Chuwa - Java Backend 1217 Assignment

Delin "Angelina" Liang

18 Dec 2025

Question 1. 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.

ANSWER

Error

(1) **Compile-time handling requirement**:
Errors do not need to be handled at compile-time and generally should not be caught

(2) **Common examples**:

- `OutOfMemoryError`
- `StackOverflowError`

(3) **Best practices for handling**:
Since Errors generally indicate serious JVM-level problems that applications are not expected to recover from, instead of catching/throwing them around, the best practice is to avoid these kinds of issues through proper system design or memory management

Checked Exceptions

(1) **Compile-time handling requirement**:
Checked exceptions **must be handled at compile-time** either through a `try-catch` block or the `throws` keyword

(2) **Common examples**:

- `IOException` (e.g., `FileNotFoundException`)
- `SQLException`/`NetworkException`

(3) **Best practices for handling**:
As mentioned above, you should either:

- `catch` the exception when the program can take meaningful recovery actions, or
- `throws` them to an upper level for proper handling

Design-wise, one should avoid handling exceptions too broadly (e.g., just catching `Exception`)

Unchecked Exceptions

(1) **Compile-time handling requirement**:
Unchecked exceptions **do not need to be handled at compile-time** since they generally indicates **runtime issues**

(2) **Common examples**:

- `NullPointerException`
- `IndexOutOfBoundsException`

(3) **Best practices for handling**:
Runtime exceptions usually indicates **programming errors or logic bugs**, therefore, the best way to handle would include:

- Preventing them through proper validation and defensive coding
- Fixing the root cause first instead of catching all
- Only do handling when the caller cannot reasonably recover from the error

Question 2. 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?

ANSWER

Execution Order of `try-catch-finally`:
The order is as follows:

- The `try` block is executed first
- If in the `try` block an exception occurs, the corresponding `catch` block is executed
- The `finally` block is **always** executed regardless the exception is handled or not

If both `catch` and `finally` return, the receiving method will get the returned value from the **finally block**

The use of `return` statements in a `finally` block is strongly discouraged because:

- It **suppresses exceptions**, making errors harder to detect and debug
- It **overrides return values**, leading to confusing and unexpected behavior
- It **breaks the clarity of the program**, since `finally` is intended for cleanup

Question 3. 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?

ANSWER

The rule "catch scope should be from small to large"means that while during exception handling, **more specific exception types must be caught before more general exception types**

This is because in Java, exceptions are matched based on the order they are placed in the `catch` block. If a more general case of exception was caught first, the remaining sub-types that include more details will be **unreachable**. Therefore, to **ensure a more precise exception handling logic**, which further ensures the maintainability of our code, we would want to catch the more detailed exception cases first

If we place a general exception before a specific one, the code will **fail to compile**

Question 4. 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.

ANSWER

Where is each used in code?

- `throw` is used **inside a method body** to explicitly throw an exception at runtime
- `throws` is used in a **method's signature** to indicate that the method may pass an exception to its caller

What follows each keyword?

- `throw` is followed by **an exception object**
- `throws` is followed by **one or more exception types**

Provide one practical example demonstrating both keywords working together in a DAO-Service-Controller architecture

```
// DAO Layer
public class OrderDao {
    public Order findById(String id) throws OrderNotFoundException {
        Order order = database.find(id);
        if (order == null) {
            throw new OrderNotFoundException("Order not found");
        }
        return order;
    }
}

// Service Layer
public class OrderService {
    private OrderDao dao = new OrderDao();

    public Order getOrder(String id) throws OrderNotFoundException {
        return dao.findById(id);
    }
}

// Controller Layer
public class OrderController {
    private OrderService service = new OrderService();

    public void handleRequest(String id) {
        try {
            Order order = service.getOrder(id);
        } catch (OrderNotFoundException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Question 5. 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.

ANSWER

The try-with-resources syntax is a language feature that allows resources to be declared inside

the `try` statement so that they are **automatically closed** when the block finishes execution, which simplifies resource management

To be used with try-with-resources, a class must implement the `AutoCloseable` interface

Resources are **closed automatically in reverse order of creation**, following the **Last-In, First-Out (LIFO)** order

Question 6. 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.

ANSWER

In general, the choice between extending `Exception` and extending `RuntimeException` depends on whether the error condition is **recoverable** and whether it should be **enforced at compile time**

Case: Extending `Exception`
Criteria:

- The error represents a **recoverable or expected condition** which could be handled as a **checked exception**

Example:

```
public class InsufficientBalanceException extends Exception
```

Case: Extending `RuntimeException`
Criteria:

- The error represents a **programming error or invalid state** that the caller **cannot recover** from

Example:

```
public class OrderException extends RuntimeException
```

Question 7. 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?

ANSWER

"Every element is in values"
In Java, every `enum` automatically provides a static `values()` method that returns an array containing **all enum constants** in the order they are declared

"Every element is a constructor"
In Java, all `enum` constants are created by calling the **enum's constructor**, allowing `enum` elements to carry additional data and behavior

An enum with parameters is implemented by defining enum constants with arguments and using a private constructor to assign those values to internal fields. Example as follows:

```
public enum Status {
    SUCCESS(200, "Operation successful"),
    FAILED(500, "Operation failed");
```

```
        private int code;
        private String message;

        private Status(int code, String message) {
            this.code = code;
            this.message = message;
        }
    }
```

Question 8. 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?

ANSWER

Enum template pattern (Interface + Enum + Exception):
is a common design used to represent standardized error codes and messages in a flexible and extensible way

The Four Components

1. **Interface A** -> `getCode`, `getMessage`
2. **enum B** implements the **interface A**
3. private enum constructor
4. An exception can aggregate the interface/enum

Why Use an Interface Type in the Exception?
Because it allows the exception class to accept multiple different enum implementations, not just a single enum. This decouples the exception from specific error code enums, following the **Open–Closed Principle**, enabling new error code enums to be added without modifying the exception class

Question 9. 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.

ANSWER

List

(1) **Ordering characteristics**
Maintains **insertion order** and allows access by index

(2) **Duplicate element handling**
Duplicates are **allowed**

(3) **Most commonly used implementation class**
`ArrayList`

(4) **One typical use case**
Storing and processing an ordered collection of elements, such as ranking students by score

Set

(1) **Ordering characteristics**
Generally **does not guarantee ordering** (depending on implementation).

(2) **Duplicate element handling**
Duplicates are **not allowed**

(3) **Most commonly used implementation class**
`HashSet`

(4) **One typical use case**
Ensuring uniqueness, such as removing duplicate values from a collection

Queue

(1) **Ordering characteristics**
Follows **FIFO (First-In, First-Out)** ordering

(2) **Duplicate element handling**
Duplicates are **allowed**

(3) **Most commonly used implementation class**
`LinkedList`

(4) **One typical use case**
Managing tasks or requests that need to be processed in order, such as request/ticket handling

Question 10. Explain the difference between HashMap and Hashtable. Why is Hashtable considered obsolete? What are the modern alternatives for thread-safe Map implementations?

ANSWER

Difference between `HashMap` and `Hashtable` Why is `Hashtable` considered obsolete?

	HashMap	**Hashtable**
Thread Safety	Not thread-safe	Thread-safe (synchronized)
Synchronization	No synchronization	Method-level synchronization
Performance	Faster	Slower due to synchronization
Null Keys/Values	Allows one null key and multiple null values	Does not allow null keys or null values

Because it locks the entire map for every operation, which does not scale well in modern multi-threaded applications and leads to unnecessary contention and poor performance

What are the modern alternatives for thread-safe `Map` implementations?
`ConcurrentHashMap` - Provides high-performance thread safety using fine-grained locking and lock-free reads