1. What is the difference between Comparable and Comparator?

- **Comparable**: An interface used to define the natural ordering of objects. The class itself implements this interface and provides the compareTo() method. It allows only a single ordering for a class.
- **Comparator**: An interface used to define custom orderings. It is implemented by a separate class and provides the compare() method. Multiple different orderings can be created for a class using different comparator implementations.

---

2. What is the difference between List and Set?

- **List**: A collection that allows duplicates and maintains the insertion order. It is index-based and elements can be accessed via their index.
- **Set**: A collection that doesn't allow duplicates and doesn't maintain any specific order (although specific implementations like LinkedHashSet do). It is used when uniqueness is a priority.

---

3. Can we use a custom class as a key in the HashMap and what is necessary for that?
Yes, a custom class can be used as a key in a HashMap. The class must override both the hashCode() and equals() methods to ensure the correct working of the hash-based data structure. Without these overrides, it may lead to incorrect behavior, such as failing to retrieve the correct value.

---

4. What is the purpose of the equals() method?
The equals() method is used to compare two objects for equality. It checks whether two objects are logically "equal", rather than checking for reference equality (==). The method should be overridden in a custom class to define meaningful equality based on object properties.

---

5. What will happen if we make an ArrayList collection as final in Java?
If an ArrayList is declared final, you cannot reassign the reference to a new list, but you can still modify the list itself by adding, removing, or updating elements.

---

6. What is the difference between throw and throws?

- **throw**: Used to explicitly throw an exception in code.
- **throws**: Declares the exceptions that a method might throw. It is part of the method signature, indicating that the method could potentially throw certain checked exceptions.

---

7. What will happen in a method with return type Integer, with try-catch-finally blocks raising exceptions and having different return statements?

In this case, the finally block will execute after the try and catch blocks, and if the finally block contains a return statement, it will override any return value from the try or catch blocks. So, the method will return 3 from the finally block, regardless of what happens in try or catch.

---

8. Difference between Runnable and Callable interfaces?

Runnable is the core interface provided for representing multithreaded tasks, and Java 1.5 provided *Callable* as an improved version of *Runnable*.

Runnable interface	Callable interface
It is a part of ***java.lang*** package since Java 1.0	It is a part of the ***java.util.concurrent*** package since Java 1.5.
It cannot return the result of computation.	It can return the result of the parallel processing of a task.
It cannot throw a checked Exception.	It can throw a checked Exception.
In a runnable interface, one needs to override the run() method in Java.	In order to use Callable, you need to override the call()

---

9. What is ExecutorService?

ExecutorService is a framework in Java for managing a pool of threads. It provides methods to submit tasks for execution, control thread lifecycle, and manage asynchronous task execution. It allows for more flexible thread management compared to manually creating and managing threads.

---

10. How do you test the API performance?

API performance can be tested using tools like JMeter, Postman, or Gatling. You can measure metrics such as response time, throughput, latency, and resource usage (CPU, memory) under various loads and stress conditions.

---

11. What is a Future object?
A Future represents the result of an asynchronous computation. It allows you to retrieve the result or check the completion status of the task at a later point. It provides methods like get(), isDone(), and cancel().

---

12. How does HashMap work internally?
Internally, a HashMap uses an array of buckets to store key-value pairs. When a key-value pair is added, the key's hashCode is computed, and the corresponding bucket is identified using the hashCode. If a collision occurs (i.e., two keys have the same bucket), the HashMap resolves this by chaining (using linked lists) or using binary trees (since Java 8) if the chain length exceeds a threshold.

---

13. How does Spring ensure atomicity in applications?
Spring ensures atomicity in applications through transaction management. Using the @Transactional annotation, Spring manages transactions automatically, ensuring that a series of operations either completes entirely or rolls back in case of failure.

---

14. If an application is failing due to a memory issue, how do you check for the issue in Java?
You can use tools like VisualVM, JConsole, or heap dumps to analyze memory usage. Checking for memory leaks, inefficient data structures, or over-retained objects by analyzing garbage collection logs or profiling memory consumption helps in identifying the issue.

---

15. Why is Stream lazy in nature?
Streams in Java are lazy because they don't process data immediately. Intermediate operations like filter(), map() only set up the pipeline, and the actual processing happens when a terminal operation like forEach(), collect() is called. This allows for performance optimizations, such as short-circuiting and minimizing computations.

---

16. What are the different intermediate and terminal methods for the Stream API?

- **Intermediate methods**: filter(), map(), sorted(), limit(), skip(), etc.
- **Terminal methods**: forEach(), collect(), reduce(), count(), findAny(), findFirst(), etc.

---

17. What are the design patterns used in Spring?
Some design patterns used in Spring include:

- **Singleton Pattern**: For creating a single instance of a bean.

- **Factory Pattern**: The BeanFactory and ApplicationContext are examples.
- **Proxy Pattern**: Used in Spring AOP.
- **Template Method Pattern**: Used in JdbcTemplate, RestTemplate.

---

18. How many design patterns do you know in Java?
Common design patterns in Java include:

- **Singleton**
- **Factory**
- **Abstract Factory**
- **Builder**
- **Prototype**
- **Adapter**
- **Decorator**
- **Observer**
- **Strategy**
- **Command**
- **Template Method**

---

19. What is the difference between @Component and @Service annotation? Can we delete @Component in place of @Service?

- **@Component**: A generic stereotype annotation for Spring-managed components.
- **@Service**: A specialization of @Component that indicates that the class performs some business logic.
- Yes, you can delete @Component in favor of @Service because @Service is a more specific version of @Component.

---

20. What is cyclic dependency in Spring?
A cyclic dependency occurs when two or more beans depend on each other, leading to a circular reference that Spring cannot resolve during bean creation. This can lead to errors during application startup unless proxy-based dependency injection (like @Lazy) is used.

---

21. Why do we need to implement the Cloneable interface when the Object class has a clone() method?

The Object class has a clone() method, but if a class does not implement the Cloneable interface, calling clone() will throw a CloneNotSupportedException. The Cloneable interface is a marker interface that signals to the runtime that cloning is allowed for that class. Without it, the JVM assumes that the class should not be cloned, and it enforces this by throwing an exception.

---

22. Can I use the clone() method without overriding it?

Yes, you can use the clone() method without overriding it as long as your class implements the Cloneable interface. The default implementation in Object will perform a shallow copy. However, if your class has fields that refer to other objects, this shallow copy will only duplicate the references, not the actual objects. For a deep copy, you would need to override clone() and manually clone the referenced objects.

---

23. **Which Other Modifiers Are Available in Java and What Is Their Purpose?**

There are five other modifiers available in Java:

- static
- final
- abstract
- synchronized
- volatile

Static: Static fields or methods are class members, whereas non-static ones are object members. Class members don't need an instance to be invoked. They are called with the class name instead of the object reference name**.**

Final: we have the *final* keyword. We can use it with fields, methods, and classes. When final is used on a field, it means that the field reference cannot be changed. So it can't be reassigned to another object. When final is applied to a class or a method, it assures us that that class or method cannot be extended or overridden.

Abstract: When classes are abstract, they can't be instantiated. Instead, they are meant to be subclassed. When methods are *abstract*, they are left without implementation and can be overridden in subclasses.

Synchronized: We can use it with the instance as well as with static methods and code blocks. When we use this keyword, we make Java use a monitor lock to provide synchronization on a given code fragment.

Volatile: We can only use it together with instance and static fields. It declares that the field value must be read from and written to the main memory – bypassing the CPU cache.

---

24. Static Class: A static class is a class that cannot be instantiated, or have a variable created using the new operator. Static classes can only contain static members, such as static methods and constructors. *The top level class cannot be static in java, to create a static class we must create a nested class and then make it static.*

---

25. Methods of the Object class:

clone() – returns a copy of this object

equals() – returns true when this object is equal to the object passed as a parameter

finalize() – the garbage collector calls this method while it's cleaning the memory

getClass() – returns the runtime class of this object

hashCode() – returns a hash code of this object.

notify() – sends a notification to a single thread waiting for the object's monitor

notifyAll() – sends a notification to all threads waiting for the object's monitor

toString() – returns a string representation of this object

wait() – there are three overloaded versions of this method. It forces the current thread to wait the specified amount of time until another thread calls notify() or notifyAll() on this object.