Highlights of Previous Lecture

- Final Project Goals
  1. Set up collections of Flights
  2. Maintain information about reservation availability on flights
  3. Respond to reservation requests
  4. Set up collections of Reservations on Flights
  5. Support the Flight and Reservation interfaces so that my code can invoke methods defined by them.

- Designing Objects / Object Semantics
- Designing Object Fields and Methods
Project issues
Declaring your own Exception classes

file:../week04/UnknownStudentException.java

/** This exception is thrown when a student or student identifier does ** not correspond to a real student.  
 ***/

public class UnknownStudentException extends Exception {

    public UnknownStudentException() {
    }

    public UnknownStudentException(String msg) {
        super(msg);
    }
}

Using Exceptions

Used: as in ../week04/CSCourse.java

/** Assign a homework grade to the student identified by id for the specified assignment number. **/
public void setHomeworkGrade( Object id, int assignment, String grade)
throws UnknownStudentException {
    // Find the student object whose id is id
    CSStudent s = (CSStudent) students.get(id);
    if (s != null) {
        s.assignHomeworkGrade(assignment, grade);
    }
    else {
        throw new UnknownStudentException(id.toString());
    }
}

And, in ../week04/CSTest.java

    // Exercise the exception handling
String fakeId = "000-00-0000";
try {
    cs332.setHomeworkGrade(fakeId, 1, "A");
}
catch (UnknownStudentException e) {
    System.err.println(e);
    // Or something more
}
Software Systems Change - Design for It

Most systems undergo numerous changes after their first delivery. Any model of software development that only considers the period leading to that delivery and ignores the subsequent era of change and revision is as remote from real life as those novels which end when the hero marries the heroine -- the time which, as everyone knows, marks the beginning of the really interesting part. -- Bertrand Meyer

Symptoms of a Bad Design*

- **Rigidity** - software that is difficult to change.

  Every change causes a cascade of other changes to be required

- **Fragility** - software that breaks every time something is changed

  Breakage occurs in places that (should) have little or no relationship to where the change occurred.

- **Immobility** - inability to reuse software from other projects or from parts of the same project.

  Modules contain too much unnecessary baggage.

- **Viscosity** of design - when faced with the need to make a change you have two choices: changes that preserve the design and changes that are just hacks to get the change accomplished.

  Viscosity of design is high when

  - ... the hacks are easier than the design preserving changes.
  - ... "it is easy to do the wrong thing, but hard to do the right thing."
* - from Robert C. Martin - Design Principles and Design Patterns
http://www.objectmentor.com/publications/Principles%20and%20Patterns.PDF
Principles of Object Oriented

Some approaches to object oriented implementation have been found to lead to better designs:

- The Open-Closed Principle - OCP - A class should be extensible without requiring modification
- The Liskov Substitution Principle - LSP - Derived classes should be substitutable for their base classes
- The Dependency Inversion Principle - DIP - Depend upon abstractions. Do not depend upon concretions
- The Interface Segregation Principle - ISP - Many client specific interfaces are better than one general purpose interface.
The Open-Closed Principle

A module (class) should be open for extension but closed for modification.

- Classes should be written so that they can be extended without requiring the classes to be modified.
- Enables one to change what classes do without changing the classes’ source code.
- To extend the behavior of a system (in response to a requested change) add new code, don’t modify existing code.
- Abstraction (and polymorphism) are the keys to the Open Closed Principle.
Use Polymorphism

"the ability to manipulate objects of distinct classes using only knowledge of their common properties without regard for their exact class."

Using interfaces and abstract classes are the keys to polymorphism
Liskov Substitution Principle

Subclasses should be substitutable for their base classes

- A user of a base class should still function if given an instance of a derived class instead.
- "A derived class should have some kind of specialized behavior (it should provide the same services as the superclass, only some, at least, are provided differently.)"
- The contract of the base class must be honored by the derived class.
- If this principle is violated, then you will cause the Open-Closed Principle to be violated also. Why?
The Dependency Inversion Principle

*Depend upon abstractions. Do not depend upon concretions*

- High level classes should not depend on low level classes
- Abstractions should not depend upon the details
- If the high level abstractions depend on the low level implementation details, then the dependency is inverted from what it should be.
- High level classes contain the "business logic", low level classes are the details of the (current) implementation.
- **Depend upon interfaces or abstract classes to avoid dependency inversion.**
Procedural vs OO Dependency

Typical, procedural style dependencies

Correct Object Oriented Dependency

If higher levels of the application are not going to depend on particular, low level implementations, how does one construct the necessary low level implementations?
The Interface Segregation Principle

Many client specific interfaces are better than one general purpose interface.

- If you have a class with several different uses, create separate (narrow) interfaces for each use.
- The alternative is that all uses share the same, large interface (and most uses will need only a small number of the methods.)
- Instead the uses of the class can be organized as groups of related methods, each of which serves a different purpose. Each of these groups becomes a separate interface.
- The purpose is to make clients use as small and as coherent an interface as possible.
- Fat interfaces lead to inadvertent couplings between classes: dependencies that are accidental.
Java2 Collection Framework

- Interfaces - generic ways of treating different collections alike
- Abstract classes that provide starter / partial implementations
- Full implementations of some useful data structures
- Built-in implementations of efficient algorithms for
  - sorting,
  - searching,
  - "bulk" operations: perform an operation on an entire Collection in a single method call, e.g., set operations.
- Consistent set of interfaces consistently implemented
Core Collections Interfaces

- **Collection** - an interface that defines methods for adding, removing, and querying a data structure
- **List** - an ordered collection, possibly containing duplicates, with methods for adding, removing, accessing a portion
- **Set** - a collection without duplicates, with methods for doing set operations: intersection, difference
- **SortedSet** - A set that guarantees that its iterator will traverse the set in ascending element order
- **Map** - for storing key/value pairs, with methods that to access keys, values, or key/value pairs
- **SortedMap** - A map that guarantees that it will be in ascending key order
Collection Framework Interfaces

This illustrates the relationship between the Core Collection interfaces.
Collection Interface Implementations

From Collections Framework Overview

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Hash Table</th>
<th>Resizable Array</th>
<th>Balanced Tree</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>HashSet</td>
<td></td>
<td>TreeSet</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td></td>
<td>ArrayList</td>
<td></td>
<td>LinkedList</td>
</tr>
<tr>
<td>Map</td>
<td>HashMap</td>
<td></td>
<td>TreeMap</td>
<td></td>
</tr>
</tbody>
</table>
Collections Examples

- Collections.sort(List l, Comparator c)
- Collections.binarySearch(List l, Object key, Comparator c)
- Collections.min(Collection c, Comparator c)

ItineraryCollections.java

```java
import java.util.List;
import java.util.Collections;
import java.util.Comparator;
import java.util.Date;
import reservations.Airport;
import reservations.Itinerary;
import reservations.ItineraryReader;

public class ItineraryCollections {

    private void createCollection(List list) {
        try {
            ItineraryReader reader = new ItineraryReader();
            Itinerary i;
            while ((i = reader.nextItinerary()) != null) {
                list.add(i);
                // Itinerary will insist that they are registered
                // with a Flight before permitting any methods.
                new MinimalFlightImpl(i);
            }
        } catch (java.io.IOException e) {
            System.err.println(e);
        }
        System.out.println("list using " + list.getClass().getName());
    }

    /** This method will demonstrate the collection capabilities using
       ** Itinerary’s natural ordering. */
    private void demoItineraryCollection(List list) {
        try {
            // This throws a ClassCastException (because the
```
// objects in the list do not implement the Comparable
// interface.)
Collections.sort(list);
System.out.println("sorted:");
System.out.println(list);

Object min = Collections.min(list);
System.out.println("min:" + min);
}
catch (ClassCastException e) {
    System.err.println("Bzzt! List elements do not support Comparable");
}

/** A helper class (functor) that defines an Itinerary
** ordering by starting airport code.
**/ class OrderByStartingAirport implements Comparator {
    public int compare(Object o1, Object o2) {
        if (o1 instanceof Itinerary && o2 instanceof Itinerary) {
            Itinerary i1 = (Itinerary) o1;
            Itinerary i2 = (Itinerary) o2;
            Airport a1 = i1.getStartingAirport();
            Airport a2 = i2.getStartingAirport();
            String s1 = a1.getCode();
            String s2 = a2.getCode();
            return s1.compareTo(s2);
        } else
            throw new ClassCastException("not a Itinerary");
    }
    public String toString() {
        return "comparison by StartingAirport code";
    }
}

/** This method will demonstrate the collection capabilities using
** Itinerary’s order specified by the above helper class
**/ private void demoSAItineraryCollection(List list) {
    OrderByStartingAirport comparer = new OrderByStartingAirport();
    Collections.sort(list, comparer);
    System.out.println("sorted " + comparer);
    System.out.println(list);

    Object min = Collections.min(list, comparer);
    System.out.println("min:" + min);
/** A helper class (functor) that defines an Itinerary ordering by Destination airport city name. **/ class OrderByDestinationAirport implements Comparator {
    public int compare(Object o1, Object o2) {
        if (o1 instanceof Itinerary && o2 instanceof Itinerary) {
            Itinerary i1 = (Itinerary) o1;
            Itinerary i2 = (Itinerary) o2;
            Airport a1 = i1.getDestinationAirport();
            Airport a2 = i2.getDestinationAirport();
            String s1 = a1.getCity();
            String s2 = a2.getCity();
            return s1.compareTo(s2);
        }
        else
            throw new ClassCastException("not a Itinerary");
    }
    public String toString() {
        return "comparison by Destination Airport city";
    }
}

/** This method will demonstrate the collection capabilities using ** Itinerary’s order specified by the above helper class **/ private void demoDCItineraryCollection(List list) {
    OrderByDestinationAirport comparer = new OrderByDestinationAirport();
    Collections.sort(list, comparer);
    System.out.println("sorted + comparer");
    System.out.println(list);
    Object min = Collections.min(list, comparer);
    System.out.println("min:" + min);
}

/** A helper class (functor) that defines an Itinerary ordering by departure time. **/ class OrderByDepartureTime implements Comparator {
    public int compare(Object o1, Object o2) {
        if (o1 instanceof Itinerary && o2 instanceof Itinerary) {
            Itinerary i1 = (Itinerary) o1;
            Itinerary i2 = (Itinerary) o2;
            Date d1 = i1.getDepartureTime();
            Date d2 = i2.getDepartureTime();
            return d1.compareTo(d2);
        }
        else
            throw new ClassCastException("not a Itinerary");
    }
    public String toString() {
        return "comparison by departure time";
    }
}
return d1.compareTo(d2);

else
    throw new ClassCastException("not a Itinerary");

public String toString()
    return "comparison by Departure Time";
}

/** This method will demonstrate the collection capabilities using
** Itinerary’s order specified by the above helper class
***/
private void demoDTItineraryCollection(List list) {
    OrderByDepartureTime comparer = new OrderByDepartureTime();
    Collections.sort(list, comparer);
    System.out.println("sorted "+ comparer);
    System.out.println(list);

    Object min = Collections.min(list, comparer);
    System.out.println("min:" + min);
}

public static void main(String[] args) {
    ItineraryCollections c = new ItineraryCollections();
    List list;
    if (args.length == 0)
        list = new java.util.ArrayList();
    else
        list = new java.util.LinkedList();
    c.createCollection(list);
    c.demoItineraryCollection(list);
    c.demoSAItineraryCollection(list);
    c.demoDCItineraryCollection(list);
    c.demoDTItineraryCollection(list);
}
}
Homework
Final Project