Java Short Notes

class - collect^{n} of obj., memory ✗
obj - real world entity, memory ✓

* Java
  ↳ high-level, object oriented programming lang. designed to be portable, secure & easy to use.
  - develop → Sun Micro system → 1995

characteristics.
- Simple & easy to learn
- object-oriented.
- Platform-independent
- Secure
- Robust - less chances to crash - why? • strong memory mgt. • No pointer • exception handling
- Multithreaded.

Flavours of Java
- Java SE (desktop/server app)
- Java EE (web based app)
- Java ME (mobile phones)

Variable → container, store value.
int a = 10;

* Java vs C++

Java	C++
• Platform-independent	• platform-dependent.
• No pointers	• Support pointers.
• Not supported multiple inheritance	• Supported directly.
• highly secure	• less secure.

Data Type
↓
type of data variable can hold
- Primitive (built-in)
- Non primitive (Referenced)

* Primitive (8)
  ↳ byte, short, int, long, float, double, char, boolean
* Non primitive
  ↳ String, Arrays, Classes, interfaces

* operators → perform specific op^{n}.
  - Arithmetic → (+, -, %, /, *)
  - Assignment → (=, +=, -=, *=, /=, %=)
  - Relational → (==, !=, >, <, >=, <=)
  - logical → (&&, !)
  - Unary → (+, -, ++, --, !)
  - Bitwise → (&, |, ^, ~, <<, >>)

Eg 5 ways to create obj:-
1. New keyword
2. new Instance() method
3. clone()
4. deserializat^{n}
5. factory method

* JVM, JDK, JRE
- JVM → Java Virtual Machine → executes java bytecode
- JDK → Java Development Kit → JRE + dev. tools
  ↳ req. for developing app.
- JRE → JVM + libraries + other runtime tools.
  ↳ provides everything needed to run java programs.

* Types of variable
  - local → inside method
  - Instance → inside class but outside method, belong - obj.
  - static → Belongs to class.

* Control Statements.
  ↳ controls - flow of execut^{n}.
  1. Decision Making → (if, else, else if, switch)
  2. looping st. → (for, while, do while)
  3. Jump st. → (break, continue, return)

* Classes & objects.

- Class → blueprint for creating Obj.
- Obj → real instance of class | physical entity.
- Field → var inside class.
- Method → Funct° → inside class
- Instantiat° → creating obj using new keyword.

* Constructor

↳ block of code used to initialize obj.
- name as same as class name
- does not have return type.

Syntax:-

```
class One {
    One() {
    }
}
```

Types.
- Default → no param.
- Parameterized → with param.

* Method vs Constructor

Method	Constructor
• any name	• Must match class name
• has return type	• does not have return type
• called manually	• called automatically.
• def. behavior	• initialize obj.

* Constructor Chaining

↳ calling one constructor from another within same class.
- helps reuse initializat° code, avoid duplicat°.

Types
- within same class → this()
- from parent class → super()

* OOP

↳ Object-oriented programming
- paradigm, organizes sw design around objects.

Goals
- Reusability
- Modularity
- Maintainability
- Extensibility.

Pillars (4)
- Inheritance
- Polymorphism
- Encapsulat°
- Abstract°

Constructor overloading
↓
multiple constructors in one class with diff. param. list.

Copy Constructor
↓
creates new obj. by copying fields from another existing obj.
→ Java does not provide built-in copy constructor.

Deep Copy vs Shallow Copy
- Deep Copy
  ↳ copies ref.
- Shallow Copy
  ↳ copies everything

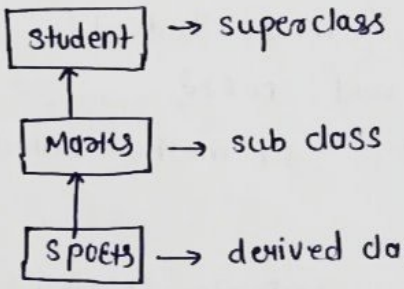
Packages = classes + interfaces

- **Inheritance** → parent-child relationship
  - ↳ class acquires properties & behaviour of another class.
  - → use "extends"

eg.
```
class Parent{
}
class child extends Parents{
}
```

Types
- → Single →
- → Multilevel
- → Hierarchical
- → hybrid
- → Multiple.

- Single → subclass derived from one super class.
- Multilevel → class inherits from class, which in turns inherit from another class

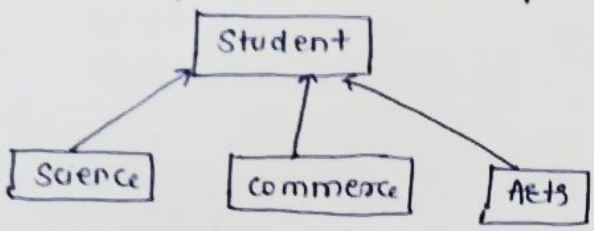


eg:-
```
class Student{
  int reg-no;
  void getNo(int no){
     reg-no = no;
  }
  void putNo(){
     SOP("Reg No:" + reg-no);
  }
}
class Marks extends Student{
     float marks;
    void getMarks(float m){
        marks = m;
     }
    void putMarks(){
     SOP("Marks" + marks);
     }
  }
class Sports extends Marks{
     float score;
    void getScore(float sc){
        score = sc;
     }
    void putScore(){
       SOP("Sore:" + score);
  }
}
```

```
public class Test{
  psvm(string args[]){
   Sports ob = new Sports();
   ob.getNo(112);
   ob.putNo();
   ob.getMarks(78);
   ob.putMarks();
   ob.getScore(68.7);
   ob.putScore();
  }
}
```

volatile
↳ tells JVM that variable may be changed by multiple threads & ensures visibility of changes to variables across threads.

- Hierarchical Inheritance
  ↳ no of classes derived from one base class.



eg:-

```
class Student{
    public void one(){
        SOP("student method called");
    }
}
class science extends Student{
    public void sci(){
        SOP("sci method called");
    }
}
class Commerce extends Student{
    public void com(){
        SOP("com method called");
    }
}
class Arts extends Student{
    public void art(){
        SOP("art method called");
    }
}
public class Test{
    psvm(String args[]){
        Science ob1 = new Science();
        Commerce ob2 = new Commerce();
        Arts ob3 = new Arts();
        ob1.sci();
        ob2.com();
        ob3.art();
    }
}
```

* Hybrid
  ↓
  Combinat^{n} of two or more types of inheritance.
  → possible only through interfaces.

* Multiple Inheritance
  ↓
  class inherits from more than one class
  → does not support in Java.
  ↓
  why?
  ↓
  - ambiguity
  - Diamond problem.

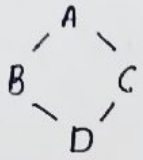
* ambiguity
  ↓
  occurs when two or more parent class have methods with same name, child class doesn't know which to use.

eg:-

```
class A{
    void show(){
        SOP("A");
    }
}
class B{
    void show(){
        SOP("B");
    }
}
```

✗ class C extends A,B{ }

• Diamond Problem.
↳ Special case of ambiguity.



• class B & C → inherits from A.
• class D → inherits from both B & C
→ To solve this, use interface.

marker interface
↳ interface with no methods / fields.

Key points:-
• main method can't get overloaded.
• static method can also get overloaded.

* Polymorphism
↓
allows method / obj. to take multiple forms.

Types
↳ Compile time → Method overloading
↳ Run time → Method Overriding.

eg:- Women

* Compile time → handle - compiler
↳ same method name but diff. param.

* Run time
↳ Two diff. classes have methods with same name & param.
- can't override private, static, final method.
- handle - by JVM.

Abstraction vs Encapsulation
↳ Focuses on what to expose, hides implementation
Encapsulation
↳ Focuses on how data is hidden inside class.

* Abstraction
↳ hiding implementation details & showing only functionality.

eg. Driving Car → you press the accelerator but you don't know engine's internal working

Mobile phone:
↳ you can tap on icons, but you can't see how OS loads to it

Abstraction can be achieved in two ways:-
1] Abstract classes
2] Interfaces.

Abstract classes	Interfaces
→ can have both abstract & concrete methods.	- only abstract method.
→ support instance var.	- only support static final variables.
→ Can have public, private, protected methods	- Methods - public by default.
→ partial abstraction	→ 100% abstraction.

Functional interface → has only one abstract method.
@FuctionalInterface.

* Encapsulation
↳ wrapping / binding of data & methods together in a single unit.
→ implementation → using private var & public getter & setters
data hiding control

eg. ATM Machine
↓
user → insert card, enter pin, withdraw cash.
Can't see → acc. details, how bal. calculated, how pin is verified.
ATM hides internal data & logic, gives access to what's necessary.

Access Modifiers
↳ accessibility / scope of any var / methods.
- Public → accessible everywhere
- Private → accessible within same class.
- Protected → outside package but only in sub class.
- Default → if no modifier specified, accessible within same package.

Keywords:-
- Static → can be used for nested classes.
↳ used → memory management.
memory ek hi baar di jati hai
- belongs to class
- class-level data.

use → variable
→ method
→ block

```
class student {
    string name;
    static string school;
}
public class OOPs {
    psvm(string args[]) {
        student.school = "SMPS";
        student ob = new student();
        ob.name = "darshana";
        sopln(ob.school);
    }
}
```
o/p: SMPS

- variable - common prop. for every obj
- method - directly access by its class name, no need to create obj.
- block - execute before main method. execute at the time of class loading.

Note:- we can have multiple static blocks.

- super
↳ access data of parent class.
- use at obj. level.
↳ access constructor, var, method of parent class

- final
→ variable
→ method
→ class

- variable - constant
- method - cannot override
- class - cannot extend.

when to use static?
↓
when memory bachani ho!

- this
↳ use → variable
→ method
→ constructor.

- variable - refer current class instance var.
- method - used to call another method of same class
- Constructor - used to call constructor within same class.
↳ use / ul → constructor chaining.

* Static vs Instance

Static	Instance
• memory allocated with class	• memory allocated separately for each obj.
• directly access by class name	• access through obj.
• common for all obj. val.	• value – different for all obj.

Exception Handling → mechanism → handle run time errors.

• Exception :- unexpected event occurs during execut° of program., disrupts normal flow.

Types
- → checked
- → Unchecked.

* can main method get overloaded? yes JVM alway calls main method which receives String as an args

• checked → • occurs at compile time.
• must handle using try, catch or throws.

eg:- • IOException
• SQLException
• FileNotFoundException
• ClassNotFoundException

Garbage collector :- delete unused data

try-with-resources.
↳ automatically close resources. – JDK 1.7.
– calls ~~to close~~)

• Runtime (Unchecked) → occurs at run time.

eg:- • NullPointerException
• ArithmeticException
• ArrayIndexOutofBoundException.

Runtime (unchecked)
↳ programmer की गलती के कारन आती है

* Error
↳ usually caused by system failures, not program.

eg:- • OutOfMemoryError – JVM runs out of memory
• StackOverflowError – due to many method execut° invocat°
• VirtualMachineError

* try, catch, finally

• try – encloses code that may throw exception
• catch – handles except° if it occurs.
• finally – always execute even if except° occurs.

* throw vs throws
- throw → only used → compile time except°
- throws → indicates caller method.
  - ↓ handle single except°
  - ↳ handles multiple except°.

Note:- we can have multiple catch blocks.
• we can use multiple except° in single catch blocks by using '|' operator since Java 7+

Multithreading.

Eg:- File downloading

↓

multiple threads runs within same process.

- Multiprocessing
  - ↳ multiple process runs independently.
- multitasking
  - ↳ execute multiple task simultaneously.

ways
- ↳ process based → each process allocates separate memory area.
- ↳ thread based → share same add. space.

- Thread class.
  - ↳ provides constructor & methods to create & perform operat^n on thread.
  - extends obj. class & implement runnable interface.

* Thread life cycle
  1. New → born
  2. Active
     - ↳ Runnable → ready to run
     - ↳ Running → currently executing
  3. Blocked / waiting → paused
  4. Terminate → finished execut^n

* Thread schedular
  - ↳ decides which thread to run & which thread to execute.

* Thread Priority.
  - ↳ provides some priorities, acc. to these priorities JVM allocates to the processor.

  Range → 1 to 10

  ↓

  Types
  - ↳ MAX_PRIORITY = 10
  - ↳ NORM_PRIORITY = 5
  - ↳ MIN_PRIORITY = 1

- If priority is set to be high, it will throw error, i.e IllegalArgumentException.
- To set thread priority, use setPriority(int priority).

methods →
- public static void setPriority()
- public final int getPriority()

Runnable vs Thread

↓

Interface, allows multiple inheritance

Thread → class, extends directly.

Runnable vs callable

- Runnable → No return, no except^n
- Callable → returns val, can throw checked except^n.

* Thread Control methods

- Sleep() – pause for specific period
- join() – wait until another thread
- yield() – stop current executing thread to execute.
- isAlive() – checks if thread is still

Synchronisation → prevent race condⁿ, ensu

↳ technique, through which we can threads, among the no. of threads one thread will enter inside the synchronised area.

eg. Ticket Booking System

purpose
↳ overcome the problem of multithreading.

Synchronized method vs block.

method: locks entire method

Block = locks only specific section.

Volatile
↳ ensures variable is always read from main memory, not CPU cache. (latest val)

Dynamic method dispatch
↓
at runtime, method call is resolved based on object, not ref. type.

* Lambda exp
↳ shortcut for writing anonymous methods / functⁿ.

Syntax:-

(param) → { code }

* Stream API (Java 8)
↳ way to process collectⁿ in functional style.

operatⁿ
↳
- filter() – filters elements
- map() – transform elements
- sorted() – sort elements
- collect() – collect results to list
- forEach() – loop through each item.

Covariant return type
↓
overridden method can return subclass type of original return type.

* File Handling
↳ provides java.io & java.nio package to read / write files.

* Serialisatⁿ
↳ converting java obj into byte stream

Deserializatⁿ → converting byte stream back to obj.

Java 8 features.

- Lambda exp
- Stream API
- FunctionalInterface
- Default & static method in Interf
- Method Reference
- Optional class
- Date & Time API

Inter thread Communicatⁿ methods:-

- wait() – Makes thread wait
- notify() → wakes up on one waiting thread
- notifyAll() → wakes up on ~~multiple~~ all waiting thread.

Strings
↳ scp. sequence of characters.

- methods
  - concat()
  - equals()
  - split()
  - length()
  - replace()
  - compareTo()
  - subString()
- Two ways to create string obj.
  1. String literal
  2. new keyword

SCP
↓
String Constant Pool
↓
memory in heap area where java stores string literals.

why Strings - immutable
↓
- security
- Thread-safety
- caching efficiency
- String pool mgmt.

* String, StringBuffer, StringBuilder

String	StringBuffer	StringBuilder
• Immutable	• mutable	• mutable
• Thread-safe	• Thread-safe	• Not thread-safe
• Slow	• Slow	• Fastest

Types of Memory in JVM
- Heap - stores obj.
- Stack - stores method calls & local variables.
- Method area - stores class metadata.
- PC Register area - stores add. of current instruct°. (add ratheta har next instruct° ka)
- Native Method Stack - for non-java code

final, finally, finalize
- final - constant (var)
- finally - block - always executes after try-catch.
- finalize() - called by garbage collector before obj. is reclaimed.

* Autoboxing → primitive → wrapper automatically.
* Unboxing → wrapper → primitive

#

```
try (BufferReader br = new
BufferReader (new FileReader
    ("data.txt"))) {
    SOP( br.readLine());
} catch (IOException e) {
    e.printStackTrace();
}
```

⇒ No need to write br.close(), because JVM calls it automatically.

Throw
→ Create except° object manually.
→ used - runtime except°
→ used - single except°
→ inside method
→ use with new keyword
→ can not write statement after throw keyword.

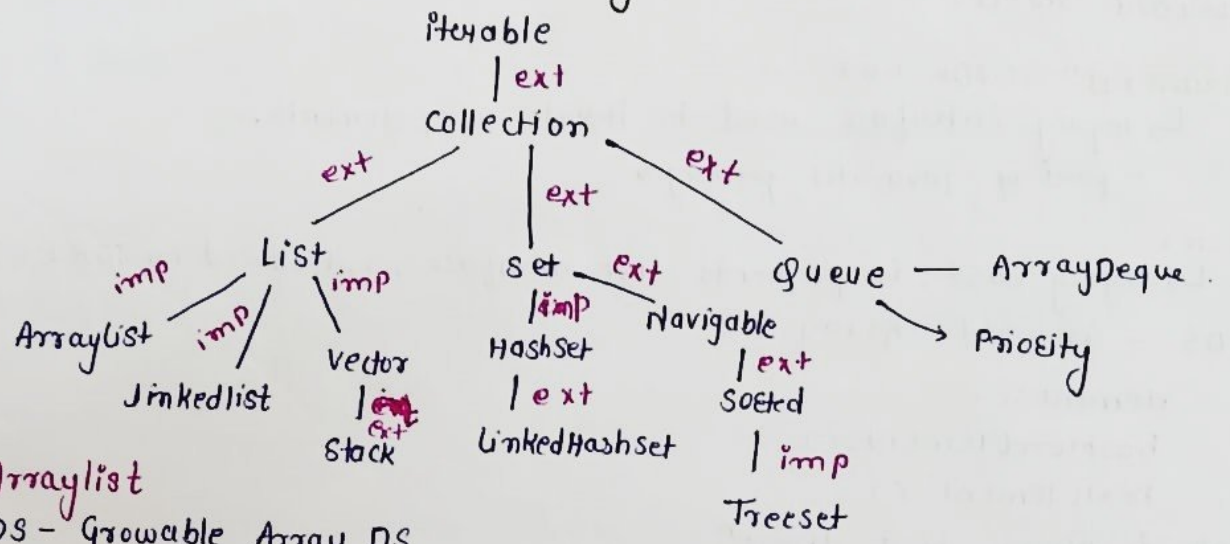
Collection Framework

↓

set of classes & interfaces, used to store, retrieve & manipulate data.

- Collection
  ↳ single entity, used to store obj. type of data.
- framework
  ↳ set of provide structured, reusable & extensible approach to handle data collection

* Collection framework Hierarchy.



* Arraylist

- DS - Growable Array DS.
- follows insertion order
- duplicacy allowed.
- store homogeneous type of data.
- store multiple null values.
- indexed based ds.

Iterable

↓

root interface in JCF, rep. collection can be iterated.

→ JDK - 1.5

* LinkedList vs Arraylist

↓	↓
• non-contiguous mem. location	• contiguous memory location
• best - insertion & deletion	• worst - insertion & deletion
• worst - searching	• best - searching
• DS - Doubly linkedlist	• DS - Growable Array DS.

Iterator

↓

interface, displays collection of obj one-by-one.

- ListIterator
- Enumeration

* Iterator vs ListIterator

Iterator	ListIterator
• displays collectn of obj one-by one	• displays only list implemented class.
• 2 methods • hasNext() • next()	: 3 methods. • hasNext() • hasPrevious() • previous()
• print elements only in forward directn	• print ele. in both directn

* Enumeratn (JDK 1.0)
  ↳ legacy interface used to iterate over elements.
  - part of java.util packages.

* Vector
  ↳ legacy class, implements list interface, introduced in JDK 1.0
  - DS - Growable Array DS.
  - elements()
    hasMoreElements()
    nextElements()
  - only displays fwd directn
  - can iterate vector through enumeratn.
  - Same like arraylist.

* Stack
  ↳ child class of vector, implemented by list interface.
  - DS - LIFO
  - indexed base
  - does not follow sorting order
  - duplicacy allowed.
  - multiple elements can be stored.

* List vs Set

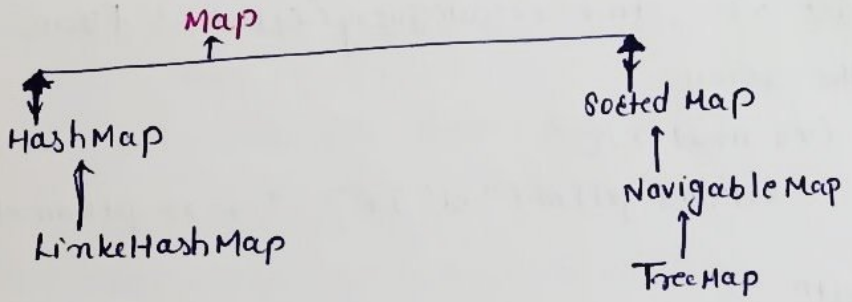
List	Set
• does not follow sorting order.	• does not follow insertn order.
• follows insertn order.	
• duplicate data - allowed	• duplicate data - not allowed.
• multiple null values - allowed	• only single null val can be stored.
• indexed based	• does not follow index based.

* Hashset
  ↳ • not indexed based ds.
  • According to hashcode val, hashset store elements.
  • does not follow insert^n order.
  • can store single null values.
  • back up by Map.

* Hashset vs LinkedHashset

HashSet	LinkedHashset
• implemented class of Set interface.	• child class of Hashset, implements Set interface.
• DS - Hash table	• DS - Hashtable + Doubly LinkedList.
• does not follow insert^n order.	• follow insert^n order.

* TreeSet
  ↓
• implemented class of navigable set, sorted set & set interface.
• DS - balanced tree
• follows sorting order.
• does not follow insert^n Order.
• Contain unique elements.
• null elements - not allowed.



• HashMap
  ↳ implemented class of Map interface.
  • DS - Hashtable
  • does not follow insert^n order
  • acc. to hashcode values, hashMap store elements.
  • duplicate keys - not allowed., but can store duplicate val.
  • Non-synchronised.
  • If we want to print key & values separately, so we use entry set.
    entrySet - entry interfaces.
      ↳ returns value inside the set
        - can retrieve data separately in HashMap.

* TreeMap
↓
implemented class of navigable Map, Sorted Map & Map interface.
- DS - Red Black Tree.
- Store homogeneous type of data.
- Data retrieve - sorting order
- null key - not allowed.
- values could be duplicate.

JDBC → Java Database Connectivity
↳ API, enables java app. to interact with DB.

Steps :-

1. Load JDBC Driver.

```
Class.forName ("com.mysql.cj.jdbc.Driver");
```

2. Establish Connection

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

3. Create Statement

```
Statement stmt = con.createStatement();
```

4. ExecuteQuery

```
ResultSet rs = stmt.executeQuery("Select * from users");
```

5. Process the result

```
while (rs.next()){
    SOP(rs.getInt("id") + "  " + rs.getName("name"));
}
```

6. Close connection

```
rs.close();
Stmt.close();
con.close();
```

- DriverManager - establish connection to db using getConnection().
- Connection - Interface, rep session blw Java app & db
- Statement - sends static SQL queries to db
- Resultset - holds data retrieved by select query.
- executeQuery() → SELECT, returns resultset.
- executeUpdate() → Insert, update, delete.

* Types of JDBC Drivers
  - Type 1 → JDBC - ODBC bridge
  - Type 2 → Native API
  - Type 3 → Native Protocol.
  - Type 4 → Thin Driver.

* SQL inject?
  - security issue where attackers injects SQL code.
  - avoid it using prepared statement.
    - execute paramaterized queries.

* Statement, prepared Statement, Callable Statement.
- Statement : for executing static SQL queries.
- Prepared statement : for dynamic SQL queries.
- callable statement : To call stored procedures from db.

Servlet → server side technology.
  - java class - used to build web app.
  - Runs on Java enabled Server like Apache Tomcat & handles HTTP request & sends responses.

* Life Cycle.
- init() - called once when servlet is created
- service() - called every time a request comes in.
- destroy() - called once when the servlet is being destroyed

* Classes
- HttpServlet - Base class for HTTP servlets.
- HttpServletRequest - Rep. HTTP req. from client.
- HttpServletResponse - Rep HTTP resp. to be sent.
- ServletConfig - for servlet configuration.
- ServletContext - for app. wide parameters.

* Methods.
- doGet() - handles HTTP GET req.
- doPost() - handles HTTP POST req.
- init() - initializes servlet.
- destroy() - called when servlet is being removed.

doget()
→ data sent in URL string
→ less secure
→ use - reading / retrieving data

query in

dopost()
→ data sent in response body
→ more secure
→ submit sensitive data

- Serialization - convert obj. to byte stream
- Deserialization → byte stream to obj.

* transient vs volatile
  - ignore var during serialization

ensures visibility of changes across threads.

JSP

↳ Stands for Java Server Pages.

- used to create dynamic HTML pages using Java code embedded inside HTML.

* Tags.

→ Scriptlet (<% code %>) – java code inside JSP

→ Expression (<%= exp %>) – o/p results of Java exp.

→ Declarat° (<%! type var = val; %>) – Declares var/methods.

→ Directive (<% @ directive %>) – Controls page behavior.

Spring boot ↳ Rapid App. Development.

↳ framework

→ Sub module of spring to build web app.

→ Develop & maintained by "Pivotal software".

↓

enterprise slw comp

Why Spring Boot?

- Auto configurat°
- embedded server.
- No xml configurat°
- Easy dependency management via starters.

* @SpringBootApplication → marks the main class of spring Boot App.

↓

combinat° of @Configurat°

@EnableAutoConfiguration

@ ComponentScan.

* auto-configurat°

↓

automatically configures beans based on dependencies using @EnableAutoConfiguration.

* Spring Boot starters.

↓

predefined set of dependencies for specific purpose.

eg:- spring-boot-starter-web

spring-boot-starter-data-jpa.

spring-boot-starter-security.

* Spring vs Spring Boot

Spring	Spring Boot
• Manual XML config.	• auto-configuratn
• Server-req. deployment	• embedded servers

*
* Default embedded server - Tomcat
* Default port - 8081
* purpose → application.properties → Server settings, DB configuratn
* Spring Boot Actuator
  ↳ exposes product ready end-points health, metrics, beans etc
  eg. /actuator/health, /actuator/info

* To connect DB in Spring Boot.
  ↳ Add JPA / JDBC dependency
  → set DB prop. in application.properties
  → Create repo using JPARepository.

• JPA
  ↳ Javax Persistent API - 2006
→ specificatn, simplifies interactn blw Java obj. & Relational DB.
→ 2019 → Jakarta Persistence API
  ↓
  provides & interfaces.
  1) CRUDRepository.
  2) JPARepository.
     ↳ • save
       • delete
       • update
       • get

• API
  ↳ Applicatn Programming Interface.
→ Bridge blw system or Devices.
  eg. LoginAPI
  PayMent Gateway API

Types.
↳ • Public
  • Partner
  • Private

• Spring framework.
  ↓
  open source, lightweight framework for building enterprise app.
  ↳ Benefits:-
  • Loose coupling blw comp.
  • Testability
  • Modularity
  • easy integratn with ORM.

• DI - Dependency Injectn.
- design pattern, one obj. injects into another.
Types
↳ Constructor-based
↳ Setter-based
↳ Field injectn.

• ~~Beanfactory vs AppContext~~

Modules of Spring.
- Core
- Beans
- Context
- AOP
- Spring MVC
- Spring JDBC

* Open API – can be used by anyone.
* Partner API – for business to business (B2B)
  eg. MakeMyTrip.
* Private API – only for specific or internal working
* Composite API – combines any two or more API's for any System / project.

Web Services
↳ specific type of API designed to follow set of protocols for communicatn.

Types
↳ SOAP → simple object Access Protocol.
↳ REST → Representational state Transfer.

* Rest API
↳ architectural style for building web services using HTTP methods like Get, POST, PUT, DELETE

* Annotatns.
- @RestController – Masks class as REST controller.
- @RequestMapping – Maps HTTP req. to handler methods.
- @GetMapping – Handles Get req., fetch data.
- @PostMapping – Handles Post req., insert data.
- @PutMapping – update the data.
- @DeleteMapping – Handles Delete req.
- @PathVariable – Extract values from URL.
- @RequestParam – Extracts query param. from URL.
- @RequestBody – Maps req. body to Java obj.
- @ResponseBody – Returns java obj. as JSON / XML.

* RestController.
↳ @Controller + @ResponseBody
– returns data directly from REST endpoints.

* @Controller vs @RestController.

@Controller	@RestController
→ use - web App.	→ use - Rest APIs.
→ returns views like HTML	→ returns data like JSON / XML.
→ combined with @Response-Body manually	→ internally includes @ResponseBody automatically.
→ Suitable - MVC apps	→ suitable - Restful services.

* @RequestBody
  ↳ used to bind HTTP request body to java obj. especially for POST / PUT

* To handle Exceptions → @ControllerAdvice & @ExceptionHandler.

* ResponseEntity
  ↳ wrapper for HTTP response, allows control over status code, headers & body.

* @Autowired
  ↳ injects bean automatically into dependent class

* @Component ↓ Generic bean
  @Service ↓ Business logic layer.
  @Repository ↓ DAO layer.
  @Controller ↓ MVC controller

* @Value
  ↳ used to inject values from application.properties.

* Spring Boot Dev Tools. - enables
  ↳ auto-restart on code changes
  - live reload.
  - Developer productivity tools.

* Spring Boot Initializr
  ↓ web tool → generate Spring Boot projects with req dependencies.

Spring Boot starters :- pre configured dependencies.

To handle transact° → use @Transactional.

Spring Beans:-
  ↳ obj, managed by IoC container.
→ responsible for creating bean, injecting dependencies. Managing life cycle, Destroying it when no longer needed.

* @Controller vs @RestController.

@Controller	@RestController
→ use - web App.	→ use - Rest APIs.
→ returns views like HTML	→ returns data like JSON / XML.
→ combined with @Response-Body manually	→ internally includes @ResponseBody automatically.
→ Suitable - MVC apps	→ suitable - Restful services.

* @RequestBody
  ↳ used to bind HTTP request body to java obj. especially for POST / PUT

* To handle Exceptions → @ControllerAdvice & @ExceptionHandler.

* ResponseEntity
  ↳ wrapper for HTTP response, allows control over status code, headers & body.

* @Autowired
  ↳ injects bean automatically into dependent class

* @Component → Generic bean
  @Service → Business logic layer.
  @Repository → DAO layer.
  @Controller → MVC controller

* @Value
  ↳ used to inject values from application.properties.

* Spring Boot Dev Tools. - enables
  ↳ auto-restart on code changes
  - live reload.
  - Developer productivity tools.

* Spring Boot Initializr
  ↓
  web tool → generate Spring Boot projects with req dependencies.

Spring Boot starters :- pre configured dependencies.

To handle transact° → use @Transactional.

Spring Beans:-
  ↳ obj, managed by IOC container.
→ responsible for creating bean, injecting dependencies. Managing life cycle, Destroying it when no longer needed.

Constructor vs Setter Injection

↓ ↳ uses final for field

- Dependencies - provided through class constructor.
- ensures bean is fully initialized when it's created.

Setter

↳ Dependencies are set via public setter methods after obj. is created.

- can reconfigure dependencies after obj. creation.
- use for optional dependencies.

Annotation

↳ special form of metadata, provides information to the compiler & spring at runtime.

- used to configure beans, enable features, & reduce XML configuration.

Types

↳ stereotype (Marked classes as Spring beans)

↳ • @Component
• @Service
• @Repository
• @Controller.

→ DI annotation

↳ • @Autowired
• @Qualifier → used with @Autowired to inject by name.
• @Primary → Marks bean as default choice when multiple beans match.

→ Configuration ann.

• @Configuration
• @Bean
• @ComponentScan

→ Web & Rest.

• @RestController
• @RequestMapping
• @GetMapping, @PostMapping.
• @RequestBody
• @ResponseBody.

Autowiring

↓

automatically injects bean into another bean w/o explicitly specifying bean in configuration

modes

- byName
- byType
- constructor.
- autodetected.

#MicroServices.

Beans can be configured in 3 main ways.
① XML config?
② Java - based
③ Annotat? - based. → @Component

Bean scopes.
→ Singleton: one instance per Spring container.
→ prototype: new instance every request.
→ request: one instance per HTTP request.
→ session: one instance per HTTP session.

AOP → Aspect Oriented Programming.
↳ AOP lets you separate cross-cutting concerns from main business logic.
↓
Code that appears in multiple places.

Real life use:-
- logging
- security checks
- Transact? Management.
- Performance monitoring
- caching.

Bean Life Cycle
1. Instantiat? - object created.
2. DI - dependencies injected.
3. Initializat? - custom init method runs.
4. Ready to use - Bean available in app. context.
5. Destruction - Destroy method call before bean is removed.

Bean Wiring
↳ Connecting beans together in Spring container so they can work with each other.
- part of DI.

Types
↳ Explicit (XML)
↳ Autowiring (@Autowired)

IoC Container.
↳ core part of framework.
- Creates objects
- Manage their life cycle
- Injects dependencies.
- Configures them based on metadata.

Types
↳ BeanFactory
↳ ApplicationContext.

BeanFactory.
↓
- Basic container
- less commonly used directly.

Applicat? Context
↓
- extends BeanFactory
- Adds AOP support, event handling. etc.