420-301-VA: Programming Patterns

Lab 1: ArrayList, Comparators, and Lambda Expressions

Computer Science & Technology
Vanier College

Introduction

In this lab, we will explore fundamental patterns for managing collections of objects in Java. We will focus on the `ArrayList` class, working with custom objects, and using the `Comparable` and `Comparator` interfaces for sorting. Finally, we will introduce a modern approach to sorting and data processing using lambda expressions.

Objectives

This lab will guide you through the essential patterns for working with basic collections of objects in Java. At the end of the lab, you should now be comfortable with:

- Using `ArrayList` to manage custom objects.
- Defining "natural ordering" with `Comparable`.
- Creating custom ordering strategies with `Comparator`.
- Modernizing your code with concise and powerful lambda expressions.

These concepts are fundamental to professional Java development. They will form the basis of more advanced topics, such as the Stream API and parallel processing, that we will cover in the future.

Instructions

- Please try out the examples first to deepen your understanding.
- Work on the exercises and copy-paste your formatted code in a separate document, followed by the screenshots of the console outputs to validate that your code worked.
- You can work with your peers, but each person must code, execute, and create the document on their own computer.

1 Part 1: `ArrayList` and Lists of Class Objects

1.1 Concept Review: The `ArrayList`

The `ArrayList` is a resizable array implementation of the `List` interface. It provides dynamic size capabilities, allowing you to add or remove elements without needing to pre-define a fixed size. It is a part of the Java Collections Framework and is one of the most commonly used data structures.

1.2 Creating a List of Objects

While `ArrayList` is great for simple types like `String` or `Integer`, its true power lies in its ability to store custom objects. This is the cornerstone of object-oriented programming in Java.

Let's define a simple `Student` class.

```java
public class Student {
    private String name;
    private int studentId;
    private double gpa;

    public Student(String name, int studentId, double gpa) {
        this.name = name;
        this.studentId = studentId;
        this.gpa = gpa;
    }

    // Getters for name, studentId, gpa
    public String getName() { return name; }
    public int getStudentId() { return studentId; }
    public double getGpa() { return gpa; }

    @Override
    public String toString() {
        return "Student{" +
               "name='" + name + '\'' +
               ", studentId=" + studentId +
               ", gpa=" + gpa +
               '}';
    }
}
```

Listing 1: Student.java

Now, we can create an `ArrayList` to hold `Student` objects:

```java
import java.util.ArrayList;
import java.util.List;

public class StudentListExample {
    public static void main(String[] args) {
        // Create a List of Student objects
        List<Student> students = new ArrayList<>();

        // Add students to the list
        students.add(new Student("Alice", 101, 3.8));
        students.add(new Student("Bob", 103, 3.2));
        students.add(new Student("Charlie", 102, 4.0));

        // Print the list
        for (Student s : students) {
            System.out.println(s);
        }
    }
}
```

Listing 2: Creating an ArrayList of Students

Exercise 1: `ArrayList` Fundamentals

1. Create a new Java class called `Book` with the following attributes: `title` (`String`), `author` (`String`), and `yearPublished` (`int`).
2. Implement a constructor and getters for all attributes.
3. Override the `toString()` method to provide a clear string representation of a `Book` object.
4. In a `main` method, create an `ArrayList` of `Book` objects. Add at least five books to the list.
5. Write code to iterate through the list and print the details of each book.
6. Write code to find and print the book with the highest `yearPublished`.

2 Part 2: The `Comparable` and `Comparator` Interfaces

2.1 Concept Review: Sorting Objects

The `Collections.sort()` method can sort a list of objects, but it needs to know how to compare them. This is where `Comparable` and `Comparator` come in.

2.1.1 `Comparable<T>`

This interface is for "natural ordering." If your class has a single, obvious way to be sorted (e.g., by student ID, name, or product ID), you should implement `Comparable`. The interface has one method: `compareTo(T o)`.

- Returns a negative integer if `this` object is less than `o`.
- Returns a positive integer if `this` object is greater than `o`.
- Returns zero if `this` object is equal to `o`.

```java
import java.util.Collections;
import java.util.List;

public class Student implements Comparable<Student> {
    // ... attributes, constructor, getters, and toString() as
       before ...

    @Override
    public int compareTo(Student other) {
        // Natural order: sort by studentId
        return Integer.compare(this.studentId, other.studentId);
    }
}
```

Listing 3: Student class with Comparable

With this implementation, `Collections.sort(students)` will now sort the list by `studentId`.

2.1.2 `Comparator<T>`

This interface is used for "custom ordering." It allows you to define multiple sorting strategies without modifying the original class. `Comparator` is a separate class that implements a single method: `compare(T o1, T o2)`.

```java
import java.util.Comparator;

public class GpaComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        // Sort in descending order of GPA
        return Double.compare(s2.getGpa(), s1.getGpa());
    }
}
```

Listing 4: Comparator for sorting by GPA

To use this `Comparator`, you pass it as a second argument to `Collections.sort()`: `Collections.sort(students, new GpaComparator());`

Exercise 2: Implementing Sorting Strategies

1. Modify your `Book` class from Exercise 1 to implement the `Comparable` interface. The natural ordering for a book should be by its title (alphabetically).
2. In your `main` method, create the `ArrayList` of `Book` objects and then sort it using `Collections.sort()`. Print the list before and after sorting to see the effect.
3. Create two separate `Comparator` classes for your `Book` objects:
   - `AuthorComparator`: Sorts books alphabetically by author's name.
   - `YearComparator`: Sorts books in ascending order of `yearPublished`.
4. In your `main` method, sort the list of books using each of the new comparators and print the results each time.

3 Part 3: Functional Programming with Lambda Expressions

3.1 Concept Review: Lambda Expressions

A lambda expression is a short block of code that takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and can be implemented right in the body of a method. They are a core feature of Java 8 and are used extensively with functional interfaces.

A functional interface is an interface with exactly one abstract method. Both `Comparable` and `Comparator` are functional interfaces.

3.2 Using Lambdas with `Comparator`

Instead of creating a new `GpaComparator` class, we can use a lambda expression directly.

```java
import java.util.Collections;
import java.util.List;

public class LambdaExample {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        // ... add students ...

        // Sort by studentId using a lambda expression
        // Equivalent to our old GpaComparator, but much shorter
        Collections.sort(students, (s1, s2) -> Double.compare(s2.
           getGpa(), s1.getGpa()));

        // Print the sorted list
        for (Student s : students) {
            System.out.println(s);
        }
    }
}
```

Listing 5: Sorting with a Lambda Expression

The syntax `(s1, s2)-> Double.compare(s2.getGpa(), s1.getGpa())` is a lambda expression that implements the `compare` method of the `Comparator` interface.

The `Comparator` interface also provides convenient default and static methods for creating comparators, such as `comparing()`.

```java
import java.util.Comparator;
import java.util.List;

public class LambdaStreamExample {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        // ... add students ...

        // Sort by GPA, descending
        students.sort(Comparator.comparing(Student::getGpa).reversed
           ());

        // Sort by name, ascending
        students.sort(Comparator.comparing(Student::getName));

        // Sort by studentId, then by name for ties
        students.sort(Comparator.comparing(Student::getStudentId)
                                  .thenComparing(Student::getName));
    }
}
```

Listing 6: Using the Comparator.comparing() method

The method reference `Student::getGpa` is even more concise than the lambda and is equivalent to `s -> s.getGpa()`.

Exercise 3: Lambda and Streams

1. Use your `Book` class from the previous exercises.
2. Instead of using the `Comparator` classes you created, rewrite the sorting logic in your `main` method using lambda expressions or `Comparator.comparing()`.
    - Sort the list of books alphabetically by title.
    - Sort the list of books by author's name.
    - Sort the list of books by `yearPublished` in descending order.
3. Print the list after each sort to confirm the correct ordering.
4. **Thinking Exercise:** What is the key advantage of using lambda expressions and method references for sorting compared to creating separate `Comparator` classes? What about compared to implementing `Comparable`?
5. **Programming Exercise:** Use a stream to find and print the book published most recently. The code should be a single line (excluding the print statement).
   Hint: look into `stream()` and `Comparator.comparing()`.

References

ArrayList

- Java Documentation for ArrayList: The official Javadoc for the `ArrayList` class.
- GeeksforGeeks: ArrayList in Java: A comprehensive tutorial with examples on how to use `ArrayList`.
- Baeldung: The ArrayList in Java: A detailed guide covering creation, operations, and best practices.

Comparable and Comparator

- Java Documentation for Comparable: Official Javadoc for the `Comparable` interface.
- Java Documentation for Comparator: Official Javadoc for the `Comparator` interface.
- Baeldung: The Java Comparable and Comparator Interfaces: An excellent article explaining the key differences and when to use each one.
- GeeksforGeeks: Comparable vs Comparator in Java: A comparison with clear examples to help you decide which to use.

Lambda Expressions

- Oracle Java Tutorials: Lambda Expressions: The official tutorial from Oracle, providing the core concepts and syntax.
- Baeldung: Introduction to Java 8 Lambda Expressions: A practical and easy-to-understand guide with lots of code examples.
- GeeksforGeeks: Lambda Expressions in Java: A quick reference for syntax and usage.

Java Stream API

- Java Documentation for Stream: The official Javadoc for the `Stream` interface, detailing all available operations.
- Baeldung: A Guide to the Java 8 Stream API: An in-depth tutorial on how to use streams, covering intermediate and terminal operations.
- GeeksforGeeks: Stream in Java: A concise overview of the Stream API with examples of common operations.