**Lamda Expression:**

**1.No Parameters:**

java

Copy

Runnable r = () -> System.out.println("Hello, world!");

r.run(); // Output: Hello, world!

1. **One Parameter:**

java

Copy

// Takes a single integer and prints it

Consumer<Integer> print = x -> System.out.println(x);

print.accept(5); // Output: 5

1. **Multiple Parameters:**

java

Copy

BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;

System.out.println(add.apply(3, 4)); // Output: 7

1. **Block of Code (Multiple Statements):**

java

Copy

Comparator<Integer> compare = (a, b) -> {

if (a > b) return 1;

if (a < b) return -1;

return 0;

};

System.out.println(compare.compare(3, 4)); // Output: -1

**Disadvantages of Optional:**

* Can lead to performance overhead.
* Overuse in non-functional code can reduce readability.
* Not designed for fields or method parameters.

**Method reference:**

* **Static method reference:** ClassName::staticMethodName
* **Instance method reference (on a specific object):** instanceObject::instanceMethodName
* **Instance method reference (on class type):** ClassName::instanceMethodName
* **Constructor reference:** ClassName::new for crezting object of class

Supplier<Person> personSupplier = Person::new; // Constructor reference

Person person = personSupplier.get();

**1. What is Dependency Injection (DI)?**

* **Answer**: Dependency Injection is a design pattern that allows a class to receive its dependencies (i.e., other objects it needs) from an external source rather than creating them itself. The goal of DI is to decouple the classes and make the system more modular, maintainable, and testable.
* Inversion of Control (IoC) is a design principle where the control of object creation and dependency management is transferred from the application to an external framework or container. It decouples components, making the system more modular and easier to manage, often implemented through dependency injection or event-driven programming.
* In short, IoC is the concept, and the Spring Container is a concrete implementation of it within the Spring Framework.
* **What is @Fetch in Hibernate?**
  + **Answer**: @Fetch specifies the fetching strategy for an association (like EAGER or LAZY). By default, @OneToMany and @ManyToMany relationships are lazily fetched, and @ManyToOne and @OneToOne are eagerly fetched.

@Version

@MappedSuperclass

**Spring bean life cycle?**

The lifecycle of a Spring Bean involves several key stages. First, the bean is instantiated by the Spring container. Next, Spring injects dependencies into the bean via setter or constructor injection. After that, the @PostConstruct method (or InitializingBean's afterPropertiesSet() method) is called for initialization tasks. If configured, custom initialization methods are invoked. The bean is then ready for use within the application. When the application context is closed or the bean is no longer needed, the @PreDestroy method (or DisposableBean's destroy() method) is called for cleanup before the bean is destroyed.

* **What is @Autowired in Spring and how does it work with ApplicationContext?**

**Answer:**  
@Autowired is used for automatic dependency injection. When a bean is annotated with @Autowired, Spring will inject the required dependency at runtime, either via constructor, setter, or field injection. The ApplicationContext will manage the wiring of dependencies automatically, provided that the beans are defined in the context.

**CascadeType.ALL**

* **Definition**: **CascadeType.ALL** is a convenience option that applies all the cascade types (PERSIST, MERGE, REMOVE, REFRESH, and DETACH) to the relationship between entities.

@joinCLoumn- any colum of child entity is the foreginkey of parent entity

* **@PrimaryKeyJoinColumn**: This annotation is used in inheritance mapping, specifically for @OneToOne or @OneToMany relationships, when the child entity's primary key is the same as the parent entity's primary key. It indicates that the join is based on the primary key of the parent entity.

MapppedBy – foregn key mapped by the other entity

ParentEntity means own the relationship which means @joincloum and @primaryJoinColumn

**11. How do you handle orphaned entities in a @OneToMany relationship in Hibernate?**

* **Answer**: Orphan removal is used to delete a child entity when it is no longer associated with the parent entity. This can be done using the orphanRemoval attribute on the @OneToMany mapping.

Example:

java

Copy

@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)

private List<OrderItem> orderItems;

#lazy instatiation of singletone class

// Static inner class responsible for holding the Singleton instance

private static class SingletonHelper {

// The Singleton instance is created when the inner class is loaded

private static final Singleton INSTANCE = new Singleton();

}

// Public method to access the Singleton instance

public static Singleton getInstance() {

return SingletonHelper.INSTANCE;

}

Alternative of singletone

* **Dependency Injection (DI):** DI allows flexible object management, promoting loose coupling and easier testing. It enables better control over object lifecycle and dependencies.
* **Factory Pattern:** Useful for creating instances when different configurations or types are needed, ensuring more control over object creation.

**Scenarios to Avoid Using the Singleton Pattern:**

* **Global State Issues:** If you need to manage mutable global state, Singletons can lead to hidden dependencies and tight coupling, making your system harder to maintain and test.
* **Concurrency Problems:** In multi-threaded environments, managing the Singleton can introduce complexity, especially when using synchronization.
* **Overuse in Large Applications:** Overusing Singletons can lead to poor scalability and testability, as they act as implicit global variables.

**3. Can encapsulation be broken in Java? (Using Reflection API)**

Yes, encapsulation **can be broken** using **Java Reflection API**. The setAccessible(true) method allows access to private fields.

**Example: Breaking Encapsulation Using Reflection**

java

CopyEdit

import java.lang.reflect.Field;

class Person {

private String name = "John Doe";

}

public class ReflectionDemo {

public static void main(String[] args) throws Exception {

Person p = new Person();

Field field = p.getClass().getDeclaredField("name");

field.setAccessible(true); // Bypasses encapsulation

field.set(p, "Hacked Name");

System.out.println(field.get(p)); // Output: Hacked Name

}

}

**How to Prevent Reflection from Breaking Encapsulation?**

* Use **SecurityManager** (Deprecated in Java 17+).
* Use **sealed classes (Java 15+)** to restrict subclassing.
* Avoid exposing **sensitive fields** that might be altered.

**4. What is dynamic method dispatch?**

**Dynamic Method Dispatch (Runtime Polymorphism)** is a mechanism in which method calls are resolved at **runtime** instead of compile-time.

* Happens **when a parent class reference points to a child class object**.
* Overridden methods in the subclass are called **at runtime**.

**Example:**

java

CopyEdit

class Parent {

void show() { System.out.println("Parent show"); }

}

class Child extends Parent {

@Override

void show() { System.out.println("Child show"); }

}

public class Test {

public static void main(String[] args) {

Parent obj = new Child();

obj.show(); // Output: Child show

}

}

**Why does "Child show" get printed?**

* Method resolution happens at **runtime** (not compile-time) based on the **actual object type**, not the reference type.

**6. What is the difference between @PreAuthorize and @Secured annotations in Spring Security?**

* **Answer:** Both annotations are used to restrict method access based on roles or permissions, but there are differences:
  + **@Secured**: Specifies roles that can access a method. It uses role names (e.g., "ROLE\_ADMIN").
  + **@PreAuthorize**: More flexible, as it allows the use of expressions (e.g., SpEL) to define access control rules, such as checking for specific permissions, user attributes, or roles.

**12. How can you protect against session fixation attacks in Spring Boot?**

* **Answer:** Session fixation attacks occur when an attacker forces a user to use a specific session ID. In Spring Security, session fixation protection is enabled by default. Spring Security generates a new session ID after successful authentication, which prevents the attacker from using the compromised session ID.

**6. How do you protect sensitive data in a JWT?**

**Answer:** While the payload in a JWT is base64-encoded, it's **not encrypted** and can be easily decoded. To protect sensitive data:

1. **Use Secure Transmission**: Always send JWTs over HTTPS to prevent them from being intercepted by attackers.
2. **Avoid Sensitive Data in the Payload**: Do not include highly sensitive information, such as passwords or personally identifiable information (PII), in the JWT payload. Store only non-sensitive data.
3. **Use Strong Signing Algorithms**: Use secure algorithms like HS256 (HMAC with SHA-256) or RS256 (RSA signature with SHA-256) to sign the JWT and ensure the integrity of the token.
4. **Encrypt the Token**: If you need to store sensitive information in the payload, you can encrypt the entire JWT using a library like JWE (JSON Web Encryption).

**Follow-up: How would you customize bean initialization using a custom init method or destroy method?**

* You can define custom initialization and destroy methods in Java configuration with @Bean(initMethod="initMethodName", destroyMethod="destroyMethodName"). In XML configuration, you can use the init-method and destroy-method attributes in the <bean> element.

**8. Can you explain how @Autowired works in terms of bean lifecycle and dependency injection?**

* The @Autowired annotation tells Spring to inject dependencies into a bean, either via constructor, setter, or field injection. During the bean initialization phase, Spring resolves the dependencies by matching the annotated bean’s required dependencies from the container. These dependencies are injected once the bean is created, but before it is fully initialized.

**What is the significance of BeanDefinition and how does it relate to the lifecycle of beans in ApplicationContext?**

* **BeanDefinition** is a key concept in Spring that represents the configuration of a bean in the Spring container. It stores metadata about the bean, including its class, scope, dependencies, initialization methods, destruction methods, and any other configurations needed for the bean’s lifecycle.
* When the ApplicationContext is initialized, Spring uses BeanDefinition to register and configure all beans defined in the context (either through XML, annotations, or Java-based configuration). The ApplicationContext processes each BeanDefinition, and based on that, it creates and manages the bean instances.

**Follow-up**: **How does Spring process BeanDefinition during context initialization?**

* During context initialization, Spring reads and loads the BeanDefinitions from various sources (XML, annotations, Java configuration). It then registers each BeanDefinition in the container. These definitions are stored in the BeanFactory or ApplicationContext.
* For each bean, Spring resolves any dependencies and applies configuration, such as setting properties or invoking initialization methods. Once this process is complete, the beans are fully initialized and available for use.\*\*\*\*\*\*\*\*

**13. How would you create and manage a custom bean post-processor to modify beans during their initialization phase in Spring?**

A **custom BeanPostProcessor** is an interface in Spring that allows you to modify bean instances before or after initialization. It’s typically used to intercept beans during their lifecycle, such as modifying properties or adding behavior.

To create and manage a custom BeanPostProcessor:

1. **Implement the BeanPostProcessor interface** and override the methods postProcessBeforeInitialization() and postProcessAfterInitialization().
2. **Register the BeanPostProcessor** as a bean in the Spring configuration, either through XML or Java-based configuration (via @Configuration or component scanning).

java

Copy

@Component

public class CustomBeanPostProcessor implements BeanPostProcessor {

@Override

public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

// Modify the bean before initialization, if necessary

return bean;

}

@Override

public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

// Modify the bean after initialization

return bean;

}

}

The BeanPostProcessor allows you to add behavior to the Spring bean lifecycle, for instance, logging initialization, modifying bean properties, or applying dynamic proxy functionality.

**Follow-up**: **Can you walk me through the process of implementing BeanPostProcessor and how it's used to intercept the Spring bean lifecycle?**

* First, implement the BeanPostProcessor interface and override the two methods:
  + **postProcessBeforeInitialization**: This is called before the @PostConstruct or custom initialization method is invoked, and it's used for pre-initialization logic (e.g., modifying bean properties).
  + **postProcessAfterInitialization**: This is called after the initialization methods are executed and is typically used to apply any final modifications to the bean before it's fully ready for use (e.g., proxying, logging).
* Once implemented, register the BeanPostProcessor with Spring, either by defining it as a bean in the configuration or marking it as a @Component for automatic component scanning.

t attributes can be specified in the @Column annotation?

**A:** The @Column annotation has several useful attributes:

* name: The name of the column in the database.
* length: The length of the column (for String columns).
* nullable: Whether the column can have a NULL value.
* unique: Whether the column value must be unique across all rows in the table.
* insertable: Whether the column is part of insert operations.
* updatable: Whether the column is part of update operations.

**Q:** How do you map a java.util.Date or java.time.LocalDate field to a column?

**A:** For java.util.Date or java.time.LocalDate, you typically use the @Temporal annotation for Date and rely on @Column for LocalDate.

Example for java.util.Date:

java

Copy

@Temporal(TemporalType.DATE)

@Column(name = "birth\_date")

private Date birthDate;

For java.time.LocalDate, JPA will handle the mapping automatically:

java

Copy

@Column(name = "birth\_date")

private LocalDate birthDate;

**7. Entity Lifecycle and Callbacks**

**Q:** How do lifecycle annotations like @PrePersist, @PostPersist, @PreUpdate, @PostUpdate, @PreRemove, and @PostRemove work?

**A:** These annotations allow you to hook into specific points in the entity lifecycle. For example:

* @PrePersist: Called before an entity is persisted.
* @PostPersist: Called after an entity is persisted.
* @PreUpdate: Called before an entity is updated.
* @PostUpdate: Called after an entity is updated.
* @PreRemove: Called before an entity is removed.
* @PostRemove: Called after an entity is removed.

Example:

java

Copy

@PrePersist

public void beforePersist() {

this.createdAt = LocalDateTime.now();

}

 **Primary Key** is useful when the identifier has real-world business meaning and doesn’t change. Entity or tables generated key

 **Surrogate Key** is beneficial for simplicity, performance, and flexibility, especially when the identifier is not business-specific or subject to change. Databse generated key

**9. What is the role of the spliterator() method in collections? How does it improve performance with parallel streams?**

* **spliterator()**:
  + The spliterator() method is used to split a collection into smaller parts for parallel processing. It is an alternative to an iterator and is especially useful for efficiently processing large collections in parallel streams.

**31. Write a method to find the intersection of two lists (both ArrayLists).**

The intersection of two lists is a set of elements that appear in both lists. You can use **retainAll()** to achieve this.

**Example Solution**:

java

Copy

import java.util.\*;

public class ListIntersection {

public static <T> List<T> findIntersection(List<T> list1, List<T> list2) {

List<T> intersection = new ArrayList<>(list1); // Copy list1

intersection.retainAll(list2); // Retain only common elements

return intersection;

}

public static void main(String[] args) {

List<Integer> list1 = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

List<Integer> list2 = new ArrayList<>(Arrays.asList(3, 4, 5, 6, 7));

List<Integer> result = findIntersection(list1, list2);

System.out.println(result); // Output: [3, 4, 5]

}

}

* **Explanation**: retainAll() removes all elements from the list that are not contained in the specified collection. It modifies the original list.

**35. Given a List<Integer>, how would you remove all elements greater than a specified threshold without affecting the order of elements?**

You can use the **removeIf()** method to filter the list based on the threshold condition.

**Example Solution**:

java

Copy

import java.util.\*;

public class RemoveGreaterThanThreshold {

public static void removeGreaterThanThreshold(List<Integer> list, int threshold) {

list.removeIf(num -> num > threshold);

}

public static void main(String[] args) {

List<Integer> list = new ArrayList<>(Arrays.asList(1, 5, 8, 3, 6, 2));

removeGreaterThanThreshold(list, 5);

System.out.println(list); // Output: [1, 5, 3, 2]

}

}

* **Explanation**: The removeIf() method removes elements from the list that satisfy the given predicate (in this case, numbers greater than the threshold).

**32. How would you implement custom sorting logic in a TreeMap using a custom comparator?**

You can implement a TreeMap with a custom comparator by providing a comparator at the time of creation. The comparator will define the custom sorting logic for the keys in the map.

**Example Solution**:

java

Copy

import java.util.\*;

public class TreeMapCustomSort {

public static void main(String[] args) {

// Custom comparator to sort keys in descending order

Comparator<String> customComparator = (key1, key2) -> key2.compareTo(key1);

TreeMap<String, Integer> map = new TreeMap<>(customComparator);

map.put("apple", 5);

map.put("banana", 3);

map.put("orange", 8);

System.out.println("TreeMap with custom sorting: " + map);

}

}

* **Explanation**: The custom comparator compares two keys and sorts them in descending order. When creating the TreeMap, you pass this comparator to control the sorting of the keys. This is useful when you need a specific sorting order that isn’t the natural ordering of the keys.

Foreginkey canot be null if optional = false by default its false

@OneToOne(optional = true) // Optional relationship it allow null

@JoinColumn(name = "passport\_id")

To get data from both the end

* **@JsonManagedReference**: Marks the forward reference, allowing serialization of the Person entity and its Passport.
* **@JsonBackReference**: Marks the back reference to avoid infinite recursion when serializing the Passport entity.

@OneToOne(fetch = FetchType.LAZY) // Lazy loading

**@AutoConfigurationPackage**:  
This annotation is used to ensure that Spring Boot can automatically detect and register your main application package for autoconfiguration. It marks the base package for component scanning, ensuring that any classes in that package are included in the auto-configuration process.

* + @Component: A generic stereotype annotation indicating that the class is a Spring-managed bean. It's the most general-purpose annotation and can be used for any class that you want Spring to manage.
  + @Repository: A specialization of @Component. It indicates that the class is a DAO (Data Access Object) and helps with exception translation, i.e., it converts database-related exceptions into Spring's DataAccessException.
  + One difference is that @Autowired can be used with required = false to allow for optional dependencies, whereas @Inject does not have this feature.

Both @Autowired (Spring-specific) and @Inject (Java standard) are used to inject dependencies into Spring beans.

0000000000000000000000000000000000000000000000000

In Spring Boot, @PersistenceContext is an annotation used to inject an EntityManager instance into a class, use with dao and service layer jpa

**6. How does Spring Boot handle the initialization order of beans when using @Component, @Service, and @Repository annotations?**

* **Answer:**  
  Spring does not guarantee the initialization order of beans unless you explicitly define it. By default, Spring Boot will initialize beans in the order they are encountered during the application context creation, based on their dependencies. If a bean is dependent on another bean, Spring will ensure that the dependent bean is initialized first.

However, you can control the order of bean initialization using the @DependsOn annotation or by specifying a @PostConstruct method to execute certain logic after the bean has been created and dependencies have been injected.

Example of using @DependsOn:

java

Copy

@Component

@DependOn(“anatherBean”)

public class MyBean {

// Will be initialized after 'anotherBean' is initialized

}

List<String> nonNullItems = items.stream()

.filter(Objects::nonNull) // Filter out null values

.collect(Collectors.toList());

List<String> result = items.stream()

.map(Optional::ofNullable) // Wrap each element in Optional

.map(opt -> opt.orElse("default")) // Provide a default value for null

.collect(Collectors.toList());

1. **What is the difference between the Heap and Stack in Java?**
   * **Explanation**:
     + **Heap**: It is a part of the JVM memory where objects are allocated dynamically at runtime. The heap is shared among all threads, and its memory is managed by the garbage collector.
     + **Stack**: It stores method frames, local variables, and references to objects. Each thread has its own stack, and memory is freed automatically when methods return.
2. **Your application is experiencing long pause times due to garbage collection. How would you troubleshoot and resolve this issue?**
   * **Explanation**: To troubleshoot long GC pauses:
     + Review the GC logs to identify if Full GCs or frequent minor GCs are causing delays.
     + Tune heap size (-Xms, -Xmx) to ensure sufficient memory is available for the application.
     + Consider switching to a low-latency GC algorithm like **G1 GC** or **ZGC**.
     + Use profiling tools to find memory leaks or excessive memory usage patterns.
     + Ensure that the application doesn’t unnecessarily allocate a large number of objects.