Carleton University
Department of Systems and Computer Engineering
SYSC 2004 - Object-Oriented Software Development - Winter 2015

Lab 4 - Grouping Objects

Objective

In this lab, you'll learn how to use an `ArrayList` object to store collections of objects.

Attendance/Demo

To receive credit for this lab, you must demonstrate your work. **Also, you must submit your lab work to cuLearn**. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will grade the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before Lab 5.**

References

Objects First with Java, Fifth Edition, Chapter 4, Sections 4.1 - 4.15 (class `ArrayList`), Chapter 5, Section 5.4 (class `Random`).

Overview of Class `ArrayList`

A summary of the methods in Java's `ArrayList` class can be found here:

http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

You'll need to read this documentation while you work on this lab, to determine which `ArrayList` methods your code should call.

Remember that, in order to use `ArrayList` objects, your code must first import the class:

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

Also, remember that there a couple of different ways we can iterate over an `ArrayList`. Suppose variable `names` refers to an `ArrayList` that is initialized with references to six `String` objects:

```
ArrayList<String> names = new ArrayList<String>();
names.add("Jack");
names.add("Gwen");
names.add("Ianto");
names.add("Owen");
names.add("Toshiko");
names.add("Rhys");
```

We can use a *for-each* `for` loop to iterate over the entire list, printing all of the names:

```
for (String name : names) {
    System.out.println(name);
}
```

During the first iteration of the loop, variable `name` is assigned the first list element (a reference to the `String "Jack"`), then the loop body is executed. During the second iteration, `name` is assigned the second list element (a reference to the `String "Gwen"`), and the loop body is executed a second time. This continues until every list element has been "visited".

Here is a `for` loop that is equivalent to the *for-each* loop:

```
for (int i = 0; i < names.size(); i += 1) {
    System.out.println(names.get(i));
}
```

Overview of Class `Random`

A summary of the methods in Java's `Random` class can be found here:

http://docs.oracle.com/javase/7/docs/api/java/util/Random.html

Remember that, in order to use `Random` objects, your code must first import the class:

```
import java.util.Random;
```

To create a new random number generator, we create an instance of class `Random`:

```
rand = new Random();
```

The `nextInt(n)` method returns a pseudorandom integer in the range 0 (inclusive) through `n` (exclusive). For example, this statement returns a pseudorandom integer in the range 0 through 9:

```
int k;
k = rand.nextInt(10);
```

Class `Random` provides similar methods for generating pseudorandom values of type `long`, `boolean`, `float` and `double`.

Getting Started

Step 1: Create a new folder named Lab 4.

Step 2: Download club.zip from cuLearn and move it to your Lab 4 folder.

Step 3: Right-click on club.zip and select Extract All... to extract all the files into a project folder named club.

Step 4: Open the club folder. Double-click on the BlueJ project file, which is named package. This will launch BlueJ and open the *club* project. A class diagram containing classes `Club`, `Membership` and `ClubDemo` will appear in the BlueJ main window.

Step 5: An incomplete implementation of the `Club` class has been provided in the *club* project. Your task is to complete the `Club` class.

If you try to compile the entire project by clicking the Compile button to the left of the class diagram, BlueJ will report compilation errors in class `ClubTest`, because methods in that class call methods that you have not yet defined in class `Club`.

There's no need to edit `ClubTest` to fix this. When working on the `Club` class, compile it by right-clicking on the `Club` class icon, then selecting Compile from the pop-up menu. Alternately, when you are editing `Club`, you can click the Compile button in the editor window. BlueJ will then compile `Club` without attempting to compile `ClubTest`.

After all the required methods have been defined in `Club` (when you're working on Exercise 7), you can compile the entire project by clicking the Compile button to the left of the class diagram.

Exercise 1 (based on OFwJ Exercise 4.40)

The `Club` class is intended to store `Membership` objects in a list collection.

Step 1: Open `Club` in the editor. Edit the lines that start with `@author` and `@version` (use your name and today's date).

Step 2: In class `Club`, define a field (instance variable) for an `ArrayList`. This list will store references to `Membership` objects. Remember to use an appropriate `import` statement.

Step 3: In the constructor, create the `ArrayList` object and assign the reference to this collection to the field.

Step 4: Make sure that the class compiles before moving on to the next exercise.

Exercise 2 (based on OFwJ Exercise 4.41)

Read the Javadoc comment for the `numberOfMembers` method. Currently, the method body has a single statement:

```
return 0;
```

Modify the `numberOfMembers` method to return the size of the collection. Until you have a method to add objects to the collection this method will always return 0, but it will be ready for further testing later.

Exercise 3 (based on OFwJ Exercise 4.42)

A membership in a club is represented by an instance of the `Membership` class. A complete version of `Membership` is already provided for you in the *club* project, and it should not need any modification, An instance contains details of a person's name, and the month and year in which they joined the club. All membership details are filled out when an instance is created.

Step 1: A new `Membership` object is added to a `Club` object's collection by calling the `Club` object's `join` method, which has the following description (Javadoc comment):

```
/**
 * Add a new member to the club's list of members.
 * @param member The member object to be added.
 */
public void join(Membership member)
```

Complete the `join` method.

Step 2: When you wish to add a new `Membership` object to the `Club` object from the object bench, there are two ways you can do this:

1. Create a new `Membership` object on the object bench, call the `join` method on the `Club` object, and click on the `Membership` object to supply the argument, or,

2. Call the `join` method on the `Club` object and type into the method's parameter dialog box:

```
new Membership("member's name...", month, year)
```

Remember, you have to provide the actual values for the member's name and the `month` and `year` arguments.

Test your `join` method by creating a new `Club` object on the object bench and adding three or four members to the club. Each time you add one, call the `numberOfMembers` method to check that the `join` method is adding `Membership` objects to the collection, and that the `numberOfMembers` method is giving the correct result.

While the `Club` object is on the object bench, create an inspector for this object. Using this inspector as a starting point, you can open inspectors for these objects: the `Club` object's `ArrayList`, the array used by the `ArrayList` (its name is `elementData`) and the `Membership` objects that are stored in the Club object's membership list (each array element stores a reference to one `Membership` object).

While all the inspectors are open, call `join` to add a new `Membership` object, and observe which objects change state.

Exercise 4

Define a method in the `Club` class with the following description:

```
/**
 * Determine if a specific person is a member of the club.
 * @param name The person's name.
 * @return true if that person is a member;
 * otherwise return false.
 */
public boolean hasMember(String name)
```

Make sure you copy the method's Javadoc comment (everything between the `/**` and `*/`) to your class, in addition to the method header. Your method is incomplete if you've written the method's code but haven't included the comment that describes the method's interface.

To determine if two `String` objects, `s1` and `s2`, are the same, don't use the expression:

```
s1 == s2
```

(This statement does not compare the characters in the two strings.)

The two `String` objects should be tested for equality with the boolean expression:

```
s1.equals(s2)
```

The expression will evaluate to `true` if the two strings are the same; otherwise it will be `false`.

Use the object bench to test your method interactively.

Exercise 5 (based on OFwJ Exercise 4.54)

Define a method in the `Club` class with the following description:

```
/**
 * Determine the number of members who joined in the
 * given month.
 * @param month The month we are interested in.
 * @return The number of members.
 */
public int joinedInMonth(int month)
```

If the `month` parameter is outside the valid range of 1-12, print an error message and return 0.

Remember to copy the Javadoc comment to your class.

Use the object bench to test your method interactively.

Exercise 6

Sometimes, a club member must be selected randomly to perform some task. Define a method in the `Club` class with the following description:

```
/**
 * Select a random member of the club.
 * @return The randomly selected member.
 */
public Membership pickAMember()
```

Remember to copy the Javadoc comment to your class.

Hint: use a `Random` object (see the summary on Page 2). Don't create a new `Random` object every time `pickAMember` is called. Instead, the `Random` object should be created when the `Club` object is initialized.

Use the object bench to test your method interactively.

Exercise 7 (based on OFwJ Exercise 4.55)

Define a method in the `Club` class with the following description:

```
/**
 * Remove from the club's collection all members who
 * joined in the given month, and return them stored
 * in a separate collection object.
 * @param month The month of the Membership.
 * @return The members who joined in the given month.
 */
public ArrayList<Membership> purge(int month)
```

If the `month` parameter is outside the valid range of 1-12, print an error message and return a collection object with no objects stored in it.

Remember to copy the Javadoc comment to your class.

Do not use a *for-each* loop in this method. You can't use a *for-each* loop to iterate over a list if statements in the loop body insert objects in or remove objects from a list. You will instead need to use a `for` loop in which you specify the index of each element in the membership list.

Before designing this method, read the API documentation for the `remove(int)` method in class `ArrayList`. Keep in mind that removing an object from an `ArrayList` shrinks the collection. For example, suppose a `Club` object's `ArrayList` contains references to six `Membership` objects. If `remove(2)` is called on the `ArrayList` to remove the object reference stored at index 2, the collection's size is reduced by one, to 5. Also, all the remaining object references, starting at index 3, are shifted one position "to the left" to close the gap created when the reference at index 2 is removed. In other words, after calling `remove(2)` on the `ArrayList`, calling `get(2)` on the `ArrayList` will return the reference that was previously stored at index 3. Similarly, calling `get(3)` on the `ArrayList` will return the reference that was previously stored at index 4.

Use the object bench to test your method interactively.

Exercise 8

Class `ClubTest` is a JUnit test class containing a *suite* of *test cases* that test class `Club`. Compile the project and run all the test cases (click the Run Tests button to the left of the class diagram). A Test Results dialogue box will appear, listing the test cases that were executed. If all the methods you wrote are correct, there should be green check-marks to the left of all the test cases. An x to the left of a test case indicates that it failed.

If any of the test cases fail, you'll need to locate and fix the bugs. Use an object inspector to help you determine where the problems are (e.g., before and after you execute a method, what values are stored in the object's fields? Which values are correct? Which values are incorrect? What section of code in the method you executed changes those values?) Edit the class and rerun the JUnit test cases until every test passes.

Wrap-Up

1. Review your `Club` class and make sure that the constructor and every method has a Javadoc comment. Make sure that your class has no more than two fields (instance variables). Novice Java programmers sometimes define fields when they should instead define variables that are local to methods. If your class has too many fields, you'll need to edit your class to fix this.

2. With one of the TAs watching, run the JUnit tests for `Club`. The TA will note how many test cases pass. The TA will review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the demo/sign-out sheet.

3. The next thing you'll do is package the project in a *jar* (Java archive) file named club.jar. To do this:

    3.1. From the menu bar, select Project > Create Jar File... A dialog box will appear. Click the Include source and Include BlueJ project files check boxes. A check-mark should appear in each box. Do not modify the Main class field.

    3.2. Click Continue. A dialog box will appear, asking you to specify the name for the jar file. Type club or select the BlueJ icon named lines in the list of files. **Do not use any other name for your jar file** (e.g., lab4, my_project, etc.).

    3.3. Click Create. BlueJ will create a file named lines that has extension .jar. (Note: you don't type this extension when you specify the filename in Step 3.2; instead, it's automatically appended when the jar file is created.) The jar file will contain copies of the Java source code and several other files associated with the project.

(The original files in your club folder will not be removed).

4. Before you leave the lab, log in to cuLearn and submit club.jar. To do this:

   4.1. Click the Submit Lab 4 link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the Add submission button. A page containing a File submissions box will appear. Drag club.jar to the File submissions box. Do not submit another type of file (e.g., a .java file, a RAR file, a .txt file, etc.)

   4.2. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status of your file is "Draft (not submitted)". If you're ready to finish submitting the file, jump to Step 4.4. If you instead want to replace or delete your "draft" file submission, follow the instructions in Step 4.3.

   4.3. You can replace or delete the file by clicking the Edit my submission button. The page containing the File submissions box will appear.

      4.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the File submissions box, then click the Overwrite button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the Save changes button.

      4.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the Delete button., then click the OK button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the Save changes button.

   4.4. Once you're sure that you don't want to make any changes to the project you're submitting, click the Submit assignment button. A Submit assignment page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the Continue button to confirm that you are ready to submit your lab work. This will change the submission status to "Submitted for grading".

Extra Practice Exercise

Currently, the `demo` method in the `ClubDemo` class adds two members to a club, then prints the number of members.

Modify this method to add several more members (including some who joined in the same month). Add code that demonstrates the behaviour provided by your `hasMember`, `joinedInMonth`, `pickAMember` and `purge` methods. Make sure you have cases where `joinedInMonth` returns 0 (no members joined in the specified month), 1 (one member joined in the specified month), and a value greater than 1 (multiple members joined in the specified month). The `demo` method should have enough `println` statements to demonstrate that your

`Club` class is correct.

Create a `ClubDemo` object on the object bench, call the `demo` method, and observe its output in the terminal window.