420-301-VA: Programming Patterns

Lab 2: Introduction to Java Generics

Computer Science & Technology
Vanier College

Introduction

In this lab, we will explore Java Generics, a powerful feature that allows for the creation of type-safe and reusable code. Generics enable us to define classes, interfaces, and methods with placeholders for types, which are then specified when the code is used. The primary benefit of this approach is stronger type checking at compile time, which helps detect errors early and prevents runtime exceptions. We will learn how to create our own generic classes and methods, implement a generic data structure, and finally, apply these concepts to stream processing with collections.

Objectives

This lab will provide a hands-on introduction to Java Generics and their practical applications. At the end of the lab, you should be comfortable with:

- Understanding the benefits of Generics for type safety and code reusability.
- Defining and using generic classes to create flexible data structures.
- Implementing generic methods that can operate on a variety of types.
- Applying basic stream operations to process collections of objects.

These skills are essential for writing robust, clean, and efficient Java applications and are a stepping stone to more advanced topics in the Java Collections Framework.

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 has to code, execute, and create the document on their own computers.

1 Part 1: Defining Generic Classes and Methods

1.1 Concept Review: Generic Classes

A generic class is a class that is parameterized over types. You define a generic class by specifying a type parameter in angle brackets (<>) after the class name. This type parameter can then be used like a regular type within the class. This allows you to create a single class that can work with different data types while maintaining type safety.

A classic example is a `Box<T>` class that can hold an object of any type.

```
public class Box<T> {
    private T content;

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

    public T getContent() {
        return content;
    }
}
```

Listing 1: A simple generic Box class

When you use this class, you specify the concrete type. This eliminates the need for manual casting and prevents you from accidentally putting the wrong type of object into the box.

```
// Create a Box for Strings
Box<String> stringBox = new Box<>();
stringBox.setContent("Hello Generics!");
String value = stringBox.getContent(); // No casting needed

// Create a Box for Integers
Box<Integer> intBox = new Box<>();
intBox.setContent(42);
Integer number = intBox.getContent(); // No casting needed

// The following line would cause a compile-time error
// stringBox.setContent(123);
```

Listing 2: Using the generic Box class

Exercise 1: Creating a Generic `Pair` Class

1. Create a new generic Java class called `Pair<K, V>`. This class will store two objects, which can be of different types (a"key" and a "value").

2. The class should have two private instance variables: one of type `K` for the key and one of type `V` for the value.

3. Implement a constructor that accepts a key and a value to initialize the `Pair` object.

4. Implement public "getter" methods, `getKey()` and `getValue()`, to retrieve the stored objects.

5. Implement public "setter" methods, `setKey()` and `setValue()`, to store the given value in these objects.

6. Override the `toString()` method to return a formatted string representation, such as `"(key=yourKey, value=yourValue)"`.

7. In a `main` method, create at least two instances of your `Pair` class with different type arguments (e.g., `Pair<String, Integer>` and `Pair<Integer, String>`) and print them to the console to verify your implementation.

2 Part 2: Implementing a Generic Data Structure

2.1 Concept Review: The Generic Repository Pattern

The Repository pattern is a common design pattern used to abstract the data layer, providing a collection-like interface for accessing domain objects. Generics make this pattern incredibly powerful because you can define a single repository interface and implementation that works for any type of data object (e.g., `User`, `Product`, `Order`).

Let's look at a generic `Repository` interface and an in-memory implementation.

```
// Generic interface for a repository
interface Repository<T> {
    void save(T item);
    T findById(int id);
    List<T> findAll();
}

// Implementation using an ArrayList
class InMemoryRepository<T> implements Repository<T> {
    private List<T> storage = new ArrayList<>();

    @Override
    public void save(T item) {
        storage.add(item);
    }

    @Override
    public T findById(int id) {
        if (id < 0 || id >= storage.size()) {
            return null; // or throw an exception
        }
        return storage.get(id);
    }

    @Override
    public List<T> findAll() {
        return new ArrayList<>(storage); // Return a copy
    }
}
```

Listing 3: Generic Repository Interface and Implementation

This generic structure allows us to create repositories for any object type without duplicating code, for example `Repository<User>` or `Repository<Product>`.

Exercise 2: Building a `Product` Repository

1. Create a simple Java class called `Product` with the following private attributes: `productId` (`int`), `name` (`String`), and `price` (`double`).

2. Implement a constructor, getters for all attributes, and override the `toString()` method for easy printing.

3. Create your own generic interface named `GenericRepository<T>` with the following methods:

   - `void add(T item);`
   - `T get(int index);`
   - `List<T> getAll();`

4. Create a concrete class called `ProductRepository` that implements `GenericRepository <Product>`. Use an `ArrayList<Product>` internally to store the products.

5. In a `main` method, create an instance of `ProductRepository`.

6. Add at least four different `Product` objects to the repository.

7. Use the `get` method to retrieve and print a single product by its index.

8. Use the `getAll` method to get the complete list of products and iterate through it to print the details of each product.

9. **Find Maximum Element (Bounded Generics):** First, modify your `Product` class to implement the `Comparable<Product>` interface. The natural ordering should be based on price, so the `compareTo` method should compare the prices of two products. Next, create a `public static` generic method with the following signature:
   `public static <T extends Comparable<T>> T findMax(List<T> items)`. This method should take a list of comparable items, iterate through them to find the element that is considered the maximum, and return it. In your `main` method, call `findMax` with your list of products and print the details of the most expensive product found.

10. **Generic Filtering Method:** Create a second `public static` generic method with the signature:
    `public static <T extends Comparable<T>> List<T> filterGreaterThan(List<T> items , T threshold)`. This method should accept a list of items and a single "threshold" item. It must create and return a new `List<T>` containing only the items from the original list that are "greater than" the provided threshold, based on the result of the `compareTo` method. In your `main` method, test this by finding and printing all products that are more expensive than a new product instance you create as the threshold (e.g., a product with a price of $40.0).

3 Part 3: Stream Processing with Collections

3.1 Concept Review: The Stream API

The Java Stream API, introduced in Java 8, provides a powerful and declarative way to process sequences of elements. A stream is not a data structure; instead, it takes input from a collection, array, or I/O channel. Streams support many aggregate operations like `filter`, `map`, `reduce`, and `collect` that can be chained together to form a processing pipeline.

For example, given a `List<Product>`, we can easily find all products that cost more than $100.

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

public class StreamExample {
    public static void main(String[] args) {
        // Assume 'productRepository' is populated from the previous
            exercise
        ProductRepository productRepository = new ProductRepository
            ();
        // ... add products ...

        List<Product> allProducts = productRepository.getAll();

        // Use a stream to find expensive products
        List<Product> expensiveProducts = allProducts.stream()
                .filter(p -> p.getPrice() > 100.0)
                .collect(Collectors.toList());

        System.out.println("Expensive products:");
        expensiveProducts.forEach(System.out::println);
    }
}
```

Listing 4: Filtering a List with a Stream

The pipeline consists of:

- `stream()`: Gets a stream from the source list.
- `filter(...)`: An intermediate operation that returns a new stream containing only elements that match a predicate.
- `collect(...)`: A terminal operation that transforms the stream into a collection (e.g., a `List`).

Exercise 3: Processing Products with Streams

1. Continue using your `ProductRepository` and `Product` class from Exercise 2. Ensure your repository is populated with several products with varying names and prices.

2. In your `main` method, get the list of all products from the repository.

3. Using the Stream API, perform the following operations. Print the results of each operation clearly.

   - **Filter**: Create a new list containing only products with a price less than $50.0 and print it.
   - **Map**: Create a list (`List<String>`) containing only the *names* of all the products and print this list of names.
   - **Filter and Map**: Find all products with a name starting with the letter 'L', and create a list of their names in uppercase. Print the resulting list.
   - **Reduce/Aggregate**: Calculate the total cost of all products in the repository and print the result. (Hint: look into `mapToDouble` and the `sum()` terminal operation).

4. **Thinking Exercise:** What is the key difference between an intermediate stream operation (like `filter` or `map`) and a terminal operation (like `collect` or `forEach`)? Why is this distinction important?

References

Generics

- Oracle Java Tutorials: Generics: The official tutorial from Oracle, covering the fundamentals.
- Baeldung: Introduction to Java Generics: A practical guide with clear examples.

Java Collections Framework

- Java Documentation for ArrayList: The official Javadoc for the `ArrayList` class.
- GeeksforGeeks: ArrayList in Java: A comprehensive tutorial on `ArrayList`.

Java Stream API

- Java Documentation for Stream: The official Javadoc for the `Stream` interface.
- Baeldung: A Guide to the Java 8 Stream API: An in-depth tutorial on how to use streams.
- GeeksforGeeks: Stream in Java: A concise overview of the Stream API.