Q1. What are the three major categories of exceptions in Java's exception hierarchy? For each category, explain: (1) Whether it must be handled at compile-time, (2) Common examples, (3) Best practices for handling.

Java exceptions are divided into checked exceptions, unchecked exceptions (runtime exceptions), and errors. For checked exceptions, (1) they must be handled or declared at compile time, (2) common examples include IOException and SQLException, and (3) best practice is to either handle them with meaningful recovery logic or propagate them upward with context. For unchecked exceptions, (1) they do not require compile-time handling, (2) examples include NullPointerException and IllegalArgumentException, and (3) best practice is to prevent them through validation and correct logic rather than catching them broadly. For errors, (1) they are not meant to be handled at compile time, (2) examples include OutOfMemoryError and StackOverflowError, and (3) best practice is not to catch them, as they indicate serious JVM-level problems.

Q2. Explain the execution order of try-catch-finally blocks. If both the catch block and finally block contain return statements, which value will be returned? Why is it strongly discouraged to use return statements in finally blocks?

In a try-catch-finally structure, the try block executes first, followed by the matching catch block if an exception occurs, and the finally block always executes regardless of whether an exception was thrown. If both the catch and finally blocks contain return statements, the value from the finally block is returned. This is strongly discouraged because it suppresses the original return value or exception, making the control flow confusing and potentially hiding errors.

Q3. What is the "catch scope should be from small to large" rule? Why must specific exception types (like OrderNotFoundException) be caught before general ones (like Exception)? What happens if you violate this rule?

The rule that catch scope should be from small to large means that more specific exception types must be caught before more general ones. This is required because a general exception type would otherwise catch all its subclasses, making the specific catch blocks unreachable. If this rule is violated, the compiler reports an error indicating unreachable code.

Q4. Compare throw and throws: (1) Where is each used in code? (2) What follows each keyword? (3) Provide one practical example demonstrating both keywords working together in a DAO-Service-Controller architecture.

The throw keyword is used inside a method body to explicitly throw an exception object, while throws is used in a method signature to declare that the method may propagate certain exceptions. The throw keyword is followed by an exception instance, whereas throws is followed by exception class names. For example, a DAO method may declare throws SQLException, a service method may catch it and throw new ServiceException(...), and the controller method may declare throws ServiceException to let a global handler process it.

Q5. What is try-with-resources syntax (introduced in Java 7)? What interface must a class implement to be used with try-with-resources? Explain the execution order when multiple resources are declared.
Try-with-resources is a syntax that automatically closes resources after use, introduced in Java 7. A resource must implement the AutoCloseable interface to be used with this syntax. When multiple resources are declared, they are closed in the reverse order of their creation after the try block completes.

Q6. When creating custom exceptions, how do you decide between extending Exception vs extending RuntimeException? Provide criteria for each choice and one example scenario for each.
A custom exception should extend Exception when the error represents a recoverable condition that callers are expected to handle, such as a business rule violation. It should extend RuntimeException when the error indicates a programming mistake or an unrecoverable condition, such as invalid method arguments. For example, a BusinessException may extend Exception, while an InvalidStateException may extend RuntimeException.

Q7. Explain the two important features of Enum: "Every element is in values" and "Every element is a constructor". How would you implement an Enum with a private constructor that accepts parameters?
Every enum element is part of the implicit values() array, meaning all defined constants are known and enumerable at runtime. Every enum element is also an instance created via the enum's constructor. An enum with parameters is implemented by defining a private constructor that accepts arguments and assigning those arguments to fields for each enum constant.

Q8. Describe the popular Enum template pattern (Interface + Enum + Exception). What are its four components? How does using an interface type (IErrorCode) allow the exception class to accept multiple different enum types?
The popular Enum template pattern consists of an interface, one or more enums implementing that interface, a custom exception class, and a common error-handling mechanism. The interface defines shared methods such as getCode() and getMessage(). By typing the exception constructor to the interface rather than a concrete enum, the exception can accept different enum types that represent different error domains.

Q9. Compare the three major Collection interfaces: List, Set, and Queue. For each, explain: (1) Ordering characteristics, (2) Duplicate element handling, (3) Most commonly used implementation class, (4) One typical use case.
A List maintains insertion order, allows duplicates, commonly uses ArrayList, and is typically used for ordered, index-based access. A Set does not allow duplicates, may or may not preserve order, commonly uses HashSet, and is used to ensure uniqueness. A Queue follows a specific processing order such as FIFO, allows duplicates, commonly uses LinkedList, and is used for task scheduling or buffering.

Q10. Explain the difference between HashMap and Hashtable. Why is Hashtable considered obsolete? What are the modern alternatives for thread-safe Map implementations?

HashMap is not thread-safe and allows one null key and multiple null values, while Hashtable is thread-safe through synchronized methods and does not allow null keys or values. Hashtable is considered obsolete due to its coarse-grained synchronization and poor performance. Modern alternatives include ConcurrentHashMap for high-performance concurrency and Collections.synchronizedMap for simpler synchronization needs.