CMSC 131: Chapter 28
Interfaces II

Review of Overloading and Overriding

Before discussing interfaces, let's review some elements of method overloading and
overriding.

When overloading a method the subclass method prototype must match exactly the
prototype of the superclass (same name, same return type, same arguments).

You may change access specifier (public, private, protected), but derived classes cannot
decrease the visibility.

Example: You be the Compiler

```java
public class Base {
  protected void someMethod( int x ) { ... }
}

public class Derived extends Base {
  public void someMethod( int x ) { ... }
  public int someMethod( int x ) { ... }
  public void someMethod( double d ) { ... }
}

(the following appears in the same package)

Base b = new Base();
Base d = new Derived();
Derived e = new Derived();
b.someMethod( 5 );
d.someMethod( 6 );
d.someMethod( 7.0 );
e.someMethod( 8.0 );
```
Object is not Abstract

We discussed abstract classes and methods. Note that Object is not an abstract class. You can create instances of Object, you just can't do very much with them.

Object o = new Object();
Object p = new Object();
System.out.println(o.toString() + " "+ p.toString());

Interfaces: Recap

We introduced the concept of interfaces earlier this semester (in the context of the cmsc131PictureLib). Recall:

Interface:
- Is defined by the keyword interface (rather than class).
- It is abstract. That is, it defines methods (as many as you like), but does not give method bodies (the executable statements that make up the method).

  public interface Y {
    public void someMethod( int z );
    public int anotherMethod();
  }

- These methods are usually public, since they are expected to be part of an object's public interface.
- An interface is not a class. Because an interface is abstract, you cannot create an instance of interface Y using "new Y".

Interfaces: Recap

Implementing an Interface:
- An interface is a convenient way for a class to say that it "promises" to implement certain methods.
- A class is said to implement an interface if it provides definitions for these methods.
- To inform Java that a class implements a particular interface Y, we add "implements Y" after the class name:

  public class X implements Y {
    // ...(instance data and other methods)...
    public void someMethod( int z ) { ... code goes here ... }
    public int anotherMethod() { ... code goes here ... }
  }

- An interface is a type: We may use a reference to an X any place that a reference to an object of type Y is expected.
Interfaces and Inheritance

Abstract methods and classes: Recall that in any class we can leave a method undefined (abstract):

```java
public abstract class Shape {
    // ... (details omitted)
    public abstract void drawMe();
}

public class Circle extends Shape {
    // ... (details omitted)
    public void drawMe() { ... Circle drawing code goes here ... }
}
```

Some questions:

- What is the difference between abstract methods and interfaces?
- Why do we need interfaces? Can't we do everything using abstract methods and classes?
- Is it possible to have interface hierarchies, in the same way we can have class hierarchies?

Multiple Inheritance

Motivation: There are many situations where a simple class hierarchy is not adequate to describe a class’s structure.

Example: Suppose that we have our class hierarchy of university people, and we also develop a class hierarchy of athletic people:

![Class Hierarchy Diagram]

StudentAthlete: Suppose we want to create an object that inherits all the elements of a Student (admission year, GPA) as well as all the elements of an Athlete (sport, amateur-status).
Multiple Inheritance

Can we define a StudentAthlete by inheriting all the elements from both Student and Athlete?

```java
public class StudentAthlete extends Student, extends Athlete
{
    ...
}
```

Alas, no. At least not in Java.

Multiple Inheritance:

- Building a class by extending multiple base classes is called multiple inheritance.
- It is a very powerful programming construct, but it has many subtleties and pitfalls. (E.g., Athlete and Student both have a name instance variable and a toString() method. Which one do we inherit?)
- Java does not support this. (Although C++ does.)
  - In Java a class can be extended from only one base class.
  - However, a class can implement any number of interfaces.

“Faking” Multiple Inheritance with Interfaces

Since Java lacks multiple inheritance, is there an alternative? What public methods do we require of an Athlete object?

- String getSport(); Return the athlete's sport
- boolean isAmateur(); Does this athlete have amateur status?

We can define an interface Athlete that contains these methods:

```java
public interface Athlete {
    String getSport();
    boolean isAmateur();
}
```

Now, we can define a StudentAthlete that extends Student and implements Athlete.
"Faking" Multiple Inheritance with Interfaces

StudentAthlete extends Student and implements Athlete:

    public class StudentAthlete extends Student implements Athlete {
        String mySport;
        boolean amateur;
        // ... other things omitted
        String getSport() { return mySport; }
        boolean isAmateur() { return amateur; }
    }

StudentAthlete can be used:

- anywhere that a Student object is expected (because it is derived from Student)
- anywhere that an Athlete object is expected (because it implements the public interface of Athlete).

So, we have effectively achieved some of the goals of multiple inheritance, by using Java's single inheritance mechanism.

Common Uses of Interfaces

Interfaces are flexible things and can be used for many purposes in Java:

- A work-around for Java's lack of multiple inheritance.
  (We have just seen this.)

- Specifying minimal functional requirements for classes.
  (This is its principal purpose.)

- For defining groups of related symbolic constants.
  (This is a somewhat unexpected use, but is not uncommon.)
Using Interfaces for Symbolic Constants

In addition to containing method declarations, interfaces can contain constants, that is, variables that are public final static. Sometimes interfaces are used for just for this purpose:

```java
public interface Months {
    public final static int JANUARY = 1;
    public final static int FEBRUARY = 2;
    public final static int MARCH = 3;
    /* ... blah blah blah ... */
    public final static int DECEMBER = 12;
}

class MonthDemo implements Months {
    public static void main(String[] args) {
        System.out.println("March is month number " + MARCH);
    }
}
```

Using Interfaces to Specify Requirements

Specifying minimal functional requirements for classes:

- **Picture** (in our cmsc131PictureLib): defines methods `getHeight()`, `getWidth()`, and `getColor(int i, int j)`

Standard interfaces from the Java class library:

- **Comparable** (java.lang): defines a method `compareTo()`. It is used in situations where a class defines objects that can be ordered.

- **Iterator** (java.util): defines methods `hasNext()`, `next()`, `remove()`. It is used for enumerating all the elements of an object consisting of a collection of items, such as `ArrayList`. 
Interface Hierarchies

Inheritance applies to interfaces, just as it does to classes. When an interface is **extended**, it inherits all the previous methods.

**Example:** As we saw before, an **Iterator** is an object that allows you to step through a collection of items. Here is its definition:

```java
public interface Iterator {
    boolean hasNext(); // any more items?
    Object next();     // return the next item
    void remove();     // remove the current item
}
```

Suppose that we want a **bi-directional iterator**, which can move both forwards and backwards. We could implement it as follows:

```java
public interface BidirectionalIterator extends Iterator {
    boolean hasPrevious(); // any prior items?
    Object previous();     // return the previous item
}
```

Polymorphism Through Interfaces

Interfaces allow us to write methods that are **polymorphic**, in the sense that they can be applied to a wide variety of objects.

**Example:** Suppose that we want to write a function, which is given an array of references to some class type, and sorts the elements in **increasing order**.

**Polymorphic Design:** Any sorting function must be able to compare two objects. Suppose we use `compareTo()` to do this. This allows us to apply the sorting algorithm to any **array** containing objects that implement the `Comparable` interface. (More on this later.)