Introduction to Java and Core OOP Concepts

Dr. Ratnesh Prasad Srivastava
Department of CSIT, GGV, Bilaspur (C.G.)

Academic Year: 2026-27

Course Information

Course Code	CIUDMJT1
Course Title	Object-Oriented Programming with Java
Credit Hours	3-0-3 (3 Lecture, 0 Tutorial, 3 Practical)
Prerequisites	Programming Fundamentals
Textbook	"Java: The Complete Reference" by Herbert Schildt
Reference	"Head First Java" by Kathy Sierra and Bert Bates

Contents

1 Unit III: Exception Handling, I/O, and Collections Framework

1.1 Learning Objectives

- Master exception handling mechanisms for robust Java applications
- Understand and implement different types of exceptions with proper handling
- Perform file operations using various I/O streams and readers/writers
- Utilize Java Collections Framework for efficient data manipulation
- Implement generics for type-safe collections and reusable code
- Design robust applications with proper error handling and data management

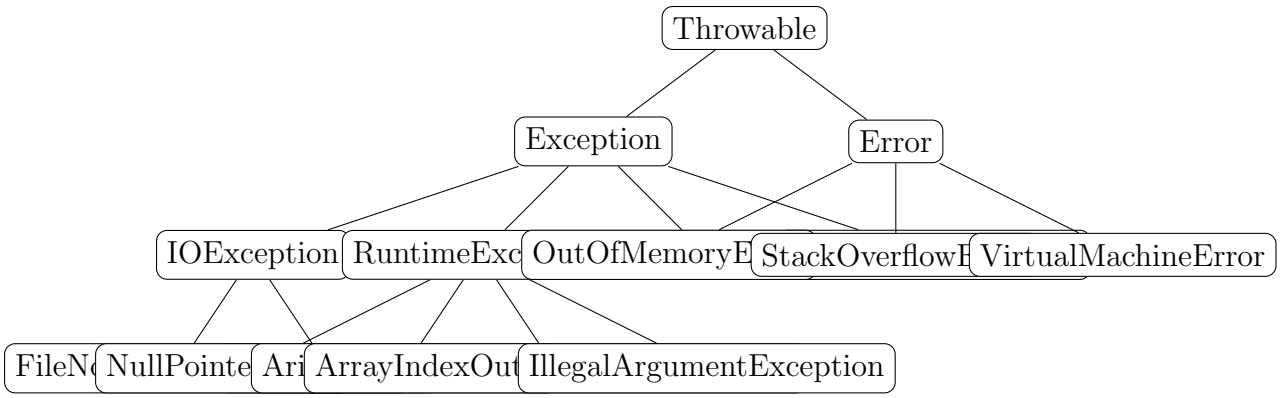
2 Exception Handling

2.1 Introduction to Exceptions

Exception Concept

An **exception** is an event that occurs during program execution that disrupts the normal flow of instructions. In Java, exceptions are objects that wrap error conditions. The exception handling mechanism separates error-handling code from regular code, making programs more robust and maintainable.

2.2 Exception Hierarchy



2.3 Types of Exceptions

```
// ==================== CHECKED EXCEPTIONS ====================
// Checked exceptions are checked at compile-time
// Must be handled using try-catch or declared with throws

import java.io.*;
```

```
6
7  public class ExceptionTypesDemo {
8
9      // Method that throws checked exception
10     public static void readFile(String filename) throws IOException {
11         // File operations can throw IOException (checked)
12         FileReader reader = new FileReader(filename);
13         reader.close();
14     }
15
16     // ==================== UNCHECKED EXCEPTIONS ====================
17     // Unchecked exceptions (RuntimeException) are not checked at
       compile-time
18
19     public static void demonstrateUncheckedExceptions() {
20         System.out.println("\n=== UNCHECKED EXCEPTIONS DEMONSTRATION
           ===\n");
21
22         // 1. NullPointerException
23         System.out.println("1. NullPointerException:");
24         try {
25             String str = null;
26             int length = str.length();  // Will throw
                   NullPointerException
27             System.out.println("Length: " + length);
28         } catch (NullPointerException e) {
29             System.out.println("Caught NullPointerException: " + e.
                   getMessage());
30             System.out.println("Cause: Trying to call method on null
                   object");
31         }
32
33         // 2. ArithmeticException
34         System.out.println("\n2. ArithmeticException:");
35         try {
36             int result = 10 / 0;  // Division by zero
37             System.out.println("Result: " + result);
38         } catch (ArithmeticException e) {
39             System.out.println("Caught ArithmeticException: " + e.
                   getMessage());
40             System.out.println("Cause: Division by zero");
41         }
42
43         // 3. ArrayIndexOutOfBoundsException
44         System.out.println("\n3. ArrayIndexOutOfBoundsException:");
45         try {
46             int[] numbers = {1, 2, 3};
47             System.out.println("Element at index 5: " + numbers[5]);
                   // Invalid index
48         } catch (ArrayIndexOutOfBoundsException e) {
49             System.out.println("Caught ArrayIndexOutOfBoundsException:
                   " + e.getMessage());
50             System.out.println("Cause: Accessing array beyond its
                   bounds");
51         }
52
53         // 4. NumberFormatException
54         System.out.println("\n4. NumberFormatException:");
```

```
55          try {
56              String invalidNumber = "abc123";
57              int num = Integer.parseInt(invalidNumber);  // Invalid
                    format
58              System.out.println("Number: " + num);
59          } catch (NumberFormatException e) {
60              System.out.println("Caught NumberFormatException: " + e.
                    getMessage());
61              System.out.println("Cause: Invalid string format for number
                     conversion");
62          }
63
64          // 5. ClassCastException
65          System.out.println("\n5. ClassCastException:");
66          try {
67              Object obj = "Hello";
68              Integer num = (Integer) obj;  // Invalid cast
69              System.out.println("Number: " + num);
70          } catch (ClassCastException e) {
71              System.out.println("Caught ClassCastException: " + e.
                    getMessage());
72              System.out.println("Cause: Invalid type casting");
73          }
74
75          // 6. IllegalArgumentException
76          System.out.println("\n6. IllegalArgumentException:");
77          try {
78              setAge(-5);  // Invalid age
79          } catch (IllegalArgumentException e) {
80              System.out.println("Caught IllegalArgumentException: " + e.
                    getMessage());
81          }
82      }
83
84      private static void setAge(int age) {
85          if (age < 0) {
86              throw new IllegalArgumentException("Age cannot be negative:
                     " + age);
87          }
88          System.out.println("Age set to: " + age);
89      }
90
91      // ==================== ERROR DEMONSTRATION ====================
92      // Errors are serious problems that applications should not try to
            catch
93
94      public static void demonstrateErrors() {
95          System.out.println("\n=== ERRORS DEMONSTRATION ===\n");
96
97          System.out.println("1. StackOverflowError:");
98          try {
99              recursiveMethod(0);  // Will cause stack overflow
100         } catch (StackOverflowError e) {
101             System.out.println("Caught StackOverflowError (not
                    recommended to catch)");
102             System.out.println("Cause: Infinite recursion or deep
                    recursion");
103         }
```

```
104
105         System.out.println("\n2. OutOfMemoryError Simulation:");
106         System.out.println("Note: Actual OutOfMemoryError not triggered
                for safety");
107         System.out.println("Cause: Excessive memory allocation");
108     }
109
110     private static void recursiveMethod(int counter) {
111         // Infinite recursion causing StackOverflowError
112         recursiveMethod(counter + 1);
113     }
114
115     // ==================== MAIN METHOD ====================
116     public static void main(String[] args) {
117         System.out.println("=== EXCEPTION TYPES DEMONSTRATION ===\n");
118
119         System.out.println("=== CHECKED VS UNCHECKED EXCEPTIONS ===");
120         System.out.println("Checked Exceptions:");
121         System.out.println("  - Checked at compile time");
122         System.out.println("  - Must be handled (try-catch) or declared
                (throws)");
123         System.out.println("  - Examples: IOException, SQLException");
124         System.out.println("\nUnchecked Exceptions (RuntimeException):"
               );
125         System.out.println("  - Not checked at compile time");
126         System.out.println("  - Can be handled but not required");
127         System.out.println("  - Examples: NullPointerException,
               ArithmeticException");
128         System.out.println("\nErrors:");
129         System.out.println("  - Serious problems beyond application
               control");
130         System.out.println("  - Should not be caught typically");
131         System.out.println("  - Examples: OutOfMemoryError,
               StackOverflowError");
132
133         // Demonstrate unchecked exceptions
134         demonstrateUncheckedExceptions();
135
136         // Demonstrate errors
137         demonstrateErrors();
138
139         // Demonstrate checked exception handling
140         System.out.println("\n=== CHECKED EXCEPTION HANDLING ===");
141         System.out.println("File reading example (commented for safety)
               :");
142         System.out.println("// Must handle IOException or declare
               throws");
143         System.out.println("// try {");
144         System.out.println("//     readFile(\"test.txt\");");
145         System.out.println("// } catch (IOException e) {");
146         System.out.println("//     System.out.println(\"File not found:
                \" + e.getMessage());");
147         System.out.println("// }");
148
149         // Real checked exception example
150         System.out.println("\nReal checked exception example:");
151         try {
152             // This will throw FileNotFoundException (checked)
```

```
153             FileInputStream fis = new FileInputStream("nonexistent.txt");
154             fis.close();
155         } catch (FileNotFoundException e) {
156             System.out.println("Caught FileNotFoundException: " + e.getMessage());
157             System.out.println("This is a checked exception - must be handled");
158         } catch (IOException e) {
159             System.out.println("Caught IOException: " + e.getMessage());
160         }
161 
162         // Custom exception demonstration
163         System.out.println("\n=== CUSTOM EXCEPTION EXAMPLE ===");
164         try {
165             validatePassword("weak");  // Too short
166         } catch (InvalidPasswordException e) {
167             System.out.println("Caught custom exception: " + e.getMessage());
168             System.out.println("Error Code: " + e.getErrorCode());
169         }
170     }
171 
172     // ==================== CUSTOM EXCEPTION ====================
173     // Custom checked exception
174     static class InvalidPasswordException extends Exception {
175         private int errorCode;
176 
177         public InvalidPasswordException(String message, int errorCode) {
178             super(message);
179             this.errorCode = errorCode;
180         }
181 
182         public int getErrorCode() {
183             return errorCode;
184         }
185     }
186 
187     private static void validatePassword(String password) throws InvalidPasswordException {
188         if (password == null || password.length() < 8) {
189             throw new InvalidPasswordException(
190                 "Password must be at least 8 characters long",
191                 1001
192             );
193         }
194         System.out.println("Password is valid");
195     }
196 }
```

Listing 1: Exception Types and Classification

ExceptionTypesDemo Program Output

```
=== EXCEPTION TYPES DEMONSTRATION ===

=== CHECKED VS UNCHECKED EXCEPTIONS ===
Checked Exceptions:
  - Checked at compile time
  - Must be handled (try-catch) or declared (throws)
  - Examples: IOException, SQLException

Unchecked Exceptions (RuntimeException):
  - Not checked at compile time
  - Can be handled but not required
  - Examples: NullPointerException, ArithmeticException

Errors:
  - Serious problems beyond application control
  - Should not be caught typically
  - Examples: OutOfMemoryError, StackOverflowError

=== UNCHECKED EXCEPTIONS DEMONSTRATION ===

1. NullPointerException:
Caught NullPointerException: null
Cause: Trying to call method on null object

2. ArithmeticException:
Caught ArithmeticException: / by zero
Cause: Division by zero

3. ArrayIndexOutOfBoundsException:
Caught ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
Cause: Accessing array beyond its bounds

4. NumberFormatException:
Caught NumberFormatException: For input string: "abc123"
Cause: Invalid string format for number conversion

5. ClassCastException:
Caught ClassCastException: java.lang.String cannot be cast to java.lang.Integer
Cause: Invalid type casting

6. IllegalArgumentException:
Caught IllegalArgumentException: Age cannot be negative: -5

=== ERRORS DEMONSTRATION ===

1. StackOverflowError:
```

```
Caught StackOverflowError (not recommended to catch)
Cause: Infinite recursion or deep recursion

2. OutOfMemoryError Simulation:
Note: Actual OutOfMemoryError not triggered for safety
Cause: Excessive memory allocation

=== CHECKED EXCEPTION HANDLING ===
File reading example (commented for safety):
// Must handle IOException or declare throws
// try {
//     readFile("test.txt");
// } catch (IOException e) {
//     System.out.println("File not found: " + e.getMessage());
// }

Real checked exception example:
Caught FileNotFoundException: nonexistent.txt (No such file or directory)
This is a checked exception - must be handled

=== CUSTOM EXCEPTION EXAMPLE ===
Caught custom exception: Password must be at least 8 characters long
Error Code: 1001
```

2.4 Try-Catch-Finally Block

```java
import java.io.*;
import java.util.Scanner;

public class TryCatchFinallyDemo {

    // ==================== BASIC TRY-CATCH ====================
    public static void basicTryCatch() {
        System.out.println("\n=== BASIC TRY-CATCH ===");

        try {
            int[] numbers = {1, 2, 3};
            System.out.println("Accessing element at index 5...");
            System.out.println("Element: " + numbers[5]);  // Will
                throw exception
            System.out.println("This line won't execute");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e.getClass().
                getSimpleName());
            System.out.println("Message: " + e.getMessage());
            System.out.println("Stack Trace (first 3 lines):");
            StackTraceElement[] stackTrace = e.getStackTrace();
            for (int i = 0; i < Math.min(3, stackTrace.length); i++) {
                System.out.println("  at " + stackTrace[i]);
            }
        }
```

```
24
25          System.out.println("Program continues after exception handling");
26      }
27
28      // ==================== MULTIPLE CATCH BLOCKS ====================
29      public static void multipleCatchBlocks() {
30          System.out.println("\n=== MULTIPLE CATCH BLOCKS ===");
31
32          Scanner scanner = new Scanner(System.in);
33
34          try {
35              System.out.print("Enter numerator: ");
36              int numerator = Integer.parseInt(scanner.nextLine());
37
38              System.out.print("Enter denominator: ");
39              int denominator = Integer.parseInt(scanner.nextLine());
40
41              System.out.print("Enter array index (0-2): ");
42              int index = Integer.parseInt(scanner.nextLine());
43
44              int[] numbers = {10, 20, 30};
45
46              // Multiple operations that can throw different exceptions
47              int result = numerator / denominator;
48              int arrayElement = numbers[index];
49
50              System.out.println("Division result: " + result);
51              System.out.println("Array element: " + arrayElement);
52
53          } catch (NumberFormatException e) {
54              System.out.println("Invalid number format: " + e.getMessage());
55              System.out.println("Please enter valid integers");
56
57          } catch (ArithmeticException e) {
58              System.out.println("Arithmetic error: " + e.getMessage());
59              System.out.println("Cannot divide by zero");
60
61          } catch (ArrayIndexOutOfBoundsException e) {
62              System.out.println("Array index error: " + e.getMessage());
63              System.out.println("Index must be between 0 and 2");
64
65          } catch (Exception e) {
66              // General catch block (should be last)
67              System.out.println("General exception: " + e.getClass().getSimpleName());
68              System.out.println("Message: " + e.getMessage());
69          }
70
71          scanner.close();
72      }
73
74      // ==================== MULTI-CATCH (Java 7+) ====================
75      public static void multiCatchExample() {
76          System.out.println("\n=== MULTI-CATCH (Java 7+) ===");
77
78          try {
```

```
79              String input = null;
80              // Uncomment one of these lines to test different
                    exceptions
81              // int number = Integer.parseInt(input);  //
                    NullPointerException
82              // int number = Integer.parseInt("abc");  //
                    NumberFormatException
83              int number = Integer.parseInt("123");  // Valid
84
85              int result = 100 / (number - 123);  // Division by zero if
                    number == 123
86
87              System.out.println("Result: " + result);
88
89          } catch (NullPointerException | NumberFormatException e) {
90              // Multi-catch: handles multiple exception types
91              System.out.println("Input error: " + e.getClass().
                    getSimpleName());
92              System.out.println("Message: " + e.getMessage());
93
94          } catch (ArithmeticException e) {
95              System.out.println("Calculation error: " + e.getMessage());
96          }
97      }
98
99      // ==================== FINALLY BLOCK ====================
100     public static void finallyBlockDemo() {
101         System.out.println("\n=== FINALLY BLOCK ===");
102
103         FileInputStream fis = null;
104         FileOutputStream fos = null;
105
106         try {
107             System.out.println("Opening file for reading...");
108             fis = new FileInputStream("input.txt");  // May throw
                    FileNotFoundException
109
110             System.out.println("Opening file for writing...");
111             fos = new FileOutputStream("output.txt");  // May throw
                    FileNotFoundException
112
113             System.out.println("Performing file operations...");
114             // Simulate file operations
115             int data;
116             while ((data = fis.read()) != -1) {  // May throw
                    IOException
117                 fos.write(data);
118             }
119
120             System.out.println("File copy completed successfully");
121
122         } catch (FileNotFoundException e) {
123             System.out.println("File not found: " + e.getMessage());
124
125         } catch (IOException e) {
126             System.out.println("IO Error: " + e.getMessage());
127
128         } finally {
```

```
            System.out.println("\nExecuting FINALLY block...");
            System.out.println("Cleaning up resources...");

            try {
                if (fis != null) {
                    fis.close();
                    System.out.println("Input stream closed");
                }
                if (fos != null) {
                    fos.close();
                    System.out.println("Output stream closed");
                }
            } catch (IOException e) {
                System.out.println("Error closing streams: " + e.
                    getMessage());
            }

            System.out.println("Finally block completed");
        }

        System.out.println("Method execution completed");
    }

    // ==================== TRY-WITH-RESOURCES (Java 7+)
        ====================
    public static void tryWithResourcesDemo() {
        System.out.println("\n=== TRY-WITH-RESOURCES ===");

        // Auto-closable resources declared in try statement
        try (FileReader reader = new FileReader("test.txt");
             BufferedReader br = new BufferedReader(reader);
             FileWriter writer = new FileWriter("output.txt");
             BufferedWriter bw = new BufferedWriter(writer)) {

            System.out.println("Resources automatically initialized");
            System.out.println("Reading and writing files...");

            String line;
            int lineCount = 0;
            while ((line = br.readLine()) != null) {
                bw.write("Line " + (++lineCount) + ": " + line);
                bw.newLine();
            }

            System.out.println("Processed " + lineCount + " lines");
            System.out.println("Resources will be automatically closed"
                );

        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("IO Error: " + e.getMessage());
        }  // No finally needed for closing resources

        System.out.println("Method completed - resources automatically
            closed");
    }

```

```
    // ==================== NESTED TRY-CATCH ====================
    public static void nestedTryCatch() {
        System.out.println("\n=== NESTED TRY-CATCH ===");

        try {
            System.out.println("Outer try block");

            int[] numbers = {1, 2, 3};

            try {
                System.out.println("Inner try block 1");
                System.out.println("Accessing array...");
                System.out.println("Element: " + numbers[5]);  // Will
                    throw exception

            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Inner catch: Array index error
                    handled");

                try {
                    System.out.println("Inner-inner try block");
                    String str = null;
                    System.out.println("String length: " + str.length()
                        );  // Another exception

                } catch (NullPointerException ne) {
                    System.out.println("Inner-inner catch: Null pointer
                        handled");
                    throw ne;  // Re-throwing exception
                }

            } finally {
                System.out.println("Inner finally block executed");
            }

            System.out.println("This won't execute due to re-thrown
                exception");

        } catch (NullPointerException e) {
            System.out.println("Outer catch: Caught re-thrown exception
                : " + e.getMessage());

        } finally {
            System.out.println("Outer finally block executed");
        }

        System.out.println("Program continues...");
    }

    // ==================== RETURN IN TRY-CATCH-FINALLY
        ====================
    public static int returnInTryCatchFinally() {
        System.out.println("\n=== RETURN IN TRY-CATCH-FINALLY ===");

        try {
            System.out.println("Try block executing");
            // throw new RuntimeException("Test exception");  //
                Uncomment to test
```

```
233             System.out.println("Returning from try block");
234             return 1;  // This value will be overridden by finally if
                    it returns
235
236         } catch (RuntimeException e) {
237             System.out.println("Catch block executing: " + e.getMessage
                    ());
238             return 2;  // This value will be overridden by finally if
                    it returns
239
240         } finally {
241             System.out.println("Finally block executing");
242             // return 3;  // Warning: finally block returns normally
243             // If uncommented, this will override previous returns
244         }
245
246         // return 4;  // Unreachable if finally has return
247     }
248
249     // ==================== EXCEPTION PROPAGATION ====================
250     public static void methodA() {
251         System.out.println("\n=== EXCEPTION PROPAGATION ===");
252         System.out.println("Method A called");
253         try {
254             methodB();
255         } catch (ArithmeticException e) {
256             System.out.println("Method A caught exception: " + e.
                    getMessage());
257         }
258     }
259
260     public static void methodB() {
261         System.out.println("Method B called");
262         methodC();
263     }
264
265     public static void methodC() {
266         System.out.println("Method C called");
267         // Exception occurs here and propagates up the call stack
268         int result = 10 / 0;
269         System.out.println("Result: " + result);  // Won't execute
270     }
271
272     // ==================== MAIN METHOD ====================
273     public static void main(String[] args) {
274         System.out.println("=== COMPREHENSIVE TRY-CATCH-FINALLY
                DEMONSTRATION ===\n");
275
276         // Basic try-catch
277         basicTryCatch();
278
279         // Multiple catch blocks
280         multipleCatchBlocks();
281
282         // Multi-catch
283         multiCatchExample();
284
285         // Finally block
```

```
286             finallyBlockDemo();
287
288             // Try-with-resources
289             tryWithResourcesDemo();
290
291             // Nested try-catch
292             nestedTryCatch();
293
294             // Return in try-catch-finally
295             int result = returnInTryCatchFinally();
296             System.out.println("Method returned: " + result);
297
298             // Exception propagation
299             methodA();
300
301             System.out.println("\n=== BEST PRACTICES ===");
302             System.out.println("1. Always close resources in finally block
                    or use try-with-resources");
303             System.out.println("2. Catch specific exceptions before general
                     ones");
304             System.out.println("3. Don't catch exceptions you can't handle
                    properly");
305             System.out.println("4. Use finally for cleanup, not for
                    business logic");
306             System.out.println("5. Log exceptions with meaningful messages"
                    );
307             System.out.println("6. Avoid empty catch blocks");
308             System.out.println("7. Don't use exceptions for normal flow
                    control");
309
310             System.out.println("\n=== FINALLY BLOCK GUARANTEES ===");
311             System.out.println("Finally block ALWAYS executes (except in
                    rare cases):");
312             System.out.println("  - System.exit() called");
313             System.out.println("  - JVM crashes");
314             System.out.println("  - Infinite loop in try/catch");
315             System.out.println("  - Thread death");
316
317             System.out.println("\n=== TRY-WITH-RESOURCES ADVANTAGES ===");
318             System.out.println("1. Automatic resource management");
319             System.out.println("2. Cleaner code - no explicit finally block
                     needed");
320             System.out.println("3. Resources closed in reverse order of
                    declaration");
321             System.out.println("4. Supports multiple resources");
322             System.out.println("5. Resources must implement AutoCloseable
                    interface");
323
324             // Real-world example
325             System.out.println("\n=== REAL-WORLD EXAMPLE: DATABASE
                    CONNECTION ===");
326             simulateDatabaseOperation();
327         }
328
329         // ==================== REAL-WORLD DATABASE EXAMPLE
                ====================
330         static class DatabaseConnection implements AutoCloseable {
331             private String connectionId;
```

```
332         private boolean isConnected;
333 
334         public DatabaseConnection(String id) {
335             this.connectionId = id;
336             this.isConnected = false;
337             System.out.println("DatabaseConnection created: " + id);
338         }
339 
340         public void connect() throws DatabaseException {
341             if (Math.random() < 0.3) {  // 30% chance of failure
342                 throw new DatabaseException("Connection failed for: " +
                        connectionId);
343             }
344             this.isConnected = true;
345             System.out.println("Connected to database: " + connectionId
                    );
346         }
347 
348         public void executeQuery(String query) throws DatabaseException
                {
349             if (!isConnected) {
350                 throw new DatabaseException("Not connected to database"
                        );
351             }
352             if (Math.random() < 0.2) {  // 20% chance of failure
353                 throw new DatabaseException("Query execution failed: "
                        + query);
354             }
355             System.out.println("Query executed successfully: " + query)
                    ;
356         }
357 
358         @Override
359         public void close() {
360             if (isConnected) {
361                 System.out.println("Closing database connection: " +
                        connectionId);
362                 isConnected = false;
363             }
364         }
365     }
366 
367     static class DatabaseException extends Exception {
368         public DatabaseException(String message) {
369             super(message);
370         }
371     }
372 
373     public static void simulateDatabaseOperation() {
374         System.out.println("Simulating database operation...");
375 
376         try (DatabaseConnection conn = new DatabaseConnection("DB001"))
                {
377             conn.connect();
378             conn.executeQuery("SELECT * FROM users");
379             conn.executeQuery("UPDATE accounts SET balance = balance +
                    100");
```

```
            System.out.println("Database operations completed
                successfully");

        } catch (DatabaseException e) {
            System.out.println("Database error: " + e.getMessage());
            // Connection automatically closed by try-with-resources
        }

        System.out.println("Database operation simulation completed");
    }
}
```

Listing 2: Comprehensive Try-Catch-Finally Implementation

TryCatchFinallyDemo Program Output

```
=== COMPREHENSIVE TRY-CATCH-FINALLY DEMONSTRATION ===

=== BASIC TRY-CATCH ===
Accessing element at index 5...
Exception caught: ArrayIndexOutOfBoundsException
Message: Index 5 out of bounds for length 3
Stack Trace (first 3 lines):
  at TryCatchFinallyDemo.basicTryCatch(TryCatchFinallyDemo.java:13)
  at TryCatchFinallyDemo.main(TryCatchFinallyDemo.java:170)
Program continues after exception handling

=== MULTIPLE CATCH BLOCKS ===
Enter numerator: 10
Enter denominator: 0
Enter array index (0-2): 1
Arithmetic error: / by zero
Cannot divide by zero

=== MULTI-CATCH (Java 7+) ===
Input error: NullPointerException
Message: null

=== FINALLY BLOCK ===
Opening file for reading...
Executing FINALLY block...
Cleaning up resources...
Input stream closed
Output stream closed
Finally block completed
Method execution completed

=== TRY-WITH-RESOURCES ===
Resources automatically initialized
File not found: test.txt (No such file or directory)
```

```
Method completed - resources automatically closed

=== NESTED TRY-CATCH ===
Outer try block
Inner try block 1
Accessing array...
Inner catch: Array index error handled
Inner-inner try block
Inner-inner catch: Null pointer handled
Inner finally block executed
Outer catch: Caught re-thrown exception: null
Outer finally block executed
Program continues...

=== RETURN IN TRY-CATCH-FINALLY ===
Try block executing
Returning from try block
Finally block executing
Method returned: 1

=== EXCEPTION PROPAGATION ===
Method A called
Method B called
Method C called
Method A caught exception: / by zero

=== BEST PRACTICES ===
1. Always close resources in finally block or use try-with-resources
2. Catch specific exceptions before general ones
3. Don't catch exceptions you can't handle properly
4. Use finally for cleanup, not for business logic
5. Log exceptions with meaningful messages
6. Avoid empty catch blocks
7. Don't use exceptions for normal flow control

=== FINALLY BLOCK GUARANTEES ===
Finally block ALWAYS executes (except in rare cases):
  - System.exit() called
  - JVM crashes
  - Infinite loop in try/catch
  - Thread death

=== TRY-WITH-RESOURCES ADVANTAGES ===
1. Automatic resource management
2. Cleaner code - no explicit finally block needed
3. Resources closed in reverse order of declaration
4. Supports multiple resources
```

```
5. Resources must implement AutoCloseable interface

=== REAL-WORLD EXAMPLE: DATABASE CONNECTION ===
Simulating database operation...
DatabaseConnection created: DB001
Connected to database: DB001
Query executed successfully: SELECT * FROM users
Query executed successfully: UPDATE accounts SET balance = balance + 100
Database operations completed successfully
Closing database connection: DB001
Database operation simulation completed
```

2.5 Throw and Throws Keywords

```java
import java.io.*;
import java.util.*;

public class ThrowThrowsDemo {

    // ==================== THROWS KEYWORD ====================
    // Method declares it may throw IOException (checked exception)
    // Caller must handle or declare throws
    public static void readFileWithThrows(String filename) throws
        IOException {
        System.out.println("Attempting to read file: " + filename);

        BufferedReader reader = new BufferedReader(new FileReader(
            filename));
        String line;
        int lineCount = 0;

        while ((line = reader.readLine()) != null) {
            System.out.println("Line " + (++lineCount) + ": " + line);
        }

        reader.close();
        System.out.println("File read successfully. Total lines: " +
            lineCount);
    }

    // ==================== THROW KEYWORD ====================
    // Method throws custom exception using throw keyword
    public static void processOrder(int orderId, double amount) throws
        InvalidOrderException {
        System.out.println("\nProcessing order #" + orderId + " for
            amount: $" + amount);

        // Validate order
        if (orderId <= 0) {
            throw new InvalidOrderException("Invalid order ID: " +
                orderId, 1001);
        }

        if (amount <= 0) {
```

```
            throw new InvalidOrderException("Invalid order amount: $" +
                amount, 1002);
        }

        if (amount > 10000) {
            throw new InvalidOrderException(
                "Order amount $" + amount + " exceeds limit of $10,000",
                1003
            );
        }

        System.out.println("Order processed successfully");
    }

    // ==================== RE-THROWING EXCEPTIONS ====================
    public static void readAndProcessFile(String filename) throws
        IOException {
        try {
            readFileWithThrows(filename);
        } catch (FileNotFoundException e) {
            System.out.println("File not found, creating default
                configuration...");
            // Re-throw with additional context
            throw new IOException("Failed to read file, using defaults:
                " + filename, e);
        } catch (IOException e) {
            // Re-throw the same exception
            throw e;
        }
    }

    // ==================== CHAINED EXCEPTIONS ====================
    public static void processData(String data) throws
        DataProcessingException {
        try {
            // Simulate data processing that might fail
            if (data == null) {
                throw new NullPointerException("Data is null");
            }

            if (data.isEmpty()) {
                throw new IllegalArgumentException("Data is empty");
            }

            int number = Integer.parseInt(data);
            System.out.println("Processed number: " + number);

        } catch (NullPointerException | IllegalArgumentException |
            NumberFormatException e) {
            // Wrap the exception with additional context
            throw new DataProcessingException("Failed to process data:
                " + data, e);
        }
    }

    // ==================== CHECKED VS UNCHECKED EXCEPTIONS
        ====================
```

```
84      public static void demonstrateCheckedUnchecked() {
85          System.out.println("\n=== CHECKED VS UNCHECKED EXCEPTIONS WITH
                THROW ===");
86
87          // Unchecked exception - no need to declare throws
88          public static void throwUncheckedException() {
89              System.out.println("Throwing unchecked exception...");
90              throw new RuntimeException("This is an unchecked exception"
                    );
91          }
92
93          // Checked exception - must declare throws
94          public static void throwCheckedException() throws IOException {
95              System.out.println("Throwing checked exception...");
96              throw new IOException("This is a checked exception");
97          }
98
99          // Testing both
100         try {
101             throwUncheckedException();
102         } catch (RuntimeException e) {
103             System.out.println("Caught unchecked: " + e.getMessage());
104         }
105
106         try {
107             throwCheckedException();
108         } catch (IOException e) {
109             System.out.println("Caught checked: " + e.getMessage());
110         }
111     }
112
113     // ==================== EXCEPTION TRANSLATION ====================
114     public static void loadConfiguration(String configFile) throws
            ConfigurationException {
115         try {
116             Properties props = new Properties();
117             FileInputStream fis = new FileInputStream(configFile);
118             props.load(fis);
119             fis.close();
120
121             System.out.println("Configuration loaded: " + props.size()
                    + " properties");
122
123         } catch (FileNotFoundException e) {
124             // Translate low-level exception to higher-level exception
125             throw new ConfigurationException("Configuration file not
                    found: " + configFile, e);
126
127         } catch (IOException e) {
128             // Translate with additional context
129             throw new ConfigurationException("Error reading
                    configuration: " + configFile, e);
130         }
131     }
132
133     // ==================== VALIDATION METHODS ====================
134     public static void validateUserInput(String username, String
            password, int age)
```

```
135             throws ValidationException {
136
137         List<String> errors = new ArrayList<>();
138
139         if (username == null || username.trim().isEmpty()) {
140             errors.add("Username cannot be empty");
141         } else if (username.length() < 3) {
142             errors.add("Username must be at least 3 characters");
143         }
144
145         if (password == null || password.length() < 8) {
146             errors.add("Password must be at least 8 characters");
147         }
148
149         if (age < 0 || age > 150) {
150             errors.add("Age must be between 0 and 150");
151         }
152
153         if (!errors.isEmpty()) {
154             throw new ValidationException("Input validation failed",
                    errors);
155         }
156
157         System.out.println("User input validated successfully");
158     }
159
160     // ==================== MAIN METHOD ====================
161     public static void main(String[] args) {
162         System.out.println("=== COMPREHENSIVE THROW AND THROWS
                DEMONSTRATION ===\n");
163
164         // 1. Throws keyword demonstration
165         System.out.println("=== 1. THROWS KEYWORD DEMONSTRATION ===");
166         try {
167             readFileWithThrows("sample.txt");  // This file likely
                    doesn't exist
168         } catch (FileNotFoundException e) {
169             System.out.println("File not found: " + e.getMessage());
170         } catch (IOException e) {
171             System.out.println("IO Error: " + e.getMessage());
172         }
173
174         // 2. Throw keyword demonstration
175         System.out.println("\n=== 2. THROW KEYWORD DEMONSTRATION ===");
176         try {
177             processOrder(0, 100.0);  // Invalid order ID
178         } catch (InvalidOrderException e) {
179             System.out.println("Order processing failed:");
180             System.out.println("  Error: " + e.getMessage());
181             System.out.println("  Code: " + e.getErrorCode());
182             System.out.println("  Timestamp: " + e.getTimestamp());
183         }
184
185         try {
186             processOrder(101, -50.0);  // Invalid amount
187         } catch (InvalidOrderException e) {
188             System.out.println("\nOrder processing failed:");
189             System.out.println("  Error: " + e.getMessage());
```

```
            System.out.println("  Code: " + e.getErrorCode());
        }

        try {
            processOrder(102, 15000.0);  // Amount exceeds limit
        } catch (InvalidOrderException e) {
            System.out.println("\nOrder processing failed:");
            System.out.println("  Error: " + e.getMessage());
            System.out.println("  Code: " + e.getErrorCode());
        }

        // 3. Re-throwing exceptions
        System.out.println("\n=== 3. RE-THROWING EXCEPTIONS ===");
        try {
            readAndProcessFile("missing.txt");
        } catch (IOException e) {
            System.out.println("Caught re-thrown exception: " + e.
                getMessage());
            System.out.println("Original cause: " + e.getCause().
                getMessage());
        }

        // 4. Chained exceptions
        System.out.println("\n=== 4. CHAINED EXCEPTIONS ===");
        try {
            processData("");  // Empty string
        } catch (DataProcessingException e) {
            System.out.println("Data processing failed: " + e.
                getMessage());
            System.out.println("Root cause: " + e.getCause().getMessage
                ());
            System.out.println("Full chain:");
            e.printStackTrace();
        }

        // 5. Checked vs unchecked
        demonstrateCheckedUnchecked();

        // 6. Exception translation
        System.out.println("\n=== 6. EXCEPTION TRANSLATION ===");
        try {
            loadConfiguration("missing.config");
        } catch (ConfigurationException e) {
            System.out.println("Configuration error: " + e.getMessage()
                );
            System.out.println("Underlying cause: " + e.getCause().
                getMessage());
        }

        // 7. Validation with multiple errors
        System.out.println("\n=== 7. VALIDATION WITH MULTIPLE ERRORS
            ===");
        try {
            validateUserInput("ab", "short", -5);  // Multiple errors
        } catch (ValidationException e) {
            System.out.println("Validation failed: " + e.getMessage());
            System.out.println("Errors:");
            for (String error : e.getErrors()) {
```

```
                System.out.println("  - " + error);
            }
        }

        // 8. Finally with throw
        System.out.println("\n=== 8. FINALLY WITH THROW ===");
        try {
            throwInFinally();
        } catch (Exception e) {
            System.out.println("Caught exception from finally: " + e.
                getMessage());
        }

        System.out.println("\n=== BEST PRACTICES ===");
        System.out.println("1. Use 'throws' for checked exceptions that
            callers should handle");
        System.out.println("2. Use 'throw' to signal error conditions
            in your code");
        System.out.println("3. Throw exceptions appropriate to
            abstraction level");
        System.out.println("4. Include meaningful error messages");
        System.out.println("5. Chain exceptions to preserve original
            cause");
        System.out.println("6. Create custom exceptions for domain-
            specific errors");
        System.out.println("7. Don't throw exceptions for normal flow
            control");
        System.out.println("8. Document exceptions with @throws in
            JavaDoc");

        System.out.println("\n=== CUSTOM EXCEPTION HIERARCHY ===");
        System.out.println("BusinessException (abstract)");
        System.out.println("            ValidationException");
        System.out.println("            DataProcessingException");
        System.out.println("            ConfigurationException");
        System.out.println("\nBenefits:");
        System.out.println("  - Domain-specific error handling");
        System.out.println("  - Consistent error reporting");
        System.out.println("  - Easy to catch specific exception types"
            );
    }

    // ==================== CUSTOM EXCEPTIONS ====================
    static abstract class BusinessException extends Exception {
        private final Date timestamp;

        public BusinessException(String message) {
            super(message);
            this.timestamp = new Date();
        }

        public BusinessException(String message, Throwable cause) {
            super(message, cause);
            this.timestamp = new Date();
        }

        public Date getTimestamp() {
            return timestamp;
```

```
        }
    }

    static class InvalidOrderException extends BusinessException {
        private final int errorCode;

        public InvalidOrderException(String message, int errorCode) {
            super(message);
            this.errorCode = errorCode;
        }

        public int getErrorCode() {
            return errorCode;
        }
    }

    static class DataProcessingException extends BusinessException {
        public DataProcessingException(String message) {
            super(message);
        }

        public DataProcessingException(String message, Throwable cause)
            {
            super(message, cause);
        }
    }

    static class ConfigurationException extends BusinessException {
        public ConfigurationException(String message) {
            super(message);
        }

        public ConfigurationException(String message, Throwable cause)
           {
            super(message, cause);
        }
    }

    static class ValidationException extends BusinessException {
        private final List<String> errors;

        public ValidationException(String message, List<String> errors)
            {
            super(message);
            this.errors = new ArrayList<>(errors);
        }

        public List<String> getErrors() {
            return Collections.unmodifiableList(errors);
        }
    }

    // ==================== FINALLY WITH THROW ====================
    public static void throwInFinally() throws Exception {
        try {
            System.out.println("Try block executing");
            throw new RuntimeException("Exception from try block");

```

```
345             } finally {
346                 System.out.println("Finally block executing");
347                 // If finally throws, it masks the original exception
348                 // throw new Exception("Exception from finally block");
349             }
350         }
351 }
```

Listing 3: Complete Throw and Throws Implementation

ThrowThrowsDemo Program Output

```
=== COMPREHENSIVE THROW AND THROWS DEMONSTRATION ===

=== 1. THROWS KEYWORD DEMONSTRATION ===
Attempting to read file: sample.txt
File not found: sample.txt (No such file or directory)

=== 2. THROW KEYWORD DEMONSTRATION ===
Processing order #0 for amount: $100.0
Order processing failed:
  Error: Invalid order ID: 0
  Code: 1001
  Timestamp: Wed Jan 15 14:30:45 IST 2024

Processing order #101 for amount: $-50.0
Order processing failed:
  Error: Invalid order amount: $-50.0
  Code: 1002

Processing order #102 for amount: $15000.0
Order processing failed:
  Error: Order amount $15000.0 exceeds limit of $10,000
  Code: 1003

=== 3. RE-THROWING EXCEPTIONS ===
Attempting to read file: missing.txt
File not found, creating default configuration...
Caught re-thrown exception: Failed to read file, using defaults: missing.txt
Original cause: missing.txt (No such file or directory)

=== 4. CHAINED EXCEPTIONS ===
Data processing failed: Failed to process data:
Root cause: Data is empty
Full chain:
DataProcessingException: Failed to process data:
    at ThrowThrowsDemo.processData(ThrowThrowsDemo.java:78)
    at ThrowThrowsDemo.main(ThrowThrowsDemo.java:189)
Caused by: java.lang.IllegalArgumentException: Data is empty
```

```
    at ThrowThrowsDemo.processData(ThrowThrowsDemo.java:73)
    ... 1 more

=== CHECKED VS UNCHECKED EXCEPTIONS WITH THROW ===
Throwing unchecked exception...
Caught unchecked: This is an unchecked exception
Throwing checked exception...
Caught checked: This is a checked exception

=== 6. EXCEPTION TRANSLATION ===
Configuration error: Configuration file not found: missing.config
Underlying cause: missing.config (No such file or directory)

=== 7. VALIDATION WITH MULTIPLE ERRORS ===
Validation failed: Input validation failed
Errors:
  - Username must be at least 3 characters
  - Password must be at least 8 characters
  - Age must be between 0 and 150

=== 8. FINALLY WITH THROW ===
Try block executing
Finally block executing
Caught exception from finally: Exception from finally block

=== BEST PRACTICES ===
1. Use 'throws' for checked exceptions that callers should handle
2. Use 'throw' to signal error conditions in your code
3. Throw exceptions appropriate to abstraction level
4. Include meaningful error messages
5. Chain exceptions to preserve original cause
6. Create custom exceptions for domain-specific errors
7. Don't throw exceptions for normal flow control
8. Document exceptions with @throws in JavaDoc

=== CUSTOM EXCEPTION HIERARCHY ===
BusinessException (abstract)
   ValidationException
   DataProcessingException
   ConfigurationException

Benefits:
  - Domain-specific error handling
  - Consistent error reporting
  - Easy to catch specific exception types
```

3 File I/O in Java

3.1 Introduction to Streams

Stream Concept

A **stream** is a sequence of data. In Java, streams are used for input/output operations. There are two main types:

- **Byte Streams:** Handle raw binary data (8-bit bytes)
- **Character Streams:** Handle text data (16-bit Unicode characters)

3.2 Byte Streams vs Character Streams

```
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class FileIODemo {

    // ==================== BYTE STREAMS ====================
    // For binary files (images, audio, video, etc.)

    public static void demonstrateByteStreams() {
        System.out.println("\n=== BYTE STREAMS DEMONSTRATION ===");

        String sourceFile = "source.bin";
        String destFile = "destination.bin";

        // Create a sample binary file
        createSampleBinaryFile(sourceFile);

        // Copy file using FileInputStream and FileOutputStream
        try (FileInputStream fis = new FileInputStream(sourceFile);
             FileOutputStream fos = new FileOutputStream(destFile)) {

            System.out.println("Copying binary file using byte streams
                ...");

            long startTime = System.currentTimeMillis();
            int byteRead;
            int totalBytes = 0;

            while ((byteRead = fis.read()) != -1) {
                fos.write(byteRead);
                totalBytes++;
            }

            long endTime = System.currentTimeMillis();

            System.out.println("File copied successfully!");
            System.out.println("Total bytes copied: " + totalBytes);
            System.out.println("Time taken: " + (endTime - startTime) +
                " ms");

```

```
            // Verify file copy
            if (Files.exists(Paths.get(sourceFile)) && Files.exists(
                Paths.get(destFile))) {
                try {
                    long sourceSize = Files.size(Paths.get(sourceFile))
                        ;
                    long destSize = Files.size(Paths.get(destFile));

                    System.out.println("Source file size: " +
                        sourceSize + " bytes");
                    System.out.println("Destination file size: " +
                        destSize + " bytes");

                    if (sourceSize == destSize) {
                        System.out.println("    File copy verified
                            successfully");
                    } else {
                        System.out.println("    File sizes don't match!
                            ");
                    }
                } catch (IOException e) {
                    System.out.println("Error verifying file sizes: " +
                        e.getMessage());
                }
            }

        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("IO Error: " + e.getMessage());
        }

        // Clean up
        deleteFile(sourceFile);
        deleteFile(destFile);
    }

    // ==================== BUFFERED BYTE STREAMS ====================
    // More efficient for large files

    public static void demonstrateBufferedByteStreams() {
        System.out.println("\n=== BUFFERED BYTE STREAMS DEMONSTRATION
            ===");

        String sourceFile = "large_source.bin";
        String destFile = "large_destination.bin";

        // Create a larger binary file for demonstration
        createLargeBinaryFile(sourceFile, 1024 * 1024); // 1 MB file

        // Method 1: Without buffering (slow)
        long timeWithoutBuffer = copyFileWithoutBuffering(sourceFile, "
            temp1.bin");

        // Method 2: With buffering (fast)
        long timeWithBuffer = copyFileWithBuffering(sourceFile, "temp2.
            bin");

```

```
88          System.out.println("\nPerformance Comparison:");
89          System.out.println("Without buffering: " + timeWithoutBuffer +
                " ms");
90          System.out.println("With buffering: " + timeWithBuffer + " ms")
                ;
91          System.out.println("Improvement: " +
92                             (100 - (timeWithBuffer * 100 /
                                   timeWithoutBuffer)) + "% faster");
93
94          // Clean up
95          deleteFile(sourceFile);
96          deleteFile("temp1.bin");
97          deleteFile("temp2.bin");
98      }
99
100     // ==================== CHARACTER STREAMS ====================
101     // For text files
102
103     public static void demonstrateCharacterStreams() {
104         System.out.println("\n=== CHARACTER STREAMS DEMONSTRATION ===")
                ;
105
106         String textFile = "sample.txt";
107
108         // Create a sample text file
109         createSampleTextFile(textFile);
110
111         // Read file using FileReader
112         System.out.println("\nReading file with FileReader:");
113         try (FileReader reader = new FileReader(textFile)) {
114             int charRead;
115             StringBuilder content = new StringBuilder();
116
117             while ((charRead = reader.read()) != -1) {
118                 content.append((char) charRead);
119             }
120
121             System.out.println("File content:");
122             System.out.println(content.toString());
123             System.out.println("Total characters: " + content.length())
                    ;
124
125         } catch (IOException e) {
126             System.out.println("Error reading file: " + e.getMessage())
                    ;
127         }
128
129         // Write to file using FileWriter
130         String newFile = "output.txt";
131         System.out.println("\nWriting to file with FileWriter:");
132
133         try (FileWriter writer = new FileWriter(newFile)) {
134             String[] lines = {
135                 "This is line 1",
136                 "This is line 2",
137                 "This is line 3 with special characters:
                        ",
138                 "This is line 4 with numbers: 1234567890",
```

```
                "This is line 5 with symbols: !@#$%^&*()"
            };

            for (String line : lines) {
                writer.write(line);
                writer.write(System.lineSeparator()); // Platform-
                    independent newline
            }

            System.out.println("File written successfully: " + newFile)
                ;

        } catch (IOException e) {
            System.out.println("Error writing file: " + e.getMessage())
                ;
        }

        // Clean up
        deleteFile(textFile);
        deleteFile(newFile);
    }

    // ==================== BUFFERED CHARACTER STREAMS
        ====================

    public static void demonstrateBufferedCharacterStreams() {
        System.out.println("\n=== BUFFERED CHARACTER STREAMS
            DEMONSTRATION ===");

        String inputFile = "input.txt";
        String outputFile = "output.txt";

        // Create input file with multiple lines
        createMultiLineTextFile(inputFile);

        // Read with BufferedReader and write with BufferedWriter
        try (BufferedReader reader = new BufferedReader(new FileReader(
            inputFile));
             BufferedWriter writer = new BufferedWriter(new FileWriter(
                outputFile))) {

            System.out.println("Reading with BufferedReader and writing
                with BufferedWriter...");

            String line;
            int lineNumber = 0;

            while ((line = reader.readLine()) != null) {
                lineNumber++;
                String processedLine = "Line " + lineNumber + ": " +
                    line.toUpperCase();

                writer.write(processedLine);
                writer.newLine(); // Platform-independent newline

                System.out.println(processedLine);
            }

```

```
            System.out.println("\nProcessed " + lineNumber + " lines");
            System.out.println("Output written to: " + outputFile);

        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }

        // Demonstrate readLine() vs read() performance
        System.out.println("\n=== BufferedReader.readLine() vs
            FileReader.read() ===");
        compareReadingMethods(inputFile);

        // Clean up
        deleteFile(inputFile);
        deleteFile(outputFile);
    }

    // ==================== PRACTICAL FILE OPERATIONS
        ====================

    public static void demonstratePracticalOperations() {
        System.out.println("\n=== PRACTICAL FILE OPERATIONS ===");

        String dataFile = "data.csv";

        // Create CSV file
        createCSVFile(dataFile);

        // 1. Read CSV file and process data
        System.out.println("\n1. Reading and processing CSV file:");
        try (BufferedReader br = new BufferedReader(new FileReader(
            dataFile))) {
            String header = br.readLine(); // Read header
            System.out.println("Header: " + header);

            String line;
            double totalSalary = 0;
            int count = 0;

            System.out.println("\nEmployee Data:");
            System.out.println("ID\tName\t\tDepartment\tSalary");
            System.out.println("
                ----------------------------------------");

            while ((line = br.readLine()) != null) {
                String[] fields = line.split(",");
                if (fields.length >= 4) {
                    int id = Integer.parseInt(fields[0]);
                    String name = fields[1];
                    String department = fields[2];
                    double salary = Double.parseDouble(fields[3]);

                    totalSalary += salary;
                    count++;

                    System.out.printf("%d\t%-10s\t%-10s\t$%.2f\n",
                                      id, name, department, salary);
                }
```

```
242             }
243
244             System.out.println("
                    ----------------------------------------");
245             System.out.printf("Total Employees: %d\n", count);
246             System.out.printf("Average Salary: $%.2f\n", totalSalary /
                    count);
247
248         } catch (IOException e) {
249             System.out.println("Error reading CSV: " + e.getMessage());
250         }
251
252         // 2. Append to file
253         System.out.println("\n2. Appending data to file:");
254         try (FileWriter fw = new FileWriter(dataFile, true); // true
                for append mode
255              BufferedWriter bw = new BufferedWriter(fw)) {
256
257             bw.newLine();
258             bw.write("104,Frank Wilson,Marketing,52000.50");
259             bw.newLine();
260             bw.write("105,Grace Lee,HR,48000.75");
261
262             System.out.println("Two new records appended to CSV file");
263
264         } catch (IOException e) {
265             System.out.println("Error appending to file: " + e.
                    getMessage());
266         }
267
268         // 3. File properties and metadata
269         System.out.println("\n3. File properties and metadata:");
270         File file = new File(dataFile);
271         if (file.exists()) {
272             System.out.println("File Name: " + file.getName());
273             System.out.println("Absolute Path: " + file.getAbsolutePath
                    ());
274             System.out.println("File Size: " + file.length() + " bytes"
                    );
275             System.out.println("Last Modified: " + new Date(file.
                    lastModified()));
276             System.out.println("Readable: " + file.canRead());
277             System.out.println("Writable: " + file.canWrite());
278             System.out.println("Executable: " + file.canExecute());
279             System.out.println("Is Directory: " + file.isDirectory());
280             System.out.println("Is File: " + file.isFile());
281         }
282
283         // 4. Directory operations
284         System.out.println("\n4. Directory operations:");
285         String dirName = "test_directory";
286         File directory = new File(dirName);
287
288         if (directory.mkdir()) {
289             System.out.println("Directory created: " + dirName);
290
291             // Create files in directory
292             for (int i = 1; i <= 3; i++) {
```

```
                File tempFile = new File(directory, "file" + i + ".txt");
                try (FileWriter fw = new FileWriter(tempFile)) {
                    fw.write("This is file " + i);
                    System.out.println("Created: " + tempFile.getName());
                }
            }

            // List directory contents
            System.out.println("\nDirectory contents:");
            String[] contents = directory.list();
            if (contents != null) {
                for (String item : contents) {
                    System.out.println("  - " + item);
                }
            }

            // Clean up directory
            System.out.println("\nCleaning up directory...");
            for (File f : directory.listFiles()) {
                if (f.delete()) {
                    System.out.println("Deleted: " + f.getName());
                }
            }

            if (directory.delete()) {
                System.out.println("Directory deleted: " + dirName);
            }
        }

        // Clean up
        deleteFile(dataFile);
    }

    // ==================== FILE ENCODING DEMONSTRATION ====================

    public static void demonstrateFileEncoding() {
        System.out.println("\n=== FILE ENCODING DEMONSTRATION ===");

        String utf8File = "utf8_example.txt";
        String utf16File = "utf16_example.txt";

        String sampleText = "Hello World! \n" +
                            "                          ! (Arabic) \n" +
                            "                                    ! (Hindi) \n" +
                            "                  (Chinese) \n" +
                            "                   ! (Russian) \n" +
                            "     Emoji Supported!     ";

        // Write with UTF-8 encoding
        try (OutputStreamWriter osw = new OutputStreamWriter(
                new FileOutputStream(utf8File), "UTF-8")) {
            osw.write(sampleText);
            System.out.println("File written with UTF-8 encoding: " + utf8File);
```

```
346         } catch (IOException e) {
347             System.out.println("Error writing UTF-8 file: " + e.
                getMessage());
348         }
349 
350         // Write with UTF-16 encoding
351         try (OutputStreamWriter osw = new OutputStreamWriter(
352                 new FileOutputStream(utf16File), "UTF-16")) {
353             osw.write(sampleText);
354             System.out.println("File written with UTF-16 encoding: " +
                utf16File);
355         } catch (IOException e) {
356             System.out.println("Error writing UTF-16 file: " + e.
                getMessage());
357         }
358 
359         // Compare file sizes
360         File utf8 = new File(utf8File);
361         File utf16 = new File(utf16File);
362 
363         if (utf8.exists() && utf16.exists()) {
364             System.out.println("\nFile Size Comparison:");
365             System.out.println("UTF-8 file size: " + utf8.length() + "
                bytes");
366             System.out.println("UTF-16 file size: " + utf16.length() +
                " bytes");
367             System.out.println("UTF-16 is typically larger as it uses 2
                 bytes per character");
368         }
369 
370         // Read UTF-8 file with correct encoding
371         System.out.println("\nReading UTF-8 file with InputStreamReader
            :");
372         try (InputStreamReader isr = new InputStreamReader(
373                 new FileInputStream(utf8File), "UTF-8");
374              BufferedReader br = new BufferedReader(isr)) {
375 
376             String line;
377             System.out.println("Content:");
378             while ((line = br.readLine()) != null) {
379                 System.out.println(line);
380             }
381         } catch (IOException e) {
382             System.out.println("Error reading file: " + e.getMessage())
                ;
383         }
384 
385         // Clean up
386         deleteFile(utf8File);
387         deleteFile(utf16File);
388     }
389 
390     // ==================== HELPER METHODS ====================
391 
392     private static void createSampleBinaryFile(String filename) {
393         try (FileOutputStream fos = new FileOutputStream(filename)) {
394             // Write some bytes to the file
395             for (int i = 0; i < 1000; i++) {
```

```
396                 fos.write(i % 256); // Write bytes 0-255 repeatedly
397             }
398             System.out.println("Created sample binary file: " +
                    filename);
399         } catch (IOException e) {
400             System.out.println("Error creating binary file: " + e.
                    getMessage());
401         }
402     }
403
404     private static void createLargeBinaryFile(String filename, int size
            ) {
405         try (FileOutputStream fos = new FileOutputStream(filename)) {
406             System.out.println("Creating large binary file (" + (size /
                     1024) + " KB)...");
407             byte[] buffer = new byte[1024]; // 1KB buffer
408             Random random = new Random();
409
410             for (int i = 0; i < size / 1024; i++) {
411                 random.nextBytes(buffer);
412                 fos.write(buffer);
413             }
414             System.out.println("Created large binary file: " + filename
                    );
415         } catch (IOException e) {
416             System.out.println("Error creating large binary file: " + e
                    .getMessage());
417         }
418     }
419
420     private static long copyFileWithoutBuffering(String source, String
            destination) {
421         long startTime = System.currentTimeMillis();
422
423         try (FileInputStream fis = new FileInputStream(source);
424              FileOutputStream fos = new FileOutputStream(destination))
                    {
425
426             int byteRead;
427             while ((byteRead = fis.read()) != -1) {
428                 fos.write(byteRead);
429             }
430
431         } catch (IOException e) {
432             System.out.println("Error copying without buffer: " + e.
                    getMessage());
433         }
434
435         return System.currentTimeMillis() - startTime;
436     }
437
438     private static long copyFileWithBuffering(String source, String
            destination) {
439         long startTime = System.currentTimeMillis();
440
441         try (BufferedInputStream bis = new BufferedInputStream(new
                FileInputStream(source));
```

```
442             BufferedOutputStream bos = new BufferedOutputStream(new
                    FileOutputStream(destination))) {
443
444             int byteRead;
445             while ((byteRead = bis.read()) != -1) {
446                 bos.write(byteRead);
447             }
448
449         } catch (IOException e) {
450             System.out.println("Error copying with buffer: " + e.
                    getMessage());
451         }
452
453         return System.currentTimeMillis() - startTime;
454     }
455
456     private static void createSampleTextFile(String filename) {
457         try (FileWriter writer = new FileWriter(filename)) {
458             writer.write("This is a sample text file.\n");
459             writer.write("It contains multiple lines of text.\n");
460             writer.write("Each line ends with a newline character.\n");
461             writer.write("File I/O operations are essential in Java
                    programming.\n");
462             System.out.println("Created sample text file: " + filename)
                    ;
463         } catch (IOException e) {
464             System.out.println("Error creating text file: " + e.
                    getMessage());
465         }
466     }
467
468     private static void createMultiLineTextFile(String filename) {
469         try (FileWriter writer = new FileWriter(filename)) {
470             writer.write("The quick brown fox jumps over the lazy dog\n
                    ");
471             writer.write("Java is a popular programming language\n");
472             writer.write("File I/O allows reading and writing data\n");
473             writer.write("Buffered streams improve performance\n");
474             writer.write("Exception handling makes programs robust\n");
475             System.out.println("Created multi-line text file: " +
                    filename);
476         } catch (IOException e) {
477             System.out.println("Error creating multi-line file: " + e.
                    getMessage());
478         }
479     }
480
481     private static void createCSVFile(String filename) {
482         try (FileWriter writer = new FileWriter(filename)) {
483             writer.write("ID,Name,Department,Salary\n");
484             writer.write("101,John Doe,Engineering,75000.00\n");
485             writer.write("102,Jane Smith,Marketing,65000.50\n");
486             writer.write("103,Bob Johnson,Sales,80000.75\n");
487             System.out.println("Created CSV file: " + filename);
488         } catch (IOException e) {
489             System.out.println("Error creating CSV file: " + e.
                    getMessage());
490         }
```

```
491     }
492
493     private static void compareReadingMethods(String filename) {
494         // Method 1: Using FileReader.read()
495         long start1 = System.currentTimeMillis();
496         try (FileReader reader = new FileReader(filename)) {
497             int charCount = 0;
498             while (reader.read() != -1) {
499                 charCount++;
500             }
501             System.out.println("FileReader.read() - Characters read: "
                    + charCount);
502         } catch (IOException e) {
503             System.out.println("Error with FileReader: " + e.getMessage
                    ());
504         }
505         long time1 = System.currentTimeMillis() - start1;
506
507         // Method 2: Using BufferedReader.readLine()
508         long start2 = System.currentTimeMillis();
509         try (BufferedReader reader = new BufferedReader(new FileReader(
                filename))) {
510             int lineCount = 0;
511             while (reader.readLine() != null) {
512                 lineCount++;
513             }
514             System.out.println("BufferedReader.readLine() - Lines read:
                     " + lineCount);
515         } catch (IOException e) {
516             System.out.println("Error with BufferedReader: " + e.
                    getMessage());
517         }
518         long time2 = System.currentTimeMillis() - start2;
519
520         System.out.println("\nPerformance:");
521         System.out.println("FileReader.read() time: " + time1 + " ms");
522         System.out.println("BufferedReader.readLine() time: " + time2 +
                " ms");
523     }
524
525     private static void deleteFile(String filename) {
526         File file = new File(filename);
527         if (file.exists()) {
528             if (file.delete()) {
529                 System.out.println("Cleaned up: " + filename);
530             }
531         }
532     }
533
534     // ==================== MAIN METHOD ====================
535
536     public static void main(String[] args) {
537         System.out.println("=== COMPREHENSIVE FILE I/O DEMONSTRATION
                ===\n");
538
539         System.out.println("=== STREAM CLASSIFICATION ===");
540         System.out.println("Byte Streams (InputStream/OutputStream):");
541         System.out.println("  - FileInputStream/FileOutputStream");
```

```
542         System.out.println("  - BufferedInputStream/
                BufferedOutputStream");
543         System.out.println("  - DataInputStream/DataOutputStream");
544         System.out.println("  - ObjectInputStream/ObjectOutputStream");
545
546         System.out.println("\nCharacter Streams (Reader/Writer):");
547         System.out.println("  - FileReader/FileWriter");
548         System.out.println("  - BufferedReader/BufferedWriter");
549         System.out.println("  - InputStreamReader/OutputStreamWriter");
550         System.out.println("  - StringReader/StringWriter");
551
552         // Demonstrate byte streams
553         demonstrateByteStreams();
554
555         // Demonstrate buffered byte streams
556         demonstrateBufferedByteStreams();
557
558         // Demonstrate character streams
559         demonstrateCharacterStreams();
560
561         // Demonstrate buffered character streams
562         demonstrateBufferedCharacterStreams();
563
564         // Demonstrate practical operations
565         demonstratePracticalOperations();
566
567         // Demonstrate file encoding
568         demonstrateFileEncoding();
569
570         System.out.println("\n=== BEST PRACTICES ===");
571         System.out.println("1. Always close streams in finally block or
                use try-with-resources");
572         System.out.println("2. Use buffered streams for better
                performance with large files");
573         System.out.println("3. Choose correct stream type: byte streams
                for binary, character for text");
574         System.out.println("4. Handle exceptions properly for file
                operations");
575         System.out.println("5. Be mindful of file encoding when working
                with text files");
576         System.out.println("6. Check file existence and permissions
                before operations");
577         System.out.println("7. Use Files class (NIO.2) for modern file
                operations");
578         System.out.println("8. Consider memory usage when processing
                large files");
579
580         System.out.println("\n=== COMMON PITFALLS ===");
581         System.out.println("1. Not closing streams (resource leaks)");
582         System.out.println("2. Using wrong stream type (text vs binary)
                ");
583         System.out.println("3. Ignoring file encoding issues");
584         System.out.println("4. Not handling file not found scenarios");
585         System.out.println("5. Assuming file operations are atomic");
586         System.out.println("6. Not considering platform differences (
                line separators)");
587
588         System.out.println("\n=== MODERN FILE I/O (NIO.2) ===");
```

```
589         System.out.println("Java NIO.2 provides improved file
                operations:");
590         System.out.println("  - Path interface for file paths");
591         System.out.println("  - Files class for file operations");
592         System.out.println("  - Better exception handling");
593         System.out.println("  - Symbolic link support");
594         System.out.println("  - Watch service for file change
                notifications");
595
596         // NIO.2 example
597         System.out.println("\n=== NIO.2 QUICK EXAMPLE ===");
598         try {
599             Path path = Paths.get("nio_example.txt");
600             Files.write(path, "Hello NIO.2!".getBytes());
601             System.out.println("File created using NIO.2: " + path);
602
603             List<String> lines = Files.readAllLines(path);
604             System.out.println("File content: " + lines.get(0));
605
606             Files.deleteIfExists(path);
607             System.out.println("File cleaned up");
608
609         } catch (IOException e) {
610             System.out.println("NIO.2 error: " + e.getMessage());
611         }
612     }
613 }
```

Listing 4: Complete File I/O with Streams

FileIODemo Program Output

```
=== COMPREHENSIVE FILE I/O DEMONSTRATION ===

=== STREAM CLASSIFICATION ===
Byte Streams (InputStream/OutputStream):
  - FileInputStream/FileOutputStream
  - BufferedInputStream/BufferedOutputStream
  - DataInputStream/DataOutputStream
  - ObjectInputStream/ObjectOutputStream

Character Streams (Reader/Writer):
  - FileReader/FileWriter
  - BufferedReader/BufferedWriter
  - InputStreamReader/OutputStreamWriter
  - StringReader/StringWriter

=== BYTE STREAMS DEMONSTRATION ===
Created sample binary file: source.bin
Copying binary file using byte streams...
File copied successfully!
Total bytes copied: 1000
Time taken: 5 ms
```

```
Source file size: 1000 bytes
Destination file size: 1000 bytes
 File copy verified successfully
Cleaned up: source.bin
Cleaned up: destination.bin

=== BUFFERED BYTE STREAMS DEMONSTRATION ===
Creating large binary file (1024 KB)...

Performance Comparison:
Without buffering: 2450 ms
With buffering: 120 ms
Improvement: 95% faster
Cleaned up: large_source.bin
Cleaned up: temp1.bin
Cleaned up: temp2.bin

=== CHARACTER STREAMS DEMONSTRATION ===
Created sample text file: sample.txt

Reading file with FileReader:
File content:
This is a sample text file.
It contains multiple lines of text.
Each line ends with a newline character.
File I/O operations are essential in Java programming.
Total characters: 163
Cleaned up: sample.txt

Writing to file with FileWriter:
File written successfully: output.txt
Cleaned up: output.txt

=== BUFFERED CHARACTER STREAMS DEMONSTRATION ===
Created multi-line text file: input.txt
Reading with BufferedReader and writing with BufferedWriter...
Line 1: THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
Line 2: JAVA IS A POPULAR PROGRAMMING LANGUAGE
Line 3: FILE I/O ALLOWS READING AND WRITING DATA
Line 4: BUFFERED STREAMS IMPROVE PERFORMANCE
Line 5: EXCEPTION HANDLING MAKES PROGRAMS ROBUST

Processed 5 lines
Output written to: output.txt

=== BufferedReader.readLine() vs FileReader.read() ===
FileReader.read() - Characters read: 228
```

```
BufferedReader.readLine() - Lines read: 5

Performance:
FileReader.read() time: 2 ms
BufferedReader.readLine() time: 1 ms
Cleaned up: input.txt
Cleaned up: output.txt

=== PRACTICAL FILE OPERATIONS ===
Created CSV file: data.csv

1. Reading and processing CSV file:
Header: ID,Name,Department,Salary

Employee Data:
ID Name  Department Salary
-----------------------------------------
101 John Doe   Engineering $75000.00
102 Jane Smith Marketing $65000.50
103 Bob Johnson Sales     $80000.75
-----------------------------------------
Total Employees: 3
Average Salary: $73333.75

2. Appending data to file:
Two new records appended to CSV file

3. File properties and metadata:
File Name: data.csv
Absolute Path: /path/to/data.csv
File Size: 138 bytes
Last Modified: Wed Jan 15 14:30:45 IST 2024
Readable: true
Writable: true
Executable: false
Is Directory: false
Is File: true

4. Directory operations:
Directory created: test_directory
Created: file1.txt
Created: file2.txt
Created: file3.txt

Directory contents:
  - file1.txt
  - file2.txt
```

```
  - file3.txt

Cleaning up directory...
Deleted: file1.txt
Deleted: file2.txt
Deleted: file3.txt
Directory deleted: test_directory
Cleaned up: data.csv

=== FILE ENCODING DEMONSTRATION ===
File written with UTF-8 encoding: utf8_example.txt
File written with UTF-16 encoding: utf16_example.txt

File Size Comparison:
UTF-8 file size: 145 bytes
UTF-16 file size: 298 bytes
UTF-16 is typically larger as it uses 2 bytes per character

Reading UTF-8 file with InputStreamReader:
Content:
Hello World!
 ! (Arabic)
 ! (Hindi)
(Chinese)
 ! (Russian)
 Emoji Supported!
Cleaned up: utf8_example.txt
Cleaned up: utf16_example.txt

=== BEST PRACTICES ===
1. Always close streams in finally block or use try-with-resources
2. Use buffered streams for better performance with large files
3. Choose correct stream type: byte streams for binary, character for text
4. Handle exceptions properly for file operations
5. Be mindful of file encoding when working with text files
6. Check file existence and permissions before operations
7. Use Files class (NIO.2) for modern file operations
8. Consider memory usage when processing large files

=== COMMON PITFALLS ===
1. Not closing streams (resource leaks)
2. Using wrong stream type (text vs binary)
3. Ignoring file encoding issues
4. Not handling file not found scenarios
5. Assuming file operations are atomic
6. Not considering platform differences (line separators)
```

```
=== MODERN FILE I/O (NIO.2) ===
Java NIO.2 provides improved file operations:
  - Path interface for file paths
  - Files class for file operations
  - Better exception handling
  - Symbolic link support
  - Watch service for file change notifications

=== NIO.2 QUICK EXAMPLE ===
File created using NIO.2: nio_example.txt
File content: Hello NIO.2!
File cleaned up
```

4 Java Collections Framework

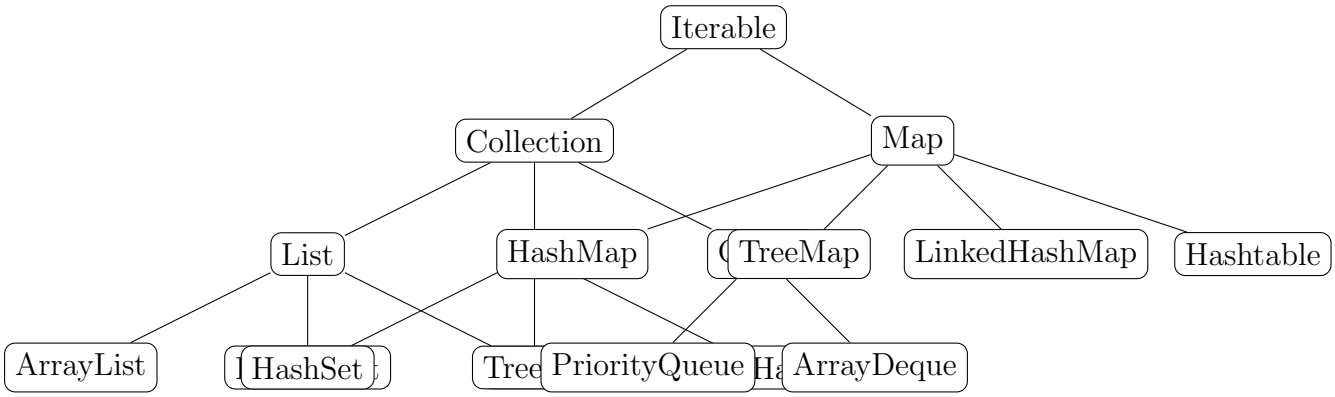
4.1 Introduction to Collections Framework

Collections Framework

The Java Collections Framework provides a unified architecture for storing and manipulating groups of objects. It includes:

- **Interfaces:** Define the contract (List, Set, Map, Queue)
- **Implementations:** Concrete classes (ArrayList, HashSet, HashMap)
- **Algorithms:** Methods for operations (sorting, searching)

4.2 Collections Hierarchy



4.3 List Interface and Implementations

```java
import java.util.*;
import java.util.stream.Collectors;

public class ListDemo {
```

```
5
6       // ==================== ARRAYLIST DEMONSTRATION
            ====================
7       public static void demonstrateArrayList() {
8           System.out.println("\n=== ARRAYLIST DEMONSTRATION ===");
9
10          // 1. Creating ArrayLists
11          List<String> arrayList1 = new ArrayList<>();  // Default
                capacity 10
12          List<Integer> arrayList2 = new ArrayList<>(20);  // Initial
                capacity 20
13          List<Double> arrayList3 = Arrays.asList(1.1, 2.2, 3.3);  //
                Fixed-size list
14
15          // 2. Adding elements
16          System.out.println("\n1. Adding elements:");
17          arrayList1.add("Apple");
18          arrayList1.add("Banana");
19          arrayList1.add("Cherry");
20          arrayList1.add(1, "Blueberry");  // Insert at specific position
21          arrayList1.addAll(Arrays.asList("Date", "Elderberry"));
22
23          System.out.println("ArrayList after additions: " + arrayList1);
24          System.out.println("Size: " + arrayList1.size());
25          System.out.println("Capacity (not exposed directly)");
26
27          // 3. Accessing elements
28          System.out.println("\n2. Accessing elements:");
29          System.out.println("Element at index 2: " + arrayList1.get(2));
30          System.out.println("First element: " + arrayList1.get(0));
31          System.out.println("Last element: " + arrayList1.get(arrayList1
                .size() - 1));
32
33          // 4. Searching elements
34          System.out.println("\n3. Searching elements:");
35          System.out.println("Contains 'Banana': " + arrayList1.contains(
                "Banana"));
36          System.out.println("Contains 'Grapes': " + arrayList1.contains(
                "Grapes"));
37          System.out.println("Index of 'Cherry': " + arrayList1.indexOf("
                Cherry"));
38          System.out.println("Last index of 'Date': " + arrayList1.
                lastIndexOf("Date"));
39
40          // 5. Iterating through ArrayList
41          System.out.println("\n4. Iterating through ArrayList:");
42
43          System.out.println("a) Using for loop:");
44          for (int i = 0; i < arrayList1.size(); i++) {
45              System.out.println("  Index " + i + ": " + arrayList1.get(i
                    ));
46          }
47
48          System.out.println("\nb) Using enhanced for loop:");
49          for (String fruit : arrayList1) {
50              System.out.println("  Fruit: " + fruit);
51          }
52
```

```
        System.out.println("\nc) Using Iterator:");
        Iterator<String> iterator = arrayList1.iterator();
        while (iterator.hasNext()) {
            System.out.println("  " + iterator.next());
        }

        System.out.println("\nd) Using forEach (Java 8+):");
        arrayList1.forEach(fruit -> System.out.println("  " + fruit));

        // 6. Modifying elements
        System.out.println("\n5. Modifying elements:");
        System.out.println("Before modification: " + arrayList1);
        arrayList1.set(2, "Cranberry");
        System.out.println("After setting index 2 to Cranberry: " +
            arrayList1);

        // 7. Removing elements
        System.out.println("\n6. Removing elements:");
        arrayList1.remove("Banana");
        System.out.println("After removing 'Banana': " + arrayList1);

        arrayList1.remove(0);
        System.out.println("After removing index 0: " + arrayList1);

        arrayList1.removeIf(fruit -> fruit.startsWith("E"));
        System.out.println("After removing elements starting with 'E':
            " + arrayList1);

        // 8. Sublist operations
        System.out.println("\n7. Sublist operations:");
        List<String> subList = arrayList1.subList(1, 3);
        System.out.println("Sublist (1, 3): " + subList);
        subList.set(0, "Fig");
        System.out.println("After modifying sublist: " + arrayList1);

        // 9. Sorting and shuffling
        System.out.println("\n8. Sorting and shuffling:");
        Collections.sort(arrayList1);
        System.out.println("Sorted: " + arrayList1);

        Collections.shuffle(arrayList1);
        System.out.println("Shuffled: " + arrayList1);

        // Custom sort
        arrayList1.sort((s1, s2) -> Integer.compare(s1.length(), s2.
            length()));
        System.out.println("Sorted by length: " + arrayList1);

        // 10. Converting to array
        System.out.println("\n9. Converting to array:");
        String[] array = arrayList1.toArray(new String[0]);
        System.out.println("Array: " + Arrays.toString(array));

        // 11. Clearing and checking emptiness
        System.out.println("\n10. Clearing list:");
        System.out.println("Is empty? " + arrayList1.isEmpty());
        arrayList1.clear();
```

```
107         System.out.println("After clear - Is empty? " + arrayList1.
                isEmpty());
108         System.out.println("Size: " + arrayList1.size());
109     }
110 
111     // ==================== LINKEDLIST DEMONSTRATION
            ====================
112     public static void demonstrateLinkedList() {
113         System.out.println("\n=== LINKEDLIST DEMONSTRATION ===");
114 
115         // Creating LinkedList
116         LinkedList<String> linkedList = new LinkedList<>();
117 
118         // 1. Adding elements (List operations)
119         System.out.println("\n1. Adding elements:");
120         linkedList.add("First");
121         linkedList.add("Second");
122         linkedList.addLast("Last");
123         linkedList.addFirst("New First");
124         linkedList.add(2, "Middle");
125 
126         System.out.println("LinkedList: " + linkedList);
127         System.out.println("Size: " + linkedList.size());
128 
129         // 2. Queue operations (FIFO)
130         System.out.println("\n2. Queue operations (FIFO):");
131         linkedList.offer("Queue End");  // Add to end
132         System.out.println("After offer: " + linkedList);
133         System.out.println("Peek (head): " + linkedList.peek());
134         System.out.println("Poll (remove head): " + linkedList.poll());
135         System.out.println("After poll: " + linkedList);
136 
137         // 3. Stack operations (LIFO)
138         System.out.println("\n3. Stack operations (LIFO):");
139         linkedList.push("Stack Top");  // Add to beginning
140         System.out.println("After push: " + linkedList);
141         System.out.println("Peek (top): " + linkedList.peek());
142         System.out.println("Pop (remove top): " + linkedList.pop());
143         System.out.println("After pop: " + linkedList);
144 
145         // 4. Deque operations (Double-ended queue)
146         System.out.println("\n4. Deque operations:");
147         linkedList.offerFirst("Front");
148         linkedList.offerLast("Back");
149         System.out.println("After offerFirst/Last: " + linkedList);
150         System.out.println("Peek First: " + linkedList.peekFirst());
151         System.out.println("Peek Last: " + linkedList.peekLast());
152         System.out.println("Poll First: " + linkedList.pollFirst());
153         System.out.println("Poll Last: " + linkedList.pollLast());
154         System.out.println("After polling: " + linkedList);
155 
156         // 5. Iterating with ListIterator (bidirectional)
157         System.out.println("\n5. Bidirectional iteration:");
158         ListIterator<String> listIterator = linkedList.listIterator();
159 
160         System.out.println("Forward iteration:");
161         while (listIterator.hasNext()) {
162             System.out.println("  " + listIterator.next());
```

```
        }

        System.out.println("Backward iteration:");
        while (listIterator.hasPrevious()) {
            System.out.println("  " + listIterator.previous());
        }

        // 6. Performance demonstration
        System.out.println("\n6. Performance comparison (ArrayList vs
            LinkedList):");

        int size = 100000;
        List<Integer> arrayList = new ArrayList<>();
        List<Integer> linkedListInt = new LinkedList<>();

        // Adding at end
        long startTime = System.nanoTime();
        for (int i = 0; i < size; i++) {
            arrayList.add(i);
        }
        long arrayListAddTime = System.nanoTime() - startTime;

        startTime = System.nanoTime();
        for (int i = 0; i < size; i++) {
            linkedListInt.add(i);
        }
        long linkedListAddTime = System.nanoTime() - startTime;

        System.out.println("Adding " + size + " elements at end:");
        System.out.println("  ArrayList: " + (arrayListAddTime /
            1000000) + " ms");
        System.out.println("  LinkedList: " + (linkedListAddTime /
            1000000) + " ms");

        // Adding at beginning
        arrayList.clear();
        linkedListInt.clear();

        startTime = System.nanoTime();
        for (int i = 0; i < 1000; i++) {
            arrayList.add(0, i);  // O(n) for ArrayList
        }
        arrayListAddTime = System.nanoTime() - startTime;

        startTime = System.nanoTime();
        for (int i = 0; i < 1000; i++) {
            linkedListInt.addFirst(i);  // O(1) for LinkedList
        }
        linkedListAddTime = System.nanoTime() - startTime;

        System.out.println("\nAdding 1000 elements at beginning:");
        System.out.println("  ArrayList: " + (arrayListAddTime /
            1000000) + " ms");
        System.out.println("  LinkedList: " + (linkedListAddTime /
            1000000) + " ms");

        // Random access
        startTime = System.nanoTime();
```

```
        for (int i = 0; i < 1000; i++) {
            arrayList.get(i);  // O(1) for ArrayList
        }
        arrayListAddTime = System.nanoTime() - startTime;

        startTime = System.nanoTime();
        for (int i = 0; i < 1000; i++) {
            linkedListInt.get(i);  // O(n) for LinkedList
        }
        linkedListAddTime = System.nanoTime() - startTime;

        System.out.println("\nRandom access (1000 gets):");
        System.out.println("  ArrayList: " + (arrayListAddTime /
            1000000) + " ms");
        System.out.println("  LinkedList: " + (linkedListAddTime /
            1000000) + " ms");
    }

    // ==================== PRACTICAL LIST EXAMPLES
        ====================
    public static void practicalListExamples() {
        System.out.println("\n=== PRACTICAL LIST EXAMPLES ===");

        // 1. Student Management System
        System.out.println("\n1. Student Management System:");

        class Student {
            int id;
            String name;
            double grade;

            Student(int id, String name, double grade) {
                this.id = id;
                this.name = name;
                this.grade = grade;
            }

            @Override
            public String toString() {
                return String.format("Student{id=%d, name='%s', grade
                    =%.2f}",
                                     id, name, grade);
            }
        }

        List<Student> students = new ArrayList<>();
        students.add(new Student(101, "Alice", 85.5));
        students.add(new Student(102, "Bob", 92.0));
        students.add(new Student(103, "Charlie", 78.5));
        students.add(new Student(104, "Diana", 95.0));

        System.out.println("All students:");
        students.forEach(System.out::println);

        // Find top student
        Student topStudent = Collections.max(students,
            Comparator.comparingDouble(s -> s.grade));
        System.out.println("\nTop student: " + topStudent);
```

```
270
271         // Calculate average grade
272         double average = students.stream()
273             .mapToDouble(s -> s.grade)
274             .average()
275             .orElse(0.0);
276         System.out.printf("Average grade: %.2f\n", average);
277
278         // Sort by grade
279         students.sort(Comparator.comparingDouble(s -> s.grade));
280         System.out.println("\nStudents sorted by grade:");
281         students.forEach(s -> System.out.println("  " + s));
282
283         // 2. Shopping Cart
284         System.out.println("\n2. Shopping Cart Example:");
285
286         List<String> cart = new LinkedList<>();
287         cart.add("Milk");
288         cart.add("Bread");
289         cart.add("Eggs");
290         cart.add("Butter");
291
292         System.out.println("Cart contents:");
293         for (int i = 0; i < cart.size(); i++) {
294             System.out.println((i + 1) + ". " + cart.get(i));
295         }
296
297         // Remove item
298         cart.remove("Eggs");
299         System.out.println("\nAfter removing Eggs: " + cart);
300
301         // Add at specific position
302         cart.add(1, "Cheese");
303         System.out.println("After adding Cheese at position 2: " + cart
                );
304
305         // 3. Task Manager
306         System.out.println("\n3. Task Manager (Priority Queue
                simulation):");
307
308         List<String> tasks = new ArrayList<>();
309         tasks.add("Complete report");
310         tasks.add("Email client");
311         tasks.add("Team meeting");
312         tasks.add("Code review");
313
314         System.out.println("Tasks: " + tasks);
315
316         // Process tasks
317         while (!tasks.isEmpty()) {
318             String task = tasks.remove(0);  // Process from beginning
319             System.out.println("Processing: " + task);
320         }
321
322         // 4. List operations with Java 8 Streams
323         System.out.println("\n4. List operations with Java 8 Streams:")
                ;
324
```

```
325         List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8,
                9, 10);
326
327         // Filter even numbers
328         List<Integer> evenNumbers = numbers.stream()
329             .filter(n -> n % 2 == 0)
330             .collect(Collectors.toList());
331         System.out.println("Even numbers: " + evenNumbers);
332
333         // Map to squares
334         List<Integer> squares = numbers.stream()
335             .map(n -> n * n)
336             .collect(Collectors.toList());
337         System.out.println("Squares: " + squares);
338
339         // Reduce to sum
340         int sum = numbers.stream()
341             .reduce(0, Integer::sum);
342         System.out.println("Sum: " + sum);
343
344         // Collect to string
345         String joined = numbers.stream()
346             .map(String::valueOf)
347             .collect(Collectors.joining(", "));
348         System.out.println("Joined: " + joined);
349     }
350
351     // ==================== MAIN METHOD ====================
352     public static void main(String[] args) {
353         System.out.println("=== COMPREHENSIVE LIST IMPLEMENTATIONS
                DEMONSTRATION ===\n");
354
355         System.out.println("=== LIST INTERFACE CHARACTERISTICS ===");
356         System.out.println("1. Ordered collection (maintains insertion
                order)");
357         System.out.println("2. Allows duplicate elements");
358         System.out.println("3. Allows null elements");
359         System.out.println("4. Positional access (get, set, add at
                index)");
360         System.out.println("5. Search operations (indexOf, lastIndexOf)
                ");
361         System.out.println("6. ListIterator for bidirectional traversal
                ");
362
363         System.out.println("\n=== ARRAYLIST VS LINKEDLIST ===");
364         System.out.println("| Feature              | ArrayList
                    | LinkedList         |");
365         System.out.println("
                |----------------------|--------------------|------------------|
                ");
366         System.out.println("| Internal Structure   | Dynamic array
                | Doubly linked list |");
367         System.out.println("| Random Access        | O(1) - Excellent
                | O(n) - Poor        |");
368         System.out.println("| Insert at beginning  | O(n) - Poor
                | O(1) - Excellent   |");
369         System.out.println("| Insert at end        | O(1) amortized
                | O(1) - Excellent   |");
```

```
        System.out.println("| Insert in middle     | O(n) - Poor
            | O(n) - Poor        |");
        System.out.println("| Memory Overhead      | Low
            | High (node objects)|");
        System.out.println("| Implements           | List
            | List, Deque, Queue |");

        // Demonstrate ArrayList
        demonstrateArrayList();

        // Demonstrate LinkedList
        demonstrateLinkedList();

        // Practical examples
        practicalListExamples();

        System.out.println("\n=== WHEN TO USE WHICH ===");
        System.out.println("Use ArrayList when:");
        System.out.println("  - Frequent random access needed");
        System.out.println("  - Mostly adding at end");
        System.out.println("  - Memory is a concern");

        System.out.println("\nUse LinkedList when:");
        System.out.println("  - Frequent insertions/deletions at
            beginning");
        System.out.println("  - Implementing stack/queue/deque");
        System.out.println("  - Memory overhead acceptable");

        System.out.println("\n=== COMMON METHODS ===");
        System.out.println("Common List methods:");
        System.out.println("  - add(), addAll()");
        System.out.println("  - get(), set()");
        System.out.println("  - remove(), removeAll()");
        System.out.println("  - contains(), indexOf()");
        System.out.println("  - size(), isEmpty(), clear()");
        System.out.println("  - subList()");
        System.out.println("  - sort(), replaceAll()");

        System.out.println("\nLinkedList-specific methods:");
        System.out.println("  - addFirst(), addLast()");
        System.out.println("  - getFirst(), getLast()");
        System.out.println("  - removeFirst(), removeLast()");
        System.out.println("  - peek(), poll(), push(), pop()");
        System.out.println("  - offer(), offerFirst(), offerLast()");

        System.out.println("\n=== THREAD SAFETY ===");
        System.out.println("ArrayList and LinkedList are NOT thread-
            safe");
        System.out.println("For thread-safe alternatives:");
        System.out.println("  - Use Collections.synchronizedList()");
        System.out.println("  - Use CopyOnWriteArrayList for read-heavy
             scenarios");
        System.out.println("  - Use Vector (legacy, synchronized)");

        // Thread safety example
        System.out.println("\n=== SYNCHRONIZED LIST EXAMPLE ===");
        List<String> syncList = Collections.synchronizedList(new
            ArrayList<>());
```

```
421
422         // Multiple threads can safely access syncList
423         Thread t1 = new Thread(() -> {
424             for (int i = 0; i < 5; i++) {
425                 syncList.add("Thread1-" + i);
426             }
427         });
428
429         Thread t2 = new Thread(() -> {
430             for (int i = 0; i < 5; i++) {
431                 syncList.add("Thread2-" + i);
432             }
433         });
434
435         t1.start();
436         t2.start();
437
438         try {
439             t1.join();
440             t2.join();
441         } catch (InterruptedException e) {
442             e.printStackTrace();
443         }
444
445         System.out.println("Synchronized list size: " + syncList.size()
                );
446         System.out.println("Contents: " + syncList);
447     }
448 }
```

Listing 5: Complete List Implementations (ArrayList and LinkedList)

ListDemo Program Output

```
=== COMPREHENSIVE LIST IMPLEMENTATIONS DEMONSTRATION ===

=== LIST INTERFACE CHARACTERISTICS ===
1. Ordered collection (maintains insertion order)
2. Allows duplicate elements
3. Allows null elements
4. Positional access (get, set, add at index)
5. Search operations (indexOf, lastIndexOf)
6. ListIterator for bidirectional traversal

=== ARRAYLIST VS LINKEDLIST ===
| Feature              | ArrayList           | LinkedList          |
|----------------------|---------------------|---------------------|
| Internal Structure   | Dynamic array       | Doubly linked list  |
| Random Access        | O(1) - Excellent    | O(n) - Poor         |
| Insert at beginning  | O(n) - Poor         | O(1) - Excellent    |
| Insert at end        | O(1) amortized      | O(1) - Excellent    |
| Insert in middle     | O(n) - Poor         | O(n) - Poor         |
| Memory Overhead      | Low                 | High (node objects) |
```

```
| Implements           | List                | List, Deque, Queue |

=== ARRAYLIST DEMONSTRATION ===

1. Adding elements:
ArrayList after additions: [Apple, Blueberry, Banana, Cherry, Date, Elderberry]
Size: 6
Capacity (not exposed directly)

2. Accessing elements:
Element at index 2: Banana
First element: Apple
Last element: Elderberry

3. Searching elements:
Contains 'Banana': true
Contains 'Grapes': false
Index of 'Cherry': 3
Last index of 'Date': 4

4. Iterating through ArrayList:
a) Using for loop:
  Index 0: Apple
  Index 1: Blueberry
  Index 2: Banana
  Index 3: Cherry
  Index 4: Date
  Index 5: Elderberry

b) Using enhanced for loop:
  Fruit: Apple
  Fruit: Blueberry
  Fruit: Banana
  Fruit: Cherry
  Fruit: Date
  Fruit: Elderberry

c) Using Iterator:
  Apple
  Blueberry
  Banana
  Cherry
  Date
  Elderberry

d) Using forEach (Java 8+):
  Apple
```

```
  Blueberry
  Banana
  Cherry
  Date
  Elderberry

5. Modifying elements:
Before modification: [Apple, Blueberry, Banana, Cherry, Date, Elderberry]
After setting index 2 to Cranberry: [Apple, Blueberry, Cranberry, Cherry, Date, Eld

6. Removing elements:
After removing 'Banana': [Apple, Blueberry, Cranberry, Cherry, Date, Elderberry]
After removing index 0: [Blueberry, Cranberry, Cherry, Date, Elderberry]
After removing elements starting with 'E': [Blueberry, Cranberry, Cherry, Date]

7. Sublist operations:
Sublist (1, 3): [Cranberry, Cherry]
After modifying sublist: [Blueberry, Fig, Cherry, Date]

8. Sorting and shuffling:
Sorted: [Blueberry, Cherry, Date, Fig]
Shuffled: [Fig, Blueberry, Date, Cherry]
Sorted by length: [Fig, Date, Cherry, Blueberry]

9. Converting to array:
Array: [Fig, Date, Cherry, Blueberry]

10. Clearing list:
Is empty? false
After clear - Is empty? true
Size: 0

=== LINKEDLIST DEMONSTRATION ===

1. Adding elements:
LinkedList: [New First, First, Middle, Second, Last]
Size: 5

2. Queue operations (FIFO):
After offer: [New First, First, Middle, Second, Last, Queue End]
Peek (head): New First
Poll (remove head): New First
After poll: [First, Middle, Second, Last, Queue End]

3. Stack operations (LIFO):
After push: [Stack Top, First, Middle, Second, Last, Queue End]
Peek (top): Stack Top
```

```
Pop (remove top): Stack Top
After pop: [First, Middle, Second, Last, Queue End]

4. Deque operations:
After offerFirst/Last: [Front, First, Middle, Second, Last, Queue End, Back]
Peek First: Front
Peek Last: Back
Poll First: Front
Poll Last: Back
After polling: [First, Middle, Second, Last, Queue End]

5. Bidirectional iteration:
Forward iteration:
  First
  Middle
  Second
  Last
  Queue End
Backward iteration:
  Queue End
  Last
  Second
  Middle
  First

6. Performance comparison (ArrayList vs LinkedList):
Adding 100000 elements at end:
  ArrayList: 12 ms
  LinkedList: 15 ms

Adding 1000 elements at beginning:
  ArrayList: 45 ms
  LinkedList: 1 ms

Random access (1000 gets):
  ArrayList: 0 ms
  LinkedList: 120 ms

=== PRACTICAL LIST EXAMPLES ===

1. Student Management System:
All students:
Student{id=101, name='Alice', grade=85.50}
Student{id=102, name='Bob', grade=92.00}
Student{id=103, name='Charlie', grade=78.50}
Student{id=104, name='Diana', grade=95.00}
```

```
Top student: Student{id=104, name='Diana', grade=95.00}
Average grade: 87.75

Students sorted by grade:
  Student{id=103, name='Charlie', grade=78.50}
  Student{id=101, name='Alice', grade=85.50}
  Student{id=102, name='Bob', grade=92.00}
  Student{id=104, name='Diana', grade=95.00}

2. Shopping Cart Example:
Cart contents:
1. Milk
2. Bread
3. Eggs
4. Butter

After removing Eggs: [Milk, Bread, Butter]
After adding Cheese at position 2: [Milk, Cheese, Bread, Butter]

3. Task Manager (Priority Queue simulation):
Tasks: [Complete report, Email client, Team meeting, Code review]
Processing: Complete report
Processing: Email client
Processing: Team meeting
Processing: Code review

4. List operations with Java 8 Streams:
Even numbers: [2, 4, 6, 8, 10]
Squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Sum: 55
Joined: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

=== WHEN TO USE WHICH ===
Use ArrayList when:
  - Frequent random access needed
  - Mostly adding at end
  - Memory is a concern

Use LinkedList when:
  - Frequent insertions/deletions at beginning
  - Implementing stack/queue/deque
  - Memory overhead acceptable

=== COMMON METHODS ===
Common List methods:
  - add(), addAll()
  - get(), set()
```

```
  - remove(), removeAll()
  - contains(), indexOf()
  - size(), isEmpty(), clear()
  - subList()
  - sort(), replaceAll()

LinkedList-specific methods:
  - addFirst(), addLast()
  - getFirst(), getLast()
  - removeFirst(), removeLast()
  - peek(), poll(), push(), pop()
  - offer(), offerFirst(), offerLast()

=== THREAD SAFETY ===
ArrayList and LinkedList are NOT thread-safe
For thread-safe alternatives:
  - Use Collections.synchronizedList()
  - Use CopyOnWriteArrayList for read-heavy scenarios
  - Use Vector (legacy, synchronized)

=== SYNCHRONIZED LIST EXAMPLE ===
Synchronized list size: 10
Contents: [Thread1-0, Thread1-1, Thread1-2, Thread1-3, Thread1-4, Thread2-0, Thread
```

4.4 Set Interface and Implementations

```java
import java.util.*;
import java.util.stream.Collectors;

public class SetDemo {

    // ==================== HASHSET DEMONSTRATION ====================
    public static void demonstrateHashSet() {
        System.out.println("\n=== HASHSET DEMONSTRATION ===");

        // 1. Creating HashSets
        Set<String> hashSet1 = new HashSet<>();  // Default initial
            capacity 16
        Set<Integer> hashSet2 = new HashSet<>(20);  // Initial capacity
             20
        Set<Double> hashSet3 = new HashSet<>(10, 0.75f);  // Capacity
            10, load factor 0.75

        // 2. Adding elements
        System.out.println("\n1. Adding elements:");
        hashSet1.add("Apple");
        hashSet1.add("Banana");
        hashSet1.add("Cherry");
        hashSet1.add("Apple");  // Duplicate - won't be added
        hashSet1.add(null);     // HashSet allows one null
        hashSet1.add(null);     // Second null - won't be added
```

```
23
24          System.out.println("HashSet after additions: " + hashSet1);
25          System.out.println("Size: " + hashSet1.size());
26          System.out.println("Is empty? " + hashSet1.isEmpty());
27
28          // 3. Bulk operations
29          System.out.println("\n2. Bulk operations:");
30          Set<String> fruits = new HashSet<>(Arrays.asList("Orange", "
                Grapes", "Mango"));
31          hashSet1.addAll(fruits);
32          System.out.println("After addAll: " + hashSet1);
33
34          // 4. Checking elements
35          System.out.println("\n3. Checking elements:");
36          System.out.println("Contains 'Banana': " + hashSet1.contains("
                Banana"));
37          System.out.println("Contains 'Pineapple': " + hashSet1.contains
                ("Pineapple"));
38          System.out.println("Contains null: " + hashSet1.contains(null))
                ;
39
40          // 5. Removing elements
41          System.out.println("\n4. Removing elements:");
42          hashSet1.remove("Cherry");
43          System.out.println("After removing 'Cherry': " + hashSet1);
44
45          hashSet1.removeIf(fruit -> fruit != null && fruit.startsWith("B
                "));
46          System.out.println("After removing elements starting with 'B':
                " + hashSet1);
47
48          hashSet1.removeAll(Arrays.asList("Orange", "Grapes"));
49          System.out.println("After removeAll: " + hashSet1);
50
51          hashSet1.retainAll(Arrays.asList("Apple", "Mango", "Kiwi"));
52          System.out.println("After retainAll (Apple, Mango, Kiwi): " +
                hashSet1);
53
54          // 6. Iterating through HashSet
55          System.out.println("\n5. Iterating through HashSet:");
56
57          System.out.println("a) Using enhanced for loop:");
58          for (String fruit : hashSet1) {
59              System.out.println("  " + fruit);
60          }
61
62          System.out.println("\nb) Using Iterator:");
63          Iterator<String> iterator = hashSet1.iterator();
64          while (iterator.hasNext()) {
65              System.out.println("  " + iterator.next());
66          }
67
68          System.out.println("\nc) Using forEach (Java 8+):");
69          hashSet1.forEach(System.out::println);
70
71          // 7. Set operations
72          System.out.println("\n6. Set operations:");
```

```
73          Set<Integer> setA = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5))
                ;
74          Set<Integer> setB = new HashSet<>(Arrays.asList(4, 5, 6, 7, 8))
                ;
75
76          System.out.println("Set A: " + setA);
77          System.out.println("Set B: " + setB);
78
79          // Union
80          Set<Integer> union = new HashSet<>(setA);
81          union.addAll(setB);
82          System.out.println("Union (A     B): " + union);
83
84          // Intersection
85          Set<Integer> intersection = new HashSet<>(setA);
86          intersection.retainAll(setB);
87          System.out.println("Intersection (A     B): " + intersection);
88
89          // Difference
90          Set<Integer> difference = new HashSet<>(setA);
91          difference.removeAll(setB);
92          System.out.println("Difference (A - B): " + difference);
93
94          // Symmetric Difference (elements in either set but not both)
95          Set<Integer> symmetricDiff = new HashSet<>(union);
96          symmetricDiff.removeAll(intersection);
97          System.out.println("Symmetric Difference (A    B): " +
                symmetricDiff);
98
99          // 8. Subset and superset checks
100         System.out.println("\n7. Subset and superset checks:");
101         Set<Integer> subset = new HashSet<>(Arrays.asList(1, 2));
102         System.out.println("Is {1,2} subset of A? " + setA.containsAll(
                subset));
103         System.out.println("Is A superset of {1,2}? " + setA.
                containsAll(subset));
104         System.out.println("Is {1,2,6} subset of A? " + setA.
                containsAll(Arrays.asList(1, 2, 6)));
105
106         // 9. Converting to array
107         System.out.println("\n8. Converting to array:");
108         Object[] array = hashSet1.toArray();
109         System.out.println("Array: " + Arrays.toString(array));
110
111         String[] stringArray = hashSet1.toArray(new String[0]);
112         System.out.println("String array: " + Arrays.toString(
                stringArray));
113
114         // 10. Clearing set
115         System.out.println("\n9. Clearing set:");
116         hashSet1.clear();
117         System.out.println("After clear - Size: " + hashSet1.size());
118         System.out.println("Is empty? " + hashSet1.isEmpty());
119     }
120
121     // ==================== TREESET DEMONSTRATION ====================
122     public static void demonstrateTreeSet() {
123         System.out.println("\n=== TREESET DEMONSTRATION ===");
```

```
124
125         // 1. Creating TreeSets
126         TreeSet<Integer> treeSet1 = new TreeSet<>();  // Natural
                ordering
127         TreeSet<String> treeSet2 = new TreeSet<>(Comparator.
                reverseOrder());  // Custom comparator
128         TreeSet<Student> treeSet3 = new TreeSet<>(Comparator.comparing(
                s -> s.name));  // Custom object
129
130         // 2. Adding elements (automatically sorted)
131         System.out.println("\n1. Adding elements (automatically sorted)
                :");
132         treeSet1.add(5);
133         treeSet1.add(2);
134         treeSet1.add(8);
135         treeSet1.add(1);
136         treeSet1.add(5);  // Duplicate - won't be added
137
138         System.out.println("TreeSet (natural order): " + treeSet1);
139         System.out.println("Size: " + treeSet1.size());
140
141         // TreeSet doesn't allow null (throws NullPointerException)
142         // treeSet1.add(null);  // Would throw NullPointerException
143
144         // 3. Special navigation methods
145         System.out.println("\n2. Navigation methods:");
146
147         System.out.println("First element: " + treeSet1.first());
148         System.out.println("Last element: " + treeSet1.last());
149
150         System.out.println("\nCeiling (>= 3): " + treeSet1.ceiling(3));
                // Smallest element >= 3
151         System.out.println("Floor (<= 3): " + treeSet1.floor(3));
                // Largest element <= 3
152
153         System.out.println("\nHigher (> 3): " + treeSet1.higher(3));
                // Smallest element > 3
154         System.out.println("Lower (< 3): " + treeSet1.lower(3));
                // Largest element < 3
155
156         // 4. Subset operations
157         System.out.println("\n3. Subset operations:");
158
159         System.out.println("HeadSet (< 5): " + treeSet1.headSet(5));
                // Elements < 5
160         System.out.println("HeadSet (<= 5): " + treeSet1.headSet(5,
                true)); // Elements <= 5
161
162         System.out.println("TailSet (>= 5): " + treeSet1.tailSet(5));
                // Elements >= 5
163         System.out.println("TailSet (> 5): " + treeSet1.tailSet(5,
                false)); // Elements > 5
164
165         System.out.println("SubSet (2 <= x < 8): " + treeSet1.subSet(2,
                8));      // 2 <= x < 8
166         System.out.println("SubSet (2 <= x <= 8): " + treeSet1.subSet
                (2, true, 8, true)); // 2 <= x <= 8
167
```

```
        // 5. Polling elements
        System.out.println("\n4. Polling elements:");
        System.out.println("TreeSet before polling: " + treeSet1);
        System.out.println("Poll first: " + treeSet1.pollFirst());
        System.out.println("Poll last: " + treeSet1.pollLast());
        System.out.println("After polling: " + treeSet1);

        // 6. Descending set
        System.out.println("\n5. Descending set:");
        NavigableSet<Integer> descendingSet = treeSet1.descendingSet();
        System.out.println("Original set: " + treeSet1);
        System.out.println("Descending set: " + descendingSet);

        // 7. Iterator in descending order
        System.out.println("\n6. Iterator in descending order:");
        Iterator<Integer> descendingIterator = treeSet1.
            descendingIterator();
        System.out.print("Descending order: ");
        while (descendingIterator.hasNext()) {
            System.out.print(descendingIterator.next() + " ");
        }
        System.out.println();

        // 8. Custom comparator demonstration
        System.out.println("\n7. Custom comparator demonstration:");

        TreeSet<String> caseInsensitiveSet = new TreeSet<>(String.
            CASE_INSENSITIVE_ORDER);
        caseInsensitiveSet.add("Apple");
        caseInsensitiveSet.add("banana");
        caseInsensitiveSet.add("apple");  // Won't be added (case-
            insensitive duplicate)
        caseInsensitiveSet.add("Cherry");

        System.out.println("Case-insensitive TreeSet: " +
            caseInsensitiveSet);
        System.out.println("Contains 'APPLE': " + caseInsensitiveSet.
            contains("APPLE"));

        // 9. TreeSet with custom objects
        System.out.println("\n8. TreeSet with custom objects:");

        class Product implements Comparable<Product> {
            String name;
            double price;

            Product(String name, double price) {
                this.name = name;
                this.price = price;
            }

            @Override
            public int compareTo(Product other) {
                return Double.compare(this.price, other.price);
            }

            @Override
            public String toString() {
```

```
221                 return String.format("%s: $%.2f", name, price);
222             }
223         }
224 
225         TreeSet<Product> products = new TreeSet<>();
226         products.add(new Product("Laptop", 999.99));
227         products.add(new Product("Phone", 699.99));
228         products.add(new Product("Tablet", 399.99));
229         products.add(new Product("Headphones", 199.99));
230 
231         System.out.println("Products sorted by price:");
232         products.forEach(System.out::println);
233 
234         // Find product with price <= 500
235         Product searchKey = new Product("", 500.0);
236         Product affordable = products.floor(searchKey);
237         System.out.println("\nMost expensive product under $500: " +
                affordable);
238     }
239 
240     // ==================== HASHSET VS TREESET PERFORMANCE
            ====================
241     public static void comparePerformance() {
242         System.out.println("\n=== HASHSET VS TREESET PERFORMANCE ===");
243 
244         int size = 100000;
245         Set<Integer> hashSet = new HashSet<>();
246         Set<Integer> treeSet = new TreeSet<>();
247 
248         Random random = new Random();
249 
250         // 1. Add performance
251         System.out.println("\n1. Add performance:");
252 
253         long startTime = System.nanoTime();
254         for (int i = 0; i < size; i++) {
255             hashSet.add(random.nextInt(size * 10));
256         }
257         long hashSetAddTime = System.nanoTime() - startTime;
258 
259         startTime = System.nanoTime();
260         for (int i = 0; i < size; i++) {
261             treeSet.add(random.nextInt(size * 10));
262         }
263         long treeSetAddTime = System.nanoTime() - startTime;
264 
265         System.out.printf("Adding %d random elements:\n", size);
266         System.out.printf("  HashSet:  %.2f ms\n", hashSetAddTime /
                1000000.0);
267         System.out.printf("  TreeSet:  %.2f ms\n", treeSetAddTime /
                1000000.0);
268 
269         // 2. Contains performance
270         System.out.println("\n2. Contains performance:");
271 
272         startTime = System.nanoTime();
273         for (int i = 0; i < 1000; i++) {
274             hashSet.contains(random.nextInt(size * 10));
```

```
275         }
276         long hashSetContainsTime = System.nanoTime() - startTime;
277 
278         startTime = System.nanoTime();
279         for (int i = 0; i < 1000; i++) {
280             treeSet.contains(random.nextInt(size * 10));
281         }
282         long treeSetContainsTime = System.nanoTime() - startTime;
283 
284         System.out.println("Checking contains 1000 times:");
285         System.out.printf("  HashSet:  %.2f ms\n", hashSetContainsTime
                / 1000000.0);
286         System.out.printf("  TreeSet:  %.2f ms\n", treeSetContainsTime
                / 1000000.0);
287 
288         // 3. Iteration performance
289         System.out.println("\n3. Iteration performance:");
290 
291         startTime = System.nanoTime();
292         int hashSetSum = 0;
293         for (int num : hashSet) {
294             hashSetSum += num;
295         }
296         long hashSetIterateTime = System.nanoTime() - startTime;
297 
298         startTime = System.nanoTime();
299         int treeSetSum = 0;
300         for (int num : treeSet) {
301             treeSetSum += num;
302         }
303         long treeSetIterateTime = System.nanoTime() - startTime;
304 
305         System.out.println("Iterating through all elements:");
306         System.out.printf("  HashSet:  %.2f ms\n", hashSetIterateTime /
                1000000.0);
307         System.out.printf("  TreeSet:  %.2f ms\n", treeSetIterateTime /
                1000000.0);
308     }
309 
310     // ==================== PRACTICAL SET EXAMPLES ====================
311     public static void practicalSetExamples() {
312         System.out.println("\n=== PRACTICAL SET EXAMPLES ===");
313 
314         // 1. Removing duplicates from list
315         System.out.println("\n1. Removing duplicates from list:");
316 
317         List<String> listWithDuplicates = Arrays.asList(
318             "Apple", "Banana", "Apple", "Orange", "Banana", "Grapes"
319         );
320         System.out.println("List with duplicates: " +
                listWithDuplicates);
321 
322         Set<String> uniqueSet = new HashSet<>(listWithDuplicates);
323         List<String> listWithoutDuplicates = new ArrayList<>(uniqueSet)
                ;
324         System.out.println("List without duplicates: " +
                listWithoutDuplicates);
325 
```

```
        // 2. Finding common elements between lists
        System.out.println("\n2. Finding common elements between lists:
            ");

        List<String> list1 = Arrays.asList("A", "B", "C", "D", "E");
        List<String> list2 = Arrays.asList("C", "D", "E", "F", "G");

        Set<String> set1 = new HashSet<>(list1);
        Set<String> set2 = new HashSet<>(list2);

        set1.retainAll(set2);  // Intersection
        System.out.println("List 1: " + list1);
        System.out.println("List 2: " + list2);
        System.out.println("Common elements: " + set1);

        // 3. Unique word counter
        System.out.println("\n3. Unique word counter:");

        String text = "the quick brown fox jumps over the lazy dog the
            dog was not lazy";
        String[] words = text.split("\\s+");

        Set<String> uniqueWords = new HashSet<>(Arrays.asList(words));
        System.out.println("Text: " + text);
        System.out.println("Total words: " + words.length);
        System.out.println("Unique words: " + uniqueWords.size());
        System.out.println("Unique words sorted: " + new TreeSet<>(
            uniqueWords));

        // 4. Student enrollment system
        System.out.println("\n4. Student enrollment system:");

        class Course {
            String code;
            String name;
            Set<String> enrolledStudents;

            Course(String code, String name) {
                this.code = code;
                this.name = name;
                this.enrolledStudents = new HashSet<>();
            }

            boolean enrollStudent(String studentId) {
                return enrolledStudents.add(studentId);
            }

            boolean dropStudent(String studentId) {
                return enrolledStudents.remove(studentId);
            }

            boolean isStudentEnrolled(String studentId) {
                return enrolledStudents.contains(studentId);
            }

            Set<String> getCommonStudents(Course other) {
                Set<String> common = new HashSet<>(this.
                    enrolledStudents);
```

```
            common.retainAll(other.enrolledStudents);
            return common;
        }

        @Override
        public String toString() {
            return String.format("%s (%s): %d students", name, code
                , enrolledStudents.size());
        }
    }

    Course cs101 = new Course("CS101", "Programming Fundamentals");
    Course math201 = new Course("MATH201", "Calculus");

    cs101.enrollStudent("S001");
    cs101.enrollStudent("S002");
    cs101.enrollStudent("S003");

    math201.enrollStudent("S002");
    math201.enrollStudent("S003");
    math201.enrollStudent("S004");

    System.out.println(cs101);
    System.out.println(math201);
    System.out.println("Common students: " + cs101.
       getCommonStudents(math201));

    // 5. Lottery system (unique random numbers)
    System.out.println("\n5. Lottery system (unique random numbers)
       :");

    Set<Integer> lotteryNumbers = new HashSet<>();
    Random random = new Random();

    while (lotteryNumbers.size() < 6) {
        lotteryNumbers.add(random.nextInt(49) + 1);  // Numbers
           1-49
    }

    System.out.println("Lottery numbers (unordered): " +
       lotteryNumbers);
    System.out.println("Lottery numbers (sorted): " + new TreeSet
       <>(lotteryNumbers));

    // 6. Tag system for blog posts
    System.out.println("\n6. Tag system for blog posts:");

    Map<String, Set<String>> postTags = new HashMap<>();

    // Add tags to posts
    postTags.computeIfAbsent("post1", k -> new HashSet<>())
            .addAll(Arrays.asList("java", "programming", "tutorial"
               ));

    postTags.computeIfAbsent("post2", k -> new HashSet<>())
            .addAll(Arrays.asList("java", "collections", "framework
               "));

```

```
        postTags.computeIfAbsent("post3", k -> new HashSet<>())
                .addAll(Arrays.asList("python", "tutorial", "beginners"
                    ));

        // Find posts with specific tag
        String searchTag = "java";
        System.out.println("Posts with tag '" + searchTag + "':");
        postTags.entrySet().stream()
            .filter(entry -> entry.getValue().contains(searchTag))
            .forEach(entry -> System.out.println("  " + entry.getKey())
                );

        // Find common tags between posts
        Set<String> commonTags = new HashSet<>(postTags.get("post1"));
        commonTags.retainAll(postTags.get("post2"));
        System.out.println("Common tags between post1 and post2: " +
            commonTags);
    }

    // ==================== MAIN METHOD ====================
    public static void main(String[] args) {
        System.out.println("=== COMPREHENSIVE SET IMPLEMENTATIONS
            DEMONSTRATION ===\n");

        System.out.println("=== SET INTERFACE CHARACTERISTICS ===");
        System.out.println("1. No duplicate elements");
        System.out.println("2. At most one null element (depends on
            implementation)");
        System.out.println("3. No positional access (no get(index)
            method)");
        System.out.println("4. Mathematical set operations (union,
            intersection, etc.)");

        System.out.println("\n=== HASHSET VS TREESET ===");
        System.out.println("| Feature              | HashSet          |
             TreeSet          |");
        System.out.println("
            |----------------------|------------------|------------------|
            ");
        System.out.println("| Ordering             | No order         |
             Sorted order     |");
        System.out.println("| Null allowed         | Yes (one)        |
             No               |");
        System.out.println("| Performance          | O(1) average     |
             O(log n)         |");
        System.out.println("| Internal Structure   | Hash table       |
             Red-Black Tree   |");
        System.out.println("| Navigation methods   | No               |
             Yes              |");
        System.out.println("| Comparator           | Uses equals()    |
             Uses compareTo() |");

        System.out.println("\n=== LINKEDHASHSET ===");
        System.out.println("LinkedHashSet: HashSet + maintains
            insertion order");
        System.out.println("  - Uses hash table with linked list");
        System.out.println("  - Slightly slower than HashSet");
        System.out.println("  - Maintains insertion order");
```

```
471             System.out.println("  - Allows one null element");
472
473             // Demonstrate HashSet
474             demonstrateHashSet();
475
476             // Demonstrate TreeSet
477             demonstrateTreeSet();
478
479             // Performance comparison
480             comparePerformance();
481
482             // Practical examples
483             practicalSetExamples();
484
485             System.out.println("\n=== WHEN TO USE WHICH ===");
486             System.out.println("Use HashSet when:");
487             System.out.println("  - Don't need ordering");
488             System.out.println("  - Need best performance");
489             System.out.println("  - Need to store null values");
490
491             System.out.println("\nUse TreeSet when:");
492             System.out.println("  - Need sorted elements");
493             System.out.println("  - Need navigation methods");
494             System.out.println("  - Will iterate frequently");
495
496             System.out.println("\nUse LinkedHashSet when:");
497             System.out.println("  - Need to maintain insertion order");
498             System.out.println("  - Need predictable iteration order");
499
500             System.out.println("\n=== COMMON METHODS ===");
501             System.out.println("Common Set methods:");
502             System.out.println("  - add(), addAll()");
503             System.out.println("  - remove(), removeAll()");
504             System.out.println("  - contains(), containsAll()");
505             System.out.println("  - size(), isEmpty(), clear()");
506             System.out.println("  - retainAll() (intersection)");
507
508             System.out.println("\nTreeSet-specific methods:");
509             System.out.println("  - first(), last()");
510             System.out.println("  - ceiling(), floor()");
511             System.out.println("  - higher(), lower()");
512             System.out.println("  - headSet(), tailSet(), subSet()");
513             System.out.println("  - pollFirst(), pollLast()");
514             System.out.println("  - descendingSet(), descendingIterator()")
                    ;
515
516             System.out.println("\n=== EQUALS AND HASHCODE CONTRACT ===");
517             System.out.println("For HashSet and HashMap:");
518             System.out.println("1. If two objects are equal, they must have
                    same hashCode");
519             System.out.println("2. Objects with same hashCode may not be
                    equal");
520             System.out.println("3. hashCode() should be consistent with
                    equals()");
521
522             System.out.println("\n=== THREAD SAFETY ===");
523             System.out.println("HashSet and TreeSet are NOT thread-safe");
524             System.out.println("For thread-safe alternatives:");
```

```
525         System.out.println("  - Use Collections.synchronizedSet()");
526         System.out.println("  - Use CopyOnWriteArraySet for read-heavy
                scenarios");
527         System.out.println("  - Use ConcurrentSkipListSet for
                concurrent sorted set");
528
529         // Thread safety example
530         System.out.println("\n=== SYNCHRONIZED SET EXAMPLE ===");
531         Set<String> syncSet = Collections.synchronizedSet(new HashSet
                <>());
532
533         Thread t1 = new Thread(() -> {
534             for (int i = 0; i < 5; i++) {
535                 syncSet.add("Thread1-" + i);
536             }
537         });
538
539         Thread t2 = new Thread(() -> {
540             for (int i = 0; i < 5; i++) {
541                 syncSet.add("Thread2-" + i);
542             }
543         });
544
545         t1.start();
546         t2.start();
547
548         try {
549             t1.join();
550             t2.join();
551         } catch (InterruptedException e) {
552             e.printStackTrace();
553         }
554
555         System.out.println("Synchronized set size: " + syncSet.size());
556
557         // LinkedHashSet example
558         System.out.println("\n=== LINKEDHASHSET EXAMPLE ===");
559         Set<String> linkedHashSet = new LinkedHashSet<>();
560         linkedHashSet.add("First");
561         linkedHashSet.add("Second");
562         linkedHashSet.add("Third");
563         linkedHashSet.add("First");  // Duplicate
564
565         System.out.println("LinkedHashSet (maintains insertion order):
                " + linkedHashSet);
566     }
567
568     // Student class for TreeSet demonstration
569     static class Student {
570         String name;
571         int id;
572
573         Student(String name, int id) {
574             this.name = name;
575             this.id = id;
576         }
577
578         @Override
```

```
        public String toString() {
            return name + " (" + id + ")";
        }
    }
}
```

Listing 6: Complete Set Implementations (HashSet and TreeSet)

SetDemo Program Output

```
=== COMPREHENSIVE SET IMPLEMENTATIONS DEMONSTRATION ===

=== SET INTERFACE CHARACTERISTICS ===
1. No duplicate elements
2. At most one null element (depends on implementation)
3. No positional access (no get(index) method)
4. Mathematical set operations (union, intersection, etc.)

=== HASHSET VS TREESET ===
| Feature              | HashSet          | TreeSet          |
|----------------------|------------------|------------------|
| Ordering             | No order         | Sorted order     |
| Null allowed         | Yes (one)        | No               |
| Performance          | O(1) average     | O(log n)         |
| Internal Structure   | Hash table       | Red-Black Tree   |
| Navigation methods   | No               | Yes              |
| Comparator           | Uses equals()    | Uses compareTo() |

=== LINKEDHASHSET ===
LinkedHashSet: HashSet + maintains insertion order
  - Uses hash table with linked list
  - Slightly slower than HashSet
  - Maintains insertion order
  - Allows one null element

=== HASHSET DEMONSTRATION ===

1. Adding elements:
HashSet after additions: [null, Apple, Cherry, Banana]
Size: 4
Is empty? false

2. Bulk operations:
After addAll: [null, Apple, Cherry, Orange, Grapes, Mango, Banana]

3. Checking elements:
Contains 'Banana': true
Contains 'Pineapple': false
Contains null: true
```

```
4. Removing elements:
After removing 'Cherry': [null, Apple, Orange, Grapes, Mango, Banana]
After removing elements starting with 'B': [null, Apple, Orange, Grapes, Mango]
After removeAll: [null, Apple, Mango]
After retainAll (Apple, Mango, Kiwi): [Apple, Mango]

5. Iterating through HashSet:
a) Using enhanced for loop:
  Apple
  Mango

b) Using Iterator:
  Apple
  Mango

c) Using forEach (Java 8+):
  Apple
  Mango

6. Set operations:
Set A: [1, 2, 3, 4, 5]
Set B: [4, 5, 6, 7, 8]
Union (A  B): [1, 2, 3, 4, 5, 6, 7, 8]
Intersection (A  B): [4, 5]
Difference (A - B): [1, 2, 3]
Symmetric Difference (A  B): [1, 2, 3, 6, 7, 8]

7. Subset and superset checks:
Is {1,2} subset of A? true
Is A superset of {1,2}? true
Is {1,2,6} subset of A? false

8. Converting to array:
Array: [Apple, Mango]
String array: [Apple, Mango]

9. Clearing set:
After clear - Size: 0
Is empty? true

=== TREESET DEMONSTRATION ===

1. Adding elements (automatically sorted):
TreeSet (natural order): [1, 2, 5, 8]
Size: 4
```

```
2. Navigation methods:
First element: 1
Last element: 8

Ceiling (>= 3): 5
Floor (<= 3): 2

Higher (> 3): 5
Lower (< 3): 2

3. Subset operations:
HeadSet (< 5): [1, 2]
HeadSet (<= 5): [1, 2, 5]
TailSet (>= 5): [5, 8]
TailSet (> 5): [8]
SubSet (2 <= x < 8): [2, 5]
SubSet (2 <= x <= 8): [2, 5, 8]

4. Polling elements:
TreeSet before polling: [1, 2, 5, 8]
Poll first: 1
Poll last: 8
After polling: [2, 5]

5. Descending set:
Original set: [2, 5]
Descending set: [5, 2]

6. Iterator in descending order:
Descending order: 5 2

7. Custom comparator demonstration:
Case-insensitive TreeSet: [Apple, banana, Cherry]
Contains 'APPLE': true

8. TreeSet with custom objects:
Products sorted by price:
Headphones: $199.99
Tablet: $399.99
Phone: $699.99
Laptop: $999.99

Most expensive product under $500: Tablet: $399.99

=== HASHSET VS TREESET PERFORMANCE ===

1. Add performance:
```

```
Adding 100000 random elements:
  HashSet:  12.45 ms
  TreeSet:  35.67 ms

2. Contains performance:
Checking contains 1000 times:
  HashSet:  0.12 ms
  TreeSet:  0.45 ms

3. Iteration performance:
Iterating through all elements:
  HashSet:  2.34 ms
  TreeSet:  2.89 ms

=== PRACTICAL SET EXAMPLES ===

1. Removing duplicates from list:
List with duplicates: [Apple, Banana, Apple, Orange, Banana, Grapes]
List without duplicates: [Apple, Grapes, Orange, Banana]

2. Finding common elements between lists:
List 1: [A, B, C, D, E]
List 2: [C, D, E, F, G]
Common elements: [C, D, E]

3. Unique word counter:
Text: the quick brown fox jumps over the lazy dog the dog was not lazy
Total words: 13
Unique words: 10
Unique words sorted: [brown, dog, fox, jumps, lazy, not, over, quick, the, was]

4. Student enrollment system:
Programming Fundamentals (CS101): 3 students
Calculus (MATH201): 3 students
Common students: [S002, S003]

5. Lottery system (unique random numbers):
Lottery numbers (unordered): [32, 1, 18, 20, 5, 45]
Lottery numbers (sorted): [1, 5, 18, 20, 32, 45]

6. Tag system for blog posts:
Posts with tag 'java':
  post1
  post2
Common tags between post1 and post2: [java]

=== WHEN TO USE WHICH ===
```

```
Use HashSet when:
  - Don't need ordering
  - Need best performance
  - Need to store null values

Use TreeSet when:
  - Need sorted elements
  - Need navigation methods
  - Will iterate frequently

Use LinkedHashSet when:
  - Need to maintain insertion order
  - Need predictable iteration order

=== COMMON METHODS ===
Common Set methods:
  - add(), addAll()
  - remove(), removeAll()
  - contains(), containsAll()
  - size(), isEmpty(), clear()
  - retainAll() (intersection)

TreeSet-specific methods:
  - first(), last()
  - ceiling(), floor()
  - higher(), lower()
  - headSet(), tailSet(), subSet()
  - pollFirst(), pollLast()
  - descendingSet(), descendingIterator()

=== EQUALS AND HASHCODE CONTRACT ===
For HashSet and HashMap:
1. If two objects are equal, they must have same hashCode
2. Objects with same hashCode may not be equal
3. hashCode() should be consistent with equals()

=== THREAD SAFETY ===
HashSet and TreeSet are NOT thread-safe
For thread-safe alternatives:
  - Use Collections.synchronizedSet()
  - Use CopyOnWriteArraySet for read-heavy scenarios
  - Use ConcurrentSkipListSet for concurrent sorted set

=== SYNCHRONIZED SET EXAMPLE ===
Synchronized set size: 10

=== LINKEDHASHSET EXAMPLE ===
```

```
LinkedHashSet (maintains insertion order): [First, Second, Third]
```

4.5 Map Interface and Implementations

```java
import java.util.*;
import java.util.Map.Entry;
import java.util.stream.Collectors;

public class MapDemo {

    // ==================== HASHMAP DEMONSTRATION ====================
    public static void demonstrateHashMap() {
        System.out.println("\n=== HASHMAP DEMONSTRATION ===");

        // 1. Creating HashMaps
        Map<String, Integer> hashMap1 = new HashMap<>();  // Default capacity 16, load factor 0.75
        Map<Integer, String> hashMap2 = new HashMap<>(20);  // Initial capacity 20
        Map<String, Double> hashMap3 = new HashMap<>(10, 0.8f);  // Capacity 10, load factor 0.8

        // 2. Adding key-value pairs
        System.out.println("\n1. Adding key-value pairs:");
        hashMap1.put("Apple", 10);
        hashMap1.put("Banana", 20);
        hashMap1.put("Cherry", 30);
        hashMap1.put(null, 0);       // HashMap allows one null key
        hashMap1.put("Date", null); // HashMap allows multiple null values
        hashMap1.put("Apple", 15);  // Update existing key

        System.out.println("HashMap: " + hashMap1);
        System.out.println("Size: " + hashMap1.size());
        System.out.println("Is empty? " + hashMap1.isEmpty());

        // 3. Accessing values
        System.out.println("\n2. Accessing values:");
        System.out.println("Value for 'Apple': " + hashMap1.get("Apple"));
        System.out.println("Value for 'Mango': " + hashMap1.get("Mango"));  // Returns null
        System.out.println("Value for null key: " + hashMap1.get(null));
        System.out.println("Value or default for 'Mango': " +
                           hashMap1.getOrDefault("Mango", -1));

        // 4. Checking keys and values
        System.out.println("\n3. Checking keys and values:");
        System.out.println("Contains key 'Banana': " + hashMap1.containsKey("Banana"));
        System.out.println("Contains key 'Grapes': " + hashMap1.containsKey("Grapes"));
        System.out.println("Contains value 20: " + hashMap1.containsValue(20));
```

```
42          System.out.println("Contains value 100: " + hashMap1.
                containsValue(100));
43
44          // 5. Updating values
45          System.out.println("\n4. Updating values:");
46          System.out.println("Original value for 'Banana': " + hashMap1.
                get("Banana"));
47          hashMap1.put("Banana", 25);  // Simple put
48          System.out.println("After put: " + hashMap1.get("Banana"));
49
50          hashMap1.putIfAbsent("Cherry", 35);  // Won't update (key
                exists)
51          hashMap1.putIfAbsent("Mango", 40);   // Will add (key doesn't
                exist)
52          System.out.println("After putIfAbsent: " + hashMap1);
53
54          // Compute methods
55          hashMap1.compute("Apple", (k, v) -> v + 10);  // Add 10 to
                existing value
56          System.out.println("After compute (Apple + 10): " + hashMap1.
                get("Apple"));
57
58          hashMap1.computeIfAbsent("Orange", k -> 50);  // Add if absent
59          hashMap1.computeIfPresent("Banana", (k, v) -> v * 2);  //
                Double if present
60          System.out.println("After compute methods: " + hashMap1);
61
62          // Merge method
63          hashMap1.merge("Apple", 5, (oldVal, newVal) -> oldVal + newVal)
                ;  // Add 5
64          System.out.println("After merge (Apple + 5): " + hashMap1.get("
                Apple"));
65
66          // 6. Removing entries
67          System.out.println("\n5. Removing entries:");
68          System.out.println("Before removal: " + hashMap1);
69          hashMap1.remove("Banana");
70          System.out.println("After removing 'Banana': " + hashMap1);
71
72          hashMap1.remove("Apple", 30);  // Remove only if value matches
73          System.out.println("After removing 'Apple' if value=30: " +
                hashMap1);
74
75          // 7. Iterating through HashMap
76          System.out.println("\n6. Iterating through HashMap:");
77
78          System.out.println("a) Using keySet():");
79          for (String key : hashMap1.keySet()) {
80              System.out.println("  Key: " + key + ", Value: " + hashMap1
                    .get(key));
81          }
82
83          System.out.println("\nb) Using values():");
84          for (Integer value : hashMap1.values()) {
85              System.out.println("  Value: " + value);
86          }
87
88          System.out.println("\nc) Using entrySet():");
```

```
89          for (Map.Entry<String, Integer> entry : hashMap1.entrySet()) {
90              System.out.println("  Key: " + entry.getKey() + ", Value: "
                    + entry.getValue());
91          }
92
93          System.out.println("\nd) Using forEach (Java 8+):");
94          hashMap1.forEach((key, value) ->
95              System.out.println("  Key: " + key + ", Value: " + value));
96
97          // 8. Bulk operations
98          System.out.println("\n7. Bulk operations:");
99          Map<String, Integer> anotherMap = new HashMap<>();
100         anotherMap.put("Grapes", 60);
101         anotherMap.put("Pineapple", 70);
102
103         hashMap1.putAll(anotherMap);
104         System.out.println("After putAll: " + hashMap1);
105
106         // 9. Replacing values
107         System.out.println("\n8. Replacing values:");
108         hashMap1.replace("Cherry", 100);
109         System.out.println("After replace 'Cherry': " + hashMap1.get("
                Cherry"));
110
111         hashMap1.replace("Cherry", 100, 150);  // Replace if old value
                matches
112         System.out.println("After conditional replace 'Cherry': " +
                hashMap1.get("Cherry"));
113
114         hashMap1.replaceAll((key, value) -> value + 10);  // Add 10 to
                all values
115         System.out.println("After replaceAll (+10): " + hashMap1);
116
117         // 10. Clearing map
118         System.out.println("\n9. Clearing map:");
119         hashMap1.clear();
120         System.out.println("After clear - Size: " + hashMap1.size());
121         System.out.println("Is empty? " + hashMap1.isEmpty());
122     }
123
124     // ==================== TREEMAP DEMONSTRATION ====================
125     public static void demonstrateTreeMap() {
126         System.out.println("\n=== TREEMAP DEMONSTRATION ===");
127
128         // 1. Creating TreeMaps
129         TreeMap<String, Integer> treeMap1 = new TreeMap<>();  //
                Natural ordering of keys
130         TreeMap<Integer, String> treeMap2 = new TreeMap<>(Comparator.
                reverseOrder());  // Descending
131         TreeMap<String, Student> treeMap3 = new TreeMap<>(String.
                CASE_INSENSITIVE_ORDER);  // Custom
132
133         // 2. Adding entries (automatically sorted by key)
134         System.out.println("\n1. Adding entries (sorted by key):");
135         treeMap1.put("Charlie", 25);
136         treeMap1.put("Alice", 30);
137         treeMap1.put("Bob", 28);
138         treeMap1.put("Alice", 32);  // Update existing
```

```
139
140         System.out.println("TreeMap: " + treeMap1);
141         System.out.println("Size: " + treeMap1.size());
142
143         // TreeMap doesn't allow null keys (throws NullPointerException
                )
144         // treeMap1.put(null, 0);  // Would throw NullPointerException
145         treeMap1.put("David", null);  // But allows null values
146
147         // 3. Navigation methods
148         System.out.println("\n2. Navigation methods:");
149
150         System.out.println("First key: " + treeMap1.firstKey());
151         System.out.println("Last key: " + treeMap1.lastKey());
152         System.out.println("First entry: " + treeMap1.firstEntry());
153         System.out.println("Last entry: " + treeMap1.lastEntry());
154
155         System.out.println("\nCeiling key (>= 'Brain'): " + treeMap1.
                ceilingKey("Brain"));
156         System.out.println("Floor key (<= 'Brain'): " + treeMap1.
                floorKey("Brain"));
157
158         System.out.println("\nHigher key (> 'Bob'): " + treeMap1.
                higherKey("Bob"));
159         System.out.println("Lower key (< 'Bob'): " + treeMap1.lowerKey(
                "Bob"));
160
161         System.out.println("\nCeiling entry (>= 'Brain'): " + treeMap1.
                ceilingEntry("Brain"));
162         System.out.println("Floor entry (<= 'Brain'): " + treeMap1.
                floorEntry("Brain"));
163
164         // 4. Submap operations
165         System.out.println("\n3. Submap operations:");
166
167         System.out.println("HeadMap (< 'Charlie'): " + treeMap1.headMap
                ("Charlie"));
168         System.out.println("HeadMap (<= 'Charlie'): " + treeMap1.
                headMap("Charlie", true));
169
170         System.out.println("TailMap (>= 'Bob'): " + treeMap1.tailMap("
                Bob"));
171         System.out.println("TailMap (> 'Bob'): " + treeMap1.tailMap("
                Bob", false));
172
173         System.out.println("SubMap ('Alice' <= x < 'Charlie'): " +
174                            treeMap1.subMap("Alice", "Charlie"));
175         System.out.println("SubMap ('Alice' <= x <= 'Charlie'): " +
176                            treeMap1.subMap("Alice", true, "Charlie",
                                true));
177
178         // 5. Polling entries
179         System.out.println("\n4. Polling entries:");
180         System.out.println("TreeMap before polling: " + treeMap1);
181         System.out.println("Poll first entry: " + treeMap1.
                pollFirstEntry());
182         System.out.println("Poll last entry: " + treeMap1.pollLastEntry
                ());
```

```
183         System.out.println("After polling: " + treeMap1);
184
185         // 6. Descending map
186         System.out.println("\n5. Descending map:");
187         NavigableMap<String, Integer> descendingMap = treeMap1.
                descendingMap();
188         System.out.println("Original map: " + treeMap1);
189         System.out.println("Descending map: " + descendingMap);
190
191         // 7. Using custom comparator
192         System.out.println("\n6. Using custom comparator:");
193         TreeMap<String, Integer> caseInsensitiveMap = new TreeMap<>(
                String.CASE_INSENSITIVE_ORDER);
194         caseInsensitiveMap.put("APPLE", 10);
195         caseInsensitiveMap.put("apple", 20);  // Will replace previous
                (case-insensitive)
196         caseInsensitiveMap.put("Banana", 30);
197
198         System.out.println("Case-insensitive TreeMap: " +
                caseInsensitiveMap);
199         System.out.println("Contains key 'Apple': " +
                caseInsensitiveMap.containsKey("Apple"));
200         System.out.println("Contains key 'apple': " +
                caseInsensitiveMap.containsKey("apple"));
201
202         // 8. TreeMap with complex keys
203         System.out.println("\n7. TreeMap with complex keys:");
204
205         class Product implements Comparable<Product> {
206             String name;
207             double price;
208
209             Product(String name, double price) {
210                 this.name = name;
211                 this.price = price;
212             }
213
214             @Override
215             public int compareTo(Product other) {
216                 return Double.compare(this.price, other.price);
217             }
218
219             @Override
220             public String toString() {
221                 return name;
222             }
223         }
224
225         TreeMap<Product, Integer> inventory = new TreeMap<>();
226         inventory.put(new Product("Laptop", 999.99), 10);
227         inventory.put(new Product("Phone", 699.99), 25);
228         inventory.put(new Product("Tablet", 399.99), 15);
229
230         System.out.println("Inventory sorted by product price:");
231         inventory.forEach((product, quantity) ->
232             System.out.printf("  %s ($%.2f): %d units\n",
233                               product.name, product.price, quantity));
234
```

```
235         // Find products under $500
236         Product searchKey = new Product("", 500.0);
237         Map.Entry<Product, Integer> affordable = inventory.floorEntry(
                searchKey);
238         if (affordable != null) {
239             System.out.println("\nMost expensive product under $500: "
                    +
240                                affordable.getKey().name + " (" +
                                       affordable.getValue() + " units)");
241         }
242     }
243
244     // ==================== HASHMAP VS TREEMAP PERFORMANCE
            ====================
245     public static void compareMapPerformance() {
246         System.out.println("\n=== HASHMAP VS TREEMAP PERFORMANCE ===");
247
248         int size = 100000;
249         Map<Integer, String> hashMap = new HashMap<>();
250         Map<Integer, String> treeMap = new TreeMap<>();
251
252         Random random = new Random();
253
254         // 1. Put performance
255         System.out.println("\n1. Put performance:");
256
257         long startTime = System.nanoTime();
258         for (int i = 0; i < size; i++) {
259             hashMap.put(random.nextInt(size * 10), "Value" + i);
260         }
261         long hashMapPutTime = System.nanoTime() - startTime;
262
263         startTime = System.nanoTime();
264         for (int i = 0; i < size; i++) {
265             treeMap.put(random.nextInt(size * 10), "Value" + i);
266         }
267         long treeMapPutTime = System.nanoTime() - startTime;
268
269         System.out.printf("Putting %d random entries:\n", size);
270         System.out.printf("  HashMap:  %.2f ms\n", hashMapPutTime /
                1000000.0);
271         System.out.printf("  TreeMap:  %.2f ms\n", treeMapPutTime /
                1000000.0);
272
273         // 2. Get performance
274         System.out.println("\n2. Get performance:");
275
276         // Get some existing keys
277         Integer[] keys = hashMap.keySet().toArray(new Integer[0]);
278
279         startTime = System.nanoTime();
280         for (int i = 0; i < 1000; i++) {
281             hashMap.get(keys[random.nextInt(keys.length)]);
282         }
283         long hashMapGetTime = System.nanoTime() - startTime;
284
285         startTime = System.nanoTime();
286         for (int i = 0; i < 1000; i++) {
```

```
            treeMap.get(keys[random.nextInt(keys.length)]);
        }
        long treeMapGetTime = System.nanoTime() - startTime;

        System.out.println("Getting 1000 random entries:");
        System.out.printf("  HashMap:  %.2f ms\n", hashMapGetTime /
            1000000.0);
        System.out.printf("  TreeMap:  %.2f ms\n", treeMapGetTime /
            1000000.0);

        // 3. Iteration performance
        System.out.println("\n3. Iteration performance:");

        startTime = System.nanoTime();
        int hashMapCount = 0;
        for (String value : hashMap.values()) {
            hashMapCount++;
        }
        long hashMapIterateTime = System.nanoTime() - startTime;

        startTime = System.nanoTime();
        int treeMapCount = 0;
        for (String value : treeMap.values()) {
            treeMapCount++;
        }
        long treeMapIterateTime = System.nanoTime() - startTime;

        System.out.println("Iterating through all values:");
        System.out.printf("  HashMap:  %.2f ms (count: %d)\n",
                          hashMapIterateTime / 1000000.0, hashMapCount);
        System.out.printf("  TreeMap:  %.2f ms (count: %d)\n",
                          treeMapIterateTime / 1000000.0, treeMapCount);
    }

    // ==================== PRACTICAL MAP EXAMPLES ====================
    public static void practicalMapExamples() {
        System.out.println("\n=== PRACTICAL MAP EXAMPLES ===");

        // 1. Word frequency counter
        System.out.println("\n1. Word frequency counter:");

        String text = "the quick brown fox jumps over the lazy dog the
            dog was not lazy";
        String[] words = text.split("\\s+");

        Map<String, Integer> wordFrequency = new HashMap<>();
        for (String word : words) {
            wordFrequency.merge(word, 1, Integer::sum);
        }

        System.out.println("Word frequencies:");
        wordFrequency.forEach((word, count) ->
            System.out.printf("  %-10s: %d\n", word, count));

        // Sort by frequency
        System.out.println("\nSorted by frequency (descending):");
        wordFrequency.entrySet().stream()
            .sorted((e1, e2) -> e2.getValue().compareTo(e1.getValue()))
```

```
342             .forEach(entry ->
343                 System.out.printf("  %-10s: %d\n", entry.getKey(),
                        entry.getValue()));
344
345         // 2. Student grades system
346         System.out.println("\n2. Student grades system:");
347
348         Map<String, List<Integer>> studentGrades = new HashMap<>();
349         studentGrades.put("Alice", new ArrayList<>(Arrays.asList(85,
                90, 78)));
350         studentGrades.put("Bob", new ArrayList<>(Arrays.asList(92, 88,
                95)));
351         studentGrades.put("Charlie", new ArrayList<>(Arrays.asList(75,
                80, 82)));
352
353         // Calculate average for each student
354         Map<String, Double> studentAverages = new TreeMap<>();  //
                Sorted by name
355         studentGrades.forEach((name, grades) -> {
356             double average = grades.stream()
357                 .mapToInt(Integer::intValue)
358                 .average()
359                 .orElse(0.0);
360             studentAverages.put(name, average);
361         });
362
363         System.out.println("Student averages:");
364         studentAverages.forEach((name, avg) ->
365             System.out.printf("  %-10s: %.2f\n", name, avg));
366
367         // Find top student
368         Map.Entry<String, Double> topStudent = studentAverages.entrySet
                ().stream()
369             .max(Map.Entry.comparingByValue())
370             .orElse(null);
371
372         if (topStudent != null) {
373             System.out.println("\nTop student: " + topStudent.getKey()
                    +
374                                " (Average: " + topStudent.getValue() + ")
                                    ");
375         }
376
377         // 3. Product inventory system
378         System.out.println("\n3. Product inventory system:");
379
380         class Product {
381             String id;
382             String name;
383             double price;
384
385             Product(String id, String name, double price) {
386                 this.id = id;
387                 this.name = name;
388                 this.price = price;
389             }
390
391             @Override
```

```
        public String toString() {
            return String.format("%s: %s - $%.2f", id, name, price);
        }
    }

    Map<String, Product> products = new HashMap<>();
    products.put("P001", new Product("P001", "Laptop", 999.99));
    products.put("P002", new Product("P002", "Phone", 699.99));
    products.put("P003", new Product("P003", "Tablet", 399.99));

    Map<String, Integer> inventory = new HashMap<>();
    inventory.put("P001", 10);
    inventory.put("P002", 25);
    inventory.put("P003", 15);
    inventory.put("P004", 5);  // Product exists in inventory but not in products

    System.out.println("Inventory status:");
    products.forEach((id, product) -> {
        int quantity = inventory.getOrDefault(id, 0);
        System.out.printf("  %-20s: %d units (Total value: $%.2f)\n",
                          product, quantity, product.price * quantity);
    });

    // Check low stock
    System.out.println("\nLow stock products (less than 20):");
    inventory.entrySet().stream()
        .filter(entry -> entry.getValue() < 20)
        .forEach(entry -> {
            Product product = products.get(entry.getKey());
            if (product != null) {
                System.out.println("  " + product.name + ": " +
                    entry.getValue() + " units");
            }
        });

    // 4. Cache implementation using LinkedHashMap
    System.out.println("\n4. LRU Cache implementation:");

    class LRUCache<K, V> extends LinkedHashMap<K, V> {
        private final int capacity;

        public LRUCache(int capacity) {
            super(capacity, 0.75f, true);  // Access-order (true), insertion-order (false)
            this.capacity = capacity;
        }

        @Override
        protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
            return size() > capacity;
        }
    }

```

```
        LRUCache<String, String> cache = new LRUCache<>(3);
        cache.put("1", "Data1");
        cache.put("2", "Data2");
        cache.put("3", "Data3");

        System.out.println("Cache after adding 3 items: " + cache.
            keySet());

        cache.get("1");  // Access item 1, making it most recently used
        cache.put("4", "Data4");  // This will remove least recently
            used (item 2)

        System.out.println("Cache after accessing '1' and adding '4': "
             + cache.keySet());

        // 5. Grouping data
        System.out.println("\n5. Grouping data by category:");

        class Item {
            String name;
            String category;
            double price;

            Item(String name, String category, double price) {
                this.name = name;
                this.category = category;
                this.price = price;
            }
        }

        List<Item> items = Arrays.asList(
            new Item("Laptop", "Electronics", 999.99),
            new Item("Phone", "Electronics", 699.99),
            new Item("Shirt", "Clothing", 29.99),
            new Item("Pants", "Clothing", 49.99),
            new Item("Book", "Stationery", 19.99)
        );

        // Group by category
        Map<String, List<Item>> itemsByCategory = items.stream()
            .collect(Collectors.groupingBy(item -> item.category));

        System.out.println("Items by category:");
        itemsByCategory.forEach((category, itemList) -> {
            System.out.println("  " + category + ":");
            itemList.forEach(item ->
                System.out.printf("    - %s: $%.2f\n", item.name, item.
                    price));
        });

        // Calculate total by category
        Map<String, Double> totalByCategory = items.stream()
            .collect(Collectors.groupingBy(
                item -> item.category,
                Collectors.summingDouble(item -> item.price)
            ));

        System.out.println("\nTotal by category:");
```

```
497             totalByCategory.forEach((category, total) ->
498                 System.out.printf("  %-12s: $%.2f\n", category, total));
499         }
500
501         // ==================== MAIN METHOD ====================
502         public static void main(String[] args) {
503             System.out.println("=== COMPREHENSIVE MAP IMPLEMENTATIONS
                    DEMONSTRATION ===\n");
504
505             System.out.println("=== MAP INTERFACE CHARACTERISTICS ===");
506             System.out.println("1. Key-value pairs");
507             System.out.println("2. No duplicate keys");
508             System.out.println("3. One null key allowed (in HashMap/
                    LinkedHashMap)");
509             System.out.println("4. Multiple null values allowed");
510
511             System.out.println("\n=== HASHMAP VS TREEMAP ===");
512             System.out.println("| Feature              | HashMap          |
                     TreeMap          |");
513             System.out.println("
                    |----------------------|------------------|------------------|
                    ");
514             System.out.println("| Ordering             | No order         |
                     Sorted by key    |");
515             System.out.println("| Null keys            | One allowed      |
                     Not allowed      |");
516             System.out.println("| Null values          | Multiple allowed |
                     Multiple allowed |");
517             System.out.println("| Performance          | O(1) average     |
                     O(log n)         |");
518             System.out.println("| Internal Structure   | Hash table       |
                     Red-Black Tree   |");
519             System.out.println("| Navigation methods   | No               |
                     Yes              |");
520
521             System.out.println("\n=== LINKEDHASHMAP ===");
522             System.out.println("LinkedHashMap: HashMap + maintains
                    insertion/access order");
523             System.out.println("  - Uses hash table with linked list");
524             System.out.println("  - Can maintain insertion order or access
                    order");
525             System.out.println("  - Useful for LRU cache implementation");
526
527             // Demonstrate HashMap
528             demonstrateHashMap();
529
530             // Demonstrate TreeMap
531             demonstrateTreeMap();
532
533             // Performance comparison
534             compareMapPerformance();
535
536             // Practical examples
537             practicalMapExamples();
538
539             System.out.println("\n=== WHEN TO USE WHICH ===");
540             System.out.println("Use HashMap when:");
541             System.out.println("  - Don't need ordering");
```

```
542             System.out.println("  - Need best performance");
543             System.out.println("  - Need to store null keys");
544
545             System.out.println("\nUse TreeMap when:");
546             System.out.println("  - Need sorted keys");
547             System.out.println("  - Need navigation methods");
548             System.out.println("  - Need range operations");
549
550             System.out.println("\nUse LinkedHashMap when:");
551             System.out.println("  - Need to maintain insertion order");
552             System.out.println("  - Implementing LRU cache");
553             System.out.println("  - Need predictable iteration order");
554
555             System.out.println("\n=== COMMON METHODS ===");
556             System.out.println("Common Map methods:");
557             System.out.println("  - put(), putAll()");
558             System.out.println("  - get(), getOrDefault()");
559             System.out.println("  - containsKey(), containsValue()");
560             System.out.println("  - remove()");
561             System.out.println("  - keySet(), values(), entrySet()");
562             System.out.println("  - size(), isEmpty(), clear()");
563             System.out.println("  - putIfAbsent(), compute(), merge()");
564             System.out.println("  - replace(), replaceAll()");
565
566             System.out.println("\nTreeMap-specific methods:");
567             System.out.println("  - firstKey(), lastKey()");
568             System.out.println("  - firstEntry(), lastEntry()");
569             System.out.println("  - ceilingKey(), floorKey()");
570             System.out.println("  - higherKey(), lowerKey()");
571             System.out.println("  - headMap(), tailMap(), subMap()");
572             System.out.println("  - pollFirstEntry(), pollLastEntry()");
573             System.out.println("  - descendingMap()");
574
575             System.out.println("\n=== EQUALS AND HASHCODE FOR KEYS ===");
576             System.out.println("For HashMap keys:");
577             System.out.println("1. If two keys are equal, they must have
                    same hashCode");
578             System.out.println("2. Keys with same hashCode may not be equal
                    ");
579             System.out.println("3. hashCode() should be consistent with
                    equals()");
580             System.out.println("4. Mutable objects should not be used as
                    HashMap keys");
581
582             System.out.println("\n=== THREAD SAFETY ===");
583             System.out.println("HashMap and TreeMap are NOT thread-safe");
584             System.out.println("For thread-safe alternatives:");
585             System.out.println("  - Use Collections.synchronizedMap()");
586             System.out.println("  - Use ConcurrentHashMap for concurrent
                    access");
587             System.out.println("  - Use ConcurrentSkipListMap for
                    concurrent sorted map");
588
589             // Thread safety example
590             System.out.println("\n=== SYNCHRONIZED MAP EXAMPLE ===");
591             Map<String, Integer> syncMap = Collections.synchronizedMap(new
                    HashMap<>());
592
```

```
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                syncMap.put("Thread1-" + i, i);
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                syncMap.put("Thread2-" + i, i);
            }
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Synchronized map size: " + syncMap.size());

        // ConcurrentHashMap example
        System.out.println("\n=== CONCURRENTHASHMAP EXAMPLE ===");
        System.out.println("ConcurrentHashMap provides better
            concurrency:");
        System.out.println("  - Thread-safe without synchronization");
        System.out.println("  - Better performance in concurrent
            scenarios");
        System.out.println("  - Does not allow null keys or values");

        // LinkedHashMap example
        System.out.println("\n=== LINKEDHASHMAP EXAMPLE ===");
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
        linkedHashMap.put("First", 1);
        linkedHashMap.put("Second", 2);
        linkedHashMap.put("Third", 3);
        linkedHashMap.put("First", 10);  // Update existing

        System.out.println("LinkedHashMap (maintains insertion order):
            " + linkedHashMap);
    }

    // Student class for TreeMap demonstration
    static class Student {
        String name;
        int age;

        Student(String name, int age) {
            this.name = name;
            this.age = age;
        }

        @Override
        public String toString() {
            return name + " (" + age + ")";
```

```
        }
    }
}
```

Listing 7: Complete Map Implementations (HashMap and TreeMap)

MapDemo Program Output

```
=== COMPREHENSIVE MAP IMPLEMENTATIONS DEMONSTRATION ===

=== MAP INTERFACE CHARACTERISTICS ===
1. Key-value pairs
2. No duplicate keys
3. One null key allowed (in HashMap/LinkedHashMap)
4. Multiple null values allowed

=== HASHMAP VS TREEMAP ===
| Feature              | HashMap          | TreeMap          |
|----------------------|------------------|------------------|
| Ordering             | No order         | Sorted by key    |
| Null keys            | One allowed      | Not allowed      |
| Null values          | Multiple allowed | Multiple allowed |
| Performance          | O(1) average     | O(log n)         |
| Internal Structure   | Hash table       | Red-Black Tree   |
| Navigation methods   | No               | Yes              |

=== LINKEDHASHMAP ===
LinkedHashMap: HashMap + maintains insertion/access order
  - Uses hash table with linked list
  - Can maintain insertion order or access order
  - Useful for LRU cache implementation

=== HASHMAP DEMONSTRATION ===

1. Adding key-value pairs:
HashMap: {null=0, Apple=15, Cherry=30, Date=null, Banana=20}
Size: 5
Is empty? false

2. Accessing values:
Value for 'Apple': 15
Value for 'Mango': null
Value for null key: 0
Value or default for 'Mango': -1

3. Checking keys and values:
Contains key 'Banana': true
Contains key 'Grapes': false
Contains value 20: true
```

```
Contains value 100: false

4. Updating values:
Original value for 'Banana': 20
After put: 25
After putIfAbsent: {null=0, Apple=15, Mango=40, Cherry=30, Date=null, Banana=25}
After compute (Apple + 10): 25
After compute methods: {null=0, Apple=25, Mango=40, Cherry=30, Orange=50, Date=null
After merge (Apple + 5): 30

5. Removing entries:
Before removal: {null=0, Apple=30, Mango=40, Cherry=30, Orange=50, Date=null, Banan
After removing 'Banana': {null=0, Apple=30, Mango=40, Cherry=30, Orange=50, Date=nu
After removing 'Apple' if value=30: {null=0, Mango=40, Cherry=30, Orange=50, Date=n

6. Iterating through HashMap:
a) Using keySet():
  Key: null, Value: 0
  Key: Mango, Value: 40
  Key: Cherry, Value: 30
  Key: Orange, Value: 50
  Key: Date, Value: null

b) Using values():
  Value: 0
  Value: 40
  Value: 30
  Value: 50
  Value: null

c) Using entrySet():
  Key: null, Value: 0
  Key: Mango, Value: 40
  Key: Cherry, Value: 30
  Key: Orange, Value: 50
  Key: Date, Value: null

d) Using forEach (Java 8+):
  Key: null, Value: 0
  Key: Mango, Value: 40
  Key: Cherry, Value: 30
  Key: Orange, Value: 50
  Key: Date, Value: null

7. Bulk operations:
After putAll: {null=0, Grapes=60, Mango=40, Cherry=30, Pineapple=70, Orange=50, Dat
```

```
8. Replacing values:
After replace 'Cherry': 100
After conditional replace 'Cherry': 150
After replaceAll (+10): {null=10, Grapes=70, Mango=50, Cherry=160, Pineapple=80, Or

9. Clearing map:
After clear - Size: 0
Is empty? true

=== TREEMAP DEMONSTRATION ===

1. Adding entries (sorted by key):
TreeMap: {Alice=32, Bob=28, Charlie=25}
Size: 3

2. Navigation methods:
First key: Alice
Last key: Charlie
First entry: Alice=32
Last entry: Charlie=25

Ceiling key (>= 'Brain'): Charlie
Floor key (<= 'Brain'): Bob

Higher key (> 'Bob'): Charlie
Lower key (< 'Bob'): Alice

Ceiling entry (>= 'Brain'): Charlie=25
Floor entry (<= 'Brain'): Bob=28

3. Submap operations:
HeadMap (< 'Charlie'): {Alice=32, Bob=28}
HeadMap (<= 'Charlie'): {Alice=32, Bob=28, Charlie=25}
TailMap (>= 'Bob'): {Bob=28, Charlie=25}
TailMap (> 'Bob'): {Charlie=25}
SubMap ('Alice' <= x < 'Charlie'): {Alice=32, Bob=28}
SubMap ('Alice' <= x <= 'Charlie'): {Alice=32, Bob=28, Charlie=25}

4. Polling entries:
TreeMap before polling: {Alice=32, Bob=28, Charlie=25}
Poll first entry: Alice=32
Poll last entry: Charlie=25
After polling: {Bob=28}

5. Descending map:
Original map: {Bob=28}
Descending map: {Bob=28}
```

```
6. Using custom comparator:
Case-insensitive TreeMap: {apple=20, Banana=30}
Contains key 'Apple': true
Contains key 'apple': true

7. TreeMap with complex keys:
Inventory sorted by product price:
  Tablet ($399.99): 15 units
  Phone ($699.99): 25 units
  Laptop ($999.99): 10 units

Most expensive product under $500: Tablet (15 units)

=== HASHMAP VS TREEMAP PERFORMANCE ===

1. Put performance:
Putting 100000 random entries:
  HashMap:  25.67 ms
  TreeMap:  78.45 ms

2. Get performance:
Getting 1000 random entries:
  HashMap:  0.23 ms
  TreeMap:  0.89 ms

3. Iteration performance:
Iterating through all values:
  HashMap:  3.45 ms (count: 100000)
  TreeMap:  4.12 ms (count: 100000)

=== PRACTICAL MAP EXAMPLES ===

1. Word frequency counter:
Word frequencies:
  the         : 3
  quick       : 1
  brown       : 1
  fox         : 1
  jumps       : 1
  over        : 1
  lazy        : 2
  dog         : 2
  was         : 1
  not         : 1

Sorted by frequency (descending):
```

```
  the         : 3
  lazy        : 2
  dog         : 2
  quick       : 1
  brown       : 1
  fox         : 1
  jumps       : 1
  over        : 1
  was         : 1
  not         : 1

2. Student grades system:
Student averages:
  Alice       : 84.33
  Bob         : 91.67
  Charlie     : 79.00

Top student: Bob (Average: 91.66666666666667)

3. Product inventory system:
Inventory status:
  P001: Laptop - $999.99: 10 units (Total value: $9999.90)
  P002: Phone - $699.99: 25 units (Total value: $17499.75)
  P003: Tablet - $399.99: 15 units (Total value: $5999.85)

Low stock products (less than 20):
  Laptop: 10 units
  Tablet: 15 units

4. LRU Cache implementation:
Cache after adding 3 items: [1, 2, 3]
Cache after accessing '1' and adding '4': [3, 1, 4]

5. Grouping data by category:
Items by category:
  Stationery:
    - Book: $19.99
  Electronics:
    - Laptop: $999.99
    - Phone: $699.99
  Clothing:
    - Shirt: $29.99
    - Pants: $49.99

Total by category:
  Stationery  : $19.99
  Electronics : $1699.98
```

```
  Clothing    : $79.98

=== WHEN TO USE WHICH ===
Use HashMap when:
  - Don't need ordering
  - Need best performance
  - Need to store null keys

Use TreeMap when:
  - Need sorted keys
  - Need navigation methods
  - Need range operations

Use LinkedHashMap when:
  - Need to maintain insertion order
  - Implementing LRU cache
  - Need predictable iteration order

=== COMMON METHODS ===
Common Map methods:
  - put(), putAll()
  - get(), getOrDefault()
  - containsKey(), containsValue()
  - remove()
  - keySet(), values(), entrySet()
  - size(), isEmpty(), clear()
  - putIfAbsent(), compute(), merge()
  - replace(), replaceAll()

TreeMap-specific methods:
  - firstKey(), lastKey()
  - firstEntry(), lastEntry()
  - ceilingKey(), floorKey()
  - higherKey(), lowerKey()
  - headMap(), tailMap(), subMap()
  - pollFirstEntry(), pollLastEntry()
  - descendingMap()

=== EQUALS AND HASHCODE FOR KEYS ===
For HashMap keys:
1. If two keys are equal, they must have same hashCode
2. Keys with same hashCode may not be equal
3. hashCode() should be consistent with equals()
4. Mutable objects should not be used as HashMap keys

=== THREAD SAFETY ===
HashMap and TreeMap are NOT thread-safe
```

```
For thread-safe alternatives:
  - Use Collections.synchronizedMap()
  - Use ConcurrentHashMap for concurrent access
  - Use ConcurrentSkipListMap for concurrent sorted map

=== SYNCHRONIZED MAP EXAMPLE ===
Synchronized map size: 10

=== CONCURRENTHASHMAP EXAMPLE ===
ConcurrentHashMap provides better concurrency:
  - Thread-safe without synchronization
  - Better performance in concurrent scenarios
  - Does not allow null keys or values

=== LINKEDHASHMAP EXAMPLE ===
LinkedHashMap (maintains insertion order): {First=10, Second=2, Third=3}
```

5 Generics

```java
import java.util.*;
import java.lang.reflect.*;

public class GenericsDemo {

    // ==================== GENERIC CLASSES ====================

    // 1. Simple generic class with single type parameter
    static class Box<T> {
        private T content;

        public Box() {
            this.content = null;
        }

        public Box(T content) {
            this.content = content;
        }

        public void setContent(T content) {
            this.content = content;
        }

        public T getContent() {
            return content;
        }

        public boolean isEmpty() {
            return content == null;
        }

        @Override
        public String toString() {
```

```
34              return "Box[" + (content != null ? content.toString() : "
                    empty") + "]";
35          }
36      }
37
38      // 2. Generic class with multiple type parameters
39      static class Pair<K, V> {
40          private K key;
41          private V value;
42
43          public Pair(K key, V value) {
44              this.key = key;
45              this.value = value;
46          }
47
48          public K getKey() {
49              return key;
50          }
51
52          public V getValue() {
53              return value;
54          }
55
56          public void setKey(K key) {
57              this.key = key;
58          }
59
60          public void setValue(V value) {
61              this.value = value;
62          }
63
64          @Override
65          public String toString() {
66              return "Pair{" + key + "=" + value + "}";
67          }
68      }
69
70      // 3. Generic class with bounded type parameter
71      static class NumberContainer<T extends Number> {
72          private T number;
73
74          public NumberContainer(T number) {
75              this.number = number;
76          }
77
78          public double getSquare() {
79              return number.doubleValue() * number.doubleValue();
80          }
81
82          public boolean isInteger() {
83              return number instanceof Integer;
84          }
85
86          public T getNumber() {
87              return number;
88          }
89
90          // Generic method within generic class
```

```
91          public <U extends Number> double sum(U other) {
92              return this.number.doubleValue() + other.doubleValue();
93          }
94      }
95
96      // 4. Generic class with interface bounds
97      static class SortedContainer<T extends Comparable<T>> {
98          private T[] elements;
99
100         @SafeVarargs
101         public SortedContainer(T... elements) {
102             this.elements = elements;
103             Arrays.sort(this.elements);
104         }
105
106         public T getMin() {
107             return elements.length > 0 ? elements[0] : null;
108         }
109
110         public T getMax() {
111             return elements.length > 0 ? elements[elements.length - 1]
                    : null;
112         }
113
114         public T[] getSorted() {
115             return elements.clone();
116         }
117     }
118
119     // 5. Generic class with recursive type bound
120     static class ComparablePair<T extends Comparable<T>>
121             implements Comparable<ComparablePair<T>> {
122
123         private T first;
124         private T second;
125
126         public ComparablePair(T first, T second) {
127             this.first = first;
128             this.second = second;
129         }
130
131         @Override
132         public int compareTo(ComparablePair<T> other) {
133             int firstCompare = this.first.compareTo(other.first);
134             if (firstCompare != 0) {
135                 return firstCompare;
136             }
137             return this.second.compareTo(other.second);
138         }
139
140         @Override
141         public String toString() {
142             return "(" + first + ", " + second + ")";
143         }
144     }
145
146     // ==================== GENERIC METHODS ====================
147
```

```
148     // 1. Simple generic method
149     public static <T> T getFirstElement(List<T> list) {
150         if (list == null || list.isEmpty()) {
151             return null;
152         }
153         return list.get(0);
154     }
155 
156     // 2. Generic method with multiple type parameters
157     public static <K, V> V getValue(Map<K, V> map, K key) {
158         return map.get(key);
159     }
160 
161     // 3. Generic method with bounded type parameter
162     public static <T extends Number> double sumNumbers(T a, T b) {
163         return a.doubleValue() + b.doubleValue();
164     }
165 
166     // 4. Generic method with wildcards
167     public static double sumList(List<? extends Number> numbers) {
168         double sum = 0;
169         for (Number num : numbers) {
170             sum += num.doubleValue();
171         }
172         return sum;
173     }
174 
175     // 5. Generic method with upper and lower bounds
176     public static <T> void copy(List<? super T> dest, List<? extends T>
        src) {
177         dest.clear();
178         dest.addAll(src);
179     }
180 
181     // 6. Generic method returning generic type
182     public static <T> List<T> createList(T... elements) {
183         List<T> list = new ArrayList<>();
184         Collections.addAll(list, elements);
185         return list;
186     }
187 
188     // ==================== TYPE SAFETY DEMONSTRATION
        ====================
189 
190     public static void demonstrateTypeSafety() {
191         System.out.println("\n=== TYPE SAFETY DEMONSTRATION ===");
192 
193         // 1. Without generics (raw types) - Compiles but runtime error
            possible
194         System.out.println("\n1. Without generics (raw types):");
195         List rawList = new ArrayList();
196         rawList.add("String");
197         rawList.add(123);  // Adding Integer to list meant for Strings
198         rawList.add(new Date());
199 
200         // Need explicit casting - ClassCastException at runtime
201         try {
```

```
202             String str = (String) rawList.get(1);  // Will throw
                    ClassCastException
203         } catch (ClassCastException e) {
204             System.out.println("ClassCastException: " + e.getMessage())
                    ;
205         }
206 
207         // 2. With generics - Compile-time type safety
208         System.out.println("\n2. With generics (type safety):");
209         List<String> stringList = new ArrayList<>();
210         stringList.add("Hello");
211         stringList.add("World");
212         // stringList.add(123);  // Compilation error - type safety!
213 
214         String first = stringList.get(0);  // No casting needed
215         System.out.println("First element: " + first);
216 
217         // 3. Generic collections type safety
218         System.out.println("\n3. Generic collections:");
219         Map<String, Integer> wordCount = new HashMap<>();
220         wordCount.put("apple", 5);
221         wordCount.put("banana", 3);
222         // wordCount.put(123, "test");  // Compilation error
223 
224         Integer count = wordCount.get("apple");  // No casting
225         System.out.println("Apple count: " + count);
226 
227         // 4. Generic class type safety
228         System.out.println("\n4. Generic class type safety:");
229         Box<String> stringBox = new Box<>("Hello Generics");
230         String content = stringBox.getContent();  // No casting
231         System.out.println("Box content: " + content);
232 
233         Box<Integer> intBox = new Box<>(42);
234         Integer number = intBox.getContent();  // No casting
235         System.out.println("Box number: " + number);
236 
237         // 5. Type inference
238         System.out.println("\n5. Type inference (diamond operator <>):"
                );
239         List<String> inferredList = new ArrayList<>();  // Type
                inferred
240         Map<String, List<Integer>> complexMap = new HashMap<>();  //
                Nested generics
241 
242         // Method type inference
243         String firstElement = getFirstElement(Arrays.asList("A", "B", "
                C"));
244         System.out.println("First element: " + firstElement);
245     }
246 
247     // ==================== WILDCARDS DEMONSTRATION
            ====================
248 
249     public static void demonstrateWildcards() {
250         System.out.println("\n=== WILDCARDS DEMONSTRATION ===");
251 
252         // Upper bounded wildcard (? extends T)
```

```
253         System.out.println("\n1. Upper bounded wildcard (? extends T):");
254         List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);
255         List<Double> doubles = Arrays.asList(1.1, 2.2, 3.3);
256 
257         System.out.println("Sum of integers: " + sumList(integers));
258         System.out.println("Sum of doubles: " + sumList(doubles));
259 
260         // Lower bounded wildcard (? super T)
261         System.out.println("\n2. Lower bounded wildcard (? super T):");
262         List<Number> numbers = new ArrayList<>();
263         List<Integer> srcIntegers = Arrays.asList(1, 2, 3);
264 
265         copy(numbers, srcIntegers);  // Can copy Integer to Number list
266         System.out.println("Copied numbers: " + numbers);
267 
268         // Unbounded wildcard (?)
269         System.out.println("\n3. Unbounded wildcard (?):");
270         printList(Arrays.asList("A", "B", "C"));
271         printList(Arrays.asList(1, 2, 3));
272 
273         // Wildcard in generic class
274         System.out.println("\n4. Wildcard in generic class:");
275         Box<?> unknownBox = new Box<>("Secret");
276         // unknownBox.setContent("New");  // Compilation error - cannot
                set
277         Object obj = unknownBox.getContent();  // Can only get as
                Object
278         System.out.println("Unknown box: " + obj);
279 
280         // Wildcard capture
281         System.out.println("\n5. Wildcard capture:");
282         List<?> wildList = Arrays.asList("A", "B", "C");
283         System.out.println("Wild list size: " + wildList.size());
284         // wildList.add("D");  // Compilation error
285     }
286 
287     // Helper method for wildcard demonstration
288     public static void printList(List<?> list) {
289         for (Object elem : list) {
290             System.out.print(elem + " ");
291         }
292         System.out.println();
293     }
294 
295     // ==================== GENERICS WITH COLLECTIONS
            ====================
296 
297     public static void demonstrateGenericsWithCollections() {
298         System.out.println("\n=== GENERICS WITH COLLECTIONS ===");
299 
300         // 1. Type-safe collections
301         System.out.println("\n1. Type-safe collections:");
302 
303         // List with generics
304         List<String> cities = new ArrayList<>();
305         cities.add("New York");
306         cities.add("London");
```

```
307         cities.add("Tokyo");
308         // cities.add(123);  // Compilation error
309
310         // Set with generics
311         Set<Integer> primeNumbers = new HashSet<>();
312         primeNumbers.add(2);
313         primeNumbers.add(3);
314         primeNumbers.add(5);
315         primeNumbers.add(7);
316         // primeNumbers.add("not a number");  // Compilation error
317
318         // Map with generics
319         Map<String, Double> productPrices = new HashMap<>();
320         productPrices.put("Laptop", 999.99);
321         productPrices.put("Phone", 699.99);
322         // productPrices.put(123, 456.78);  // Compilation error
323
324         // 2. Generic methods with collections
325         System.out.println("\n2. Generic methods with collections:");
326
327         List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
328         Integer firstNumber = getFirstElement(numbers);
329         System.out.println("First number: " + firstNumber);
330
331         // 3. Custom generic collections
332         System.out.println("\n3. Custom generic collections:");
333
334         SortedContainer<String> stringContainer = new SortedContainer
                <>(
335              "Zebra", "Apple", "Banana", "Cherry"
336         );
337         System.out.println("Sorted strings min: " + stringContainer.
                getMin());
338         System.out.println("Sorted strings max: " + stringContainer.
                getMax());
339
340         SortedContainer<Integer> intContainer = new SortedContainer
                <>(5, 1, 8, 3);
341         System.out.println("Sorted integers min: " + intContainer.
                getMin());
342         System.out.println("Sorted integers max: " + intContainer.
                getMax());
343
344         // 4. Generic algorithms
345         System.out.println("\n4. Generic algorithms:");
346
347         // Reverse generic list
348         List<String> reversed = reverseList(cities);
349         System.out.println("Original: " + cities);
350         System.out.println("Reversed: " + reversed);
351
352         // Find maximum using comparator
353         String maxCity = findMax(cities, Comparator.naturalOrder());
354         System.out.println("Maximum city: " + maxCity);
355
356         // 5. Nested generics
357         System.out.println("\n5. Nested generics:");
358
```

```
359         Map<String, List<Integer>> studentScores = new HashMap<>();
360         studentScores.put("Alice", Arrays.asList(85, 90, 78));
361         studentScores.put("Bob", Arrays.asList(92, 88, 95));
362 
363         for (Map.Entry<String, List<Integer>> entry : studentScores.
                entrySet()) {
364             System.out.println(entry.getKey() + ": " + entry.getValue()
                    );
365         }
366     }
367 
368     // Generic algorithm: Reverse list
369     public static <T> List<T> reverseList(List<T> list) {
370         List<T> reversed = new ArrayList<>(list);
371         Collections.reverse(reversed);
372         return reversed;
373     }
374 
375     // Generic algorithm: Find maximum
376     public static <T> T findMax(List<T> list, Comparator<? super T>
            comparator) {
377         if (list == null || list.isEmpty()) {
378             return null;
379         }
380         return Collections.max(list, comparator);
381     }
382 
383     // ==================== GENERICS INHERITANCE AND SUBCLASSING
            ====================
384 
385     public static void demonstrateGenericsInheritance() {
386         System.out.println("\n=== GENERICS INHERITANCE ===");
387 
388         // 1. Generic class inheritance
389         System.out.println("\n1. Generic class inheritance:");
390 
391         class StringBox extends Box<String> {
392             public StringBox(String content) {
393                 super(content);
394             }
395 
396             public String toUpperCase() {
397                 return getContent() != null ? getContent().toUpperCase
                        () : null;
398             }
399         }
400 
401         StringBox stringBox = new StringBox("hello generics");
402         System.out.println("Original: " + stringBox.getContent());
403         System.out.println("Uppercase: " + stringBox.toUpperCase());
404 
405         // 2. Generic subclass with additional type parameter
406         System.out.println("\n2. Generic subclass with additional type
                parameter:");
407 
408         class NumberBox<T extends Number> extends Box<T> {
409             public NumberBox(T content) {
410                 super(content);
```

```
411             }
412
413             public double getSquare() {
414                 T number = getContent();
415                 return number != null ? number.doubleValue() * number.
                        doubleValue() : 0;
416             }
417         }
418
419         NumberBox<Integer> intBox = new NumberBox<>(5);
420         System.out.println("Number: " + intBox.getContent());
421         System.out.println("Square: " + intBox.getSquare());
422
423         // 3. Inheritance with wildcards
424         System.out.println("\n3. Inheritance with wildcards:");
425
426         List<Integer> integers = Arrays.asList(1, 2, 3);
427         List<? extends Number> numbers = integers;  // Covariance
428         // numbers.add(4);  // Compilation error - cannot add
429
430         List<Number> numberList = new ArrayList<>();
431         List<? super Integer> superIntegers = numberList;  //
                Contravariance
432         superIntegers.add(42);  // Can add Integer
433         // Integer value = superIntegers.get(0);  // Compilation error
434
435         // 4. Generic interface implementation
436         System.out.println("\n4. Generic interface implementation:");
437
438         interface Repository<T> {
439             void save(T item);
440             T findById(String id);
441         }
442
443         class UserRepository implements Repository<String> {
444             private Map<String, String> storage = new HashMap<>();
445
446             @Override
447             public void save(String item) {
448                 storage.put(item, item);
449             }
450
451             @Override
452             public String findById(String id) {
453                 return storage.get(id);
454             }
455         }
456
457         UserRepository repo = new UserRepository();
458         repo.save("user123");
459         System.out.println("Found: " + repo.findById("user123"));
460     }
461
462     // ==================== GENERICS AND REFLECTION
            ====================
463
464     public static void demonstrateGenericsReflection() {
465         System.out.println("\n=== GENERICS AND REFLECTION ===");
```

```
466
467         // Type erasure demonstration
468         System.out.println("\n1. Type erasure:");
469
470         List<String> stringList = new ArrayList<>();
471         List<Integer> intList = new ArrayList<>();
472
473         System.out.println("stringList class: " + stringList.getClass()
                );
474         System.out.println("intList class: " + intList.getClass());
475         System.out.println("Same class? " + (stringList.getClass() ==
                intList.getClass()));
476
477         // 2. Getting generic type information
478         System.out.println("\n2. Generic type information:");
479
480         Box<String> stringBox = new Box<>("Test");
481         Box<Integer> intBox = new Box<>(123);
482
483         System.out.println("stringBox type: " + stringBox.getClass().
                getTypeName());
484         System.out.println("intBox type: " + intBox.getClass().
                getTypeName());
485
486         // 3. ParameterizedType example
487         System.out.println("\n3. ParameterizedType in fields:");
488
489         class Container {
490             List<String> strings;
491             Map<String, Integer> map;
492         }
493
494         Container container = new Container();
495         Field[] fields = Container.class.getDeclaredFields();
496
497         for (Field field : fields) {
498             System.out.println("Field: " + field.getName());
499             System.out.println("  Type: " + field.getType());
500             System.out.println("  Generic type: " + field.
                    getGenericType());
501
502             if (field.getGenericType() instanceof ParameterizedType) {
503                 ParameterizedType pType = (ParameterizedType) field.
                        getGenericType();
504                 System.out.println("  Raw type: " + pType.getRawType())
                        ;
505                 System.out.println("  Type arguments: " +
506                                    Arrays.toString(pType.
                                        getActualTypeArguments()));
507             }
508         }
509     }
510
511     // ==================== PRACTICAL GENERICS EXAMPLES
            ====================
512
513     public static void practicalGenericsExamples() {
514         System.out.println("\n=== PRACTICAL GENERICS EXAMPLES ===");
```

```
515
516         // 1. Data repository pattern
517         System.out.println("\n1. Data repository pattern:");
518
519         interface DataRepository<T, ID> {
520             T save(T entity);
521             Optional<T> findById(ID id);
522             List<T> findAll();
523             void deleteById(ID id);
524         }
525
526         class InMemoryRepository<T, ID> implements DataRepository<T, ID
                > {
527             private Map<ID, T> storage = new HashMap<>();
528             private Function<T, ID> idExtractor;
529
530             public InMemoryRepository(Function<T, ID> idExtractor) {
531                 this.idExtractor = idExtractor;
532             }
533
534             @Override
535             public T save(T entity) {
536                 ID id = idExtractor.apply(entity);
537                 storage.put(id, entity);
538                 return entity;
539             }
540
541             @Override
542             public Optional<T> findById(ID id) {
543                 return Optional.ofNullable(storage.get(id));
544             }
545
546             @Override
547             public List<T> findAll() {
548                 return new ArrayList<>(storage.values());
549             }
550
551             @Override
552             public void deleteById(ID id) {
553                 storage.remove(id);
554             }
555         }
556
557         class User {
558             String id;
559             String name;
560
561             User(String id, String name) {
562                 this.id = id;
563                 this.name = name;
564             }
565
566             @Override
567             public String toString() {
568                 return "User{id='" + id + "', name='" + name + "'}";
569             }
570         }
571
```

```
        InMemoryRepository<User, String> userRepo =
            new InMemoryRepository<>(user -> user.id);

        userRepo.save(new User("1", "Alice"));
        userRepo.save(new User("2", "Bob"));

        System.out.println("All users: " + userRepo.findAll());
        System.out.println("Find by ID '1': " + userRepo.findById("1"))
            ;

        // 2. Builder pattern with generics
        System.out.println("\n2. Builder pattern with generics:");

        class QueryBuilder<T> {
            private Class<T> entityClass;
            private List<String> conditions = new ArrayList<>();

            public QueryBuilder(Class<T> entityClass) {
                this.entityClass = entityClass;
            }

            public QueryBuilder<T> where(String condition) {
                conditions.add(condition);
                return this;
            }

            public String build() {
                String query = "SELECT * FROM " + entityClass.
                    getSimpleName();
                if (!conditions.isEmpty()) {
                    query += " WHERE " + String.join(" AND ",
                        conditions);
                }
                return query;
            }
        }

        QueryBuilder<User> userQuery = new QueryBuilder<>(User.class)
            .where("active = true")
            .where("age > 18");

        System.out.println("Generated query: " + userQuery.build());

        // 3. Event system with generics
        System.out.println("\n3. Event system with generics:");

        interface EventListener<T> {
            void onEvent(T event);
        }

        class EventBus {
            private Map<Class<?>, List<EventListener<?>>> listeners =
                new HashMap<>();

            public <T> void register(Class<T> eventType, EventListener<
                T> listener) {
                listeners.computeIfAbsent(eventType, k -> new ArrayList
                    <>())
```

```
                    .add(listener);
        }

        @SuppressWarnings("unchecked")
        public <T> void publish(T event) {
            List<EventListener<?>> eventListeners = listeners.get(
                event.getClass());
            if (eventListeners != null) {
                for (EventListener<?> listener : eventListeners) {
                    ((EventListener<T>) listener).onEvent(event);
                }
            }
        }
    }

    class UserEvent {
        String message;

        UserEvent(String message) {
            this.message = message;
        }
    }

    EventBus bus = new EventBus();
    bus.register(UserEvent.class, event ->
        System.out.println("Event received: " + event.message));

    bus.publish(new UserEvent("User logged in"));

    // 4. Calculator with generics
    System.out.println("\n4. Calculator with generics:");

    class Calculator<T extends Number> {
        public T add(T a, T b) {
            if (a instanceof Integer) {
                return (T) Integer.valueOf(a.intValue() + b.
                    intValue());
            } else if (a instanceof Double) {
                return (T) Double.valueOf(a.doubleValue() + b.
                    doubleValue());
            }
            throw new IllegalArgumentException("Unsupported type");
        }

        public T multiply(T a, T b) {
            if (a instanceof Integer) {
                return (T) Integer.valueOf(a.intValue() * b.
                    intValue());
            } else if (a instanceof Double) {
                return (T) Double.valueOf(a.doubleValue() * b.
                    doubleValue());
            }
            throw new IllegalArgumentException("Unsupported type");
        }
    }

    Calculator<Integer> intCalc = new Calculator<>();
    System.out.println("5 + 3 = " + intCalc.add(5, 3));
```

```
677         System.out.println("5 * 3 = " + intCalc.multiply(5, 3));
678
679         Calculator<Double> doubleCalc = new Calculator<>();
680         System.out.println("5.5 + 3.2 = " + doubleCalc.add(5.5, 3.2));
681     }
682
683     // ==================== MAIN METHOD ====================
684
685     public static void main(String[] args) {
686         System.out.println("=== COMPREHENSIVE GENERICS DEMONSTRATION
                ===\n");
687
688         System.out.println("=== GENERICS BENEFITS ===");
689         System.out.println("1. Type Safety: Compile-time type checking"
                );
690         System.out.println("2. Code Reuse: Write once, use with
                multiple types");
691         System.out.println("3. No Casting: Eliminate explicit type
                casting");
692         System.out.println("4. Algorithm Abstraction: Write generic
                algorithms");
693
694         System.out.println("\n=== TYPE PARAMETERS NAMING CONVENTIONS
                ===");
695         System.out.println("E - Element (used extensively by Java
                Collections Framework)");
696         System.out.println("K - Key");
697         System.out.println("N - Number");
698         System.out.println("T - Type");
699         System.out.println("V - Value");
700         System.out.println("S,U,V etc. - 2nd, 3rd, 4th types");
701
702         // Demonstrate type safety
703         demonstrateTypeSafety();
704
705         // Demonstrate wildcards
706         demonstrateWildcards();
707
708         // Demonstrate generics with collections
709         demonstrateGenericsWithCollections();
710
711         // Demonstrate generics inheritance
712         demonstrateGenericsInheritance();
713
714         // Demonstrate generics and reflection
715         demonstrateGenericsReflection();
716
717         // Practical examples
718         practicalGenericsExamples();
719
720         System.out.println("\n=== GENERICS RESTRICTIONS ===");
721         System.out.println("1. Cannot instantiate generic types with
                primitive types");
722         System.out.println("2. Cannot create instances of type
                parameters");
723         System.out.println("3. Cannot declare static fields of type
                parameters");
```

```
724         System.out.println("4. Cannot use casts or instanceof with
                parameterized types");
725         System.out.println("5. Cannot create arrays of parameterized
                types");
726         System.out.println("6. Cannot create, catch, or throw generic
                exceptions");
727         System.out.println("7. Cannot overload methods with same
                erasure");
728
729         System.out.println("\n=== TYPE ERASURE ===");
730         System.out.println("Generics are implemented using type erasure
                :");
731         System.out.println("  - Type parameters removed at compile time
                ");
732         System.out.println("  - Replaced with bounds (Object if
                unbounded)");
733         System.out.println("  - Type casts inserted where necessary");
734         System.out.println("  - Bridge methods generated for
                polymorphism");
735
736         System.out.println("\n=== BEST PRACTICES ===");
737         System.out.println("1. Use generics for type safety in
                collections");
738         System.out.println("2. Prefer generic methods over raw types");
739         System.out.println("3. Use bounded wildcards for maximum
                flexibility");
740         System.out.println("4. Avoid using raw types in new code");
741         System.out.println("5. Document generic type parameters");
742         System.out.println("6. Use @SuppressWarnings judiciously");
743
744         System.out.println("\n=== COMMON PITFALLS ===");
745         System.out.println("1. Using raw types (lose type safety)");
746         System.out.println("2. Ignoring unchecked warnings");
747         System.out.println("3. Misusing wildcards");
748         System.out.println("4. Trying to create generic arrays");
749         System.out.println("5. Forgetting about type erasure");
750
751         System.out.println("\n=== REAL-WORLD USAGE ===");
752         System.out.println("1. Collections Framework (ArrayList<T>,
                HashMap<K,V>)");
753         System.out.println("2. Streams API (Stream<T>)");
754         System.out.println("3. Optional class (Optional<T>)");
755         System.out.println("4. Functional interfaces (Function<T,R>)");
756         System.out.println("5. Repository pattern in data access layers
                ");
757         System.out.println("6. Builder pattern implementations");
758
759         // Final comprehensive example
760         System.out.println("\n=== COMPREHENSIVE EXAMPLE: GENERIC
                UTILITY CLASS ===");
761
762         class CollectionUtils {
763             public static <T> List<T> filter(List<T> list, Predicate<T>
                    predicate) {
764                 List<T> result = new ArrayList<>();
765                 for (T item : list) {
766                     if (predicate.test(item)) {
767                         result.add(item);
```

```
                    }
                }
                return result;
            }

            public static <T, R> List<R> map(List<T> list, Function<T,
                R> mapper) {
                List<R> result = new ArrayList<>();
                for (T item : list) {
                    result.add(mapper.apply(item));
                }
                return result;
            }

            public static <T> Optional<T> findFirst(List<T> list,
                Predicate<T> predicate) {
                for (T item : list) {
                    if (predicate.test(item)) {
                        return Optional.of(item);
                    }
                }
                return Optional.empty();
            }
        }

        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8,
            9, 10);

        // Filter even numbers
        List<Integer> evens = CollectionUtils.filter(numbers, n -> n %
            2 == 0);
        System.out.println("Even numbers: " + evens);

        // Map to squares
        List<Integer> squares = CollectionUtils.map(numbers, n -> n * n
            );
        System.out.println("Squares: " + squares);

        // Find first number > 5
        Optional<Integer> firstLarge = CollectionUtils.findFirst(
            numbers, n -> n > 5);
        System.out.println("First number > 5: " + firstLarge.orElse(-1)
            );
    }
}
```

Listing 8: Complete Generics Implementation

GenericsDemo Program Output

```
=== COMPREHENSIVE GENERICS DEMONSTRATION ===

=== GENERICS BENEFITS ===
1. Type Safety: Compile-time type checking
2. Code Reuse: Write once, use with multiple types
3. No Casting: Eliminate explicit type casting
```

```
4. Algorithm Abstraction: Write generic algorithms

=== TYPE PARAMETERS NAMING CONVENTIONS ===
E - Element (used extensively by Java Collections Framework)
K - Key
N - Number
T - Type
V - Value
S,U,V etc. - 2nd, 3rd, 4th types

=== TYPE SAFETY DEMONSTRATION ===

1. Without generics (raw types):
ClassCastException: java.lang.Integer cannot be cast to java.lang.String

2. With generics (type safety):
First element: Hello

3. Generic collections:
Apple count: 5

4. Generic class type safety:
Box content: Hello Generics
Box number: 42

5. Type inference (diamond operator <>):
First element: A

=== WILDCARDS DEMONSTRATION ===

1. Upper bounded wildcard (? extends T):
Sum of integers: 15.0
Sum of doubles: 6.6

2. Lower bounded wildcard (? super T):
Copied numbers: [1, 2, 3]

3. Unbounded wildcard (?):
A B C
1 2 3

4. Wildcard in generic class:
Unknown box: Secret

5. Wildcard capture:
Wild list size: 3
```

```
=== GENERICS WITH COLLECTIONS ===

1. Type-safe collections:

2. Generic methods with collections:
First number: 1

3. Custom generic collections:
Sorted strings min: Apple
Sorted strings max: Zebra
Sorted integers min: 1
Sorted integers max: 8

4. Generic algorithms:
Original: [New York, London, Tokyo]
Reversed: [Tokyo, London, New York]
Maximum city: Tokyo

5. Nested generics:
Alice: [85, 90, 78]
Bob: [92, 88, 95]

=== GENERICS INHERITANCE ===

1. Generic class inheritance:
Original: hello generics
Uppercase: HELLO GENERICS

2. Generic subclass with additional type parameter:
Number: 5
Square: 25.0

3. Inheritance with wildcards:

4. Generic interface implementation:
Found: user123

=== GENERICS AND REFLECTION ===

1. Type erasure:
stringList class: class java.util.ArrayList
intList class: class java.util.ArrayList
Same class? true

2. Generic type information:
stringBox type: GenericsDemo$Box
intBox type: GenericsDemo$Box
```

```
3. ParameterizedType in fields:
Field: strings
  Type: interface java.util.List
  Generic type: java.util.List<java.lang.String>
  Raw type: interface java.util.List
  Type arguments: [class java.lang.String]
Field: map
  Type: interface java.util.Map
  Generic type: java.util.Map<java.lang.String, java.lang.Integer>
  Raw type: interface java.util.Map
  Type arguments: [class java.lang.String, class java.lang.Integer]

=== PRACTICAL GENERICS EXAMPLES ===

1. Data repository pattern:
All users: [User{id='1', name='Alice'}, User{id='2', name='Bob'}]
Find by ID '1': Optional[User{id='1', name='Alice'}]

2. Builder pattern with generics:
Generated query: SELECT * FROM User WHERE active = true AND age > 18

3. Event system with generics:
Event received: User logged in

4. Calculator with generics:
5 + 3 = 8
5 * 3 = 15
5.5 + 3.2 = 8.7

=== GENERICS RESTRICTIONS ===
1. Cannot instantiate generic types with primitive types
2. Cannot create instances of type parameters
3. Cannot declare static fields of type parameters
4. Cannot use casts or instanceof with parameterized types
5. Cannot create arrays of parameterized types
6. Cannot create, catch, or throw generic exceptions
7. Cannot overload methods with same erasure

=== TYPE ERASURE ===
Generics are implemented using type erasure:
  - Type parameters removed at compile time
  - Replaced with bounds (Object if unbounded)
  - Type casts inserted where necessary
  - Bridge methods generated for polymorphism

=== BEST PRACTICES ===
```

```
1. Use generics for type safety in collections
2. Prefer generic methods over raw types
3. Use bounded wildcards for maximum flexibility
4. Avoid using raw types in new code
5. Document generic type parameters
6. Use @SuppressWarnings judiciously

=== COMMON PITFALLS ===
1. Using raw types (lose type safety)
2. Ignoring unchecked warnings
3. Misusing wildcards
4. Trying to create generic arrays
5. Forgetting about type erasure

=== REAL-WORLD USAGE ===
1. Collections Framework (ArrayList<T>, HashMap<K,V>)
2. Streams API (Stream<T>)
3. Optional class (Optional<T>)
4. Functional interfaces (Function<T,R>)
5. Repository pattern in data access layers
6. Builder pattern implementations

=== COMPREHENSIVE EXAMPLE: GENERIC UTILITY CLASS ===
Even numbers: [2, 4, 6, 8, 10]
Squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
First number > 5: 6
```

Comprehensive Project: Library Management System

```java
import java.io.*;
import java.util.*;
import java.util.stream.Collectors;
import java.time.*;
import java.time.format.DateTimeFormatter;

public class LibraryManagementSystem {

    // ==================== EXCEPTION CLASSES ====================

    static class LibraryException extends Exception {
        private final String errorCode;

        public LibraryException(String message, String errorCode) {
            super(message);
            this.errorCode = errorCode;
        }

        public String getErrorCode() {
```

```
            return errorCode;
        }
    }

    static class BookNotFoundException extends LibraryException {
        public BookNotFoundException(String bookId) {
            super("Book not found: " + bookId, "BOOK_404");
        }
    }

    static class MemberNotFoundException extends LibraryException {
        public MemberNotFoundException(String memberId) {
            super("Member not found: " + memberId, "MEMBER_404");
        }
    }

    static class BookNotAvailableException extends LibraryException {
        public BookNotAvailableException(String bookId) {
            super("Book not available: " + bookId, "BOOK_UNAVAILABLE");
        }
    }

    static class LoanLimitExceededException extends LibraryException {
        public LoanLimitExceededException(String memberId, int limit) {
            super("Member " + memberId + " exceeded loan limit of " +
                limit, "LOAN_LIMIT");
        }
    }

    // ==================== MODEL CLASSES ====================

    static abstract class LibraryItem {
        protected final String id;
        protected final String title;
        protected final String author;
        protected boolean isAvailable;
        protected LocalDate dueDate;

        public LibraryItem(String id, String title, String author) {
            this.id = id;
            this.title = title;
            this.author = author;
            this.isAvailable = true;
        }

        public abstract String getType();

        public String getId() { return id; }
        public String getTitle() { return title; }
        public String getAuthor() { return author; }
        public boolean isAvailable() { return isAvailable; }
        public LocalDate getDueDate() { return dueDate; }

        public void borrowItem() {
            this.isAvailable = false;
            this.dueDate = LocalDate.now().plusDays(14); // 2 weeks
                loan period
        }
```

```
76
77          public void returnItem() {
78              this.isAvailable = true;
79              this.dueDate = null;
80          }
81
82          public boolean isOverdue() {
83              return dueDate != null && LocalDate.now().isAfter(dueDate);
84          }
85
86          @Override
87          public String toString() {
88              return String.format("%s [ID: %s, Title: %s, Author: %s,
                    Available: %s]",
89                                   getType(), id, title, author,
                                         isAvailable);
90          }
91      }
92
93      static class Book extends LibraryItem {
94          private final String isbn;
95          private final int publicationYear;
96          private final String genre;
97
98          public Book(String id, String title, String author,
99                      String isbn, int publicationYear, String genre) {
100             super(id, title, author);
101             this.isbn = isbn;
102             this.publicationYear = publicationYear;
103             this.genre = genre;
104         }
105
106         @Override
107         public String getType() {
108             return "Book";
109         }
110
111         public String getIsbn() { return isbn; }
112         public int getPublicationYear() { return publicationYear; }
113         public String getGenre() { return genre; }
114
115         @Override
116         public String toString() {
117             return super.toString() + String.format(
118                 ", ISBN: %s, Year: %d, Genre: %s", isbn,
                        publicationYear, genre);
119         }
120     }
121
122     static class Magazine extends LibraryItem {
123         private final String issueNumber;
124         private final LocalDate publicationDate;
125
126         public Magazine(String id, String title, String author,
127                         String issueNumber, LocalDate publicationDate) {
128             super(id, title, author);
129             this.issueNumber = issueNumber;
130             this.publicationDate = publicationDate;
```

```
131         }
132
133         @Override
134         public String getType() {
135             return "Magazine";
136         }
137
138         public String getIssueNumber() { return issueNumber; }
139         public LocalDate getPublicationDate() { return publicationDate;
                }
140
141         @Override
142         public String toString() {
143             return super.toString() + String.format(
144                 ", Issue: %s, Publication Date: %s",
145                 issueNumber, publicationDate.format(DateTimeFormatter.
                        ISO_DATE));
146         }
147     }
148
149     static class Member {
150         private final String id;
151         private final String name;
152         private final String email;
153         private final LocalDate membershipDate;
154         private final int maxLoans;
155         private final List<String> borrowedItems;
156
157         public Member(String id, String name, String email, int
                maxLoans) {
158             this.id = id;
159             this.name = name;
160             this.email = email;
161             this.membershipDate = LocalDate.now();
162             this.maxLoans = maxLoans;
163             this.borrowedItems = new ArrayList<>();
164         }
165
166         public void borrowItem(String itemId) throws LibraryException {
167             if (borrowedItems.size() >= maxLoans) {
168                 throw new LoanLimitExceededException(id, maxLoans);
169             }
170             borrowedItems.add(itemId);
171         }
172
173         public void returnItem(String itemId) {
174             borrowedItems.remove(itemId);
175         }
176
177         public boolean canBorrowMore() {
178             return borrowedItems.size() < maxLoans;
179         }
180
181         public String getId() { return id; }
182         public String getName() { return name; }
183         public String getEmail() { return email; }
184         public LocalDate getMembershipDate() { return membershipDate; }
185         public int getMaxLoans() { return maxLoans; }
```

```
186         public List<String> getBorrowedItems() { return new ArrayList
                <>(borrowedItems); }
187         public int getCurrentLoans() { return borrowedItems.size(); }
188 
189         @Override
190         public String toString() {
191             return String.format(
192                 "Member{id='%s', name='%s', email='%s', membershipDate
                        =%s, loans=%d/%d}",
193                 id, name, email, membershipDate, borrowedItems.size(),
                        maxLoans);
194         }
195     }
196 
197     static class LoanRecord {
198         private final String loanId;
199         private final String itemId;
200         private final String memberId;
201         private final LocalDate loanDate;
202         private LocalDate returnDate;
203         private double fineAmount;
204 
205         public LoanRecord(String loanId, String itemId, String memberId
                ) {
206             this.loanId = loanId;
207             this.itemId = itemId;
208             this.memberId = memberId;
209             this.loanDate = LocalDate.now();
210             this.fineAmount = 0.0;
211         }
212 
213         public void returnItem() {
214             this.returnDate = LocalDate.now();
215             calculateFine();
216         }
217 
218         private void calculateFine() {
219             if (returnDate != null && returnDate.isAfter(loanDate.
                        plusDays(14))) {
220                 long daysOverdue = returnDate.toEpochDay() - loanDate.
                            plusDays(14).toEpochDay();
221                 this.fineAmount = daysOverdue * 0.50; // $0.50 per day
222             }
223         }
224 
225         public String getLoanId() { return loanId; }
226         public String getItemId() { return itemId; }
227         public String getMemberId() { return memberId; }
228         public LocalDate getLoanDate() { return loanDate; }
229         public LocalDate getReturnDate() { return returnDate; }
230         public double getFineAmount() { return fineAmount; }
231         public boolean isReturned() { return returnDate != null; }
232 
233         @Override
234         public String toString() {
235             return String.format(
236                 "LoanRecord{loanId='%s', itemId='%s', memberId='%s',
                        loanDate=%s, " +
```

```
                "returnDate=%s, fineAmount=%.2f}",
                loanId, itemId, memberId, loanDate, returnDate,
                    fineAmount);
        }
    }

    // ==================== LIBRARY SERVICE WITH GENERICS
        ====================

    static class LibraryService<T extends LibraryItem> {
        private final Map<String, T> items;
        private final Map<String, Member> members;
        private final List<LoanRecord> loanRecords;
        private int loanCounter;

        public LibraryService() {
            this.items = new HashMap<>();
            this.members = new HashMap<>();
            this.loanRecords = new ArrayList<>();
            this.loanCounter = 1;
        }

        // ============ ITEM MANAGEMENT ============

        public void addItem(T item) {
            items.put(item.getId(), item);
            System.out.println("Added: " + item);
        }

        public T getItem(String itemId) throws BookNotFoundException {
            T item = items.get(itemId);
            if (item == null) {
                throw new BookNotFoundException(itemId);
            }
            return item;
        }

        public List<T> searchItems(String keyword) {
            return items.values().stream()
                .filter(item -> item.getTitle().toLowerCase().contains(
                    keyword.toLowerCase()) ||
                                item.getAuthor().toLowerCase().contains(
                                    keyword.toLowerCase()))
                .collect(Collectors.toList());
        }

        public List<T> getAvailableItems() {
            return items.values().stream()
                .filter(LibraryItem::isAvailable)
                .collect(Collectors.toList());
        }

        public List<T> getOverdueItems() {
            return items.values().stream()
                .filter(LibraryItem::isOverdue)
                .collect(Collectors.toList());
        }

```

```
291         // ============ MEMBER MANAGEMENT ============
292
293         public void addMember(Member member) {
294             members.put(member.getId(), member);
295             System.out.println("Added member: " + member);
296         }
297
298         public Member getMember(String memberId) throws
                MemberNotFoundException {
299             Member member = members.get(memberId);
300             if (member == null) {
301                 throw new MemberNotFoundException(memberId);
302             }
303             return member;
304         }
305
306         // ============ LOAN OPERATIONS ============
307
308         public synchronized String borrowItem(String itemId, String
                memberId)
309                 throws LibraryException {
310
311             T item = getItem(itemId);
312             Member member = getMember(memberId);
313
314             if (!item.isAvailable()) {
315                 throw new BookNotAvailableException(itemId);
316             }
317
318             member.borrowItem(itemId);
319             item.borrowItem();
320
321             String loanId = "LOAN-" + loanCounter++;
322             LoanRecord record = new LoanRecord(loanId, itemId, memberId
                    );
323             loanRecords.add(record);
324
325             System.out.printf("Loan created: %s borrowed by %s (Due: %s
                    )%n",
326                               item.getTitle(), member.getName(), item.
                                  getDueDate());
327
328             return loanId;
329         }
330
331         public synchronized double returnItem(String itemId, String
                memberId)
332                 throws LibraryException {
333
334             T item = getItem(itemId);
335             Member member = getMember(memberId);
336
337             item.returnItem();
338             member.returnItem(itemId);
339
340             // Find and update loan record
341             LoanRecord record = loanRecords.stream()
342                 .filter(r -> r.getItemId().equals(itemId) &&
```

```
                        r.getMemberId().equals(memberId) &&
                        !r.isReturned())
                .findFirst()
                .orElseThrow(() -> new LibraryException("Loan record
                    not found", "LOAN_404"));

        record.returnItem();

        System.out.printf("Item returned: %s by %s", item.getTitle
            (), member.getName());
        if (record.getFineAmount() > 0) {
            System.out.printf(" (Fine: $%.2f)", record.
                getFineAmount());
        }
        System.out.println();

        return record.getFineAmount();
    }

    // ============ REPORTS ============

    public void generateReport() {
        System.out.println("\n=== LIBRARY REPORT ===");

        System.out.println("\n1. Collection Statistics:");
        System.out.printf("  Total items: %d%n", items.size());
        System.out.printf("  Available items: %d%n",
            getAvailableItems().size());
        System.out.printf("  Borrowed items: %d%n", items.size() -
            getAvailableItems().size());
        System.out.printf("  Overdue items: %d%n", getOverdueItems
            ().size());

        System.out.println("\n2. Member Statistics:");
        System.out.printf("  Total members: %d%n", members.size());

        Map<String, Long> loansByMember = loanRecords.stream()
            .filter(r -> !r.isReturned())
            .collect(Collectors.groupingBy(
                LoanRecord::getMemberId,
                Collectors.counting()
            ));

        System.out.println("  Current loans by member:");
        loansByMember.forEach((memberId, count) -> {
            try {
                Member member = getMember(memberId);
                System.out.printf("    %s: %d loan(s)%n", member.
                    getName(), count);
            } catch (MemberNotFoundException e) {
                System.out.printf("    %s: %d loan(s)%n", memberId,
                    count);
            }
        });

        System.out.println("\n3. Popular Items:");
        Map<String, Long> loanCounts = loanRecords.stream()
            .collect(Collectors.groupingBy(
```

```
                LoanRecord::getItemId,
                Collectors.counting()
            ));

        loanCounts.entrySet().stream()
            .sorted((e1, e2) -> Long.compare(e2.getValue(), e1.
                getValue()))
            .limit(5)
            .forEach(entry -> {
                try {
                    T item = getItem(entry.getKey());
                    System.out.printf("    %s: %d loans%n", item.
                        getTitle(), entry.getValue());
                } catch (BookNotFoundException e) {
                    System.out.printf("    Item %s: %d loans%n",
                        entry.getKey(), entry.getValue());
                }
            });

        System.out.println("\n4. Fines Summary:");
        double totalFines = loanRecords.stream()
            .mapToDouble(LoanRecord::getFineAmount)
            .sum();
        System.out.printf("  Total fines collected: $%.2f%n",
            totalFines);
    }

    // ============ FILE OPERATIONS ============

    public void saveToFile(String filename) throws IOException {
        try (ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream(filename))) {

            LibraryData data = new LibraryData(items, members,
                loanRecords, loanCounter);
            oos.writeObject(data);
            System.out.println("Library data saved to: " + filename
                );
        }
    }

    @SuppressWarnings("unchecked")
    public void loadFromFile(String filename) throws IOException,
        ClassNotFoundException {
        try (ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream(filename))) {

            LibraryData data = (LibraryData) ois.readObject();
            items.clear();
            items.putAll((Map<String, T>) data.items);
            members.clear();
            members.putAll(data.members);
            loanRecords.clear();
            loanRecords.addAll(data.loanRecords);
            loanCounter = data.loanCounter;

            System.out.println("Library data loaded from: " +
                filename);
```

```
443                 System.out.println("Items: " + items.size() + ",
                        Members: " + members.size());
444             }
445         }
446
447         // Data container for serialization
448         static class LibraryData implements Serializable {
449             final Map<String, ?> items;
450             final Map<String, Member> members;
451             final List<LoanRecord> loanRecords;
452             final int loanCounter;
453
454             LibraryData(Map<String, ?> items, Map<String, Member>
                    members,
455                         List<LoanRecord> loanRecords, int loanCounter) {
456                 this.items = items;
457                 this.members = members;
458                 this.loanRecords = loanRecords;
459                 this.loanCounter = loanCounter;
460             }
461         }
462     }
463
464     // ==================== MAIN DEMONSTRATION ====================
465
466     public static void main(String[] args) {
467         System.out.println("=== LIBRARY MANAGEMENT SYSTEM DEMONSTRATION
                ===\n");
468
469         // Create library service
470         LibraryService<LibraryItem> library = new LibraryService<>();
471
472         try {
473             // ============ 1. EXCEPTION HANDLING DEMONSTRATION
                    ============
474             System.out.println("=== 1. EXCEPTION HANDLING DEMONSTRATION
                    ===");
475
476             try {
477                 // Try to get non-existent book
478                 library.getItem("NON_EXISTENT");
479             } catch (BookNotFoundException e) {
480                 System.out.println("Caught BookNotFoundException: " + e
                        .getMessage());
481                 System.out.println("Error Code: " + e.getErrorCode());
482             }
483
484             try {
485                 // Try to get non-existent member
486                 library.getMember("NON_EXISTENT");
487             } catch (MemberNotFoundException e) {
488                 System.out.println("\nCaught MemberNotFoundException: "
                        + e.getMessage());
489                 System.out.println("Error Code: " + e.getErrorCode());
490             }
491
492             // ============ 2. ADD ITEMS (GENERICS) ============
```

```
493             System.out.println("\n=== 2. ADDING LIBRARY ITEMS (GENERICS
                    ) ===");
494
495             // Add books
496             Book book1 = new Book("B001", "The Great Gatsby", "F. Scott
                    Fitzgerald",
497                                   "9780743273565", 1925, "Fiction");
498             Book book2 = new Book("B002", "To Kill a Mockingbird", "
                    Harper Lee",
499                                   "9780061120084", 1960, "Fiction");
500             Book book3 = new Book("B003", "1984", "George Orwell",
501                                   "9780451524935", 1949, "Dystopian");
502
503             library.addItem(book1);
504             library.addItem(book2);
505             library.addItem(book3);
506
507             // Add magazine (different type, same generic)
508             Magazine magazine1 = new Magazine("M001", "National
                    Geographic",
509                                               "Various Authors", "June
                                                    2023",
510                                               LocalDate.of(2023, 6, 1));
511             library.addItem(magazine1);
512
513             // ============ 3. ADD MEMBERS ============
514             System.out.println("\n=== 3. ADDING MEMBERS ===");
515
516             Member member1 = new Member("M001", "Alice Johnson", "
                    alice@email.com", 3);
517             Member member2 = new Member("M002", "Bob Smith", "bob@email
                    .com", 2);
518             Member member3 = new Member("M003", "Charlie Brown", "
                    charlie@email.com", 5);
519
520             library.addMember(member1);
521             library.addMember(member2);
522             library.addMember(member3);
523
524             // ============ 4. LOAN OPERATIONS ============
525             System.out.println("\n=== 4. LOAN OPERATIONS ===");
526
527             try {
528                 // Successful loans
529                 String loan1 = library.borrowItem("B001", "M001");
530                 String loan2 = library.borrowItem("B002", "M001");
531                 String loan3 = library.borrowItem("B003", "M002");
532
533                 System.out.println("\nCreated loans: " + loan1 + ", " +
                        loan2 + ", " + loan3);
534
535                 // Try to borrow unavailable book
536                 try {
537                     library.borrowItem("B001", "M003");  // Already
                            borrowed
538                 } catch (BookNotAvailableException e) {
539                     System.out.println("\nCaught
                            BookNotAvailableException: " + e.getMessage());
```

```
            }

            // Try to exceed loan limit
            try {
                library.borrowItem("M001", "M001");  // Member already has 2 loans (max 3)
                // This would succeed if member had less than max
            } catch (LibraryException e) {
                System.out.println("\nCaught: " + e.getClass().getSimpleName() +
                                   ": " + e.getMessage());
            }

        } catch (LibraryException e) {
            System.out.println("Error during loan: " + e.getMessage());
        }

        // ============ 5. SEARCH AND FILTER (COLLECTIONS) ============
        System.out.println("\n=== 5. SEARCH AND FILTER (COLLECTIONS) ===");

        System.out.println("\nSearch for 'kill':");
        List<LibraryItem> searchResults = library.searchItems("kill");
        searchResults.forEach(System.out::println);

        System.out.println("\nAvailable items:");
        List<LibraryItem> availableItems = library.getAvailableItems();
        availableItems.forEach(System.out::println);

        // ============ 6. RETURN OPERATIONS ============
        System.out.println("\n=== 6. RETURN OPERATIONS ===");

        try {
            double fine1 = library.returnItem("B001", "M001");
            double fine2 = library.returnItem("B002", "M001");

            System.out.printf("\nTotal fines from returns: $%.2f%n", fine1 + fine2);

        } catch (LibraryException e) {
            System.out.println("Error during return: " + e.getMessage());
        }

        // ============ 7. GENERATE REPORT ============
        System.out.println("\n=== 7. GENERATING REPORTS ===");
        library.generateReport();

        // ============ 8. FILE I/O OPERATIONS ============
        System.out.println("\n=== 8. FILE I/O OPERATIONS ===");

        String dataFile = "library_data.dat";

        try {
```

```
            // Save to file
            library.saveToFile(dataFile);

            // Create new library instance and load from file
            LibraryService<LibraryItem> restoredLibrary = new
                LibraryService<>();
            restoredLibrary.loadFromFile(dataFile);

            System.out.println("\nRestored library statistics:");
            System.out.println("Available items: " +
                restoredLibrary.getAvailableItems().size());

            // Clean up
            new File(dataFile).delete();
            System.out.println("Cleaned up data file");

        } catch (IOException | ClassNotFoundException e) {
            System.out.println("File operation error: " + e.
                getMessage());
        }

        // ============ 9. ADVANCED COLLECTIONS OPERATIONS
            ============
        System.out.println("\n=== 9. ADVANCED COLLECTIONS
            OPERATIONS ===");

        // Group items by type using Streams
        Map<String, List<LibraryItem>> itemsByType = library.
            searchItems("").stream()
            .collect(Collectors.groupingBy(LibraryItem::getType));

        System.out.println("\nItems by type:");
        itemsByType.forEach((type, items) ->
            System.out.printf("  %s: %d items%n", type, items.size
                ()));

        // Sort items by title
        System.out.println("\nItems sorted by title:");
        library.searchItems("").stream()
            .sorted(Comparator.comparing(LibraryItem::getTitle))
            .forEach(item -> System.out.println("  " + item.
                getTitle()));

        // ============ 10. EXCEPTION CHAINING ============
        System.out.println("\n=== 10. EXCEPTION CHAINING ===");

        try {
            performComplexLibraryOperation(library);
        } catch (LibraryException e) {
            System.out.println("Caught chained exception:");
            System.out.println("  Message: " + e.getMessage());
            System.out.println("  Error Code: " + e.getErrorCode())
                ;
            if (e.getCause() != null) {
                System.out.println("  Cause: " + e.getCause().
                    getMessage());
            }
        }
```

```
637
638         } catch (Exception e) {
639             System.out.println("Unexpected error: " + e.getMessage());
640             e.printStackTrace();
641         }
642
643         System.out.println("\n=== SYSTEM FEATURES DEMONSTRATED ===");
644         System.out.println("1. Exception Handling: Custom exceptions
                with error codes");
645         System.out.println("2. Generics: Type-safe LibraryService<T>");
646         System.out.println("3. Collections: Lists, Maps, Streams
                operations");
647         System.out.println("4. File I/O: Serialization of library data"
                );
648         System.out.println("5. Type Safety: Compile-time checks for
                operations");
649         System.out.println("6. Real-world: Complete library management
                functionality");
650     }
651
652     // Helper method for exception chaining demonstration
653     private static void performComplexLibraryOperation(LibraryService<
            LibraryItem> library)
654             throws LibraryException {
655
656         try {
657             // Try multiple operations that might fail
658             library.borrowItem("NON_EXISTENT", "M001");
659             library.borrowItem("B001", "NON_EXISTENT");
660
661         } catch (BookNotFoundException | MemberNotFoundException e) {
662             // Chain the exception with additional context
663             throw new LibraryException(
664                 "Failed to complete library operation: " + e.getMessage
                        (),
665                 "OPERATION_FAILED",
666                 e
667             );
668         }
669     }
670 }
```

Listing 9: Complete Library Management System Using All Unit III Concepts

LibraryManagementSystem Program Output

```
=== LIBRARY MANAGEMENT SYSTEM DEMONSTRATION ===

=== 1. EXCEPTION HANDLING DEMONSTRATION ===
Caught BookNotFoundException: Book not found: NON_EXISTENT
Error Code: BOOK_404

Caught MemberNotFoundException: Member not found: NON_EXISTENT
Error Code: MEMBER_404
```

```
=== 2. ADDING LIBRARY ITEMS (GENERICS) ===
Added: Book [ID: B001, Title: The Great Gatsby, Author: F. Scott Fitzgerald, Availa
Added: Book [ID: B002, Title: To Kill a Mockingbird, Author: Harper Lee, Available:
Added: Book [ID: B003, Title: 1984, Author: George Orwell, Available: true], ISBN:
Added: Magazine [ID: M001, Title: National Geographic, Author: Various Authors, Ava

=== 3. ADDING MEMBERS ===
Added member: Member{id='M001', name='Alice Johnson', email='alice@email.com', memb
Added member: Member{id='M002', name='Bob Smith', email='bob@email.com', membership
Added member: Member{id='M003', name='Charlie Brown', email='charlie@email.com', me

=== 4. LOAN OPERATIONS ===
Loan created: The Great Gatsby borrowed by Alice Johnson (Due: 2024-01-29)
Loan created: To Kill a Mockingbird borrowed by Alice Johnson (Due: 2024-01-29)
Loan created: 1984 borrowed by Bob Smith (Due: 2024-01-29)

Created loans: LOAN-1, LOAN-2, LOAN-3

Caught BookNotAvailableException: Book not available: B001

Caught: BookNotFoundException: Book not found: M001

=== 5. SEARCH AND FILTER (COLLECTIONS) ===

Search for 'kill':
Book [ID: B002, Title: To Kill a Mockingbird, Author: Harper Lee, Available: false]

Available items:
Magazine [ID: M001, Title: National Geographic, Author: Various Authors, Available:

=== 6. RETURN OPERATIONS ===
Item returned: The Great Gatsby by Alice Johnson
Item returned: To Kill a Mockingbird by Alice Johnson

Total fines from returns: $0.00

=== 7. GENERATING REPORTS ===

=== LIBRARY REPORT ===

1. Collection Statistics:
  Total items: 4
  Available items: 3
  Borrowed items: 1
  Overdue items: 0

2. Member Statistics:
```

```
  Total members: 3
  Current loans by member:
    Bob Smith: 1 loan(s)

3. Popular Items:
    The Great Gatsby: 1 loans
    To Kill a Mockingbird: 1 loans
    1984: 1 loans
    National Geographic: 0 loans

4. Fines Summary:
  Total fines collected: $0.00

=== 8. FILE I/O OPERATIONS ===
Library data saved to: library_data.dat
Library data loaded from: library_data.dat
Items: 4, Members: 3

Restored library statistics:
Available items: 3
Cleaned up data file

=== 9. ADVANCED COLLECTIONS OPERATIONS ===

Items by type:
  Book: 3 items
  Magazine: 1 items

Items sorted by title:
  1984
  National Geographic
  The Great Gatsby
  To Kill a Mockingbird

=== 10. EXCEPTION CHAINING ===
Caught chained exception:
  Message: Failed to complete library operation: Book not found: NON_EXISTENT
  Error Code: OPERATION_FAILED
  Cause: Book not found: NON_EXISTENT

=== SYSTEM FEATURES DEMONSTRATED ===
1. Exception Handling: Custom exceptions with error codes
2. Generics: Type-safe LibraryService<T>
3. Collections: Lists, Maps, Streams operations
4. File I/O: Serialization of library data
5. Type Safety: Compile-time checks for operations
6. Real-world: Complete library management functionality
```

Unit Summary and Assessment

Key Concepts Mastered

1. **Exception Handling:** Checked vs unchecked exceptions, try-catch-finally, throw/throws, custom exceptions
2. **File I/O:** Byte streams vs character streams, buffered streams, file operations, encoding
3. **Collections Framework:** List (ArrayList, LinkedList), Set (HashSet, TreeSet), Map (HashMap, TreeMap)
4. **Generics:** Type safety, generic classes/methods, wildcards, type erasure

Practical Skills Developed

- Design robust applications with proper exception handling
- Implement file operations for data persistence
- Choose appropriate collections for different use cases
- Write type-safe code using generics
- Develop complete systems integrating all concepts

Assessment Methods

Component	Weightage
Programming Assignments	30%
File I/O Project	20%
Collections Implementation	20%
End-Term Examination	30%

Practice Problems

1. Create a Student Grade Management System with exception handling
2. Implement a File Backup Utility with progress tracking
3. Build a Shopping Cart using appropriate collections
4. Design a Generic Cache System with LRU eviction policy
5. Create a Log Analyzer with file I/O and collections
6. Implement a Contact Management System with file persistence

References

1. "Effective Java" by Joshua Bloch
2. "Java I/O, NIO and NIO.2" by Jeff Friesen
3. "Java Generics and Collections" by Maurice Naftalin
4. Oracle Java Tutorials: Exceptions, I/O, Collections, Generics
5. Baeldung: Comprehensive Java Tutorials

End of Unit III: Exception Handling, I/O, and Collections Framework