**Interview questions**

**1. Core Java Questions:**

**Q1: What are the key features of Java?**

Sample Answer:

* Platform-Independent: Java uses the JVM (Java Virtual Machine) to enable cross-platform functionality. Write once, run anywhere (WORA).
* Object-Oriented: Java supports OOP principles like inheritance, encapsulation, and polymorphism.
* Multithreading: Java has built-in support for multi-threading with the Thread class and Runnable interface.
* Automatic Memory Management: Java has a garbage collector that automatically deallocates memory.

**Q2: What is the difference between ArrayList and LinkedList in Java?**

Sample Answer:

* ArrayList: Implements a dynamic array. It provides faster random access (O(1)) but slower insertion/removal at arbitrary positions (O(n)).
* LinkedList: Implements a doubly linked list. It offers faster insertion/removal (O(1) for head and tail), but slower random access (O(n)).

**Q3: Explain the difference between HashMap and Hashtable.**

Sample Answer:

* HashMap: Not synchronized, so it’s faster but not thread-safe. Allows one null key and multiple null values.
* Hashtable: Synchronized and thread-safe, but slower. It does not allow null keys or values.

**2. Object-Oriented Programming (OOP):**

**Q1: Can you explain the four pillars of OOP?**

Sample Answer:

* Encapsulation: Wrapping data (fields) and methods into a single unit (class) and restricting access using access modifiers (private, protected).
* Abstraction: Hiding implementation details and showing only necessary functionality, often through interfaces or abstract classes.
* Inheritance: Mechanism where one class inherits properties and behavior (methods) from another. For example, class Dog extends Animal.
* Polymorphism: Ability for methods to take many forms, such as method overloading (compile-time) or overriding (runtime polymorphism).

**Q2: What is method overloading and method overriding?**

Sample Answer:

* Method Overloading: Defining multiple methods with the same name but different parameter lists within the same class (compile-time polymorphism).
* Method Overriding: Redefining a method in a subclass that is already defined in the parent class, to provide a specific implementation (runtime polymorphism).

**3. Multithreading & Concurrency:**

**Q1: What is the difference between Runnable and Thread?**

Sample Answer:

* Runnable: It is an interface that is implemented by a class to define the task that needs to be executed in a thread. You pass a Runnable instance to a Thread object.
* Thread: A class that represents a thread. You can either extend the Thread class or implement the Runnable interface to create a thread. Runnable is preferred since Java doesn’t support multiple inheritance (you can implement multiple interfaces, but extend only one class).

**Q2: What are synchronized blocks in Java?**

Sample Answer: The synchronized keyword is used to control the access of multiple threads to a shared resource. It ensures that only one thread at a time can access a method or a block of code marked with synchronized. This helps prevent thread interference and memory consistency errors.

**4. Java Collections Framework:**

**Q1: What is the difference between List, Set, and Map?**

Sample Answer:

* List: An ordered collection of elements that allows duplicate values (e.g., ArrayList, LinkedList**).**
* Set: A collection that does not allow duplicates and is unordered (e.g., HashSet, TreeSet).
* Map: A collection of key-value pairs. Each key is unique, but values can be duplicated (e.g., HashMap, TreeMap).

**Q2: How does HashMap work internally in Java?**

Sample Answer: HashMap uses a combination of an array and linked lists (or red-black trees). When you insert an entry, the hashCode() method generates a hash code that determines the index in the array. If two keys generate the same hash code (a collision), their entries are stored in a linked list at that index. Since Java 8, if the number of elements in a bucket becomes too large, the linked list is converted into a red-black tree for faster lookups.

**5. Java Frameworks (Spring):**

**Q1: What is Spring Boot, and why is it used?**

Sample Answer: Spring Boot is an extension of the Spring Framework that simplifies the development of stand-alone, production-ready Spring applications. It removes the need for extensive XML configuration by providing auto-configuration. Spring Boot is used for creating microservices, REST APIs, and enterprise applications with minimal setup.

Q2: Explain dependency injection (DI) in Spring.

Sample Answer: Dependency Injection (DI) is a design pattern in which an object’s dependencies (such as another class or service) are injected at runtime rather than being instantiated within the class itself. In Spring, this is done via constructor injection or setter injection, which makes the code more flexible, testable, and maintainable.

**6. Database & ORM (Hibernate):**

**Q1: What is Hibernate, and why do we use it?**

Sample Answer: Hibernate is an ORM (Object Relational Mapping) framework that simplifies database operations by mapping Java objects to database tables. It eliminates the need for complex SQL queries, instead allowing developers to interact with the database using simple Java methods. Hibernate supports automatic table generation and handles data persistence efficiently.

Q2: What are the benefits of using JPA (Java Persistence API)?

Sample Answer: JPA is a specification for accessing, persisting, and managing data between Java objects and relational databases. Benefits include:

* Standardized API: It provides a unified API for different ORM frameworks like Hibernate.
* Simplified Code: You don’t need to write explicit SQL queries, as JPA translates Java method calls into SQL queries.
* Portability: JPA makes your code portable across different ORM implementations.

**7. Java 8 Features:**

**Q1: What are the new features introduced in Java 8?**

Sample Answer:

* Lambda Expressions: Introduced functional programming in Java, allowing concise code by passing functions as parameters.
* Streams API: Allows processing sequences of elements (like a collection) in a functional style, making tasks like filtering, mapping, and reducing much easier.
* Default Methods: Interfaces can now have default implementations, making it easier to add methods to interfaces without breaking the existing code.
* Optional Class: Avoids NullPointerException by providing a container object that may or may not contain a value.

**Q2: What is a lambda expression in Java?**

Sample Answer: A lambda expression is a concise way to represent an anonymous function (a method without a name) in Java. It is used to implement functional interfaces and can be passed as parameters to methods. For example:

java

Copy code

List<Integer> numbers = Arrays.asList(1, 2, 3);

numbers.forEach(n -> System.out.println(n));

**8. Miscellaneous Questions:**

**Q1: What is garbage collection in Java?**

Sample Answer: Garbage collection is the process by which Java automatically deallocates memory used by objects that are no longer referenced. The Garbage Collector runs in the background and frees up memory to prevent memory leaks.

**Q2: What is a ClassLoader in Java?**

Sample Answer: The ClassLoader is a part of the Java Runtime Environment that dynamically loads Java classes into memory when they are needed. It ensures that classes are loaded in the right order and prevents duplication of class definitions.

**Behavioral Questions:**

**Q1: Can you describe a challenge you faced in your last project and how you solved it?**

Sample Answer:

"In the Online Bookstore project, we faced performance issues when retrieving large sets of book data. The application became slow during peak traffic. After analyzing the database, I realized that the SQL queries could be optimized. I introduced pagination to limit the data retrieved in each request and used Redis for caching frequent queries. This reduced response times by 20%, and the system performed better under heavy load."

**Q2: How do you handle tight deadlines or pressure?**

Sample Answer:

"When facing tight deadlines, I prioritize tasks based on their impact on the project and break them down into smaller, manageable parts. I also communicate clearly with the team, providing status updates regularly to ensure we’re on track. I find that working in a focused, organized way under pressure helps me stay productive without co\*mpromising quality."

**Comparable vs Comparator :**

When you want to sort the objects then you have to follow these procedures

1. Your class must implement the comparable and must override the compareTo method. It allows nature order

Or

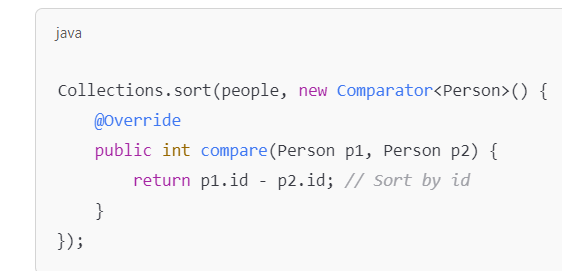
1. You have to create a comparator and that will ne passed to collections.sort () method. It allows custom order
2. Refer below

 **Implement the Comparable interface** in your class, which defines the "natural ordering" of the objects.

 **Provide a Comparator** when calling Collections.sort(), which allows you to define a custom sorting logic externally without modifying the class.

**Note**:so when you want to perform sort method you have to implement the comparable interface or you can provide comparator inside collections.sort method

**Ex**: **Comparator**



**Ex: comparable**

**A screen shot of a computer code

Description automatically generated**

**By using comparator we can sort the list on multiple conditions or custom conditions like we can sort it by name or we can sort it by id or by city.**

**Ex :**

**A screenshot of a computer program

Description automatically generated**

**Note:** Comparable is part of the class itself, and you can only define one sorting rule when you implement the compareTo() method in the Comparable interface. You can't define multiple sorting rules (like by id, name, or age) using Comparable alone.

**Comparable interface** allows you to define a natural order by implementing compareTo().

 **Collections.sort(list)** can be used if the list elements implement Comparable.

 **Comparator** can still be used with Collections.sort(list, comparator) for custom or alternate sorting criteria.

**Working of equals() and hashcode()**

So hashcode() is for giving a hash value (integer) for every object. On the basis of hash value objects are stored in buckets.

It gives the same hash value for same objects so that objects are stored in same buckets. And some times hashcode gives the same hash value for different objects too. This is called hash collision.

So equals() method is used here , equals method compares the objects which are having same hashcode or which are in same bucket by comparing the content value in object if they are equal then only one object is stored in the bucket otherwise two objects are stored in the bucket.

**Default Behaviour of equals()**

The default behavior of equals() is compaing only references. i.e if two references(addresses) are pointing to same objects.

Here in the list we are creating objects so every object has a different reference so list allows duplicates too. So by giving a custom implementation for equals() and write a hash code method we can remove duplicates in list.

These two methods are internally implemented in hashset and hashmap that’s the reason hashmap , hashset haven’t allow any duplicates. But which is not implemented in List so we have to implement manually.

**Collectors.groupingBy() in Java**

In Java, groupingBy() is a powerful **collector** method provided by the Collectors class in the java.util.stream package.

It is used to group elements of a stream based on a classifier function and returns a Map where the keys are the result of applying the classifier and the values are lists of items that match that key.

A screenshot of a computer program

Description automatically generated

 List.stream().sorted() returns a **new stream** of sorted elements and does **not modify the original list**. You must collect the sorted results back into a new list using collect(Collectors.toList()).

 Collections.sort(List) **modifies the original list** in-place, sorting it directly.

If a banking application is upgraded from an older Java version (say Java 6 or 7) to **Java 8**, this brings improvements in performance, security, and access to modern language features introduced in Java 8. Java 8 is particularly significant because it introduced major changes, including **lambdas, the Stream API, and Optional**. Here’s what this type of upgrade would involve and some examples of how code might be modernized to leverage Java 8 features.

**Why Upgrade a Banking Application to Java 8?**

Upgrading to Java 8 provides the following benefits:

1. **Improved Performance and Security**: Java 8 includes performance enhancements in the JVM and improved security features, which are critical for a banking application.
2. **New Language Features**: Java 8 introduced lambdas, the Stream API, default methods in interfaces, Optional, and other features that make code more readable and concise.
3. **Better Memory Management**: Java 8's garbage collectors are more efficient, and the addition of Metaspace (replacing PermGen) helps with memory management, which is beneficial for high-throughput applications.

**Steps to Upgrade to Java 8 in Production**

1. **Code Refactoring**: Existing code can be refactored to take advantage of Java 8’s features, making the application codebase cleaner and more maintainable.
2. **Library Compatibility Check**: Ensure that all external libraries and frameworks used in the banking application are compatible with Java 8.
3. **Testing**: Comprehensive testing (unit, integration, and performance testing) should be conducted in a staging environment to verify compatibility and catch any regressions before moving to production.
4. **Monitoring and Validation**: In production, monitoring tools can be used to validate that the upgrade doesn't impact performance or stability.

**Example Code Changes After Upgrading to Java 8**

After upgrading to Java 8, you could refactor parts of the code to use new language features. Here are some examples:

**1. Using Lambda Expressions**

Suppose you have a method in Java 7 that iterates over a list of transactions and filters them by type:

**Java 7 Code:**

java

Copy code

List<Transaction> filterTransactionsByType(List<Transaction> transactions, String type) {

List<Transaction> result = new ArrayList<>();

for (Transaction t : transactions) {

if (t.getType().equals(type)) {

result.add(t);

}

}

return result;

}

**Java 8 Code with Lambda and Stream API:** With Java 8, you can simplify this code using lambdas and the Stream API, making it more concise and readable:

java

Copy code

List<Transaction> filterTransactionsByType(List<Transaction> transactions, String type) {

return transactions.stream()

.filter(t -> t.getType().equals(type))

.collect(Collectors.toList());

}

**2. Using Optional to Avoid Null Checks**

In Java 7, you might handle a null account holder check like this:

**Java 7 Code:**

java

Copy code

String getAccountHolderName(Account account) {

if (account != null && account.getHolder() != null) {

return account.getHolder().getName();

} else {

return "Unknown";

}

}

**Java 8 Code with Optional:** Using Optional, you can make this code cleaner and reduce the need for explicit null checks.

java

Copy code

String getAccountHolderName(Account account) {

return Optional.ofNullable(account)

.map(Account::getHolder)

.map(Holder::getName)

.orElse("Unknown");

}

**3. Default Methods in Interfaces**

In Java 7, adding new methods to an interface required implementing it in all classes, which could be challenging in large codebases.

**Java 8 Solution:** Java 8 allows adding **default methods** to interfaces, which helps avoid breaking existing implementations when adding new methods.

java

Copy code

public interface AccountService {

void deposit(double amount);

void withdraw(double amount);

// New default method in interface

default double calculateInterest(double balance) {

return balance \* 0.02;

}

}

**4. Improved Date and Time API**

Java 8 introduced the java.time package, which offers a better, more flexible, and thread-safe date and time API compared to java.util.Date and java.util.Calendar.

**Java 7 Code:**

java

Copy code

SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

Date today = new Date();

System.out.println("Today: " + dateFormat.format(today));

**Java 8 Code:** Using LocalDate and DateTimeFormatter, Java 8 makes date manipulation more straightforward and thread-safe.

java

Copy code

LocalDate today = LocalDate.now();

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

System.out.println("Today: " + today.format(formatter));

**Summary**

Upgrading to Java 8 for a banking application can enhance performance, security, and code readability by introducing new language features. The examples above demonstrate how common operations can be refactored to leverage lambda expressions, Optional, the Stream API, and the new date and time classes in Java 8. These changes lead to a more maintainable and efficient codebase, which is crucial in high-stakes applications like banking.

i dont want to allow the person objects with same name irrespective of other variables then i used to write object.equals(naa

o.name) till this i understand but why should i generate the hashcode in hash method for name only. its just a code right? then why cant we generate hashcode like hash(name,city,age)???

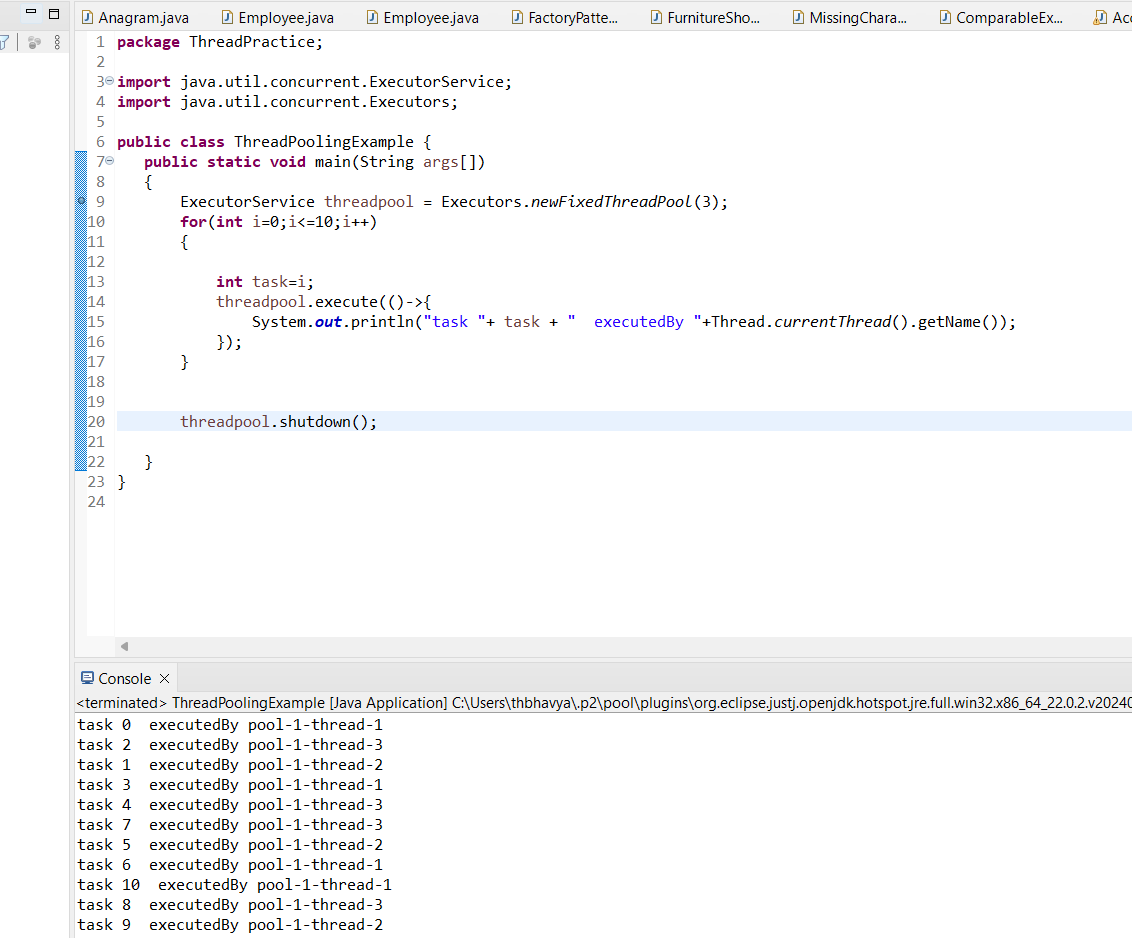
hashCode method determines the "bucket" in a hash-based data structure, while the equals method determines if two objects in the same bucket are indeed equal. If the two methods aren't consistent, it can lead to incorrect behavior in these data structures.

For example:

* If you override equals to compare only the name field, but hashCode is based on name, city, and age, then:
  + Two Person objects with the same name but different city and age would have different hash codes.
  + This violates the contract that **equal objects must have the same hash code**.

Threadpooling vs without threadpooling

Using threadpooling



Without threadpooling

A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated

Here in threadpooling only 3 threads created and completed 10 tasks but in multithreading without threadpooling executed 10 tasks by creating 10 threads.

**Volatile :**

**public class BankAccount {**

**// Shared account balance, marked as volatile**

**private volatile double balance = 0.0;**

**// Deposit method**

**public void deposit(double amount) {**

**balance += amount; // Add amount to the balance**

**System.out.println(Thread.currentThread().getName() + " deposited " + amount + ". New balance: " + balance);**

**}**

**// Withdraw method**

**public void withdraw(double amount) {**

**if (balance >= amount) {**

**balance -= amount; // Subtract amount from the balance**

**System.out.println(Thread.currentThread().getName() + " withdrew " + amount + ". New balance: " + balance);**

**} else {**

**System.out.println(Thread.currentThread().getName() + " attempted to withdraw " + amount + ", but insufficient balance.");**

**}**

**}**

**// Getter method for balance**

**public double getBalance() {**

**return balance;**

**}**

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

**BankAccount account = new BankAccount();**

**// Create threads that will simulate deposits and withdrawals**

**Thread thread1 = new Thread(() -> {**

**account.deposit(1000); // Thread 1 deposits 1000**

**account.withdraw(500); // Thread 1 withdraws 500**

**}, "Thread 1");**

**Thread thread2 = new Thread(() -> {**

**account.deposit(1500); // Thread 2 deposits 1500**

**account.withdraw(200); // Thread 2 withdraws 200**

**}, "Thread 2");**

**// Start the threads**

**thread1.start();**

**thread2.start();**

**// Wait for both threads to finish**

**thread1.join();**

**thread2.join();**

**// Print final balance**

**System.out.println("Final balance: " + account.getBalance());**

**}**

**}**