Practical Exercise 4 – Java Collections Framework

Overall Objective

To explore Java Collections Framework and use the common methods defined in the Collection interface for operating collections.

Background

You will need to know:

1. basic Java programming knowledge
2. classes and interfaces
3. generics
4. **Java Collections Framework**

Description

Part 1: Discussion

Collections

1. What is a data structure?
   A data structure is a collection of data organized in some fashion.
   In object-oriented thinking, a data structure is an object that stores other objects, referred to as data or elements. So some people refer a data structure as a *container object* or a *collection object*. To define a data structure is essentially to declare a class.

2. Describe the Java Collections Framework.
   The Java Collections Framework defines the Java API for handling common data structures tasks in Java. It defines classes and interfaces for storing and manipulating data in sets, lists, and maps.
   A convenience class is an abstract class that partially implements an interface. The Java Collections Framework defines interfaces, convenience abstract classes, and concrete classes.

3. Can a collection object be cloned and serialized?
   Yes. The concrete classes of Set, List, and Map implements the `clone()` method in the `Cloneable` interface.

4. What method do you use to add all the elements from one collection to another collection? **Answer: `addAll(Collection c)`.**

5. Which collection stores an ordered collection of elements? **Answer: List.**

6. How to handle a method that has no meaning in the subclass?
   If a method has no meaning in the subclass, you can implement it in the subclass to throw `java.lang.UnsupportedOperationException`, a subclass of `RuntimeException`. This is a good design that you can use in your project. If a method has no meaning in the subclass, you can implement it as follows:

```
public void someMethod() {
      throw new UnsupportedOperationException
            ("Method not supported");
}
```

Iterators

7. How do you obtain an iterator from a collection object?
   The `Collection` interface extends the `Iterable` interface. You can obtain an *iterator* from a collection using the `iterator()` method.

8. What method do you use to obtain an element in the collection from an iterator?
   Use the `next()` method.

9. Can you use a *for-each* loop to traverse the elements in any instance of `Collection`?
   Answer: Yes.
   Do you need t use the **`next()`** or **`hasNext()`** methods in an iterator when using a for-each loop to traverse all elements in a collection?
   Answer: No. They are implicitly used in a *for-each* loop.

Lists

10. How do you traverse a list in both directions?
    Use the `listIterator()` to obtain an iterator. This iterator allows you to traverse the list bi-directional.

11. Suppose that `list1` is a `LinkedList` that is empty. What is the content of `list1` after executing the following statements:

• `list1.add(0, "red");`	**`[red]`**
• `list1.add(1, "yellow");`	**`[red, yellow]`**
• `list1.add(1, "green");`	**`[red, green, yellow]`**
• `list1.addFirst("blue");`	**`[blue, red, green, yellow]`**
• `list1.removeLast();`	**`[blue, red, green]`**
• `list1.removeFirst();`	**`[red, green]`**

12. What are the differences between `ArrayList` and `LinkedList`? When to use `LinkedList<>` over `ArrayList<>`?
    `ArrayList` and `LinkedList` can be operated similarly. The critical differences between them are their internal implementation, which impacts the performance.
    `ArrayList` is efficient for retrieving elements, and for adding and removing elements from the end of the list;
    `LinkedList` is efficient for adding and removing elements anywhere in the list.

13. Are all the methods in `ArrayList` also in `LinkedList`?
    What methods are in `LinkedList` but not in `ArrayList`?
    All the methods in `ArrayList` are also in `LinkedList` except the `trimToSize` method. The methods `getFirst`, `getLast`, `addFirst`, `addLast` are in `LinkedList`, but not in `ArrayList`.

The `Comparator` Interface

14. What are the differences between the `Comparable` interface and the `Comparator` interface?
    The `Comparable` interface contains the `compareTo` method and `Comparator` interface contains the `compare` method and `equals` method. Normally, if the objects of a class have natural order (e.g., String, Date), let the class implement the `Comparable` interface. The `Comparator` interface is more flexible in the sense that it enables you to define a new class that contains the `compare(Object, Object)` method to compare two objects of other classes.

    The `Comparable` interface is in the `java.lang` package, and the `Comparator` interface is in the `java.util` package.

15. The `Comparator` interface contains the `equals` method. Why is the method not implemented in the `GeometricObjectComparator` class in (**GeometricObjectComparator.java**)?
    Since The equals method is also defined in the `Object` class. Therefore, you will not get a compilation error if you don't implement the `equals` method in your custom comparator class. However, in some cases implementing this method may improve performance by allowing programs to determine that two distinct comparators impose the same order.

Sets and Maps

16. What are the differences between `HashSet`, `LinkedHashSet`, and `TreeSet`?
    `HashSet` is unsorted, but `TreeSet` is sorted. `HashSet` is more efficient than `TreeSet` if you don't want the elements in a set to be sorted. If you create a `TreeSet` using its default constructor, the `compareTo` method is used to compare the elements in the set, assuming that the class of the elements implements the `Comparable` interface. To use a comparator, you have to use the constructor `TreeSet(Comparator comparator)` to create a sorted set that uses the compare method in the comparator to order the elements in the set. A runtime error would occur if you add an element that cannot be compared with the existing elements in the tree set.

17. What data structure would you use in the follow tasks/problems?
    (a) Suppose you need to write a program that stores non-duplicate elements. **HashSet**
    (b) Suppose you need to write a program that stores non-duplicate elements in the order of insertion. **LinkedHashSet**
    (c) Suppose you need to write a program that stores non-duplicate elements in the increasing order of the element value. **TreeSet**

(d) Suppose you need to write a program that stores a fixed number of the elements (possibly duplicate). **Array**

(e) Suppose you need to write a program that stores the elements in a list with frequent operations to add and insert elements at the end of the list. **ArrayList**

(f) Suppose you need to write a program that stores the elements in a list with frequent operations to add and insert elements at the beginning of the list. **LinkedList**

Part 2: Programming Exercise

1. ***Displaying words in ascending alphabetical order***

Write a program that reads words from a text file and displays all the words (duplicates allowed) in ascending alphabetical order. You may assume all words start with a letter.

```
import java.io.File;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;

public class DisplayWordsAsc {
  public static void main(String[] args) {
    String filename = "AnyText.txt";

    // Create a list to hold the words
    ArrayList<String> list = new ArrayList<String>();

    try {
      Scanner in = new Scanner(new File(filename));

      String line;

      while (in.hasNext()) {
        line = in.nextLine();
        String[] words = line.split("[ \n\t\r.,:?)({}\\[\\]]");

        for (String word: words) {
          if (word.trim().length() > 0 && word.trim().matches("\\w+"))
            list.add(word.trim());
        }
      }
    }
    catch (Exception ex) {
      System.err.println(ex);
    }

    // Get an iterator for the list
    Collections.sort(list);

    // Display mappings
    System.out.println("\nDisplay words in ascending order ");
    for (String word: list) {
      System.out.println(word);
    }
  }
}
```

2. ***Sort points in a plane***

   Write a program that meets the following requirements:

   - Define a class named `Point` with two data fields `x` and `y` to represent a point's x- and y-coordinates. Implement the `Comparable` interface for comparing the points on x-coordinates. If two points have the same x-coordinates, compare their y-coordinates.
   - Define a class named `CompareY` that implements `Comparator<Point>`. Implement the `compare` method to compare two points on their y-coordinates. If two points have the same y-coordinates, compare their x-coordinates.
   - Randomly create 100 points and apply the `Arrays.sort` method to display the points in increasing order of their x-coordinates and in increasing order of y-coordinates, respectively.

     [Note that in Java, there is a method `Math.random()`, which returns a double value between 0.0 and 1.0. And there is another method `Random.nextInt(int n)`, which returns a random value in the range of 0 (inclusive) and n (exclusive).]

```
import java.util.Arrays;

public class SortedPoints {
  public static void main(String[] args) {
    Point[] points = new Point[100];
    for (int i = 0; i < points.length; i++) {
      points[i] = new Point(Math.random() * 100, Math.random() * 100);
    }

    System.out.println("Sort on x-coordinates");
    Arrays.sort(points);
    for (Point e: points) {
      System.out.println(e);
    }

    System.out.println("Sort on y-coordinates");
    Arrays.sort (points, new CompareY());
    for (Point e: points) {
      System.out.println (e);
    }
  }
  /** Define a class for a point with x- and y- coordinates */
  static class Point implements Comparable<Point> {
    double x;
    double y;

    Point(double x, double y) {
      this.x = x;
      this.y = y;
    }

    @Override
    public int compareTo(Point p2) {
      if (this.x < p2.x)
        return -1;
      else if (this.x > p2.x)
        return 1;
      else {
        // Secondary order on y-coordinates
        if (this.y < p2.y)
          return -1;
        else if (this.y > p2.y)
          return 1;
        else
          return 0;
      }
    }

    @Override
    public String toString() {
      return "(" + x + ", " + y + ")";
    }
  }

  /**
   * A comparator for comparing points on their y-coordinates. If y-
coordinates
   * are the same, compare their x-coordinator.
   */
  static class CompareY implements java.util.Comparator<Point> {
    @Override
    public int compare(Point p1, Point p2) {
      if (p1.y < p2.y)
        return -1;
      else if (p1.y > p2.y)
        return 1;
      else {
        // Secondary order on x-coordinates
        if (p1.x < p2.x)
          return -1;
        else if (p1.x > p2.x)
          return 1;
        else
          return 0;
      }
    }
  }
}
```

3. ***Use iterators on linked lists***

Write and test a Java program that stores 5 million integers in a linked list and test the time to traverse the list using an *iterator* (*for-each loop*) vs. using the `get(index)` method (i.e. compare on running time).

[Note that in Java, there is a method `System.currentTimeMillis()`, which returns the current time in milliseconds.]

```
import java.util.LinkedList;

public class TestLinkedListIterators {
  public static void main(String[] args) {
    LinkedList<Integer> list = new LinkedList<Integer>();

    for (int i = 0; i < 5000000; i++)
      list.add(i);

    long startTime = System.currentTimeMillis();
    for (int i = 0; i < list.size(); i++)
      list.get(i);
    long endTime = System.currentTimeMillis();
    System.out.print("Travese  time  using  index  is  "  +  (endTime  -
startTime));

    int x;
    startTime = System.currentTimeMillis();
    for (int i: list) {
      x = i;
    }
    endTime = System.currentTimeMillis();
    System.out.print  ("Travese  time  using  iterator  is  "  +  (endTime  -
startTime));

  }
}
```