Highlights of Last Week

- Using programmer defined Object types rather than primitives
- Access modifiers
- Introduction to Collection classes: ArrayList, HashMap
- Introduction to Inheritance: every class extends java.lang.object
Submitting Homework

- Jar archives only
- Name jars your_gmu_id.jar
- Jar should only contain a single top level directory/folder that is also named your_gmu_id:

```
jar tf 123-45-6789.jar
META-INF/
META-INF/MANIFEST.MF
123-45-6789/
123-45-6789/Airplane.java
123-45-6789/Airline.java
123-45-6789/TestAirlines.java
```

- Not

```
jar tf emailname.jar
META-INF/
META-INF/MANIFEST.MF
emailname/
emailname/Airplane.java
emailname/Airline.java
emailname/TestAirlines.java
```

- Not

```
jar tf nameless.jar
META-INF/MANIFEST.MF
HW2/Airline.class
HW2/TestAirlines.class
HW2/Airplane.class
```

- Not
jar tf nameless.jar
META-INF/
META-INF/MANIFEST.MF
C:/nameless/
C:/nameless/Airplane.java
C:/nameless/Airline.java
C:/nameless/Airtest.java
HashMap clarification

```java
HashMap h = new HashMap();
String k = new String("key");
String v = new String("value");

k.hashCode()

h.put(k, v)

v = (String) h.get(k);
```

- Any of the Map implementations store only a **single** reference value
- There is no allowance for collisions of hashcodes. That is, each value to be stored in the Map must have a distinct key.
- The programmer is responsible for handling hashcode collisions, e.g., store as a value a List of colliding values.
Text Issues

The text says (p. 104):

"declaring attributes as public should only be done for the simplest of attributes, i.e., those which do not require careful monitoring and maintenance."

This breaks encapsulation

Guideline: do not access attributes (fields, variables) in other objects directly

Use accessor methods: `getXxx()` , `setXxx()`.
Problems in Last Week’s Examples

In `CSCourse.java` we find this code:

```java
/** Enroll the student whose name is provided into this course. **/
public void enroll(String name, String id) {
    CSStudent current = new CSStudent(name, id);
    // Remember this student by id number
    students.put(id, current);
}
```

What is wrong with this?

- This doesn’t model the way the world works.
- Courses don’t create students; students exist independently of courses.
- Fixing this is going to have a ripple effect
Step 1 - Decouple student construction from enrollment

Create students externally (in CSTest for example):

```java
// Create some students, enroll them in the course
for (int i = 0; i < names.length; i++) {
    CSStudent student = new CSStudent(names[i], ids[i]);
    cs332.enroll(student);
}
```

CSCourse’s enroll method now has a single responsibility:

```java
/** Enroll a student into this course. */
public void enroll(CSStudent student) {
    // Remember this student as having enrolled
    students.put(student, student);
}
```
Another problem

- CSCourses remember what students are enrolled in them
  
  // Remember this student as having enrolled
  students.put(student, student);

- But the course can’t identify those students by ID.

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

Two choices to address this problem:

- Change the signature of `setHomeworkGrade` to
  
  `setHomeworkGrade(CSStudent s, int assignment, String grade)`

  Require caller to look up student by id

- CSCourse would still have no way to access specific students
- Or change CSCourse so it can again look up enrolled students by id
Further re-factoring

Choose the latter, so `CSCourse.java` now has:

```java
public void enroll(CSStudent student) {
    // Remember this student as having enrolled
    // Note addition of a getId method to CSStudent
    students.put( student.getId(), student);
}
```

`CSStudent.java` needs to expose `id` (to at least other objects in the same package)

```java
Object getId() {  
    return id;
}
```

Note the type it returns

And the signature of `setHomeWorkGrade` changes slightly:

```java
public void setHomeworkGrade( Object id, int assignment, String grade) {  
    // Find the student object whose id is id
    CSStudent s = (CSStudent) students.get(id);
```
Iterators

Example in CSCourse.java

```java
/** Provide access to the students enrolled in
 ** the course
 */
public Iterator getStudentIterator() {
    java.util.Collection collection = students.values();
    return collection.iterator();
}
```

Example usage CSTest.java

```java
// Let each student in the course identify herself
Iterator iter = cs332.getStudentIterator();
while (iter.hasNext()) {
    student = (CSStudent) iter.next();
    System.out.println(student);
}
```
Exceptions

Definition (from the Java Tutorial):

"An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions."

"Java requires that a method either catch or specify all checked exceptions that can be thrown within the scope of the method."

- You catch an Exception by bracketing the code that could potentially throw an Exception in a try / catch block:

  ```java
  try {
    // Code that could potentially throw an exception
  }
  catch (Exception type) {
    ...
  }
  ```

- You specify the exception(s) that your method can potentially throw:

  ```java
  ... methodName(args) throws ExceptionType1 [, ExceptionType2] { ...
  ```
Purpose of Using Exceptions

- Separate error handling from expected normal conditions.
- Propagate errors that your code can’t/shouldn’t deal with up the call stack.
- Force users of your class’ objects to live up to their part of the coding contract (in the case of checked exceptions.)
Separate Error Handling

An example from the Java tutorial. First, pseudo-code that outlines what the code will do:

```java
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```
Procedural approach to error handling

Next, what code looks like as written by procedural, C developers:

```c
errorCodeType readFile() {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                } else {
                    errorCode = -2;
                }
            } else {
                errorCode = -3;
            }
        } else {
            errorCode = -4;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```
Using exceptions to handle errors

Finally, code that is written using Exception handling:

```java
void readFile() {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```
Propagate Errors / Forcing Users to Handle Problems

Consider two possibilities. Which is better?

```java
public boolean adjustBalance(double amount) {
    if (amount > currentBalance)
        return false;
    else {
        currentBalance -= amount;
        return true;
    }
}

public void adjustBalance(double amount) throws InsufficientFundsException {
    if (amount > currentBalance)
        throw new InsufficientFundsException();
    else {
        currentBalance -= amount;
    }
}
```
Hierarchical Exceptions and Finally

The complete form of try/catch in Java is

```java
try {
    // block of code that could potentially throw a number of
    // different exceptions
} catch (ExceptionType) {
    ...
} catch (AnotherExceptionType) {
    ...
} catch (YetAnotherExceptionType) {
    ...
} finally {
    // code that will execute whether any exceptions are
    // thrown (or not.)
}
```

Note:

- The catch blocks need to be in increasing hierarchical order if the exceptions that can be thrown in the try block are related by inheritance.
- The [optional] finally block gives your code a chance to do clean up, release resources, etc.
- Guideline: It is **not** good practice to just catch the "mother of all Exceptions" java.lang.Exception.
- Guideline: It is **not** (generally) good practice to enclose each method call that could throw an exception in an individual try/catch block.
Exception Classes

Two broad categories of exceptions:

- Checked Exceptions - derived from java.lang.Exception
- Runtime Exceptions - derived from java.lang.RuntimeException

Checked exceptions must be specified or handled by a method: the compiler will check that you deal with checked exceptions.

Runtime exceptions aren’t checked for by the compiler

Runtime exceptions are occurrences such as

- Divide by zero - ArithmeticException
- Illegal access - NullPointerException, IndexOutOfBoundsException, SecurityException
- Various unpredictable events
Declaring/using Exceptions

UnknownStudentException.java

package week03;

/** 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);
    }
}

Used: as in 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 CSTest.java
// Exercise the exception handling
String fakeId = "000-00-0000";
try {
    cs332.setHomeworkGrade(fakeId, 1, "A");
} catch (UnknownStudentException e) {
    System.err.println(e);
}
Exception Guidelines

- "Do not throw a runtime exception or create a subclass of RuntimeException simply because you don’t want to be bothered with specifying the exceptions your methods can throw."
- Similarly, do not throw a runtime exception or create a subclass of RuntimeException simply because you don’t want to be bothered with writing try/catch blocks around your method calls.
- Do not use the Exception types built-in to Java unless the condition you are signally really matches the description.

More information (not required reading) about Exceptions can be found in:

- The Java Tutorial: Handling Errors with Exceptions
- The article Taking Exception which appeared in Unix Review in June 1997.
Homework 4

Reading

- Chapter 5 - Relationships Between Objects
- Text book pages on Java Exception Handling pp. 320 - 322

Programming

This assignment has three goals: 1) is to get some experience creating helper objects, 2) To explore using Collection classes, 3) To further develop classes developed in last week’s assignment. You may use your version or mine

- Create a helper class called PlaneId. The purpose of PlaneId objects is to be little more than a data structure that will bind two data items together. A PlaneId should be composed of two thing:
  1. something that will uniquely identify an Airplane, either an Airplane object reference or the same String as is used to identify Airplanes when they are created; and
  2. something that will uniquely identify an Airline, either an Airline object reference or its name (the same string that is used to identify an Airline when an Airline object is created.
  - A given Airplane / Airline pair should always correspond to only one PlaneId object
  - Hint: To uniquely identify a PlaneId object you will probably need to over-ride the default hashcode and equals() methods.
  - Hint: You may define any private, implementation details you might like to help you uniquely identify a PlaneId object that has the same Airplane and Airline values.
• Create a class named **AirTrafficController**. An AirTrafficController object should have the following characteristics:
  ○ a collection of **PlaneId** objects that the AirTrafficController is responsible for monitoring.
  ○ a method to add a new **PlaneId** object to the collection
  ○ a method to remove a specified **PlaneId** from those in the collection, returning a reference to the **PlaneId** removed.
  ○ a method to return an Iterator for a AirTrafficController’s PlaneIds

• Any other object types that you would find useful. Note that this includes Exception types.

• Modify the test class (or create a new one) so that will do the following:
  ○ Have the test class create at least two **AirTrafficController** objects
  ○ As the test class creates new Airplane objects and adds them to an Airline, have the test class also create PlaneId objects that will identify that Airplane / Airline combination.
  ○ Assign each PlaneId to one AirTrafficController. Be sure that each AirTrafficController gets a roughly equal number of PlaneIds. That is, at least assign PlaneIds to a different AirTrafficController than the previous PlaneId was assigned to.
  ○ Have each AirTrafficController list the PlaneIds of the Airplanes she is monitoring.
  ○ Bonus: exercise AirTrafficController’s ability to remove a PlaneId that is being monitored from her collection and add that PlaneId to some other AirTrafficController. List the result before and after for at least the two AirTrafficControllers involved.