CORE JAVA:

Q1: What is the Java String Pool?

A: The **Java String Pool** (also called the **String Intern Pool**) is a special memory region inside the Java heap where **string literals** are stored.

- When you create a string using double quotes (e.g., "Hello"), Java checks the pool first.
- If the string already exists in the pool, the same reference is returned instead of creating a new object.

Q2: Why does the main method in Java have String[] args and what is its purpose?
A: The String[] args lets us pass **command-line arguments** to a Java program. Each value typed after the program name is stored in this array, so the program can use external inputs at runtime. For example, running java Test Hello World makes args[0] = "Hello" and args[1] = "World".

👉 Short takeaway: *String[] args is used to capture command-line inputs so programs can accept external values at runtime.*

Q3: How is a Java program executed?
A: First, the Java source code is compiled by javac into bytecode. The JVM then loads the .class file, verifies it, and executes the bytecode using an interpreter or JIT compiler. Finally, the program runs with JVM support like garbage collection, memory management, and thread scheduling.

👉 Short takeaway: *Java code → compiled to bytecode → loaded by JVM → executed via interpreter/JIT → runs on any platform.*

Q4: What is the difference between String, StringBuilder, and StringBuffer in Java?
A:

- **String** → Immutable. Any change creates a new object, so less efficient for frequent modifications.
- **StringBuilder** → Mutable and faster, but not thread-safe. Best for single-threaded use.
- **StringBuffer** → Mutable and thread-safe because methods are synchronized. Slightly slower but safe in multi-threaded environments.

Q5: What is the difference between an abstract class and an interface in Java?
A:

- Abstract class can have both abstract and concrete methods, variables, and constructors.

- Interface mainly defines abstract methods (with default/static allowed since Java 8).
- A class can extend only one abstract class, but can implement multiple interfaces.

Q6: Can we overload the main method in Java?
A: Yes, we can overload the main method by changing its parameter list (e.g., main(String args[]), main(int x), main(double d, String s)). However, only the standard signature public static void main(String[] args) is used by the JVM as the entry point of the program. The overloaded versions will not be called automatically; they can only be invoked explicitly from within the standard main method or other methods.

👉 Short and clear: *Yes, overloading is possible, but only the standard main(String[] args) is recognized by JVM as the starting point*.

Q7: What is a Singleton class in Java and how do we create it?
A: A Singleton class ensures only one instance exists in the JVM, reusing the same object across the application. It's useful for logging, configuration, cache, or database connections. To create it:

- Make the constructor **private** to prevent direct instantiation.
- Define a **static instance** variable to hold the single object.
- Provide a **public static method** that returns the instance.

```
class Singleton {
  private static Singleton instance;   // static instance
  private Singleton() {}           // private constructor

  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();  // lazy initialization
    }
    return instance;
  }
}
```

```
public class Test {
   public static void main(String[] args) {
      Singleton obj1 = Singleton.getInstance();
      Singleton obj2 = Singleton.getInstance();
      System.out.println(obj1 == obj2); // true → same instance
   }
}
```

Q8: Explain the internal working of a HashMap in Java.
A: A HashMap stores data as key-value pairs using an array of buckets. Each key's hashCode() is processed to find the bucket index. If multiple keys map to the same bucket (collision), entries are stored in a linked list (Java 7) or a balanced tree (Java 8+ when collisions are high). Lookup, insert, and delete operations are generally **O(1)**, but can degrade to **O(log n)** in case of many collisions.

Q9: What is the difference between HashMap, Hashtable, and ConcurrentHashMap?
A:

- **HashMap** → Non-synchronized, allows one null key and multiple null values, faster in single-threaded use.
- **Hashtable** → Synchronized, thread-safe but slower, does not allow null keys or values.
- **ConcurrentHashMap** → Thread-safe with better performance than Hashtable, uses advanced locking, and does not allow null keys or values.

Q10: What happens if two keys have the same hashcode in HashMap?
A: If two keys have the same hashcode, they go into the same bucket. HashMap then uses equals() to distinguish them. If equals() returns true, the new value replaces the old one; if false, both entries coexist in the bucket (linked list/tree).

Q11: What is a Marker Interface in Java?
A: A marker interface is an empty interface (no methods/fields) used to tag a class so JVM or frameworks know it has special behavior. For example, Serializable is a marker interface; when a class implements it, objects of that class can be converted into a byte stream for storage or transfer. Similarly, Cloneable marks classes whose objects can be cloned.

👉 Short takeaway: *Marker interfaces are empty interfaces used to signal special behavior to JVM or frameworks.*

Q12: Can we override .equals() in Java? How?
A: Yes. We override .equals() to define logical equality for our class. It must follow the **contract**: reflexive, symmetric, transitive, consistent, and return false for null. When overriding .equals(), we should also override hashCode() to maintain consistency in collections like HashMap and HashSet.

Q13: Difference between Checked and Unchecked exceptions?
A:

- **Checked** → Compile-time checked, must be handled or declared (e.g., IOException, SQLException).
- **Unchecked** → Runtime exceptions, not forced to handle (e.g., NullPointerException, ArrayIndexOutOfBoundsException).

  👉 Checked = recoverable, Unchecked = programming errors

Q14: What is try-with-resources in Java?
A: It's a feature (Java 7+) that automatically closes resources like files, streams, or DB connections once the try block finishes. Any class implementing AutoCloseable can be used. This avoids boilerplate finally blocks and prevents resource leaks.

MULTITHREADING:

Q1: Explain the thread lifecycle in Java.
A: A thread in Java passes through several states defined in java.lang.Thread.State.

- **New (Created):** When a thread object is created but start() has not been called yet.
- **Runnable:** After start() is called, the thread is ready to run and waiting for CPU scheduling.
- **Running:** When the thread scheduler picks it, the thread's run() method executes.
- **Blocked:** The thread is waiting to acquire a lock to enter a synchronized block/method.
- **Waiting:** The thread waits indefinitely until another thread signals it (using notify()/notifyAll()).
- **Timed Waiting:** The thread waits for a specified time (e.g., sleep(ms), join(ms), wait(ms)).

- **Terminated (Dead):** Once the run() method finishes, the thread is considered dead and cannot be restarted.

Q2: What is the purpose of volatile in Java?
A: The volatile keyword ensures that changes to a variable are always visible to all threads. It forces reads/writes to happen directly from main memory instead of thread-local caches. This guarantees **visibility** but not **atomicity**, so it's useful for flags or simple shared variables, not for compound operations like count++.

👉 Short takeaway: *volatile ensures visibility of variable changes across threads, but does not make operations atomic.*

JAVA 8:

Q1: What are the main features of Java 8?
A: Java 8 introduced lambda expressions, functional interfaces, Streams API, Optional, new Date/Time API, and default/static methods in interfaces. These features brought functional programming style and improved code readability and maintainability.

Q2: What is a Lambda expression?
A: A lambda is a short block of code that takes parameters and returns a value, used to represent anonymous functions. It reduces boilerplate and enables functional programming, e.g. (x, y) -> x + y.

Q3: What is a Functional Interface?
A: It's an interface with exactly one abstract method (SAM), annotated with @FunctionalInterface. Examples include Runnable and Comparator. They are the foundation for lambdas and method references.

Q4: What is the Streams API?
A: Streams allow processing collections declaratively with operations like map, filter, and reduce. They support parallel execution and make data processing concise and efficient compared to traditional loops

Q5: Difference between map() and flatMap()?
A: map() transforms each element into another object, while flatMap() flattens nested structures. For example, flatMap() is used to convert List<List> into a single List.

Q6: What are Default methods in interfaces?
A: Default methods let you add new functionality to interfaces without breaking existing implementations. They provide backward compatibility and allow interfaces to evolve more flexibly

Q7: What is the Optional class?
A: Optional is a container object to handle null values safely. It provides methods like isPresent(), orElse(), and ifPresent() to avoid NullPointerException and write cleaner code.

Q8: Explain the new Date/Time API.
A: Java 8 introduced LocalDate, LocalTime, LocalDateTime, and ZonedDateTime. These classes are immutable, thread-safe, and much more user-friendly than the old Date and Calendar.

Q9: What is forEach() in Java 8?
A: forEach() is a method in Iterable and Stream that lets you iterate elements using lambdas. Example: list.forEach(System.out::println); makes iteration concise and expressive.

Q10: How does Java 8 support parallelism?
A: Streams can be executed in parallel using parallelStream(). This leverages multi-core processors to process large datasets faster, improving performance in data-intensive applications.

Q11: What are Intermediate operations in Streams?
A: Operations that transform data and return a Stream (lazy). Examples: map(), filter(), flatMap(), distinct(), sorted(), limit(), skip().

Q12: What are Terminal operations in Streams?
A: Operations that produce a result and end the Stream. Examples: forEach(), collect(), reduce(), count(), min(), max(), findFirst().

Q13: What are Short-circuit operations in Streams?
A: Special operations that stop early once condition/result is found. Examples: findFirst(), findAny(), anyMatch(), allMatch(), noneMatch(), limit().

SPRINGBOOT:

Q1: How do we implement caching?
A: We store frequently used data in memory to avoid repeated DB calls. In Spring, we enable caching with @EnableCaching and use annotations like @Cacheable (store),

@CacheEvict (remove), and @CachePut (update). For microservices, we usually back this with Redis or Memcached.

Q2: If a Spring Boot application is experiencing performance issues, what will you do?
A:

- **Check logs & metrics** → Identify slow endpoints, DB queries, or memory issues.
- **Profile the app** → Use tools like JProfiler, VisualVM, or Spring Actuator.
- **Optimize DB calls** → Add indexes, reduce joins, use caching (@Cacheable, Redis).
- **Tune JVM & threads** → Adjust heap size, thread pool configs.
- **Enable async/non-blocking** → Use WebFlux or async calls for heavy I/O.
- **Monitor infra** → Check CPU, memory, network bottlenecks.

Q3: How do we handle versioning in REST APIs?
A: Versioning ensures backward compatibility when APIs evolve. Common approaches:

- **URI versioning:** GET /api/v1/users → simplest and most widely used.
- **Query parameter:** GET /api/users?version=1 → less common.
- **Header versioning:** Custom header like Accept: application/vnd.company.v1+json.
- **Content negotiation:** Use Accept header with media type versioning.

Q4: What is @SpringBootApplication in Spring Boot?
A: It is a **meta-annotation** that marks the main class of a Spring Boot app. It combines three annotations:

- @Configuration → allows defining beans.
- @EnableAutoConfiguration → enables auto-config based on classpath dependencies.
- @ComponentScan → scans the package and sub-packages for Spring components.

Q5: What do Spring Boot Actuator endpoints do and how do we secure them?
A: Actuator provides endpoints like /actuator/health, /actuator/info, /actuator/metrics, etc. to monitor and manage the app (health, metrics, environment, logs). To secure them, we expose only required endpoints via

management.endpoints.web.exposure.include, protect with **Spring Security** (role-based access), and restrict sensitive endpoints using authentication, IP whitelisting, or disabling them.

Q6: How do we handle multiple beans of the same type in Spring?
A: When multiple beans of the same type exist, Spring needs to know which one to inject. We handle this by:

- **@Primary** → Marks one bean as the default choice.
- **@Qualifier** → Used with @Autowired to specify the exact bean by name.
- **Bean name injection** → Inject by explicitly referencing the bean name.

👉 Short takeaway: *Use @Primary for default bean, @Qualifier for explicit selection*

Q7: What are best practices for managing transactions in Spring Boot?
A:

- Use **@Transactional** annotation at the **service layer** (not controller) to keep business logic atomic.
- Keep transactions **short-lived** → avoid long operations inside them (like remote calls).
- Use **propagation settings** (REQUIRED, REQUIRES_NEW) carefully to control nested transactions.
- Handle **rollback rules** explicitly (rollbackFor = Exception.class) when needed.
- Prefer **declarative transactions** (@Transactional) over programmatic for cleaner code.

👉 Short takeaway: *Use @Transactional at service layer, keep transactions short, configure propagation/rollback wisely, and use saga for microservices*

Q8: How do you approach testing a Spring Boot application?
A:

- **Unit Testing:** Test individual classes/methods with **JUnit** + **Mockito** (mock dependencies).
- **Integration Testing:** Use **@SpringBootTest** to load context and test interaction with DB, REST, etc.
- **Slice Testing:** Use annotations like @WebMvcTest, @DataJpaTest to test only specific layers.
- **Test Data:** Use **Testcontainers** or in-memory DB (H2) for reliable integration tests.

- **REST APIs:** Test endpoints with **MockMvc** or **WebTestClient**.
- **Automation:** Integrate tests into CI/CD pipeline for continuous validation.

Q9: How do we test in Spring Boot using Mockito?
A: Mockito is used to mock dependencies so we can test a class in isolation.

- Annotate test class with @ExtendWith(MockitoExtension.class) (JUnit 5).
- Use @Mock to create mock objects.
- Use @InjectMocks to inject those mocks into the class under test.
- Define behavior with when(...).thenReturn(...).
- Verify interactions with verify(...).

Q10: What are the merits and demerits of application.yml vs application.properties?
A:

- **YAML merits:** Hierarchical, readable, supports lists/maps, less repetition.
- **YAML demerits:** Indentation sensitive, harder to debug.
- **Properties merits:** Simple flat format, easy, less error-prone.
- **Properties demerits:** Repetitive, poor for complex configs

Q11: What is profiling in Spring Boot and how is it used?
A: Profiling in Spring Boot means running the app with different configurations for different environments (dev, test, prod). We use **Spring Profiles** to activate specific beans or properties.

- Define environment-specific configs in application-dev.yml, application-prod.yml.
- Annotate beans with @Profile("dev") or @Profile("prod").
- Activate a profile via spring.profiles.active=dev in properties/YAML or command line.

Example:

@Profile("dev") @Bean public DataSource devDataSource() { ... } @Profile("prod") @Bean public DataSource prodDataSource() { ... }

👉 Short takeaway: *Profiles let us switch configs/beans per environment (dev, test, prod) easily.*

Q12: What is AOP in Spring Boot and how do we implement it?
A: Aspect Oriented Programming (AOP) separates cross-cutting concerns like logging, security, and transactions from business logic. In Spring Boot, we implement AOP by:

- Defining an **Aspect** class with @Aspect and @Component.
- Writing **Advice** (@Before, @After, @Around) methods to run at specific points.
- Using **Pointcut expressions** (execution(* com.example.service.*.*(..))) to target methods.
- Spring weaves these aspects into the app at runtime, applying the advice transparently.

Q13: What is authentication and authorization in Spring Boot?
A:

- **Authentication** → Verifying *who* the user is (identity check). In Spring Boot, handled via **Spring Security** with login forms, JWT tokens, OAuth2, etc.
- **Authorization** → Deciding *what* the authenticated user can access (permissions/roles). In Spring Boot, enforced via role-based access (@PreAuthorize, @Secured, hasRole() expressions).

Q14: How do we implement pagination in Spring Boot?
A: Pagination is handled using **Spring Data JPA's Pageable and Page**.

- Extend JpaRepository → it supports pagination by default.
- Pass a Pageable object (PageRequest.of(page, size)) to repository methods.
- The result is a Page object with data + metadata (total pages, elements).
- Sorting can be added via PageRequest.of(page, size, Sort.by("field")).

Q15: How do you handle exceptions in Spring Boot?
A:

- Use **@ControllerAdvice + @ExceptionHandler** → Centralized handling of exceptions across controllers.
- Use **ResponseEntityExceptionHandler** → Extend this class to customize REST error responses.
- For validation errors → use **@Valid + @ExceptionHandler(MethodArgumentNotValidException)**.

- For global fallback → define a **custom error response DTO** with status, message, timestamp.
- Optionally use **@RestControllerAdvice** for REST APIs to return JSON error responses.

👉 Short takeaway: *Spring Boot exceptions are handled centrally with @ControllerAdvice and @ExceptionHandler for clean, consistent error responses.*

MICROSERVICES:

Q1: How do microservices communicate with each other?
A: They communicate either **synchronously** (REST APIs, gRPC, FeignClient) or **asynchronously** (message brokers like Kafka, RabbitMQ). Choice depends on whether immediate response or decoupled event-driven flow is needed

Q2: What is the difference between synchronous and asynchronous communication in microservices?
A:

- **Synchronous:** Request/response, caller waits (e.g., REST call to Payment Service). Simple but tightly coupled.
- **Asynchronous:** Event/message based, caller doesn't wait (e.g., Order Service publishes event to Kafka). Loosely coupled, scalable, but eventually consistent.

Q3: What is Spring Cloud and how is it useful?
A: Spring Cloud is a set of tools built on top of Spring Boot for developing **distributed systems and microservices**. It provides solutions for common challenges in microservice architecture such as:

- **Service Discovery** → using Netflix Eureka or Consul to locate services dynamically.
- **Configuration Management** → centralized configs via Spring Cloud Config Server.
- **Load Balancing & Routing** → Ribbon, Spring Cloud Gateway for smart routing.
- **Fault Tolerance** → Hystrix/Resilience4j for circuit breakers and retries.
- **Distributed Tracing** → Sleuth + Zipkin for monitoring requests across services.
- **Security & Communication** → OAuth2, Feign clients for secure inter-service calls.

👉 **Usefulness:** It simplifies building scalable, resilient microservices by handling infrastructure concerns (discovery, config, routing, monitoring) so developers can focus on business logic.

Q4: How do we create a Dockerfile for a Spring Boot app?
A:

1. **Build JAR** → mvn clean package
2. **Create Dockerfile** with steps:

- FROM openjdk:17-jdk → base image
- WORKDIR /app → set working dir
- COPY target/app.jar app.jar → copy JAR
- EXPOSE 8080 → expose port
- ENTRYPOINT ["java","-jar","app.jar"] → run app

SQL & JPA:

Q1: Find the second highest salary from Employee table.

SELECT MAX(salary) FROM Employee

WHERE salary < (SELECT MAX(salary) FROM Employee);

SELECT DISTINCT salary FROM Employee

ORDER BY salary DESC LIMIT 1 OFFSET 1;

Q2: Find the nth highest salary (say 3rd).

SELECT DISTINCT salary FROM Employee e1

WHERE 3 = (SELECT COUNT(DISTINCT salary) FROM Employee e2

WHERE e2.salary >= e1.salary);

Q3: What is an index in SQL and why do we use it?
A: An index is a database structure that speeds up data retrieval by avoiding full table scans. It's created on columns used frequently in WHERE, JOIN, or ORDER BY clauses.
Example: CREATE INDEX idx_employee_name ON Employee(name);

Q4: What is JPA and Spring Data JPA, and how are they used in Spring Boot?
A:

- **JPA (Java Persistence API)** → A specification for ORM (Object-Relational Mapping) to manage relational data in Java apps. It defines how to map Java objects to database tables.
- **Spring Data JPA** → A Spring framework module built on top of JPA/Hibernate. It simplifies data access by providing ready-made repository interfaces (JpaRepository, CrudRepository) with built-in CRUD, pagination, and query methods.

👉 Short takeaway: *JPA defines ORM, Spring Data JPA implements it with easy repositories, and Spring Boot auto-configures everything for fast DB access*

Q5: What are the key annotations in JPA?
A:

- @Entity → Marks a class as a JPA entity (mapped to a DB table).
- @Table → Specifies table name and details.
- @Id → Marks primary key field.
- @GeneratedValue → Defines auto-generation strategy for primary key.
- @Column → Maps field to a specific DB column.
- @Transient → Field not persisted in DB.
- @OneToOne, @OneToMany, @ManyToOne, @ManyToMany → Define relationships between entities.
- @JoinColumn → Specifies foreign key column for relationships.
- @Embedded / @Embeddable → For composite/embedded objects.

Q6: What is JPA Repository and how do we write custom queries in it?
A:

- **JPA Repository** → It's part of Spring Data JPA, extending CrudRepository and PagingAndSortingRepository. It provides built-in CRUD, pagination, and sorting methods without writing SQL.
- **Custom Queries** → You can define queries in two ways:

  Derived Query Methods → Spring generates queries from method names.

  List<User> findByName(String name);

List<User> findByAgeGreaterThan(int age);

@Query Annotation → Write JPQL or native SQL directly.

@Query("SELECT u FROM User u WHERE u.email = ?1")

User findByEmail(String email);

@Query(value="SELECT * FROM users WHERE status=?1", nativeQuery=true)

List<User> findByStatus(String status);

Q7: What is the difference between Lazy and Eager loading in JPA?
A:

- **Lazy Loading** → Data is fetched **only when accessed**. Proxy objects are created, and actual DB query runs when you call the getter. Saves memory and improves performance for large relationships.
  Example: @OneToMany(fetch = FetchType.LAZY)
- **Eager Loading** → Data is fetched **immediately with the parent entity**. All related entities are loaded in a single query (or joins). Useful when you always need related data.
  Example: @OneToMany(fetch = FetchType.EAGER)

👉 Short takeaway: *Lazy = load on demand, Eager = load upfront*

Q8: What is the N+1 problem in JPA/Hibernate and why is it an issue?
A: The N+1 problem happens when fetching parent entities triggers **1 query for the parent** and then **N separate queries for each child entity**. For example, fetching 10 users with orders → 1 query for users + 10 queries for orders = 11 queries total.

This causes **performance bottlenecks** due to excessive queries and database round trips, especially with large datasets.

Solution: Use JOIN FETCH in JPQL, @EntityGraph, or batch fetching to load parent + child in a single optimized query.

👉 Short takeaway: *N+1 = one query for parent + N queries for children → fix with JOIN FETCH or entity graphs.*

Q9: What is cascading in JPA/Hibernate and what are its main types?
A: Cascading means parent operations (persist, merge, remove) automatically apply to child entities. Main types: PERSIST, MERGE, REMOVE, REFRESH, DETACH, and ALL.

Q10: What is EntityManager in JPA?
A: EntityManager is the core JPA interface used for **manual database operations**. It manages the persistence context and provides APIs for CRUD (persist, find, merge, remove) and query execution. Typical use case → building **custom repositories** where fine-grained control is needed beyond JpaRepository

REACT

Q: What is React?
A: React is an open-source JavaScript library by Meta used to build fast, reusable, component-based UIs for Single Page Applications (SPAs), powered by the Virtual DOM for efficient rendering

Q: What is a Single Page Application (SPA)?
A: An SPA is a web application that loads a single HTML page and dynamically updates content using JavaScript and APIs without full page reloads. It provides a fast, seamless user experience similar to desktop/mobile apps, but comes with trade-offs like SEO challenges and heavier initial load.

👉 Short takeaway: *SPA = one-page, dynamic updates, smooth UX, but SEO complexity.*

Q: What are the advantages of React?
A:

- **Reusable Components** → Build modular UI pieces that reduce duplication and speed up development.
- **Virtual DOM Efficiency** → Updates only changed parts of the DOM, improving performance.
- **Fast Rendering** → Diffing algorithm ensures minimal updates for smooth UI.
- **Flexibility** → Works for web (React.js) and mobile (React Native), integrates with other libraries.
- **Strong Community & Ecosystem** → Rich set of tools, libraries, and support.
- **SEO Support** → Server-side rendering makes SPAs more indexable

Q: What is the difference between Angular and React?
A:

- **Type:** Angular is a **framework**; React is a **library**.
- **Developer:** Angular → Google; React → Meta (Facebook).
- **Language:** Angular uses **TypeScript**; React uses **JavaScript + JSX**.

- **Architecture:** Angular follows **MVC/MVVM**; React focuses only on the **View layer**.

Q: What is the difference between DOM and Virtual DOM?
A: The **DOM (Document Object Model)** is the browser's tree structure representing an HTML page, where direct updates can be slow for large apps. The **Virtual DOM** is a lightweight, in-memory copy used by React to compare changes with a diffing algorithm and update only the affected parts of the real DOM, making rendering faster and more efficient.

👉 Short takeaway: *DOM = actual page structure; Virtual DOM = optimized copy for efficient updates*

Q: What is a component in React?
A: A component is a **reusable, independent UI block** in React, defined as a function or class that returns JSX. It manages its own state, accepts props, and helps build scalable, modular applications.

👉 Short takeaway: *Component = modular, reusable UI unit in React*

Q: What are the step-by-step commands to set up a React project (CRA and Vite)?
A: **Prerequisite:** Verify Node.js (node -v) and npm (npm -v) installation

Option A: Create React App (CRA)

- npx create-react-app my-app
- cd my-app
- npm start
- npm run build

Option B: Vite (faster modern setup)

- npm create vite@latest my-app
- cd my-app
- npm install
- npm run dev
- npm run build

👉 Short takeaway: run CRA or Vite scaffold → enter folder → install (Vite) → start dev → build when ready.

Q: What is npm and what is the use of the node_modules folder?
A: npm (Node Package Manager) is the default package manager for Node.js, used to install, update, and manage JavaScript libraries and dependencies. The **node_modules**

folder is where npm stores all the installed packages and their dependencies for a project, allowing your code to import and use them locally.

👉 Short takeaway: *npm = package manager; node_modules = storage for installed libraries*.

Q: What is the use of the public and src folders in React?
A:

- **public folder** → Stores **static assets** (like index.html, images, icons) that are served directly without processing by Webpack.
- **src folder** → Contains the **actual React code** (components, App.js, index.js, styles, logic) that gets compiled and bundled by Webpack into the final app.

👉 Short takeaway: *public = raw static files; src = core React codebase*

Q: What is index.html in React?
A: The index.html file (inside the public folder) is the **single HTML template** for a React app. It contains a

element where React injects and renders the entire component tree. This file also holds metadata, favicons, and links to static assets, but the dynamic UI is controlled by React through JavaScript.

👉 Short takeaway: *index.html = base HTML file with a root div where React mounts the app*.

Q: What is the role of App.js in React?
A: App.js is the **main component** of a React application. It acts as the **root container** where other components are imported and combined. Typically, it defines the overall structure of the UI, manages state or props passed to child components, and serves as the entry point for rendering inside the index.js file.

👉 Short takeaway: *App.js = root component that organizes and renders the app's UI*.

Q: What is the role of function and return in App.js?
A:

- **Function** → In App.js, the function (e.g., function App() { … }) defines a **React functional component**. It contains the logic and structure of how the component should behave and render.
- **Return** → Inside that function, the return statement outputs **JSX** (HTML-like code). This JSX describes what UI should be displayed on the screen.

👉 Short takeaway: *Function = defines the component; Return = renders the UI via JSX*.

Q: Can we have a function without return in React?
A: A **React component function must return JSX (or null)** because React needs something to render on the screen. Without return, the component won't display anything. However, you can still have **regular helper functions inside React files** (like in App.js) that don't return anything — they're used for side effects such as logging or calculations.

👉 Short takeaway: *Component functions → must return JSX/null; Helper functions → can skip return if only used for side effects*

Q: What is the role of export default in App.js?
A: export default makes the App component the **default export** of the file, so it can be imported elsewhere without curly braces. In React, index.js uses this to import App and render it inside the root div.

👉 Short takeaway: *export default = allows App to be imported easily as the main component.*

Q: What is the role of index.js, ReactDOM, and render in React?
A:

- **index.js** → Entry point of the React app; it connects React code to the browser DOM.
- **ReactDOM** → Acts as the bridge, converting the virtual React component tree into the real DOM.
- **render (or createRoot().render in React 18)** → Mounts the root component (usually App) into the

inside index.html.

👉 Short takeaway: *index.js = entry file, ReactDOM = bridge to real DOM, render = mounts React app into root div.*

Q: How does a React app load and display components in the browser?
A:

1. **Browser loads index.html** → contains

.

2. **index.js runs** → imports App.js and uses ReactDOM.createRoot().render() to mount it into root.
3. **ReactDOM bridges Virtual DOM → Real DOM** → converts React components into actual HTML elements.

4. **App.js executes** → returns JSX, which defines the UI.
5. **Components inside App.js** → are rendered recursively, building the full UI tree.
6. **Browser displays final DOM** → React keeps syncing changes via Virtual DOM for updates.

👉 Short takeaway: *index.html root div → index.js mounts App → ReactDOM renders JSX → browser shows components.*

Q: What is JSX in React?
A: JSX (JavaScript XML) is a **syntax extension for JavaScript** that looks like HTML but is compiled into React.createElement() calls. It allows developers to write UI code directly inside JavaScript, mixing markup with logic.

👉 Short takeaway: *JSX = HTML-like syntax inside JS, converted into React elements for rendering.*

Example:

```
const element = <h1>Hello, world!</h1>; // JSX

// Compiles to:

const element = React.createElement("h1", {}, "Hello, world!");
```

Q: What are the advantages of using JSX in React?
A:

- **Readability** → JSX looks like HTML, making component structure easy to understand.
- **Declarative UI** → Clearly shows what the UI should look like.
- **Integration with JS** → Allows embedding JavaScript expressions directly inside markup.
- **Cleaner Code** → Less verbose than React.createElement() calls.
- **Component-based** → Naturally supports React's reusable component design.
- **Debugging Ease** → Errors point directly to the JSX code, simplifying fixes.

👉 Short takeaway: *JSX makes React code more readable, declarative, and dynamic by combining HTML-like syntax with JavaScript logic.*

Q: Can a browser read JSX directly? What is Babel?
A:

- **Browser & JSX** → No, browsers cannot understand JSX directly because it's not valid JavaScript. JSX is a syntax extension that must be converted into plain JavaScript before the browser can execute it.
- **Babel** → Babel is a JavaScript compiler that **transforms JSX into standard JavaScript** (e.g., React.createElement() calls). It also transpiles modern ES6+ features into code that older browsers can understand.

👉 Short takeaway: *Browser can't read JSX → Babel converts JSX + modern JS into plain JavaScript the browser can run.*

Q: What is a transpiler?
A: A transpiler (source-to-source compiler) is a tool that converts code written in one high-level language or version into another high-level language or version, at the same abstraction level.

👉 Short takeaway: *Compiler → high-level to machine code; Transpiler → high-level to high-level.*

Examples:

- **Babel** → Transpiles JSX/ES6 into plain JavaScript (ES5) for browsers.
- **TypeScript Compiler (tsc)** → Transpiles TypeScript into JavaScript.

Q: Is it possible to use JSX without React?
A: Yes, but with some caveats. JSX is not tied only to React — it's just a **syntax extension for JavaScript**. By default, JSX compiles into React.createElement() calls, but you can configure tools like **Babel** to transpile JSX into function calls for other libraries or even your own custom renderer.

👉 Short takeaway: *JSX can be used outside React if you set up a transpiler (like Babel) to convert it into something else — React just popularized it.*

Q: What is the use of Fragment and <> in React?
A:

- **Fragment** → Used to group multiple child elements without adding an extra wrapper node (like a

) to the DOM. Keeps the structure clean and avoids unnecessary markup.

- **<> ... (shorthand)** → A shorter syntax for ... . It works the same way but is more concise.

👉 Short takeaway: *Use Fragment or <> when you want to return multiple elements from a component without cluttering the DOM.*

Q: How do you iterate over a list in JSX and what is map?
A: In JSX, you use JavaScript's map() method to loop through arrays and return elements. map() creates a new array by applying a function to each item, and in React that function usually returns JSX. Each item should have a unique key prop.

👉 Example

```
const items = ["Apple","Banana","Cherry"];

<ul>

 {items.map((item, i) => <li key={i}>{item}</li>)}

</ul>
```

Q: What are props in JSX?
A: Props (short for *properties*) are a way to pass data from a parent component to a child component in React. They make components **reusable and dynamic** by allowing customization through external values.

👉 Short takeaway: *Props = inputs to a component, passed like attributes in JSX*

Q: What is the use of the spread operator?
A: Spread operater (...) is used to expand or spread and array or object. In React, the spread operator (...) is often used to **pass all properties of an object as props** to a component in a concise way.

Example:

```
const user = { name: "Rahul", age: 28 };

function Profile(props) {

 return <h2>{props.name} - {props.age}</h2>;

}

// Using spread operator

<Profile {...user} />
```

👉 Short takeaway: *Spread operator lets you forward multiple props at once without writing them individually.*

Q: What are the types of conditional rendering in JSX?
A:

- **If/Else** → Use standard JS control flow for complex conditions.
- **Ternary (? :)** → Inline concise true/false rendering.
- **Logical AND (&&)** → Render only if condition is true.
- **Switch** → Handle multiple discrete states cleanly.
- **Helper Functions** → Encapsulate conditional logic for readability.

👉 Short takeaway: *Conditional rendering in JSX can be done with if/else, ternary, logical AND, switch, or helper functions depending on complexity.*