Highlights of Last Week

- Using Collection classes
- Marking methods `protected` to permit access only from derived classes
- Marking classes `abstract` to force inheritance
- Marking classes `final` to prevent inheritance
- Marking methods `final` to prevent derived classes from over-riding required behavior
- Using `static` fields to maintain information that is common to all objects of a class
Inheritance vs. Composition

- Use inheritance to model IS-A-KIND-OF relationships.
- Use composition to model HAS-A relationships.
- When in doubt, try forming a sentence:
  - A Car IS-A-KIND-OF Vehicle - sounds right

    A Triangle IS-A-KIND-OF Shape - sounds right

    Inheritance relationship is natural

  - A Car IS-A-KIND-OF Engine - doesn’t sound correct

    A Triangle IS-A_KIND-OF Vertex - bzzt

    Inheritance relationship isn’t natural

  - A Car HAS-A Engine - sounds right

    A Triangle HAS-A Vertex - yup

    Composition is the right approach
Guidelines - Inheritance

- Derived classes inherit the **interface** and, by default, the **implementation** of their base class(es).
- 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)
- A derived class may also offer new services not offered by the base class
- A derived class should **not** try to remove services offered by the base class
- Do **not** use inheritance unless the IS-A-KIND-OF relationship is true: don’t use inheritance simply to get services for free. In those cases use composition instead.
Inheritance and method invocation - C++

Consider the following C++ code; what will the result be?

Upcast.cc

// Example of C++’s rules for method calls when
// extending a base class.

#include <iostream.h>

/** This C++ class is going to be the parent class.
 ** It defines a toString member function.
 ***/
class Parent {
public:
    Parent() {};
    ~Parent() {};

    char * toString() {
        return "I’m Parent’s toString()";
    }
};

/** C++ class that inherits from Parent, also defining
 ** a toString() member function.
 ***/
class Derived : public Parent {
public:
    Derived() {};
    ~Derived() {};

    char *toString() {
        return "I’m Derived’s toString()";
    }
};
/** Test driver */
main() {
    Derived d;
    cout << d.toString() << endl;

    // Get at parent class’s behaviour
    Parent p = (Parent) d;
    cout << p.toString() << endl;

    // And, of course, there is no Object that everything
    // derives from in C++ so there are no other
    // possibilities.
}

I’m Derived’s toString()
I’m Parent’s toString()
Inheritance and method invocation - Java

Upcast.java

// Example of Java’s rules for method calls when
// extending a base class.

/** This class simply over-rides java.lang.Object’s
 ** toString() method. **/
class Parent {
    public String toString() {
        return "I’m Parent’s toString()";
    }
}

/** Class that inherits from Parent, which also
 ** over-rides toString()
 **/
class Derived extends Parent {

    public String toString() {
        return "I’m Derived’s toString()";
    }
}

/** Test driver */
public class Upcast {
    public static void main(String[] args) {
        Derived d = new Derived();
        System.out.println("Derived’s toString() : " +
                d.toString());

        // This works because Derived IS-A Parent
        Parent p = (Parent) d;
        System.out.println("Parent’s toString() : " +
                p.toString());

        // And this works because everything IS-A Object
Object o = (Object) d;
System.out.println("Object’s toString() : " + 
o.toString());
System.out.println("explict cast to Object’s toString() : " +
((Object) o).toString());

•
•
•
•

Derived’s toString() : I’m Derived’s toString()
Parent’s toString() : I’m Derived’s toString()
Object’s toString() : I’m Derived’s toString()
explict cast to Object’s toString() : I’m Derived’s toString()

• Java is different than C++
• If you invoke a method that is defined for both a base class and a derived class on a reference whose actual type is derived, you always get the derived version of the method. Other, external class objects can NOT access base class behavior.
• The derived class itself can access base class behavior if it chooses to. How?
Interfaces

- "the place at which independent and often unrelated systems meet and act on or communicate with each other" - Merriam-Webster
- "A set of method signatures ... which collectively define what it means to assume a certain role ..."
- Provide flexibility
  - During design
  - During extension of an existing system
- Interfaces define the behavior of a class (or classes)
Java Interfaces

- In Java, an interface is a way of specifying a set of methods that a class must implement.
- Interfaces may contain only
  - method signatures
  - static final variables (Java’s version of constants.)

An interface can not contain any instance variables or method definitions

- Think of an interface as if it were a small contract.
  - Classes agree to abide by the contract by saying they implement the interface.
  - The compiler will make sure that classes that say they implement an interface abide by the contract.
Java interface definition

A pair of Java interfaces

Billable.java

public interface Billable {
    public void payBill(Amount bill, Payable whoToPay );
}

Payable.java

public interface Payable {
    public void pay(Amount bill, Billable whoPayed );
}
Interfaces as Types

- You can define variables of an interface type.

In InterfaceType.java:

```java
public interface InterfaceType {
...
}
```

In some other class:

```java
...
InterfaceType name = methodReturningInterfaceType();
```

or

```java
...
InterfaceType name = methodReturningTypeImplementingInterface();
```

- You can **not** instantiate an interface type

```java
InterfaceType fieldname = new InterfaceType() // compiler error
```

- Interface types can be used as method arguments

```java
public void doSomething(InterfaceType it) {
...
  it.someMethodSpecifiedInInterface();
}
```

- **Only methods defined in the interface can be accessed via a field of InterfaceType.**
Interface Guidelines

- Use Java interfaces to say "What objects can do," or occasionally, "What can be done to an object." - Bill Venners.
- Java interfaces don’t say "What objects are."
- Java interfaces say what objects can do no matter what the implementation objects are
Design with Interfaces

- When we define a type with a class, we define and **bind** that type to some particular implementation.
- When we define a type in terms of an interface, we are defining a type with no requirement on the implementation other than the types of the input (arguments) and output (return type).
- When **designing** a system we should be concerned if we find ourselves writing more classes than interfaces.

Jim Waldo - *Interfaces and Classes*
Interface Benefits

- An interface type guarantees certain behavior without needing to know or care about the implementation.
- One object may have many different types of behavior - polymorphism.
A Java interface implementation

FullTimeStudent.java

```java
public class FullTimeStudent implements Billable {

    Amount bankBalance;

    public void payBill(Amount amountOwed, Payable whoToPay) {
        bankBalance = bankBalance.adjust(amountOwed);
        whoToPay.pay(amountOwed, this);

        // mumble
    }

    // Other methods relating to being a FullTimeStudent that
    // don’t relate to being billed, and how to keep
    // bankBalance omitted
}
```
A Java interface in-use
Registrar.java

import java.util.Collection;
import java.util.Iterator;

public class Registrar implements Payable {
    java.util.Collection billPayers;
    // ...

    public void sendBills(Amount tuition) {
        Iterator i = billPayers.iterator();
        while (i.hasNext()) {
            Billable b = (Billable) i.next();
            b.payBill(tuition, this);
        }
    }

    public void pay(Amount bill, Billable billPayer) {
        // look up billPayer in collection of billPayers
        // credit their account by the bill Amount
    }

    // ... remaining Registrar operations omitted
}
Notice that both classes are completely de-coupled from one another: they interact only through common interfaces.
Abstract Classes vs. Interfaces

Interfaces are similar to abstract classes except ...

- abstract classes can have default implementations of methods
- A Java class can only directly extend one class (abstract or otherwise)
- A Java class can implement many interfaces

Other similarities include:

- An object of a derived class can be assigned to a reference of a super class, abstract or not

  \[
  \text{AbstractClass abstractClassObject = new DerivedClass();}
  \]

  Similarly, any time a method takes a particular type as an argument, you can pass a more specialized type.

  ```java
  void method( Type t ) {
    ...
  }
  
  DerivedType dt = new DerivedType();
  obj.method(dt);
  
  This is good practice, just as using interface references was: keep method arguments as general as possible so that methods are not coupled to overly specialized classes.

- If you invoke a method that is defined for both a base class and a derived class on a reference whose actual type is derived, you always get the derived version of the method. This is different from C++: You cannot upcast to get at base class behavior

- As with interfaces, given a reference of a base class type, you can only invoke methods that are defined for the base class type, additional methods that the
actual type may support are inaccessible.
Polymorphism

- "the quality or state of being able to assume different forms" - Webster's
- "the ability to manipulate objects of distinct classes using only knowledge of their common properties without regard for their exact class."

An object can expose different sets of capabilities depending on its dynamic type.

In Java, polymorphism is accomplished when one class **extends** another class and **implements** one or more different **interfaces**.

```java
public class Something extends SomethingElse 
    implements OneInterface, AnotherInterface {
    ...
}
...

Something s = new Something();
s.method();
...

SomethingElse se = s;
se.method2();
...

OneInterface oi = s;
oi.imethod();
...

AnotherInterface ai = s;
ai.method();
...
```
Homework 6

Mid-term Exam

- Will be on the evening of 22 October (The final will be on 17 December)
- Will include material covered in lecture and assigned readings
- I’ll review questions on anything we’ve covered so far in next Monday, 15 Oct’s lecture.

Reading

- The article Maximize flexibility with interfaces and abstract classes
  for the theory
- The article Abstract classes and interfaces practicum
  for the practice.

Programming

The goal of this homework assignment is to explore using inheritance. This assignment will use new versions of the Airplane, Airline, and test classes developed in the second homework assignment. Once again, you can start from my sample solution versions or use your own.

- Create three classes/types of air planes: BigPlane, MediumPlane, and SmallPlane. All three classes will have the following characteristics:
  - An identifying string ("1", "2", ... etc. is sufficient),
  - A total number of seats. Once assigned this does not change for an Airplane though different Airplanes of the same type may have different total number of seats.
  - A number of currently available seats.
  - A cost per seat in each of several categories, see below.
- A method to change the cost per seat for a seat category.
- An implementation of the `public String toString()` method that will return a String containing the type of airplane (Big, Medium, Small), the airplane ID and the current costs per seat.

However, these three classes should differ from each other in the following ways:

- The total number of seats should be between 200-300 for BigPlanes, 50-200 for MediumPlanes, and 8-50 for SmallPlanes.
- BigPlanes and MediumPlanes should have three costs per seat, one each for FirstClass, BusinessClass, and EconomyClass. SmallPlanes should have only BusinessClass and EconomyClass cost per seat values.

- Create an abstract base class, AirPlane. Use this class to model the characteristics that are common to all three airplane classes above.

**You decide what belongs in the base class and what belongs in the specific classes.** You should try to push as much common information into the base class. Unnecessary code duplication in derived classes is to be avoided. The base Airplane class should **not** have any knowledge of specific derived classes.

- Modify the Airline class from the previous assignments, if necessary, to use AirPlane references. You may use my sample solution version or your own. The Airline objects should save the Airplane references in a collection class. The Airline class should not have any knowledge of specific classes derived from Airplane.
- Modify the test class from the second assignment or create a new test class to create several of each kind of airplane. Add these airplanes one at a time through a call to Airline’s `addAirplane` method to one or more Airline objects. (An airplane should only be added to one airline’s collection.)
- Have each Airline object list all the Airplanes in its collection: their class, id, and current cost per seat values.

This assignment is due next Monday, 15 Oct. Pay strict attention to the requirements above.