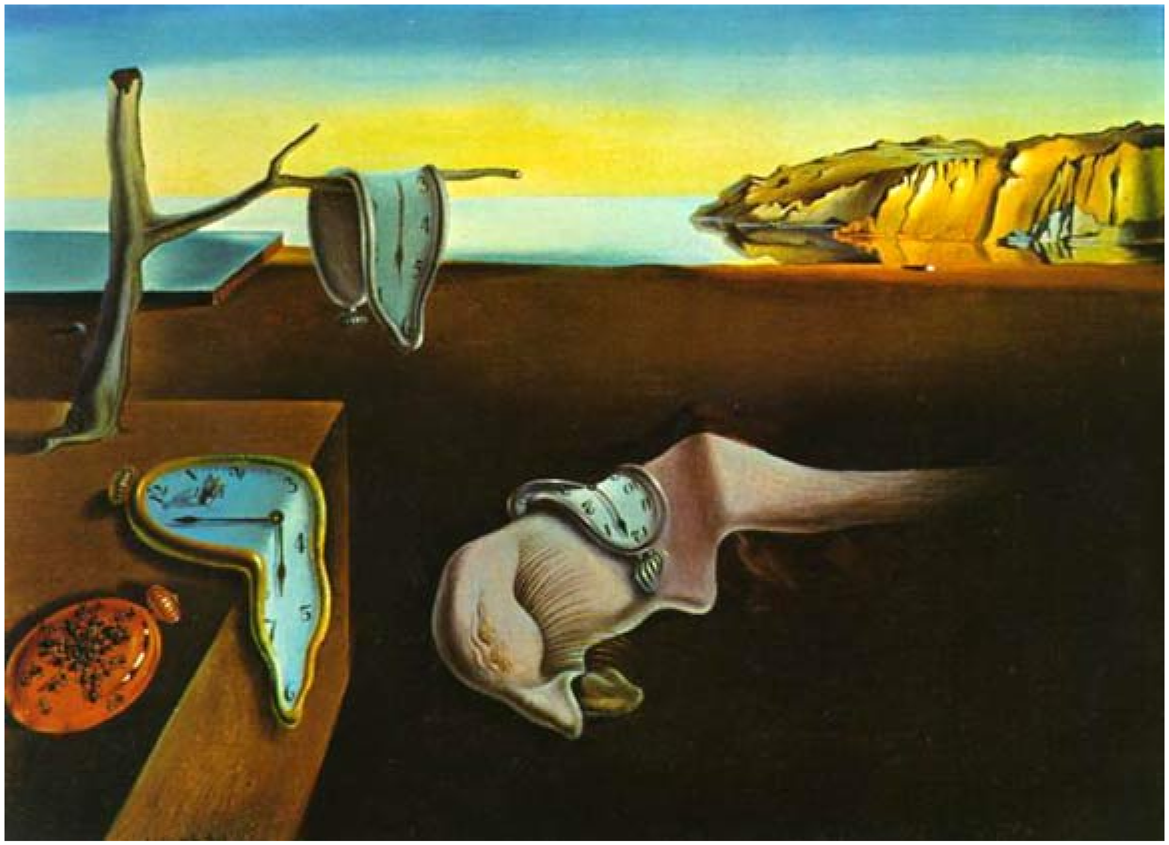
CS 2334: Project 3
Object Input/Output and Collections



Dali, 1931

Foci for Today

- Extending the core Java classes
- List iteration
- Project 3
  - Requirements
  - Get started on your design
- Project 2 demos can happen today, but project 3 is the priority (don’t spend the lab fixing project 2)

Extending Core Java Classes

Suppose I want to create a class called FinchSensorList

- This class must encapsulate a list of FinchSensor objects
- Various list operations must be defined

Extending Core Java Classes

There are two ways to implement such a class:

1. Create a new class from scratch
   - The list will be an instance variable of this class
   - Desired list operations will be manually defined by the new class

Example: Creating a New Class

```
public class FinchSensorList{
   private ArrayList<FinchSensor> fsList;

   public FinchSensorList(){
       fsList = new ArrayList<FinchSensor>();
   }
   public void add(FinchSensor fs){
       fsList.add(fs);
   }
   public FinchSensor remove(int index{
       return fsList.remove(index);
   }
   public boolean contains(FinchSensor fs){…}

   public int size(){…}
   …
}
```

Extending Core Java Classes

There are two ways to implement such a class:

1. Create a new class from scratch
   - The list will be a member variable of this class
   - Desired list operations will be manually defined

2. Extend an existing Java List class
   - Both the list and desired list operations will be defined already by the superclass
   - New functionality can be implemented, if desired

Example: Extending a Java List

```
public class FinchSensorList extends
  ArrayList<FinchSensor>{
  // We don’t need to define anything else here
  // We can override default Java functionality if we
  want
}
```

I can call any methods defined by ArrayList on an instance of FinchSensorList

Example: Extending a Java List

```
public class FinchSensorList extends
  ArrayList<FinchSensor>{
  // We don’t need to define anything else here
  // We can override default Java functionality if we
  want
}
```

- Example: suppose that fs1, fs2, etc. are FinchSensor objects:

```
FinchSensorList FSL = new FinchSensorList();
FSL.add(fs1); FSL.add(fs2);  // FSL = [fs1, fs2]
FSL.add(1,fs3);  // FSL = [fs1,fs3,fs2]
FSL.remove(2);  // returns fs2;  Now, FSL = [fs1,fs3]
```

Extending a Java List (cont.)

ArrayList methods can be called without referencing “this” or “super”

```
public class StringList extends ArrayList<String>{
  public void foo(){
      add(“foo”);
      add(“baz”);
  }
}
```

List Iteration

- Recall the *Iterator* interface:
  - *next*() – returns an element from the collection
  - *hasNext*() – there are more elements for *next*() to return
  - *remove*() – remove the element just returned by *next*() from the collection
- Every collection provides an iterator
- Lists can be traversed forwards and backwards
  - This is true for both arrays and doubly-linked lists

List Iteration (cont.)

ListIterator takes advantage of list sequentiality and defines additional methods for traversing lists

- *next*(), *hasNext*(), and *remove*() are the same as in *Iterator*

- *previous*() – returns the previous element in the list
  - *next*() traverses forward, while *previous*() traverses backward

- *hasPrevious*() – true if calling *previous*() would not return null

List Iteration (cont.)

ListIterator

- *nextIndex*() – returns the index of the element that would be returned by calling *next*()

- *previousIndex*() – equivalent of *nextIndex*() for *previous*()

- *set*(Object *o*) – replace the element just returned (by either *next*() or *previous*()) with *o*.

- *add*(Object *o*) – insert *o* into the list at the current iterator position

ListIterator Example

```
ArrayList<String> l = new
  ArrayList<String>();

l.add(“a”); l.add(“b”); l.add(“c”);
// l = [a, b, c]

ListIterator<String> li = l.listIterator();
// l = [∧a, b, c]
```

Project 3 Objectives

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

- Extend classes defined by the Java API
- Read/Write Java objects from/to a file
- Merge multiple collections of objects to form a new collection

Milestones

1. Use a LinkedList to represent FinchActionList
   - FinchActionList now extends LinkedList
   - LinkedList provides add() and iteration()
   - Your extended class still provides execute() and display()

Milestones

2. Display/Execute FinchActions in both natural and reverse order.

   Update FinchActionList:

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

User commands access these new methods

Milestones

3. Add a new command “write” that allows the user to save the current FinchActionList to a binary file

   Update FinchActionList:

   ```
   void write(String fileName,
              String actionName)
   ```

Note: object I/O will be covered in lecture on Friday & Monday

Milestones

4. Add the “read” user command to load a FinchActionList from a binary file

   Update FinchActionList with new constructor:

   ```
   FinchActionList(String fileName)
   ```

Milestones

5. Add the “union” and “intersect” user commands

Update FinchActionList:

```
FinchActionList union(String fileName)
FinchActionList intersect(String fileName)
```

Each of these methods first reads a new FinchActionList from the specified file and combines it with the current FinchActionList.

Milestones

1. Use a LinkedList to represent FinchActionList
2. Show/Execute FinchActions in both natural and reverse order
3. Add a new command “write” that allows the user to save the current FinchActionList to a binary file
4. Add the “read” user command to load a FinchActionList from a binary file
5. Add the “union” and “intersect” user commands

New for this Project

- Designs must include a plan for which group member will implement which classes
  - This person should be the one primarily at the keyboard during implementation and testing in the next phase
- UML diagrams:
  - Still show class relationships
  - Only show details for the FinchActionList class

Extra Credit!

- There are new opportunities for extra credit if you make creative improvements to your project (up to 5 points)
  - See the project 3 specification for suggestions

- As always, early demos (Oct 26th by 5pm) receive 5% extra credit

Deadlines

- October 21st @5:00pm: design
- October 28th @5:00pm: final version, including demonstration
  - If all elements are completed by October 26th @5:00pm, a 5% bonus will be awarded