what is difference between collection and collections

"`Collection` is an **interface** in the `java.util` package.
It's the **root interface** of the Java Collection Framework that defines the basic methods for working with groups of objects — like `add()`, `remove()`, `size()`, and `iterator()`.

`Collections`, on the other hand, is a **utility class** in the same package (`java.util.Collections`).
It provides **static helper methods** to operate on collections — such as sorting, searching, reversing, or making a collection synchronized or unmodifiable.

In short, `Collection` defines the structure, while `Collections` provides utility methods to manipulate that structure."

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

"Both `ArrayList` and `LinkedList` are implementations of the `List` interface and store elements in an ordered way, but their internal structures and performance differ.

`ArrayList` is based on a **dynamic array**, so it provides **fast random access** (O(1)) but **slower insertions and deletions** (O(n)) in the middle, because elements need to be shifted.

`LinkedList`, on the other hand, is based on a **doubly linked list**, where each node holds references to the previous and next nodes.
So, insertions and deletions are **faster (O(1))** once you reach the position, but accessing elements by index is **slower (O(n))** because traversal is needed.

In real-world scenarios —
• Use **ArrayList** when you need **fast access/read** and fewer insert/delete operations.
• Use **LinkedList** when your use case involves **frequent insertions/deletions**, especially from the beginning or middle of the list."

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

“Inserting into the middle of a LinkedList with 10 million elements looks O(1), but actually it’s O(n) because traversal is required.
It needs to walk through roughly 5 million nodes before inserting.

ArrayList also takes O(n) because of element shifting, but in practice, ArrayList is faster due to its contiguous memory structure and better CPU caching.

So even though both are O(n), ArrayList usually performs better in such large-scale insert operations.”
So, in practice, both perform poorly for mid insertions, and ArrayList may still be faster.”

“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

“Both HashMap and Hashtable store data in key–value pairs and internally use hashing, but there are some key differences.

HashMap is **non-synchronized**, so it’s faster and used in single-threaded or read-heavy applications.
Hashtable, on the other hand, is **synchronized**, meaning it’s thread-safe but slower — and considered **legacy** since Java 1.0.

HashMap allows one **null key** and multiple **null values**, while Hashtable doesn’t allow any null keys or values.

In real-world projects, we rarely use Hashtable. For concurrent scenarios, we prefer **ConcurrentHashMap**, which provides thread-safety with better performance.

HashMap was introduced in Java 1.2 as part of the Collections Framework, while Hashtable existed since Java 1.0.”

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

“Even though Hashtable is synchronized, it’s not efficient for highly concurrent use cases because it locks the entire map for every operation.
So if multiple threads are reading or writing simultaneously, they all get blocked, which reduces throughput.

That’s why in modern Java, we use ConcurrentHashMap — it allows multiple threads to work on different parts of the map at the same time, improving performance significantly.”

“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

“I’d mark the class as final and all fields as private and final.
If it uses a mutable collection like ArrayList internally, I’ll create a defensive copy inside the constructor and return an unmodifiable view in the getter.

That way, even if someone modifies the original list outside, my class remains unaffected.
This approach ensures true immutability in a class that internally uses a mutable collection.”

“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

“If two objects have the same hashCode but different equals logic, HashSet will put them in the same bucket but treat them as separate entries.

HashSet first checks hashCode to find the bucket and then uses equals to confirm uniqueness.

So if equals returns false, both objects are stored — but too many collisions will slow down lookup performance.
That’s why it’s important to override hashCode and equals consistently.”

“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

No, constructors are **not inherited** in Java. Each class defines its own constructor.
However, when we create a subclass object, the **parent’s constructor is always executed first**, either automatically or through an explicit `super()` call.
This ensures that the parent part of the object is properly initialized before the child part.

For example, if I have a `Base` and `Derived` class, the derived class constructor can call `super()` to initialize the base fields, but it doesn’t inherit the constructor itself.

Question:

“Can we call a parent constructor with parameters from a child class?”

Answer:

Yes, we can call a parent class constructor with parameters from the child class using `super(parameter_list)`.
It’s very common in real-world projects when we want to pass initialization data to the parent class.
The `super()` call must always be the **first line** inside the child constructor; otherwise, the code won’t compile.

“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

No, we can’t instantiate an abstract class directly because it may contain abstract methods without implementation.
However, we can **create a reference** of the abstract class and **assign it to a subclass object** — that’s a common pattern in polymorphism.
Another way is through **anonymous inner classes**, where we provide the implementation of abstract methods immediately while creating the object.

“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 absolutely possible.
In fact, that’s one of the main reasons abstract classes exist — to define a common type that multiple subclasses can implement differently.
We can create a reference of the abstract class and assign it a concrete class object.
This enables **runtime polymorphism**, where the actual method that gets executed depends on the object type, not the reference type.

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

Answer:

“Yes, we can assign multiple different subclass objects to the same abstract class reference, but one at a time.
This is a classic example of **runtime polymorphism** — where the reference type is abstract (or parent class), and the actual object type changes dynamically at runtime.
The JVM decides which method to execute based on the actual object the reference is pointing to.

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?”

To ensure high cohesion and low coupling in a system design, I follow a few core design practices.
First, I ensure that every class has a **single clear responsibility** — this maintains high cohesion.
Second, I use **interfaces and dependency injection** to decouple components — this provides low coupling.
I also follow SOLID principles and use design patterns like **Factory, Strategy, and Observer**, depending on the scenario.

For example, in a recent project, my `OrderService` depended on a `PaymentService` interface, not on a specific implementation.
At runtime, Spring injected the actual `CreditCardPayment` or `UPIPayment` class.
This way, I could change payment logic without touching the order logic — which is a perfect example of **high cohesion and low coupling** in real-world systems.

Why is it important to minimize coupling and maximize 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

Whenever I upgrade Spring Boot in a project, I always follow a **controlled migration approach**.

I start by checking the **Spring Boot release notes** and **migration guide** to identify any breaking or deprecated features. Then, I upgrade **minor versions first** to reduce risk.

I rely on **Spring Boot Starters** and the **dependency BOM**, so all transitive dependencies stay version-aligned.

After upgrading, I execute the **full test suite (unit + integration)** to ensure backward compatibility, and I validate **Actuator endpoints**, **security configurations**, and any **custom beans**.

Finally, I deploy the upgraded build in a **staging environment** before production rollout.

This step-by-step process has helped me safely upgrade from Spring Boot 2.5 to 2.7 and later to 3.2 in my last project without any major regressions.

“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

In Spring Boot, caching is used to **improve performance** by storing frequently accessed data in memory, so repeated calls don’t hit the database or external APIs.

I usually enable caching using `@EnableCaching` and apply it to methods with `@Cacheable`. For example, in one of my projects, we cached product and user data in Redis, which reduced DB calls significantly.

Spring Boot provides a **common caching abstraction**, so I can switch between providers like **Ehcache**, **Caffeine**, or **Redis** just by changing configuration — no code change needed.

We also used `@CacheEvict` to clear caches after updates, ensuring data consistency.

Overall, caching helped us improve API response time from 2-3 seconds to under 100 milliseconds.

“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

By default, Spring Boot’s caching mechanism is **local to the JVM**, which means it’s not suitable for distributed systems where multiple instances of the application are running.

In distributed environments, we need **a shared cache store** so that all instances access the same cached data.

In my project, we integrated Spring Boot caching with **Redis**, which served as a distributed cache for our microservices. It ensured **data consistency and faster responses** across all nodes.

The good part is — Spring Boot provides a **cache abstraction layer**, so I didn’t have to change my code — just configured the cache provider in `application.properties`.

So yes, Spring Boot caching can be used in distributed systems, **but only when backed by a distributed cache provider like Redis or Hazelcast.**

“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 the standard way to secure Spring Boot applications. It provides both authentication and authorization out of the box.
In my projects, I’ve used it mainly in two ways — form-based login for web modules and JWT-based security for REST APIs.
For microservices, I prefer JWT or OAuth2 because they are stateless and integrate well with API gateways.
It also supports advanced configurations like role-based access control, CSRF protection, and method-level security using `@PreAuthorize` annotations.

So overall, it gives you flexibility — whether you’re building a monolith or distributed system, you can choose the right strategy as per your use case.

“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?”

For highly scalable microservices, I prefer using **asynchronous, event-driven communication**.

This approach allows each microservice to work independently — they communicate through **message brokers like Kafka or RabbitMQ**, instead of direct REST calls.

Since there’s no blocking or waiting for responses, services can handle large workloads efficiently and scale horizontally without tight coupling.

In one of my projects, we implemented Kafka-based communication between Order, Payment, and Notification services — it improved throughput and system resilience significantly compared to synchronous REST calls.

what is @springbootApplication annotation

It tells Spring Boot to start scanning for components, enable auto-configuration based on dependencies, and mark this class as the main configuration entry point.

In simple terms, it bootstraps the entire Spring Boot application.

For example, when I run my main class annotated with `@SpringBootApplication`, Spring Boot automatically configures beans like DataSource, JPA, or MVC based on the libraries present in the classpath — so I don't have to write any XML or manual configuration.

"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` annotation from a Spring Boot application and still have it run correctly?

Yes, it's possible to remove `@SpringBootApplication` and still run a Spring Boot app, but only if I manually add its three core annotations:
`@SpringBootConfiguration`, `@EnableAutoConfiguration`, and `@ComponentScan`.

If I remove it completely without replacements, Spring Boot won't perform auto-configuration or component scanning, and my beans won't be initialized — so the application may start, but it won't behave as expected.

In practice, we always use `@SpringBootApplication` since it makes the code cleaner and ensures all the necessary setup is done automatically.

“What happens if you remove only @EnableAutoConfiguration?”

Answer:

If I remove only `@EnableAutoConfiguration`,
Spring Boot will lose its ability to auto-configure components based on the classpath.

The application will still start since `@ComponentScan` will still detect annotated beans,
but no default beans like `DispatcherServlet`, `DataSource`, or `EntityManager` will be created automatically.

In short, I’ll have to manually configure everything like a traditional Spring MVC application.

That’s why `@EnableAutoConfiguration` is a core part of the `@SpringBootApplication` annotation — it’s what makes Spring *Boot* truly “Boot”.

what testing tools will you prefer in your projects

In my projects, I mainly use **JUnit 5** and **Mockito** for unit testing,
since they provide a clean way to test business logic independently.

For integration tests, I use **Spring Boot Test** with `@SpringBootTest` annotation,
which loads the ApplicationContext and lets me verify service and repository layers together.

For API-level testing, I prefer **Rest Assured** for automation and **Postman** for manual validation.

We also integrate **JaCoCo** and **SonarQube** in our CI/CD pipeline to ensure good code coverage and maintain code quality standards.

For performance and load testing, tools like **Apache JMeter** are used to validate scalability.

What are RESTful Web Services, and why do we need them?

RESTful Web Services are APIs that follow REST architectural principles.
They expose resources through endpoints and use standard HTTP methods like GET, POST, PUT, and DELETE to perform operations.

The main reason we use RESTful services is because they are **lightweight, stateless, scalable, and easily consumed** by web or mobile clients.

In my projects, I've implemented RESTful APIs using **Spring Boot's @RestController** and **ResponseEntity** for structured responses.

REST is preferred over SOAP in modern microservice architectures because it supports **JSON**, **loose coupling**, and **better performance**.

What is the difference between RESTful Web Services and SOAP Web Services?

The key difference is that **SOAP is a protocol**, while **REST is an architectural style**.

SOAP uses **XML messages** and is more **strict and heavyweight**, whereas REST works over **HTTP** and supports **JSON**, making it **lightweight, faster, and easier to implement**.

In my projects, I've mostly used **RESTful services** with Spring Boot for microservice communication since they are **stateless, scalable**, and **easily consumed by web and mobile clients**.

However, if strict security or ACID-compliant transactions are required (e.g., in banking systems), SOAP might still be preferred.

how to handle exception in springboot

In my Spring Boot projects, I handle exceptions using a **global exception handler** with `@RestControllerAdvice`.

I create **custom exceptions** for domain-specific errors (like `UserNotFoundException`) and return a **structured response** with proper HTTP status codes.

This ensures consistent error handling across all REST APIs and helps avoid exposing internal stack traces.

For example, I extend `ResponseEntityExceptionHandler` to handle validation errors, and use `@ExceptionHandler` for custom runtime exceptions.

This approach improves **readability**, **debugging**, and **API client experience**.

How do you call one microservice from another microservice in Spring Boot?

“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));
    }
}
```