**what is difference between collection and collections**

“Collection is an interface in java.util package — it’s the root interface of the Java Collection Framework that defines the basic methods for working with groups of objects, such as add, remove, or iterate.

Whereas Collections is a utility class in the same package that provides static methods like sort, reverse, max, min, and synchronize to perform operations on Collection objects.

So in short, Collection is an interface, and Collections is a helper class.”

**difference btwn arraylist and linkedlist and inwhich  senaio would you use each**

“ArrayList is based on a dynamic array whereas LinkedList is based on a doubly linked list.

ArrayList provides faster random access since it uses an index-based system, but insertion and deletion are slow because elements need to be shifted.

On the other hand, LinkedList provides faster insertion and deletion since it only changes node links, but random access is slower as it needs sequential traversal.

So, I’d use ArrayList when my application requires frequent read or search operations, and LinkedList when frequent insertion or deletion is needed.”

**if you try to insert an element into the middle of linkedlist with 10 million elements how will you it perform compared to an arraylist**

“Even though LinkedList offers O(1) insertion, in practice inserting into the middle of a LinkedList with 10 million elements will still perform poorly, because you must first traverse around 5 million nodes to reach the middle — that’s O(n) traversal.

ArrayList, on the other hand, provides O(1) access to the middle index but inserting there requires shifting about 5 million elements, also O(n).

So both perform roughly O(n) overall, but ArrayList might be slightly faster due to better cache locality.

In real systems, for huge datasets, I'd consider a different data structure like CopyOnWriteArrayList, batching inserts, or using a database/stream buffer depending on the use case."

"**So which would you prefer for 10 million data inserts?"**

 Say confidently:

"Neither is optimal for large random inserts — I'd choose a data structure designed for faster inserts like a LinkedHashMap, a TreeMap, or use batch inserts in a database layer."

**explain the difference btwn hashmap and hashtable**

"HashMap is not synchronized and therefore faster, but it's not thread-safe. Hashtable, on the other hand, is synchronized and thread-safe, but slower due to locking overhead.

HashMap allows one null key and multiple null values, whereas Hashtable doesn't allow any null key or value.

Hashtable is a legacy class from Java 1.0, while HashMap is part of the Java Collections Framework introduced in Java 1.2.

In modern applications, we prefer HashMap for single-threaded environments and ConcurrentHashMap for multi-threaded ones."

**If your application is multi-threaded and multiple users can update the same map concurrently, which one will you choose?**

 Answer:

I would choose ConcurrentHashMap, not Hashtable, because it provides thread safety with better performance through segment-level locking instead of synchronizing the whole map.

**why would using a hashtable lead to potential issue in a highly concurent enviroment even through it is syncronized**

"Although Hashtable is synchronized, it uses a single lock for the entire map.

This means if one thread is writing or even reading, all other threads must wait for that lock to be released — which creates heavy contention and reduces performance in highly concurrent environments.

In contrast, ConcurrentHashMap uses finer-grained locking at the bucket level, allowing multiple threads to operate concurrently without blocking each other.

That's why Hashtable becomes a bottleneck under high concurrency, and ConcurrentHashMap is preferred."

**"So what is the concurrency level in ConcurrentHashMap?"**

 You say:

"By default, it's 16 — meaning up to 16 threads can operate on different segments concurrently without blocking each other."

**how would you design a custom immutable class that use a collection like arraylist internally**

"To create an immutable class that uses a collection like ArrayList,

I would make the class final, declare all fields as private and final,

avoid setters, and perform defensive copying inside the constructor.

For getters, I'd return an unmodifiable view using Collections.unmodifiableList().

This ensures no external code can modify the internal state,

even though the collection type itself is mutable."

**"Can we make an immutable class using Set or Map instead of List?"**

 Yes. Same principle applies — use defensive copies and return unmodifiable views like

Collections.unmodifiableSet() or Collections.unmodifiableMap().

**what happen if you add elements to a hashset with dupllicate hashcode() values but different equals() implementations**

"When we add elements to a HashSet that have the same hashCode but different equals() results,

the HashSet will store both elements.

This happens because hashCode() only determines the bucket,

while equals() determines equality within that bucket.

So if equals() returns false, the HashSet treats them as distinct objects

and keeps both, even if they share the same hashCode."

**"What if both hashCode() and equals() are not implemented properly?"**

✅You say:

"It will break the contract. The HashSet may store duplicates or fail to find existing elements, leading to unpredictable behavior."

**what is the purpose of a constructor? can a constructor be static or final**

"The purpose of a constructor is to initialize an object when it's created.

It's automatically called when we use the 'new' keyword and helps set up the initial state of the object.

A constructor cannot be static because static members belong to the class, not to an object,

and it cannot be final because constructors are not inherited, so overriding them isn't possible."

"**Can you make a private constructor?"**

 You say:

"Yes, to restrict object creation — commonly used in Singleton design pattern or utility classes."

can a constructor have a return type if yes what would it mean for the constructor to return something

"A constructor in Java cannot have a return type.

If we declare a return type, it becomes an ordinary method and will not act as a constructor.

This means it won't be called automatically during object creation,

and we'd have to call it manually like a normal method."

Question:

**"So how does a constructor return an object if it has no return type?"**

Answer:

"Internally, constructors don't return anything explicitly.

But the new keyword returns a reference to the newly created object in memory.

The return is implicit and handled by the JVM, not by the constructor itself."

**can a constructor be inherited**

"Constructors cannot be inherited in Java. Each class is responsible for constructing its own objects.

When a subclass object is created, the JVM first calls the superclass constructor (either explicitly via super() or implicitly) to initialize inherited members, and then executes the subclass constructor.

This ensures proper initialization of all fields from parent to child."

Question:

**"Can we call a parent constructor with parameters from a child class?"**

Answer:

"Yes, using super(parameters), we can explicitly call a specific parent constructor to initialize fields or perform tasks before the child constructor executes."

if a coonstructor cannot be inherite can it still caolled in a subclass and how

“Even though constructors cannot be inherited, a subclass can call a parent class constructor using the super() keyword.

When a subclass object is created, the parent constructor is executed first, followed by the subclass constructor.

If no explicit super() is used, the JVM automatically calls the parent’s no-argument constructor.

This ensures that all inherited fields are properly initialized.”

**“What happens if the parent has only a parameterized constructor and you don’t use super() in the child?”**

Answer:

“Compilation error occurs because JVM tries to call the default no-arg constructor, which doesn’t exist.

To fix it, we must explicitly call the parent’s parameterized constructor using super(parameters).”

**can you create an object of an abstract class**

“You cannot create an object of an abstract class directly because it may have abstract methods without implementation.

However, you can create a reference of the abstract class and assign it a concrete subclass object.

Alternatively, you can use an anonymous inner class to instantiate an object on the fly and provide implementations for all abstract methods.”

**“Can an abstract class have constructors?”**

Answer:

“Yes, abstract classes can have constructors. They cannot be used to instantiate the abstract class directly, but the constructors are called when a subclass object is created, to initialize inherited fields.”

**is it possible to create a reference variable of a abstract class type and assign it a concrete class object**

“Yes, it’s possible to create a reference variable of an abstract class type and assign it a concrete class object.

This is upcasting in Java. The abstract class reference can call all methods defined in the abstract class,

and if the concrete subclass overrides abstract methods, the overridden implementation will be executed.

This is commonly used to achieve runtime polymorphism.”

**“Can we assign multiple different subclass objects to the same abstract class reference?”**

Answer:

“Yes, that’s the beauty of polymorphism. For example:

**explain the concept of coupling and cohesion in oops**

“Cohesion refers to how closely the responsibilities of a single class or module are related. High cohesion means the class does one thing well, improving readability and maintainability.

Coupling refers to the dependency between classes or modules. Low coupling means classes are independent and changes in one class have minimal impact on others.

In OOP, the goal is to maximize cohesion and minimize coupling for clean, maintainable, and flexible code.”

**“If you have to design a system, how do you ensure high cohesion and low coupling?”**

Answer:

High cohesion: Break functionality into small classes, each handling a single responsibility (SRP – Single Responsibility Principle).

Low coupling: Use interfaces, dependency injection, or observer pattern to reduce direct dependencies between classes.

**why is it important to minimize coupling and maximeze cohesion**

“Minimizing coupling is important because it reduces dependency between classes, making the system easier to maintain, test, and reuse.

Maximizing cohesion is important because each class or module has a focused responsibility, improving readability, maintainability, and reducing complexity.

Together, high cohesion and low coupling result in a clean, flexible, and scalable design, which is a core principle of good OOP and software engineering.”

**“In a real project, how do you enforce high cohesion and low coupling?”**

Answer:

High cohesion: Follow Single Responsibility Principle (SRP) → each class does one job.

Low coupling: Use interfaces, dependency injection, or observer pattern to reduce tight dependencies.

Regular code reviews and design discussions also help maintain these principles.

**can you have low coupling and low cohesion at the same time provide an example**

“Yes, it’s possible to have low coupling and low cohesion at the same time.

For example, a utility class that doesn’t depend on any other class (low coupling) but performs multiple unrelated tasks like calculating invoices, sending emails, and generating reports (low cohesion).

While the class is independent, it is internally messy and violates the Single Responsibility Principle.

This is generally considered poor design.”

**“How would you improve this design?”**

Answer:

Increase cohesion: Split the utility class into multiple focused classes:

InvoiceCalculator, EmailSender, ReportGenerator.

Maintain low coupling: Keep each class independent or use interfaces if interaction is needed.

**difference btwn spring and springboot**

“Spring is a comprehensive Java framework for building enterprise applications, but it requires manual configuration and deployment to an external server.

Spring Boot is built on top of Spring to simplify development. It provides auto-configuration, starter dependencies, and embedded servers, enabling rapid application development with minimal boilerplate code.

Essentially, Spring Boot reduces complexity while still using the core Spring framework features.”

**“Why do we prefer Spring Boot for microservices over Spring?”**

Auto-configuration and starter dependencies reduce setup time.

Embedded server allows standalone microservice jars.

Easier integration with Spring Cloud, REST APIs, and DevOps pipelines.

**what are actuators in springboot**

“Spring Boot Actuators are built-in endpoints that allow developers to monitor and manage applications in production.

They provide real-time information like health status, metrics, environment properties, and custom info.

Actuators are added via spring-boot-starter-actuator dependency and can be secured and customized according to production needs.

Essentially, they act like a dashboard for your Spring Boot application.”

**“How do you secure actuator endpoints in production?”**

Use Spring Security to restrict access to authorized users.

Expose only required endpoints using:

management.endpoints.web.exposure.include=health,info

Disable sensitive endpoints in production to prevent info leak.

**how will you ensure compatibility when upgrading to the latest version of springboot**

“To ensure compatibility when upgrading to the latest Spring Boot version, I follow a structured approach:

First, I review the official release notes to identify deprecated features and breaking changes.

Next, I update the Spring Boot version and related dependencies, ensuring third-party libraries are compatible.

I run all automated tests — unit, integration, and API tests — to catch any issues.

I check configuration files for changes in defaults and deploy the upgraded version to a staging environment for monitoring using actuator endpoints and logs.

Finally, I maintain a rollback plan to quickly revert to the previous version if any critical issues arise.

This systematic approach ensures smooth and compatible upgrades.”

**“What if a third-party library isn’t compatible with the new Spring Boot version?”**

Check if a newer version of the library is available.

If not, consider temporary workarounds, custom adapters, or delay upgrade until compatible.

Always document such exceptions for future upgrades.

**about caches in springboot**

“Caching in Spring Boot is a mechanism to store frequently accessed data in memory to reduce expensive database calls or computations.

Spring Boot provides caching via the Spring Cache abstraction.

By enabling caching with @EnableCaching and annotating methods with @Cacheable, we can store and retrieve method results from the cache.

@CachePut allows updating cache without skipping method execution, and @CacheEvict helps in removing stale data.

Using caching improves application performance and reduces latency significantly.”

**“Which cache providers can we use with Spring Boot?”**

Answer:

In-memory: ConcurrentMapCacheManager (default), Caffeine, EhCache

Distributed: RedisCacheManager, Hazelcast

Choose based on scale, performance, and cluster requirements.

**can springboot built in caching machanism be used in a distributed system**

“Spring Boot’s built-in caching mechanism is local by default, which means it stores cached data in the memory of a single application instance.

In a distributed system with multiple instances, this local cache is not shared, so it cannot ensure cache consistency.

To use caching in a distributed environment, we integrate a distributed cache provider like Redis, Hazelcast, or clustered EhCache.

Spring Boot allows plugging in these distributed cache managers seamlessly, ensuring all application instances share the same cache and remain consistent.”

Question:

**“If multiple microservices share the same Redis cache, how do you avoid stale data issues?”**

Answer:

Use TTL (Time-to-Live) for cache entries.

Use @CacheEvict or @CachePut to invalidate/update cache after DB changes.

Consider message brokers (Kafka/RabbitMQ) to notify other services of updates if necessary.

**about spring security and ways to do it**

“Spring Security is a powerful framework for securing Spring applications. It provides authentication, authorization, session management, CSRF protection, and password encoding.

We can implement security in multiple ways: form-based login, HTTP Basic authentication, JWT or token-based security for REST APIs, OAuth2 for SSO, LDAP/Active Directory for enterprise authentication, and method-level security using annotations like @PreAuthorize.

It is highly customizable and integrates seamlessly with Spring Boot applications.”

**“Which security approach would you use for a microservices-based REST API?”**

Use JWT / token-based authentication for stateless security.

Secure sensitive endpoints using @PreAuthorize or custom filters.

Optionally integrate OAuth2 or OpenID Connect for SSO across multiple services.

**how do microservices communicate with each other**

“Microservices communicate with each other using synchronous or asynchronous methods.

In synchronous communication, a service calls another service via REST APIs or gRPC and waits for a response. This is simple but tightly coupled.

In asynchronous communication, services use message brokers like Kafka or RabbitMQ to exchange messages without waiting for immediate responses. This approach is loosely coupled, scalable, and resilient.

The choice depends on the use case: real-time operations use synchronous calls, whereas background processing and event-driven workflows use asynchronous messaging.”

**“Which approach is better for highly scalable microservices?”**

Answer:

Asynchronous messaging is preferred because services are loosely coupled and can scale independently.

Synchronous REST/gRPC can be used for critical, real-time interactions.

Often a hybrid approach is used in real systems.

what would happen if you try to use synchronous communication for critical microservices that have varying latencies? what alternatives would you consider

“Using synchronous communication for critical microservices with varying latency can cause request blocking, cascading delays, timeouts, and reduce overall system resilience.

Alternatives include:

Asynchronous communication using message brokers like Kafka or RabbitMQ to decouple services.

Circuit breaker pattern (Resilience4j/Hystrix) to detect slow or failing services and provide fallback responses.

Configure timeouts and retries with exponential backoff.

Bulkhead pattern to isolate threads or queues for critical services.

A hybrid approach: synchronous for reliable fast services and asynchronous for slow/unpredictable services.

These approaches help maintain scalability, resilience, and consistent performance in distributed systems."

**what is @springbootApplication annotation**

"@SpringBootApplication is a convenience annotation in Spring Boot that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan.

It marks the main class as a Spring Boot application, enables component scanning to detect beans and controllers, and activates auto-configuration to automatically configure Spring components based on the classpath and dependencies.

Essentially, it reduces boilerplate and makes application setup faster and easier."

**"Can we replace @SpringBootApplication with the three individual annotations?"**

Yes, you can write:

"Yes, `@SpringBootApplication` can be replaced with the three annotations: `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`. Functionally it is the same, but `@SpringBootApplication` is a convenient shortcut that reduces boilerplate by combining all three."

**can you remove the @springbootApplication annotaion from a springboot application and still have it run correctly**

"Technically, you can remove @SpringBootApplication and still run a Spring Boot application, but only if you explicitly add its constituent annotations: @Configuration, @EnableAutoConfiguration, and @ComponentScan.

If you remove @SpringBootApplication without replacing them, the application may start, but component scanning and auto-configuration will not work, and the application will behave like a plain Spring application rather than a fully configured Spring Boot app.

Essentially, @SpringBootApplication is a convenience annotation that simplifies setup but is not strictly mandatory if the underlying annotations are provided."

**“What happens if you remove only @EnableAutoConfiguration?”**

Answer:

Auto-configuration won’t run → default beans like DataSource, DispatcherServlet, or embedded Tomcat won’t be created automatically.

The app may fail if you rely on auto-configured components.

What happens if the app relies on them

If your code expects these beans and they don’t exist, Spring will throw exceptions at startup like:

NoSuchBeanDefinitionException

BeanCreationException

Example:

A REST controller uses a JPA repository.

Without auto-configuration, DataSource isn’t created → app fails to start.

3 Key takeaway

Auto-configuration is critical for Spring Boot apps that rely on default setups.

Removing @SpringBootApplication without providing @EnableAutoConfiguration or manually defining all required beans can break the app.

**what testing tools will you prefer in your projects**

“In my projects, I prefer a layered approach to testing.

For unit testing, I use JUnit 5 with Mockito for mocking dependencies and AssertJ for assertions.

For integration testing, I use Spring Boot Test with Testcontainers or H2 in-memory database to verify component interactions.

For end-to-end or API testing, I use RestAssured or Postman and sometimes Cucumber for BDD style tests.

Additionally, for performance and load testing, tools like JMeter or Gatling are preferred.

This approach ensures reliable, maintainable, and high-quality software.”

**what are restfull web services why do we need them**

“RESTful web services are web services that follow REST principles, enabling client-server communication over HTTP.

They are stateless, resource-based, and lightweight, typically using JSON or XML.

We need RESTful services because they allow interoperability across platforms, easy scalability, fast and lightweight communication, and flexibility to support web apps, mobile apps, and microservices.

In Spring Boot, RESTful services are easily implemented using @RestController and standard HTTP methods like GET, POST, PUT, and DELETE.”

**what is the difference btwn restfull services and soap**

“RESTful web services are lightweight, stateless, and use flexible formats like JSON or XML. They follow resource-based principles and are ideal for microservices, web, and mobile applications.

SOAP web services are strictly XML-based, heavier, and support built-in standards for security, transactions, and reliability. They are ideal for enterprise legacy systems where strict contracts and security are required.

In short, REST is flexible and fast, while SOAP is formal, standardized, and feature-rich.”

**how to handle exception in springboot**

“In Spring Boot, exceptions can be handled either at the controller level using @ExceptionHandler or globally using @ControllerAdvice or @RestControllerAdvice.

@ExceptionHandler allows handling exceptions for specific controllers, while @ControllerAdvice handles exceptions across all controllers.

For REST APIs, it’s a best practice to return structured JSON responses containing timestamp, HTTP status, and error message.

This approach ensures clean, maintainable, and user-friendly error handling in Spring Boot applications.”

**how to call one microserives from another microservices**

“One microservice can call another either synchronously or asynchronously.

In synchronous communication, a service calls another via REST API, Feign Client, or gRPC and waits for the response. This is simple but blocking.

In asynchronous communication, services communicate via message brokers like Kafka or RabbitMQ without waiting for a response. This approach is loosely coupled, scalable, and resilient.

The choice depends on the use case: real-time operations use synchronous calls, whereas background processing or event-driven workflows use asynchronous messaging.”

**merge two unsorted arrays and print the merged one in sorted way by using stream api**

```
import java.util.Arrays;
import java.util.stream.IntStream;

public class MergeArraysStream {
    public static void main(String[] args) {
        int[] arr1 = {5, 2, 9};
        int[] arr2 = {1, 7, 3};

        // Merge two arrays using IntStream.concat
        int[] mergedSorted = IntStream.concat(Arrays.stream(arr1), Arrays.stream(arr2))
                                      .sorted()
                                      .toArray();

        // Print
        System.out.println(Arrays.toString(mergedSorted));
    }
}
```