DEBUGGING

EXERCISES

1. DEBUGGING VS. TESTING

A. TESTING

- Write JUnit tests for a method that checks if a given string is a palindrome. Consider different cases like empty strings, strings with spaces, and strings with mixed-case letters.

B. DEBUGGING

- Introduce a bug in the palindrome checking method (e.g., incorrect comparison logic) and use the IntelliJ IDEA debugger to identify and fix the bug.

C. COMBINED

- Write a method that calculates the average of an array of numbers. First, write JUnit tests to cover different scenarios (empty array, array with positive numbers, array with negative numbers). Then, introduce a bug in the method (e.g., incorrect sum calculation) and use the debugger to find and fix the bug.

2. TYPES OF ERRORS

A. SYNTAX ERROR

- Introduce a syntax error (e.g., missing semicolon, mismatched brace) into a simple Java program and observe the compiler error message. Fix the error and recompile the code.

B. RUNTIME ERROR

- Write a program that intentionally throws a NullPointerException or ArrayIndexOutOfBoundsException. Use a try-catch block to handle the exception and print an informative error message.

C. LOGIC ERROR

- Create a program with a logic error (e.g., incorrect calculation, wrong conditional statement). Use print statements or the debugger to identify the error and correct it. For example, write a program to calculate the factorial of a number, but introduce an off-by-one error in the loop.

3. LOGGING

A. IDENTIFY LOGGING OPPORTUNITIES

- Consider a simple Java application you've worked on (e.g., a basic calculator or a "Hello, World!" program). Identify at least three places where adding logging statements would be beneficial for debugging or monitoring. Specify what information you would log at each location and at what logging level (e.g., DEBUG, INFO, WARN, ERROR).

B. DESIGN A LOG MESSAGE FORMAT

- Create a template for a log message format that includes a timestamp, logging level, class name, method name, and a message. Explain why you chose the specific format and how it would improve log readability and analysis.

C. ERROR HANDLING AND LOGGING

- Write a short Java code snippet that simulates a potential error scenario (e.g., dividing by zero, accessing an out-of-bounds array element). Implement error handling using a try-catch block and log the exception details (including the stack trace) at the ERROR level.

4. JAVA LOGGING FRAMEWORKS

A. Log4j 2 CONFIGURATION

- Modify the log4j2.xml file to log messages with level WARN or higher to a separate file appender.

B. SLF4j IMPLEMENTATION SWITCHING

- Change the SLF4j implementation from Logback to Log4j 2. What changes are required in your project configuration and code?

C. java.util.logging CUSTOM HANDLER

- Create a custom Handler for JUL that writes log messages to a database table.

5. LOGGING LEVELS

A. MODIFY THE LOGGING LEVEL

- Change the root logger level in the log4j2.xml file to debug. Run the LoggingExample class again. What output do you see? Now, change it to warn. What changes?

B. IMPLEMENT CONDITIONAL LOGGING

- Modify the LoggingExample class to log a DEBUG message only if a certain condition is met (e.g., a boolean variable is true).

C. CREATE A CUSTOM LOGGER

- Create a new class and create a logger for that class. Log messages at different levels from this new class.

D. SIMULATE DIFFERENT ERROR SCENARIOS

- Modify the LoggingExample class to simulate different error scenarios (e.g., invalid input, network timeout) and log appropriate messages at the WARN or ERROR level.

6. CONFIGURING LOGS

- Configure a Log4j 2 application to log messages to both the console and a file. Set the console appender to log DEBUG messages and the file appender to log INFO messages.
- Configure a Logback application to use a rolling file appender. Set the rolling policy to rotate log files daily and keep a maximum of 7 days of history.
- Research and configure a database appender for either Log4j 2 or Logback. Write log messages to a database table.

7. INTELLIJ IDEA DEBUGGER

A. LINE BREAKPOINT

- Write a simple Java program that calculates the factorial of a number. Set a line breakpoint inside the loop to observe the intermediate values.

B. METHOD BREAKPOINT

- Create a class with multiple methods. Set method breakpoints on different methods and observe the order in which they are hit during execution.

C. FIELD WATCHPOINT

- Create a class with a private field. Set a field watchpoint on the field and observe when it's accessed and modified.

D. EXCEPTION BREAKPOINT

- Write a program that intentionally throws a NullPointerException. Set an exception breakpoint for NullPointerException and observe the program pausing when the exception is thrown.