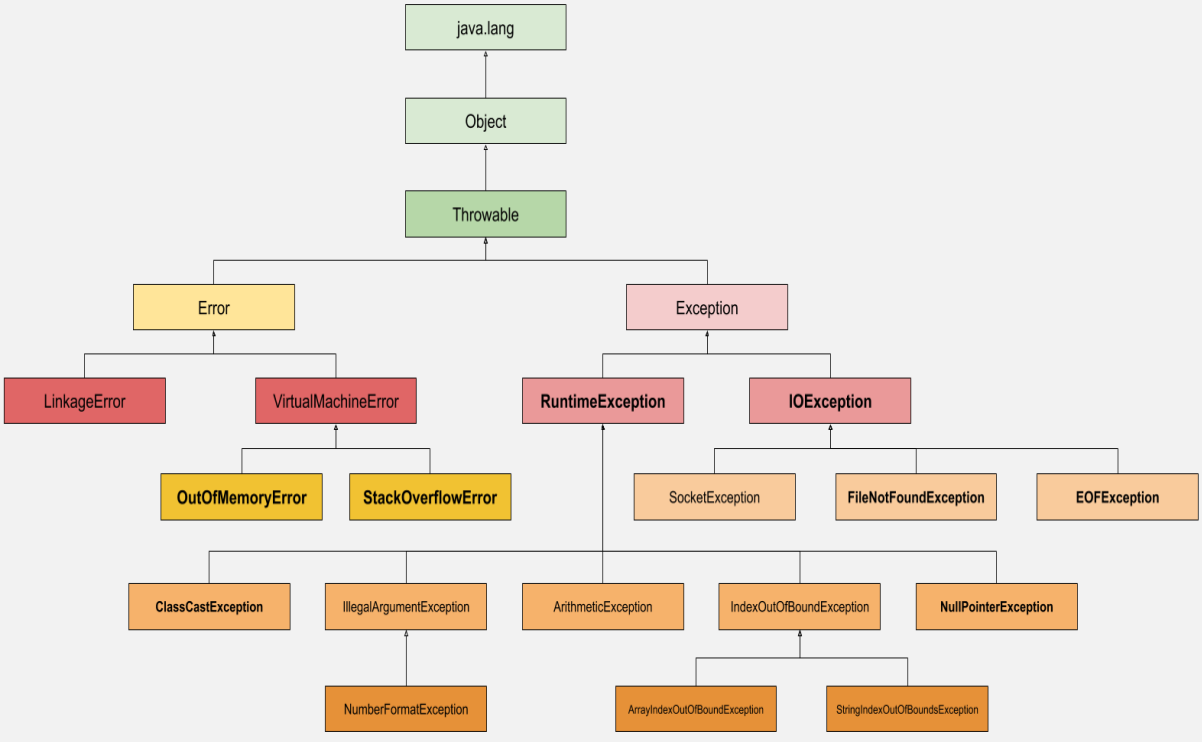

What java.util.collection should I use?
Mapping Key to Value?
No
Duplicate Entries Allowed?
No
Yes
Yes
Use Set
Order is Important?
No
Use HashSet
Unordered
Unique entries
Yes
Sort?
No
Use LinkedHashSet
Insertion order
Unique entries
Yes
Use TreeSet
Sorted
Unique entries
Use Map
Order is Important?
No
HashMap
Unordered
Map Key to Value
Yes
Sort?
No
LinkedHashMap
Insertion order
Map Key to Value
Yes
TreeMap
Sorted by Key
Map Key to Value
Use List
Most access at start or end of list?
Yes
List is of significant size
No
Yes
Entries often Inserted or removed
Yes
LinkedList
Slow Access by index
Efficient insert/delete at head
Uses more memory
Slower in most cases
ArrayList
Fast Access by index
Slow insert/delete at head
Efficient memory usage
Faster in most cases

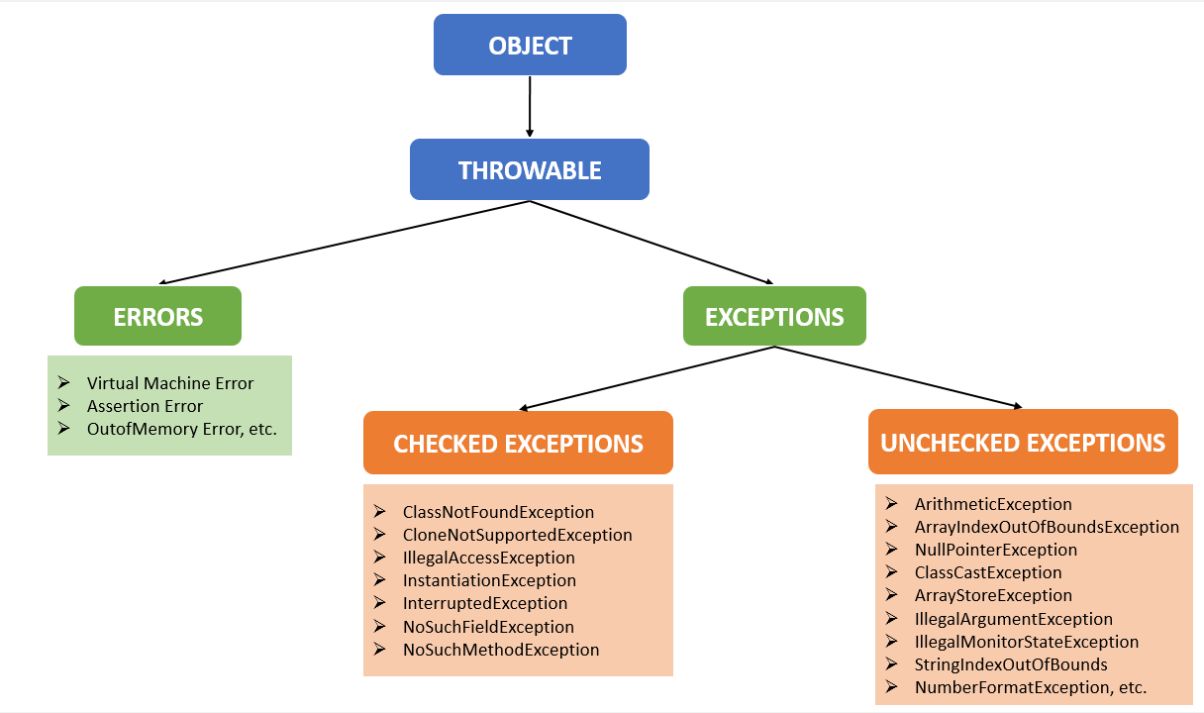

Object
java.io
ByteArrayInputStream
FileInputStream
BufferedInputStream
InputStream
FilterInputStream
DataInputStream
ObjectInputStream
PipedInputStream
Reader
BufferedReader
LineNumberReader
CharArrayReader
InputStreamReader
FileReader
File
PipedReader
StringReader
OutputStream
ByteArrayOutputStream
FileOutputStream
DataOutputStream
FilterOutputStream
BufferedOutputStream
ObjectOutputStream
FilterWriter
PipedOutputStream
Writer
BufferedWriter
CharArrayWriter
OutputStreamWriter
FileWriter
PipedWriter
PrintWriter
StringWriter

Exceptions


Object
Throwable
Exceptions
Errors
Check Exceptions
Uncheck Exceptions
StackOverFlowError
VirtualMachineError
OutOfMemoryError
IOException
SQLException
ClassNotFoundException
ArithmeticException
NullPointerException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsExcpetion
StringIndexOutOfBoundsExcpetion


Dinesh On Java
---a way to learn
Object
Throwable
Error
Exception
OutOfMemoryError
StackOverflowError
RuntimeException
IOException
LinkageError
IllegalArgumentException
IndexOutOfBoundsException
FileNotFoundException
NumberFormatException
ArithmeticException
ArrayIndexOutOfBoundsException
SocketException


java.lang
Object
Throwable
Error
Exception
LinkageError
VirtualMachineError
RuntimeException
IOException
OutOfMemoryError
StackOverflowError
SocketException
FileNotFoundException
EOFException
ClassCastException
IllegalArgumentException
ArithmeticException
IndexOutOfBoundException
NullPointerException
NumberFormatException
ArrayIndexOutOfBoundException
StringIndexOutOfBoundsException



JDBC

JDBC is an API in Java used to connect and interact with databases. It allows Java programs (backend) to talk with databases (like MySQL, Oracle, PostgreSQL) using drivers.

Think of JDBC as a bridge between Java code and the database.

Why Do We Need Drivers?

Databases don't understand Java directly. So we need drivers (small pieces of software) that translate Java instructions into database-specific language.

Types of JDBC Drivers

1. **Type 1 (JDBC-ODBC bridge / DSN):**
    - Requires creating a DSN (Data Source Name) in each PC.
    - Not safe and not practical → obsolete.
2. **Type 2 (Native API):**
    - Uses native libraries of the database.
    - Needs installation on each client machine.
    - Heavy dependency problem.

3. **Type 3 (Network Protocol):**

    - Drivers on the server/network.
    - Platform independent but rarely used now.

4. **Type 4 (Thin driver):**

    - Pure Java driver → most common today.
    - Just need to add a .jar file (e.g., mysql-connector-java.jar).
    - No native libraries or DSN required.

Steps to Connect Java with MySQL using JDBC

1. Add the driver library

- Download mysql-connector-java-x.x.x.jar
- Add to your project:
  Right click project → Build Path → Configure Build Path → Add External JARs

2. Create a db.properties file inside resources/

This file contains the database credentials.
db.url=jdbc:mysql://localhost:3306/student_database
db.username=root
db.password=root
This way we avoid hardcoding sensitive information in Java files.

3. Utility to Read Properties → DBPropertyUtil.java

```
package com.wipro.util;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public class DBPropertyUtil {

   public static String getPropertyString(String filename) {
      Properties props = new Properties();

      // try-with-resources ensures the file stream closes automatically
      try (FileInputStream fis = new FileInputStream(filename)) {
         props.load(fis);
      } catch (IOException e) {
         System.err.println("Failed to connect: " + e.getMessage());
```

```
        }

        String url = props.getProperty("db.url");
        String user = props.getProperty("db.username");
        String password = props.getProperty("db.password");

        System.out.println("Loaded from file: " + url + " | " + user);
        return url + "|" + user + "|" + password;
    }
}
```

Purpose: To fetch credentials from the .properties file.

4. Utility to Establish Connection → DBConnUtil.java

```
package com.wipro.util;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBConnUtil {

    private static Connection conn;  // Singleton Connection

    public static Connection getConn() {
        if (conn == null) {
            try {
                String connectionString =
DBPropertyUtil.getPropertyString("resources/db.properties");
                String[] connectionCredentials = connectionString.split("\\|");
                String url = connectionCredentials[0];
                String user = connectionCredentials[1];
                String password = connectionCredentials[2];

                conn = DriverManager.getConnection(url, user, password);
                System.out.println(" SQL Connected Successfully");
            } catch (SQLException e) {
                System.err.println(" Failed to establish connection: " + e.getMessage());
            }
        }
        return conn;
    }
}
```

Purpose: To connect only once and reuse the connection (Singleton design pattern).

5. Project Architecture

- **entity/** → Model class (e.g., Student.java)
- **dao/** → Interface for CRUD methods
- **service/** → Implementation of business logic
- **util/** → Utility classes for DB connection & properties
- **main/** → StudentManagement.java (entry point)

Flow of Execution

1. db.properties → Holds DB credentials.
2. DBPropertyUtil → Reads credentials from file.
3. DBConnUtil → Creates/reuses DB connection.
4. DAO → Defines operations like insertStudent(), getAllStudents().
5. Service → Implements logic using DAO.
6. Main → Runs the program.