Chapter 04

Programming Principles

Data Structures, Collections & Programming Principles

Mr. SOK Pongsametrey
metreysk@gmail.com

Programming Concepts vs Programming Principles

🧠 **Programming Concepts – *What you learn***

These are the **foundational building blocks** of writing code. Examples include:

- Variables and data types
- Conditionals (if-else)
- Loops (for, while)
- Arrays and collections
- Functions/methods
- Classes and objects

✅ These are practical and technical — what a we *does* in code.

📐 **Programming Principles – *How you think***

These refer to **best practices and philosophies** behind writing good code:

- DRY (Don’t Repeat Yourself)
- KISS (Keep It Simple, Stupid)
- Separation of concerns
- Modularity and readability
- Reusability and maintainability
- SOLID (for OOP design)

✅ These are conceptual and strategic — how we *think* about structuring code.

This week we connect data structures directly to programming principles like modularity, separation of concerns, DRY, and clarity. We'll explore arrays, Java Collections Framework, and dive deeper into Lambda expressions for cleaner, more expressive code with complete working examples.

ABOUT

Why Do We Need Data Structures?

- Organize and manage data clearly (readability)
- Enable reusable logic (DRY)
- Support separation of concerns (data storage vs logic)
- Improve efficiency and scalability
- Real-world analogies:
-  Contacts list in your phone
-  Student grades in a gradebook
-  Inventory tracking in a store
-  Playlist in music apps

Java Arrays - The Foundation

- **Principle tie-in:** Predictable structure improves clarity and testing
- Arrays store elements in fixed size and order. You can access any element by its index. They're efficient but limited.

java

```java
// Declaration and initialization
int[] scores = {90, 80, 85, 95, 78};
String[] names = {"Alice", "Bob", "Charlie"};

// Access elements
System.out.println(scores[0]); // Output: 90
System.out.println("Array length: " + scores.length); // Output: 5
```

Visual Representation:

```
Array Index:  [0] [1] [2] [3] [4]
Array Value:  [90][80][85][95][78]
              ↑               ↑
            First           Last
```

Java Arrays

- Fixed size, same data type
- Index starts at 0
- **Initializing and Iterating Arrays**
  - Common loop pattern to process array contents
- **Limitations of Arrays**
  - Fixed size (can’t add/remove easily)
  - Manual resizing is complex
  - No built-in utility methods

Complete Array Example - Student Grade Manager

```java
public class GradeManager {
    public static void main(String[] args) {
        // Student data
        String[] students = {"Alice", "Bob", "Charlie", "Diana"};
        int[] grades = {85, 92, 78, 96};

        // Display all grades
        System.out.println("=== Student Grades ===");
        for (int i = 0; i < students.length; i++) {
            System.out.println(students[i] + ": " + grades[i]);
        }

        // Calculate average
        double average = calculateAverage(grades);
        System.out.println("\nClass Average: " + average);

        // Find top student
        int topIndex = findTopStudent(grades);
        System.out.println("Top Student: " + students[topIndex] +
                           " with " + grades[topIndex] + " points");
    }

    // Modular method for calculating average (DRY principle)
    public static double calculateAverage(int[] grades) {
        int sum = 0;
        for (int grade : grades) {
            sum += grade;
        }
        return (double) sum / grades.length;
    }

    // Modular method for finding top student (Separation of concerns)
    public static int findTopStudent(int[] grades) {
        int maxIndex = 0;
        for (int i = 1; i < grades.length; i++) {
            if (grades[i] > grades[maxIndex]) {
                maxIndex = i;
            }
        }
        return maxIndex;
    }
}
```

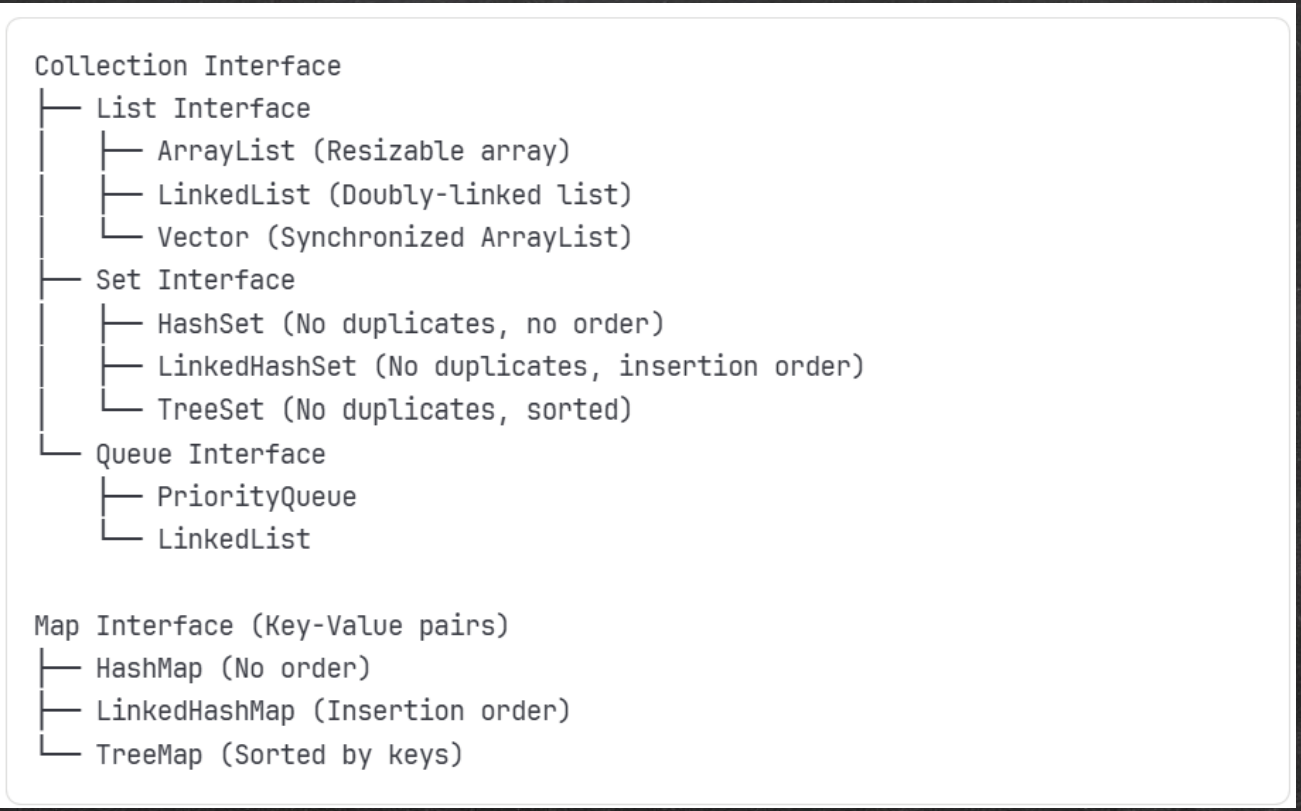
Output:

```
=== Student Grades ===
Alice: 85
Bob: 92
Charlie: 78
Diana: 96

Class Average: 87.75
Top Student: Diana with 96 points
```

Modular methods avoid code duplication and improve maintainability

Java Collections Framework - Dynamic and Flexible



- **Principle tie-in:** Abstraction and modularity — choose the right tool for the job

ArrayList - Dynamic Arrays

Basic ArrayList Operations:

```java
import java.util.ArrayList;

public class ArrayListDemo {
    public static void main(String[] args) {
        // Create ArrayList
        ArrayList<String> cities = new ArrayList<>();

        // Add elements
        cities.add("Phnom Penh");
        cities.add("Siem Reap");
        cities.add("Battambang");
        cities.add("Sihanoukville");

        // Display size and elements
        System.out.println("Number of cities: " + cities.size());
        System.out.println("Cities: " + cities);

        // Access specific element
        System.out.println("First city: " + cities.get(0));

        // Check if contains
        System.out.println("Contains Siem Reap? " + cities.contains("Siem Reap"));

        // Remove element
        cities.remove("Battambang");
        System.out.println("After removing Battambang: " + cities);

        // Traditional loop
        System.out.println("\nUsing traditional for loop:");
        for (int i = 0; i < cities.size(); i++) {
            System.out.println((i + 1) + ". " + cities.get(i));
        }

        // Enhanced for loop
        System.out.println("\nUsing enhanced for loop:");
        for (String city : cities) {
            System.out.println("- " + city);
        }
    }
}
```

Output:

```
Number of cities: 4
Cities: [Phnom Penh, Siem Reap, Battambang, Sihanoukville]
First city: Phnom Penh
Contains Siem Reap? true
After removing Battambang: [Phnom Penh, Siem Reap, Sihanoukville]

Using traditional for loop:
1. Phnom Penh
2. Siem Reap
3. Sihanoukville

Using enhanced for loop:
- Phnom Penh
- Siem Reap
- Sihanoukville
```

- **Principle tie-in:** Dynamic sizing supports maintainability and adaptability

HashMap - Key-Value Storage

Complete Phone Book Example:

java

```java
import java.util.HashMap;
import java.util.Scanner;

public class PhoneBook {
    private HashMap<String, String> contacts;

    public PhoneBook() {
        contacts = new HashMap<>();
        // Pre-populate with some data
        contacts.put("Alice", "012-345-678");
        contacts.put("Bob", "011-234-567");
        contacts.put("Charlie", "010-123-456");
    }

    public void addContact(String name, String phone) {
        contacts.put(name, phone);
        System.out.println("✓ Contact added: " + name);
    }

    public void searchContact(String name) {
        String phone = contacts.get(name);
        if (phone != null) {
            System.out.println("📞 " + name + ": " + phone);
        } else {
            System.out.println("❌ Contact not found: " + name);
        }
    }

    public void displayAllContacts() {
        System.out.println("\n📋 All Contacts:");
        System.out.println("================");
        if (contacts.isEmpty()) {
            System.out.println("No contacts found.");
        } else {
            for (String name : contacts.keySet()) {
                System.out.println(name + " -> " + contacts.get(name));
            }
        }
    }

    public void removeContact(String name) {
        if (contacts.remove(name) != null) {
            System.out.println("✓ Contact removed: " + name);
        } else {
            System.out.println("❌ Contact not found: " + name);
        }
    }
```

```java
    public static void main(String[] args) {
        PhoneBook phoneBook = new PhoneBook();
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("\n=== Phone Book Menu ===");
            System.out.println("1. Add Contact");
            System.out.println("2. Search Contact");
            System.out.println("3. Display All Contacts");
            System.out.println("4. Remove Contact");
            System.out.println("5. Exit");
            System.out.print("Choose option: ");

            int choice = scanner.nextInt();
            scanner.nextLine(); // Consume newline

            switch (choice) {
                case 1:
                    System.out.print("Enter name: ");
                    String name = scanner.nextLine();
                    System.out.print("Enter phone: ");
                    String phone = scanner.nextLine();
                    phoneBook.addContact(name, phone);
                    break;

                case 2:
                    System.out.print("Enter name to search: ");
                    String searchName = scanner.nextLine();
                    phoneBook.searchContact(searchName);
                    break;

                case 3:
                    phoneBook.displayAllContacts();
                    break;

                case 4:
                    System.out.print("Enter name to remove: ");
                    String removeName = scanner.nextLine();
                    phoneBook.removeContact(removeName);
                    break;

                case 5:
                    System.out.println("👋 Goodbye!");
                    return;

                default:
                    System.out.println("❌ Invalid option. Try again.");
            }
        }
    }
}
```

Principle tie-in: Separation of concerns — each method handles one specific task, making the code modular and testable

HashSet - Unique Collections

```java
import java.util.HashSet;
import java.util.ArrayList;

public class VisitorTracker {
    private HashSet<String> uniqueVisitors;
    private ArrayList<String> allVisits;

    public VisitorTracker() {
        uniqueVisitors = new HashSet<>();
        allVisits = new ArrayList<>();
    }

    public void recordVisit(String visitorName) {
        allVisits.add(visitorName);
        uniqueVisitors.add(visitorName); // Automatically handles duplicates

        if (uniqueVisitors.contains(visitorName) &&
            allVisits.lastIndexOf(visitorName) > allVisits.indexOf(visitorName)) {
            System.out.println("🔁 " + visitorName + " - Returning visitor!");
        } else {
            System.out.println("👋 " + visitorName + " - New visitor!");
        }
    }

    public void generateReport() {
        System.out.println("\n📊 Visitor Report");
        System.out.println("================");
        System.out.println("Total visits: " + allVisits.size());
        System.out.println("Unique visitors: " + uniqueVisitors.size());
        System.out.println("Return rate: " +
            String.format("%.1f%%",
                ((double)(allVisits.size() - uniqueVisitors.size()) / allVisits.size()

        System.out.println("\nUnique visitors list:");
        for (String visitor : uniqueVisitors) {
            long visitCount = allVisits.stream().filter(v -> v.equals(visitor)).count
            System.out.println("- " + visitor + " (" + visitCount + " visits)");
        }
    }

    public static void main(String[] args) {
        VisitorTracker tracker = new VisitorTracker();

        // Simulate website visits
        tracker.recordVisit("Alice");
        tracker.recordVisit("Bob");
        tracker.recordVisit("Alice");    // Duplicate
        tracker.recordVisit("Charlie");
        tracker.recordVisit("Bob");      // Duplicate
        tracker.recordVisit("Diana");
        tracker.recordVisit("Alice");    // Duplicate

        tracker.generateReport();
    }
}
```

Example: Unique Visitor Tracker

Output:

```
👋 Alice - New visitor!
👋 Bob - New visitor!
🔁 Alice - Returning visitor!
👋 Charlie - New visitor!
🔁 Bob - Returning visitor!
👋 Diana - New visitor!
🔁 Alice - Returning visitor!

📊 Visitor Report
================
Total visits: 7
Unique visitors: 4
Return rate: 42.9%

Unique visitors list:
- Alice (3 visits)
- Bob (2 visits)
- Charlie (1 visits)
- Diana (1 visits)
```

Principle tie-in: Data integrity and correctness through automatic duplicate handling

Lambda Expressions - Functional Programming Power

- Introduced in Java 8 for concise, functional-style code
- Reduces boilerplate in loops and conditional logic
- **Principle tie-in:** Improves readability, supports DRY, and encourages modular thinking

Lambda Expressions – Common usages

- **Iteration**
  - names.forEach(name -> System.out.println(name));
- **Filtering**
  - names.stream().filter(n -> n.startsWith("A")).forEach(System.out::println);
- **Mapping**
  - List<Integer> lengths = names.stream().map(String::length).toList();
- **Sorting**
  - names.sort((a, b) -> a.compareToIgnoreCase(b));
- **Reducing**
  - int totalLength = names.stream().mapToInt(String::length).sum();

Lambda Expression - Examples

Basic Syntax:

```java
// Traditional way
Collections.sort(names, new Comparator<String>() {
    public int compare(String a, String b) {
        return a.compareToIgnoreCase(b);
    }
});

// Lambda way
Collections.sort(names, (a, b) -> a.compareToIgnoreCase(b));
```

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

public class LambdaShowcase {
    public static void main(String[] args) {
        // Sample data
        List<String> programmingLanguages = Arrays.asList(
            "Java", "Python", "JavaScript", "C++", "Go",
            "Rust", "Kotlin", "Swift", "TypeScript", "C#"
        );

        System.out.println("Original list: " + programmingLanguages);

        // 1. ITERATION with forEach
        System.out.println("\n1. Displaying with forEach:");
        programmingLanguages.forEach(lang -> System.out.println("- " + lang));

        // 2. FILTERING - Languages starting with 'J'
        System.out.println("\n2. Languages starting with 'J':");
        programmingLanguages.stream()
            .filter(lang -> lang.startsWith("J"))
            .forEach(System.out::println);

        // 3. MAPPING - Convert to uppercase and get lengths
        System.out.println("\n3. Language lengths:");
        List<String> langWithLengths = programmingLanguages.stream()
            .map(lang -> lang + " (" + lang.length() + " chars)")
            .collect(Collectors.toList());
        langWithLengths.forEach(System.out::println);

        // 4. SORTING - Multiple ways
        System.out.println("\n4. Sorted alphabetically:");
        programmingLanguages.stream()
            .sorted()
            .forEach(System.out::println);

        System.out.println("\n5. Sorted by length (shortest first):");
        programmingLanguages.stream()
            .sorted((a, b) -> Integer.compare(a.length(), b.length()))
            .forEach(System.out::println);

        // 6. REDUCING - Find total character count
        int totalChars = programmingLanguages.stream()
            .mapToInt(String::length)
            .sum();
        System.out.println("\n6. Total characters in all names: " + totalChars);

        // 7. COMPLEX OPERATIONS - Find languages with more than 4 chars,
        //    convert to uppercase, and sort
        System.out.println("\n7. Long names (>4 chars), uppercase, sorted:");
        List<String> processedLanguages = programmingLanguages.stream()
            .filter(lang -> lang.length() > 4)
            .map(String::toUpperCase)
            .sorted()
            .collect(Collectors.toList());
        processedLanguages.forEach(System.out::println);

        // 8. GROUPING - Group by first letter
        System.out.println("\n8. Grouped by first letter:");
        Map<Character, List<String>> groupedByFirstLetter = programmingLanguages.s
            .collect(Collectors.groupingBy(lang -> lang.charAt(0)));

        groupedByFirstLetter.forEach((letter, langs) -> {
            System.out.println(letter + ": " + langs);
        });
    }
}
```

Principle tie-in: Lambdas improve readability, support DRY, and encourage modular thinking

Data Structure Performance Comparison

Visual Performance Chart:

Operation	Array	ArrayList	HashMap	HashSet
Access by index	O(1)	O(1)	N/A	N/A
Search	O(n)	O(n)	O(1)	O(1)
Insert at end	N/A	O(1)	O(1)	O(1)
Insert at beginning	N/A	O(n)	O(1)	O(1)
Delete	N/A	O(n)	O(1)	O(1)
Memory usage	Low	Medium	Medium	Medium

Choosing the Right Structure - Decision Tree:

```
Do you need key-value pairs?
├── Yes → HashMap/TreeMap
└── No
    ├── Need unique elements only?
    │   ├── Yes → HashSet/TreeSet
    │   └── No
    │       ├── Fixed size known?
    │       │   ├── Yes → Array
    │       │   └── No → ArrayList/LinkedList
    └── Need ordering?
        ├── Insertion order → LinkedHashSet/LinkedHashMap
        └── Sorted order → TreeSet/TreeMap
```

What is Big O Notation?

Big O notation describes **how the performance of an algorithm changes as the input size grows**. It answers the question: *"If I have more data, how much slower will my code get?"*

O(1) - Constant Time ⚡ BEST
Meaning: No matter how much data you have, it always takes the same amount of time.

O(n) - Linear Time 📈 GROWS WITH DATA
Meaning: If you double the data, the time roughly doubles too.

O(n²) - Quadratic Time 😱 BAD
Meaning: If you double the data, the time becomes 4 times longer!

Lab Activity – Contact Manager

- Use HashMap<String, String> for name → phone
- Add, search, remove, and display contacts
- Use Lambda with forEach for display
- Principle tie-in: Each feature in its own method (modularity)

Assignment – Word Frequency Counter

- Create a comprehensive word frequency analyzer that demonstrates multiple programming principles and data structures.

Core Requirements:

- Input: Accept either user input or read from a text file
- Processing: Count word frequency ignoring case and punctuation
- Output: Display results in multiple formats
- Data Structures: Use HashMap for frequency counting, ArrayList for sorting
- Lambda Usage: Implement filtering and sorting using Lambda expressions
- Reflection: How do principles like DRY and clarity improve your design?

Wrap-up and Q&A

- Data structures + collections support clean, maintainable, modular code
- Lambdas add expressive, functional-style operations
- Next: OOP Principles – Designing Custom Data Models

AI Integration - Prompts for Data Structure Learning

Beginner Level AI Prompts:

- "Explain the difference between ArrayList and Array in Java with simple examples“
- "Write a Java program that uses HashMap to store student names and their grades, include methods to add, find, and display students“
- "Show me how to use Lambda expressions to filter a list of numbers and keep only even numbers“
- "Create a simple Java example that demonstrates when to use HashSet vs ArrayList"

Intermediate Level AI Prompts

- "Design a Java inventory management system using appropriate data structures (HashMap, ArrayList, HashSet) with full CRUD operations and explain why you chose each structure“
- "Write a Java program that reads a text file, counts word frequency using HashMap, and displays the top 5 most common words using Lambda expressions and streams“
- "Create a student management system that groups students by grade level, sorts them by name within each grade, and provides search functionality - use multiple data structures and explain the design choices“
- "Implement a simple recommendation engine that suggests items based on user preferences using HashSet for user interests and HashMap for item ratings"

Advanced Level AI Prompts

- "Design a multi-threaded Java application that processes large datasets using concurrent HashMaps and parallel streams, ensuring thread safety and optimal performance“
- "Create a caching system using LinkedHashMap with LRU (Least Recently Used) eviction policy, include performance metrics and explain when this would be useful in real applications“
- "Build a social network friend suggestion system using multiple data structures: HashMap for user profiles, HashSet for friendships, and custom algorithms for finding mutual connections“
- "Design and implement a data analytics pipeline that processes JSON data, performs complex filtering and grouping operations using Lambda expressions, and outputs formatted reports"

AI Prompt Best Practices

- **Be specific** about requirements and constraints
- **Ask for explanations** of design choices
- **Request multiple approaches** when appropriate
- **Include edge cases** and error handling requirements
- **Ask for code comments** and documentation