**✅ Session Ground Rules**

**🧑‍🏫 Learning & Participation**

1. **Be Present:** Please join on time and stay for the full session.
2. **Stay Engaged:** Participate actively — ask questions, share thoughts, and collaborate **when prompted.**
3. **Use the Chat Wisely:** Drop questions, but avoid spamming or unrelated messages.

**🎧 Tech Etiquette**

1. **Mute When Not Speaking:** Keep your mic muted unless you’re asking or answering something.
2. **Camera Optional but Encouraged:** If you're comfortable, keep it on — it helps build connection.

**📚 Content & Recording**

1. **No Unauthorized Recording:** Please don’t record or share the session without permission.
2. **Materials Access:** Any slides, code, or recordings (if applicable) will be shared after class.

**🤝 Respect & Inclusion**

1. **Respect All Opinions:** There are no “silly” questions — we’re all here to learn.
2. **Keep It Professional:** No offensive, disrespectful, or disruptive behavior will be tolerated.

**🚨 Support & Issues**

1. **Facing Tech Issues?** Let me or the moderator know in the chat — we’ll try to help immediately.
2. **Missed Something?** Don’t worry, you’ll get resources to review.

**💡 Final Thought:**

“Learning is a team sport — **let's support each other and make the most of our time together**!”

**Phase 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 28/06/2025 | Java-1 | 4 | What is Java? Installing and cofiguring Java, Byte code, Compilation Process, Data Types, Operators, Conditional Statements, Looping statements, |
| 2 | 29/06/2025 | Java-2 | 4 | OOPS, Abstraction, Polimerphism, Inheritance, Encapsulation, Access Specifiers, Methods, Constructors**,** |
| 3 | 5/7/2025 | Java-3 | 4 | Static, final, Memory Management, Multi threading, Arrays, String Handling, |
| 4 | 6/7/2025 | Java-4 | 4 | Exception Handling, Collections |
| 5 | 12/7/2025 | Maven | 4 | Java 11 features, Web programming, CS Architecture, **HTTP Protocol**, Servlet Lifecycle, Web Server |
| 6 | 13/07/2025 | JDBC | 4 | Servlet API, Maven, Write first program, Servlet Filters, |
| 7 | 19/07/2025 | Servlet-1 | 4 | Servlet Session management, JDBC API, Servlet with JDBC Example |
| 8 | 20/07/2025 | Servlet-2 | 4 | JDBC CRUD Operations along with Statement , Prepared Statement, Callable Statement, JSP introduction(Scriping tags, implicit objects) |
| 9 | 26/07/2025 | JSP-1 | 4 | Directives, Action Tags |
| 10 | 27/07/2025 | Mongo -1 | 4 | Mongo-1 |
| 11 | 2/8/2025 | Mongo-2 | 4 | Mongo-2 (data modelling, validations, index) |
| 12 | 3/8/2025 | QA and PE project | 4 | Phase end project explaination |
| 13 |  |  | 4 |  |

**13/09/2025**

Spring AOP

1. Aspect

A module that encapsulates cross-cutting concerns (e.g. LoogingAspect).

1. Joint Point

A Point in execution of a program where an aspect can be applied.

Ex: Method execution, exception thrown, object creation

1. Advice

Action taken at a joint point

* @Before 🡪 Runs before the method
* @After 🡪 Runs after method completes
* @AfterReturning 🡪 Runs only after successful execution
* @AfterThrowing 🡪 Runs only if method throws exception
* @Around 🡪 Runs before and after (most powerful, can control method execution)

1. Pointcut

Expression that matches the joint point

Execution(\* com.example.service.\*.\*(..)) 🡪

**Why Use AOP?**

* It reduces boilerplate code (don’t need to repeat logging everywhere)
* Makes code cleaner and maintainable
* Helps in separating concerns 🡪 business logic styas focused.

**Cross cutting concerns:**

* Logging
* Security
* Transactio

Void test() {

}

Void aspect() {

}

**Assessment**

**Usecase:**  Student – Course Registration System

1. Student Service

* Manage students (CRUD operation)

**API**:

POST: /students 🡪 add a new student

GET: /students/{id} 🡪 Get student details

1. Course Service

* Manage courses and enrolments
* Needs to talk to student service to validate student before registering them in a course (RestTemplate)
* APIS:
  + POST /courses/{coursed}/enroll/{studentId} 🡪 Enroll student into course
  + GET /courses/{coursed}/students 🡪 List all enrolled students

**Flow**:

* Client calls Course service 🡪 /courses/101/enroll/1
* Course service 🡪 call student service 🡪 students/1 to check if student exists
* If student exist 🡪 enroll them
* If not 🡪 return error “Student not found”.

**07/09/2025**

**DB Management patterns:**

1. **Database per service pattern**

In monolithic, this is only one big database shared by all modules.

e.g. users, orders, products, payments tables all in single DB

In microservice, each service is independent (autonomous)

* To achive this, each service must own its database.
* This avoids tight coupling between services.

Why to use this?

* Loose coupling
* Independent scalability
* Technology choice
* Fault isolation

Challenges

* Cross -service queries become difficult
  + In monolythic, you could use JOIN
  + In microservice, you need APIs or event driven approach
* Transactions across services – No simple ACID guarantees. Need Saga pattern or eventual consistency.

1. **Shared database**

Multiple services use the same database instance and schema.

Each services directly accesses the shared tables.

This is common in monolith applications and sometimes used in microservices during early migrations.

When to use it?

* Simplicity
* Cross-service queries
* No complex distributed transactions

Challenges:

* Tight coupling
* Scaling limits
* Team independence lost
* Failure impact

1. **CQRS (Command Query Responsibility Segregation)**

In traditional applications, read (queries) and writes (Commands) use the same data model and the same database.

But in large scale systems, read and write needs very different:

* Reads are often frequent, complex, optimized for reporting
* Writes need to be transactional, normalized, strict on business rules

Why to use this pattern?

* Performance
* Scalability 🡪 reads and writes scale independently
* Flexibility 🡪 Read model can use different storage (SQL, NoSQL , Search engine)
* Complex domains 🡪 Supports different needs of reporting vs transaction processing

**Challenges:**

* Increased complexity 🡪 manage two modes
* Eventual consistency 🡪 Read DB lag behind write DB.
* Operation overhead 🡪 Need sync mechanism

**E commerce**

* **Write 🡪** Place order, payment processed --. Stored in normalized tx DB (MySQL)
* **Read 🡪** Order history, dashboard, product search 🡪 stored in Elastic search or mongo db

**User gets fast reads** even during high order traffic

1. **Event sourcing**

In most systems, we store only current state of data.

* Example, a bank account table has a column balance = 10000.

Problem? History ?

**Event sourcing pattern**

Instead of storing just a latest state, we store **sequence of event**s that represent every change

Why to use this pattern?

* Auditability 🡪 Every action Is recorded
* Debugging/reply🡪 You can replay events to reconstruct the systems’s state at any point in time

Challenge:

* Data growth 🡪 Event store keeps growing , needs snapshot for performance
* Complexity
* Querying

Real world usecase:

* Banking system
  + Instead of storing just balance, store every credit and debit event.
* Ecommerce order management
  + Events: Order placed, PaymentAuthorized, OrderShipped, OrderDelivered
  + Helps with nalytics, fraud detection , customer service queries

1. **Saga pattern**

IN case of monolith, 🡪 single DB 🡪 ACID

But In Microservices, each service has its own database 🡪 you can’t use single ACID transaction across multiple services.

Solves, Long transactions in to a **sequence of local transactions**.

* Each local transaction updates its own database
* If something fails, a compensating transaction is triggered to undo previous steps

**C**horeography Saga Pattern (Event – based)

* Services communicate through events
* Each services listen for events and reads accordingly
* No central controller

Sinple, decentralized.

Harder to track/debug

**Orchestration (Cental controller**)

* A Saga orchestrator tells each service what to do.
* Services only execute steps, Orchestrator handles flow and rollbacks.

Coordinate sagas through central controller

Orchastrator handles flow and rollbacks.

Easier to manage and monitor

Central orchestrator can become bottleneck

1. **Outbox pattern**

**The** Outbox pattern ensures automicity between database writes and event publishing.

How it works:

* When writing data to the services’s DB, also insert an event record into an “Outbox table”
  + This gurrantees both are stored together
* A separate outbox processor reads from outbox table
* It publishes events to the message broker reliably
* After successful publish, event is marked as processed (or deleted)

**06/09/2025**

**NFRs – Non Functional requirements**

1. **Scalability**

Systems ability to handle increased load (users, data, traffic) without breaking performance.

Vertical scaling (scale up)🡪 adding up more power in existing machine

Horizontal scaling (Scale out)🡪 Add more machines/nodes

1. **Availability**

System’s ability to remain operational and accessible.

Usually measured as uptime% 🡪 99.9% available 🡺 in a year 8.7 hours downtime

Techniques:

* Redundancy
* Failover system
* Load balancing across regions

1. **Reliability**

Probability that the system works correctly and consistently over time.

Availability – is it up?

Reliability 🡪 Does it work as expected when it’s up?

Techniques:

* Retry logic with backoff
* Constency checks
* Circuit breakers

1. **Maintainability**

Ease with which system can be modified, extended, debugged.

Practices

* Modular design
* Layed architecture (controlle r🡪 service 🡪 repository)
* Good logging & monitoring

1. **Usability**

How easy the system is for end-users to understand and operate.

Practices:

* Clear UI/UX
* Consistent APIs (Error code 🡪 )
* Documentation

1. **Efficiency**

Optimal use of system resources (CPU, memory, network, storage)

Practices:

* Caching
* Optimized DB Queries
* Async processing

**Scalability & availability 🡪 handled at architectural level (Cloud infrastructure)**

**Reliability and maintainability 🡪 handled in code + devops practices**

**Usability 🡪 handled in UI/UX + API design**

**Efficiency 🡪 Handled in performance tuning.**

**Monolythic architecture**

Single, Unified, codebase where all applications components (UI, Busienss logic, data access) are tightly coupled and deployed togher as one unit.

When to use it?

* Small applications or startups
* When team size is very small.
* When features are limited and well defined.

Pros:

* Simple to develop, test and deploy
* Easy to debug
* Good for small teams

Cons:

* Scalability
* Maintainability
* Deployment risk
* Technology lock-in

**SoA (Service Oriented Architecture)**

**Distributed monolith**

**ESB – Enterprice service bus**

A screenshot of a computer

AI-generated content may be incorrect.

**ESB –** Centralize software components

Connectivity, messaging, routing, , composition of multiple requests 🡪 ESB

Pros:

* Handle all cross cutting concerns like message valiations, transformation, content based routing, security, load balancing

Problem

* Single point of failure

**Microservices**Microservices are small business services that can work together and can be deployed independently. Those services are communicating with each other via network

Characteristics:

* Small, independent, Loosely coupled services.
* Separate codebase
* Persisting their own data (Polygot persistence)
* Well defined API (Communicated with each other)
* Technology agnostics

Characteristics by “Martin flowler”

* Componentization via services
* Organized by business capability
* Products not projects
* Smart endponts and dumb pipes
* Decentralized governance
* Decentralized data management
* Infrastructure automation
* Design for failure

Benefits:

* Agility
* Small and focused teams
* Small and focused code base
* Right tool for job
* Fault isolation
* Scalability
* Data isolation

Challenges:

* Complexity
* Network problems and latency
  + Reduce latency by asynchronous ms communivation (message broker)
* Data Integrity

Decomposition patterns:

1. Decomposition by Business capatibility

* Services must be cohesive
* Services must be loosely coupled (isolated)

1. Decompose by SubDomain

Using DDD – Domain Driven Design

DDD is an approach to designing complex software by modelling the software to match the problem domain, using collaboration with domain expert and disciplined language first modelling practice.

Goals of DDD

* Captures real business intent in the code
* Make complexity manageable by partitioning the domain
* Improves communication between developers and domain expoerts through **shared common language**

**Ubiquitous Language**

* A single vocabulary by both domain expoert and developers. It’s the source of the truth for the teams used in conversations, doc APIs.

**Identify Bounded Context**

A logical boundary within which a particular model is consistent.

**Context mapping patterns**

**Relationship types**

* **Partnership – Context**
* **Shared kernel – Common part of context are extracted to another context**
* **Customer supplier –** connection between two context where one context produces data and another consume it
* **Confirmist –** Upstream and downstream
* **Anticurruption layer –** Used for legacy system to adapt them to new architecture gradually

Question:

Is bounded context == microservice?

Entire decision taken based on requirements and scalability

Microserivces is autonomous and responsible by domain capability

**Microservice communication**

Synchronous communication:

1. REST 🡪 HTTPS protocol
2. gRPC

Opensource Remote Procedure calls, system initially developed by google.

Focused on high performance for inter service communication

HTTP2 protocol (to transport binary message)

Asynchronous communication

* AMQP
  + RabbitMQ
* Kafka

Benefits of API gateway

* Routing
* Request aggregation
* Utilise Service discovery to identify location of service
* Load balancing
* Authentication and authorization
* Throttling
* Logging
* Considered as middleware

**31/08/2025**

**ORM Relationships**

In Object oriented programming we use **references** between classes.

Class Car {

Engine engine;

Car(Eigine e) {

This.engine = e;

}

}

Car c = new Car(new Engine());

entityManager.save(c)

**In Relational DB (RDBMS) we use foreign keys.**

Hibernate/JPA brige this gap:

* Java object references 🡪 db foreign key relationship
* We define associations (one-to-one, one-to-many, many-to-many)

Types of relationsihps:

1. One to one

One record in table A is associated with exactly one record in Table B.

User has profile

Class User {

Profile profile;

}

1. One to Many / Many to One

One record in table A can be associated with many records in Table b, each record in B belong to exactly one in A.

Department has employees.

Bank has many accounts

1. Many to Many

Student has courses

Author has books

**@Entity**

**@Table(name="student")**

**class Student {**

**@Id**

**@GeneratedValue**

**@Column("s\_id")**

**private int student\_id;**

**@Column("fname")**

**private String fname;**

**@Column("lname")**

**private String lname;**

**@OneToMany(mappedBy="stu")**

**List<PhoneNumber> phoneNumbers;**

**}**

**class PhoneNumber {**

**@ManyToOne**

**@JoinColumn(name="stu\_id")**

**Student stu;**

**}**

**save(student)**

**Student**

**id fname lname**

**1 Dhruvik Parikh**

**PhoneNumber**

**id number stu\_id**

**1 999999 1**

**2 889898 1**

**Remember:**

**Always many side of entity --> create foreign key column**

**1. @OneToMany and @ManyToOne**

**2. Many side of entity (Owner side) --> @JoinColumn**

**3. Non owner side (one side) --> mappedBy value is property namess**

**In-Memory Database:**

In memory DB rely on system as opposed to disk space for storage of data. Because memory access is faster than disk access.

Creating test cases (integration testcases)

**H2 DB**

**HSQL DB**

**Derby**

**SpringDataJPA**

* **No more DAO implementations.**

**Auto generate dao class for us.ss**

**Create interface 🡪 extends 🡪 JpaRepository**

**Assignments:**

1. **Student CRUD operations (Similar to movie CRUD implemented)**
2. **Learn Many to many relationship (learn and add implementation Student has courses)**
3. **Write custom join query using Query annotation**

**30/08/2025**

**REST**

* **Representational State Transfer**
* It is not a protocol or framework, but architectural style for designing distributed systems, web services
* REST was introduced by ROY FIELDING in PhD dissertations (2000)

**In REST:**

* A resource (data or service) is identified by URI (Uniform Resource Identifier)
* Client interact with resources using HTTP methods (GET, POST, PUT, DELETE)
* Data is usually transferred by JSON or ~~XML~~ (mostly JSON now a days)

**Constraints**

1. **Uniform Interface**
   1. Consistent resource identifications using URIs
   2. Standard operations (GET, POST, PUT, DELETE

Resource based

Manipulation of resources through representational methods (HTTP)

Self descriptive url

1. **Stateless 🡪 HTTP Authentication**
2. **Cachable**
3. **Client - server**
4. **Layered system**
5. **Code on Demand(optional)**

**Client upload any file 🡪**

**MultiPartFile**

**ORM Framework:**

**ORM – Object Relational Mapping**

* It is a technique to map Java Objects to database tables and vice versa.
* Instead of writing SQL manually for every insert, update, delete we can just use objects and let ORM handle SQL generation.

Traditional way:

* String sql = “Insert into employee(id, name, salary) values (?,?,?)
* Preparedstatement ---. executeUpdate

ORM Way:

Employee emp = new Employee(101,”DRK”, 500000);

entityManger.persist(emp);

**Benefits**:

* No boilerplate JDBC code
* Database independence
* Caching for performance
* Transaction management support
* Works with OOP concepts (Inheritance, associations)]

Popular ORM frameworks: Hibernate, EclipseLink, OpenJPA, iBatis

JPA – Java Persistence API

* Specification provided by Jakarta EE
* It defines interfaces, annotations, rules for the ORM
* Think of it as contract/blueprint that ORM frameworks must follow.

Example JPA annotations:

@Entity 🡪 Marks java class as database entity

@Id 🡪 Primary Key

@GeneratedValue 🡪 Auto increment PK

@Column 🡪 Maps fields to the table column

@Table 🡪 Maps entity to table.

Hibernate as JPA implementation

* Hibernate is most popular implementation of JPA
* JPA defines what should be done, Hibernate provides how it is done.

Why?

* Full JPA support + advance features
  + Caching (1st and 2nd level cache)
  + Criteria API support
  + Better dialect support for different databases
  + Lazy loading and fetch strategies.

**24/08/2025**

JDBC Template

* Before JDBCTemplate, if you use plain JDBC
  + You had to write boilerplate code, for opening/closing connections, creating statements, handling exceptions
  + You have to manually map ResultSets into Java Objects
  + If you forgot to close connection, you risked memory leaks.
* Spring’s JDBC Template solved this by:
  + Handling connection management (via DataSource)
  + Reducing boilerplate code
  + Providing convenient query and update methods
  + Support RowMapper to easily map database rows 🡪 java objects
* Core concepts:
  + DataSource 🡪 Provides DB Connections (managed by spring boot auto-config)
  + JdbcTemplate 🡪 main class to perform SQL queries/updates
  + RowMapper 🡪 maps each row of ResultSet to Java Objects
  + Exception Translation 🡪 Converts SQLException to DataAccessExceptions hierarchy

A diagram of service

AI-generated content may be incorrect.

**Stereotype annotations**

* @Controller
* @Service
* @Repository
* @Component

**23/08/2025**

**Spring Framework – Fundamentals**

Before spring,

* J2EE Development was heavy, lot of boilerplate code, complex XML configs and dependency on application servers (tomcat, )
* Developers struggled with testability, tight coupling and configuration overhead

Spring was introduced as a **lightweight** alternative to simply enterprise java development.

**Goals**:

* Simplify JavaEE development (no need heavy containers like Tomcat/JBOSS/WEbLogicc)
* Promote Loose coupling with DI (dependency injection)
* Enable easy testing
* Reduce boilerplate code with templates (JDBC, JMS etc)
* Provides modular ecosystem (you use what you need)

**Spring Core Principles**

1. IoC (Inversion of Control)

Objects don’t create their dependencies, they are injected.

1. Dependency Injection (DI)

Loosely coupling between classes, Spring container wires objects together.

1. Aspect-Oriented Programming (AOP)

Separate cross-cutting concerns (like Logging, exception handling, transactions, security)

1. Template based abstraction

Eliminates repetitive code (e.g. JdbcTemplate handles connections/cleanup)

**Spring Architectures (Layers and modules)**

**Core Container**

* **Spring-core:** IoC, DI
* **Spring-beans:** BeanFactory, managing lifecycle of beans
* **Spring-context:** ApplicationContext (events, resource loading)
* **Spring-expression:** SpEL (Spring Expression Language)

**Data Acces/Integration**

* **Spring-jdbc, spring-tx :** Simmplify DB access, transaction mgmt.
* **Spring-orm :** Hibernate/JPA integration
* **Spring-oxm:** Object/XML mapping
* **Spring-jms -** Messaging

**Web Layer**

* **Spring-web** - Basic web integration, multipart, REST
* **Spring-webflux:** Reactive, non-blocking (reactor)

**Test**

* Support for Junit, TestNG, mocking etc

**Spring Ecosystem**

Spring is not just one framework – it’s a whole ecosystem.

* Spring framework – Core, DI, AOP, web
* **Spring Boot** – Opinionated Convention over configuration
* **Spring Data** – Simplified layer for JPA, Mongo, Redis , Cassandra
* **Spring Security** – Authentication, Authorization, OAuth2, JWT
* **Spring Cloud** - Config server, service discovery, circuit breakers etc
* **Spring Batch** – Batch processing (ETL, Large data sets)
* **Spring Integration** – Enterprise integration patterns (messaging)
* **Spring WebFlux** – Reactive nonblocking programming
* **Spring GraphQL**  - API Query support
* **Spring AI** – (New) integration with LLMs

**DI - Dependency Injection – Fundamentals**

**Dependency** – An object that needs another object

**Injection** – Supplying the dependency from outside rather than creating it inside.

**Class Car {**

**Private Engine engine = new Engine()**

**}**

If we want PetrolEngine vs DieselEngine , we must change code.

Class Car {

Private Engine engine;

Car(Engine engine) {

this.engine = engine;

}

}

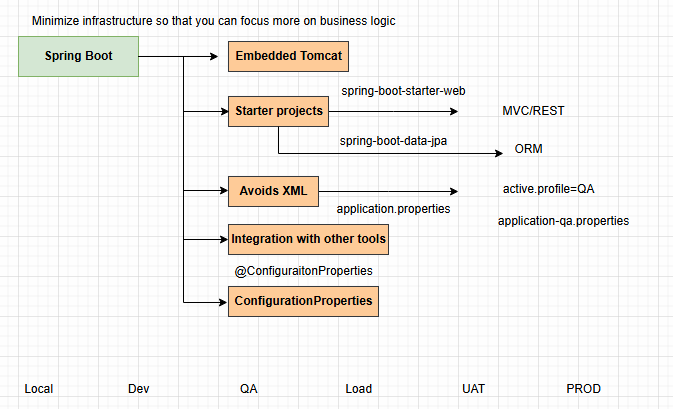
**IoC container**

* Normally we control object creation via (new keyword)
* With IoC, the container controls object creation and wiring
* Ioc Container manages:
  + Creating objects (Beans)
  + Injecting dependencies
  + Managing lifecycles (int, destroy)

Types of IoC containers:

* + BeanFactory(Basic, lazy initialization)
  + ApplicationContext (advanced, event handling, eager initialization, AOP, web support)

**Spring Boot**



**Spring Boot MVC**

**MVC** – Model View Controller - Design pattern for building web applications.

**Model** 🡪 Represents Data

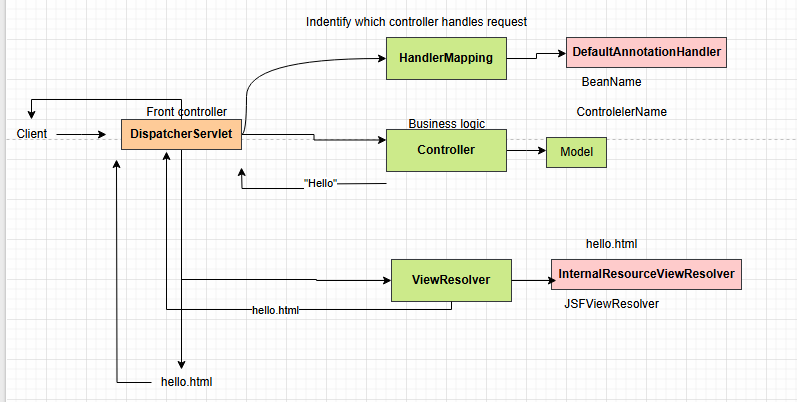
**View** 🡪 Represents UI (presentationa layer)

**Controller** 🡪 Handles request, coordinate between model and view

**Purpose :** Separation of Concern (SoC)

* Business logic is separate from Presentation logic
* Easy to maintain, test scale etc

**How Spring implements MVC**:



Thymeleaf

* Thymeleaf is a server-side java template enine for rendering HTML.
* Integrates tightly with Spring MVC
* Used as view layer in MVC applications
* Syntax is natural HTML –

Why Thymeleaf? (Vs JSP)

* Medern, HTML5 friendly
* Rich features (loops, conditions, fragments, layouts)
* No need for heavy JSP engine
* Good integration with Spring

Expressions:

* ${} 🡪 Access model attributes
* \*{} 🡪 Access selected object
* #{} 🡪 Message (i18)
* @{} 🡪 urls
* <p th:text = “${student.name}” > </p>
* <p th:text = “\*{name}” > </p>
* <p th:text = “@{/home}” > </p>

<p th:if= “${student.id == 1}”> This is ID 1 </p>

<p th:unless= “${student.name == ‘john’}”> not john </p>

**PHASE 4**

**02/08/2025**

* **Data types**
* **Data Modelling**
  + **Aggregations**
* **Schema validation**
* **MongoDB Index**

**Data types:**

* Text 🡪 “Dhruvik”
* Boolean 🡪 true/false
* Number 🡪 Integer (int32), NumberLong(int64), NumberDecimal (12.99)
* ObjectId 🡪
* ISODate 🡪 2024-6-06,
* Timestamp 🡪 11221532
* Embedded Document 🡪 {}
* Array 🡪 []

**Data Modelling:**

Mongo DB is schema-less, but structuring data correctly is still crucial

Data modelling in mongodb means deciding how to organize and relate data between documents to optimize read/write performance, scalability and developer experience.

**Two type of modelling**

1. **Embedded Documents**

You embed related data directly inside a document as a sub-document or array.

When?

* + One-to-One or One-to-few relationship
  + Related data is always access together

Benefits:

* Fewer collections
* Automic updates (single document)
* Faster reads

Cons

* Document size limit: 16MB
* Redundant data if reused across many documents
* Not suitable for frequently updated subdocuments

1. **Referenced Documents**

You store related data in separate collections and link then using ObjectIds (like foreign keys in RDBMS)

When?

* One-to-many or many-to-many
* Sub-documents grow large and change frequently.
* Reuse or related data (e.g. product categories)

**Simulating a Join (with $lookup)**

MongoDB doesn’t have joins like SQL, but you can simulate them using $lookup

|  |  |  |
| --- | --- | --- |
| Feature | Embedded | Referenced |
| Access Pattern | Access together | May access separately |
| Read performance | Fast | Slightly Slower (extra query) |
| Update complexity | Easy (Single doc) | Multiple writes |
| Document size | Must be within 16 MB | Multiple writes |
| Relationship Type | One-to-One, One-to-few | One-to-many, many-to-many |
| Data Resuse | Redundant | Centralized |
| Joins needed | No | Yes (via $lookup) |

Usecase

1. User with address

Why embedded?

* Each user has one address, and it changes rarely
* Address is tightly coupled with user an fetched together
* No need to query address independently.

1. Blog Post with comments (**limited**)

Why embedded?

* Commments are always displayed with blog post
* Bounded array size

1. Order with Line Items

Why embedded?

* Order and itmes are always accessed together
* Items don’t exist independently.

1. Users and Roles

Each user may have multiple roles, and each role is reused across users.

Why Reference?

* Roles are shared
* You want to update the role once and reflect changes in all users

1. Author and books

Author writes multiple books, books are access independently

Why Reference ?

* Books are queried independently (e.g. “list all books”)
* Author’s info is common but small.

**Bonus Tip:**

Ask:

* Will this sub-document grow unbounded?
* Do I always access it with the parent?
* Is it reused?

If:

* Small, single-use, access together 🡪 use Embed
* Shared, large, or queried independently 🡪 Reference

**MongoDB Schema Validation**

Although MongoDB is schema-less , real-world applications benefit from Data consistency.

* Enforce field types
* Require certain fields
* Apply constraints like min/max, enum etc
* Prevent garbage or unexpected data

Common needs for schema validations:

1. Preventing bad data (age:”abc”)
2. Enforcing required fields(like email in user record)
3. Restricting values (status: “active”|”inactive”)
4. Avoid partial/invalid documents
5. Protect consistent format

MongoDB supports schema validations via **$jsonSchema** keyword in collection or updates.

Validation level:

“strict” (default): Rejects any invalid insert/update

“moderate”: Allows existing invalid data but prevents new invalid data.

**Index in MongoDB:**

An index in MongoDB is like index of a book: it helps MongoDB find documents faster, without scanning entire collection.

Why?

Without indexes, MongoDB performs collection scan – reading every document to find matches.

With Indexes:

* Queries are faster
* Can support sorting
* Improve filter + pagination performance
* Reduce CPU & IO cost

|  |  |  |
| --- | --- | --- |
| Types | Use case | Example |
| Single Field | Index one field | {name: 1} |
| Compound | Index multiple fields | {name: 1, age: -1} |
| Multikey | For indexing array fields | Tage:1 |
| Text | Full text search | {description: “text”} |
| Hashed |  | {userId: “hashed”} |
| Geospatial |  | {location: “2dsphere”) |
| Wildcard | Index unknown or dynamic fields | {“$\*\*”:1} |
| TTL (Time to Live) | Auto delete documents after a time | {createdAt: 1} with TTL Option  db.usrs.createIndex({createdAt: 1}, {expireAfterSeconds: 3600})  Any doc older than 1 hour will be deleted |

**27/07/2025**

# MongoDB

What is MongoDB?

* MongoDB is a NoSQL, document oriented database that stores data in BSON (Binary JSON) format.
* Unlike relational database (RDBMS), which use **tables**, monbodb uses **collections of documents**.

**History:** created by MongoDB Inc. in 2007, MongoDB is popular for its high performance, scalability and ease of use in distributed environments.

**Why MongoDB?**

|  |  |
| --- | --- |
| **Dynamic Schema** | No need to predefine columns like in RDBMS |
| **Scalability** | Supports automatic sharding for large datasets |
| **High Availability** | Replica sets provide failover and data redundancy |
| **Rich Queries** | Powerful filtering, projection, and aggregation |
| **Geospatial Support** | Location-aware queries supported |
| **Cloud ready** | MongoDB Atlas offers cloud hosted DBaaS |

**Use cases**

* Real-time analytics
* Product catalogs
* IoT applications
* Content management
* Mobile and web apps with changing schema

**MongoDB EcoSystem**

1. **MongoDB Core**
   * Stores documents in collections
   * Supports CRUD operations
   * Handles Indexing, replication, and sharding
2. **MongoDB Compass**

GUI tool to interact with MongoDB.

|  |  |
| --- | --- |
| Features | Description |
| Visualize Data | Browse Collections and documents |
| Schema Explorer | View inferred schema with data types |
| Query Builder | Drag-drop or write complex queries |
| Index Analysis | See which indexes are used in queries |

Great for beginners and data analysts.

1. **MongoDB BI Connector**

Allows you to connect MongoDB to SQL based BI Tools like:

* + Tableau
  + Power BI
  + Qlik

Converts MongoDB documents to a virtual schema with tables/ columns, so traditional BI tools can run SQL queries.

1. **MongoDB Charts**

A Built in visualization and dashboarding tool for MongoDB Data.

|  |  |
| --- | --- |
| **Features** | **Description** |
| Built-in | No ETL needed, directly uses collections |
| Charts | Bar, line, pie, geo etc |
| Shareabl | Can embed in websites or share links |
| Secure | Uses Atlas permissions if on cloud |

1. **MongoDB Stich (now called Realm functions)**

A **serverless backend** that lets you:

* + Create APIs without without writing server code
  + Authenticate users with Google, Facebook, email etc

Useful for building mobile/web apps quickly without writing your own backend logic.

1. **MongoDB Atlas (Cloud)**

**Fully managed cloud version of MongoDB** , hosted by MongoDB Inc.

|  |  |
| --- | --- |
| Feature | Description |
| Multi-cloud | AWS, Azure, GCP supported |
| Auto-scalling | Increases/decreases capacity as needed |
| Backup and monitoring | Built-in |
| Global Clusters | For low-latency across worldwild |
| Security | TLS, IP whitelisting, encryption at rest |
| Free tier | Ideal for dev/testing |

Also integrate with:

* Charts
* Realm
* Triggers
* Search

1. **Mobile (Realm SDK)**

Used to build mobile apps with offline-first support.

|  |  |
| --- | --- |
| Feature | Description |
| Realm DB | Local Database for iOS/Android |
| Sync Engine | Syncs local DB with MongoDB Atlas |
| SDKs | Available for Swift, Kotlin, React Native, Flutter |
| Reactive | Data auto-updates on change |
| Secure | End-to-end encryption and user authentication |

<https://www.mongodb.com/try/download/community>

<https://www.mongodb.com/try/download/shell>

**Create**

insertOne(data, options)

insertMany(data, options)

**Read**

find(filter, options)

findOne(filter, options)

**Update**

updateOne(filter, data, options)

updateMany(filter, data, options)

replaceOne(filter, data, options)

**Delete**

deleteOne(filter, options)

deleteMany(filter, options)

**Embedded Documents**:

MongoDB supports 100 nested levels

**26/07/2025**

JSP Directive

**JSP Directive is a special instruction to the JSP container that sets up the overall structure of the resulting servlet.** It doesn’t produce any output but rather affects how the page is compiled and executed.

<%@ directive\_name attribute=”value” %>

3 types of directives:

1. <%@ page attribute=”value” %> 🡪 Defines page-level settings

* language -- Language used
* import -- Java classes to import
* contentType – MIME type(text/html)
* errorPage -- Page to redirect if an exception occurs
* isErrorPage -- true if current page handles exception
* session -- **true**/false to use session
* buffer -- Output buffer size
* autoFlush -- Whether buffer auto flushe

1. <%@ include %> 🡪 Includes static file at page-translation time
2. <%@ taglib %> 🡪 Declare a tag library to use custom tags (like JSTL)

JSP directive used to declare and use custom tag libraries, like JSTL or user defined tag libraries.

It enables you to use XML-like tags <c:forEach>, <fmt:formatDate> instead of scriptlets(<% %>) in JSP , which makes code much

* Clearner
* More readable
* Easier to maintain
* Safer

<%@ taglib prefix=”prefix” uri=”uri” %>

Pefix 🡪 used to access tags (like c fmt etc)

Uri 🡪 unique identifier for the tag library

**Common Tags libraries:**

|  |  |  |
| --- | --- | --- |
| **Prefix** | **URI** | **Purpose** |
| Core – c | <http://java.sun.com/jsp/jstl/core> | Core (loops, conditionals, variables> |
| Format – fmt | <http://java.sun.com/jsp/jstl/fmt> | Formatting (date, numbers, i18n) |
| Sql | <http://java.sun.com/jsp/jstl/sql> | SQL queries |
| Fn | <http://java.sun.com/jsp/jstl/functions> | Functions (e.g. string manipulations) |

JSP Action Tags:

JSP Action tags are predefined XML-based tags that perform specific tasks at runtime, like forwarding a request, including content, passing data to java beans, or instantiating them.

This will help to replace java scriptlets (<% %) and promote clear, MVC-friendly JSP code.

|  |  |
| --- | --- |
| <jsp:useBean> | Creates or locates a javabean |
| <jsp:setProperty> | Sets bean property |
| <jsp:getProperty> | Gets a bean property |
| <jsp:include> | Includes content from another resource (at runtime) |
| <jsp:forward> | Forwards the request to another resource |
| <jsp:param> | Passes parameters to another resource or bean |

**20/07/2025**

**What is JSP?**

* **JSP (Java Server Pages)** is a Server-side technology provided by JavaEE to create Dynamic web pages.
  + **An** HTML page + Embedded java code (using scriptlet tags)
  + Processed on the server by **JSP Engine**
  + Converts into a Servlet behind the scenes.

**Why do we use JSP?**

* To Build dynamic, data-driven web pages, where content changes based on user inputs or database values.
* **Easier to write and maintain** than plain servlets because you can mix HTML & Java directly.

**How Does JSP Work? (Lifecycle)**

* Client (browser) sends a request to .jsp page
* The JSP engine converts it into a **Java Servlet source file**.
* Compiles it into a .class file
* Servlet runs 🡪 generate HTML 🡪 sends back to browser.

**Advantages over Servlets**

* Cleaner separation of Presentation (HTML) and logic (Java)
* Easier to maintain for designer and developers working together.
* Supports **custom tags**, **EL**(Expression language), and **JSTL Tags**

**test.jsp**

**Scriptlet tags**

**A** Scriptlet is a block of Java code written inside a JSP file.

<% … java code ..%>

When JSP is converted into a servlet, the code inside the scriptlet is placed inside service() method of that servlet.

**Best Practices:**

* Avoid writing too much java code in the scriptlet – it mixes logic with presentation.
* Prefer EL & JSTL tags for cleaner code

**Expression Tag**:

Expression is used to output the value of java expression directly into html

<%= %>

This expression is evaluated and converted into a String.

**Use expression when you just want to output something to the client**

**JSP Declaration:**

* A declaration lets you to declare variables, method, or fields at class level of generated servlet.
* <%! .. %>
* Unlike Scriptlets, declaration are outside of **service()** method.

**Use declaration to define reusable methods, fields that maintain state (rate & not thread safe)**

# JSP Implicit Objects

There are 9 implicit objects that are automatically available to you in .jsp file.

1. request

Type: HttpServletRequest

Represents the current HTTP request, and lets you to access parameters, headers, cookies etc

1. response

Type: HttpServletResponse

Represents the response you send back to the client – allows you to set header, cookie status codes, etc.

You rarely use respone in JSP directly -= but good for sending custom headers

1. session

Type: HttpSession

Represents the user session – you can store/retrieve data that persist across multiple requests by the same user.

1. application

Type: ServletContext

Represents the entire web application context - data here is shared across all users and sessions.

1. Out

Type: JspWriter

Used to write content to the response (HTML)

System.out 🡪

out is buffered – content is written to the buffer before being sent to the client

1. Config

Type: ServletConfig

Holds configuration for this specific JSP/servlet

1. pageContext

Provides access to all scopes (page, request, session, application) and also helps to manage attributes and forward/include other resources.

1. Page

Type: Object (actually this – the JSP page itself)

Refers to the current JSP page instance

Not commonly used, but good to know

1. exception  
   Type: Throwable

Used only in error pages- holds exception that was thrown.

Exercise:

1. Build a page with a form

* Ask user to enter their name
* Store their name in session
* Display welcome message, Welcome , [name] on the next request
* Also set an application level attribute like visitorCount in the application scrope and increment it with every visit

**19/07/2025**

**HttpSession**

HttpSession is the easiest and most common way to manage user sessions in servlets:

* Server maintains session data
* Client gets a session ID (JSessionID)
* On each request, the browser sends this cookie (JSessionID)🡪 server retrieves session

Methods

* getSession() 🡪 get/create session
* setAttribute(String, Object) 🡪 Store data
* getAttribute(String) 🡪 retrieve data
* invalidate() 🡪 destroy session object
* getId() 🡪 get SessionID

**What is JDBC?**

JDBC provides standard API to communicate with any DB (if you have an appropriate driver)

You don’t need to write database specific code 🡪 thanks to JDBC Drivers handles it.

Components

JDBC API 🡪 Interfaces and classes in javax.sql

* DriverManager 🡪 Manages database drivers and connections
* Driver 🡪 Vendor provided implementation (MySQL, Oracle etc)
* Connection 🡪 Session between Java app and DB
* Statement 🡪 Executes SQL queries
* ResultSet 🡪 Holds results of a query

Besic Steps:

1. **Load and Register Driver**

Class.forName(“com.mysql.cj.jdbc.Driver”);

1. **Establish Connection**

Connection con = DriverManager.getConnection(“jdbc:mysql://localhost:3306/jfsd”,”root”,”root”)

Connection object represents the connection between your java app and the database

1. **Create Statement**

Statement 🡪 Simple SQL queries

PreparedStatement 🡪 parameterized queries (safer and faster)

CallableStatement 🡪 to call stored procedure

Statement stmt = con.createStatement();

PreparedStatement pstmt = con.preparedStatement(“Insert into users values(?,?)”)

1. **Execute Query**

ResultSet rs = stmt.executeQuery(“Select \* from users”);

For INSRET, UPDATE, DELETE:

int rows = stmt.executeUpdate(“INSERT INTO users VALUES(1, ‘John’)”);

1. **Process Results**

while(rs.next()) {

System.out(rs.getInt(‘id’));

}

1. **Close Resources**

rs.close();

stmt.close();

con.close();

**13/07/2025**

**Self Study:**

* **SQL Language**
* **MySQL DB**
* **Executor framework**
* **Java concurrency package**

**JDK 21 features:**

1. Pattern matching for switch
2. Record patterns
3. Virtual threads

Thread mapped directly to an OS thread 🡪 heavy and expensive

Virtual threads (from project loom), are lightweight and managed by the JVM, so you can create thousands of/millions of threads easily

🡪

* JVM Schedules virtual threads on small pool of carrier threads (OS threads)
* When a VT is blocked, it’s unmounted and doesn’t hold a carrier thread
* When it’s ready, then its remounted-on carrier thread.

1. Sequenced Collections

Collections like (List, Sets Maps) that have a defined encounter order, now implements:

* SequencedCollections
* SequencedSet
* SequencedMap

You can access first and last elements easily

1. ScopedValue

ThreadLocal 🡪 class used to create per thread variables

Normally, all threads share the same variable in the memory

Sometimes, you need separate copy of variable of each thread, so that no two threads see or modify each other’s value

Problems with ThreadLocal

* Easy to forget remove() – can cause memory leaks
* Mutable – can cause bugs
* Not designed for supporting virtual Threads

ScopedValue is a new feature in Java 21 (preview) designed to replace usecases of ThreadLocal

Provide per thread context, but,

* Immutable
* Scoped explicitly (exist only within specific block of the code)
* Much safer and faster – especially with virtual threads

1. String Templates (preview)

f” hello {name}”

String.format(“”, values)

String msg = STR.”Name: \{name}”;

1. Unnamed patterns and variables (preview)

You can now ignore variables of part of a pattern using \_

* Servlet Execution

**Steps:**

1. **Crate Dynamic web project**

You will find Target runtime automatically populated, if not don’t worry, you might not have configured server (in servers tab)

1. **Convert to Maven**

Right click on project 🡪 configure 🡪 convert to maven project

Open Pom.xml

<dependencies>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.0.1</version>

<scope>provided</scope>

</dependency>

</dependencies>

**Context Parameter and Init Parameter**

* Context Parameter (application wide)
* Init Parameter (per servlet)

Both are the way to pass configurations to your servlet without hardcoding values.

**Servlet Filter:**

A Filter sits between the client and target servlet.

* Intercept incoming requests before they reach the servlet
* Intercept outgoing responses before they go back to the client

Think of it as pre-processor and post-processor.

**Why Use filters?**

* Logging requests and responses
* Authentication and Authorization
* Compressing responses (e.g. zip)
* Modifying request and response headers
* Input validation or sanitization
* Blocking unwanted requests

‘

Client🡪 **Filter Chain** 🡪 Servlet 🡪 **Filter Chain** 🡪 Client

Client🡪 F1🡺F2 🡺 Servlet 🡺 F2 🡺F1🡺 Client

How to implement this?

javax.servlet.Filter interface

* Init(FilterConfig) 🡪 Initializaiton code, runs only once
* doFilter(ServletRequest req, ServletResponse res)🡪 Actual filtering logic
* destroy() 🡪 Cleanup code, runs once at shutdown

**Session Management:**

HTTP protocol is **stateless** protocol – every HTTP request from a browser to the server is independent and the server doesn’t remember anything about the previous request.

But real web aps (like shopping cart, login system) you need to maintain state across multiple requests from the same user

* Tracking session (or session Management)

Why?

* Identify user across requests
* Store user specific data (like username, image, cart content)
* Support login/logout functionality

Common Session Management Techniques

|  |  |  |
| --- | --- | --- |
| Technique | How it works | Example |
| Cookies | Small piece of data stored in browser and sent with each request | Set-cookie header |
| URL Rewriting | Appends information as query string to every URL | /profile?name=Dhruvik |
| Hidden form fields | Include session info in hidden <input> in forms | <input type=”hidden” name=”user” value=”Dhruvik”/> |
| HTTP Session API | Serverside session object provided by servlet API | request.getSession() |

**12/07/2025**

**Self study:**

**JDK 8 – Functional interface, Lambda expression**

**Java 11:**

Released in Sep 2018 -- LTS

1. Local-Variable syntax for **Lambda parameters**

Introducing var keyword in lambda parameters

1. New String methods

isBlank()

lines()

strip()/ stripLeading()/stripTrailing() 🡪 Unicode aware triming

repeat(n) 🡪

1. Java.nio.file.Files
2. **HttpClient (standardized)**

* They support synchronous and asynchronous calls
* Http 1.1/ and Http/2
* Web sockets

1. **Removed and deprecated**

Jaxb, jax-ws, corba etc

appletviewer, javaws etc

1. **New Garbage collector**

ZGC collector 🡪 as experimental

**What is Internet?**

* Global system of interconnected computer network (**TCP/IP** protocol)
* A network of networks
* Enables communication, file sharing, web browing, email streaming etc

**Key components:**

* **Servers:** Machine that provides resources and services
* **Client:** User’s computer /devices accessing services
* **Routers and Switches –** Manage & direct data flow
* **Protocols:** Rules for communication (HTTP, TCP/IP, FTP)

**Timeline:**

* **ARPANET (1969)**

US military project – small package switched network

* **Email, FTP, Telnet introduced**
  + 1990
* Tim Berners-Lee invents WWW with HTTP and HTML
  + 1995-2000
* Commercialization

2000-2010

* **Web2.0 🡺** social media, cloud computing…
* **Web 3.0 🡪 (**decentralization, AI, IOT)

**Web Programming**

* **Web applications**
  + WEbsites
  + Web apps (gmail, facebook, ….)
  + APIs (Backed services)
* **Technologies?**
  + **Markup Languages** – Presentations of the content (HTML Structure, CSS (styling), XML/JSOn 🡪 data)
* **Database**
  + **Relational DB –** MySQL, PostgreSQL
  + **NoSQL –** MongoDB, Cassandra
* **Programming Language:**
  + Backend – Java, Python, php,ndejs, ruby…..
  + Frontend 🡪 JavaScript, Typescript
* **Business Applications**
  + Domain specific applications 🡪 Ecommerce, banking, CRM, ERP

**Client Server architecture**

The Client (Browser/app) sends a request

The Server processes the request and sends a response

**Presentation logic Business logic DB**

**Two-Tier Architecture**

* Direct communication between client and database server
* Client has the UI + Some business logic
* DB Server stores and manages the data.

Ex.

Desktop application 🡪 client directly connects to the db

**Pros**:

* Simple to build
* Good for small applications

**Cons**:

* Not Scalable
* High load on DB server

**Three-Tier Architecture**

What is it?

Divides applications into three layers

* Presentation layer (client)
* Application layer (Server/business logic)
* Data layer (Database)

Why?

* Better scalability
* Better security
* Easier maintenance

Ex:

* Ecommerce site
  + Submit order 🡪 presentation
  + Order processing logic 🡪 Application (server)
  + Stores order details in DB 🡪 Data layer

**Web server/Application server**

Web server 🡪

* Accepts http requests from client (like browsers)
* Respond with static content

|  |  |  |
| --- | --- | --- |
| **Feature** | **Web server** | **Application Server** |
| Purpose | Handles HTTP requests & responses, serves static content | Provides business logic for applications, can serve dynamic content |
| Content | Static (HTML, CS, JS) | Dynamic , Transactioal, complex apps |
| Protocols | HTTP/HTTPS | RMI, JMS etc |
| Examples | Apache, Nginx, IIS | Tomcat,JBoss/Wildfly, Weblogic.. |

* A Web server can serve static pages quickly
* Application server can also execte java code to build some dynamic pages.

Examples:

* Images, CSS, JS 🡪 web server
* Order processing, DB Intereaction, 🡪 Application server

## HTTP Protocol

HTTP – Hypertext transfer protocol

Foundation of communication on the web, allowing client (browser) and server to exchange information.

* Application layer protocol
* Request response based
* **Stateless** 🡪 Each request is independent of others
* Connectionless 🡪
* Media independent 🡪

Components of http::

1. HTTP Methods

|  |  |
| --- | --- |
| **Method** | **Purpose** |
| **GET** | **Retrieve resource (Safe and idempotent)** |
| **POST** | **Submit data to the server** |
| **PUT** | **Replace a resource** |
| **DELETE** | **Delete a resource** |
| PATCH | Partially update a resource |
| HEAD | Retrive only headers |
| Options | Get supported methodsd |

1. HTTP Request structure
2. HTTP Response Structure
3. HTTP Status codes

1xx 🡪 Informational

2xx 🡪 Success (200 Ok, 201 🡪 Created)

3xx 🡪 Redirections (301 Moved permanently, 302 Foud)

4xx 🡪 Client error (400 Bad request, 404 not found)

Versions of HTTP

* HTTP/1.0 🡪 One request per connection
* HTPP/1.1 🡪 Persistent connections, chunked transfer
* HTTP/2 🡪 Binary, Header compression, Multiplexing
* HTTP/3 🡪 Uses QUIC instead TCP, faster, and secure

# Servlet

Servlet programming🡪 writing java programs that runs on a application server on the web to handle client requests and generate dynamic responses.

Why?

* Before servlets 🡪 CGI scripts 🡪 heavy and slow
* Servlets run inside a java process (servlet container) more efficient.

**What servlet is?**

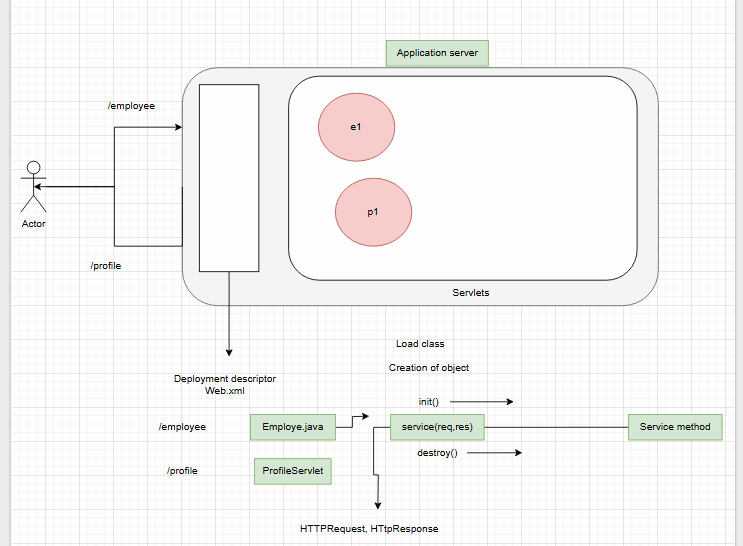
* A servlet is a java class that handles HTTP request & generates HTTP responses.
* Runs in a **servlet container** (Tomcat)
* Acts middle layer (3 layer) allows you to do business logic/ database

**Servlet lifecycle**

Init() 🡪 initialization of servlet

Service() 🡪 Request handling logic

Destroy() 🡪 destroy logic



javax.servlet.Servlet

two things:

serer 🡪 hold all the libraries

**Maven**

Maven is a **build automation** and **project management tool** for java and JVM based projects

Developed by Apche

Handls:

* Compilation
* Packaging
* Testing
* Dependency management
* Deployment

So you don’t have to manually download jars, and write complex build scripts or track dependencies

Why to use maven?

* Standardized project structure
* Makes builds portable
* Easily integrates with CI/CD tools
* Supports lot of plugins to extend functionality

Development workflow:

* Write code
* Pom.xml (define all dependency)
* Run maven command (mvn clean install)
* Get:
  + Compiled code
  + Jar/War file
  + Reports

**Setup project**

1. Download and install maven (<https://maven.apache.org/download.cgi>)
   1. Download zip file
   2. Extract it
   3. Create MAVEN\_HOME environment variable
   4. Include %MAVEN\_HOME%\bin in path variable (just like Java)
2. Download tomcat 8.1 (https://tomcat.apache.org/download-90.cgi) (extract it)
3. Configure eclipse to use tomcat
   1. Servers 🡪 click link 🡪 Select extracted jar 🡺 click finish
4. Configure eclipse to use your maven configuration
   1. Preferences 🡪 Maven🡪 installation 🡪 add 🡪select where you have extracted (maven)
   2. Apply and finish

**06/07/2025**

**Exception Handling**

**What is Exception?**

An exception is an unwanted or unexpected event that occurs during the execution of a program and disrupts the normal flow.

**Why Use Exception handling?**

* To handle errors gracefully, without crashing the program
* To Separate error-handling code from regular code
* To provide meaningful message about error
* To recover from errors and continue execution

**Errors and Exceptions**

* Errors generally cannot be recoverable
  + E.g. OutOfMemoryError, StackOverflowError

Best Practices:

* Catch Specific exceptions, not just Exception
* Always clean up resources in finally or use **try-with-resources**.
* Don’t overuse checked exceptions – only where needed.

**Java Collections framework:**

A set of interfaces and classes,

* Helps you to manipulate group of objects

Why?

Instead of writing your own code for:

* Array that grows dynamically
* Removing duplicates
* Sorting elements

Ready-made, tested, fast and flexible classes.

**Benefits**:

* Reduce development time
* Easy to use and maintain
* Many ready-made datastructurs: List,Set,Queue, Map etc
* Built in algorithms : sort, search, shuffle etc

Iterable

* Root interface for JCF
* Any collections like (List, Set etc) implements Iterable
* It represents group of elements you can hoop over one by one

iterator()

forEach()

for(String name: **names**) {

}

**Collection:**

* Collection is the root interface for all collections in java. (Except Map)
* It extends Iterable, so it already supports iteration
* It represents group of elements.

Common operations:

* Add elements
* Remove Elements
* Check Size
* Check if empty
* Clear all elements
* Convert to an array

**List Interface**

* **Subinterface** of Collection.
* Represents an **ordered collection** of elements
* Duplicates are allowed
* You can access elements by index (position)

LinkedList:

* Backed by doubly-linked list
* Good for frequent insertions and deletions
* Slightly slower for random access.

Set interface:

* Set is subinterface of Collection
* Represents a collection of unique elements
* No **duplicates** allowed
* **Order is not guaranteed** (but some implementations maintain order)

**HashSet**:

* Backed by hash table
* Doesn’t maintain order

**LinkedHashSet**: (maintain insertion order)

* Backed by hasthtable + linked list

**TreeSet**:

* Backed by a Red-Black Tree
* Maintains sorted order (ascending by default)
* Slower than HashSet.
* Set of employee objects (id, name, address, …) , how you sort it using TreeSet?

**Queue:**

* Queue is subinterface of Collection
* Represents a collection of elements in a specific order, usually FIFO.
* Think of it as ticket counter – first person in the line gets served first.

Why?

* Process eleemtns in order of arrival
* Useful for scheduling tasks, messaging, job queues, etc.

**Methods:**

* add(E e) 🡪 Adds an element (throws exception if full)
* offer(E e) 🡪 Adds an element (return false if full)
* remove() 🡪 Removes and return head ( throws exception if empty)
* poll() 🡪 Removes and return head (return null if empty)
* element() 🡪 Returns head without removing (throws exception if empty)
* peek() 🡪 Returns head without removing (returns null if empty)

Implemented class:

* LinkedList
* PriorityQueue – Orders elements by priority (natural)
* ArrayDequeue – (Double-ended queue- acts as both queue and stack)

Map

* Map is not subtype of Collection – its separate interface
* Represents a collection of key-value pairs
* Keys must be unique, but values can be duplicated
* You use it when you want to associate a key to a value -

Why use?

* Data by unique id
  + Student roll no 🡪 name
  + employeeId 🡪 Employee Details
  + Word 🡪 frequency of word

Methods:

* Put(x,y) 🡪 Add a keyvalue pair
* get(k) 🡪 get value by key
* remove(k) 🡪 Remove key-value pair
* containsKey(k) 🡪 check if key exist
* ContainsValue(v) 🡪 check if value exist
* keyset() 🡪 returns set of keys
* values() 🡪 Returns collection of values
* entrySet() 🡪 Returns a set of Map.Entry (key value pair)

Implementaitnos:

* HashMap
* LinkedHashMap
* TreeMap

**05/07/2025**

**Static:**

Used to define variables, methods, blocks, or nested classes that **blong to the class rather than to any specific object**

* Normally , members of class (fields or methods) are belong to the instances (objects)
* But static members belong to the class itself, so they are shared by all instances and can be accessed without creating an object.

**Why Static?**

* To save memory – only one copy exists for the class
* To represent things that are **common** to all objects of the class.
* To allow access without creating an object.

**What is Memory Management?**

Memory Management in java is the process of

* Allocating memory to objects
* Keeping track of allocated memory and
* Automatically reclaiming memory that is no longer needed.

M/R management Is handled mostly automatically (by JVM), and GC (Garbage collector)

Two parts:

1. Heap Memory
2. Stack Memory

There are few smaller regions for internal management.

**Heap Memory:**

* Used for dynamic memory allocation (objects, arrays)
* Whenever you use new keyword to create an object, memory allocated in the heap.
* Managed by Garbage collector.

**Stack Memory:**

* Used for method calls and local variables.
* Each thread gets its own stack.
* Follows LIFO
* Memory is automatically freed when the method call ends.
* Stores
  + Local varaiables (int , float)
  + Reference to the objects (but actual object is on heap)

**Method Area:**

* Stores bytecode, static variuables, method metadata (blueprints)

**Final keyword**

* The final keyword in java is non-access modifier that is used to declare something as constant, non-changeable, non-extendable, depending on where it is applied.
* Final with
  + Variable
  + Method
    - Method cannot be overridden in a subclass
  + Class
    - Cannot be extended (inherited)

**Why?**

* To create constants (Unchangeable variable)

**Multi-Threading**

* What is Multithreading?
* Process vs Thread
* Where threads are stored in memory
* Thread Lifecycle
* Race Conditions
* Synchronization
* Inter-Thread communication (Wait/notify)

**What is Multithreading?**

Multi-threading means running multiple threads concurrently within a single program (process)

* Thread is the smallest unit of execution
* Java supports multi-threading to perform multiple tasks at the same time- increasing resource utilization and performance.

**Why Use Multithreading?**

* Better CPU Utilization
* Improved responsiveness of program
* Useful for task like file I/O, GUI events, network calls, background calculations (spell checker in word)

|  |  |  |
| --- | --- | --- |
| **Feature** | **Process** | **Thread** |
| Definition | Independent executing program | Smallest unit of execution inside a process |
| Memory | Each has its own memory space | Threads share memory of the process |
| Communication | Difficult (inter-process communication) | Easy(shared memory) |
| Overhead | High(more resources) | Low(Ligghtweight) |
| Example | Running MS Word & chrome as separate program | Two tabs running in the chrome  Spell checker in word..etc |

**Where Thread stored in Memory?**

* Each thread has its own stack (for method calls and local variables)
* But all the threads share heap memory of the process – meaning they can access and modify same objects

**Thread Lifecycle**

1. **New** – Thread is created but not yet started
2. **Runnable** (Ready) – Thread is ready to run but waiting for CPU
3. **Running** – Thread is currently executing
4. **Waiting/Blocked** – Thread is waiting indefinitely (or blocked) for another thread or some time
5. **Terminated** (dead) – Thread has finished execution

How you create thread?

Two ways:

1. **Implementing thread interface**
2. Extending thread class

**Race Condition**

A race condition occurs when:

* Two or more threads access shared data at the same time
* And at least one thread modifies it
* And result depend on the order of execution of threads

Synchronization (solution to race condition)

* Java provides **synchronization** to ensure only one thread at a time can access critical section.

**Every thread object has special property**

1. Lock – ~~1~~ 0
2. WaitListedSet {t1,t2,t3 }

**Inter-Thread communication**

Sometimes, one thread needs to wait for another thread to complete or send a signal.

Wait can only be called within **synchronized block**

Instead of busy-waiting (wasting CPU) time.

Mohit, Dhruvik, Thiru, Supriya

this.wait()

this.wait()

this.notifyAll()

Wait and notify

Wait() – tells the thread to go into waiting state

Notify() – wakes up one waiting thread

NotifyAll() – wakes up all waiting threads

Customer

1000 - balance

Deposit withdrawal

**29/06/2025**

**OOP – Concepts**

**Class and Objects**

Class – blueprint/Template

Class car {

Void drive() {

System.out.println(“”);

}

}

Car c = new Car();

c.drive()

**Encapsulation** – “Data Hiding and control”

Encapsulation means binding data and methods together and **restricting direct access to internal variables.**

Key points:

* Ensures security and control
* Improves maintainability

**Inheritance – “Code Reusability”**

Inheritance is the ability of one class to inherit fields and methods of another class

Key points:

* Promotes code reuse
* Supports Hierarchical and multilevel inheritance
* Base class == super class, derived class = subclass

**Polymorphism – “Many forms of one thing”**

Polymorphism means that a method or objects behaves differently in different context.

**Two type:**

1. **Compile-time Polymorphism (Method overloading)**

Same method name, different parameters in the same class

1. **Runtime Polymorphism (Method overriding)**

Same method name and signature in parent class and child class, behavior depends on the object type at runtime.

“Parent class can refer to the instance of the child class”

**Key points**

* Overloading – same class, compile time
* Overriding – Different class (Inheritance), runtime
* Achieves flexibility and extensibility

Dynamic Method dispatch is the mechanism by which a call to an overridden method is resolved at runtime, not compile time.

**Abstraction:**

Abstraction is the process of **hiding internal implementation** details and only **exposing essential features**.

Achieved using abstract class and interface.

* You define what needs to be done, not how
* Hides internal complexity
* Focus subclasses to implement logic

**Banking System**

Scenario:

You are building banking system with different types of accounts:

* SavingsAccount
* CurrentAccount
* FixedDepositAccount

Each account has a method called calculateInterest(int days) that behaves differently depending on the account type.

Parent class: BankAccount 🡪 Override the method in each subclass,

**What are access specifiers?**

Access specifiers in java are keywords used to set visibility levels for the classes, methods, variables and constructors.

They can control who can access a particular piece of code – whether it’s within the class, package, or across other packages.

1. **private**

Accessible only within the **same class**.

1. default

Accessible only **within the same package**.

Analogy: your local society notice board - accessible only to residents, not outsiders

1. protected

**Accessible**:

* within the same package
* In the subclasses (even in other package)

Analogy:

You share your family Netflix password only within family members (subclasses), even if they live in different cities(packages).

1. Public

Accessible from anywhere – any class, any package

Best practice:

* Use private for data hiding (encapsulation)
* Use public for class/Interface meant for external use
* Use protected when designing extensible classes (subclass access)
* Avoid using **default** unless you have package-scoped logic

**Methods:**

A method is block of code that performs a **specific task**, **can be reused**, , and **may return a result**.

**Constructors in java**

A constructor is a special method used to **initialize objects** when they are created. It has no return type (not even void) and **same name as the class**.

Types:

1. Default – No parameters, added by **compiler** if **none defined**.
2. No-Arg – User defined constructor with no parameters
3. Parameterized – Accepts parameters to initialize object values

If you do not call super() method (first line of the constructor) then compiler will add it.

Java.lang.Object – be the super class for all the classes that you create.

**What is copy constructor?**

A copy constructor is a special constructor that creates a new object by copying the fields of an existing object.

**28/06/2025**

What is Java?

* High level
* Object oriented
* Class based programming language
* Platform independent (**WORA** – Wrinte once, Run Anywhere)
* **Robust** and Secure

Code.java 🡪 .class 🡪 JRE

* Multithreading
* High performance with **JIT Compiler**
* Automatic Memory Management (Garbage collection)

Installing from OpenJDK

Configure JAVA\_HOME env variable, and add %JAVA\_HOME%\bin to your path variable

To verify : java –version (on command prompt)

JDK – Development kit (Compiler + JRE)

Public class Hello {

Public static void main(String[] args) {

System.out.println(“Hello world!”);

}

}

Hello.java 🡪 hello.class 🡪 JVM (resides under JRE)

Consider this bytecode as a universal language between your code and your machine.

**Data types**:

Type of the Data a variable can hold.

Strictly type language – means every variable must have a declared type.A diagram of data types

AI-generated content may be incorrect.

1 byte = 8 bits

1024 byts – 1 MB

1024 MB - 1 GB

1024 GB – 1 TB

**Unicode** is universal international standard character encoding system that **is capable of representing most of the world’s languages**.

* ASCII – for US
* ISO 8859-1 : for western europeon languages
* KOI-8 : for Russian
* GB 18030 and BIG-5 – for chenese and so on.

|  |  |  |
| --- | --- | --- |
| boolean | 1 bit (logical) | Only two numbers : true or false; **boolean flag=true** |
| char | 2 bytes | Single character (Unicode) , char c = ‘A’ |
| byte | 1 bytes (8 bit) | Small intergers (-128 to 127); byte b = 100 |
| Short | 2 bytes (16 bit) | Small integers (-215 to 215-1) ; short s = 32000; |
| Int | 4 bytes | Standard interger typs, int x = 10000; |
| Long | 8 bytes | Large interger types |
| Float | 4 bytes | Decimal numbers (single precision), float f= 10.5f; |
| Double | 8 bytes | Decimal numbers (Double precision), double d = 20.99; |

* Primitive data types are built-in in java
* Non Premitive are User defined types

**Type casting:**

* **Widening Type casting:** Converting a lower data type into higher one is called widening type casting. Or Implicit type conversion . it is automatically done by java. It is safe, because there is no chance of loosingdata.

**Byte🡪short🡪char🡪int🡪long🡪float🡪double**

* **Narrowing type casting:** Converting higher data types into lower one. Explicit conversion.

Double🡪float🡪long🡪int🡪char🡪short🡪byte

**Operators:**

1. Arithmatic operators (addition,substraction, multiplication, division, modulus)
2. **Relational (Comparision) Operators**:
   * + == 🡪 **5==5**
     + != 🡪 **5!=3**
     + > 🡪 5>3
     + < 🡪 3<5
     + >=
     + <=
3. **Logical operators**
   * + && 🡪Logical AND 🡪 true && false - false
     + !
4. **Assignment operators**
   * + **= 🡪** Assign value , a=10
     + **+= 🡪** Add and assign , a+=5
     + **-= 🡪** Subtract and assign, a-=3
     + \*=
     + /=
     + %=
5. **Unary Operator**

**++ 🡪 increment**

1. Bitwise operators

&,^,~,<<,>>,>>>

1. Ternary Operator

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Phase 1** | **JFSD: Data Structures and Algorithms** | | |
| 1 | 6/7/2025 |  | 4 | Foundations of Data structures and Algorithms (Time and space complexity), Arrays, Multidimentional array , |
| 2 | 6/8/2025 |  | 4 | LinkedList, Operations on LinkedList, Stacks, Queue |
| 3 | 6/14/2025 |  | 4 | Trees Binary, AVL tree implemenation , Graphs and its types (Graph traversal implementation |
| 4 | 6/15/2025 |  | 4 | HashMap Bubble sort and Selection sort implementation Insertion sort, merge sort, quick sort implementation |
| 5 | 6/21/2025 |  | 4 | Heap sort, count sort,  Linear search, Binary Search, Jump Search algorithm |
| 6 | 6/22/2025 |  | 4 | radix sort implementation Phase end project |

Phase 1:

22/062025

**Heap Sort:**

Heap Sort is ,

1. Build a **Max Heap** from the input array
2. Repeatedly Swaps the max (root) element with the last element
3. **Shrinks the heap** and happifies again to maintain max heap.

**What is Max Heap?**

A Max heap is a binary tree where,

* Every parent is greater than or equal to its children
* Its stored in array
  + Parent at i
  + Left child at 2i+1
  + Right Child at 2i+2

Example: [4,10,3,5,1]

Heapify index 1 (value 10)

[10,4,3,5,1]

[10,5,3,4,1]

[**1,5,3,4**,10]

[5,4,3,1,10]

[**1,4,3**,5,10]

[4,1,3,5,10]

[3,1,**4,5,10**]

[3,1,4,5,10]

[1,3,4,5,10]

**Count Sort:**

Counting sort is **a non-comparison-based sorting algorithm**.

Conditions:

* You must have non-negative integers
* The range of values(max-min) is not very large

arr= [4,2,2,8,3,3,1]

**Step 1**: Find max value

Max = 8

**Step 2: Create count array**

* Indexes from 0 to max
* Initialize all with 0 value

count = [0,0,0,0,0,0,0,0,0] // 9 elements

**Step 3: Count occurrences**

Loop through input array and update counts;

**count = [0,1,2,2,1,0,0,0,1]**

**Step 4: Reconstruct the sorted array**

Loop through count[], for each I, place I in result array [count[i] times.

Sorted: [1,2,2,3,3,4,8]

------------------------------------------------------------------------------------------------------------------------------------------  
**21-June-2025**

-----------------------------------------------------------------------------------------------------------------------------------------

**Merge Sort:**

Merge Sort is a classic **Divide and Conquer** algorithm that works by,

1. Dividing the array into halves until each subarray has only one element
2. Merging those sorted halves back together in order.

[2] [1]

[1,2]

[38] [27,43]

[27, 38, 43]

~~[27,~~**~~38~~**~~,43]~~ ~~[3~~**~~,~~**~~9,10~~,**82, 99,100,102**]

[3, 9 , 10, 27, 38, 43, 82]

**Quick Sort**

Quick Sort is also a Divide and Conquer algorithm.

Unlike merge sort, which divides array in half, quicksort works by selecting a **“pivot”** element, then partitions the array so:

* All elements < pivot go to the left
* All elements > pivot go to the right

Then recursively softs the left and right subarrays.

Binary Search:

Idea: Cut the sorted array in half repeatedly until element is found.

* **Use when the array is sorted**
* You want fast performance (much faster than linear search)

Target=11

[1,3,5,7,9,11]

16

[1,3**,**5,7,9,11,13,23,25,28,40,45,46,53,57,67,**68**,69,72,**89**,100]

i=1

while((i<n) && arr[i]<target) {

i=i\*2;

}

I=32

------------------------------------------------------------------------------------------------------------------------------------------  
**15-June-2025**

-----------------------------------------------------------------------------------------------------------------------------------------

**HashMap**

A hashMap is a key-value pair datastructure

Fast insertion, searching and deletion operations

Uses Hash Function to map keys to indices in array (buckets)

Name – Dhruvik

Name – Mohit

Age= 25

Characteristics:

* Key-Value pairs –{ name – Dhruvik, }
* No Duplicate keys allowed
* Allow Null Values as key and value
* Unordered – Doesn’t maintain insertion order
* Efficient lookup – O(1)

------------------------------------------------------------------------------------------------------------------------------------------  
**14-June-2025**

-----------------------------------------------------------------------------------------------------------------------------------------

**What is a Non-Linear Data Structure?**

Data elements are not arranged sequentially (not in straight line). Instead, they form a hierarchy or relationship between elements (like tree or graph)

* Elements can have multiple relations
* Not stored in contiguous memory locations
* Traversal is complex (Compared to array or linked list)

Examples:

* Tree
* Graph
* Heap

Why Non-Linear?

* To manage relationships (e.g. file system, organization charts)
* For efficient search and insert operations

**Tree**

A Tree is a hierarchical, non-linear datastructure made up of nodes

* Top node – root
* Each node (except root) has one parent
* Nodes may have children
* There are no cycle

A level 0

B C level 1

C E F

Use case?

* File system hierarchy
* XML/HTML DOM
* Databases (B-Tree, B+Tree)
* Routing algorithms in networks

Advantage

* Dynamic insertion/deletions
* Fast searching with ordered tree
* Mirrors natural hierarchical relationships

Terminologies:

* Node – Basic unit of tree with data
* Root Node – The topmost node
* Parent Node – A node that has children
* Child node – Node descended from another node
* Leaf node – A node with no children
* Internal Node – A node with atleast one child
* Degree – Number of children of a node
* Level – Distance from root (root – level 0)
* Height of Node – Longest path from node to a leaf
* Depth of node – Path length from root to that node

Types of tree:

* Binary Tree
* BST Tree
* AVL Tree
* B Tree

**Binary Tree**

* Each node has at most two children, (0,1,2)
* Foundation of more advanced trees like Binary Search Trees, AVL Trees, Heaps etc

Class TreeNode {

int data;

TreeNode left;

TreeNode right;

}

**Types of Binary Trees**

1. **Full Binary Tree –** Every node has 0 or 2 children
2. **Perfect Binary Tree** 
   1. **All interior nodes have two children**
   2. **All leaves at the same level**

**1**

**2 3**

**4 5 6 7**

1. **Complete Binary Tree**

All levels are fully filled except last, and the last level is filled left to right.

1. **Skewed Binary Tree**

All nodes have only one child (either left or right)

**Binary Search Tree**

A Binary Search tree is binary tree with a special properties

* All nodes in the left subtree have values less than node’s value
* All the nodes in the right subtree have values greater than node’s value
* This rule applies recursively to all nodes

**AVL Tree**

An AVL Tree is self-balancing Binary Search Tree (BST) where the difference in heights between left and right subtree of any node is at most 1.

The height difference is called Balance Factor –

Balance Factor = Height(left\_subtree) – height(right\_subtree)

Valid values of BF - -1,0,1

How ?

LL – Left Left - Inserted into left of left

RR – Right Right

LR – Left Right

RL – Right Left

**B- Tree**

A B-Tree is a self-balancing search tree that is optimized for systems that read and write large blocks of data like datasets and file systems

Generalization of BST but allows a node to have more than two childrens

Fundamentals:

1. Multi-way tree
   * A B-Tree of order m, can have upto m children per node
   * Each node can hold multiple keys (upto m-1 keys) arranged in **sorted order**
2. Balanced Tree
   * The B-Tree is always balanced – all leafs nodes are at the same level
   * It keeps height small for fast access.
3. Sorted Keys
   * Keys in each node are kept in sorted order
   * Search is done using binary search within the node
4. Broad and shallow

**Tree Traversals**

1. Depth First Traversal (DFT)
   1. Inorder (Left -> root -> right)
   2. Preorder (Root -> left -> right)
   3. Postorder (Left -> right -> root)
2. Breadth First Traversal (BFT)

**Graph**

Graph is a non linear datastructure consisting of:

Vertices (or nodes) – Points

Edge (or arcs) – connection between Vertices

Real-world examples:

* Social network (people – nodes, relationships – edge)
* Maps (Cities – nodes, roads – edge)
* Internet (Web pages = nodes, hyperlinks- edge)

Treee and Graph?

|  |  |  |
| --- | --- | --- |
| Feature | Tree | Graph |
| Structure | Hierarchical | Network-like |
| Cycles | No cycles | May have cycle |
| Parent/Child | Defined | No such concept |
| Connectivity | One path between nodes | Multiple paths may exist |
| Edge Count | n-1 edges | Can have any number of edges |
| Root | Always has one root | No root required |
| Direction | Usually one direction | Can be directed/undirected |

|  |  |  |
| --- | --- | --- |
| Term | Meaning | |
| Vertex(Node) | | Fundamental unit of graph |
| Edge(Arc) | | Connection between two nodes |
| Adjacency | | Two nodes are adjacent if they are connected by an edge |
| Path | | A sequence of vertices connected by edges |
| Degree | | Number of edges connected to a node |
| Cycle | | A path that starts and ends at the same vertex |
|  | |  |

**Types of graphs**

1. **Based on Structure**
   1. Finite Graph
      1. Has finite number of nodes and edges
   2. Infinite Graph
      1. Theoretically contains infinite nodes/edges
      2. Not practical in memory, used in theoretical math
   3. Trivial Graph
      1. Has only one vertex and **no edges**
   4. Null Graph
      1. Contains n vertices and no edges
   5. Simple Graph
      1. No Self-loops or multiple edges
   6. Multi Graph
      1. Has multiple edges (parallel edges) between two nodes
   7. Psuedo Graph
      1. Contains self loops (an edge from one node to itself)
2. **Based on Connectivity**
   1. Connected Graph
      1. There is a path between every pair of vertices (e.g. social network where everyone is somehow connected)
   2. Disconnected graph
      1. Some Nodes are isolated (no connection to rest)
3. **Based on Direction and Weight**
   1. Directed Graph (Digraph)
      * Edges have direction(A -> B is not B-> A)
   2. Undirected Graph
      * Edges do not have direction (A- B)
   3. Weighted Graph
      * Each edge has weight or cost (distance, time)
   4. Unweighted Graph
      * All edges are treated equally
4. **Based on cycles**
   1. Cyclic Graph
      * Contains at least one cycle (e.g. A->B->C->A)
   2. Acyclic Graph
      * No cycles present
   3. Directed Acyclic Graph (DAG)
      * Directed and acyclic
      * Scheduling, Orchastrator, Building systems
5. **Others**
   1. Complete Graph
      * Every node is connected to every other node
   2. Regular Graph
      * All vertices have same degree
   3. Sub Graph
      * A subset of the original graph’s vertices and edges

**Graph Representations:**

**Adjacency matrix**

2D matrix – adj[i][j] = 1, if there’s any edge from vertex i to j

Space: O(V2)

A B C

A-B B-C

|  |  |  |  |
| --- | --- | --- | --- |
|  | **A** | **B** | **C** |
| **A** | **0** | **1** | **0** |
| **B** | **1** | **0** | **1** |
| **C** | **0** | **1** | **0** |

**Pros:**

Fast edge lookup : O(1)

**Cons:**

Space inefficient for sparse graphs

**B. Ajdacency List**

- Each vertex stores list of adjacent vertices

A -> B

B ->A, C

C -> B

Pros:

* Efficient for spase graphs
* Easy to integrate neighbors

Cons:

* Slower edge lookup

------------------------------------------------------------------------------------------------------------------------------------------  
**08-June-2025**

-----------------------------------------------------------------------------------------------------------------------------------------

**Assignment**:

For given array return transpose array.

Transpose Array

1 2 3

4 5 6

1. 4
2. 5
3. 6

3 X2 result

Multi-Dimensional Arrays - Two Dimensional Arrays

[2,3,4,5]

[ [1,2], [3,4] , [5,6] ] -2D array

1 2

3 4

5 6

3 X2 array

Int[][] arr = new int[3][3];

Int[][] arr2 = {{1,2}, {3,4}, {5,6}}

1 2 5 6

3 4 7 8

6 8

10 12

**Linked List**

A linked list is a linear data structure where elements are stored in nodes.

Each node has data and a pointer to the next node.

Why Linked List?

* Dynamic size
* Efficient Insertion and Deletion operations

Types of linked lists:

* Singly Linked List - Each node points to the next
* Doubly Linked List - Each node points to the both next and previous nodes
* Circular Linked List - Last node points to the first node

1,2,3

1,2,3,4

|  |  |  |
| --- | --- | --- |
| Feature | Array | LinkedList |
| Size | Fixed (Static) | Dynamic (Grows/s |
| Insert/Delete | Costly (shift elements) | Fast (change pointers) |
| Access time | Fast (random access) | Slower (sequential access) |
| Memory Allocation | Contiguous | Non-Contigous |

**Stack:**

A stack is a linear data structure that follows the LIFO (Last In, First Out) principle.

* You can add (push) and remove (pop) elements from the top of the stack
* Pile of plates 🡪

**Queue:**

* A queue is linear data structure that follow the FIFO (First In, First Out) Principle
  + Elements are added at the rear (enqueue) and removed from the front(dequeue).
  + Think of it as a line at a ticket counter.

------------------------------------------------------------------------------------------------------------------------------------------  
**07-June-2025**

-----------------------------------------------------------------------------------------------------------------------------------------

Datastructure –

Way of organizing and storing data so that it can be accessed and modified efficiently.

**Commons operations in Data structures:**

* Insertion
* Deletion
* Searching
* Traversal
* Sorting

**Examples:**

* **Array** – Fixed size collection of elements of the same type
* **Linked List** – Collection of nodes connected by references
* **Stacks** – LIFO (Last In First Out) data structure
* **Queues** – FIFO (First in First Out) data structure
* **Trees** – Hierarchical data structure
* **Graphs**: Set of nodes connected by edges

**DataStructures vs Data Types**

**Data Type:** Type of variable

Int, float

**Why need for data structure?**

* Efficient data storage and access
* Reduce Time Complexity of operations
* Better organization for complex data
* Essential in system design and scalable applications

**“**Alice”, 23, BEIT, BOB,24, MCA

[

{name: Alice, age: 23, course: BEIT},

{name: Bob, age: 24, course: MCA}

]

Classification of Data structures:

**Linear Data structures**:

Arrays, Linked Lists, Stack, Queues

[1] 🡪 [2] 🡪 [3] 🡪 [4]

**Non-Linear Data structures:**

Trees, Graphs

1

**Algo Characteristics:**

* Takes input
* Gives Output
* Finiteness
* Effectiveness

**Assignment**:

1. Write a program to find the sum of all elements in a integer array

[2,4,6,8] 🡪 **output**: 20

1. Write program to reverse the elements of an array:

[1,2,3,4] 🡪 **Output**: [4,3,2,1]

1. Find maximum and Minimum in an array

Input: [5,8,3,1,9] 🡪 **Output** : Max: 9, Min: 1

1. Count even and Odd numbers in an array

**Input**: [2,3,5,6,8] 🡪 **Output**: Even : 3, Odd: 2

1. Print all elements at even indexes:

**Input**: [10,20,30,40,50] 🡪 **Output**: 10 30 50

**Time Complexity**

Amount of time taken by algorithm to run.

Time and Space complexity can define effectiveness of an algorithm.

**Big O Notation** – mathematical way to describe the time complexity of an algorithm in terms of input size n. It tells you how the performance of your algorithm scales as the input grows.

Compare different algorithms

**Different types of Time Complexities:**

1. Constant Time – O(1) - Time doesn’t grow with input size
2. Linear Time – O(n) - Grows proportionally with the input size
3. Logarithmic Time : O(log n) - Cuts input in half each time
4. Quadratic Time: O(n^2) - Slower for large input
5. Cubic Time – O(n^3) -
6. Exponential Time - O(2^n) - Very slow, doubles time each step
7. Factorial Time – O(n!) - Extremely slow even for small n

What can cause time in function?

1. Operations (+,-….)
2. Comparisons (>,<,==)
3. Looping (for, while)

Rules:

1. Always consider worst case
2. Remove constants
3. Remove non dominant terms

Iteration: x

1. x
2. x/2
3. x/4
4. x/8

….

K x/2^k

2^k = x

K = log2(x)

**Space Complexity**

Space complexity is the total memory used by an algorithm as a function of input size n.

It includes:

* Memory used by input data
* Memory used by variables
* Memory used by recursion stack or function calls
* Memory used by auxiliary data structures (like arrays, hashmaps exc)