# Programming Structures and Abstractions (CS 2334)
# Project 3

October 13, 2010

## Introduction

**Serialization.** When executed, many of our programs create a set of objects in order to represent something with which we perform some computation. When the program stops executing, the memory that is used to store these objects is reclaimed for use by other programs (and hence the objects disappear). However, there are many situations in which the objects need to have some lifetime beyond the execution of our program. For example, the objects might form some database that we will want to query in the future. Or, the objects may be shipped to another computer on the network so that new work can be performed on them. In either case, the objects now have the property of *persistence* that extends beyond our program.

We have several ways of making objects persistent: we can write them into files or send them off to another computer on the network. In either case, the objects must be broken down into the fundamental unit of storage in the computer: the byte (or the bit, depending on your perspective). This process of transforming an object into a sequence of bytes for storage or transmission is called *serialization*. Before object oriented languages, serialization was performed in large part by the programmer, who had to explicitly break each component object into a specific sequence of bytes. This is particularly challenging when an object contains a reference to another object. In Java, we have a variety of abstractions that allow us, as programmers, to think/program at the level of objects, rather than their component pieces. In this project, we will make use of these facilities to store objects (specifically, FinchActionLists) in files so that they may be easily retrieved in the future.

**Extending the Java API.** Until this point, we have maintained a separation between the classes that we design and the classes that are defined by the Java API. For example, if a class of our own design involves a Map, then we declare an instance variable in our class that refers to a Map object. If our own class is conceptually a Map itself (with some added features), this approach is a bit awkward: our own class must expose, through its accessors/mutators, all of the key functionality of the internal Map.

On the other hand, it is possible for us to create new classes that *extend* the Java API classes. This ability is very powerful because it allows us to maintain the power of the classes themselves, but to customize their behavior to fit our particular needs. So, declaring our new class as extending a Map automatically endows our class with all of the features (including

methods) of the Map. As programmers, we are then left to simply add the new functionality. We will explore this idea in project 3 by changing our implementation of **FinchActionList** so that it extends a form of **LinkedList**. This idea will also be very prominent as we explore the Java Graphical User Interface facilities in the coming weeks.

# Objectives

By the end of this project, you should be able to:

- extend a class that is provided by the Java API to create a new class with custom functionality,
- read/write Java objects from/to a file,
- combine multiple collections of objects together to form a new collection (e.g., by "merging" the collections).

# Milestones

1. Use a **LinkedList** to represent **FinchActionList** (15 pts)
2. Show/execute FinchActions in both natural and reverse order (10 pts)
3. Add user command **write** that writes the FinchActionList out to a binary file (10 pts)
4. Add user command **read** that reads a new **FinchActionList** from a binary file (10 pts)
5. Add user commands **union** and **intersect** that combine the existing **FinchActionList** with a newly read one (15 pts)

Other components:

- Develop and use a proper design (UML and class stubs) (15 pts)
- Use proper documentation and formatting (javadoc and in-line documentation) (15 pts)
- Perform a short demonstration of your work (10 pts)
- Extra credit is possible – see below (up to 5 pts)

Note: of the design and documentation components, a total of 15 points are available during the design phase (7.5 points for each for the UML and the code stubs). The remaining 85 points are obtainable for the final submission of the project.

This lab is due in two phases:

1. Thursday, October $21^{st}$ at 5:00pm: design.
2. Thursday, October $28^{th}$ at 5:00pm: completed program and short demonstration. The early deadline, for a 5% bonus, is Tuesday, October $26^{th}$ at 5:00pm.

More details for what to hand-in and when may be found below.

# Resources

Main web page / projects / project3 :

- project3.pdf: this project description
- project3-slides.pdf: a copy of the lab section slides

# Input Files

We will keep the same input text file format as we used with project 2.

# Milestones

## Milestone 1: LinkedLists to represent FinchActionLists

Change your FinchActionList implementation so that it uses LinkedLists (instead of an array of FinchActions). Specifically:

- Declare FinchActionList class as follows:

```
public class FinchActionList extends LinkedList<FinchAction> implements Serializable
```

  This declaration makes **FinchActionList** a subclass of **LinkedList** – in other words, an instance of **FinchActionList** IS an instance of **LinkedList**. From within this new class, you will be able to invoke the methods provided by the parent class.

- Alter the set of properties that are explicitly contained within the FinchActionList (note that the key properties will now be handled by the LinkedList class).
- Alter FinchActionList so that it provides just the functionality that is required above and beyond LinkedList (i.e., you will have to remove or update some of your FinchActionList methods). For example, **sort()** now can use the implementation provided by the *Collections* class.
- Add a **FinchActionList** constructor that adheres to the following prototype:

```
public FinchActionList(FinchActionList list, String name)
```

This constructor creates a new **FinchActionList** that contains only those **FinchActions** in *list* that match *name*.

- Alter FinchAction to implement **Serializable**:

```
public abstract class FinchAction implements Comparable<FinchAction>, Serializable
```

- Eclipse will raise a warning about not defining **serialVersionUID**. This is a static variable that is used to check the compatibility between the current class definitions and the files. To make this warning go away, add the following to FinchActionList and all of your concrete child classes of FinchAction:

```
static final long serialVersionUID = 1138L;
```

The value that you use is not particularly important for our purposes (though you should understand the cultural importance of that particular number)...

- Create a **Milestone1.java** file that tests the functionality of your new implementation of **FinchActionList**.

## Milestone 2: Show/execute FinchActions in reverse order

In project 2, we defined a *natural order* to our **FinchActions**, and we defined methods to display and execute our actions in the natural order. Here, we will add functionality that allows a user to request that a list of actions be displayed/executed in reverse order.

Specifically:

- In your FinchActionList class, alter the implementations of the display() and execute() methods so that they adhere to these prototypes:

```
public void display(String name, boolean reverse)

public void execute(Finch myFinch, String name, boolean reverse)
```

The **reverse** parameter indicates whether the FinchActions should be displayed (or executed) in natural (**false**) or reverse (**true**) order. Natural order will still be determined by the **Comparable** property of the **FinchActions**.

- Create a **Milestone2.java** file that tests the functionality of these two new method implementations.

- In your driver class, allow the user to indicate whether to show/execute FinchActions in natural or reverse order.

Notes:

- execute() and display() perform the same set of operations – from the perspective of iterating through a list and checking to make sure that a name matches. This suggests that their core functionality should actually be built into a single method (as opposed to replicating it in two methods).
- LinkedList provides another type of iterator that will help in the implementation of the newly requested functionality.

  In all, you should be able to implement display/execute and forward/reverse using a single loop.

## Milestone 3: Add a "write" user command

In this milestone, we will add the ability for the user to write a FinchActionList to a file.

Specifically:

- Add the following method to **FinchActionList**:

```
public void write(String fname, String actionName)
```

  This method will open the specified file as an **ObjectOutputStream**, write the elements of the list that match *actionName* to the file, and close the file.

  Hints:

  - Remember that **FinchActionList** already provides the facilities to create a list that matches a String name.
  - You will only have to write a single object to the file.

- Give the user the ability to specify a **write** command from your text interface. The user can optionally specify the name with which to filter. Examples:

  - If the user types "write foo.dat" at the command line, then all **FinchActions** in the existing **FinchActionList** will be written to the file "foo.dat"
  - If the user types "write foo.dat dance" at the command line, then all **FinchActions** with a name of "dance" in the **FinchActionList** will be written to the file "foo.dat"

## Milestone 4: Add the "read"' user command

- Add the following constructor to **FinchActionList**:

```
public FinchActionList(String fname) throws IOException
```

This method will read a **FinchActionList** from the specified file. If there is an error, then it should *throw* an **IOException** (more on this soon).

- Create a **Milestone4.java** file that creates a small number of **FinchActions**, inserts them into a **FinchActionList**, writes them to a file, reads the list from the file, and displays the recently read list.
- Give the user the ability to **read** from text interface. Example:
  - If the user types "read foo.dat" into your text interface, then the newly read **FinchActionList** will replace the existing one.

## Milestone 5: Add the "union" and "intersect" commands

- Add the following method to **FinchAction**:

```
public boolean equals(Object o)
```

This method returns *true* if *this* and *o* are the same according to *compareTo*. You may assume that it is safe to cast the object to **FinchAction**. This method is used by methods such as *Collection.contains()*.

- Add the following methods to **FinchActionList**:

```
public FinchActionList union(String fname) throws IOException

public FinchActionList intersect(String fname) throws IOException
```

These two methods will first read a new **FinchActionList** from the specified file. For *union()*, the returned **FinchActionList** is the combination of *this* and the newly read list, with any duplicates removed.

For *intersect()*, the returned **FinchActionList** contains only those **FinchActions** that exist in both *this* and the newly read list.

For both methods: they must not alter *this*. Instead, any changes must be made to the newly created **FinchActionList**.

Hints:

- *Collection.addAll()* is useful for adding all of the elements from one collection to another.
- If a list is sorted by natural order, then duplicate entries will occur next to each other in the list. Duplicate entries can be identified using the *compareTo()* method.
- *Collection.contains()* is useful for finding out whether an object exists within a collection.

  - Iterators are your friend. Carefully examine the methods that they provide.

- Create a **Milestone5.java** file that tests the functionality of these two new methods (suggestion: follow the same strategy as with the previous milestone).

- Give the user the ability to **union** and **intersect** from the text interface. Examples:

  - If the user types "union foo.dat" into your text interface, then the existing **FinchActionList** will be replaced by a list that combines the existing one with the newly read one (i.e., the current list will be replaced by the one returned by *union()*).
  - If the user types "intersect baz.dat" into your text interface, then the existing **FinchActionList** will be replaced by a list that contains only those **FinchActions** that exist in both the current list and the newly read one (i.e., the current list will be replaced by the one returned by *intersect()*).

# Extra Credit

An additional 5 points (out of 100) are available for functionality that we do not explicitly request in this assignment. Possible additions include:

- Add a command that allows a user to specify that all selected actions change their priority by some specfied amount (e.g., add 9 to all of the priorities of actions that match a particular name).

- Move your text file reading procedure into a constructor of **FinchActionList**. Note that you will have to deal with one constructor that takes a file name parameter that can either be a text or a binary (object) file.

- Add *load* command that allows the user to load a new text file into the current **FinchActionList** (or modify *read* to handle both text and object files).

# Hand-In Procedures

## Part 1: Design

Deadline: Thursday, October $21^{st}$ at 5:00pm

Each group must hand in one copy of each of the following:

1. A printed cover page that lists the group members, any outside citations, and a plan for which group member will be primarily responsible for each class. Turn in the hardcopy to the TA or the lecturer.

2. UML diagram on engineering paper: turn in the hardcopy to the TA or the lecturer. The diagram must contain all of the relevant classes. However, **only detail the FinchActionList class (class variables and methods)**.

3. project3_design.zip to D2L. This is the zip file produced by Eclipse that contains:

   - Each of the classes with class variables, method header documentation and method stubs (prototypes). **If you are re-using methods from project 2, then it is okay to leave these implementations intact. However, for any new methods, only include dummy return calls or calls to this() or super(), and not any other code.**
   - html description of your project produced by javadoc. Make sure to check that the resulting html files contain all of the correct information.

### Part 2: Complete program and short demonstration.

Deadline: Thursday, October $28^{th}$ at 5:00pm. The early deadline, for a 5% bonus, is Tuesday, October $26^{th}$ at 5:00pm.

Each group must do the following:

1. Turn in a printed cover page that lists the group members, work contributed by each, and any outside citations. Turn in the hardcopy to the TA or the lecturer.
2. Hand in the modified UML diagram on engineering paper: turn in the hardcopy to the TA or the lecturer. The requirements are the same as for the design phase.
3. Turn in project3.zip to D2L. This is the zip file produced by Eclipse that contains:

   - Each of the class implementations with documentation. **Do not include your names in any of these files.**
   - html description of your project produced by javadoc.
4. A short demonstration. We will reserve time during your laboratory section for you to demonstrate your working program. You may also attend office hours or make appointments to perform the demonstrations.

## General Hints and Notes

- Your program must be your own work. Do not discuss or look at the solutions of other groups in the class. However, you may discuss general issues (i.e., not directly related to the project requirements) with your classmates, as well as use the book and the resources available on the net.
- Start your work early. This is not a trivial programming assignment.
- Ask for help early. If you are stuck on something, talk to the TA or the instructor sooner than later (this is what we are here for).