Java Practical with Collections

Objectives

The objectives of this practical session are to:

- Use Collections instead of arrays in the Employee application
- Create random test data
- Sort the collections
- Use Inner Class syntax in Java to implement custom sorting objects

Overview

This practical consists of two parts. In the first part you will refactor your code to use the Collection API, with improved functionality over arrays. You then create random test data, which allows you to create a large number of employees and managers and then test the performance of your application to benchmark different choices of collection.

The second part uses `Comparator`s to sort collection elements. A suitable implementation of these data utilities might involve inner classes for coupling control.

Practical

Using Collections and Creating Random Data

1. In the project you created earlier, refactor the code in `EmployeeTest` and `Employee` to use `Collection` classes rather than arrays. Recall that `Collection` is an interface, which has many implementation classes. Which are most suitable for this application.
2. Use a the enhanced 'for-each' loop `for( : )` syntax when iterating around the collection. Compile, run and test your code.
3. Create a large number of `Employee` objects with random details. The method `Math.random()` returns a random double between 0.0 and 1.0, which you can use to scale up to random ages for example:

```
int age = (int) ((Math.random() * 40) + 18);
```

In the above expression, do you need all of the brackets?

4. You must also give the employees random names, which can be done in a similar way:

```
char c = (int) ((Math.random() * 26) + 65);
```

which gives you a random character, which you can assemble into `String`s, and use the methods `toUpperCase()` and `toLowerCase()` in the `String` class to make their names have the appropriate case.

5. If you want to have 'normally' distributed ages, you might try experimenting with the class `java.util.Random`, which has an instance method `nextGaussian()`. But remember the ages are not unlimited!
6. Refactor your code so all this random stuff is encapsulated. Perhaps the `Employee` class could have a method `createRandomEmployee()`?
7. Similarly, create random `Managers`, and devise some way to build a random organisation of `Managers` and `Employees`.
8. In such a fashion, create a million `Employees`!
9. The class method `System.currentTimeMillis()` tells you the epoch-based time (i.e. the number of milliseconds since the 1st Jan 1970!). You can use this to measure the time your application runs for. Do this and record your execution time.
10. Experiment with different `Collection` implementation classes to see which one is faster.

Practical

Part 2. Sorting and Comparator Classes

Objectives

The objective of this part is to define `Comparator` objects to sort your `Employee` objects in a `List`.

Overview

A `Comparator` object is an object that contains logic to sort other objects. You implement the `java.util.Comparator` generic class, and provide the method `compare()`, which takes two parameter and returns an `int`. The `int` must be a positive number or a negative number or zero, depending on whether the passed objects are in ascending order, descending order, or equal (as in the `equals()` method).

1. In the project you created earlier locate your `Employee` class. Inside this class definition, provide a new class `EmployeeNameComparator`, which implements `java.util.Comparator`. Make the class public and static (thus it is what is called a 'Nested Inner Class').

```
public class Employee {
  ...
  public static class EmployeeNameComparator
                  implements Comparator<Employee> {...
  }
}
```

2. Note the generic type parameter, which means this `Comparator` compares two `Employee` objects. Compile your code, and obey instructions to add the unimplemented methods.

3. Thus, provide the method:

```
public int compare(Employee e1, Employee e2) {
      ...
}
```

   This needs to return an `int`, so use the `String`'s `compareTo()` instance method on the `Employee`'s name, which sorts Strings alphabetically, and return precisely this value.

4. Note that, since the class is defined within the `Employee` class, you do in fact have access to the private variable directly, i.e. you do not even need to use your `getName()` method. That's part of the purpose of Inner Classes in Java.

5. In your main program, create yourself a `List` of `Employee` objects (perhaps some `Managers`), and print it out using a loop. Create an instance of your `Comparator` class. Now sort the `List` using the static method:

```
Collections.sort(yourList, yourComparator);
```

6. Print out the contents of the `List` once more, and test to see if your sorting has worked. Run and test your application.

7. In your `main()` method, provide another sort, this time via the `Employee`'s age. Use an anonymous class to do this:

```
Collections.sort(yourList, new Comparator<Employee>() {
   public int compare(Employee e1, Employee e2) {
      ...
   }
});
```

8. If two `Employee`'s have the same age (quite likely), then use a secondary sort criterion of their name. What is the best way to achieve this?

9. The benefit of anonymous classes is that you can use local variables from the outer method inside their code, and that the coupling is low (i.e. only the line of code which contains the class can use it, so any changes can be made in the assurance that there are no repercussions anywhere else).

10. Test your code.

11. Do the results of your performance tests earlier still hold true?