CS 122 Final Exam Study Guide

Hello everyone,

Congratulations on reaching the final exam! This comprehensive study guide covers Lectures 10-17 and builds on your midterm knowledge to prepare you for the comprehensive final assessment. The final exam will not only test these new topics but may also integrate earlier concepts in new ways—for example, using OOP principles with collections, or applying exception handling to file I/O.

Exam Format Reminder:

- **Section 1: Multiple-Choice & Short Answer:** Tests conceptual understanding of data structures, exception handling, file operations, recursion, and testing methodologies. Questions may integrate earlier topics.
- **Section 2: Programming Challenge:** Assesses your ability to apply concepts holistically—reading files into collections, handling exceptions, writing recursive algorithms. Reference materials allowed.

Use this guide to understand *why* these tools matter, not just memorize facts. Think about trade-offs and real-world applications.

Good luck with your studies!

Best,

Prof. Sarah L

Table of Contents

- CS 122 Final Exam Study Guide
  - Table of Contents
  - 10. Collections Framework & ArrayList
    - Understanding the Framework Concept
    - ArrayList: A Resizable Array
    - Autoboxing: Bridging Primitives and Objects
    - Iterating Over Collections
    - ArrayList Printout
  - 11. LinkedList and Performance Analysis
    - The Node Structure
    - The Critical Trade-Off
    - A Performance Horror Story
    - When to Choose Each
    - Additional LinkedList Methods
  - 12. Sets and Queues
    - Set Interface: Guaranteeing Uniqueness
    - HashSet: Fast, Unordered, Efficient
    - Queue Interface: First-In, First-Out
    - ArrayBlockingQueue: Thread-Safe, Bounded, Blocking

- 13. Maps as Lookup Tables
  - Why Maps Matter
  - HashMap Implementation Details
  - Key Rules
  - Real-World Example: Caching
  - Iteration
- 14. Binary Search: Efficient Searching
  - The Critical Prerequisite
  - The Algorithm
  - Example: Finding 23 in [2, 5, 8, 12, 16, 23, 38, 45, 56, 67, 78]
  - Speed Advantage
  - Implementation in Java
- 15. Exception Handling & Recovery
  - Exceptions vs. Errors
  - Try-Catch Block: The Fundamental Pattern
  - Multiple Catch Blocks
  - Finally Block: Code That Always Runs
  - Exception Object Methods
  - Checked vs. Unchecked Exceptions
  - The Throws Keyword: Passing Responsibility
- 16. File Input/Output Operations
  - File Class: Representing Files and Directories
  - Creating Files
  - FileWriter: Writing Text to Files
  - BufferedWriter: Efficient Writing
  - FileReader and BufferedReader: Reading Text
  - Paths: Relative vs. Absolute
  - Creating Directories
  - IOException: The Checked Exception
- 17. Recursion and Testing with JUnit
  - Recursion: A Method Calling Itself
  - Example: Factorial
  - Example: Fibonacci
  - Common Pitfalls
  - Testing: The Professional Approach
  - JUnit 5: Automated Testing Framework
  - Assert Statements: Verification
  - Edge Cases: Testing the Boundaries
  - Test Naming Convention
  - Testing Collections and File I/O
- Integration & Advanced Questions
- Interactive Study Application: The Java Competency Crucible
  - Example Run

---

10. Collections Framework & ArrayList

In the midterm, you learned about arrays—a fundamental way to store multiple items. Arrays are powerful, but they have a critical limitation: their size is fixed at creation. What if you don't know how many students will enroll? What if your application needs to grow dynamically?

The Collections Framework is Java's answer. It provides pre-built, optimized data structures for common programming tasks. Rather than reinventing resizable arrays yourself, you use `ArrayList`—a battle-tested class that handles growth automatically.

Understanding the Framework Concept

A framework is a toolkit of reusable types (classes and interfaces) that solve common problems. The Collections Framework, in the `java.util` package, gives you several data structure options. Using a framework saves time, reduces bugs, and lets you focus on business logic.

ArrayList: A Resizable Array

```
ArrayList<String> students = new ArrayList<>();
```

This creates an empty list that grows as needed. The angle brackets `<String>` specify the type—**generics** ensure type safety. If you accidentally try to add an Integer to `ArrayList<String>`, the compiler catches it immediately.

Key methods:

- `.add(object)` — Adds to the end (O(1) amortized)
- `.get(index)` — Retrieves by index (O(1))
- `.set(index, object)` — Replaces an element (O(1))
- `.remove(index)` — Removes and shifts remaining elements (O(n) in worst case)
- `.size()` — Returns the number of elements (O(1))
- `.clear()` — Removes all elements (O(n))

When capacity is exceeded, ArrayList creates a larger internal array and copies everything over. This happens behind the scenes.

Autoboxing: Bridging Primitives and Objects

Collections store only objects, not primitives. When you add a primitive, Java automatically wraps it:

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(5);  // Automatically becomes new Integer(5)
int first = numbers.get(0);  // Automatically unwraps to int
```

This **autoboxing** (and unboxing) makes code cleaner while maintaining type safety.

Iterating Over Collections

The for-each loop is perfect for collections:

```
for (String name : students) {
    System.out.println(name);
}
```

This is cleaner and less error-prone than index-based loops, and it works with any Collection type.

ArrayList Printout

ArrayList has a built-in `toString()` method that displays all elements:

```
ArrayList<Integer> nums = new ArrayList<>();
nums.add(1); nums.add(2); nums.add(3);
System.out.println(nums);  // Output: [1, 2, 3]
```

- **Guiding Questions:**
  - Why is ArrayList called a "resizable" array? How does it know when to grow?
  - If you have 1,000 items in an ArrayList and call `.remove(500)`, what happens to items at indices 501-999?
  - What is the time complexity of `.add()` at the end of an ArrayList? What about in the middle?
  - Why does Java provide generics (`ArrayList<String>`)? What errors does it prevent?
  - When you use `.add(5)` on `ArrayList<Integer>`, what specifically does autoboxing do?
- **Code to Review:**
  - Create an ArrayList of test scores, add 10 scores, remove the lowest score, then calculate and print the average.
  - Write a program that reads strings from user input into an ArrayList until the user types "done", then prints all strings in reverse order.
  - Compare the speed of printing an ArrayList using direct `.println()` versus a for-each loop.

11. LinkedList and Performance Analysis

ArrayList is great for random access—retrieving element at index 500 takes the same time whether you have 1,000 or 1,000,000 elements. But adding or removing elements in the middle requires shifting many elements, which is slow for large lists.

LinkedList takes a completely different approach. Instead of a contiguous array, it chains elements together using **nodes**. Each node contains data and a reference to the next (and previous) node.

The Node Structure

Imagine a train where each car is connected to the next:

```
[Data: "A" | Next: ●] → [Data: "B" | Next: ●] → [Data: "C" | Next: null]
```

LinkedList is **doubly-linked** in Java, meaning each node also links to the previous node. This allows efficient traversal in both directions.

The Critical Trade-Off

ArrayList:

- Random access: Fast (O(1)) — just calculate the memory address
- Adding/removing in middle: Slow (O(n)) — must shift elements

LinkedList:

- Random access: Slow (O(n)) — must traverse from start (or end)
- Adding/removing in middle: Fast (O(1)) — just reconnect links

A Performance Horror Story

This LinkedList code is deceptively slow:

```
LinkedList<Integer> nums = new LinkedList<>();
// ... add 1,000 elements ...

for (int i = 0; i < nums.size(); i++) {
    System.out.println(nums.get(i));
}
```

Why? Each `.get(i)` call traverses from the start of the list. Getting element 0 requires 1 traversal, element 1 requires 2 traversals, ..., element 999 requires 1,000 traversals. Total: ~500,000 traversals! ($O(n^2)$ complexity—terrible!)

The correct approach:

```
for (Integer num : nums) {
    System.out.println(num);
}
```

The for-each loop uses an **Iterator**, which walks through the list once sequentially. Total: exactly 1,000 steps. (O(n)—much better!)

When to Choose Each

Use **ArrayList** when:

- You frequently access elements by index
- Your list size is relatively stable
- You only occasionally add/remove in the middle

Use **LinkedList** when:

- You frequently add/remove elements at the beginning or middle
- You iterate through elements more than you access by index
- Your list size changes dramatically

Additional LinkedList Methods

LinkedList provides convenience methods for queue-like operations:

- `.addFirst(object)` — Adds to the beginning
- `.addLast(object)` — Adds to the end
- `.removeFirst()` — Removes from the beginning
- `.removeLast()` — Removes from the end
- `.getFirst()` — Retrieves first element
- `.getLast()` — Retrieves last element
- **Guiding Questions:**
  - Draw a picture of how a doubly-linked list stores three elements. Show the nodes, data, and links.
  - For a LinkedList of 1,000,000 elements, what is the time complexity of `.get(999999)`?
  - If you need to insert 1,000 elements at the beginning of a list, would ArrayList or LinkedList be faster?
  - Why does Java's LinkedList provide `.addFirst()` but ArrayList doesn't need a separate method?
  - In what scenario would an Iterator be better than a traditional for loop for LinkedList?
- **Code to Review:**
  - Implement a function that adds 100 elements to both ArrayList and LinkedList, then removes every 10th element. Time both and compare.
  - Write code to simulate a **deque** (double-ended queue) using LinkedList's `.addFirst()`, `.addLast()`, `.removeFirst()`, `.removeLast()`.
  - Demonstrate the O(n²) problem by timing a LinkedList index-based loop versus a for-each loop.

12. Sets and Queues

Not all problems require order. Sometimes you only care: "Does this item exist?" or "Are all items unique?" This is where **Sets** come in. Other times, you need strict ordering: first-in, first-out. That's where **Queues** help.

Set Interface: Guaranteeing Uniqueness

A Set is an unordered collection that guarantees every element is unique. If you try to add a duplicate, it's silently ignored.

```
HashSet<String> uniqueWords = new HashSet<>();
uniqueWords.add("Java");
uniqueWords.add("Java");  // Ignored
uniqueWords.add("Programming");
System.out.println(uniqueWords.size());  // Output: 2
```

Real-world uses:

- Finding unique words in a document
- Removing duplicates from a list
- Checking membership ("is this user already subscribed?")
- Caching visited items (to avoid reprocessing)

HashSet: Fast, Unordered, Efficient

HashSet uses **hashing** to achieve O(1) average lookup time. Here's how:

1. When you add an item, Java computes its hash code (like a fingerprint)
2. The hash code determines a "bucket" where the item is stored
3. When you check if something exists, Java recomputes the hash code and looks in that bucket
4. No scanning required—direct access!

Key methods:

- `.add(object)` — Adds an element (O(1) average)
- `.remove(object)` — Removes an element (O(1) average)
- `.contains(object)` — Checks if element exists (O(1) average)
- `.size()` — Returns the count (O(1))

Allows null: HashSet can store a single null value (since null has a hash code of 0).

No order guaranteed: Elements are stored based on hash code, not insertion order. If you iterate over a HashSet twice, you might see items in different orders.

Queue Interface: First-In, First-Out

Imagine a line at a coffee shop. The first customer in line is the first served. This is FIFO (First-In-First-Out) ordering.

```
Queue<String> orders = new LinkedList<>();
orders.add("Espresso");
orders.add("Latte");
String nextOrder = orders.remove();  // Returns "Espresso"
```

(Note: Queue is an interface, so you use LinkedList or another implementation.)

ArrayBlockingQueue: Thread-Safe, Bounded, Blocking

For concurrent programs where multiple threads need to communicate safely, `ArrayBlockingQueue` is essential:

```
ArrayBlockingQueue<Document> printQueue = new ArrayBlockingQueue<>(10);
```

This creates a queue with capacity 10. If a thread tries to add when full, it **blocks** (pauses) until space opens. If a thread tries to remove when empty, it blocks until a document arrives.

Key methods:

- `.add(object)` or `.offer(object)` — Adds to the tail
  - `.add()` throws an exception if full
  - `.offer()` returns false if full
- `.remove()` or `.poll()` — Removes from the head
  - `.remove()` throws an exception if empty
  - `.poll()` returns null if empty
- `.take()` / `.put()` — Blocking operations (wait if needed)

Does NOT allow null: Unlike HashSet, ArrayBlockingQueue rejects null values.

- **Guiding Questions:**
  - Why would you use a Set instead of an ArrayList for checking if an element exists?
  - What does it mean for HashSet to be "unordered"? Can you rely on iteration order?
  - If you add 1,000,000 items to a HashSet and then call `.contains()` on a random item, how long does it take?
  - In a coffee shop queue, if person A arrives before person B, is A served before B? Is that FIFO?
  - Why is ArrayBlockingQueue called "blocking"? When does it block a thread?
  - What's the difference between `.add()` and `.offer()` on an ArrayBlockingQueue when it's full?
- **Code to Review:**
  - Write a program that reads a list of usernames (with duplicates) from user input and prints only the unique names using HashSet.
  - Simulate a simple printer queue using ArrayBlockingQueue with one thread adding documents and another printing them.
  - Remove duplicates from an ArrayList using HashSet, then convert back to ArrayList.

13. Maps as Lookup Tables

Suppose you're building a music streaming app and need to look up a song by its ID and retrieve its metadata (title, artist, duration). Or you're building an email client and need to map each email address to a contact object. This is what **Maps** are for.

A Map stores **key-value pairs**. You use the key to look up the value, like a real-world dictionary where a word (key) maps to its definition (value).

```
HashMap<String, Double> studentGPA = new HashMap<>();
studentGPA.put("Alice", 3.8);
studentGPA.put("Bob", 3.5);
double aliceGPA = studentGPA.get("Alice");  // Returns 3.8
```

Why Maps Matter

Consider storing student data in an ArrayList of objects. To find a student by name, you'd loop through all students comparing names—O(n) time. With a HashMap, you hash the name and go directly to the student—O(1) time.

For 1,000,000 students, that's the difference between 500,000 comparisons and 1 comparison. That's the real-world impact of choosing the right data structure.

HashMap Implementation Details

Like HashSet, HashMap uses hashing. But it stores both the key and the value:

1. When you put a key-value pair, Java hashes the key
2. The hash determines a bucket where the pair is stored
3. When you get a key, Java recomputes the hash and retrieves the value from that bucket
4. If two keys hash to the same bucket (a collision), HashMap handles it with linked lists or trees

Key Rules

Keys must be unique: If you put a value under an existing key, the old value is overwritten.

```
HashMap<String, Integer> ages = new HashMap<>();
ages.put("Alice", 25);
ages.put("Alice", 26);  // Overwrites previous value
System.out.println(ages.size());  // Output: 1
```

Keys can be any object type: Strings, Integers, custom objects—anything. (But the object must have a proper `hashCode()` method.)

Values can be anything, including null: Two different keys can map to the same value, or even to null.

Unordered: Like HashSet, HashMap doesn't guarantee iteration order.

Real-World Example: Caching

A weather app fetches data from an API (slow). To avoid repeated network calls, cache the results:

```
HashMap<String, WeatherData> cache = new HashMap<>();

public WeatherData getWeather(String city) {
    if (cache.containsKey(city)) {
        return cache.get(city);  // Fast, cached result
```

```
    } else {
        WeatherData data = fetchFromAPI(city);  // Slow
        cache.put(city, data);  // Cache for next time
        return data;
    }
}
```

Iteration

To process all key-value pairs:

```
for (String name : studentGPA.keySet()) {
    double gpa = studentGPA.get(name);
    System.out.println(name + ": " + gpa);
}

// Or more efficiently:
for (Map.Entry<String, Double> entry : studentGPA.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

- **Guiding Questions:**
    - If you store data about customers in an ArrayList and frequently look them up by customer ID, would you use ArrayList or HashMap<Integer, Customer>? Why?
    - What happens if you put two entries with the same key in a HashMap?
    - How would you iterate through a HashMap in a way that doesn't require looking up values by key twice?
    - What is a real-world scenario where HashMap's O(1) lookup significantly outperforms ArrayList?
    - Why might you choose HashMap<String, String> instead of List<Pair<String, String>>?
- **Code to Review:**
    - Create a HashMap mapping product IDs (Integers) to product names (Strings). Add 5 products, look up one by ID, update one's name, and remove one.
    - Write a program that counts the frequency of each word in a sentence using HashMap<String, Integer>.
    - Implement a simple phone book using HashMap<String, String> that allows adding, looking up, and removing contacts.

14. Binary Search: Efficient Searching

You've learned several data structures now: lists, sets, maps. But a fundamental operation is **searching**. "Is this item in my collection? Where is it?"

The simplest approach is **linear search**: check each element one by one. For a list of 10 items, that's at most 10 checks. For 1 million items, that's 1 million checks.

Binary search is different. It works like looking up a word in a dictionary: you don't start at page 1. You flip to the middle, see if your word comes before or after, then eliminate half the dictionary. Repeat until you find it (or determine it's not there).

The Critical Prerequisite

Binary search ONLY works on SORTED arrays. This cannot be overstated. If your array isn't sorted, binary search gives wrong results.

```
int[] numbers = {3, 1, 4, 1, 5, 9, 2, 6};
Arrays.sort(numbers);  // Must sort first!
// Now: [1, 1, 2, 3, 4, 5, 6, 9]
int index = binarySearch(numbers, 5);  // Returns 5
```

The Algorithm

1. Start with `left = 0`, `right = array.length - 1`
2. Calculate `middle = left + (right - left) / 2`
3. Compare `array[middle]` with target:
    - If equal: Found it! Return `middle`
    - If target < array[middle]: Search left half (set `right = middle - 1`)
    - If target > array[middle]: Search right half (set `left = middle + 1`)
4. Repeat until `left > right` (not found)

Example: Finding 23 in [2, 5, 8, 12, 16, 23, 38, 45, 56, 67, 78]

```
Step 1: left=0, right=10, middle=5, array[5]=23 → Found! Return 5
```

That's just one step! Compare to linear search, which would need 6 steps.

Speed Advantage

For a sorted array of 1,000,000 elements:

- **Linear search:** Average ~500,000 comparisons (O(n))
- **Binary search:** ~20 comparisons (O(log n))

That's 25,000 times faster!

The trade-off: Sorting takes O(n log n) time. So binary search is worth it when you do multiple searches on the same array.

Implementation in Java

```
public static int binarySearch(int[] array, int target) {
    int left = 0;
```

```
        int right = array.length - 1;

        while (left <= right) {
            int middle = left + (right - left) / 2;
            int middleValue = array[middle];

            if (middleValue == target) {
                return middle;
            } else if (target < middleValue) {
                right = middle - 1;
            } else {
                left = middle + 1;
            }
        }

        return -1;  // Not found
    }
```

Note: `middle = left + (right - left) / 2` is used instead of `(left + right) / 2` to avoid overflow on very large numbers.

Java also provides `Arrays.binarySearch()`, but understanding the algorithm is essential.

- **Guiding Questions:**
  - What happens if you use binary search on an unsorted array? Give a specific example.
  - For an array of 1 billion elements, how many comparisons does binary search require in the worst case?
  - Why is `middle = left + (right - left) / 2` preferred over `middle = (left + right) / 2`?
  - What is the time complexity of the entire process: sort + binary search + binary search? Is it worth sorting if you only search once?
- **Code to Review:**
  - Implement binary search from scratch. Test it on an unsorted array, a sorted array, and edge cases (target at beginning, end, not found).
  - Write a program that generates a random sorted array, then times linear search vs. binary search for multiple targets.
  - Manually trace through a binary search algorithm step-by-step, showing `left`, `right`, and `middle` values.

15. Exception Handling & Recovery

In a perfect world, programs run flawlessly. In reality, unexpected things happen: files don't exist, network connections fail, users enter invalid data, the disk is full. Your program must handle these situations gracefully.

Java uses **exceptions** to signal problems. Rather than returning an error code, a method can throw an exception that the caller must handle (or propagate to their caller). This forces you to think about error

cases.

Exceptions vs. Errors

Exceptions: Problems the application can recover from.

- File not found: Open a different file, ask the user, or retry
- Invalid input: Prompt the user for correct input
- Network timeout: Retry the connection

Errors: Serious problems the application likely cannot recover from.

- Out of memory: Cannot allocate more memory
- Stack overflow: Infinite recursion has corrupted the stack

You handle exceptions; you don't handle errors.

Try-Catch Block: The Fundamental Pattern

```
try {
    // Code that might throw an exception
    int[] nums = {1, 2, 3};
    System.out.println(nums[10]);  // ArrayIndexOutOfBoundsException!
} catch (ArrayIndexOutOfBoundsException e) {
    // Handle the specific exception
    System.out.println("Invalid index: " + e.getMessage());
}
```

If an exception occurs in the `try` block, Java stops executing it and jumps to the matching `catch` block. If no exception occurs, the `catch` block is skipped.

Multiple Catch Blocks

Different errors need different handling:

```
try {
    // Code that might throw multiple types of exceptions
} catch (FileNotFoundException e) {
    System.out.println("File not found: " + e.getMessage());
} catch (IOException e) {
    System.out.println("IO error: " + e.getMessage());
} catch (Exception e) {
    // Catch-all for any other exception
    System.out.println("Unexpected error: " + e.getMessage());
}
```

Java checks catch blocks in order. The first match wins. So order matters—put specific exceptions before general ones.

Finally Block: Code That Always Runs

Sometimes you need cleanup code that must run whether an exception occurred or not (closing files, releasing resources):

```
try {
    // Read from file
    FileReader reader = new FileReader("data.txt");
    String data = reader.readLine();
} catch (IOException e) {
    System.out.println("Error: " + e.getMessage());
} finally {
    // This runs regardless
    reader.close();  // Always clean up
}
```

The `finally` block executes whether an exception occurred, was caught, or didn't occur at all.

Exception Object Methods

The exception object passed to `catch` is rich with information:

```
try {
    // ...
} catch (Exception e) {
    e.getMessage();  // Human-readable message
    e.getClass();  // Exception type
    e.printStackTrace();  // Full stack trace for debugging
}
```

Checked vs. Unchecked Exceptions

Checked Exceptions (e.g., `IOException`, `FileNotFoundException`):

- Must be handled (try-catch or throws)
- The compiler forces you to deal with them
- Examples: file I/O, network operations

Unchecked Exceptions (e.g., `ArrayIndexOutOfBoundsException`, `NullPointerException`):

- Handling is optional
- Caused by programming errors (wrong index, null pointer)
- You *should* fix the bug, not just catch the exception

The Throws Keyword: Passing Responsibility

Instead of handling an exception, you can pass responsibility to the caller:

```
public void readFile(String filename) throws IOException {
    FileReader reader = new FileReader(filename);
    // ...
}
```

This tells callers: "I might throw IOException. You must handle it." The caller can use try-catch or throws it further up the chain.

Use `throws` when:

- The method doesn't know how to recover
- Handling should be centralized at a higher level
- You want to reduce duplicate try-catch code

```
public static void main(String[] args) throws IOException {
    readFile("data.txt");  // Responsibility flows to main
}
```

If `main` throws an exception, the program terminates and Java prints the stack trace.

- **Guiding Questions:**
    - What is the difference between an Exception and an Error?
    - If you have multiple catch blocks, does order matter? Why?
    - When is `finally` useful? Give a real-world scenario.
    - What is the difference between a checked and unchecked exception?
    - When should you use `throws` versus try-catch?
- **Code to Review:**
    - Write a try-catch-finally block that reads user input and handles multiple exception types.
    - Create a method that declares `throws IOException` and call it from main using try-catch.
    - Write code that demonstrates a NullPointerException and handle it gracefully.

16. File Input/Output Operations

Programs often need to read data from files and write results to files. Hard-coding data is impractical; file I/O is essential.

File Class: Representing Files and Directories

The `File` class represents a file or directory (not the content—just the path).

```
import java.io.File;

File myFile = new File("data.txt");
File myDir = new File("C:/Users/Student/Documents");
```

Key methods:

- `.exists()` — Returns true if the file exists
- `.isFile()` — Returns true if it's a file (not a directory)
- `.isDirectory()` — Returns true if it's a directory
- `.createNewFile()` — Creates the file; returns true if successful
- `.delete()` — Deletes the file; returns true if successful
- `.mkdir()` — Creates a directory; returns true if successful
- `.listFiles()` — Returns array of files in the directory

Creating Files

```
try {
    File file = new File("output.txt");
    if (file.createNewFile()) {
        System.out.println("File created: " + file.getName());
    } else {
        System.out.println("File already exists");
    }
} catch (IOException e) {
    System.out.println("Error creating file: " + e.getMessage());
}
```

The `.createNewFile()` method throws `IOException` (a checked exception) if something goes wrong (permission denied, invalid path, etc.).

FileWriter: Writing Text to Files

FileWriter writes characters to a file. By default, it **overwrites** existing content:

```
try {
    FileWriter writer = new FileWriter("output.txt");
    writer.write("Hello, World!");
    writer.write("\n");
    writer.write("Second line");
    writer.close();  // Always close when done
} catch (IOException e) {
    e.printStackTrace();
}
```

To **append** instead of overwriting:

```
FileWriter writer = new FileWriter("output.txt", true);  // true = append
mode
```

BufferedWriter: Efficient Writing

For large amounts of text, `BufferedWriter` is faster. It accumulates text in a buffer and writes in chunks:

```
try {
    FileWriter writer = new FileWriter("output.txt");
    BufferedWriter buffered = new BufferedWriter(writer);

    for (int i = 0; i < 1000000; i++) {
        buffered.write("Line " + i);
        buffered.newLine();
    }
    buffered.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

FileReader and BufferedReader: Reading Text

FileReader reads characters from a file:

```
try {
    FileReader reader = new FileReader("input.txt");
    int character;
    while ((character = reader.read()) != -1) {
        System.out.print((char) character);
    }
    reader.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

The `.read()` method returns the next character (as an int), or -1 if the end of file is reached.

For line-based reading, use `BufferedReader`:

```
try {
    FileReader reader = new FileReader("input.txt");
    BufferedReader buffered = new BufferedReader(reader);

    String line;
    while ((line = buffered.readLine()) != null) {
        System.out.println(line);
    }
    buffered.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

The `.readLine()` method returns the next line (excluding newline), or null if the end is reached.

Paths: Relative vs. Absolute

Relative path: Location relative to the current working directory.

```
File file = new File("data.txt");  // In the current directory
File file = new File("files/data.txt");  // In a subdirectory
```

Absolute path: Full path from the root of the file system.

```
File file = new File("C:/Users/Student/Documents/data.txt");  // Windows
File file = new File("/home/student/Documents/data.txt");  // Linux/Mac
```

Creating Directories

```
File dir = new File("myDirectory");
if (dir.mkdir()) {
    System.out.println("Directory created");
}

File file = new File("myDirectory/file.txt");
file.createNewFile();  // Creates file in existing directory
```

IOException: The Checked Exception

Most file operations throw `IOException`, a **checked exception** that must be handled:

```
public void readFile(String filename) throws IOException {
    FileReader reader = new FileReader(filename);  // Throws if file not
found
    // ...
}
```

Common causes:

- File doesn't exist (FileNotFoundException is a subclass of IOException)
- No permission to read/write
- Disk is full
- File is locked by another process

- **Guiding Questions:**
  - What is the difference between FileWriter and BufferedWriter? When would you use each?
  - If you create a FileWriter for a file that already exists, what happens?
  - How do you append to a file instead of overwriting it?
  - What are the differences between relative and absolute paths?
  - Why must you call `.close()` on a file reader/writer?
- **Code to Review:**
  - Write a program that reads from one file and writes to another, converting all text to uppercase.
  - Create a program that appends user input to a file until the user types "quit".
  - Write code that lists all files in a directory using `.listFiles()`.

17. Recursion and Testing with JUnit

Recursion: A Method Calling Itself

Recursion is when a method calls itself to solve a problem by breaking it into smaller subproblems. It's a powerful technique for problems with inherent recursive structure (trees, searching, mathematical sequences).

Every recursive method has two essential parts:

Base case: A condition that stops the recursion.

```
if (n <= 1) return 1;  // Stop here
```

Recursive case: The method calls itself with a simpler input.

```
return n * factorial(n - 1);  // Progress toward base case
```

Example: Factorial

```
public static int factorial(int n) {
    if (n <= 1) {  // Base case
        return 1;
    } else {
        return n * factorial(n - 1);  // Recursive case
    }
}

factorial(5)
= 5 * factorial(4)
= 5 * (4 * factorial(3))
= 5 * (4 * (3 * factorial(2)))
= 5 * (4 * (3 * (2 * factorial(1))))
```

```
= 5 * (4 * (3 * (2 * 1)))
= 120
```

Example: Fibonacci

```
public static int fibonacci(int n) {
    if (n <= 1) {  // Base case
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);  // Recursive case
    }
}
```

Warning: This naive Fibonacci recalculates the same values many times, leading to exponential time complexity. Better to use iteration or memoization.

Common Pitfalls

Missing or wrong base case: Results in infinite recursion (stack overflow).

```
public static int bad(int n) {
    return bad(n - 1);  // No base case—crashes!
}
```

Not progressing toward base case: Recursion doesn't terminate.

```
public static int bad(int n) {
    if (n == 0) return 1;
    return bad(n);  // Still calls itself with same argument—crashes!
}
```

Testing: The Professional Approach

Professional developers don't just hope their code works—they **test** it systematically.

Manual testing (testing by hand) is limited. For a function with many cases, you can't feasibly test them all. **Automated testing** lets you write once and run hundreds of tests in seconds.

JUnit 5: Automated Testing Framework

JUnit is Java's standard testing framework. A test is a method marked with `@Test` that verifies a specific behavior:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {
    @Test
    public void addTwoNumbers_returnsSum() {
        Calculator calc = new Calculator();
        int result = calc.add(2, 3);
        assertEquals(5, result);
    }
}
```

Assert Statements: Verification

Assertions compare the expected result with the actual result:

```
assertEquals(5, 2 + 3);  // Passes: 5 == 5
assertTrue(5 > 0);  // Passes: true is true
assertFalse(5 < 0);  // Passes: false is false
assertNull(obj);  // Passes if obj is null
assertNotNull(obj);  // Passes if obj is not null
```

If an assertion fails, the test fails and shows what went wrong.

Edge Cases: Testing the Boundaries

A good test suite covers not just happy paths but also edge cases—unusual or extreme inputs:

```
@Test
public void divide_handlesDivisionByZero() {
    assertThrows(ArithmeticException.class, () -> {
        Calculator calc = new Calculator();
        calc.divide(10, 0);
    });
}

@Test
public void factorial_handleZero() {
    assertEquals(1, factorial(0));  // Edge case: 0! = 1
}

@Test
public void factorial_handleNegative() {
    // How should negative factorial behave?
}
```

The Ariane 5 rocket exploded in 1996 due to a software overflow bug—a failure to test edge cases. NASA now takes testing extremely seriously.

Test Naming Convention

Name tests clearly so failures are informative:

```
methodUnderTest_expectedBehavior_conditions
```

Examples:

- `add_returnsSum_withPositiveNumbers()`
- `factorial_throws_withNegativeInput()`
- `search_returnsIndex_whenElementExists()`

When a test fails, you immediately know what should have happened.

Testing Collections and File I/O

Testing complex code requires setup and cleanup:

```
@Test
public void ArrayList_removesElementCorrectly() {
    ArrayList<String> list = new ArrayList<>();
    list.add("A");
    list.add("B");
    list.add("C");

    list.remove(1);

    assertEquals(2, list.size());
    assertEquals("A", list.get(0));
    assertEquals("C", list.get(1));
}

@Test
public void fileWriter_createsFile_andWritesContent() throws IOException {
    File file = new File("test.txt");
    FileWriter writer = new FileWriter(file);
    writer.write("Hello");
    writer.close();

    assertTrue(file.exists());

    // Cleanup
    file.delete();
}
```

- **Guiding Questions:**

    - What are the two essential parts of a recursive method?
    - What happens if a recursive method doesn't have a base case?
    - For a problem that can be solved with either recursion or iteration, which is usually better?
    - What is the advantage of automated testing with JUnit over manual testing?
    - What are edge cases, and why is it important to test them?
    - How would you test a method that throws an exception?

- **Code to Review:**

    - Implement factorial recursively and trace through it manually.
    - Write a JUnit test class for a simple Calculator class with multiple test methods.
    - Create tests that verify both normal behavior and edge cases.

Integration & Advanced Questions

As you prepare for the final exam, think about how these topics connect:

- How could you use Collections to store file contents efficiently?
- If an exception occurs while writing to a file in the middle of processing, how do you ensure cleanup?
- How would you test recursive algorithms? What edge cases matter?
- If you need to frequently search a large sorted collection, which Collection type and which search algorithm would you use?
- How could you use HashMap to cache file contents to avoid repeated disk reads?

These integrated questions will likely appear on the exam. Practice thinking holistically about your toolbox.

Interactive Study Application: The Java Competency Crucible

To complement this study guide, you will find a file named `FinalExamStudyGuide.java`. This is an interactive, hands-on, gamified application that tests your readiness for advanced concepts.

First, you will act as the developer by filling in all `// TODO:` sections, which will build code for Collections management, exception handling, and file I/O. After completing the implementation, you can compile and run the application to test your conceptual knowledge through an interactive quiz.

Example Run

After successfully completing all `// TODO:` sections in `FinalExamStudyGuide.java`:

```
==============================================
    JAVA COMPETENCY CRUCIBLE: INITIATED
==============================================

Prove your final exam readiness, developer!

========= ADVANCED MODULES =========
1. Module 1: Collections Challenge (Lists, Sets, Maps)
2. Module 2: Exception & File Handler (Error Handling, I/O)
```

```
3. Module 3: Recursion & Testing Trials (Logic & JUnit Concepts)
4. Module 4: Integration Gauntlet (Bring It All Together)
5. Exit Crucible
==================================
Select your advanced module (1-5): >> 1

--- Module 1: Collections Challenge ---
Q1: Which Collections Framework type stores unique elements with no
guaranteed order?
>> HashSet
...Access Granted. HashSet is correct. [+1 XP]

Q2: What does ArrayList provide that regular Java arrays do not?
>> dynamic sizing
...Access Granted. ArrayList dynamically resizes as needed. [+1 XP]

Q3: What is the average time complexity of a HashMap lookup operation?
>> O(1)
...Access Granted. HashMap lookups are O(1) on average. [+1 XP]

========= ADVANCED MODULES =========
1. Module 1: Collections Challenge (Lists, Sets, Maps)
2. Module 2: Exception & File Handler (Error Handling, I/O)
3. Module 3: Recursion & Testing Trials (Logic & JUnit Concepts)
4. Module 4: Integration Gauntlet (Bring It All Together)
5. Exit Crucible
==================================
Select your advanced module (1-5): >> 2

--- Module 2: Exception & File Handler ---
Q1: What keyword allows you to pass exception responsibility to the
caller?
>> throws
...Access Granted. The 'throws' keyword delegates exception handling. [+1
XP]

Q2: Which class efficiently reads text files line by line?
>> BufferedReader
...Access Granted. BufferedReader reads entire lines efficiently. [+1 XP]

Q3: What block always executes, regardless of whether an exception occurs?
>> finally
...Access Granted. The 'finally' block always executes. [+1 XP]

========= ADVANCED MODULES =========
1. Module 1: Collections Challenge (Lists, Sets, Maps)
2. Module 2: Exception & File Handler (Error Handling, I/O)
3. Module 3: Recursion & Testing Trials (Logic & JUnit Concepts)
4. Module 4: Integration Gauntlet (Bring It All Together)
5. Exit Crucible
==================================
Select your advanced module (1-5): >> 3

--- Module 3: Recursion & Testing Trials ---
```

```
Q1: What condition stops a recursive function from calling itself
infinitely?
>> base case
...Access Granted. The base case prevents infinite recursion. [+1 XP]

Q2: What annotation marks a method as a JUnit test?
>> @Test
...Access Granted. @Test indicates a test method. [+1 XP]

Q3: Which assertion verifies that an exception is thrown?
>> assertThrows
...Access Granted. assertThrows validates exception behavior. [+1 XP]

========= ADVANCED MODULES =========
1. Module 1: Collections Challenge (Lists, Sets, Maps)
2. Module 2: Exception & File Handler (Error Handling, I/O)
3. Module 3: Recursion & Testing Trials (Logic & JUnit Concepts)
4. Module 4: Integration Gauntlet (Bring It All Together)
5. Exit Crucible
====================================
Select your advanced module (1-5): >> 4

--- Module 4: Integration Gauntlet ---
This module demonstrates real-world programming combining multiple
advanced concepts.
Press Enter to initiate integration simulation...>>

[SIM] Initializing Collections and utilities...
...[OK] ArrayList, HashSet, and HashMap initialized and populated.
[SIM] Demonstrating exception handling...
...[OK] Exceptions caught and handled gracefully.
[SIM] Executing file operations...
...[OK] Data successfully written to file and verified.
[SIM] Running recursive algorithms...
...[OK] Recursive utility methods executed correctly.

[SUCCESS] Integration simulation complete. [+4 XP]

Q1: When should you use LinkedList over ArrayList?
>> when frequently inserting or deleting in the middle
...Intel Confirmed. LinkedList excels at middle operations. [+1 XP]

Q2: What is the primary purpose of the base case in recursion?
>> to prevent infinite recursion
...Intel Confirmed. The base case stops recursive calls. [+1 XP]

========= ADVANCED MODULES =========
1. Module 1: Collections Challenge (Lists, Sets, Maps)
2. Module 2: Exception & File Handler (Error Handling, I/O)
3. Module 3: Recursion & Testing Trials (Logic & JUnit Concepts)
4. Module 4: Integration Gauntlet (Bring It All Together)
5. Exit Crucible
====================================
Select your advanced module (1-5): >> 5
```

```
--- CRUCIBLE COMPLETE ---
Final Competency Score: 16 / 16 XP
STATUS: Expert Ready! You are exceptionally prepared for the final exam.
```