What is memory management in java And how it is handled?

* Memory management in Java is primarily handled through **automatic garbage collection**, which helps in reclaiming the memory by destroying unused objects.  
  **Heap**: Stores objects and class instances. This is the main area managed by the garbage collector.
* **Stack**: Stores method call frames, local variables, and references to objects in the heap.
* **Method Area (MetaSpace in Java 8+)**: Stores class metadata, static variables, and method definitions.
* **Program Counter (PC) Register**: Keeps track of the current instruction being executed.
* **Native Method Stack**: Used for native (non-Java) method calls.  
    
    
  Q. Explain each best practice to avoid memory leak in java with example of code now.  
    
  **1. Avoid Unintentional Object References (especially in Collections)**
* **Problem**: Objects stored in collections (like List, Map) are not garbage collected if not removed.
* **Best Practice**: Remove unused objects from collections.

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

Map<string, object> cache = new HashMap<>();

public void addToCache(String key, Object value) {

    cache.put(key, value);

}

public void clearCache(String key) {

    cache.remove(key); // Prevents memory leak

}

**[2]Use Weak References for Caches**

**Problem**: Strong references in caches prevent GC.

**Best Practice**: Use WeakHashMap or WeakReference  
  
**What is a Strong Reference?**

In Java, **strong references** are the default type of reference. If an object is strongly referenced, the **Garbage Collector (GC)** will **never** remove it from memory as long as the reference exists.

**🔍 What is a Weak Reference?**

A **weak reference** allows the GC to collect the object **even if it is still referenced**, as long as it's only weakly referenced. This is useful for **caching**, where you don’t want the cache to prevent memory cleanup.

**WeakHashMap vs HashMap**

* HashMap uses **strong references** for keys and values.
* WeakHashMap uses **weak references** for **keys**. If a key is no longer used elsewhere, it can be garbage collected, and the entry is removed from the map.

import java.util.HashMap;

import java.util.WeakHashMap;

public class CacheExample {

    public static void main(String[] args) {

        // Strong reference example

        HashMap<object, string> strongMap = new HashMap<>();

        Object strongKey = new Object();

        strongMap.put(strongKey, "Strong Reference");

        // Weak reference example

        WeakHashMap<object, string> weakMap = new WeakHashMap<>();

        Object weakKey = new Object();

        weakMap.put(weakKey, "Weak Reference");

        // Remove strong references

        strongKey = null;

        weakKey = null;

        // Suggest GC

        System.gc();

        // Wait a bit for GC to run

        try { Thread.sleep(1000); } catch (InterruptedException e) {}

        System.out.println("StrongMap: " + strongMap);

        System.out.println("WeakMap: " + weakMap);

    }

}  
  
ouptput :  
StrongMap: {java.lang.Object@1b6d3586=Strong Reference}

WeakMap: {}

* The strongMap still holds the object.
* The weakMap entry is **gone** because the key was weakly referenced and GC collected it

**When to Use WeakHashMap or WeakReference?**

* **Caches**: You want to store data temporarily and allow GC to clean it up when memory is low.
* **Listeners or Callbacks**: Avoid memory leaks by not preventing GC of unused objects.

--------------------------------------------------------------------------------------------------------  
[3] **Close Resources Properly**

**Problem**: Open streams/sockets hold memory.

**Best Practice**: Use try-with-resources.

try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {

    String line = reader.readLine();

} // reader is auto-closed  
  
[4] **Avoid Static References to Large Objects**

**Problem**: Static fields live for the lifetime of the app.

**Best Practice**: Don’t store large objects statically.

// Bad

public static List bigList = new ArrayList<>();

// Better

public List getList() {

    return new ArrayList<>();

}  
  
[5] **Use Profiling Tools**

**Best Practice**: Use tools like **VisualVM**, **JProfiler**, or **Eclipse MAT** to detect memory leaks.

[6] **Be Careful with Inner Classes**

**Problem**: Non-static inner classes hold reference to outer class.

**Best Practice**: Use static inner classes when possible.

// Bad

class Outer {

    class Inner {

        // Holds reference to Outer

    }

}

// Good

class Outer {

    static class Inner {

        // No reference to Outer

    }

}

How do we implement security in spring boot application?

Implementing **security in a Spring Boot application** typically involves using **Spring Security**, a powerful and customizable authentication and access-control framework  
  
**Key Components of Spring Security**

1. **Authentication** – Verifying who the user is.
2. **Authorization** – Determining what the user is allowed to do.
3. **Password Encoding** – Storing passwords securely.
4. **Role-Based Access Control (RBAC)** – Restricting access based on user roles.
5. **JWT (JSON Web Token)** – For stateless authentication in REST APIs.  
     
   In Java (and specifically in Spring Boot), a **stateless REST API** means that the server does **not store any client session information** between requests. Each request from the client must contain **all the information** needed for the server to process it.

**Basic Steps to Implement Security**

**1. Add Spring Security Dependency**

In pom.xml:  
  
<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>  
---------------------------------------------------------------------------  
2.**Create a Security Configuration Class**

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

    protected void configure(HttpSecurity http) throws Exception {

        http

            .authorizeRequests()

            .antMatchers("/public/\*\*").permitAll() // No auth needed

            .antMatchers("/admin/\*\*").hasRole("ADMIN") // Role-based access

            .anyRequest().authenticated()

            .and()

            .httpBasic(); // or .formLogin() for web apps

    }  
@Override

    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

        auth.inMemoryAuthentication()

            .withUser("user").password(passwordEncoder().encode("password")).roles("USER")

            .and()

            .withUser("admin").password(passwordEncoder().encode("admin")).roles("ADMIN");

    }

    @Bean

    public PasswordEncoder passwordEncoder() {

        return new BCryptPasswordEncoder();

    }

* }  
    
  Explanation of Code:  
  **[1]@EnableWebSecurity:** Enables Spring Security’s web security support.
* **WebSecurityConfigurerAdapter:** A base class that allows you to customize Spring Security by overriding methods like configure(HttpSecurity) and configure(AuthenticationManagerBuilder).  
    
  In Spring Security 6+, WebSecurityConfigurerAdapter is deprecated. You should use SecurityFilterChain beans instead.
* **http.authorizeRequests():** Starts defining access rules for HTTP endpoints.
* **.antMatchers("/public/\*\*").permitAll**(): Allows unrestricted access to any endpoint under /public/.
* **.antMatchers("/admin/\*\*").hasRole("ADMIN"):** Restricts access to /admin/\*\* endpoints to users with the ADMIN role.
* .**anyRequest().authenticated():** All other endpoints require authentication.
* **.httpBasic():** Enables basic authentication (username/password via HTTP headers). You could replace this with .formLogin() for form-based login in web apps.
* **auth.inMemoryAuthentication():** Defines users in memory (not from a database).
* **.withUser("user")...roles("USER"):** Adds a user with username "user", password "password" (encoded), and role "USER".
* **.withUser("admin")...roles("ADMIN"):** Adds an admin user with role "ADMIN".
* @Bean: Registers this method as a Spring bean.
* BCryptPasswordEncoder: A secure password hashing algorithm used to encode passwords.

Spring Security requires passwords to be encoded. This bean ensures that the passwords used in inMemoryAuthentication() are properly hashed.

**What Does inMemoryAuthentication() Mean?**

It tells Spring Security to **store user credentials in memory** (RAM) while the application is running. This is useful for:

* **Testing or prototyping** applications
* **Simple apps** that don’t need persistent user storage
* Avoiding the need for a database or external identity provide
* **How It Works**
* Creates a user named "user" with a password and role "USER"
* Stores this user in an internal memory structure (like a map or list)
* Uses this data to authenticate incoming requests

---------------------------------------------------------------------------------  
[3]**Use JWT for REST APIs (Optional but Recommended)**

* Generate JWT on login.
* Validate JWT on each request.
* Use filters to intercept and validate tokens.

**🛡️ Additional Security Features**

* **CSRF Protection** (enabled by default for web apps).
* **CORS Configuration** for cross-origin requests.
* **OAuth2/OpenID Connect** for third-party login (Google, GitHub, etc.).
* **Method-Level Security** using @PreAuthorize, @Secured.

Here’s a simple example of **method-level security** in a Spring Boot application using @PreAuthorize and @Secured.  
[1]Adding dependency in pom.xml

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-security</artifactId>

</dependency>

**Step 2: Enable Method Security**

In your main application class or a configuration class, add:

@EnableMethodSecurity  // For Spring Security 6+

@Configuration

public class SecurityConfig {

    // You can define custom security rules here if needed

}

[3] **step 3: Use Security Annotations on Methods**

You can now annotate service or controller methods with:

**🔹 @PreAuthorize**

Checks before method execution.

@PreAuthorize("hasRole('ADMIN')")

public void deleteUser(Long userId) {

    // Only ADMIN can delete users

}

**@PostAuthorize**

Checks after method execution.

@PostAuthorize("returnObject.owner == authentication.name")

public Document getDocument(Long id) {

    // Only the owner can access the document

}

[5]**@Secured**

Simpler role-based access.

@Secured("ROLE\_USER")

public void viewProfile() {

    // Only users with ROLE\_USER can view

}

**Step 4: Configure Authentication**

You’ll need to configure users and roles either in-memory, via a database, or using an external provider (like OAuth2, LDAP, etc.).

@Bean

public UserDetailsService userDetailsService() {

    UserDetails user = User.withDefaultPasswordEncoder()

        .username("admin")

        .password("password")

        .roles("ADMIN")

        .build();

    return new InMemoryUserDetailsManager(user);

}

**how to implement jwt token security in spring boot application?**

To implement JWT (JSON Web Token) security in a Spring Boot application, you typically follow these steps:  
  
**Step 1: Add Dependencies**

In your pom.xml (for Maven):(these does not come under spring boot starter security)

<dependency>

    <groupId>io.jsonwebtoken</groupId>

    <artifactId>jjwt-api</artifactId>

    <version>0.11.5</version>

</dependency>

<dependency>

    <groupId>io.jsonwebtoken</groupId>

    <artifactId>jjwt-impl</artifactId>

    <version>0.11.5</version>

    <scope>runtime</scope>

</dependency>

<dependency>

    <groupId>io.jsonwebtoken</groupId>

    <artifactId>jjwt-jackson</artifactId>

    <version>0.11.5</version>

    <scope>runtime</scope>

</dependency>  
  
**Step 2: Create JWT Utility Class**

This class handles token creation and validation.

public class JwtUtil {

    private final String SECRET\_KEY = "your\_secret\_key";

    public String generateToken(UserDetails userDetails) {

        return Jwts.builder()

            .setSubject(userDetails.getUsername())

            .setIssuedAt(new Date())

            .setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \* 10)) // 10 hours

            .signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

            .compact();

    }

    public String extractUsername(String token) {

        return Jwts.parser()

            .setSigningKey(SECRET\_KEY)

            .parseClaimsJws(token)

            .getBody()

            .getSubject();

    }

    public boolean validateToken(String token, UserDetails userDetails) {

        final String username = extractUsername(token);

        return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));

    }

    private boolean isTokenExpired(String token) {

        Date expiration = Jwts.parser()

            .setSigningKey(SECRET\_KEY)

            .parseClaimsJws(token)

            .getBody()

            .getExpiration();

        return expiration.before(new Date());

    }

}

**Step 3: Create JWT Filter**

Intercept requests and validate JWT.

public class JwtRequestFilter extends OncePerRequestFilter {

    @Autowired

    private JwtUtil jwtUtil;

    @Autowired

    private UserDetailsService userDetailsService;

    @Override

    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)

        throws ServletException, IOException {

        final String authHeader = request.getHeader("Authorization");

        String username = null;

        String jwt = null;

        if (authHeader != null && authHeader.startsWith("Bearer ")) {

            jwt = authHeader.substring(7);

            username = jwtUtil.extractUsername(jwt);

        }

        if (username != null && SecurityContextHolder.getContextDetails, null, userDetails.getAuthorities());

                authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

                SecurityContextHolder.getContext().setAuthentication(authToken);

            }

        }

        chain.doFilter(request, response);

    }

}  
  
**Step 4: Configure Security**

@Configuration

@EnableMethodSecurity

public class SecurityConfig {

    @Autowired

    private JwtRequestFilter jwtRequestFilter;

    @Bean

    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        http.csrf().disable()

            .authorizeHttpRequests(auth -> auth

                .requestMatchers("/authenticate").permitAll()

                .anyRequest().authenticated()

            )

            .sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

        http.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);

        return http.build();

    }

}