Java Programming 3-3: Collections – Part 2

Practice Activities

Vocabulary Definitions

1.

A double-ended queue; a queue that can add and remove elements to the front or back of the list.

2.

- **Deque** (Double-Ended Queue): A type of queue that supports adding and removing elements from both ends.

3.

The links of a LinkedList.

4.

- **Nodes**: Each element in a `LinkedList` is contained in a node, which holds a reference to the next and possibly the previous node.

5.

An interface used to define a group of objects. This includes lists and sets.

6.

- **Collection**: The root interface in the Java Collections Framework, representing a group of objects.

7.

Maps that link a Key to a Value and may have duplicate Keys but cannot have duplicate Values.

8.

- **HashMap**: A map that associates keys with values, where keys are unique but values can be duplicated.

9.

A list of elements that is dynamically stored.

10.

- **LinkedList**: A doubly-linked list implementation of the `List` interface, where elements are dynamically stored.

11.

A list of elements with a first in first out ordering.

12.

- **Queue**: A collection that supports adding elements at the end and removing elements from the front, adhering to FIFO (First In, First Out) order.

1. Difference Between Queue and Stack

-

Queue:

-
- **Description**: Follows FIFO (First In, First Out) principle.
- **Example**: A queue at a ticket counter where the first person in line is the first one to be served.
- **Implementation**: `LinkedList` or `PriorityQueue`.
-

Stack:

-
- **Description**: Follows LIFO (Last In, First Out) principle.
- **Example**: A stack of plates where the last plate placed is the first one to be removed.
- **Implementation**: `Stack` class or `ArrayDeque`.

2. Implementing a Generic Stack

a. Create a project named `genericstack`

b. Create `GenericStackException`

```
public class GenericStackException extends RuntimeException {
    public GenericStackException(String message) {
        super(message);
    }
}
```

c. Create `GenericStack`

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

```
public class GenericStack<T> {
    private ArrayList<T> items;
    private int top;

    public GenericStack() {
        items = new ArrayList<>();
        top = 0;
    }

    private boolean isEmpty() {
        return top == 0;
    }

    public void push(T item) {
        items.add(item);
        top++;
    }

    public T pop() {
        if (isEmpty()) {
            throw new GenericStackException("Underflow Error");
        }
        T item = items.get(top - 1);
        items.remove(top - 1);
        top--;
        return item;
    }
}
```

d. Create `StackDriver` Class

```
public class StackDriver {
    public static void main(String[] args) {
        GenericStack<Integer> stack = new GenericStack<>();

        // Push items
        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);

        // Pop items
        try {
            System.out.println(stack.pop());
            System.out.println(stack.pop());
            System.out.println(stack.pop());
            System.out.println(stack.pop());
            System.out.println(stack.pop()); // This will trigger an exception
        } catch (GenericStackException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

3. Adding Nodes to a LinkedList

- **Adding to the Beginning**: Yes, it is possible. Use the `addFirst(E e)` method in a `LinkedList`.

- **Adding to the End**: Yes, it is possible. Use the `add(E e)` method or `addLast(E e)` method in a `LinkedList`.

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();
        list.add("Element 1");  // Adds to the end
        list.addFirst("Element 2"); // Adds to the beginning
        System.out.println(list);  // Output: [Element 2, Element 1]
    }
}
```

4. Purpose of Implementing Comparable Interface

- **Purpose**: To define a natural ordering for objects of a class. Implementing `Comparable` allows objects to be compared and sorted based on a defined order. For instance, implementing `Comparable` allows objects to be sorted in collections like `TreeSet` or `ArrayList` when using `Collections.sort()`.

5. Storing Courses and Their Codes

- **Appropriate Collection**: `HashMap` is suitable because it stores key-value pairs, allowing efficient lookups.

a. Print List of Courses

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

public class CourseCatalog {
    public static void main(String[] args) {
        Map<String, String> courses = new HashMap<>();
        courses.put("CIT", "Computing and Information Technology");
        courses.put("CHI", "Childcare and Early Education");
        courses.put("MVS", "Motor Vehicle Systems");
        courses.put("BTH", "Beauty Therapy");
        courses.put("GDE", "Graphic Design");
        for (Map.Entry<String, String> entry : courses.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

b. Use `get` Method

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

public class CourseCatalog {
    public static void main(String[] args) {
        Map<String, String> courses = new HashMap<>();
        courses.put("CIT", "Computing and Information Technology");
        courses.put("CHI", "Childcare and Early Education");
        courses.put("MVS", "Motor Vehicle Systems");
        courses.put("BTH", "Beauty Therapy");
        courses.put("GDE", "Graphic Design");

        // Get course name by code
        String courseCode = "CIT";
        System.out.println(courseCode + ": " + courses.get(courseCode));
    }
}
```