1. What is Spring Boot used for?
Spring Boot is an open-source, microservice-based Java web framework offered by Spring, particularly useful for software engineers developing web apps and microservices.
Spring Boot is an open-source tool that makes it easier to create microservices and web apps. Spring Boot is a popular choice for developing Spring-based applications because it makes development faster and easier.
It provides a number of features that save you from having to write a lot of boilerplate code, and it includes a number of other features that make it a powerful and flexible framework.
Spring Boot is a framework that makes it easy to create Spring-based applications.
It provides a number of features that make it easier to get started, including auto-configuration and dependency management.

2. Key components of springBoot
Spring Boot is a framework that makes it easy to create Spring-based applications.
It provides a number of features that make development faster and easier, including:

Starters : Starters are a set of pre-configured dependencies that can be used to quickly add features to your application. For example, there are starters for web development, data access, and messaging.

Auto-configuration : Spring Boot can automatically configure many aspects of your application, such as the embedded web server, database connection, and transaction management. This saves you from having to write a lot of boilerplate code.

Command-line interface: Spring Boot provides a command-line interface that can be used to start, stop, and manage your application. This makes it easy to deploy and run your application in production.

Actuator: Spring Boot Actuator provides a set of endpoints that can be used to monitor and manage your application. This information can be used to troubleshoot problems and improve the performance of your application.
In addition to these core features, Spring Boot also includes a number of other features that make it is a powerful and flexible framework for developing Spring-based applications. These features include:

Embedded servers : Spring Boot includes an embedded web server that can be used to run your application without the need for a separate server. This makes it easy to develop and test your application locally.

Database support : Spring Boot provides support for a variety of databases, including MySQL, PostgreSQL, and Oracle. This makes it easy to choose the right database for your application.
Security : Spring Boot includes support for Spring Security, which provides a comprehensive set of security features for your application. This includes features such as authentication, authorization, and auditing.

3. What will increase the performance in Spring Boot, and what techniques are there ?
Improving performance in a Spring Boot application involves various techniques and strategies. Here are some common approaches to enhance performance:

Database Optimization: Optimize database queries by using proper indexing, minimizing the number of queries, and optimizing the data access layer. Techniques like caching (using tools like Redis or Spring's caching abstraction) can also reduce database load.

Caching: Utilize caching mechanisms to store frequently accessed data in memory, reducing the need for repeated computations or database access. Spring provides support for caching through annotations like @Cacheable, @CachePut, and @CacheEvict.

Concurrency: Utilize multithreading and asynchronous processing to handle concurrent requests efficiently. Spring provides features like @Async annotation

Optimized Dependency Injection: Be mindful of the number and size of dependencies injected into beans. Avoid injecting unnecessary dependencies, and consider using constructor injection over field injection for better testability and performance.

Optimized Exception Handling: Implement efficient exception handling mechanisms to avoid unnecessary overhead. Handle exceptions at an appropriate level in the application stack and avoid logging exceptions excessively.

4. What is SpringBoot Security
Spring Security is a powerful and customizable authentication and access control framework provided by the Spring ecosystem. When used with Spring Boot, it becomes Spring Boot Security.
Key Features of Spring Boot Security:
Authentication: Spring Boot Security provides comprehensive support for various authentication mechanisms, including form-based login, HTTP Basic authentication, OAuth, JWT (JSON Web Tokens), and more.
Authorization: Spring Boot Security enables fine-grained access control and authorization based on roles, permissions, expressions, or custom logic. You can configure access control at both URL level and method level using annotations or XML configuration.
Session Management: It offers features for managing user sessions, including session fixation protection, concurrent session control, session timeout configuration, and more.

5. Use of Spring-web dependency
The Spring-Web dependency is a Spring Boot starter that provides the dependencies needed to create web applications.
It includes the Spring MVC framework, which is a popular framework for developing web applications in Java.

6. What is spring actuator, what the use
Spring Boot Actuator is a sub-project of Spring Boot that provides a variety of production-ready features for monitoring and managing Spring Boot applications.
It exposes various information about the application, such as its health, metrics, environment variables, and thread dumps, through HTTP endpoints or JMX beans.
This information can be used to monitor the application's performance, identify and troubleshoot issues, and make informed decisions about its management.

7. What is JPA, what's the use
Java Persistence API (JPA) is a framework that manages relational data in Java applications.
It's a Java specification that provides standards and functionality for object-relational mapping (ORM) tools. JPA is part of the Java Enterprise Edition (Java EE) platform.
JPA allows developers to map Java objects to relational database tables without writing low-level SQL code. It provides a convenient way to interact with databases.

8. Difference between @Query and @NativeQuery
The @Query annotation is used in Spring Data JPA to define custom queries.
It allows developers to define JPQL (Java Persistence Query Language) or native SQL queries for repository methods.
The @Query annotation can also be used with native queries.
To define a native query, you can use the @Query annotation in combination with the nativeQuery flag set to true.

```
@Query("SELECT u FROM User u WHERE u.status = 1")
List<User> findActiveUsers();
@Query(value = "SELECT * FROM USERS u WHERE u.status = 1", nativeQuery = true)
List<User> findActiveUsersNative();
```

9. Lazy loading and Eager loading in spring boot
While lazy loading delays the initialization of a resource, eager loading initializes or loads a resource as soon as the code is executed.
Eager initialization also known as eager loading, this strategy initializes beans during the application's startup, regardless of whether they are immediately needed or not.
Lazy initialization also known as lazy loading, this strategy delays the creation of beans until they are explicitly requested.

```
@Eager
@Bean
public DataSource dataSource() {
return new DriverManagerDataSource();
}
@Lazy
@Bean
public MyService myService() {
return new MyServiceImpl();
}
```

10. What is Circuit breaker in java microservice.
A circuit breaker is a design pattern used to enhance the resilience and fault tolerance of distributed systems by managing failures in communication between services. It's especially valuable in scenarios where services are communicating over unreliable networks, such as the internet, where failures are more common.

11. What is JIRA
JIRA is a tool developed by Australian Company Atlassian. This software is used for bug tracking, issue tracking, and project management. The JIRA full form is actually inherited from the Japanese word “Gojira” which means “Godzilla”.
Jira is an agile project management tool used by teams to plan, track, release and support world-class software with confidence. It is the single source of truth for your entire development lifecycle

12. JDBC and steps in java
Java Database Connectivity (JDBC) is an API that enables Java applications to connect to and interact with databases. JDBC is included in the Java Platform, Standard Edition (Java SE).
Here are the steps involved in using JDBC in a Java application:
1. Import the necessary JDBC packages.
2. Register the JDBC driver for the database you want to connect to.
3. Create a Connection object to establish a connection to the database.
4. Create a Statement object to execute SQL statements.
5. Execute the SQL statement and process the results.
6. Close the Statement and Connection objects.

JDBC is a powerful tool that can be used to develop a wide range of database-driven applications.

```
import java.sql.*;
public class JDBCExample {
public static void main(String[] args) throws Exception {
// Register the JDBC driver
Class.forName("com.mysql.cj.jdbc.Driver");
// Create a Connection object
Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/test",
"root", "password");
// Create a Statement object
Statement statement = connection.createStatement();
// Execute the SQL statement
ResultSet resultSet = statement.executeQuery("SELECT * FROM users");
// Process the results
while (resultSet.next()) {
System.out.println(resultSet.getString("name"));
}
// Close the Statement and Connection objects
statement.close();
connection.close();
}
}
```

13. What is mvc in java
MVC stands for Model-View-Controller. It is a software design pattern that separates the user interface (view), data (model), and application logic (controller).
This separation of concerns makes it easier to develop and maintain applications, as each components can be changed or updated independently.
In Java, MVC is often used to develop web applications. The view is typically implemented using JavaServer Pages (JSP) or servlets, the model is implemented using JavaBeans, and the controller is implemented using servlets or Spring MVC.

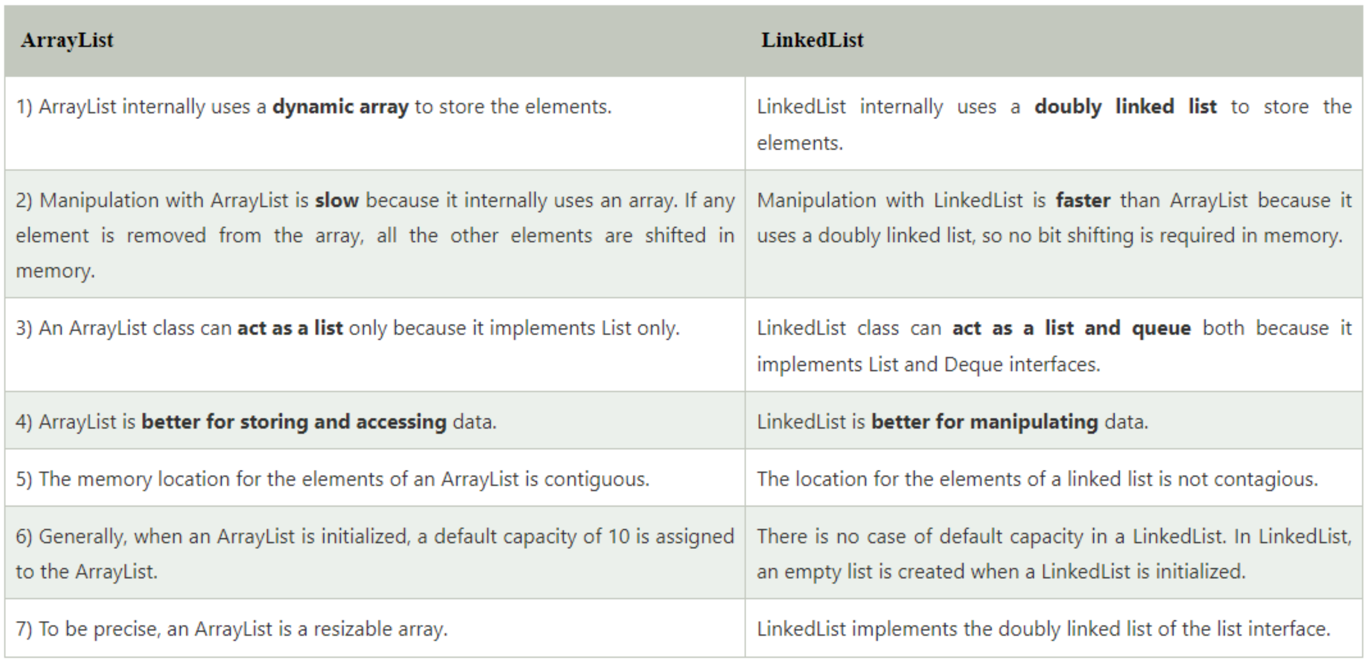
14. What is kafka in spring boot
Apache Kafka is an open-source, distributed streaming platform for processing and streaming events. It was originally developed by LinkedIn engineers and is now part of the Apache Software Foundation.
Kafka is designed to handle large amounts of data in real-time, making it suitable for applications that require high throughput, fault tolerance, and real-time data streaming.

15. How to manage logs in Spring Boot application?
It is possible to activate the debug and trace level by starting the application with –debug flag or –trace flag as follows:
java -jar target/log-0.0.1-SNAPSHOT.jar –debug. ...
debug=true. ...
spring.output.ansi.enabled=always.
logging.file.path=logs/ ...
Note: For @Slf4j: org.slf4j.Logger.

16. What is the difference between @RestController and @controller annotation?
@Controller is used to mark classes as Spring MVC Controller. @RestController annotation is a special controller used in RESTful Web services, and it's the combination of @Controller and @ResponseBody annotation. It is a specialized version of @Component annotation.

17. Why map interface does not extend collection interface
The main reason Map doesn't extend the Collection interface is that the add(E e) method of the Collection interface doesn't support the key-value pair like Map interface's put(K, V) method. It might not extend the Collection interface but still is an integral part of the Java Collections Framework.

18. Difference Between ArrayList and LinkedList
ArrayList and LinkedList both implement the List interface and maintain insertion order. Both are non-synchronized classes.
However, there are many differences between the ArrayList and LinkedList classes that are given below.



ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.
5) The memory location for the elements of an ArrayList is contiguous.	The location for the elements of a linked list is not contagious.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.	There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.
7) To be precise, an ArrayList is a resizable array.	LinkedList implements the doubly linked list of the list interface.

19. Difference Between hashmap and hasTable
HashMap and HashTable are both key-value storage classes in Java. HashMap is non-synchronized, making it faster for single-threaded tasks, while HashTable is inherently synchronized, providing thread safety.
HashTable doesn't allow any null keys or values, but HashMap lets you have one null key and several null values

20. What is filter in spring boot and use
A filter in Spring Boot is a component that intercepts and manipulates HTTP requests and responses.
It can be used to perform various tasks such as authentication, logging, debugging, and so on.
Filters are a part of the Servlet API, and they are executed for each incoming request and outgoing response.
Here are some common use cases for filters in Spring Boot:
Authentication : Filters can be used to validate the credentials of a user before they are allowed to access a particular resource.
Logging : Filters can be used to log all incoming and outgoing requests, which can be useful for debugging and troubleshooting purposes.
Input validation : Filters can be used to validate the input data of a request before it is processed by the controller.

21. How to read application.properties data into the class
In Spring Boot, you can read properties from an application.properties file into your classes using the @Value annotation

```
# application.properties
app.name=MyApp
app.version=1.0
```

```
@Value("${app.name}")
private String appName;
```

22. To create an immutable class in Java
you typically follow these steps:
1.Declare the class as final to prevent inheritance.
2.Make all fields private and final.
3.Remove any setters for the fields.
4.Provide constructors to initialize all the fields.

```
public final class ImmutableClass {
private final String name;
private final int age;
public ImmutableClass(String name, int age) {
        this.name = name;
        this.age = age;
}
public String getName() {
        return name;
}
public int getAge() {
        return age;
}
}
```

23. Difference between monolithic and springBoot
Monolithic Architecture
Monolithic architecture is a traditional software development approach where the application is built as a single, self-contained unit.
All components, functionalities, and services are tightly integrated and deployed together.
Monolithic architecture is characterized by its simplicity in development and deployment, as everything is managed within a single codebase and deployment unit.
However, monolithic architecture can become complex and challenging to update or change over time.
It can also be difficult to scale monolithic applications, especially when certain components need to handle a large volume of traffic.

Spring Boot
Spring Boot is a Java framework that makes it easy to create Spring-based applications.
It provides a wide range of features and functionality out of the box, including auto-configuration, embedded web servers, and starters for popular technologies.
Spring Boot can be used to build both monolithic and microservices-based applications.
However, it is particularly well-suited for microservices development, as it provides a number of features that make it easy to create, deploy, and manage independent services.

24. Real time scenario of Thread in springBoot
In Spring Boot, threads can be managed using the ExecutorService interface. This interface provides a number of methods for creating, managing, and executing threads.
Here is an example of how to use the ExecutorService interface to create and execute a thread in Spring Boot :

```
ExecutorService executorService = Executors.newFixedThreadPool(10);
executorService.submit(() -> {
// Do something
});
```

This code will create a new thread pool with 10 threads. The submit() method will then submit a new task to the thread pool. The task will be executed by one of the threads in the thread pool.

25. How to achieve MultiThread in SpringBoot
One common approach is to use Spring's @Async annotation to mark methods that should be executed asynchronously in separate threads
Firstly, make sure to enable async support in your Spring Boot application by adding @EnableAsync annotation in your main class or configuration class

```
@SpringBootApplication
@EnableAsync
```

Then, you can create a service class with methods annotated with @Async to run asynchronously:

```
@Async
   public void asyncMethod1() {
       System.out.println("Async method 1 started by thread: " +
Thread.currentThread().getName());
       // Your asynchronous logic here
   }

   @Async
   public void asyncMethod2() {
       System.out.println("Async method 2 started by thread: " +
Thread.currentThread().getName());
       // Your asynchronous logic here
   }
```

Now, whenever asyncMethod1() or asyncMethod2() is called, Spring will execute them asynchronously in separate threads.
You can inject and use MyService in your controllers or other Spring-managed beans to call these methods asynchronously.

Remember to include the necessary dependencies in your pom.xml or build.gradle for Spring's asynchronous support (spring-boot-starter-async for Maven,
or spring-boot-starter-async for Gradle).

26. What is java8 and its features
Java8 is a new version of Java and was released by Oracle on 18 March 2014.
Java provided support for functional programming, new Java 8 APIs, a new JavaScript engine, new Java 8 streaming API, functional interfaces, default methods, date-time API changes, etc
Top Java 8 Features With Examples

❖ Functional Interfaces And Lambda Expressions.
❖ forEach() Method In Iterable Interface.
❖ Optional Class.
❖ Default And Static Methods In Interfaces.
❖ Java Stream API For Bulk Data Operations On Collections.
❖ Java Date Time API.
❖ Collection API Improvements.
❖ Java IO Improvements.

27. What is Lambda Expression
Lambda expression is an anonymous function
i.e.
1. Not having any name
2. Not having any datatype
3. Not having any modifier

Example 1 :
```
public void sayHello(){
        System.out.println("Hello !");
}
```

Remove modifier , datatype , name and place arrow
()->{System.out.println("Hello !");}

Example 2 :
```
private int add(int a , int b){
        return a+b;
}
```

Remove modifier , datatype , name and place arrow
(int a , int b)->{return a+b;}

we can remove datatype also complier auto guess datatype, and if we having single statement in body remove curly brackets   (a,b)->return a+b;

Benefits of Lambda expression
1. To enable functional programming
2. To make code concise,maintainable and easy readable
3. Jar file size reduce
4. Eliminate of shadow variable

28. What is Functional Interface

Interface having single abstract method(SAM) but can have any number of default and static methods.
We can invoke lambda expression by using functional interface
Here are some examples of functional interfaces in Java 8:
Predicate: This interface takes one argument and returns a boolean value.
Consumer: This interface takes one argument and returns nothing.
Function: This interface takes one argument and returns a value.
Supplier: This interface takes no arguments and returns a value.

```
//Allows only one Abstract method and any numbers of default and static methods
@FunctionalInterface
public interface MyInterface {
        public void sayHello();  //Abstract Method

        default void sayBye() {
        }

        public static void sayOk() {
        }
}
```

Why we use @FunctionalInterface annotation
1. It restrict Interface to be a Functional Interface.

29. What is default method inside Interface
before java 1.8, Interface allows only abstract method
but in java 1.8 we can create default method inside Interface

For Example :
Previous we writing Interface like this
```
public interface MyInterface {
        public void sayHello();  //Abstract Method
}
```
But now with new feature we can write default method with body description
```
public interface MyInterface {
	default void sayBye() {
		System.out.println("Hello !");
	}
}
```

30. What is static method inside Interface
we can define static methods in Interface using static keyword

For Example :
```
public interface MyInterface {
	static void sayBye() {
		System.out.println("Hello !");
	}
}
```
If we want to call static method in Class from Interface we need use InterfaceName
```
interface DemoInterface {
	static void sayBye() {
		System.out.println("Hello !");
	}
}
class MyInterface implements DemoInterface{
	public static void main(String[] args) {
		DemoInterface.sayBye();
	}
}
```

31. How to use lambda expression in functional Interface

Example 1 :
```
@FunctionalInterface
interface Employees {
	String getName();
}

public class Engineers{
	public static void main(String[] args) {
		Employees e = ()->"Computer Engineer";  //Lambda expression
		System.out.println(e.getName());

		Employees e1 = ()->"Civil Engineer";  //Lambda expression
		System.out.println(e1.getName());
	}
}
```

Example 2 : In this Runnable Is an Functional Interface which having one abstract method in it that is run().

```
public class Engineers{
	public static void main(String[] args) {

		Runnable runnable1 = ()-> System.out.println("Hello : "); //Lambda expression
		Thread thread1 = new Thread(runnable1);
		thread1.run();

		Runnable runnable2 = ()->{
			for(int i=1;i<=10;i++)
			{
				System.out.println("Bye : "+i);
			}
		};
		Thread thread2 = new Thread(runnable2);
		thread2.run();
	}
}
```

Example 3 : How we use lambda expression in Comparator Interface

```
List<Integer> list = new ArrayList<>();
list.add(20);list.add(10);list.add(5);list.add(90);

//It will sorting in ascending order
Collections.sort(list);
System.out.println(list);

//If we need to sort in descending order we need to use Comparator which is also an function
Interface
Collections.sort(list,(a,b)->b-a); // here we used Lambda expression in Comparator Functional
Interface
System.out.println(list);
```

32. What is Stream in java 8

Stream is a collections of data like Arrays,String,List,Integer,Objects,...etc
To perform an operation on Stream ,first we need to convert data into stream.
For Example :

```
int[] array={2,4,45,78,5}; Arrays.stream(array);
String[] array={"rg","ad","lskw","sadkl"}; Arrays.stream(array);
List<Integer> list = Arrays.asList(1,5,3,76,78,6,9,6);		list.stream()
```

Now will perform an operation on Stream
//Use of Stream and use of filter(),map() and
distinct(),sorted(),limit(),min(),max(),count(),parallelStream

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamExample{
        public static void main(String[] args) {

List<Integer> list = Arrays.asList(1,5,3,76,78,6,9,6,98,67,100,56,48,12,9432);
System.out.println("Original List : "+list);
List<Integer> mapfilterList = list.stream()
                .filter(n->n%2==0)      // It filter an stream performs lambda expression
                .map(x->x/2)            // It performs lambda expression
                .distinct()             // It gives distinct output
                .sorted((a,b)->(b-a))   //Sort in desc order we used comparator lambda expression
                .limit(5)               // It shows only 10 data
                .skip(1)                // It skip last 1 data
                .collect(Collectors.toList());   // It collects stream and convert into List
System.out.println("MapfilterList : "+mapfilterList);

int min = list.stream().min((a,b)->a-b).get(); // It gives minimum value, we used comparator
lambda expression
int max = list.stream().max((a,b)->a-b).get(); // It gives maximum value, we used comparator
lambda expression
long count = list.stream().count();             // It gives length/counts of elements

List<Integer> parallelStream = list.parallelStream().collect(Collectors.toList()); //It divide your
data into chunks and run parallelly
System.out.println("Min : "+min+", Max : "+max+" , Count: "+count);
System.out.println("Parallel Stream : "+parallelStream);

        }
}
```

OUTPUT :

```
Original List : [1, 5, 3, 76, 78, 6, 9, 6, 98, 67, 100, 56, 48, 12, 9432]
MapfilterList : [50, 49, 39, 38]
Min : 1, Max : 9432 , Count: 15
Parallel Stream : [1, 5, 3, 76, 78, 6, 9, 6, 98, 67, 100, 56, 48, 12, 9432]
```

33. What is Serialization and deserialization in java

Serialization and deserialization are processes used to convert Java objects into a format that can be easily stored, transmitted, or reconstructed later.

Serialization: Serialization is the process of converting an object into a byte stream. This byte stream can then be saved to a file, sent over a network, or stored in a database. In Java, serialization is achieved by implementing the Serializable interface. When an object is serialized, all of its non-transient fields are converted into a byte stream.

Deserialization: Deserialization is the process of reconstructing an object from its serialized byte stream. In Java, deserialization is done using the ObjectInputStream class.

34. What is String constant pool in java.
String constant pool is a special area in the Java heap memory where String literals are stored. When you create a String object using a string literal (e.g., "hello"), Java checks if the same string literal already exists in the String constant pool. If it does, Java returns a reference to the existing string object from the pool rather than creating a new one. This allows Java to conserve memory by reusing common string values.

35. What is Optional Class and why it is used.
Optional class is used to remove nullPointerException
Suppose we retriing a name from DB by using id, where we know it should return null also
if we print/retrive the value String str = getName(id=2); and str.toUpperCase() , here we performing operation on Null value
it give nullPointerException to avoid this we use Optional class

```
public class StreamExample{
        public static void main(String[] args) {
                Optional<String> name = geName(2);
                if(name.isPresent()) {
                        System.out.println(name.get());
                }
                name.ifPresent(System.out::println);
        }
        private static Optional<String> geName(int i) {
                String name="Ram";
                return Optional.of(name);
        }
}
OutPut : Ram
```

36. Exceptions and what is global exception
In Spring Boot, exceptions are errors that can occur during the execution of an application. They are typically handled using the @ExceptionHandler annotation, which allows you to specify a method that will be invoked when a particular exception is thrown.
You can also use the @ControllerAdvice annotation to handle exceptions globally.

In Java, an exception is an event that disrupts the normal flow of the program's instructions during execution. Exceptions are typically caused by errors in the program's logic, unexpected conditions, or external factors such as input/output errors or resource exhaustion. When an exception occurs, it can be handled by the program to gracefully recover from the error condition.

Java has a built-in mechanism for handling exceptions using try-catch blocks
Global Exception: There is no explicit concept of a "global exception" in Java. However, you can define a catch block at a higher level in the call stack to catch exceptions that are not caught

by catch blocks in lower levels.

In Java, there isn't a specific concept called "global exception". However, the term "global exception handling" can refer to the practice of having a centralized mechanism to handle uncaught exceptions that occur anywhere within the application. This is often achieved by setting a default uncaught exception handler using Thread.setDefaultUncaughtExceptionHandler() or by defining a catch block at a higher level in the call stack to catch exceptions that are not caught by catch blocks in lower levels, as I mentioned earlier.

For example:

```
try {
// Code that may throw an exception
} catch (Exception e) {
// Global exception handling code
}
```

37. What is synchronization in java ?
synchronization is a mechanism that allows multiple threads to access shared resources concurrently without causing data corruption or inconsistency. When multiple threads try to access and modify shared data simultaneously, it can lead to race conditions, where the final outcome depends on the timing and interleaving of thread execution. Synchronization prevents such issues by ensuring that only one thread can access a shared resource at a time.

Synchronized Methods: You can declare methods as synchronized, which means only one thread can execute the synchronized method for a given object at any given time. Other threads attempting to execute the same synchronized method will be blocked until the first thread completes its execution.

```
public synchronized void synchronizedMethod() {
// Synchronized method body
}
```

Reentrant Locks: Java provides the ReentrantLock class from the java.util.concurrent.locks package, which allows for more flexible locking mechanisms compared to synchronized methods and blocks. It supports features like fairness, interruptible lock acquisition, and lock timeouts.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class MyClass {
private final Lock lock = new ReentrantLock();
public void performTask() {
        lock.lock();
        try {
        // Critical section
        } finally {
        lock.unlock();
```

```
    }
}
```

38. What is concurrent modification exception in java
In Java, a ConcurrentModificationException is an exception that occurs when a collection is modified concurrently (i.e., while it's being iterated) and such modification is not allowed or expected by the iterator.
This exception typically occurs in scenarios where you are iterating over a collection (like ArrayList, HashMap, etc.) using an iterator (or a for-each loop), and another thread modifies the collection structurally (i.e., adds or removes elements) while the iteration is in progress.

39. What is ConcurrentHashMap ?
ConcurrentHashMap is a class in the Java Collections Framework that provides a concurrent and thread-safe implementation of the Map interface. It allows multiple threads to access and modify the map concurrently without causing data corruption or inconsistency.
ConcurrentHashMap achieves this concurrency by dividing the map into segments, each of which can be independently locked.

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.Map;
public class Main {
public static void main(String[] args) {
Map<String, Integer> concurrentMap = new ConcurrentHashMap<>();
concurrentMap.put("One", 1);
concurrentMap.put("Two", 2);
concurrentMap.put("Three", 3);
System.out.println(concurrentMap.get("Two")); // Output: 2
}
}
```

ConcurrentHashMap is commonly used in multi-threaded environments where multiple threads need to access and modify a shared map concurrently. It provides better scalability and performance compared to traditional synchronized maps like HashMap when used in concurrent scenarios.