Logic & Data Structures

Review Notebook

Victor Lopes Dos Santos

Contents

Chapter 1 Basic Programming Concepts

Section 1.1: What is Programming?

Programming is the process of designing and building executable computer programs to accomplish specific tasks. It consists of translating real-world problems into precise instructions that a computer can understand and execute.

At its core, programming involves:

- **Problem decomposition**: Breaking a complex problem into smaller, manageable parts.
- **Algorithm design**: Defining a step-by-step procedure to solve the problem.
- **Implementation**: Translating the algorithm into a programming language.
- **Testing and debugging**: Verifying correctness and fixing errors.

Programming requires logical reasoning, abstraction, and attention to detail. A well-written program should not only work correctly but also be readable, efficient, and maintainable.

Section 1.2: Compilation vs Interpretation

Programming languages differ in how their source code is transformed into executable instructions. The two main approaches are compilation and interpretation.

1.2.1 Compiled Languages

In compiled languages, the entire source code is translated into machine code before execution.

- **Process**: Source code → Compiler → Machine code → Execution
- **Examples**: C, C++, Rust, Go, Java (partially)
- **Advantages**:
  - Faster execution
  - Better performance optimizations
  - Errors detected before execution
- **Disadvantages**:
  - Platform-dependent binaries
  - Longer compilation time

1.2.2 Interpreted Languages

Interpreted languages execute code through an interpreter at runtime. The interpreter reads and executes the source code line by line or in small chunks without a separate compilation step.

- **Process**: Source code → Interpreter → Direct execution
- **Examples**: Python, JavaScript, Ruby, PHP
- **Advantages**:
  - Platform independence (run on any system with the interpreter)
  - Faster development cycle (no compilation needed)
  - Dynamic features (runtime code modification, dynamic typing)
  - Easier debugging (immediate error feedback)
- **Disadvantages**:
  - Slower execution (interpretation overhead)
  - Errors detected only at runtime
  - Less optimization opportunities

Many modern languages like Python and JavaScript use hybrid approaches with Just-In-Time (JIT) compilation to improve performance.

1.2.3 Java's Hybrid Approach

Java adopts a hybrid compilation model, combining advantages of both approaches.

Source code (.java) $\rightarrow$ Compiler (javac) $\rightarrow$ Bytecode (.class) $\rightarrow$ JVM (java) $\rightarrow$ Machine-specific execution

The source code is first compiled into an intermediate representation called *bytecode*. This bytecode is then executed by the Java Virtual Machine (JVM), which translates it into machine code specific to the underlying hardware.

Section 1.3: Memory Concepts

Understanding how memory is organized is fundamental to writing efficient and correct programs.

1.3.1 Memory Organization

A running program typically uses multiple memory regions, each serving a specific purpose:

- **Stack**: Stores function calls, parameters, and local variables. Operates in a Last-In-First-Out (LIFO) manner.
- **Heap**: Stores dynamically allocated memory, such as objects and data structures created at runtime.
- **Static/Global**: Holds global variables and static fields whose lifetime spans the entire program execution.
- **Code/Text**: Contains the compiled program instructions.

1.3.2 Variables in Memory

Different types of variables are stored in different memory regions:

```
int x = 10;         // Stack: stores value 10
double y = 3.14;    // Stack: stores 3.14
String s = "Hello"; // Stack: reference, Heap: string object
```

Primitive types usually store their values directly, while reference types store memory addresses pointing to objects in the heap.

Section 1.4: Syntax, Semantics, and Pragmatics

Programming languages can be analyzed from three complementary perspectives.

1.4.1 Syntax

Syntax defines the formal rules that determine which sequences of symbols form valid programs.

```
// Valid syntax
int x = 10;
if (x > 5) { ... }

// Invalid syntax
int x = 10      // Missing semicolon
if x > 5        // Missing parentheses
```

Syntax errors are detected by the compiler or interpreter before execution.

1.4.2 Semantics

Semantics refers to the meaning of syntactically valid programs. A program can be syntactically correct but semantically incorrect if it doesn't produce the intended behavior.

```
// Semantically correct
int x = 10;
int y = x + 5;  // y receives the value 15

// Semantically incorrect examples:
int a = 10;
int b = 0;
int c = a / b;  // Division by zero - runtime error

String str = null;
int length = str.length();  // NullPointerException

boolean flag = true;
if (flag = false) {  // Assignment instead of comparison
    // This block will never execute
}
```

1.4.3 Pragmatics

Pragmatics concerns how programs are written and used in practice, focusing on practical aspects beyond formal correctness.

- **Readability and clarity**: Using meaningful names, consistent formatting
- **Maintainability**: Writing modular, well-documented code
- **Performance**: Considering time and space efficiency
- **Security**: Protecting against vulnerabilities
- **Team collaboration**: Following coding standards and conventions

Section 1.5: Basic Program Structure

1.5.1 Minimum Java Program

Every Java application must define a class containing a `main` method, which serves as the program entry point.

```
public class ProgramName {
    public static void main(String[] args) {
        // Program execution starts here
    }
}
```

1.5.2 Key Components

- `class`: Defines a blueprint for objects.
- `main`: Entry point of the program.
- `public`: Makes the method accessible to the JVM.
- `static`: Allows execution without creating an object.
- `void`: Indicates no return value.
- `String[] args`: Receives command-line arguments.

Section 1.6: Development Tools

1.6.1 Integrated Development Environment (IDE)

An IDE provides comprehensive tools to assist software development:

- Code editor with syntax highlighting and autocompletion
- Compiler and interpreter integration
- Debugging tools with breakpoints and variable inspection
- Version control support (Git integration)
- Project and dependency management **??**esting frameworks integration

Popular Java IDEs include IntelliJ IDEA, Eclipse, and NetBeans.

1.6.2 Command Line Interface (CLI)

The Command Line Interface (CLI) is a text-based interface where users type commands to perform tasks. It's a powerful tool for developers because it allows precise control and automation of tasks. The basic CLI usage are:

- Commands are typed in a terminal or command prompt
- Each command performs a specific action
- Commands can be combined and automated

Java CLI Tools

Java includes several command-line tools for development:

```
# Compile Java source code
javac Program.java

# Run compiled Java program
java Program

# Create documentation
javadoc Program.java

# Package into JAR file
jar cvf program.jar *.class
```

Learning to use the CLI effectively can make you a more efficient developer, especially when working with servers, automation, or complex build processes.

Section 1.7: Data Types and Type Systems

A **data type** specifies the kind of values that can be stored and the operations that can be performed on them.

Common data type categories include:

- **Primitive types**: integers, floating-point numbers, characters, booleans
- **Composite types**: arrays, records, structures
- **Reference types**: objects, strings, collections

A **type system** enforces rules governing how different types interact, helping detect errors and improve program reliability.

1.7.1 Static vs Dynamic Typing

- **Static typing**: Types are checked at compile time (e.g., Java, C++).
- **Dynamic typing**: Types are checked at runtime (e.g., Python, JavaScript).

Section 1.8: Variables, Constants, and Scope

A **variable** is a named memory location whose value may change during execution. A **constant** represents a fixed value that cannot be altered after initialization.

```
final int MAX = 100;
int counter = 0;
```

1.8.1 Scope and Lifetime

Scope defines where a variable is accessible, while lifetime determines how long it exists in memory.

- **Local scope**: Variables declared inside methods or blocks.
- **Class or global scope**: Variables accessible throughout a class or program.

Section 1.9: Expressions and Operators

An **expression** combines variables, literals, and operators to produce a value.

```
int result = (a + b) * c;
```

Operators are commonly classified as:

- Arithmetic operators: `+ - * / %`
- Relational operators: `== != > < >= <=`
- Logical operators: `&& || !`
- Assignment operators: `= += -= *=`

Operator precedence and associativity determine the order of evaluation in expressions.

Section 1.10: Control Structures

Control structures determine the flow of execution in a program by allowing selective and repetitive execution of code blocks.

1.10.1 Sequential Execution

Instructions are executed in the order they appear unless altered by control structures.

1.10.2 Conditional Statements

Conditional statements allow decision-making based on logical expressions.

```
if (x > 0) {
    // positive
} else if (x < 0) {
    // negative
} else {
    // zero
}
```

1.10.3 Repetition Structures

Loops allow repeated execution of code blocks.

```
for (int i = 0; i < 10; i++) {
    // repeated 10 times
}

while (condition) {
    // repeated while condition is true
}

do {
    // executed at least once
} while (condition);
```

Section 1.11: Functions and Methods

Functions (or methods) encapsulate reusable units of behavior and promote modularity.

```
int sum(int a, int b) {
    return a + b;
}
```

Functions improve:

- Code reuse
- Abstraction
- Program organization
- Testability

Section 1.12: Input and Output

Input and Output (I/O) mechanisms enable programs to interact with users and external systems.

```
Scanner sc = new Scanner(System.in);
int value = sc.nextInt();
System.out.println(value);
```

I/O operations may involve:

- Standard input and output
- Files
- External devices or networks

Section 1.13: Language Idioms

Programming idioms are recurring patterns of code that represent the most natural, efficient, and readable way to accomplish common tasks within a specific programming language. These patterns emerge from the language's design philosophy, syntax characteristics, and community conventions over time.

1.13.1 Characteristics of Language Idioms

- **Expressiveness**: Idioms allow complex operations to be written concisely while maintaining clarity
- **Performance**: Idiomatic code often leverages language optimizations and avoids common pitfalls
- **Maintainability**: Following established idioms makes code more predictable for other developers
- **Cultural Alignment**: Idioms reflect the programming community's shared understanding and best practices

1.13.2 Java Idioms Examples

```java
// Enhanced for loop (for-each) for collections
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
for (String name : names) {
    System.out.println(name);
}

// Try-with-resources for automatic resource management
try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {
    String line = reader.readLine();
}

// Builder pattern for complex object creation
```

```
Person person = new Person.Builder()
    .name("John")
    .age(30)
    .address("123 Main St")
    .build();

// Factory methods for immutable collections
List<String> list = List.of("a", "b", "c");
Set<Integer> set = Set.of(1, 2, 3);

// Optional for null-safe value handling
Optional<String> optional = Optional.ofNullable(getString());
String result = optional.orElse("default");
```

Section 1.14: Programming Paradigms

A programming paradigm is a fundamental style of computer programming that defines how programmers structure and organize code. Paradigms provide different approaches to problem decomposition, data organization, and control flow management.

1.14.1 Imperative Programming

- **Philosophy**: Focuses on *how* to achieve results through explicit commands
- **Characteristics**: Sequence of statements that change program state
- **Key Concepts**: Variables, assignment, loops, conditionals
- **Example Languages**: C, Pascal, early versions of BASIC

```
// Imperative approach to sum an array
int sum = 0;
for (int i = 0; i < numbers.length; i++) {
    sum += numbers[i];
}
```

1.14.2 Procedural Programming

- **Philosophy**: Extends imperative programming with procedures/functions
- **Characteristics**: Programs organized into reusable procedures
- **Key Concepts**: Functions, parameters, local variables, modularity

- **Example Languages**: C, Pascal, Fortran

```
// Procedural approach with functions
public int calculateSum(int[] numbers) {
    int sum = 0;
    for (int num : numbers) {
        sum += num;
    }
    return sum;
}

public double calculateAverage(int[] numbers) {
    return calculateSum(numbers) / (double) numbers.length;
}
```

1.14.3 Object-Oriented Programming (OOP)

- **Philosophy**: Models real-world entities as objects with state and behavior
- **Characteristics**: Encapsulation, inheritance, polymorphism, abstraction
- **Key Concepts**: Classes, objects, methods, interfaces, inheritance
- **Example Languages**: Java, C++, Python, C#

```
// OOP approach with classes and objects
public class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public double getBalance() {
        return balance;
    }
}

// Usage
BankAccount account = new BankAccount(1000);
account.deposit(500);
```

1.14.4 Functional Programming

- **Philosophy**: Treats computation as evaluation of mathematical functions
- **Characteristics**: Immutable data, first-class functions, no side effects
- **Key Concepts**: Pure functions, higher-order functions, recursion
- **Example Languages**: Haskell, Lisp, Scala, JavaScript (supports)

```
// Functional approach in Java (using Streams)
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
                .filter(n -> n % 2 == 0)      // Keep even numbers
                .map(n -> n * 2)              // Double them
                .reduce(0, Integer::sum);     // Sum all values
```

1.14.5 Multi-Paradigm Languages

Most modern programming languages support multiple paradigms, allowing developers to choose the most appropriate approach for each problem:

- **Java**: Primarily OOP, but supports imperative, procedural, and functional (via Streams)
- **Python**: Supports OOP, imperative, procedural, and functional paradigms
- **JavaScript**: Supports OOP, imperative, procedural, and functional paradigms
- **Scala**: Blends OOP and functional programming seamlessly
- **C++**: Supports OOP, imperative, procedural, and generic programming

Chapter 2
Programming Concepts

Section 2.1: Variables and Data Types

2.1.1 Variable Declaration and Scope

```
// Local variables (method scope)
public void method() {
    int localVar = 10;  // Only accessible within method
}

// Instance variables (object scope)
public class MyClass {
    private int instanceVar;  // Each object has its own copy
}

// Class variables (class scope)
public class MyClass {
    private static int classVar;  // Shared by all objects
}
```

2.1.2 Data Type Hierarchy

Java data types are organized in a hierarchical structure:

- **Primitive Types**
  - **Integer types**: byte (8-bit), short (16-bit), int (32-bit), long (64-bit)
  - **Floating-point**: float (32-bit), double (64-bit)
  - **Character**: char (16-bit Unicode)
  - **Boolean**: boolean (true/false)
- **Reference Types**

- **Classes**: String, Object, custom classes
- **Arrays**: Single and multi-dimensional
- **Interfaces**: Runnable, Comparable, List
- **Enumerations**: Enum types

2.1.3 Type Conversion and Casting

```
// Implicit conversion (widening) - no data loss
int i = 100;
long l = i;           // OK: int to long
double d = i;         // OK: int to double

// Explicit casting (narrowing) - potential data loss
double price = 19.99;
int intPrice = (int) price;  // 19 (truncation)
long big = 10000000000L;
int small = (int) big;       // Possible overflow

// Special cases
char c = 'A';
int ascii = c;               // 65 (ASCII value)
boolean flag = true;
// int num = (int) flag;     // ERROR: cannot cast boolean to int

// String conversions
int num = Integer.parseInt("123");
String str = String.valueOf(456);
```

Section 2.2: Expressions and Statements

2.2.1 Expression Evaluation

```
// Complex expression with precedence
int result = (a + b) * c / d % e;

// Multiple assignments
int x = 5, y = 10, z = 15;

// Expression with side effects
int count = 0;
int value = ++count * 2;  // count=1, value=2
```

```
// Compound assignment
x += 5;    // Equivalent to x = x + 5
x *= y;    // Equivalent to x = x * y
```

2.2.2 Statement Types

- **Declaration**: `int x;`
- **Assignment**: `x = 10;`
- **Method call**: `System.out.println("Hello");`
- **Control flow**: `if, for, while, switch, etc.`
- **Block**: `{ statements }`
- **Return**: `return value;`
- **Break/Continue**: Alter loop execution

Review Questions: Chapter 2

1. Explain the different variable scopes in Java with examples. When would you use each?
2. Compare primitive types and reference types. What are the implications for memory and performance?
3. Given the expression: `int x = 5 + 3 * 2 / (4 - 2);` What is the value of x? Show the step-by-step evaluation.
4. Write a program that demonstrates all compound assignment operators and show the results.
5. What is type casting? When is implicit casting allowed and when is explicit casting required?
6. Write code that converts between: int and double, char and int, boolean and String.

Chapter 3
Decision Structures

Section 3.1: Conditional Logic

3.1.1 Boolean Algebra Fundamentals

- **Identity Laws**: $A \wedge true = A$, $A \vee false = A$
- **Domination Laws**: $A \wedge false = false$, $A \vee true = true$
- **Idempotent Laws**: $A \wedge A = A$, $A \vee A = A$
- **Double Negation**: $\neg(\neg A) = A$
- **De Morgan's Laws**: $\neg(A \wedge B) = \neg A \vee \neg B$, $\neg(A \vee B) = \neg A \wedge \neg B$

3.1.2 Complex Conditional Expressions

```
// Multiple conditions with proper grouping
if ((age >= 18 && hasLicense) || (age >= 16 && withParent)) {
    // Can drive
}

// Nested conditions simplified
boolean canVote = age >= 18 && isCitizen && !hasFelony;
boolean canDrink = age >= 21 && !isSoberDriveSuspended;
```

Section 3.2: If-Else Structures

3.2.1 Chained If-Else-If

```
if (score >= 90) {
    grade = 'A';
} else if (score >= 80) {
    grade = 'B';
} else if (score >= 70) {
    grade = 'C';
} else if (score >= 60) {
    grade = 'D';
} else {
    grade = 'F';
}

// Equivalent using ternary (less readable)
grade = (score >= 90) ? 'A' :
        (score >= 80) ? 'B' :
        (score >= 70) ? 'C' :
        (score >= 60) ? 'D' : 'F';
```

3.2.2 Switch Statement Details

```
// Traditional switch (pre-Java 14)
switch (month) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        days = 31;
        break;
    case 4: case 6: case 9: case 11:
        days = 30;
        break;
    case 2:
        days = isLeapYear ? 29 : 28;
        break;
    default:
        throw new IllegalArgumentException("Invalid month");
}

// Switch expression (Java 14+)
String season = switch (month) {
    case 12, 1, 2 -> "Winter";
    case 3, 4, 5 -> "Spring";
    case 6, 7, 8 -> "Summer";
    case 9, 10, 11 -> "Fall";
    default -> {
        System.err.println("Invalid month: " + month);
        yield "Invalid";
    }
};
```

```
// Switch with pattern matching (Java 21 preview)
String description = switch (obj) {
    case Integer i -> "Integer: " + i;
    case String s -> "String: " + s;
    case null -> "Null object";
    default -> "Unknown type";
};
```

Section 3.3: Pattern Matching

Pattern matching simplifies common coding patterns by combining type checking and variable assignment.

3.3.1 Type Patterns (Java 16+)

```
Object obj = "Hello";

// Traditional approach
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.length());
}

// Pattern matching
if (obj instanceof String s) {
    System.out.println(s.length());  // s is automatically cast
    // s is in scope here
}

// Can be used in while loops
while (obj instanceof String s && s.length() > 0) {
    System.out.println(s.charAt(0));
    obj = s.substring(1);
}
```

Review Questions: Chapter 3

1. Simplify the boolean expression: `!(a && b) || (!a && !b) || (a && !b)`

2. Given three numbers, write code to find the maximum without using Math.max().
3. Explain the difference between deep and shallow equality for objects. How does this apply to Strings?
4. Write a program that validates a date (day, month, year) considering leap years.

Chapter 4
Repetition Structures

Section 4.1: Loop Patterns

Common loop patterns provide reusable templates for solving recurring problems.

4.1.1 Accumulator Pattern

Used to accumulate values through iteration:

```
// Sum of array elements
int[] numbers = {1, 2, 3, 4, 5};
int sum = 0;

for (int num : numbers) {
    sum += num;  // Accumulate
}
System.out.println("Sum: " + sum);

// Product accumulation
double[] values = {1.5, 2.0, 3.5, 4.0};
double product = 1.0;
for (double val : values) {
    product *= val;
}

// String concatenation accumulator
String[] words = {"Hello", " ", "World", "!"};
StringBuilder message = new StringBuilder();
for (String word : words) {
    message.append(word);
}
```

4.1.2 Search Pattern

Used to find elements matching specific criteria:

```
// Linear search
int[] array = {10, 20, 30, 40, 50};
int target = 30;
boolean found = false;
int index = -1;

for (int i = 0; i < array.length; i++) {
    if (array[i] == target) {
        found = true;
        index = i;
        break;  // Early exit
    }
}

// Find maximum value
int max = Integer.MIN_VALUE;
for (int value : array) {
    if (value > max) {
        max = value;
    }
}

// Find all matches
List<Integer> matches = new ArrayList<>();
for (int value : array) {
    if (value > 25) {
        matches.add(value);
    }
}
```

4.1.3 Count-Controlled vs Event-Controlled Loops

```
// Count-controlled (definite iteration) - known number of iterations
for (int i = 0; i < 10; i++) {
    System.out.println("Iteration: " + i);
}

// Event-controlled (indefinite iteration) - depends on conditions
Scanner scanner = new Scanner(System.in);
String input;
do {
    System.out.print("Enter 'quit' to exit: ");
    input = scanner.nextLine();
    // Process input
} while (!input.equalsIgnoreCase("quit"));

// While loop for reading until sentinel value
int sum = 0;
int value;
while ((value = scanner.nextInt()) != -1) {
    sum += value;
}
```

Section 4.2: Nested Loops

4.2.1 Multiplication Table

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.printf("%4d", i * j);
    }
    System.out.println();  // New line after each row
}

// Output:
//    1   2   3   4   5   6   7   8   9  10
//    2   4   6   8  10  12  14  16  18  20
//    ... etc
```

4.2.2 Pattern Generation

```
// Right triangle pattern
int n = 5;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print("* ");
    }
    System.out.println();
}

// Output:
// *
// * *
// * * *
// * * * *
// * * * * *

// Pyramid pattern
for (int i = 1; i <= n; i++) {
    // Print spaces
    for (int j = 1; j <= n - i; j++) {
        System.out.print("  ");
    }
    // Print stars
    for (int j = 1; j <= 2 * i - 1; j++) {
        System.out.print("* ");
    }
    System.out.println();
}
```

Section 4.3: Loop Optimization

4.3.1 Loop Invariant Code Motion

Move calculations that don't change inside the loop to outside:

```
// BEFORE: Calculation inside loop
for (int i = 0; i < array.length; i++) {
    double result = Math.PI * radius * array[i];  // Math.PI * radius is invariant
}

// AFTER: Move invariant outside loop
double constant = Math.PI * radius;
for (int i = 0; i < array.length; i++) {
    double result = constant * array[i];  // Faster
}

// BEFORE: Method call in loop condition
for (int i = 0; i < list.size(); i++) {  // size() called each iteration
    // ...
}

// AFTER: Cache size
int size = list.size();
for (int i = 0; i < size; i++) {
    // ...
}
```

4.3.2 Loop Unrolling

Reduce loop overhead by processing multiple elements per iteration:

```
// Standard loop
int sum = 0;
for (int i = 0; i < 100; i++) {
    sum += array[i];
}

// Partially unrolled (process 4 elements per iteration)
int sum = 0;
for (int i = 0; i < 100; i += 4) {
    sum += array[i] + array[i+1] + array[i+2] + array[i+3];
}
// Handle remaining elements (if any)
for (int i = 96; i < 100; i++) {
    sum += array[i];
}
```

Section 4.4: Recursion vs Iteration

4.4.1 Factorial Comparison

```
// Iterative factorial (preferred in Java)
int factorialIterative(int n) {
    if (n < 0) throw new IllegalArgumentException();
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}

// Recursive factorial (simpler but less efficient)
int factorialRecursive(int n) {
    if (n < 0) throw new IllegalArgumentException();
    if (n <= 1) return 1;
    return n * factorialRecursive(n - 1);
}

// Tail-recursive factorial (Java doesn't optimize tail calls)
int factorialTail(int n, int accumulator) {
    if (n <= 1) return accumulator;
    return factorialTail(n - 1, n * accumulator);
}
// Wrapper method
int factorial(int n) {
    return factorialTail(n, 1);
}
```

4.4.2 When to Use Recursion

- **Use recursion for**:
  - Tree/graph traversals
  - Divide-and-conquer algorithms
  - Problems with recursive mathematical definitions
- **Use iteration for**:
  - Simple linear processing
  - Performance-critical code
  - Deep recursion that might cause stack overflow

Review Questions: Chapter 4

1. Compare and contrast for, while, and do-while loops. When should each be used?
2. Write a program that prints the first 20 Fibonacci numbers using three different loop types.
3. Implement bubble sort and analyze its time complexity in terms of Big O notation.
4. Write a program that finds all prime numbers between 1 and 100 using the Sieve of Eratosthenes algorithm.
5. Explain loop optimization techniques with concrete examples.
6. Compare iterative and recursive approaches for calculating factorial. Which is more efficient and why?
7. Write a program that reverses an array in-place (without creating a new array).
8. Implement binary search and compare its efficiency with linear search.

Chapter 5

Basic Data Structures

Section 5.1: Arrays

5.1.1 Array Declaration Variations

```
// One-dimensional arrays
int[] arr1 = new int[5];           // All zeros [0,0,0,0,0]
int[] arr2 = {1, 2, 3, 4, 5};     // Initialized
int[] arr3 = new int[]{1, 2, 3};  // Alternative syntax

// Multi-dimensional arrays
int[][] matrix1 = new int[3][4];   // 3x4 matrix (all zeros)
int[][] matrix2 = {{1,2}, {3,4}}; // 2x2 matrix
int[][] jagged = new int[3][];     // Jagged array (ragged)
jagged[0] = new int[2];            // First row has 2 columns
jagged[1] = new int[3];            // Second row has 3 columns
jagged[2] = new int[1];            // Third row has 1 column

// Array of objects
String[] names = new String[3];
names[0] = "Alice";
names[1] = "Bob";
names[2] = "Charlie";
```

5.1.2 Array Algorithms

Bubble Sort

```
void bubbleSort(int[] arr) {
    int n = arr.length;
    boolean swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        // If no swaps, array is sorted
        if (!swapped) break;
    }
}
```

Binary Search

```
int binarySearch(int[] arr, int target) {
    int left = 0;
    int right = arr.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;  // Avoid overflow

        if (arr[mid] == target) {
            return mid;  // Found
        } else if (arr[mid] < target) {
            left = mid + 1;  // Search right half
        } else {
            right = mid - 1;  // Search left half
        }
    }
    return -1;  // Not found
}

// Recursive version
int binarySearchRecursive(int[] arr, int target, int left, int right) {
    if (left > right) return -1;

    int mid = left + (right - left) / 2;
    if (arr[mid] == target) return mid;
    if (arr[mid] < target)
        return binarySearchRecursive(arr, target, mid + 1, right);
    else
```

```
        return binarySearchRecursive(arr, target, left, mid - 1);
}
```

5.1.3 Array Utilities

```
import java.util.Arrays;
import java.util.Collections;

int[] arr = {5, 2, 8, 1, 9};

// Sorting
Arrays.sort(arr);                  // Ascending: {1, 2, 5, 8, 9}
Arrays.sort(arr, 1, 4);           // Sort subset [1,4)

// Searching (array must be sorted)
int index = Arrays.binarySearch(arr, 5);  // 2

// Filling
Arrays.fill(arr, 0);              // {0, 0, 0, 0, 0}
Arrays.fill(arr, 1, 3, 99);       // Fill positions 1-2 with 99

// Copying
int[] copy = Arrays.copyOf(arr, arr.length);
int[] range = Arrays.copyOfRange(arr, 1, 4);  // Elements 1-3

// Comparing
boolean equal = Arrays.equals(arr1, arr2);
int compare = Arrays.compare(arr1, arr2);  // Lexicographic

// Streaming
int sum = Arrays.stream(arr).sum();
double avg = Arrays.stream(arr).average().orElse(0);
```

Section 5.2: ArrayLists (Introduction)

5.2.1 Basic Operations

```
import java.util.ArrayList;
import java.util.Collections;

ArrayList<Integer> list = new ArrayList<>();
```

```
// Adding elements
list.add(10);                    // [10]
list.add(20);                    // [10, 20]
list.add(1, 15);                 // [10, 15, 20] - insert at index 1
list.addAll(Arrays.asList(30, 40)); // [10, 15, 20, 30, 40]

// Accessing elements
int first = list.get(0);         // 10
int size = list.size();          // 5
boolean contains = list.contains(20); // true
int index = list.indexOf(20);    // 2
int lastIndex = list.lastIndexOf(20); // 2

// Modifying elements
list.set(1, 25);                 // [10, 25, 20, 30, 40]
list.replaceAll(x -> x * 2);     // [20, 50, 40, 60, 80]

// Removing elements
list.remove(0);                  // Remove first: [50, 40, 60, 80]
list.remove(Integer.valueOf(40)); // Remove by value: [50, 60, 80]
list.removeIf(x -> x > 70);      // Remove if >70: [50, 60]
list.clear();                    // Empty list

// Iterating
for (int num : list) {
    System.out.println(num);
}

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

list.forEach(System.out::println);  // Method reference
```

5.2.2 ArrayList vs Array

Feature	**Array**	**ArrayList**
Size	Fixed at creation	Dynamic (grows/shrinks automatically)
Performance	Faster access (direct indexing)	Slightly slower (bounds checking)
Type safety	Compile-time (for primitives)	Runtime (generics, type erasure)
Memory	Less overhead	More overhead (object wrapper)
Methods	Limited (length field)	Rich API (add, remove, sort, etc.)
Primitives	Direct support	Requires wrapper classes (Integer, etc.)
Multi-dimensional	Supported	Lists of Lists

Section 5.3: Common Data Structure Problems

5.3.1 Two Sum Problem

Find two numbers that sum to a target value:

```
// Brute force approach: O(n²) time, O(1) space
int[] twoSumBruteForce(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[i] + nums[j] == target) {
                return new int[]{i, j};
            }
        }
    }
    return new int[]{-1, -1};
}

// Optimized using HashMap: O(n) time, O(n) space
int[] twoSumHashMap(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[]{map.get(complement), i};
        }
        map.put(nums[i], i);
    }
    return new int[]{-1, -1};
}
```

5.3.2 Maximum Subarray

Find the contiguous subarray with the largest sum (Kadane's Algorithm):

```
// Kadane's Algorithm: O(n) time, O(1) space
int maxSubArray(int[] nums) {
    if (nums.length == 0) return 0;

    int maxSoFar = nums[0];
    int maxEndingHere = nums[0];

    for (int i = 1; i < nums.length; i++) {
        // Either extend the previous subarray or start new
        maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);
        maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }

    return maxSoFar;
}

// Version that also returns the subarray indices
int[] maxSubArrayWithIndices(int[] nums) {
    int maxSoFar = nums[0], maxEndingHere = nums[0];
    int start = 0, end = 0, tempStart = 0;

    for (int i = 1; i < nums.length; i++) {
        if (nums[i] > maxEndingHere + nums[i]) {
            maxEndingHere = nums[i];
            tempStart = i;
        } else {
            maxEndingHere = maxEndingHere + nums[i];
        }

        if (maxEndingHere > maxSoFar) {
            maxSoFar = maxEndingHere;
            start = tempStart;
            end = i;
        }
    }

    return new int[]{maxSoFar, start, end};
}
```

Review Questions: Chapter 5

1. Explain the difference between arrays and ArrayLists. When would you choose one over the other?
2. Implement a method that rotates an array by k positions to the right.
3. Write a program that finds the most frequent element in an array.
4. Implement Kadane’s algorithm to find the maximum subarray sum.
5. Write a program that merges two sorted arrays into a single sorted array.
6. Explain how multi-dimensional arrays are stored in memory.

Chapter 6
String Manipulation

Section 6.1: String Fundamentals

6.1.1 String Creation

```
// Different ways to create strings
String s1 = "Hello";                    // String literal (from string pool)
String s2 = new String("Hello");        // Using constructor (new object)
char[] chars = {'H','e','l','l','o'};
String s3 = new String(chars);          // From char array
String s4 = String.valueOf(123);        // From other types
String s5 = String.format("Value: %d", 42);  // Formatted string

// String concatenation
String concat1 = "Hello" + " " + "World";  // "Hello World"
String concat2 = "Result: " + 42;          // "Result: 42"

// Important: String immutability
String s = "Hello";
s = s + " World";  // Creates new string "Hello World", doesn't modify "Hello"
```

6.1.2 String Pool

Java maintains a string pool to reuse immutable string objects:

```
String s1 = "Hello";        // Goes to string pool
String s2 = "Hello";        // Reuses from pool (s1 == s2 is true)
String s3 = new String("Hello");  // New object (s1 == s3 is false)
String s4 = s3.intern();    // Adds to pool or returns existing

// == compares references, equals() compares content
System.out.println(s1 == s2);        // true (same reference)
System.out.println(s1.equals(s2));   // true (same content)
System.out.println(s1 == s3);        // false (different references)
System.out.println(s1.equals(s3));   // true (same content)
System.out.println(s1 == s4);        // true (s4 is interned)
```

Section 6.2: String Methods

6.2.1 Basic Operations

```
String str = "  Hello World  ";

// Length and emptiness
int len = str.length();           // 15 (includes spaces)
boolean empty = str.isEmpty();    // false
boolean blank = str.isBlank();    // false (Java 11+) - checks if empty or whitespace

// Character access
char first = str.charAt(2);       // 'H' (index 2, after spaces)
char last = str.charAt(str.length() - 1);  // ' ' (last space)

// Substrings
String sub1 = str.substring(2);       // "Hello World  "
String sub2 = str.substring(2, 7);    // "Hello"
String sub3 = str.substring(8, 13);   // "World"

// Case conversion
String upper = str.toUpperCase();    // "  HELLO WORLD  "
String lower = str.toLowerCase();    // "  hello world  "

// Trimming
String trimmed = str.trim();         // "Hello World" (removes leading/trailing spaces)
String strip = str.strip();          // "Hello World" (Java 11+, Unicode-aware)
String stripLeading = str.stripLeading();  // "Hello World  "
String stripTrailing = str.stripTrailing();// "  Hello World"
```

6.2.2 Searching and Comparing

```
String str = "Hello World";

// Searching
int index1 = str.indexOf('o');        // 4 (first 'o')
int index2 = str.indexOf('o', 5);     // 7 (start from index 5)
int index3 = str.indexOf("World");    // 6
int lastIndex = str.lastIndexOf('o'); // 7
int lastIndex2 = str.lastIndexOf('o', 6); // 4 (search backward from index 6)

// Checking content
boolean starts = str.startsWith("Hello");   // true
boolean ends = str.endsWith("World");       // true
boolean contains = str.contains("lo W");    // true

// Comparison
String s1 = "Hello";
String s2 = "hello";
boolean eq1 = s1.equals(s2);           // false (case-sensitive)
boolean eq2 = s1.equalsIgnoreCase(s2); // true (case-insensitive)
int cmp = s1.compareTo(s2);            // -32 (negative, 'H' < 'h')
int cmpIgnore = s1.compareToIgnoreCase(s2); // 0

// Matching patterns
boolean matches = str.matches("^Hello.*");  // true (regex)
```

Section 6.3: String Building

6.3.1 StringBuilder vs StringBuffer

```
// StringBuilder (not thread-safe, faster for single thread)
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" ");
sb.append("World");
sb.insert(5, ",");           // "Hello, World"
sb.replace(5, 6, " ");       // "Hello World"
sb.delete(5, 6);             // "HelloWorld"
sb.reverse();                // "dlroW olleH"
String result1 = sb.toString();

// StringBuffer (thread-safe, synchronized, slower)
StringBuffer buffer = new StringBuffer();
buffer.append("Hello");
buffer.append(" ");
buffer.append("World");
String result2 = buffer.toString();

// Method chaining
```

```
String str = new StringBuilder()
    .append("Hello")
    .append(" ")
    .append("World")
    .reverse()          // "dlroW olleH"
    .toString();
```

6.3.2 Efficiency Considerations

```
// INEFFICIENT: Creates many intermediate strings (O(n²) time)
String result = "";
for (int i = 0; i < 1000; i++) {
    result += i;  // Creates new string each iteration
}

// EFFICIENT: Uses StringBuilder (O(n) time)
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    sb.append(i);
}
String result = sb.toString();

// Pre-sizing StringBuilder for large operations
StringBuilder sb2 = new StringBuilder(10000);  // Initial capacity
for (int i = 0; i < 10000; i++) {
    sb2.append(i);
}
```

Section 6.4: String Formatting

6.4.1 printf-style Formatting

```
String name = "John";
int age = 25;
double salary = 50000.50;

// Format string
String formatted = String.format(
    "Name: %s, Age: %d, Salary: $%,.2f",
    name, age, salary
);
// Result: "Name: John, Age: 25, Salary: $50,000.50"
// Direct printing
System.out.printf("Name: %s, Age: %d%n", name, age);

// Common format specifiers:
// %s - String
// %d - Decimal integer
// %f - Floating point (%.2f for 2 decimal places)
// %c - Character
// %b - Boolean
// %n - New line (platform-independent)
// %% - Percent sign
// %t - Date/time
// %, - Thousands separator

// Padding and alignment
System.out.printf("%-10s %5d%n", "John", 25);   // Left-aligned name
System.out.printf("%10s %5d%n", "Jane", 30);    // Right-aligned
```

Section 6.5: Regular Expressions

6.5.1 Basic Patterns

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

String text = "The price is $19.99 and $29.50. Contact: info@example.com";

// Pattern matching
Pattern pricePattern = Pattern.compile("\\$\\d+\\.\\d{2}");
Matcher priceMatcher = pricePattern.matcher(text);

while (priceMatcher.find()) {
    System.out.println("Found price: " + priceMatcher.group());
}
// Output: Found price: $19.99
//         Found price: $29.50

// Common validation patterns
Pattern email = Pattern.compile("^[\\w.-]+@[\\w.-]+\\.[a-zA-Z]{2,}$");
Pattern phone = Pattern.compile("^\\+?[\\d\\s-]{10,}$");
Pattern date = Pattern.compile("^\\d{4}-\\d{2}-\\d{2}$");
Pattern url = Pattern.compile("^(https?://)?([\\w-]+\\.)+[a-zA-Z]{2,}(/\\S*)?$");
```

```java
// Using String methods with regex
String[] words = text.split("\\s+");  // Split by whitespace
String cleaned = text.replaceAll("\\$", "");  // Remove dollar signs
boolean hasDigits = text.matches(".*\\d+.*");  // Check if contains digits
```

Section 6.6: Practical Applications

6.6.1 Palindrome Check

```java
public boolean isPalindrome(String str) {
    // Remove non-alphanumeric and convert to lowercase
    String clean = str.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();

    // Two-pointer approach
    int left = 0;
    int right = clean.length() - 1;

    while (left < right) {
        if (clean.charAt(left) != clean.charAt(right)) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}

// Alternative using StringBuilder
public boolean isPalindrome2(String str) {
    String clean = str.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();
    String reversed = new StringBuilder(clean).reverse().toString();
    return clean.equals(reversed);
}

// Test cases
isPalindrome("A man, a plan, a canal: Panama");  // true
isPalindrome("race a car");                       // false
isPalindrome(" ");                                // true (empty or whitespace)
```

6.6.2 Word Count

```java
import java.util.HashMap;
import java.util.Map;

public Map<String, Integer> wordCount(String text) {
    Map<String, Integer> countMap = new HashMap<>();

    if (text == null || text.isBlank()) {
        return countMap;
    }

    // Split into words (handle punctuation and multiple spaces)
    String[] words = text.toLowerCase()
                         .replaceAll("[^a-z\\s]", " ")  // Replace non-letters with space
                         .trim()                          // Remove leading/trailing spaces
                         .split("\\s+");                  // Split by one or more spaces

    // Count occurrences
    for (String word : words) {
        countMap.put(word, countMap.getOrDefault(word, 0) + 1);
    }

    return countMap;
}

// Enhanced version with stop words filtering
public Map<String, Integer> wordCountWithFilter(String text, Set<String> stopWords) {
    Map<String, Integer> countMap = new HashMap<>();

    String[] words = text.toLowerCase()
                         .replaceAll("[^a-z\\s]", " ")
                         .trim()
                         .split("\\s+");

    for (String word : words) {
        if (!stopWords.contains(word) && !word.isEmpty()) {
            countMap.put(word, countMap.getOrDefault(word, 0) + 1);
        }
    }

    return countMap;
}
```

Review Questions: Chapter 6

1. Explain string immutability and its implications for performance. How can StringBuilder help?
2. Write a program that checks if two strings are anagrams of each other.
3. Implement a method that compresses a string (e.g., ”aaabbbccc” □ ”a3b3c3”) but returns the original if compression doesn’t help.
4. Write a program that validates and parses an email address using regular expressions.
5. Implement a simple template engine that replaces placeholders like {{name}} with values from a map.
6. Write a program that counts word frequency in a text document.
7. Explain the difference between String, StringBuilder, and StringBuffer.
8. Write a program that formats a phone number from various input formats to a standard format.

Chapter 7
Answers to Review Questions

Section 7.1: Chapter 1 Answers

1. **Compilation vs Interpretation**: Compiled languages are translated to machine code before execution (faster, platform-specific). Interpreted languages are executed line by line (slower, platform-independent). Java uses both: compiled to bytecode, then interpreted by JVM.
2. **Java Process**: Source (.java) □ Compiler (javac) □ Bytecode (.class) □ JVM □ Execution. Hybrid because bytecode is compiled but needs JVM interpretation.
3. **Memory Areas**:
   - Stack: Local variables, method calls
   - Heap: Objects, dynamic allocation
   - Static: Class variables, constants
   - Code: Program instructions
4. **Syntax, Semantics, Pragmatics**:
   - Syntax: `if (x > 0) {}` - correct structure
   - Semantics: `x > 0` means ”x is positive”
   - Pragmatics: Using meaningful variable names, proper formatting
5. **JVM Purpose**: Executes bytecode, provides memory management, security, platform abstraction, garbage collection.
6. **Bytecode**: Intermediate representation between source and machine code, platform-independent, executed by JVM.
7. **Java Program Components**:
   - `class`: Defines object blueprint
   - `main`: Program entry point

- `public`: Access modifier (JVM can call it)
- `static`: Class-level method (no object needed)
- `void`: No return value
- `String[] args`: Command-line arguments array

Section 7.2: Chapter 2 Answers

1. **Variable Scopes**:
   - Local: Inside method/block, accessible only within
   - Instance: Non-static fields, each object has its own copy
   - Class: Static fields, shared across all objects
   - Use instance for object state, class for shared data, local for temporary values
2. **Primitive vs Reference**:
   - Primitives: Store values directly (stack), 8 types, faster
   - References: Store memory addresses (heap), point to objects, can be null
   - Implications: Primitives use less memory, references enable complex structures
3. **Expression Evaluation**: `5 + 3 * 2 / (4 - 2)` = `5 + 3 * 2 / 2` = `5 + 6 / 2` = `5 + 3` = `8`
4. **Compound Assignment**:

   ```
   int x = 10;
   x += 5;   // x = 15
   x -= 3;   // x = 12
   x *= 2;   // x = 24
   x /= 4;   // x = 6
   x %= 5;   // x = 1
   x <<= 2;  // x = 4 (bitwise left shift)
   x >>= 1;  // x = 2 (bitwise right shift)
   ```

5. **Type Casting**:
   - Implicit: Widening conversions (int to double, byte to int)
   - Explicit: Narrowing conversions (double to int, long to short)
   - Required when information loss possible or incompatible types
6. **Type Conversions**:

```
// int to double
int i = 42;
double d = i;           // Implicit

// double to int
double price = 19.99;
int intPrice = (int) price;  // Explicit, truncates to 19

// char to int
char c = 'A';
int ascii = c;                // 65

// boolean to String
boolean flag = true;
String s = String.valueOf(flag);  // "true"
```

Section 7.3: Chapter 3 Answers

1. **Boolean Simplification**: `!(a && b) || (!a && !b) || (a && !b)` Using De Morgan: `(!a || !b) || (!a && !b) || (a && !b)` Distributive: `!a || !b || (!a && !b) || (a && !b)` Simplifies to: `!a || !b`
2. **Find Maximum**:

```
int max = a;
if (b > max) max = b;
if (c > max) max = c;
return max;

// Alternative using ternary
int max = (a > b) ? (a > c ? a : c) : (b > c ? b : c);
```

3. **Equality Types**:
   - Shallow equality: Compares references (==)
   - Deep equality: Compares content (equals() method)
   - Strings: Use equals() for content comparison, == for reference comparison
4. **Date Validation**:

```
boolean isValidDate(int day, int month, int year) {
    if (year < 1 || month < 1 || month > 12) return false;
    int[] daysInMonth = {31,28,31,30,31,30,31,31,30,31,30,31};

    // Leap year check
    boolean isLeapYear = (year % 400 == 0) ||
                        (year % 4 == 0 && year % 100 != 0);
    if (isLeapYear) daysInMonth[1] = 29;

    return day >= 1 && day <= daysInMonth[month-1];
}
```

Section 7.4: Chapter 4 Answers

1. **Loop Comparison**:
   - for: Known iterations, counter-based
   - while: Condition-based, may execute zero times
   - do-while: Condition-based, executes at least once
2. **Fibonacci**:

```
// Using for loop
int a=0, b=1;
System.out.print(a + " " + b + " ");
for(int i=2; i<20; i++) {
    int c = a + b;
    System.out.print(c + " ");
    a = b;
    b = c;
}

// Using while
int a=0, b=1, count=2;
System.out.print(a + " " + b + " ");
while(count < 20) {
    int c = a + b;
    System.out.print(c + " ");
    a = b;
    b = c;
    count++;
}
```

3. **Bubble Sort Analysis**:
   - Time complexity: O(n^2) worst and average case, O(n) best case (already sorted)
   - Space complexity: O(1) in-place
   - Stable: Equal elements maintain relative order
4. **Sieve of Eratosthenes**:

```
boolean[] isPrime = new boolean[101];
Arrays.fill(isPrime, true);
isPrime[0] = isPrime[1] = false;

for(int i=2; i*i<=100; i++) {
    if(isPrime[i]) {
        for(int j=i*i; j<=100; j+=i) {
            isPrime[j] = false;
        }
    }
}

for(int i=2; i<=100; i++) {
    if(isPrime[i]) System.out.print(i + " ");
}
```

5. **Loop Optimization**:
   - Loop invariant code motion: Move calculations outside loop
   - Loop unrolling: Process multiple elements per iteration
   - Strength reduction: Replace expensive operations with cheaper ones
   - Loop fusion: Combine adjacent loops
6. **Factorial Comparison**:
   - Iterative: O(n) time, O(1) space, more efficient
   - Recursive: O(n) time, O(n) space (call stack), simpler code
   - Iterative preferred for performance and stack overflow prevention
7. **Array Reversal**:

```
void reverseArray(int[] arr) {
    int left = 0, right = arr.length-1;
    while(left < right) {
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
```

```
            left++;
            right--;
        }
    }
```

8. **Binary Search vs Linear**:
    - Linear search: O(n) time, works on unsorted arrays
    - Binary search: O(log n) time, requires sorted array
    - Binary search 100x faster for 1,000,000 elements

Section 7.5: Chapter 5 Answers

1. **Arrays vs ArrayLists**:
    - Arrays: Fixed size, better performance, direct primitive support
    - ArrayLists: Dynamic size, rich API, requires object wrappers for primitives
    - Choose arrays for performance-critical code, ArrayLists when size changes frequently
2. **Array Rotation**:

```
void rotateRight(int[] arr, int k) {
    k = k % arr.length;
    reverse(arr, 0, arr.length-1);
    reverse(arr, 0, k-1);
    reverse(arr, k, arr.length-1);
}

void reverse(int[] arr, int start, int end) {
    while(start < end) {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}
```

3. **Most Frequent Element**:

```
int mostFrequent(int[] arr) {
    Map<Integer, Integer> freq = new HashMap<>();
    int maxFreq = 0, mostFreq = arr[0];

    for(int num : arr) {
        freq.put(num, freq.getOrDefault(num, 0) + 1);
        if(freq.get(num) > maxFreq) {
            maxFreq = freq.get(num);
            mostFreq = num;
        }
    }
    return mostFreq;
}
```

4. **Kadane's Algorithm**:

```
int maxSubArray(int[] nums) {
    int maxSoFar = nums[0];
    int maxEndingHere = nums[0];

    for(int i=1; i<nums.length; i++) {
        maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);
        maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }
    return maxSoFar;
}
```

5. **Merge Sorted Arrays**:

```
int[] merge(int[] a, int[] b) {
    int[] result = new int[a.length + b.length];
    int i=0, j=0, k=0;

    while(i < a.length && j < b.length) {
        result[k++] = (a[i] <= b[j]) ? a[i++] : b[j++];
    }

    while(i < a.length) result[k++] = a[i++];
    while(j < b.length) result[k++] = b[j++];

    return result;
}
```

6. **Multi-dimensional Storage**:

- Java uses row-major order: elements of each row stored contiguously
- 2D array is array of arrays: each row can have different length (jagged)
- Memory address calculation: base + (row * cols + col) * elementSize

Section 7.6: Chapter 6 Answers

1. **String Immutability**:
    - Strings cannot be modified after creation
    - Operations create new strings, leading to performance issues
    - StringBuilder provides mutable alternative for concatenation-heavy code
2. **Anagram Check**:

```
boolean isAnagram(String s1, String s2) {
    if(s1.length() != s2.length()) return false;

    char[] c1 = s1.toLowerCase().toCharArray();
    char[] c2 = s2.toLowerCase().toCharArray();

    Arrays.sort(c1);
    Arrays.sort(c2);

    return Arrays.equals(c1, c2);
}
```

3. **String Compression**:

```
String compress(String str) {
    if(str.length() <= 2) return str;

    StringBuilder compressed = new StringBuilder();
    int count = 1;

    for(int i=1; i<str.length(); i++) {
        if(str.charAt(i) == str.charAt(i-1)) {
            count++;
        } else {
            compressed.append(str.charAt(i-1)).append(count);
            count = 1;
        }
    }
```

```java
    compressed.append(str.charAt(str.length()-1)).append(count);

    return compressed.length() < str.length() ?
           compressed.toString() : str;
}
```

4. **Email Validation**:

```java
boolean isValidEmail(String email) {
    return email != null &&
           email.matches("^[\\w.-]+@[\\w.-]+\\.[a-zA-Z]{2,}$");
}
```

5. **Template Engine**:

```java
String renderTemplate(String template, Map<String, String> data) {
    String result = template;
    for(Map.Entry<String, String> entry : data.entrySet()) {
        String placeholder = "{{" + entry.getKey() + "}}";
        result = result.replace(placeholder, entry.getValue());
    }
    return result;
}
```

6. **Word Frequency**:

```java
Map<String, Integer> wordFrequency(String text) {
    Map<String, Integer> freq = new HashMap<>();
    String[] words = text.toLowerCase()
                         .replaceAll("[^a-z\\s]", " ")
                         .trim()
                         .split("\\s+");

    for(String word : words) {
        if(!word.isEmpty()) {
            freq.put(word, freq.getOrDefault(word, 0) + 1);
        }
    }
    return freq;
}
```

7. **String, StringBuilder, StringBuffer**:

- String: Immutable, thread-safe, stored in string pool
- StringBuilder: Mutable, not thread-safe, faster for single thread
- StringBuffer: Mutable, thread-safe (synchronized), slightly slower

8. **Phone Formatter**:

```
String formatPhone(String phone) {
    String digits = phone.replaceAll("\\D", "");

    if(digits.length() == 10) {
        return String.format("(%s) %s-%s",
            digits.substring(0, 3),
            digits.substring(3, 6),
            digits.substring(6));
    } else if(digits.length() == 11 && digits.startsWith("1")) {
        return String.format("+1 (%s) %s-%s",
            digits.substring(1, 4),
            digits.substring(4, 7),
            digits.substring(7));
    }
    return phone; // Return original if format not recognized
}
```