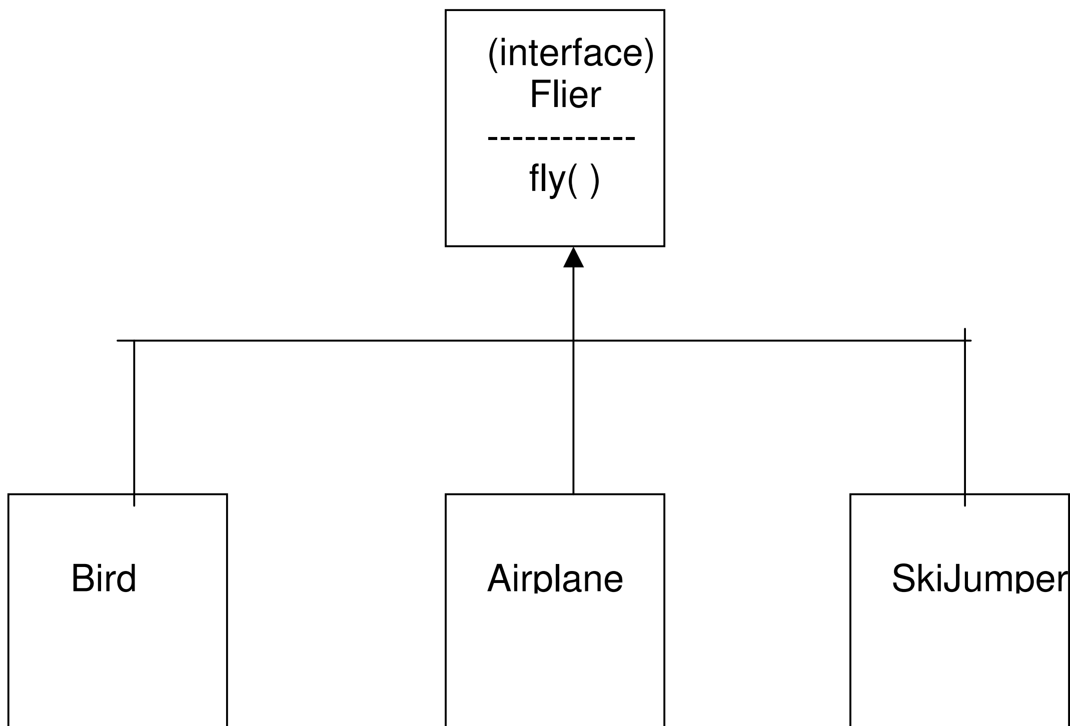
AP Programming - Chapter 9 Lecture

Interfaces and Polymorphism

I. ***Reusable Solutions*** (9.1)

A Java interface declares methods that classes are expected to implement. It encapsulates a set of associated behaviors that can be used by dissimilar classes. Interfaces contain no instance fields and cannot be instantiated. A class realizes (implements) an interface by promising to provide the implementation for all of the methods in the interface. An interface can be implemented by many classes and a class can implement multiple interfaces.

ex.1) Consider the following trivial but illustrative example. A Flier is one that flies.



It is clear that the three classes Bird, Airplane, and SkiJumper are three dissimilar classes. Each of these classes defines the method fly( ) in a very different way. The interface for this trivial descriptive example would be

```
public interface Flier
{
    void fly( );
}
```

Notice that the public visibility modifier is not included. The methods declared in an interface must be **public** and **abstract** and are the defaults. *An abstract method is one where there is no body for the method. The above method could have been written as: **public abstract void** fly( );
but real programmers would laugh at you since real programmers know that all methods in an interface are automatically **public** and **abstract**.

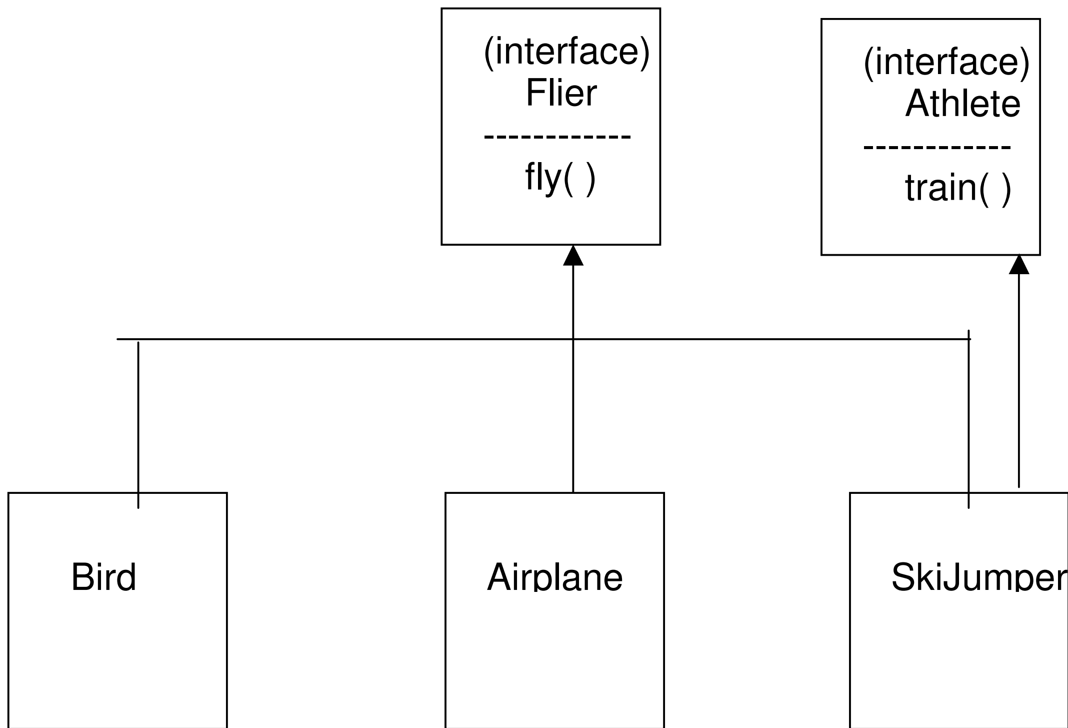
*Important Note: The classes that realize (use) an interface
1) implements it by the keyword **implements** (see example next page)
2) must supply code inside of braces for any methods in the interface.

The classes that realize the above interface might include the following code implementation for the fly method. Note the modifier **public** must be included with the methods that actually realize the fly( ) method.

```
public interface Flier
{
    void fly( );    // public by default
}

public class Bird implements Flier
{
    public void fly( )   // public must be written here
    {
        System.out.println("Using my wings to fly");
    }
}

public class Airplane implements Flier
{
    public void fly( )
    {
        System.out.println("Using my jet engines to fly ");
    }
}

public class SkiJumper implements Flier
{
    public void fly( )
    {
        System.out.println("Using skis to take me into the air ");
    }
}
```

AP Programming - Chapter 9 Lecture

An athlete is one that competes. A SkiJumper is an Athlete. An athlete trains for hours to perfect performance in a particular sport. Most of us would agree that neither a Bird nor an Airplane is an Athlete. Therefore, if Athlete were an interface, a SkiJumper might implement Athlete as well as Flier.



```
public interface Athlete
{
    void train(double hours);
}

public class SkiJumper implements Flier, Athlete
{
    // Constructor
    public SkiJumper(String fName, String lName)
    {
        firstName = fName;
        lastName = lName;
        numberOfJumps = 0;
        hoursTraining = 0;
    }

    public void fly( )
    {
        System.out.println("Using skis to take " + firstName + " " + lastName +
            " into the air.");
        numberOfJumps++;
    }

    public void train(double hours)
    {
        System.out.println("I am on the slopes " + hours + " hours today.");
        hoursTraining += hours;
    }

    public double getHoursTrained( )
    {
        return hoursTraining;
    }

    public int getJumps( )
    {
        return numberOfJumps;
    }

    private String firstName; private String lastName;
    private double hoursTraining; private int numberOfJumps;
}
```

Now that you have an intuitive understanding of an interface and how it works, look at two examples that might be a bit more useful.

Use *interface types* to make code more reusable:

In Chapter 7, the book's author created a DataSet to find the average and maximum of a set of values (*numbers*)

What if we want to find the maximum of a set of BankAccount values?

```
public class DataSet    // Modified for BankAccount objects
{
  public void add(BankAccount x)
  {
    sum = sum + x.getBalance( );
    if (count == 0 || maximum.getBalance( ) < x.getBalance( ))
      maximum = x;
    count++;
  }

  public BankAccount getMaximum( )
  {
    return maximum;
  }

  private double sum;
  private BankAccount maximum;
  private int count;
}
```

What if we want to find the coin with the highest value from a set of coin values? We would need to modify the previous DataSet class.

```
public class DataSet   // Modified for Coin objects
{
  public void add(Coin x)
  {
    sum = sum + x.getValue( );
    if (count == 0 || maximum.getValue( ) < x.getValue( ))
      maximum = x;
    count++;
  }

  public Coin getMaximum( )
  {
    return maximum;
  }

  private double sum;
  private Coin maximum;
  private int count;
}
```

Note: The mechanics of analyzing the data is the same in both cases; just the details of measurement differ.

We can implement a single reusable DataSet class whose add method looks like this:

```
sum = sum + x.getMeasure( );
if (count == 0 || maximum.getMeasure( ) < x.getMeasure( ))
  maximum = x;
count++;
```

What is the type of the variable x? x should refer to any class that has a getMeasure method.

ex.2) Section 9.1 in your text defines the Measurable interface.

```
public interface Measurable
{
    double getMeasure( );
}
```

The Measurable interface is realized by Coin and BankAccount, two dissimilar classes. The only method declared in this interface is getMeasure( ). This method simply returns a double. In the text example, either a coin's value or a bank account's balance is returned. The DataSet class in Section 9.1 (see below) becomes usable in different circumstances by implementing a method add whose parameter realizes the Measurable interface. Interfaces are not instantiated i.e. interfaces can not create an actual object since they are not a class but the interface can be used to create a reference to an object. The variable x below refers to a class object that realizes Measurable.

```
public void add(Measurable x)    // see code on the next page
```

This method will add Coin values or BankAccount balances (or any other object instantiated by a class that realizes Measurable) and will keep track of the largest Coin value or BankAccount balance in the private instance field maximum.

Use implements keyword to indicate that a class implements an interface type

```
public class BankAccount implements Measurable
{
  public double getMeasure( )
  {
    return balance;
  }
  // Additional methods and fields
}


public class Coin implements Measurable
{
  public double getMeasure( )
  {
    return value;
  }
}
```

Generic DataSet for Measurable Objects:

```
1     public class DataSet
2     {
3        // More code here
4        public void add(Measurable x)
5        {
6           sum = sum + x.getMeasure( );
7           if (count == 0 || maximum.getMeasure( ) < x.getMeasure( ))
8              maximum = x;
9           count++;
10       }
11       public Measurable getMaximum( )
12       {
13          return maximum;
14       }
15       // Other variables here
16       private Measurable maximum;
17       private int count;
18    }
```

Line 4: The parameter x is of any type that realizes the Measurable interface.
If the declaration were: DataSet bankData = **new** DataSet( );
the call to this method might be: bankData.add(**new** BankAccount(100));
and we have: Measurable x = **new** BankAccount(100)

If the declaration were: DataSet coinData = **new** DataSet( );
the call to this method might be: coinData.add(**new** Coin(0.25, "quarter"));
and we have: Measurable x = **new** Coin(0.25, "quarter")

Line 6: Adds values as defined in the getMeasure method of the individual class. If x were instantiated from the BankAccount class, we would be adding balances. If x were instantiated from the Coin class, we would be adding Coin values.

Lines 7-8: Checks for a new largest object based on the comparison of double values and resets present value of maximum if necessary. Notice that maximum is a type that realizes the Measurable interface. For the BankAccount class, maximum will store a BankAccount object. For the Coin class, maximum will store a Coin object.

Line 13: getMaximum will return the object with the largest measure.

Line 16: The private instance field is of type Measurable. This field can refer to any class that realizes the Measurable interface.

UML Diagram of DataSet and Related Classes:

- Interfaces can reduce the coupling between classes
- UML notation:
  - Interfaces are tagged with a "stereotype" indicator «interface»
  - A dotted arrow with a triangular tip denotes the "is-a" relationship between a class and an interface
  - A dotted line with an open v-shaped arrow tip denotes the "uses" relationship or dependency
- Note that DataSet is *decoupled* from BankAccount and Coin

ex.3) Our third example of an interface is one that we have already used with the string class.

class Java.lang.String **implements** java.lang.Comparable

The Comparable interface has one method declared, compareTo( ). The string class implements this method so that it compares two strings lexicographically:

Method	*Method Summary*
int compareTo (Object other)	Returns value < 0 if *this* String is less than other. Returns value = 0 if *this* String is equal to other. Returns value > 0 if *this* String is greater than other.

interface java.lang.Comparable

Method	*Method Summary*
int compareTo (Object other)	Returns value < 0 if *this* is less than other. Returns value = 0 if *this* is equal to other. Returns value > 0 if *this* is greater than other.

We might want a student class to allow for the ordering of students, perhaps by the student GPA, or the student ID number. If a student class realized Comparable, we would add our preferred definition of compareTo to the Student class implementation. The Comparable interface can be implemented by the Coin class, the Car class, and any class that may want to provide an ordering for objects instantiated from the class. Coin objects may be compared based on their value. Car objects may be compared based on their mileage or their price.

A Car class which implements Comparable might implement compareTo in the following way.

```
public int compareTo(Object obj)
{
    Car temp = (Car) obj;    // note required cast
    if (mileage < temp.mileage)
        return (-1);
    if (mileage > temp.mileage)
        return (1);
    return (0);
}
```

The Coin class defined below would implement compareTo in the following way:

```
public class Coin implements Comparable
{
    public Coin(double aValue, String aName)
    {
        public double getValue( ) {. . .}
        public String getName( ) {. . .}
        private double value;
        private String name;
    }
    public int compareTo(Object obj)
    {
        Coin tempMoney = (Coin) obj;    // note required cast
        if (value < tempMoney.value)
            return -1;
        else
            if (value > tempMoney. value)
                return 1;
        else
            return 0;
    }
}
```

Note: Anytime you implement an abstract method, you must use the same method signature as that of the original method. The same number of parameters and the parameter types(s) must be used as this is part of the method signature. The compareTo(Object) method in the Comparable interface can only have a generic object passed to it and then you would need to convert it (cast it) to the correct type before using it.

ex.1a)

```
public class HeyPayAttention implements Comparable
{
    public int compareTo(Object obj)
    {
        HeyPayAttention temp = (HeyPayAttention) obj;    // note required cast
        ...
        return etc;
    }
}
```

ex.1b)

```
public class HeyPayAttention implements Comparable
{
    public int compareTo(HeyPayAttention obj)  // an ERROR!
    {ffffff3f3
        ...
        return etc;
    }
}
```

Java 5 has a shortcut that can be used!

ex.2)

```
public class HeyPayAttention implements Comparable <HeyPayAttention>
{
    public int compareTo (HeyPayAttention obj)  // now ok!
    {
        ...                                           // casting no longer needed
        return etc;
    }
}
```

Interfaces vs. Classes
An interface type is similar to a class, but there are several important differences:

- All methods in an interface type are abstract; they don't have an implementation
- All methods in an interface type are automatically by default public
- An interface type does not have instance fields
- An interface can have constants but they must be declared public static final

Converting Between a Class and an Interface Types (9.2)

- You can convert from a class type to an interface type, provided the class implements the interface
- BankAccount account = **new** BankAccount(10000);
  Measurable x = account;   // OK
- Coin dime = **new** Coin(0.1, "dime");
  Measurable x = dime;   // Also OK
- Cannot convert between unrelated types
  Measurable x = **new** Rectangle(5, 10, 20, 30);   // ERROR
  Because Rectangle doesn't implement Measurable
- *In Java, method calls are always determined by the type of the actual object, not the type of the reference object

Casts
Add coin objects to DataSet:

```
DataSet coinData = new DataSet( );
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.1, "dime"));
Measurable max = coinData.getMaximum( );   // Get the largest coin
```

but

```
String name = max.getName( );    // ERROR
```

(It's not of type Coin and getName( ) is a Coin method)

*You need a cast to convert from an interface type to a class type
You know it's a coin, but the compiler doesn't.
Apply a cast:

```
Coin maxCoin = (Coin) max;
String name = maxCoin.getName( );
```

or `String name = (Coin) max.getName( );`

Note: If you are wrong and max isn't a coin, the compiler throws an exception
*Difference with casting numbers:
When casting number types you agree to the information loss: **int** x = (**int**)2.5;
When casting object types you agree to that risk of causing an exception

ex. 1) Are the following Valid or Invalid given that Bird and SkiJumper both implement Flyer and SkiJumper also implements Athlete

a) Athlete a = new SkiJumper( ); — a) Valid

b) SkiJumper b = new Athlete( ); — b) Invalid

c) Athlete c = new Bird( ); — c) Invalid

d) SkiJumper d = new Bird( ); — d) Invalid

e) Bird e = new Bird( );
Flyer x = e; — e) Valid

f) Bird f = new Bird( );
Flyer x;
x = f; — f) Valid

g) Bird g = new Bird( );
Flyer x;
g = x; — g) Invalid

h) Bird h = new Bird( )
Flyer x;
h = (Bird) x; — h) Valid

i) Bird i = new Bird( );
SkiJumper m = new SkiJumper( );
i = (Bird) m; — i) Invalid

j) SkiJumper j = new SkiJumper( );
Flyer x;
x = (Flyer) j; — j) Invalid

k) Athlete k = new SkiJumper( );
Flyer x = (Flyer) k; — k) Valid

l) Flyer l = new Flyer( ); — l) Invalid

Polymorphism (9.3) (means many shapes or forms) - behavior can vary depending on the actual type of an object

Interface variable holds reference to object of a class that implements the interface:
Measurable x;
x = **new** BankAccount(10000);
x = **new** Coin(0.1, "dime");

Note that the object to which x refers doesn't have type Measurable;
the type of the object is some class that implements the Measurable interface.

You can call any of the interface methods such as:
double m = x.getMeasure( );

Which method is called?   Depends on the actual object
If x refers to a bank account, calls BankAccount.getMeasure
If x refers to a coin, calls Coin.getMeasure

This is called *late (or dynamic) binding* and is resolved at runtime in by the virtual machine, This is different from overloading; overloading is resolved by the compiler (*early binding*) and will be discussed in chapter 11.

Checking for object type

The instanceof operator is used to determine whether an object belongs to a particular type.

ex.1) max instanceof Coin
will return true if the object max is a Coin instance and false otherwise

AP Programming - Chapter 9 Lecture

Inner Classes

A class can be defined inside a method

ex.1)

```
public class DataSetTester3
{
  public static void main(String[ ] args)
  {
    class RectangleMeasurer implements Measurer
    {
      . . .
    }
    Measurer m = new RectangleMeasurer( );
    DataSet data = new DataSet(m);
    . . .
  }
}
```

Note: If inner class is defined inside an enclosing class, but outside its methods, it is available to all methods of enclosing class. The scope of the class is restricted to a single method or the methods of a single class. An inner class method can access variables declared in that inner class, local variables that are declared final, and instance field members of the outer class.

Declared inside a method

```
class OuterClassName
{
  method signature
  {
    . . .
    class InnerClassName
    {
      // methods
      // fields
    }
    . . .
  }
  . . .
}
```

Declared inside the class

```
class OuterClassName
{
  // methods
  // fields
  accessSpecifier class InnerClassName
  {
    // methods
    // fields
  }
  . . .
}
```