Highlights of Last Week

- Refactoring classes to reduce coupling
- Passing Object references to reduce exposure of implementation
- Exception handling
- Defining/Using application specific Exception types
**Sample Homework Solution - PlaneId**

**PlaneId.java**

```java
/** A class that maintains a relationship between an Airplane and ** and Airline to identify a particular plane **/

public class PlaneId {

/** Identify an Airline / Airplane combination */
private String planeIdentifier;

public PlaneId (Airline airline, Airplane plane) {
    // PlaneId internally is simply the space separated
    // concatenation of the airline name with the plane id
    planeIdentifier =
        airline.getName() + " " + plane.getId();
}

/** Returns a hashcode that identifies this PlaneId */
public int hashCode() {
    return planeIdentifier.hashCode();
}

/** Compares this PlaneId to the specified object */
public boolean equals(Object o) {
    if (o instanceof PlaneId) {
        PlaneId po = (PlaneId) o;
        if (hashCode() == po.hashCode())
            return true;
    }
    return false;
}

/** Return a string identifing this PlaneId instance */
public String toString() {
    return planeIdentifier;
}
```

2
protected String getPlaneIdentifier() {
    return planeIdentifier;
}
}
Sample Homework Solution - AirTrafficController

AirTrafficController.java

```java
import java.util.HashMap;
import java.util.Set;                   // interface
import java.util.Iterator;              // interface

public class AirTrafficController {

    private HashMap planeIdsBeingMonitored = new HashMap();

    /** Causes the AirTrafficController to remember a planeid. **/
    public void monitor(PlaneId pid) throws InvalidPlaneIdException {
        if (planeIdsBeingMonitored.containsKey(pid)) {
            // The plane id is already being monitored by this
            // AirTrafficController.
            // This is probably an error.

            throw new InvalidPlaneIdException(pid.toString() +
                " is already being monitored");
        } else {
            // Remember this planeid
            planeIdsBeingMonitored.put(pid, pid);
        }
    }

    /** Provide some way for the AirTrafficController to stop
    ** monitoring a PlaneId. **/
    public PlaneId remove(PlaneId id) {
        return id;
    }
```
public Iterator getPlaneIds() {
    Set s = planeIdsBeingMonitored.keySet();
    return s.iterator();
}

class Monitoring {
    // Implementation details...
}

class Monitoring {
    public String toString() {
        Set s = planeIdsBeingMonitored.keySet();
        Iterator i = s.iterator();
        StringBuffer b = new StringBuffer("Monitoring: ");
        boolean first = true;
        while (i.hasNext()) {
            PlaneId p = (PlaneId) i.next();
            if (! first)
                b.append(",
            else
                first = false; // from here on
            b.append(p.toString());
        }
        return b.toString();
    }
}
Sample Homework Solution - InvalidPlaneIdException

InvalidPlaneIdException.java

/** This exception is thrown when a plane id is invalid for some reason. */
public class InvalidPlaneIdException extends Exception {

    public InvalidPlaneIdException() {
    }

    public InvalidPlaneIdException(String msg) {
        super(msg);
    }
}
Sample Homework Solution - AirTrafficTest

AirTrafficTest.java

import java.util.Iterator;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Collection;  // interface
import java.util.Set;       // interface

/** A class that will be used to test Airline, Airplane,
 ** PlaneId and AirTrafficController capabilities.
 **/ public class AirTrafficTest {

 /** Keeps track of airline names and the number of 
 ** airplanes for each.
 **/ private HashMap airlineInfo = new HashMap();

 /** Keeps track of the airline names and the 
 ** corresponding Airline objects themselves.
 **/ protected HashMap airlines = new HashMap();

 /** The two Maps somewhat duplicate the information 
 ** that is kept for an Airline in two places. A 
 ** better approach might be to create a helper class, 
 ** instances of which would encapsulate the 
 ** information we want to maintain for an Airline. 
 ** However, this way keeps things a bit simpler.
 **/ 

 /** Keeps track of the AirTrafficControllers */
 protected ArrayList controllers = null;

 /** Constructor */
 protected AirTrafficTest() {
 }

 /** Creates the desired number of 
 ** AirTrafficControllers.
 */
```java
/**
void createControllers(int numControllers) {
    controllers = new ArrayList(numControllers);
    for (; numControllers > 0; numControllers--) {
        AirTrafficController c = new AirTrafficController();
        controllers.add(c);
    }
}

/** Save an airline name and the number of planes the
** airline will have.
**/
void saveAirlineInfo(String name, int numPlanes) {

    // put into the name-th slot of the airlineInfo Map
    // the number of planes that the airline will have.
    // Note only objects can be saved in a collection, so
    // the number of planes is wrapped in an instance of
    // java.lang.Integer. No allowance is made for a
    // duplicate airline name; later occurrences of the
    // name simply replace previous occurrences.

    airlineInfo.put(name, new Integer(numPlanes));
}

/** Iterate through the Map of airlineInfo and create
** airlines, saving a reference to each Airline
** created. These references are saved simply to
** enable them to be identified later.
**/
void createAirlines() {

    Set s = airlineInfo.keySet();       // set of airline names
    Iterator iter = s.iterator();       // one at a time
    while (iter.hasNext()) {

        // Create a new Airline
        String name = (String) iter.next();
        Airline a = new Airline(name);

        // and save it to give airplanes to later
        airlines.put(name, a);
    }
    // What happens to the Set and the Iterator objects?
```
/** Iterate through the Map of airlineInfo and create
** the desired number of airplanes, adding them to the
** airline.
**/
void createAirplanes() {

Set s = airlineInfo.keySet();       // set of airline names
Iterator iter = s.iterator();       // one at a time
int serialNum = 1;                  // for airplane ids
while (iter.hasNext()) {

    String name = (String) iter.next();

    // airlineInfo map associates airline names with
    // number of airplanes requested
    Integer numPlanes = (Integer) airlineInfo.get(name);

    // airline map associates airline name with the
    // airline object
    Airline airline = (Airline) airlines.get(name);

    for (int i = numPlanes.intValue(); i > 0; i--) {

        // Create the next Airplane, giving it the next
        // serial number for an id (with the serial
        // number represented as a String.
        Airplane plane = // sequentially "number" airplanes
                         new Airplane(Integer.toString(serialNum++));

        // add the airplane to the airline
        airline.addAirplane(plane);
    }
}

/** Iterate through the Map of airlines, ask each for
** an iterator of its planes, and create PlaneId
** objects for each assigning those to
** AirTrafficControllers.
**/
void assignAirplanes() {

Set s = airlines.keySet();
Iterator airlineIter = s.iterator();
int controllerIndex = 0;

while (airlineIter.hasNext()) {
    String name = (String) airlineIter.next(); // next airline name
    // Get the next airline object by name
    Airline airline = (Airline) airlines.get(name);
    Iterator airplaneIter = airline.airplaneIterator();
    while (airplaneIter.hasNext()) {
        Airplane plane = (Airplane) airplaneIter.next();

        // Create the PlaneId
        PlaneId pid = new PlaneId(airline, plane);

        // Assign it to an AirtrafficController
        AirTrafficController c =
            (AirTrafficController) controllers.get(controllerIndex);
        try {
            c.monitor(pid);
        }
        catch (InvalidPlaneIdException e) {
            // Shouldn’t happen
            System.err.println(e);
        }

        // Get ready to assign next
        controllerIndex++;
        controllerIndex = controllerIndex % controllers.size();
    }
}

void identifyAirlines(java.io.PrintStream out) {

    // Cause the Airline objects and the Airplane objects to
    // identify themselves and, in the case of the Airplanes,
    // their current cost per seat.

    Collection c = airlines.values();
    Iterator airlineIter = c.iterator();
    while (airlineIter.hasNext()) {
        Airline a = (Airline) airlineIter.next();
        out.println(a.toString());
    }
void identifyControllers(java.io.PrintStream out) {

    // Cause the AirTrafficController objects to identify
    // themselves. They will in turn identify the
    // PlaneIds they are monitoring.

    for (int i = 0; i < controllers.size(); i++) {
        AirTrafficController atc =
            (AirTrafficController) controllers.get(i);
        out.print("Controller " + i + " ");
        out.println(atc.toString());
    }
}

void performTests(String[] args) {
    int arg = 0;
    int numControllers = 0;

    // Parse the application’s arguments from the command
    // line. The Integer.parseInt method calls (to
    // convert the String arguments into int primitives)
    // may result in NumberFormatExceptions being thrown.

    try {
        // First, determine how many controllers are desired
        numControllers = Integer.parseInt(args[0]);
        createControllers(numControllers);

        // Next, save the names of the airlines and the
        // desired number of planes for each.

        for ( arg = 1; arg < args.length; arg+= 2) {
            // Name of airline
            String name = args[arg];
            // Number of planes
            int numPlanes = Integer.parseInt(args[arg+1]);
            saveAirlineInfo(name, numPlanes);
        }
    }
    catch (NumberFormatException e) {
        System.out.println(args[arg+1] + " not an integer");
        e.fillInStackTrace(); // record where this occurred
        throw e; // bailout and let JVM report it
    }
}
// Time to create the airlines
createAirlines();

// and their airplanes
createAirplanes();

// Assign Airplanes to controllers
assignAirplanes();

// show the state of the air traffic controllers
identifyControllers(System.out);
}

public static void main(String[] args) {

String usage =
    "usage: AirTrafficTest number_of_controllers " +
    "[airline_name number_of_planes] ...";

if  ( args.length < 3 || args.length % 2 == 0 ) {
    System.out.println(usage);
} else {

    AirTrafficTest tester = new AirTrafficTest();
    tester.performTests(args);
}
}
Inheritance

- classes that are defined in terms of other classes

  ● You say one class extends another when you want to re-use most of the capabilities of a class, but want to add some additional capabilities, over-ride some, or provide other specialization.
  ● The purpose of inheritance is specialization.
  ● Inheritance models the "IS-A" (or "IS-A-KIND-OF") relationship
  ● Textbook Issue, p 127: "This is one of the significant ways to achieve productivity with an object-oriented language: find a built-in class that does much of what you need, and create a subclass of that class, adding just those features that you need for your own purposes; ..."

Guideline: Not unless your subclass "IS-A-KIND-OF" the parent class.
java.lang.Object

Every Java class implicitly extends Object.

The least specialized class is sometimes called the base, parent, or super class.

Student.java

/** A Java class that models information common to all student types.  
 ** @author Jonathan Doughty  
 ***/
public class Student /* extends Object */ {

    // Fields associated with an individual Student object.
    private String      name;
    private String      id;
    private double      tuitionPaid = 0.0;

    // Constructor for Student objects
    public Student(String newStudentName, String id) {
        name = newStudentName;
        this.id = id;
    }

    /** Method called when asking Student’s to pay their bills  
    ***/
    public void payTuition(double amount) {
        tuitionPaid += amount;
        // remainder of paying tuition left as an exercise
    }

    public String getName() {
        return name;
    }
Object getId() {
    return id;
}

public String toString() {
    String result = "Student name: " + name + " Id: " + id;
    result += " tuition paid:" + tuitionPaid;
    return result;
}
// ... other methods common to all Students
Inheriting methods

A more specialized class extends the super class. The more specialized class is sometimes called the derived or child class.

CSStudent.java

import java.util.HashMap;

/** A Java class that models information kept about a CS student. This CSStudent class inherits capabilities that all students share from the base class. ** @author Jonathan Doughty ** @version 3.0 ***/
public class CSStudent extends Student {

    // Fields associated with the CSStudent specialization of Students.
    private CSStudent   labPartner = null;
    private HashMap     homeworkGrades = new HashMap();

    // Constructor for CSStudent objects
    public CSStudent(String newStudentName, String id) {
        super(newStudentName, id);
    }

    // Look mom, no payTuition() method
    // Nor getName(), getId() either.
    // Derived classes get those for free.

    public String toString() {
        String endl = System.getProperty("line.separator");
        String result = super.toString();
        if (labPartner != null)
            result += " Lab partner: " + labPartner.getName();
        return result + endl;
    }
}
if (homeworkGrades.size() > 0) {
    result += endl + "Homework grades: ";
    for (int i = 0; i < homeworkGrades.size(); i++) {
        Integer k = new Integer(i);
        result += "" + i + "-" + homeworkGrades.get(k);
    }
    return result;
}

// Since these are "package protected" (without an explicit
// accessibility modifier) only classes in the same package can
// invoke these methods.

void assignPartner(CSStudent partner) {
    labPartner = partner;
}

void assignHomeworkGrade(int assignmentNumber, String grade) {
    Integer assignment = new Integer(assignmentNumber);
    homeworkGrades.put(assignment, grade);
}
Derived classes

- Derived classes inherit methods from their base class(es)
- There is no requirement, if the derived class is not going to add any specialization to a method defined in the base class, for the derived class to have an implementation of the method.
- The derived class should certainly not duplicate the logic of the base class’s method implementation.
Super

- The `super` keyword allows a class to use the `immediate` parent class’ constructors and to access accessible over-ridden fields, and methods.
- The derived class only has access via `super` to the class it extends, not to any classes that that class may extend.
- `super` will access only the most immediate method implementation.
- Note that derived classes may override methods of any of the base classes to redefine the implementation.
- Derived classes may also add methods not defined in any base class.

`CSGradStudent.java`

```java
/** Encapsulate the characteristics of a CS Grad student
* @author: Jonathan Doughty
*/

public class CSGradStudent extends CSStudent {

    private boolean isTeachingAssistant = false;
    private Amount fellowshipValue = new Amount(10000.0);

    // constructor
    public CSGradStudent( String studentName, String id) {
        super(studentName, id);
    }

    // override base class’s (actually base class’s base class) implementation. You can over-ride any method defined by a parent class up to and including java.lang.Object methods.
    public void payTuition(double tuitionAmount) {
        super.payTuition(tuitionAmount);
        fellowshipValue =
            new Amount( fellowshipValue.getAsDouble() - tuitionAmount);
    }

} // end CSGradStudent
```
void writeThesis() {
    // mumble
}

void appointTA() {
    isTeachingAssistant = true;
}

public String toString() {
    return "Grad:" + super.toString();
}

    // Other methods associated with being a grad student ...
}
Abstract: Forcing a class to be extended

- You can force users of a class you create to sub-class your class to be able to use it by qualifying your parent class (and/or possibly some or all of the methods as abstract.
- Abstract classes may provide implementations of some or all methods
- You can’t instantiate objects of an abstract class

    // compiler error:
    SomeAbstractClass abstractClassObject = new SomeAbstractClass()

- Why would an abstract class be desirable?
  - 
  - 
  - 
  - When it doesn’t make sense to have objects of the base class
  - When default implementations of some methods or fields are desirable.

The Student.java class should probably be abstract
Using Abstract Class References

- You can assign objects of a derived class to a reference of an abstract class type

```csharp
// Okay:
SomeAbstractClass abstractClassObject =
    new SomeNonAbstractDerivedClass()
```

- Why would an abstract class reference be desirable?
  - When you want to treat all objects of derived types in the same way
  - When you want to handle objects polymorphically
Accessability - Protected

The protected accessability modifier

- methods are only callable from the methods of the class and any derived classes.
- variables are accessible within the instance methods of the class and any derived classes.
- Note that you can derive from a class that is in a different package
- So protected can make implementation details visible to out of package classes.
Inheritance Example - Unique Plane Ids

**UniquePlaneId.java**

/** A class that maintains a relationship between an 
** Airplane and and Airline to identify a particular plane 
** uniquely. 
**/

import java.util.HashMap;

public class UniquePlaneId extends PlaneId {

    private static HashMap usedIds = new HashMap();

    // Save the unique id for each UniquePlaneId instance. 
    // The "final" says uniqueIdentifier can only be assigned 
    // to once.
    private final String uniqueIdentifier;

    public UniquePlaneId (Airline airline, Airplane plane) 
        throws InvalidPlaneIdException {

        // Invoke parent class’s constructor 
        super(airline, plane);

        // The "super." below is only to make explicit where 
        // getPlaneIdentifier() comes from. "super." is not 
        // needed. (unless this class had an over-riding 
        // getPlaneIdentifier() method.)

        uniqueIdentifier = super.getPlaneIdentifier();

        if (usedIds.get(uniqueIdentifier) != null) {
            throw new InvalidPlaneIdException(
                uniqueIdentifier + " already in use");
        }
    }
}
else {
    usedIds.put(uniqueIdentifier, uniqueIdentifier);
}
}

/** When/if the garbage collector reaps this
** UniquePlaneId finalize will insure the saved
** uniqueIdentifier gets forgotten as well.
**/
protected void finalize() {
    usedIds.remove(uniqueIdentifier);
}
Inheritance In Use

AirTrafficUniqueTest.java

```java
import java.util.Iterator;
import java.util.Set;                   // interface

public class AirTrafficUniqueTest extends AirTrafficTest {

    /** Iterate through the Map of airlines, ask each for an
     ** iterator of its planes, and create UniquePlaneId
     ** objects for each assigning those to
     ** AirTrafficControllers.
     **/
    void assignAirplanes() {

        Set s = airlines.keySet();
        Iterator airlineIter = s.iterator();
        int controllerIndex = 0;

        Airline airline = null;
        Airplane plane = null;

        while (airlineIter.hasNext()) {

            String name = (String) airlineIter.next(); // next airline name
            // Get the next airline object by name
            airline = (Airline) airlines.get(name);
            Iterator airplaneIter = airline.airplaneIterator();

            while (airplaneIter.hasNext()) {

                plane = (Airplane) airplaneIter.next();

                // Create the UniquePlaneId
                try {

                    // Note that a UniquePlaneId can be assigned to a
                    // field of type PlaneId: a UniquePlaneId IS-A
                    // PlaneId.

            ```
```java
PlaneId pid = new UniquePlaneId(airline, plane);

// Assign it to an AirtrafficController
AirTrafficController c =
    (AirTrafficController) controllers.get(controllerIndex);
    c.monitor(pid);
}
catch (InvalidPlaneIdException e) {
    // Shouldn’t happen
    System.err.println(e);
}

// Get ready to assign next
controllerIndex++;
controllerIndex = controllerIndex % controllers.size();
}

// Try to create one last UniquePlaneId with the same
// airline and plane as the last

try {
    PlaneId id = new UniquePlaneId(airline, plane);
} catch (InvalidPlaneIdException e) {
    // Should happen
    System.err.println(
        "Success: caught attempt to make a non-unique " + e");
}

public static void main(String[] args) {

    String usage =
        "usage: AirTrafficUniqueTest number_of_controllers " +
        "[airline_name number_of_planes] ...";

    if ( args.length < 3 || args.length % 2 == 0 ) {
        System.out.println(usage);
    } else {
        AirTrafficUniqueTest tester = new AirTrafficUniqueTest();
```
tester.performTests(args);

}
Final: preventing a class or method from being extended

final classes

The keyword `final` has two uses: it can be used to mark a class, much as the `abstract` keyword is used:

```java
public final class SomeClass {
    ...
}
```

- A class marked `final` can not be extended
- Why would a final class be desirable?

Final methods

The keyword `final` can also be applied to individual methods:

```java
public class SomeClass {
    ...

    public final double getSalary() {
        ...
    }
}
```

- A method marked final can not be overridden in a derived class.
- Marking a method `final` does not prevent creating derived classes
- Why would a final method be desirable?
When you don’t want to allow sub classes to over-ride certain, required behavior
Homework 5

Reading

- Chapter 6 - Collections of Objects
- Chapter 7 - Some Final Concepts

Programming

No specific programming assignment this week

If you want to get a jump on next week’s assignment, study and understand inheritance and abstract classes using tonight’s examples.