CORE JAVA

1) What is Java?

Java is a high-level, object-oriented, robust, secure programming language, platform-independent, high-performance, Multithreaded, and portable programming language. It was developed by James Gosling and his team at Sun Microsystems in June 1991. It can also be known as the platform, as it provides its own JRE and API. Java is designed to be versatile, secure, and portable, allowing developers to write code that can run on various platforms without modification.

2) List the features of Java Programming language.

https://images.tpointtech.com/images/core/java-features.png

3) What do you understand by Java Virtual Machine (JVM)?

Java Virtual Machine is a virtual machine that enables a computer to run Java programs. It is a virtualized execution environment that allows Java programs to run on any device with a compatible JVM. JVM is the specification that must be implemented in the computer system. It interprets Java bytecode and translates it into machine code for the underlying hardware.

4) What is a ClassLoader?

Classloader is a subsystem of the JVM that is used to load class files. It loads Java classes into memory during runtime. Responsible for finding and loading class files needed by a Java program.

There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader:**This is the first classloader, which is the superclass of Extension classloader. It loads the *jar* file, which contains all class files of Java Standard Edition, like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes, etc.
2. **Extension ClassLoader:**This is the child classloader of Bootstrap and the parent classloader of the System classloader. It loads the jar files located inside *$JAVA_HOME/jre/lib/ext*

3. **System/Application ClassLoader:**This is the child classloader of the Extension classloader. It loads the class files from the classpath. By default, the classpath is set to the current directory. You can change the classpath using the "-cp" or "-classpath" switch. It is also known as the Application classloader.

5) What if we write static public void instead of public static void?

The program compiles and runs correctly because the order of specifiers does not matter in Java.

In Java, the placement of access modifiers (public, private, protected) and other specifiers (static, final, etc.) before the return type (void in this case) is flexible. Both public static void and static public void are accepted and considered correct syntax.

6. What are the various access specifiers in Java?

In Java, access specifiers are the keywords that are used to define the access scope of a method, class, or variable. There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

Access Specifiers

public: public classes, methods, or variables that are defined as public can be accessed by any class or method.

protected: The protected classes, methods, or variables can be accessed by the class of the same package, or by a subclass of this class, or within the same class.

private: The private class, methods, or variables defined as private can be accessed within the class only.

default (package-private): If no access specifier is used, it is considered package-private. Members with default access are accessible only within the same package.

7. What are the various access specifiers in Java?

In Java, access specifiers are the keywords that are used to define the access scope of a method, class, or variable. There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

Access Specifiers

public: public classes, methods, or variables that are defined as public can be accessed by any class or method.

protected: The protected classes, methods, or variables can be accessed by the class of the same package, or by a subclass of this class, or within the same class.

private: The private class, methods, or variables defined as private can be accessed within the class only.

default (package-private): If no access specifier is used, it is considered package-private. Members with default access are accessible only within the same package.

8. What is an object?

The Object is a real-time entity having some state and behavior. In Java, an Object is an instance of a class that has instance variables as the state of the object and methods as the behavior of the object. The object of a class can be created by using the new keyword.

9. What is the difference between an object-oriented programming language and an object-based programming language?

There are the following basic differences between object-oriented languages and object-based languages.

Object-oriented languages follow all the concepts of OOP, whereas object-based languages do not follow all the concepts of OOP, like inheritance and polymorphism. It allows the creation of classes and objects, and it supports the defining and implementation of methods within those classes.

Object-oriented languages do not have the inbuilt objects, whereas Object-based languages have the inbuilt objects; for example, JavaScript has the window object. It allows the creation of objects and encapsulation of data, but it might not provide the complete set of features associated with full OOP.

10. What is the constructor?

The constructor can be defined as a special type of method that is used to initialize the state of an object. It is invoked when the class is instantiated, and the memory is allocated for the object. Every time an object is created using the new keyword, the default

constructor of the class is called. The name of the constructor must be similar to the class name. The constructor must not have an explicit return type.

11. Is the constructor inherited?

No, Constructors are not inherited. The superclass constructor can be called from the first line of a subclass constructor by using the keyword super and passing appropriate parameters to set the private instance variables of the superclass.

12. Can you make a constructor final?

No, in Java, constructors cannot be declared as final. The final keyword is used to indicate that a class, method, or variable cannot be further subclassed, overridden, or modified. However, it does not apply to constructors.

13. Can we overload the constructors?

Yes, the constructors can be overloaded by changing the number of arguments accepted by the constructor or by changing the data type of the parameters.

Each constructor provides a different way to initialize an object of the class. The choice of which constructor to invoke is based on the arguments passed during object creation.

14. What is the static variable?

The static variable is used to refer to the common property of all objects (that is not unique for each object), for example, the company name of employees, the college name of students, etc. Static variable gets memory only once in the class area at the time of class loading. Using a static variable makes your program more memory efficient (it saves memory). A static variable belongs to the class rather than the object. The following Java program depicts the use of a static variable.

15. What is the static method?

In Java, a static variable is a class-level variable that belongs to the class rather than to instances of the class. It is shared among all instances of the class and is initialized only once when the class is loaded into memory.

16. What are the restrictions that are applied to the Java static methods?

Java static methods have certain restrictions and characteristics that differentiate them from instance methods. Here are the key restrictions and considerations for Java static methods:

The two main restrictions applied to the static methods are as follows:

- The static method cannot use a non-static data member or call a non-static method directly.
- This and super keyword cannot be used in a static context as they are non-static.
- Static methods in Java cannot be overridden. Even if a subclass declares a static method with the same signature as a static method in its superclass, it is considered method hiding, not overriding.

17. Why the main() method is static?

Because the object is not required to call the static method, if we make the main() method non-static, the JVM will have to create its object first and then call the main() method, which will lead to extra memory allocation.

18. Can we override the static methods?

No, static methods cannot be overridden in Java. While a subclass can declare a static method with the same signature as a static method in its superclass, it is considered method hiding, not method overriding.

19. Can we make constructors static?

No, constructors cannot be declared as static in Java, as we know that the static context (method, block, or variable) belongs to the class, not the object. Since Constructors are invoked only when the object is created, there is no sense in making the constructors

static. Making a constructor static would imply that it belongs to the class and not to instances, which contradicts the fundamental purpose of a constructor. However, if we try to do so, the compiler will show the compiler error..

20. Can we make the abstract methods static in Java?

No, abstract methods cannot be declared as static in Java. In Java, if we make the abstract methods static, they will become part of the class, and we can directly call them, which is unnecessary. Since abstract methods are expected to be overridden by subclasses, making them static would be contradictory to their purpose. Calling an undefined method is completely useless; therefore, it is not allowed.

21. Can we declare the static variables and methods in an abstract class?

Yes, it is possible to declare static variables and methods in an abstract class in Java. As we know that there is no requirement to make the object access the static context; therefore, we can access the static context declared inside the abstract class by using the name of the abstract class.

22. What is this keyword in Java?

In Java, this keyword is a reference variable that refers to the current object. There are various uses of this keyword in Java. It is primarily used within the instance methods of a class to refer to the object on which the method is invoked. It can be used to refer to current class properties such as instance methods, variables, constructors, etc. It can also be passed as an argument to the methods or constructors. It can also be returned from the method as the current class instance.

23. Can we assign the reference to this variable?

No, it is not possible to assign a new value to this variable in Java. this keyword is an implicit reference to the current instance of the class and is automatically set by the Java runtime system when a method is invoked. However, if we try to do so, the compiler error will be shown. Consider the following example.

24. Can this keyword be used to refer to static members?

Yes, it is possible to use this keyword to refer to static members because it is just a reference variable that refers to the current class object. However, as we know, it is unnecessary to access static variables through objects. Therefore, it is not the best practice to use this to refer to static members. Consider the following example.

25. What is the Inheritance?

Inheritance is a mechanism by which one object acquires all the properties and behavior of another object of another class. It is used for Code Reusability and Method Overriding. Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (subclass or derived class) to inherit attributes and behaviors from another class (superclass or base class). Moreover, we can add new methods and fields in your current class also. Inheritance represents the IS-A relationship that is also known as a parent-child relationship.

here are five types of inheritance in Java:

- **Single-level inheritance**
- **Multi-level inheritance**
- **Multiple Inheritance**
- **Hierarchical Inheritance**
- **Hybrid Inheritance**

26. Which class is the superclass for all the classes?

In Java, the Object class is the superclass of all other classes. Every class in Java directly or indirectly extends the Object class. The Object class provides fundamental methods that are inherited by all classes, such as toString(), equals().

27. Why is multiple inheritance not supported in Java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in Java. Consider a scenario where A, B, and C are three classes. The C class inherits A and

B classes. If A and B classes have the same method and we call it from a child class object, there will be ambiguity in calling the method of A or B class.

28. What is aggregation?

Aggregation can be defined as the relationship between two classes where the aggregate class contains a reference to the class it owns.

Aggregation is a type of association in object-oriented programming where one class contains an object of another class, forming a relationship between them. It represents a "has-a" relationship, indicating that a class has an entity as a part of its structure.

For example, the aggregate class Employee, having various fields such as age, name, and salary, also contains an object of the Address class, having various fields such as Address-Line 1, City, State, and pin-code.

29. What is composition?

Composition is a stronger form of aggregation in object-oriented programming. Holding the reference of a class within another class is known as composition. When an object contains another object, if the contained object cannot exist without the existence of the container object, then it is called composition. In other words, we can say that composition is a particular case of aggregation, which represents a stronger relationship between two objects. Example: A class contains students. A student cannot exist without a class. There exists a composition between the class and the students.

30. What is the difference between aggregation and composition?

Aggregation represents a weak relationship, whereas composition represents a strong relationship. Aggregation represents a "has-a" relationship, where one class has a reference to another class, but the referenced class can exist independently. Composition also represents a "has-a" relationship, but it is a stronger form of association. For example, the bike has an indicator (aggregation), but the bike has an engine (composition).

31. What is super in Java?

In Java, super is a keyword that is used to refer to the immediate parent class object. Whenever you create an instance of the subclass, an instance of the parent class is created implicitly, which is referred to by the super reference variable. The super() is called in the class constructor implicitly by the compiler if there is no super or this.

32. What is object cloning?

Object cloning in Java refers to the process of creating an exact copy of an object. The clone() method of the Object class is used to clone an object. The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we do not implement Cloneable interface, clone() method generates CloneNotSupportedException. The clone() method creates a new object with the same state as the original object.

33. What is method overloading?

Method overloading is the polymorphism technique that allows us to create multiple methods with the same name but different signatures. Method overloading in Java occurs when a class has multiple methods with the same name but different parameter lists (number, type, or order of parameters). We can achieve method overloading in the following two ways:

34. Can we overload the main() method?

Yes, we can have any number of main() methods in a Java program by using method overloading. However, the JVM looks for the standard public static void main(String[] args) signature when starting a Java program. Overloaded main() methods can be called from within the standard main() method, but the program execution begins from the standard main() method.

35. What is method overriding?

If a subclass provides a specific implementation of a method that is already provided by its parent class, it is known as Method Overriding. Method overriding in Java occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. It is used for runtime polymorphism and to implement the interface methods.

36. Can we override the static method?

No, we cannot override the static method because they are a part of the class, not the object.

37. Why can we not override a static method?

Static methods cannot be overridden in the traditional sense of dynamic method dispatch that is associated with polymorphism. It is because the static method is a part of the class, and it is bound with the class, whereas the instance method is bound with the object, and static gets memory in the class area, and instance gets memory in the heap.

38. Can we override the private methods?

No, we cannot override the private methods in Java because the scope of private methods is limited to the class, and we cannot access them outside of the class. But private methods are not accessible outside their class. Therefore, private methods cannot be overridden because subclasses do not inherit them.

39. What is the final variable?

In Java, the final variable is used to restrict the user from updating it. If we initialize the final variable, we cannot change its value. The final keyword is used to declare a final variable. In other words, we can say that the final variable, once assigned to a value, can never be changed after that. The final variable, which is not assigned to any value, can only be assigned through the class constructor.

40. What is the final method?

If we change any method to a final method, we cannot override it. In Java, a final method is a method that subclasses cannot override. When a method is declared as final in a superclass, it means that no subclass can provide a different implementation for that method.

41. What is the final class?

If we make a class final, we cannot inherit it into any of the subclasses.

In Java, a final class is a class that cannot be subclassed. When a class is declared as final, it means that no other class can extend (inherit from) it.

42. Can we initialize the final blank variable?

Yes, if it is not static, we can initialize it in the constructor. If it is a static, blank final variable, it can be initialized only in the static block.

43. Can we declare the main() method as final?

Yes, we can declare the main() method as public static final void main(String[] args){}. it is technically allowed to declare the main method as final, but it would not have any impact or special significance. The final keyword, when applied to a method, indicates that subclasses cannot override the method.

44. Can we declare a constructor as final?

No, constructors cannot be declared as final in Java because it is never inherited. Constructors are not ordinary methods; therefore, there is no sense in declaring constructors as final. However, if we try to do so, the compiler will throw an error.

45. Can we declare an interface as final?

No, we cannot declare an interface as final because some class must implement the interface to provide its definition. The whole idea of an interface is to provide a contract that classes can implement. Making an interface final would defeat the purpose of allowing classes to implement it. However, if we try to do so, the compiler will show an error.

46. What are the differences between the final method and the abstract method?

Final method: A final method in a class cannot be overridden by subclasses. Once a method is declared final, it cannot be modified or overridden by any subclass.

Abstract method: An abstract method is a method declared in an abstract class or interface that has no implementation in the class/interface where it is declared. Subclasses or implementing classes must provide a concrete implementation for abstract methods.

47. What is Runtime Polymorphism?

Runtime polymorphism also known as dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method is called through the reference variable of a superclass. It is

achieved through method overriding in object-oriented programming, where a subclass provides a specific implementation for a method that is already defined in its superclass.

48. Can we achieve runtime polymorphism by data members?

No, runtime polymorphism is primarily related to methods/functions, not data members because method overriding is used to achieve runtime polymorphism and data members cannot be overridden. We can override the member functions but not the data members. Data members (fields or properties) do not exhibit polymorphic behavior in the same way. Consider the example given below.

49. What is the abstraction?

Abstraction is a process of hiding the implementation details and showing only functionality to the user. It displays just the essential things to the user and hides the internal information, it involves simplifying the system by modelling classes based on real-world entities and focusing on relevant attributes and behaviors. For example, sending SMS where you type the text and send the message. We do not know the internal processing about the message delivery. Abstraction enables us to focus on what the object does instead of how it does it. Abstraction lets you focus on what the object does instead of how it does it.

50. What is the difference between abstraction and encapsulation?

Abstraction and encapsulation are related concepts but have distinct meanings. Abstraction is about showing only the essential features of an object while hiding the unnecessary details. Encapsulation, on the other hand, is the bundling of data and methods that operate on that data into a single unit, often a class. Abstraction is more about design and modelling, while encapsulation is about implementation and data hiding.

51. What is the abstract class?

A class that is declared as abstract is known as an abstract class. It needs to be extended and its method implemented. It cannot be instantiated. Abstract classes can have both

abstract and concrete methods and can be extended by other classes. It can also have the final methods which will force the subclass not to change the body of the method.

52. Can there be an abstract method without an abstract class?

No, an abstract method must belong to an abstract class. Abstract methods are defined in abstract classes and must be implemented by the subclasses that extend the abstract class.

53 Can we use abstract and final both with a method?

No, we cannot use both abstract and final together for a method. An abstract method is meant to be overridden by subclasses, while a final method cannot be overridden.

54. What is the interface?

The interface is a blueprint for a class that has static constants and abstract methods. It can be used to achieve full abstraction and multiple inheritance. An interface in Java is a collection of abstract methods. It is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. Java Interface also represents the IS-A relationship. It cannot be instantiated just like the abstract class. However, we need to implement it to define its methods. Since Java 8, we can have the default, static, and private methods in an interface.

55. Can we declare an interface method static?

No, interface methods cannot be declared as static prior to Java 8. However, starting from Java 8, we can declare static methods in interfaces.

56. Can the Interface be final?

No, because an interface needs to be implemented by the other class and if it is final, it cannot be implemented by any class.

57. What is the package?

A package is a group of similar type of classes, interfaces, and sub-packages. It provides access protection and removes naming collision. The packages in Java can be categorized into two forms, inbuilt package, and user-defined package. There are many built-in packages such as Java, lang, awt, javax, swing, net, io, util, sql, etc. Consider the following example to create a package in Java.

58. What are the advantages of defining packages in Java?

By defining packages, we can avoid the name conflicts between the same class names defined in different packages. Packages also enable the developer to organize the similar classes more effectively. For example, one can clearly understand that the classes present in java.io package are used to perform io related operations.

59. What are the advantages of defining packages in Java?

By defining packages, we can avoid the name conflicts between the same class names defined in different packages. Packages also enable the developer to organize the similar classes more effectively. For example, one can clearly understand that the classes present in java.io package are used to perform io related operations.

60. What is Exception Handling?

Exception Handling is a mechanism that is used to handle runtime errors (exceptions) to prevent the abnormal termination of a program. It is used primarily to handle checked exceptions. Exception handling maintains the normal flow of the program. There are mainly two types of exceptions: checked and unchecked. Here, the error is considered as the unchecked exception.

61. Explain the hierarchy of Java Exception classes?

The hierarchy of Java Exception classes includes the top-level class java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:

https://d2jdgazzki9vjm.cloudfront.net/images/throwable.png

62. What is the difference between Checked Exception and Unchecked Exception?

1) Checked Exception

Checked exceptions are checked at compile-time, and the programmer must handle them using try-catch or declare them in the method's throws clause. The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions, for example, IOException, SQLException, etc.

2) Unchecked Exception

Checked exceptions are not checked at compile-time and include runtime exceptions and errors. The classes that extend RuntimeException are known as unchecked exceptions, for example, ArithmeticException, NullPointerException, etc.

63. What is the base class for Error and Exception?

The Throwable class is the base class for Error and Exception.

64. What is finally block?

In programming languages like Java, the finally block is a part of exception handling. It contains code that will be executed regardless of whether an exception is thrown or not. The finally block is often used to ensure that certain clean-up or resource release operations take place, whether an exception occurs or not.

Finally block follows try or catch block. If we do not handle the exception, before terminating the program, JVM runs finally block, (if any). The finally block is mainly used to place the clean-up code such as closing a file or closing a connection. Here, we must know that for each try block there can be zero or more catch blocks, but only one finally block.

The finally block will not be executed if program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

65. Can finally block be used without a catch?

Yes, a finally block can be used without a catch block. In exception handling, we can have a try block followed by either a catch block, a finally block, or both. According to the definition of finally block, it must be followed by a try or catch block, therefore, we can use try block instead of catch. More details.

66. Can an exception be rethrown?

Yes, an exception can be rethrown. In Java, we can catch an exception in one catch block and then throw the same exception (or a different one) again. It can be useful for logging or handling the exception at a higher level in the program.

67. What is String Pool?

The String Pool is a special memory area in Java where string literals are stored. The main advantage of using the String pool is whenever we create a string literal; the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string does not exist in the pool, a new string instance is created and placed in the pool. This mechanism helps in memory optimization by reducing the number of duplicate string objects. Therefore, it saves the memory by avoiding the duplicacy.

68. What is the meaning of immutable regarding String?

The simple meaning of immutable is unmodifiable or unchangeable. In Java, String is immutable, i.e., once string object has been created, its value cannot be changed. Consider the following example for better understanding.

69. What are the differences between String and StringBuffer?

The differences between the String and StringBuffer is given in the table below.

No.	String	StringBuffer
1)	The String class is immutable.	The StringBuffer class is mutable.
2)	The String is slow and consumes more memory when we concat too many strings because every time it creates a new instance.	The StringBuffer is fast and consumes less memory when you concat strings.
3)	The String class overrides the equals() method of Object class. So we can compare the contents of two strings by equals() method.	The StringBuffer class does not override the equals() method of Object class.

70. What are the differences between StringBuffer and StringBuilder?

The differences between the StringBuffer and StringBuilder is given below.

No.	StringBuffer	StringBuilder
1)	StringBuffer is *synchronized*, i.e., thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized*,i.e., not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.

71. How can we create an immutable class in Java?

We can create an immutable class by defining a final class having all of its members as final. Consider the following example.

a. Make the class final, so it cannot be subclassed.

b. Declare all fields private and final.

c. Do not provide setter methods for the fields.

d. Ensure that any mutable objects within the class are defensively copied to prevent external modification.

e. If the class has mutable fields, make sure to return defensive copies in getter methods.

72. What is the purpose of the toString() method in Java?

The toString() method is used to obtain a string representation of an object. It is defined in the Object class and can be overridden by custom classes to provide a meaningful representation of the object's state depending on implementation. By overriding the toString() method of the Object class, we can return the values of the object, so we do not need to write much code. Consider the following example.

73. What is Garbage Collection?

Garbage Collection is the automatic process in Java that identifies and removes unused objects, freeing up memory and preventing memory leaks. In other words, we can say that it is the process of removing unused objects from the memory to free up space and make this space available for Java Virtual Machine. Due to garbage collection Java gives 0 as output to a variable whose value is not set, i.e., the variable has been defined but not initialized. For this purpose, we were using the free() function in the C language and delete() in C++. In Java, it is performed automatically. So, Java provides better memory management.

74. How garbage collection is controlled?

Garbage collection is managed by Java Virtual Machine (JVM). It is performed when there is not enough space in the memory and memory is running low. We can externally call the System.gc() method for the garbage collection. Developers can influence garbage

collection behavior by adjusting parameters such as heap size, garbage collection algorithms, and tuning options. However, it depends upon the JVM whether to perform it or not.

75. What is the purpose of the finalize() method?

The finalize() method is invoked just before the object is garbage collected. It is used to perform clean-up processing. The Garbage collector of JVM collects only those objects that are created by new keyword. However, it is important to note that the finalize() method is rarely used and is considered somewhat unreliable for resource cleanup. So if we have created an object without new, we can use the finalize method to perform clean-up processing (destroying remaining objects). The clean-up processing is the process to free up all the resources, network which was previously used and no longer needed. It is essential to remember that it is not a reserved keyword, finalize method is present in the object class hence it is available in every class as object class is the superclass of every class in Java. Here, we must note that neither finalization nor garbage collection is guaranteed. Consider the following example.

76. What is the use of transient keyword?

In Java, the transient keyword is used to indicate that a field should not be serialized when the object is serialized. By determining transient keyword, the value of variable need not persist when it is restored.

77. What is the difference between Serializable and Externalizable interface?

No.	Serializable	Externalizable
1)	The Serializable interface does not have any method, i.e., it is a marker interface.	The Externalizable interface contains is not a marker interface, It contains two methods, i.e., writeExternal() and readExternal().

2)	It is used to "mark" Java classes so that objects of these classes may get the certain capability.	The Externalizable interface provides control of the serialization logic to the programmer.
3)	It is easy to implement but has the higher performance cost.	It is used to perform the serialization and often result in better performance.

78. What are wrapper classes?

In Java, wrapper classes are the classes that allow primitive types to be accessed as objects. In other words, we can say that wrapper classes are built-in Java classes that allows the conversion of objects to primitives and primitives to objects. The process of converting primitives to objects is called auto-boxing, and the process of converting objects to primitives is called unboxing. There are eight wrapper classes present in java.lang package is given below.

79. What is object cloning?

Object cloning refers to the process of creating an exact copy of an object. The clone() method of the Object class is used to clone an object. The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. Object cloning creates a new object with the same state as the original object, but the two objects are independent of each other. If we do not implement Cloneable interface, the clone() method generates CloneNotSupportedException. The clone() method is defined in the Object class. The syntax of the clone() method is as follows:

80. What is Locale?

A Locale object represents a specific geographical, political, or cultural region. Locale objects are used to specify formats for dates, numbers, currencies, and messages to be displayed to users in their preferred language and format. This object can be used to get the locale-specific information such as country name, language, variant, etc.

81. How will you load a specific locale?

One can load specific locale by using ResourceBundle.getBundle(?) method.

SPRING CORE

1. What is Spring?

It is a lightweight, loosely coupled and integrated framework for developing enterprise applications in java.

2 What are the advantages of spring framework?

1. Predefined Templates
2. Loose Coupling
3. Easy to test
4. Lightweight
5. Fast Development
6. Powerful Abstraction
7. Declarative support

3 What are the modules of spring framework?

1. Test
2. Spring Core Container
3. AOP, Aspects and Instrumentation
4. Data Access/Integration
5. Web

4 What is IOC and DI?

4 IOC (Inversion of Control) and DI (Dependency Injection) is a design pattern to provide loose coupling. It removes the dependency from the program.

5 Let's write a code without following IOC and DI.

5 What is the role of IOC container in spring?

IOC container is responsible to:

- create the instance
- configure the instance, and
- assemble the dependencie

6 What are the types of IOC container in spring?

There are two types of IOC containers in spring framework.

1. BeanFactory
2. ApplicationContext

7 What is the difference between BeanFactory and ApplicationContext?

BeanFactory is the **basic container** whereas ApplicationContext is the **advanced container**. ApplicationContext extends the BeanFactory interface. ApplicationContext provides more facilities than BeanFactory such as integration with spring AOP, message resource handling for i18n etc.

8 What is the difference between constructor injection and setter injection?

No.	Constructor Injection	Setter Injection
1)	No Partial Injection	Partial Injection
2)	Desn't override the setter property	Overrides the constructor property if both are defined.
3)	Creates new instance if any modification occurs	Doesn't create new instance if you change the property value
4)	Better for too many properties	Better for few properties.

9 What is autowiring in spring? What are the autowiring modes?

Autowiring enables the programmer to inject the bean automatically. We don't need to write explicit injection logic.

Let's see the code to inject bean using dependency injection.

10. What are the different bean scopes in spring?

There are 5 bean scopes in spring framework.

No.	Scope	Description
1)	singleton	The bean instance will be only once and same instance will be returned by the IOC container. It is the default scope.
2)	prototype	The bean instance will be created each time when requested.
3)	request	The bean instance will be created per HTTP request.
4)	session	The bean instance will be created per HTTP session.
5)	globalsession	The bean instance will be created per HTTP global session. It can be used in portlet context only.

10. In which scenario, you will use singleton and prototype scope?

Singleton scope should be used with EJB **stateless session bean** and prototype scope with EJB **stateful session bean**.

11. What are the advantages of JdbcTemplate in spring?

Less code: By using the JdbcTemplate class, you don't need to create connection,statement,start transaction,commit transaction and close connection to execute different queries. You can execute the query directly.

12. What are classes for spring JDBC API?

1. JdbcTemplate
2. SimpleJdbcTemplate
3. NamedParameterJdbcTemplate
4. SimpleJdbcInsert
5. SimpleJdbcCall

13.) How can you fetch records by spring JdbcTemplate?

You can fetch records from the database by the **query method of JdbcTemplate**. There are two interfaces to do this:

1. ResultSetExtractor
2. RowMapper

14. What is AOP?

AOP is an acronym for Aspect Oriented Programming. It is a methodology that divides the program logic into pieces or parts or concerns.

It increases the modularity and the key unit is Aspect.

15. What are the advantages of spring AOP?

AOP enables you to dynamically add or remove concern before or after the business logic. It is **pluggable** and **easy to maintain**.

16. What are the AOP terminology?

AOP terminologies or concepts are as follows:

- JoinPoint
- Advice
- Pointcut
- Aspect
- Introduction
- Target Object
- Interceptor
- AOP Proxy
- Weaving

17. What is Advice?

Advice represents action taken by aspect.

18. What is the front controller class of Spring MVC?

The **DispatcherServlet** class works as the front controller in Spring MVC.

19. What does @Controller annotation?

The **@Controller** annotation marks the class as controller class. It is applied on the class.

---

20) What does @RequestMapping annotation?

The **@RequestMapping** annotation maps the request with the method. It is applied on the method.

---

21) What does the ViewResolver class?

The **View Resolver** class resolves the view component to be invoked for the request. It defines prefix and suffix properties to resolve the view component.

---

22) Which ViewResolver class is widely used?

The **org.springframework.web.servlet.view.InternalResourceViewResolver** class is widely used.

---

23) Does spring MVC provide validation support?

Yes.

Spring DATA Jpa

1. What Is Spring Data JPA, and What Are Its Main Features?

Spring Data JPA is a part of the larger Spring Data family which aims to simplify data access within the Java Persistence API (JPA). Its primary goal is to reduce the amount of boilerplate code required to implement data access layers, making developers' lives significantly easier. One of the main features of Spring Data JPA is its repository support, which abstracts CRUD (Create, Read, Update, Delete) operations, thereby eliminating the need for boilerplate code. It also offers query derivation from method names, allowing for the execution of queries simply by defining repository methods. Another noteworthy feature is its support for pagination and sorting, which simplifies the implementation of these common requirements in applications. Its ability to seamlessly integrate with the Spring ecosystem provides transaction management and enhances the overall development experience. By leveraging Spring Data JPA, I can focus more on the business logic rather than the data access setup, significantly increasing productivity and code quality.

2. How Does Spring Data JPA Simplify the Development of Data Access Layers in Java Applications?

In my experience, Spring Data JPA significantly simplifies the development of data access layers in Java applications by automating the boilerplate code required for CRUD operations. This automation is made possible through its repository abstraction layer,

which allows me to focus on the business logic rather than the data access code. Additionally, the feature of query methods generation enables me to create complex queries without writing custom SQL, thus enhancing productivity and ensuring a cleaner, more maintainable codebase.

×

3. What Is The Difference Between Spring Data JPA And Hibernate?

Spring Data JPA acts as a layer on top of JPA providers like Hibernate, offering an additional layer of abstraction. This abstraction is mainly through repositories, making data access more straightforward and reducing boilerplate code significantly. On the other hand, Hibernate is a JPA implementation that directly interacts with the database, providing a more granular level of control over database operations. Using Spring Data JPA, I can leverage the underlying Hibernate features without directly dealing with its complexities, streamlining the development process and focusing on business logic rather than data access code. This combination allows for rapid, efficient development, especially in large-scale applications where data management is a primary concern.

4. What Are the Benefits of Using Spring Data JPA Over JDBC or JPA Alone?

In my experience, Spring Data JPA significantly simplifies the implementation of data access layers compared to JDBC or JPA alone. It abstracts the boilerplate code needed for CRUD operations, which means I can focus more on the business logic rather than the data access code. By using repositories and derived query methods, I can easily perform complex queries without writing any SQL. This approach not only speeds up development but also reduces the chance of errors. Additionally, Spring Data JPA integrates seamlessly with Spring's transaction management, making it easier to manage transactions and ensuring data consistency in my applications.

×

5. What Is A Spring Data JPA Repository, And How Does It Differ From A Regular Spring Repository?

In my experience, a Spring Data JPA repository is a powerful interface in Spring framework designed to simplify the implementation of data access layers. It automatically handles CRUD operations, significantly reducing the need for boilerplate code. This is a stark contrast to regular Spring repositories, where such operations might require manual implementation. One key difference is that Spring Data JPA repositories can derive queries directly from method names, removing the need to explicitly define them. Additionally, it offers more flexibility by allowing the integration of custom repository logic, providing a seamless way to extend beyond standard CRUD operations. This versatility makes Spring Data JPA repositories an essential tool in my development toolkit, enabling me to focus on more complex business logic rather than data access code.

6. How Do You Create A Spring Data JPA Repository For A Given Entity?

In creating a Spring Data JPA repository for an entity, the first step I take is to define an interface for that entity. This interface then extends one of Spring Data's repository interfaces, such as CrudRepository or JpaRepository. These interfaces come with several methods for common operations like save, delete, and find. I choose the interface to extend based on the needs of my application; for example, JpaRepository offers pagination and sorting capabilities in addition to CRUD operations.

To enable Spring Data JPA to implement this repository interface, I also make sure my project is correctly set up with the necessary Spring configuration and entity scanning annotations, such as @EnableJpaRepositories and @EntityScan. By doing this, Spring Data JPA can automatically generate the implementation at runtime, saving me the effort of manually writing the DAO layer code.

7. What Are The Different Types Of Spring Data JPA Repositories, And When Should You Use Each One?

In Spring Data JPA, there are primarily three types of repositories: CrudRepository, JpaRepository, and PagingAndSortingRepository. I choose CrudRepository when I need basic CRUD functionality without pagination or sorting. It's simple and straightforward for basic operations. JpaRepository extends CrudRepository and is usually my go-to for JPA related operations because it provides JPA related methods such as flushing and batch operations. It makes managing entities easier and more efficient, especially when dealing

with large datasets. Lastly, PagingAndSortingRepository is ideal when I'm implementing complex APIs that require pagination and sorting. It abstracts the boilerplate code needed for these operations, allowing me to focus on business logic. Depending on my project's needs, whether it's basic CRUD operations, handling large datasets, or implementing complex queries with pagination and sorting, I select the repository type accordingly.

8. How Do You Implement Custom Repository Methods in Spring Data JPA?

In Spring Data JPA, implementing custom repository methods involves creating an interface for the custom functionality and then integrating it with the main repository interface. First, I define an interface that declares the custom methods I need. For instance, if I need more complex queries that can't be covered by the query derivation mechanism, I create an interface named CustomRepository. Next, I create an implementation class for this interface, say CustomRepositoryImpl, and write the custom method implementations there. It's crucial that the implementation class name follows the naming convention of appending Impl to the custom interface name. Finally, I extend my main repository interface with the custom interface. This way, Spring Data JPA automatically detects the custom implementation and integrates it with the predefined CRUD operations, allowing me to leverage complex queries and operations without straying from the Spring Data ecosystem. This approach ensures that I can extend the functionality of my repositories in a structured and maintainable manner.

9. What Is The Difference Between @Query And Query Annotations In Spring Data JPA?

In Spring Data JPA, the @Query annotation allows me to define custom JPQL or native queries directly on my repository methods. This is extremely useful when the query I need to use is not simple enough to be covered by the naming conventions of Spring Data. On the other hand, there isn't a direct "Query" annotation in Spring Data JPA. I believe the confusion might come with the "Query" objects used in JPA to execute JPQL or native queries programmatically. These are not annotations but rather a way to construct queries in a dynamic and programmatic manner, allowing for more flexibility in setting parameters and executing the query. The @Query annotation gives me the advantage of defining complex queries right within my repository interface, keeping my code clean and maintainable.

In my experience, implementing pagination in Spring Data JPA is essential for handling large datasets efficiently. By doing so, it significantly enhances the application's performance and user experience. Spring Data JPA makes this process straightforward through the Pageable interface and the Page class. To implement pagination, I usually extend my repository interface from PagingAndSortingRepository or use Pageable as a parameter in my repository methods. This allows me to specify the page number, size, and sorting criteria dynamically. When a method is invoked, Spring Data JPA handles the pagination logic automatically, returning a Page object that contains the requested data along with useful metadata, such as total elements and pages. This approach has consistently proven to be effective and efficient in my projects.

11. How Do You Handle Transactions in Spring Data JPA?

In Spring Data JPA, managing transactions is crucial for maintaining the consistency and integrity of data. I handle transactions using the @Transactional annotation. This approach allows me to define the boundaries of a transaction at the method level in my repository or service layer. It simplifies transaction management by automatically handling the opening, committing, or rolling back of transactions based on the execution outcome. When implementing transactional operations, I ensure to specify the transactional context properly, like setting the readOnly flag to true for read-only operations to optimize performance. Additionally, I carefully consider the propagation behavior to manage how transactions relate to each other, especially in complex scenarios involving multiple transactional methods.

12. What Is The Difference Between @Transactional And @Modifying Annotations In Spring Data JPA?

In Spring Data JPA, @Transactional and @Modifying annotations serve different purposes. @Transactional indicates that a method or an entire class should be executed within a transactional context, ensuring consistency and rollback capabilities in case of errors. It's crucial for maintaining data integrity during read and write operations. On the other hand, @Modifying is used specifically with @Query annotations to indicate that a query method should execute a modifying query such as INSERT, UPDATE, or DELETE. This annotation makes it clear that the operation changes the state of the database and requires a transaction, but it doesn't start one by itself. It's vital to use @Modifying for custom DML operations to ensure they are executed within the transactional context provided by @Transactional.

13. How Do You Implement Caching in Spring Data JPA?

In Spring Data JPA, implementing caching starts with enabling the cache configuration in the application. I do this by adding @EnableCaching annotation to my configuration class. This step is crucial as it tells Spring to start looking for caching opportunities within the application. Next, I focus on the areas where caching can be most beneficial, such as frequently read but rarely updated data.

For entity-level caching, I use the @Cacheable annotation on my entity classes. This way, I instruct Spring to store the entity in the cache after the first database access, reducing the load on the database for subsequent reads. Additionally, I configure the cache properties in my application.properties file, specifying the cache manager and the cache names, which allows me finer control over the cache behavior, like eviction policies and expiration times.

Managing cache effectively means understanding when to invalidate stale data. For this, I use the @CacheEvict annotation on methods that update or delete data, ensuring the cache remains fresh and consistent with the database state.

14. What Are The Different Types Of Caching Strategies In Spring Data JPA?

In my experience, caching is pivotal in enhancing the performance of applications by reducing the load on the database. In Spring Data JPA, there are primarily two types of caching strategies I've worked with: first-level and second-level caching. First-level caching is associated with the persistence context and is enabled by default. It ensures that within a single session, data is retrieved from the cache after the first database hit, which significantly reduces the number of database queries.

To further optimize performance, I've implemented second-level caching in several projects. This is shared across sessions and can drastically reduce database calls for frequently accessed data. Utilizing specific configurations and cache providers, like

EhCache or Hazelcast, I managed to achieve noticeable improvements in application responsiveness and throughput. Understanding when and how to implement these caching strategies effectively is key to maximizing the benefits they offer.

15. How Do You Handle Lazy Loading in Spring Data JPA?

In Spring Data JPA, lazy loading is a strategy I often use to optimize the performance of my applications. Essentially, it allows me to defer the loading of related entities until they are explicitly accessed. This is in contrast to eager loading, where all related entities are loaded at once. For instance, when working with an entity that has a collection of child entities, using lazy loading ensures that the database hit for the child entities only occurs when I access this collection. To implement lazy loading, I annotate the relationships in my entities with @OneToMany(fetch = FetchType.LAZY) or similar, depending on the relationship type. This approach significantly reduces the initial load time and memory consumption, especially beneficial in situations where not all related entities are needed. It's crucial, however, to be mindful of the N+1 select issue and to use entity graphs or join fetch to mitigate it when accessing multiple related entities.

16. How Do You Handle Lazy Loading in Spring Data JPA?

In Spring Data JPA, lazy loading is a mechanism that I use to optimize the performance of my applications. It allows me to delay the loading of associated data until it's specifically requested. To implement lazy loading, I typically use the fetch attribute of the @OneToMany or @ManyToOne annotations, setting it to FetchType.LAZY. This approach helps in reducing the initial load time and memory consumption. However, it's crucial to access the lazily loaded data within the scope of an open session to avoid LazyInitializationException. To tackle the N+1 select issue that can arise with lazy loading, I make use of entity graphs or join fetch in JPQL queries, which allows me to specify the related entities to fetch eagerly, optimizing the number of queries fired against the database.

17. How Do You Handle Versioning in Spring Data JPA?

In my experience, handling versioning in Spring Data JPA is crucial for managing concurrent data access and ensuring data integrity. Spring Data JPA utilizes the @Version

annotation to implement optimistic locking. I use the @Version annotation on a version field in my entity classes. This way, every time an entity gets updated, JPA increments the version field, ensuring that the entity has not been altered by another transaction since it was fetched. If there's a discrepancy, JPA throws an OptimisticLockException. This approach has been beneficial in my projects, particularly in scenarios with high concurrency, as it prevents data corruption and loss, ensuring that every transaction operates on the latest state of the entity.

18. What Is the Difference Between Optimistic And Pessimistic Locking In Spring Data JPA?

In Spring Data JPA, optimistic locking uses a version field in the entity to track changes. Each time the entity is updated, the version number increments. If I try to save an entity with an outdated version number, Spring Data JPA throws an OptimisticLockingFailureException. This approach is lightweight and ideal for high-read environments, reducing the risk of database locks.

Pessimistic locking, on the other hand, locks the record for the duration of the transaction. This means that no other transaction can change the locked record until the current one completes. I use pessimistic locking in scenarios where I anticipate high contention on specific records. While it guarantees data integrity by preventing concurrent updates, it can lead to performance bottlenecks if not used judiciously.

19. How Do You Handle Inheritance in Spring Data JPA?

In Spring Data JPA, handling inheritance involves choosing among the single-table, table-per-class, or joined inheritance strategies. I decide based on the application's needs and the trade-offs in performance and database design. For instance, I use the single-table strategy for simplicity and performance when dealing with a small hierarchy with few distinct fields per class. In contrast, I opt for the joined strategy when dealing with a complex hierarchy, prioritizing data normalization and avoiding null columns. My choice always aims to balance between efficient database structure and optimal query performance, aligning with the project's goals.

20. What Is the Difference Between Single-Table, Table-Per-Class, and Joined Inheritance Strategies in Spring Data JPA?

In Spring Data JPA, inheritance strategies define how entities and their hierarchy are mapped to database structures. The Single-Table strategy maps all entities in the inheritance tree to a single table, distinguishing them with a discriminator column. This approach is efficient for querying but can lead to sparse tables if there are many attributes that are specific to subclasses.

The Table-Per-Class strategy creates a separate table for each entity in the hierarchy, which avoids nulls and ensures data normalization. However, it can lead to performance issues, especially with polymorphic queries, as it requires unions over multiple tables to retrieve data.

Lastly, the Joined strategy uses a table for each class but links them through foreign keys, combining normalization with efficient querying for relationships. This can be a balanced approach, though it might introduce join operations that can impact performance for deep inheritance trees. Selecting the right strategy depends on the specific requirements of your application, especially considering the trade-offs in terms of performance and database design.

21. How Do You Handle Associations in Spring Data JPA?

In handling associations in Spring Data JPA, I first determine the type of relationship between the entities, whether it's one-to-many, many-to-one, many-to-many, or one-to-one. For instance, in a one-to-many relationship, I use the @OneToMany annotation and configure it with the mappedBy attribute to establish the relationship from the other side. I pay close attention to the cascade types and fetch strategies because they significantly affect how entities are persisted and retrieved, impacting application performance. I usually go for a LAZY fetch strategy to avoid unnecessary data loading and select cascade types based on the specific needs of the operation, ensuring efficient and effective data management.

22. What Is the Difference Between mappedBy and @JoinColumn Annotations in Spring Data JPA?

In Spring Data JPA, mappedBy and @JoinColumn serve different purposes in relationships between entities. mappedBy is used to indicate the owner of a relationship. It's used in the entity that does not control the relationship, specifying the field in the owning entity that references back. For instance, in a one-to-many relationship, mappedBy would be used in the collection side, indicating it's mapped by the field in the owner entity.

On the other hand, @JoinColumn is used to specify the actual column used for joining an entity association or element collection. It's placed in the entity that owns the relationship, defining the name of the foreign key column. In a one-to-one or many-to-one relationship, @JoinColumn would be used in the entity that contains the foreign key column to establish the link. In practice, I use mappedBy for bi-directional relationships where I need to define the non-owning side, and @JoinColumn to denote the owning side and to customize the foreign key column in any association.

23. How Do You Handle Many-To-Many Relationships in Spring Data JPA?

In managing many-to-many relationships in Spring Data JPA, I use the @ManyToMany annotation. This requires a join table to connect the two sides of the relationship. For example, if I have entities Student and Course, each student can enroll in many courses, and each course can have many students. I define @ManyToMany annotations on both entities, specifying the @JoinTable on one side to establish the relationship table in the database. It's crucial to manage the relationship from both entities to keep the data consistent. Managing this relationship effectively allows for easier queries and operations on the data, ensuring efficient data access patterns.

24. How Do You Handle Bidirectional Relationships in Spring Data JPA?

In managing bidirectional relationships with Spring Data JPA, I pay close attention to defining the owning and inverse sides correctly. For a One-to-One or One-to-Many relationship, I use the mappedBy attribute on the non-owning side to link it back to the owning side. This helps in ensuring that JPA understands the relationship direction and can cascade operations appropriately. I also use the @JoinColumn annotation on the

owning side to specify the column used for joining an entity association. Handling these relationships carefully is crucial for maintaining data integrity and ensuring seamless navigation between associated entities.

25. How Do You Handle Cascading Operations In Spring Data JPA?

In Spring Data JPA, handling cascading operations is crucial for maintaining the integrity and consistency of data across related entities. When I define relationships in my entities, I always consider which cascading types are appropriate based on the use case. For instance, CascadeType.PERSIST allows me to save an entity and automatically persist its related entities, significantly reducing boilerplate code for saving each entity individually. In a recent project, I used CascadeType.REMOVE in a parent-child relationship to ensure that deleting a parent entity would automatically remove its related child entities, which streamlined the data management process. By carefully selecting cascading types, I ensure data consistency and simplify CRUD operations in my applications.

26. What Are The Different Types Of Cascading Operations In Spring Data JPA?

In Spring Data JPA, cascading operations are crucial for managing entity state transitions in relation to their parent-child relationships. Cascading types like PERSIST, MERGE, REMOVE, REFRESH, and DETACH dictate how operations applied to a parent entity affect its child entities. For instance, when using PERSIST, if a parent entity is persisted, its child entities are also persisted. In my projects, I often use MERGE during data updates to ensure modifications in the parent entity reflect in the child entities. Choosing the right cascading type depends on the specific requirements of the application, and understanding these nuances allows for more efficient data management and integrity.

27. How Do You Handle Composite Keys in Spring Data JPA?

In Spring Data JPA, handling composite keys involves using either the @IdClass or @EmbeddedId annotations. I prefer @EmbeddedId when I want to encapsulate the composite key in an embeddable class, making it part of the entity. This approach helps in keeping the entity class clean and focuses on the key's structure separately. On the other hand, @IdClass is useful when I want to keep the composite key's properties within the entity class itself. It's crucial to implement equals() and hashCode() methods in the

composite key classes to ensure proper identification and comparison of entity instances by JPA.

In my projects, when JPQL doesn't suffice for complex queries, I turn to native queries with Spring Data JPA. I use the @Query annotation, specifying my SQL directly and setting nativeQuery = true. This approach lets me leverage the full power of SQL while maintaining the simplicity and integration of Spring Data JPA. For example, if I need to perform a complex join not easily achievable with JPQL, I'll write the SQL query and annotate my repository method with @Query, ensuring my application can execute this high-performance query seamlessly.

29. How Do You Handle Stored Procedures in Spring Data JPA?

In Spring Data JPA, I manage stored procedures by leveraging the @Procedure annotation in my repository interfaces. This approach allows me to directly map the stored procedure from my database to a method in my Java application. I ensure that the procedure name and any parameters required are correctly specified within the annotation. This method simplifies the process of invoking stored procedures, making it more intuitive and integrated within the Spring Data JPA framework. By doing so, I can maintain a clean and efficient data access layer while utilizing the powerful features of stored procedures for complex database operations.

30. How Do You Handle Dynamic Queries in Spring Data JPA?

In dealing with dynamic queries in Spring Data JPA, I focus on the adaptability and efficiency of the data access layer. Utilizing the Criteria API allows me to construct complex queries programmatically, ensuring that I can cater to varying business requirements without compromising code maintainability. Similarly, by integrating Querydsl, I leverage its fluent API to build type-safe queries, enhancing both development speed and application stability. This approach ensures that my applications remain robust, scalable, and adaptable to changing business needs.

31. How Do You Handle Batch Updates in Spring Data JPA?

In handling batch updates with Spring Data JPA, I first ensure that batch processing is enabled in the application.properties file, as it significantly improves performance for large volumes of data. I use the @Transactional annotation to wrap multiple repository calls within a single transaction. This approach minimizes the I/O operations to the database. Additionally, I leverage the saveAll method provided by Spring Data repositories when dealing with collections of entities to be updated or inserted. This method optimizes the persistence process by batching the operations together, which is crucial for maintaining high performance in data-intensive applications.

32. How Do You Handle Bulk Deletes in Spring Data JPA?

In handling bulk deletes with Spring Data JPA, I ensure to carefully plan and test the operation to prevent any adverse effects on the database performance or data integrity. I start by creating a custom method in my repository interface. This method is annotated with @Modifying to indicate that it will modify the database. Alongside, I use the @Transactional annotation to ensure the operation is wrapped within a transaction, providing rollback capabilities in case of errors. For example, to delete all records that meet a certain condition, my method would look something like this:

```
@Modifying

@Transactional

@Query("DELETE FROM EntityName e WHERE e.condition = :condition")

void deleteByCondition(@Param("condition") String condition);
```

This approach allows me to efficiently execute bulk deletes while maintaining control over the transaction and ensuring data consistency.

33. How Do You Handle Concurrency Issues in Spring Data JPA?

In handling concurrency issues with Spring Data JPA, I primarily utilize two strategies: optimistic and pessimistic locking. Optimistic locking is my go-to choice for most scenarios because it allows for better performance by avoiding database locks. I use the @Version annotation on a version field in my entity classes. This approach ensures that if

two transactions attempt to update the same entity simultaneously, one will succeed, and the other will receive an exception, thus preventing data inconsistency.

SPRING BOOT

1) What is Spring Boot?

Spring Boot is a Spring module that provides RAD (Rapid Application Development) features to the Spring framework.

It is used to create stand-alone spring-based applications that you can just run because it needs very little spring configuration.

2) What are the advantages of Spring Boot?

- Create stand-alone Spring applications that can be started using java -jar.
- Embed Tomcat, Jetty, or Undertow directly. You don't need to deploy WAR files.
- It provides opinionated 'starter' POMs to simplify your Maven configuration.
- It automatically configures Spring whenever possible.

3 What are the features of Spring Boot?

Some key features of Spring Boot include:

- **Auto-configuration:** It automatically configures application components based on project dependencies.
- **Embedded web server support:** Spring Boot includes embedded servers like Tomcat, Jetty, and Undertow, making it easy to deploy web applications.
- **Spring Boot Starter:** Pre-defined templates for common use cases, simplifying dependency management.
- **Production:** ready metrics, health checks, and externalized configuration.
- **Spring Boot CLI:** Command Line Interface for rapid application development

4. How to create a Spring Boot application using Maven?

There are multiple approaches to creating a Spring Boot project. We can use any of the following approaches to develop an application.

- Spring Maven Project
- Spring Starter Project Wizard
- Spring Initializr
- Spring Boot CLI

5. How to create a simple Spring Boot application?

- To create an application, we can use STS (Spring Tool Suite) IDE. It includes the various steps that are explained in steps.

6 What is Spring Boot dependency management?

Spring Boot manages dependencies and configuration automatically. We do not need to specify the version for any of those dependencies.

Spring Boot upgrades all dependencies automatically when you upgrade Spring Boot.

7 . What are the Spring Boot properties?

Spring Boot provides various properties that can be specified inside our project's **application.properties** file. These properties have default values and you can set that inside the properties file. Properties are used to set values like server port number, database connection configuration, etc.

8 . What are the Spring Boot Starters?

Starters are a set of convenient dependency descriptors that we can include in our application.

Spring Boot provides built-in starters which makes development easier and rapid. For example, if we want to get started using Spring and JPA for database access, just include the **spring-boot-starter-data-jpa** dependency in your project.

9 . What is a Spring Boot Actuator?

Spring Boot provides an actuator to monitor and manage our application. The actuator is a tool that has HTTP endpoints. When the application is pushed to production, you can choose to manage and monitor your application using HTTP endpoints.

10. How to connect the Spring Boot application to the database using JDBC?

Spring Boot provides starters and libraries for connecting to our application with JDBC. Here, we are creating an application that connects with the MySql database. It includes the following steps to create and set up JDBC with Spring Boot

11. What is @RestController annotation in Spring Boot?

The @RestController is a stereotype annotation. It adds @Controller and @ResponseBody annotations to the class. We need to import org.springframework.web.bind.annotation package in our file in order to implement it.

12. What is @RequestMapping annotation in Spring Boot?

The **@RequestMapping** annotation is used to provide routing information. It tells the Spring that any HTTP request should map to the corresponding method. We need to import org.springframework.web.annotation package in our file.

13. How to create a Spring Boot application using Spring Starter Project Wizard?

There is one more way to create a Spring Boot project in STS (Spring Tool Suite). Creating a project by using IDE is always a convenient way. Follow the following steps in order to create a Spring Boot Application by using this wizard.

14. What are the major differences between Spring Vs. Spring Boot?

Spring is a web application framework based on Java. It provides tools and libraries to create a completely customized web application.

Spring Boot is a spring module that is used to create a spring application project that can just run.

15. In how many ways Spring Boot applications can be packaged for deployment?

Spring Boot Applications can be packaged for deployment in the following ways.

Deployment Artifact	Produced By	Target Environment
Executable JAR	Maven, Gradle, or Spring Boot CLI	Cloud environments, including Cloud Foundry and Heroku, as well as container deployment, such as with Docker.
WAR	Maven or Gradle	Java application servers or cloud environments such as Cloud Foundry.

16. How can you monitor and manage Spring Boot applications in a production environment?

In a production environment, we can monitor and manage Spring Boot applications by using tools like **Spring Boot Actuator**. It provides endpoints for **health checks, metrics,** and **logging**. For in-depth application monitoring and management, we can integrate monitoring solutions like **Prometheus** and **Grafana.**

17. What is the purpose of using @ComponentScan annotation in the Class Files?

The annotation **@ComponentScan** is used to specify the packages to scan for annotated components. It helps Spring to detect and register beans with annotations like **@Component, @Service, @Repository,** and **@Controller** automatically.

18. What are Spring Boot Actuator endpoints commonly used for monitoring and management of Spring Boot applications?

Spring Boot Actuator

provides various endpoints for monitoring and managing applications. Some commonly used endpoints are as follows:

- **/actuator/health:** It provides application health status.
- **/actuator/info:** It provides custom application information.
- **/actuator/metrics:** It exposes application metrics (for example, memory usage and request counts).
- **/actuator/env:** It displays environment properties.
- **/actuator/loggers:** It allows dynamic log-level configuration.

19. How do you enable an SSL certificate in a Spring Boot application?

1. Get a certificate from a Certificate Authority (CA).
2. Set up SSL properties in application.properties, including keystore path, password, and key alias.
3. Ensure that the server is configured to use HTTPS by setting the appropriate properties.

20. What is Spring Boot caching support, and why is it useful?

Spring Boot provides caching support through annotations like **@Cacheable, @CacheEvict**, and **@CachePut.** It is used to improve application performance by storing frequently accessed data in memory. Spring Boot simplifies the setup and management of caching mechanisms, such as Ehcache, Caffeine, and Redis, making it easier to implement caching strategies.

21. What type of applications can we design with the Spring Boot?

Besides the web applications, we can design **non-web applications, console applications, batch applications,** and **microservices** with the help of Spring Boot.

22. What are the differences between RequestMapping and GetMapping?

Features	@RequestMapping	@GetMapping
Annotations	@RequestMapping	@GetMapping
Purpose	It is used with various types of HTTP requests like GET, POST, etc.	Specifically handles HTTP GET requests.
Example	@RequestMapping(value = "/example", method = RequestMethod.GET)	@GetMapping("/example")

23. What are the differences between @Controller and @RestController?

Features	@Controller	@RestControllerth
Usage	It identifies a class as a controller class.	It is a combination of two annotations i.e @Controller and @ResponseBody.
Request handling and Mapping	We can use it while annotating a method with @RequestMapping to map HTTP requests.	It handles requests like GET, PUT, POST, and DELETE.
Application	It is used for Web applications.	It is used for RESTful APIs.

24. How to handle data validations in the Spring Boot application?

There are various mechanisms to handle data validations in the Spring Boot application. The most used way to do that is to leverage validation annotations defined by the Bean Validation API. For example, **@NotNull, @Size,** and **@Pattern**. It is used on the fields of object model.

By implementing these validation annotations, the Spring Boot application automatically validates the data and generates the validation errors. It also enables us to have our own validation logic by writing validation classes and methods.

25. What do you mean by Bean scope in Spring Boot?

Bean Scopes in Spring Boot application Bean scopes in Spring Boot application define the lifecycle and visibility of Spring-managed beans. The following are the most commonly used bean scopes:

- **Singleton:** A single instance of the bean will be created and shared in the entire application context.
- **Prototype:** Each time a request is made for the bean, a new instance of it is instantiated.
- **Request:** A new bean instance is created on every HTTP request. But it's only valid in a web application context.
- **Session:** A new bean instance will be created per user session. It applies just pertaining to web applications.
- **Custom Scopes:** We can define our own scopes in Spring Boot by implementing the Scope interface and registering custom scopes in the application context.

26. How to optimize Spring Boot application startup time?

- Reducing the number of beans loaded.
- Using lazy initialization.
- Restricting unused auto-configurations.
- Analyzing startup logs to identify slow components.

27. How do we manage database transactions in a Spring Boot application?

When managing database transactions, Spring Boot provides **@Transactional** annotation. By implementation, Spring Boot automatically operates transaction boundaries. Additionally, Spring Boot is integrated into a variety of data sources and JPA providers for uniform transaction management.

28. How do you deal with Spring Boot memory management?

We can handle memory management in the Spring Boot application as follows:

- Monitoring memory usage with Actuator endpoints.
- Configuring heap size and garbage collection settings.
- Detect memory leaks by using the profiling tools.
- Optimizing code to reduce memory footprint.

29. hich tools and libraries are helpful for Spring Boot development?

For Spring Boot application development, helpful tools and libraries include **Spring Tool Suite (STS), IntelliJ IDEA, Lombok,** and **MapStruct.**

30. How does the Constructor differ from the Setter Injection?

Injection Type	Description	Pros	Cons
Constructor Injection	In this, dependencies are provided via the constructor.	It ensures immutability and promotes mandatory dependencies.	Harder to handle optional dependencies
Setter Injection	In this, dependencies are provided via setter methods.	It is useful for optional dependencies and is more flexible.	It can lead to an inconsistent state if not handled properly.

31. How to configure a database in Spring Boot?

The following configuration is required to configure the database in the Spring Boot application. This configuration must be specified in the application.properties file.

1. spring.datasource.url=jdbc:mysql://localhost:3306/db_name
2. spring.datasource.username=root
3. spring.datasource.**password**=
4. spring.jpa.hibernate.ddl-auto=**update**