CS 2334: Lab 4
Generics and Collections

Lab 4 Objectives

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

1. use Java Lists (specifically, the ArrayList class) to store and manipulate a list of objects of a specified type,
2. create Comparator object classes that enable a sorting of the ArrayList in different ways
3. use the ArrayList to compute statistics over the objects stored in the list

Collections, Lists, and Sets

- *Collection* is an interface that captures the notion of a group of objects, and specifies some operations upon them
- *List* and *Set* are both subinterfaces of *Collection*
  - A List is an *ordered* collection of objects
    - Can have duplicate elements
  - A Set is a collection of objects with no duplicate elements
    - No meaningful notion of object order

List Operations

- Using a List, one can (among many things):
  - Append an object
  - Add an object at a specific location
  - Return an object at a specific location
  - Remove an object at a specific location
- For full details, consult the Java API

Concrete Implementations of Lists

- ArrayList<E>
- LinkedList<E>
- Stack<E>
- Vector<E>

ArrayList Example

```
ArrayList<String> lst = new ArrayList<String>;

lst.add(“a”); lst.add(“b”); lst.add(“c”);
// At this point lst = [“a”,“b”,“c”]

lst.add(“d”);     // lst = [ a, b, c, d]
lst.add(2,“e”);   // lst = [a, b, e, c, d]
lst.set(4,“g”);   // lst = [a, b, e, c, g]
lst.get(4)        // Returns “g”
lst.indexOf(“e”) // Returns 2
lst.remove(4);    // lst = [a, b, e, c]
lst.remove(“e”); // lst = [a, b, c]
```

Iterators

- Iterators provide the means to enumerate the elements of a collection
- Iterators encapsulate the details of a particular data structure, and present the user with a uniform interface
- Example: a List can be enumerated by starting at the front of the List, then moving from one element to the next, until the end of the List is reached.
- How does one enumerate the elements of a tree?

The *Iterator* Interface

Iterator defines the following methods:

- next() – returns the next item in the collection
- hasNext() – the iterator has a next element to return
  - I.e., there are still more elements to iterate through
- remove() – removes the last element returned by the iterator
  - This is optional; don’t worry about it for now

Using Iterators

```
ArrayList<String> lst = new ArrayList<String>;
ListIterator li;

lst.add(“a”); lst.add(“b”); lst.add(“c”);

li = lst.iterator();
li.hasNext()    // Returns True
li.next()       // Returns “a”
li.hasNext()    // Returns True
li.next()       // Returns “b”
li.hasNext()    // Returns True
li.next()       // Returns “c”
li.hasNext()    // Returns False

li.next()       // Run-time Exception!!
```

Implicit Iterators

- Iterators can be used implicitly with the For-Each syntax
- Formal example:

```
for (T x: lst){
      foo(x);
}
```

  - There is a collection *lst*
  - For each element *x* in *lst* of type *T*, call *foo(x)*
- The For-Each syntax implicitly refers to the iterator of *lst*, so the user doesn't have to

Implicit Iterators (cont.)

- Consider the ArrayList<String> lst again, where lst contains [“a”, “b”, “c”]
- When the following code is executed,

```
for (String s: lst){
    System.out.println(s);
}
```

The following output is printed:

```
a
b
c
```

Comparable Interface

- The Comparable interface defines an ordering by which a List of objects can be sorted using:

```
ArrayList<String> lst = new
  ArrayList<String>;
// Add some elements to lst

Collections.sort(lst);
// lst is now sorted
```

- The compareTo() method defined by String determines a ***natural ordering*** of the String objects

Comparators

- However: we may wish to sort objects differently in different contexts
- Example: a list containing student information objects may be sorted by either height, age or shoe size

Comparators (cont.)

- We want to be able to define different orderings without having to change our the class implementation
- Comparators allow the ordering of objects to be *decoupled* from the implementation of those objects
- A comparator induces a new order upon a collection of objects

Comparators (cont.)

- A comparator extends the *Comparator* interface
  - `public int compare(Object o1, Object o2)`
- If o1 is to be ordered before o2, then
  - `c.compare(o1,o2) < 0`
- If o1 is to be ordered after o2, then
  - `c.compare(o1,o2) > 0`
- Otherwise (they are equal)
  - `c.compare(o1,o2) == 0`

Comparators (cont.)

- There is also a generic form of Comparator
- Syntax:

```
Comparator<T>
```

- For example, from Lab4:

```
SensorComparator implements Comparator<FinchSensor>
```

- This will allow for generic code and compile time checking

General “To Do”

- Download Lab4.zip
- Analyze the code (and draw the UML diagram)
- Answer questions for Milestone 2
- Provide implementations for: TemperatureComparator, ZaccelComparator, AccelComparator, and LightComparator
  - Implement compare, doubleValue, and toString
- Implement median() and mean() in sensorDriver
- Perform experiments
- Demonstrate