Java Exception Handling Interview Questions and Answers

Author: Ramesh Fadatare

Core Java Exception Handling Interview

In this article, we will discuss important Java Exception Handling interview questions and answers.

1. What is an Exception in Java?

An exception in Java is an event that occurs during the execution of a program, disrupting the normal flow of instructions. Exceptions are objects that encapsulate information about an error condition that has occurred within a method or block of code.

2. How does Exception Handling Work in Java?

Java exception handling works by using a combination of `try`, `catch`, `finally`, `throw`, and `throws` keywords. When an exception occurs in a `try` block, it is thrown to the corresponding `catch` block. If the exception is not caught, it propagates up the call stack. The `finally` block is executed regardless of whether an exception was thrown or caught. The `throw` keyword is used to explicitly throw an exception, while the `throws` keyword is used to declare that a method might throw one or more exceptions.

3. What are the Exception Handling Keywords in Java?

The exception-handling keywords in Java are:

- `try`
- `catch`
- `finally`
- `throw`
- `throws`

4. What is the purpose of the throw and throws keywords?

- `throw`: Used to explicitly throw an exception in a method or block of code.
- `throws`: Used in a method signature to declare that the method might throw one or more exceptions.

5. How can you handle an exception?

You can handle an exception using a `try-catch` block. Place the code that might throw an exception inside the `try` block, and handle the exception in the `catch` block. Optionally, you can use a `finally` block to execute code regardless of whether an exception was thrown or caught.

Example:

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
} finally {
    // Code that will always execute
}
```

6. Explain Java Exception Hierarchy?

The exception hierarchy in Java is as follows:

```
java.lang.Object
   └── java.lang.Throwable
       ├── java.lang.Exception
       │   ├── java.io.IOException
       │   ├── java.sql.SQLException
       │   └── java.lang.RuntimeException
       │       ├── java.lang.NullPointerException
       │       ├── java.lang.ArrayIndexOutOfBoundsException
       │       └── java.lang.ArithmeticException
       └── java.lang.Error
           ├── java.lang.OutOfMemoryError
           ├── java.lang.StackOverflowError
           └── java.lang.VirtualMachineError
```

7. How can you catch multiple exceptions?

You can catch multiple exceptions by using multiple `catch` blocks or a single `catch` block that handles multiple exception types (Java 7 and later).

Example:

```
try {
    // Code that might throw multiple exceptions
} catch (IOException | SQLException e) {
    // Code to handle IOException or SQLException
} catch (Exception e) {
    // Code to handle other exceptions
}
```

8. What is the difference between Checked and Unchecked Exceptions in Java?

- **Checked Exceptions**: Checked at compile-time. Must be either caught or declared in the method signature using the `throws` keyword. Examples: `IOException`, `SQLException`.
- **Unchecked Exceptions**: Not checked at compile-time. They are subclasses of `RuntimeException`. Examples: `NullPointerException`, `ArrayIndexOutOfBoundsException`.

9. What is the difference between the throw and throws keywords in Java?

- **throw**: Used to explicitly throw an exception. **Example:**
- `throw new IllegalArgumentException("Invalid argument");`
- **throws**: Used in a method signature to declare that the method might throw one or more exceptions. **Example:**
- `public void method() throws IOException, SQLException {`
- `    // Method body`
- `}`

10. What is the difference between an exception and an error?

- **Exception**: Represents conditions that a program might want to catch and handle. They are recoverable conditions.
- **Error**: Represents serious issues that a reasonable application should not try to catch. They are usually external to the application and indicate problems with the environment, such as the Java Virtual Machine (JVM) running out of memory.

11. What is OutOfMemoryError in Java?

`OutOfMemoryError` is an error that occurs when the Java Virtual Machine (JVM) cannot allocate an object because it is out of memory. The JVM throws this error to indicate that the heap memory has been exhausted.

12. What are Chained Exceptions in Java?

Chained exceptions allow you to relate one exception with another, forming a chain of exceptions. This is useful when an exception occurs as a direct result of another exception. You can create a chained exception by passing the original exception as a parameter to the constructor of the new exception.

Example:

```
public class ChainedExceptionDemo {
    public static void main(String[] args) {
        try {
            method1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void method1() throws Exception {
        try {
            method2();
        } catch (Exception e) {
            throw new Exception("Exception in method1", e);
        }
    }

    public static void method2() throws Exception {
        throw new Exception("Exception in method2");
    }
}
```

13. How to write custom exceptions in Java?

You can create custom exceptions by extending the `Exception` class or any of its subclasses. Custom exceptions are useful for specific error conditions relevant to your application.

Example:

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught custom exception: " + e.getMessage());
        }
    }

    public static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older.");
        }
        System.out.println("Age is valid.");
    }
}
```

14. What is the difference between final, finally, and finalize in Java?

- **final**: A keyword used to define constants, prevent inheritance, and prevent method overriding.
- **finally**: A block that is executed after the `try-catch` block, regardless of whether an exception was thrown or caught.
- **finalize**: A method called by the garbage collector before an object is destroyed. It is used to perform cleanup operations.

15. What happens when an exception is thrown by the main method?

If an exception is thrown by the `main` method and not caught within the `main` method, the JVM handles it by printing the stack trace to the standard error stream and terminating the program.

16. What is a try-with-resources statement?

The `try-with-resources` statement is a `try` statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The `try-with-resources` statement ensures that each resource is closed at the end of the statement.

Example:

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

17. What is a stack trace and how does it relate to an exception?

A stack trace is a list of method calls that the application was in the middle of when an exception was thrown. It provides information about the sequence of method calls that led to the exception, helping developers debug the error by showing the exact point where the exception occurred.

18. What are the Advantages of Java Exceptions?

- **Separation of Error Handling Code from Regular Code**: Improves readability and maintainability of code.
- **Propagating Errors Up the Call Stack**: Allows a method to catch and handle exceptions thrown by methods it calls.
- **Grouping and Differentiating Error Types**: Provides a way to handle different types of errors in different ways.

19. Can you throw any exception inside a lambda expression’s body?

Yes, you can throw exceptions inside a lambda expression's body. However, if the exception is a checked exception, the functional interface method that the lambda expression implements must declare that it throws the exception.

Example:

```
@FunctionalInterface
interface ThrowingConsumer<T> {
    void accept(T t) throws Exception;
}

public class LambdaExceptionDemo {
    public static void main(String[] args) {
        ThrowingConsumer<Integer> consumer = (Integer i) -> {
            if (i < 0) {
                throw new Exception("Negative value not allowed");
            }
            System.out.println(i);
        };

        try {
            consumer.accept(-1);
        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }
}
```

20. What are the rules we need to follow when overriding a method that throws an exception?

When overriding a method that throws an exception:

- The overridden method can only throw the same exceptions or subclasses of the exceptions declared by the parent method.
- The overridden method cannot throw new or broader checked exceptions than those declared by the parent method

.

Example:

```
class Parent {
    public void method() throws IOException {
        // Method body
    }
}

class Child extends Parent {
    @Override
    public void method() throws FileNotFoundException {
        // Method body
    }
}
```

21. Java Exception Handling Best Practices

- **Catch Specific Exceptions**: Catch the most specific exception first to handle known error conditions.
- **Use Finally Block for Cleanup**: Ensure that resources are properly closed using the `finally` block or `try-with-resources`.
- **Log Exceptions**: Log exceptions with sufficient details to help with debugging.
- **Throw Custom Exceptions for Business Logic Errors**: Create custom exceptions for specific error conditions relevant to your application.
- **Avoid Swallowing Exceptions**: Do not catch exceptions without handling them or rethrowing them.

Example of Best Practices:

```
public class ExceptionHandlingBestPractices {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new
FileReader("file.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.err.println("File not found: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("I/O error: " + e.getMessage());
        }
    }
}
```

By following these best practices, you can write robust and maintainable code that handles exceptions effectively, ensuring your application can recover gracefully from unexpected errors.