**MAHARISHI UNIVERSITY of MANAGEMENT**

*Engaging the Managing Intelligence of Nature*

**Computer Science Department**

# CS401 Modern Programming Practices (MPP)

Professor Paul Corazza

# Lecture 5: Inheritance and Abstractions

*Engaging Abstract Levels to Enrich Life*

## Wholeness of the Lesson

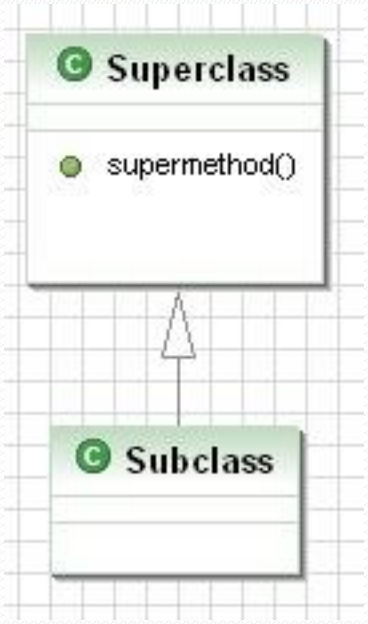
Both abstract classes and interfaces can be used in conjunction with polymorphism, but interfaces provide even more flexibility. Both make possible a variety of implementations or expressions of fundamental themes, the most extreme example of these being the fact that all Java classes inherit from Object. Likewise in the universe, objects form hierarchies of wholeness which express the unmanifest field of pure creative intelligence into all the specific structures of existence and intelligence.

## Overview of Topics

- Review of inheritance and its implementation in Java
- Inheritance and polymorphism
- Using abstract classes with polymorphism
- Interfaces in Java and using them with polymorphism
- Best practices concerning interfaces and some applications

# Review of Inheritance

```
class Superclass {
    protected void supermethod() {
        int x = 0;
    }
}
class Subclass {
}
class Main {
    public static void main(String[] args) {
        Superclass sub = new Subclass();
        //subclass has access to data and
        // methods of superclass
        sub.supermethod();
    }
}
```

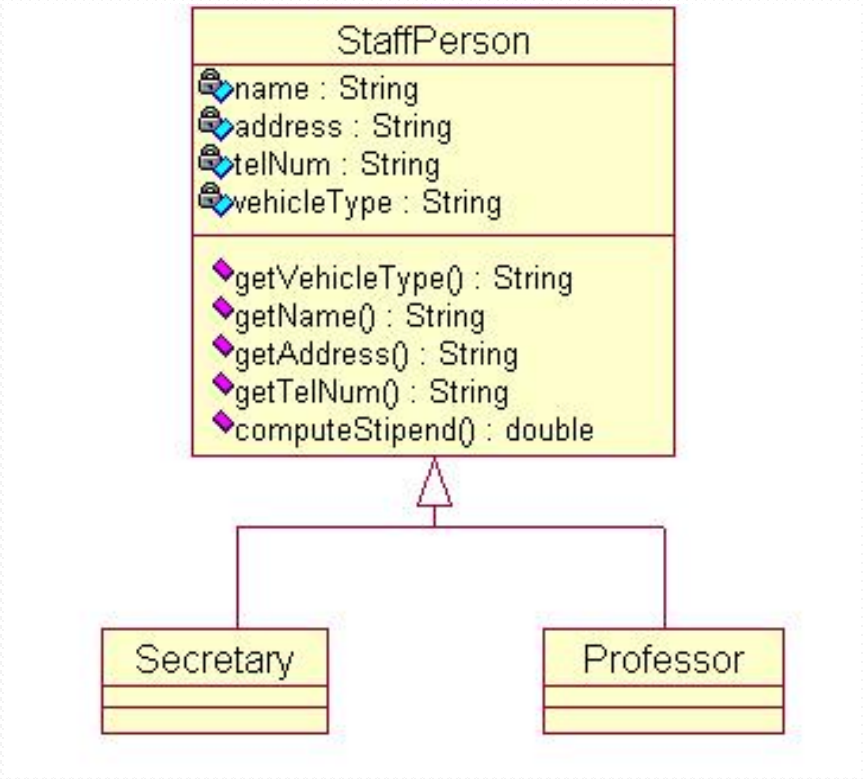


# Inheritance Arises . . .

As a way to *generalize* data and behavior of related classes

Secretary
- properties:
  - name
  - address
  - phone number
  - drivesVehicle
  - salary
- behavior:
  - computeSalary()

Professor
- properties:
  - name
  - address
  - phone number
  - drivesVehicle
  - salary
- behavior:
  - computeSalary()



See demos in `lesson5.lecture.polymorphism1, lesson5.lecture.polymorphism2`

## And . . .

As a way to *extend* the behavior of a particular class

```
class Employee {
   //constructor
   Employee(String aName,
            double aSalary) {
         name = aName;
         salary = aSalary;
   }
   public String getName() {
         return name;
   }
   public double getSalary() {
         return salary;
   }
   public void raiseSalary(double byPercent) {
         double raise = salary * byPercent / 100;
         salary += raise;
   }

   private String name;
   private double salary;
}
```

```
class Manager extends Employee {
   public Manager(String name, double salary) {
         super(name,salary);
         bonus = 0;
   }
   @Override
   public double getSalary() {
         //no direct access to private
         //variables of superclass
         double baseSalary = super.getSalary();
         return baseSalary + bonus;
   }
   public void setBonus(double b) {
         bonus = b;
   }
   private double bonus;
}
```

## Rules Concerning Inheritance

- A subclass constructor must make use of one of the superclass constructors (see Manager class), but does not need the same signature as any of these constructors
- A class can have multiple (overloaded) constructors. To call one constructor from another, "this" is used (must be the first line of the constructor). Example:

```
    public Employee(String name) {
        this(name, 0.00);
    }
```

- A constructor can call a superclass constructor using "super". See Manager class (also notice super is used in another way to call a superclass method).
- If A is a subclass of class B, when the constructor of A is invoked, there is a specific sequence of steps by which the static/instance variables are initialized and the bodies of the two constructors are executed.

  DEMO: package lesson5.lecture.orderofexec

# Using the Default Constructor

A subclass may make use of the implicit (default) constructor *only if* either

- the no-argument constructor of the superclass has been explicitly defined, OR
- no constructor in the superclass is explicitly defined

In either of these cases, the subclass may make use (possibly implicitly) of the superclass' default constructor.

Example

```
//This is ok
class Employee{
            //…//
}
class Manager extends Employee {
            //…//
}
```

Example

```
//This is ok

class Employee{
   Employee(String name, double salary) {
            //…//
   }

    //explicit coding of default constructor
    //since another constructor is present
    Employee() {
         //…//
    }
}

class Manager extends Employee {
   //no explicit constructor call here,
   //so the  superclass default
   //constructor is used implicitly
}
```

## Inheritance ⟶ Polymorphism

```
class ManagerTest {
   public static void main(String[] args) {
      Manager boss = new Manager("Boss Guy", 80000, 1987, 12, 15);
      boss.setBonus(5000);

      Employee[] staff = new Employee[3];

      staff[0] = boss;
      staff[1] = new Employee("Jimbo", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tommy", 40000, 1990, 3,15);

      //print names and salaries
      for(Employee e : staff) {
         System.out.println("name: " + e.getName() +
                                    "salary: "+ e.getSalary());
      }
   }
```

# Polymorphism and Late Binding

*Polymorphic types*. The 0th element of the `staff` array was defined to be of type `Manager`, yet we placed it in an array of `Employee` objects. The fact that an object variable can refer to an object of a given type as well as objects that belong to subtypes of the given type is called *polymorphism*.

*Dynamic binding*. When the `getSalary` method is called on `staff[0]`, the version of `getSalary` that is used is the version that is found *in the Manager class*. This is possible because the JVM keeps track of the actual type of the object when it was created (that type is set with execution of the "new" operator). The correct method body (the version that is in `Manager`) is associated with the `getSalary` method at runtime – this "binding" of method body to method name is called *late binding* or *dynamic binding*.

## Main Point 1

When a method is called on a subclass, the JVM by default uses *dynamic binding* to determine the correct method body to execute. Early binding (and hence a slight improvement in performance) can be forced by declaring a method `final`.

In a similar way, it is said (Maharishi, *Science of Being*) that an enlightened individual need not continually plan and prepare in order to meet the needs of his daily life – instead, the enlightened enjoys spontaneous support of nature, and sees what to do as situations arise.

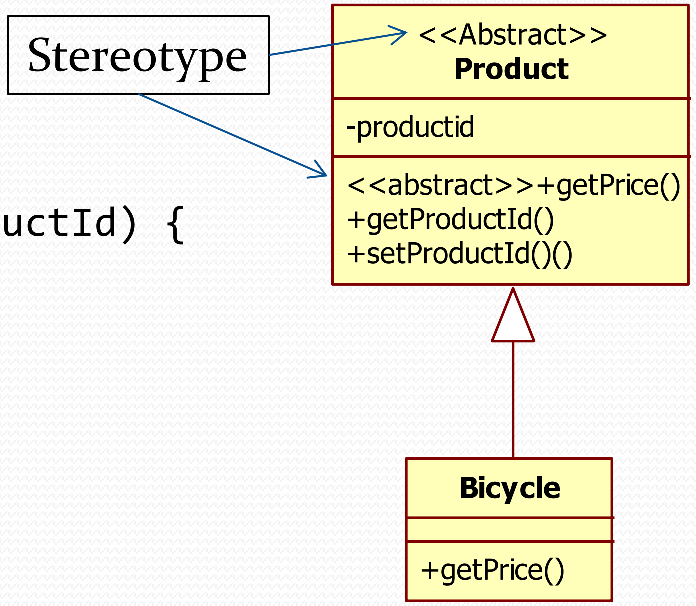
Such individuals are an analogue to "late binding".

## Abstract Classes and Methods

- When a class is declared to be abstract, it cannot be instantiated directly.
- When a method in a class is declared abstract, it means no implementation of the method is provided, and it must be implemented by a subclass.
- When a method is declared to be abstract, its enclosing class must also be declared abstract.
- Abstract classes may include instance variables and other non-abstract (implemented) methods

# Abstract Class Example

```
public abstract class Product {
    private String productId;

    public abstract double getPrice();

    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
}


public class Bicycle extends Product {

    @Override
    public double getPrice() {
        return 230.45;
    }
}
```



## Abstract Classes and Polymorphism

When using polymorphism:

- *Default implementation.* Sometimes, a method common to subclasses has a natural default implementation.

  Example: The `getSalary` method of `Employee`.

- *Abstract method.* At other times, a common method has no default implementation and so it is declared *abstract* – the implementation of the method in this case must be handled by subclasses.

  Example: The `computeStipend` method of `StaffPerson`

# Java Interfaces

A Java *interface* is like an abstract class execpt:

- No instance variables or implemented methods can occur. [Public static final variables can be defined, but not instance variables.]
- Can implement more than one interface. [Note: no class can have more than one *superclass*.] Syntax:

```
        MyClass implements Intface1, Intface2, Intface3
```

- Can also extend *and* implement. Syntax:

```
        MyClass extends SuperClass implements Intface1, Intface2
```

*Example*: In Java, `ArrayList` implements 6 interfaces and extends one class. (What are they?)

Other features:

- One interface can extend another. Example: `List` extends `Collection`
- In many cases, when an abstract class is used for polymorphism, an interface could be used instead.
- All methods are automatically public and abstract

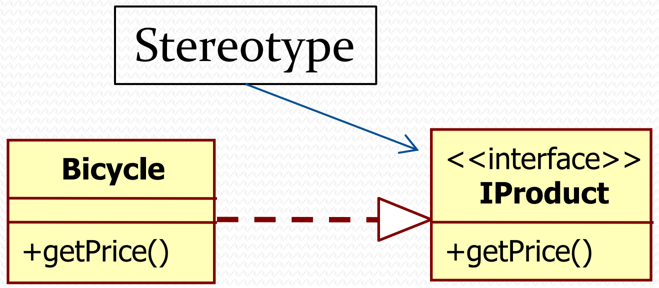
Demo: See package `lesson5.lecture.intfaces2`

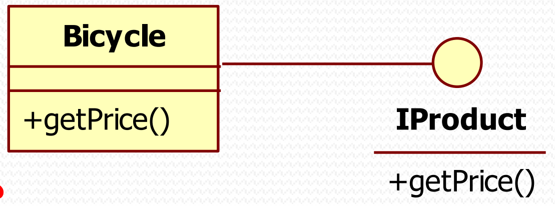
# Interface Example

UML has two different notations for Interfaces

```
public interface IProduct {
    public abstract double getPrice();
}


public class Bicycle implements IProduct {

    @Override
    public double getPrice() {
        return 230.45;
    }
}
```

Question: Is there a good reason to use an interface instead of an abstract class?

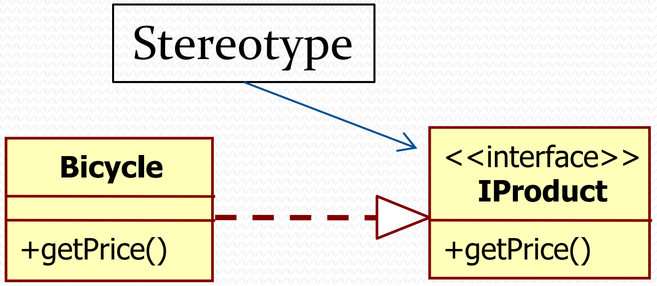


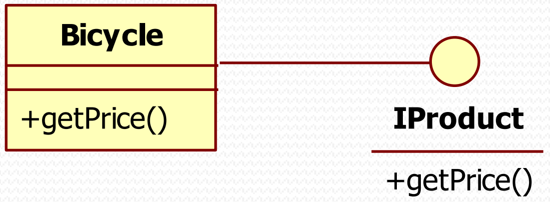


# Interface Example

UML has two different notations for Interfaces

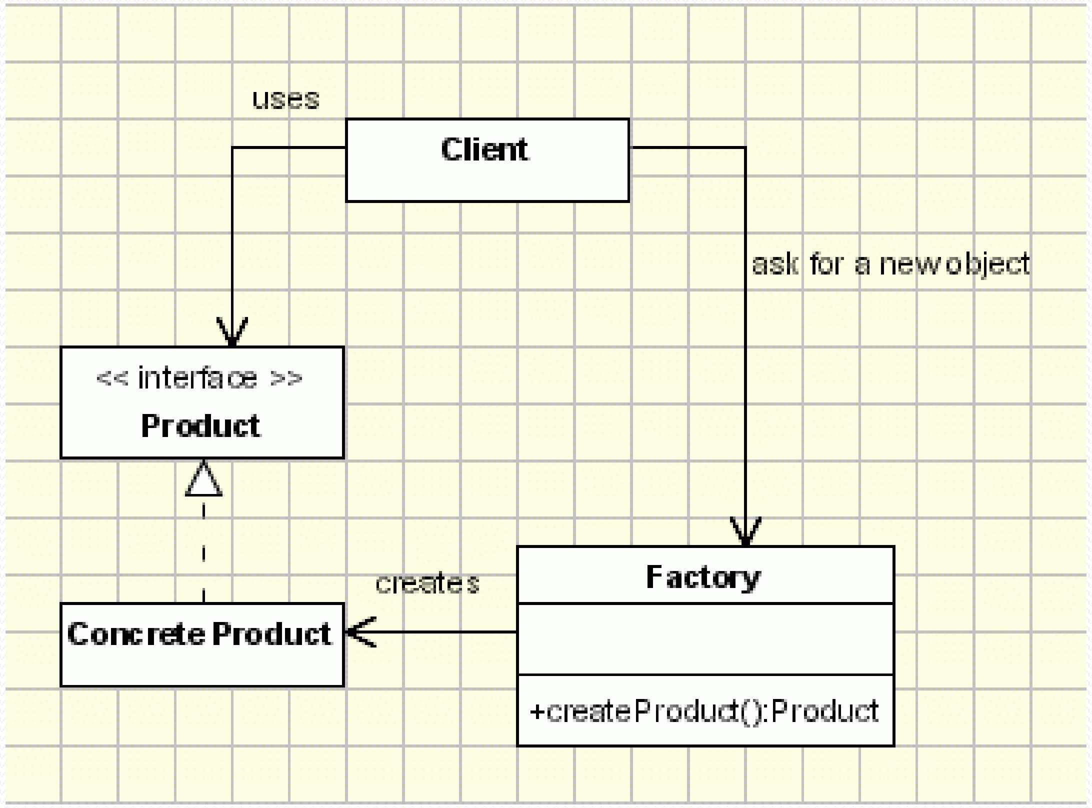
```
public interface IProduct {
    public abstract double getPrice();
}



public class Bicycle implements IProduct {

    @Override
    public double getPrice() {
        return 230.45;
    }
}
```





One benefit: Bicycle ***extends*** the superclass Vehicle and at the same time ***implements*** IProduct as does every product in our system.

## Application of Interfaces: Object Creation-Factory



Demo: lesson5.lecture.factorymethods.ConnectExample

Issues:

- Classes without public, protected, or package-level constructors cannot be subclassed
- The factory method must be distinguished from other static methods.
  - Use conventional naming
    - getInstance [often used to invoke a Singleton]
    - newInstance [used in Class to obtain an instance from a class]
    - getType
    - newType
    - openConnection [URLConnection example]

# Interfaces as *Types*

- Primitives (int, float, etc) are types
- Classes provide an 'interface' and an implementation
  - In this context the interface is 'The publicly exposed methods'
    - More specifically 'what you can do with a class of this type'
  - This 'interface' is therefore considered to be the type

- A Java Interface provides a pure 'type'
  - Just specifies what you can do with an implementer of the interface

# Interfaces and Polymorphism

- Since interfaces are types just like classes, they can be used in the same polymorphic ways that classes can be used. [Recall that `List` is an interface in the Java collections library]
  - As variable type:

    ```
    List<Student> students = new ArrayList<Student>();
    ```
  - As argument type:

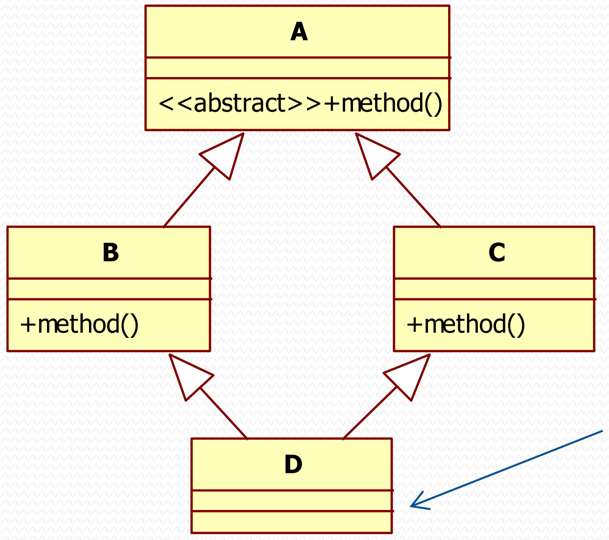
    ```
    public void createTranscripts(List<Student> students)
    ```
  - As return value type:

    ```
    public List<Student> findStudents(String country)
    ```

See Demos in `lesson5.lecture.intfaces1`, `lesson5.lecture.intfaces2`

## Multiple Inheritance in Other Languages (like C++)

- Diamond Problem
  - Which (conflicting) implementation do we use?

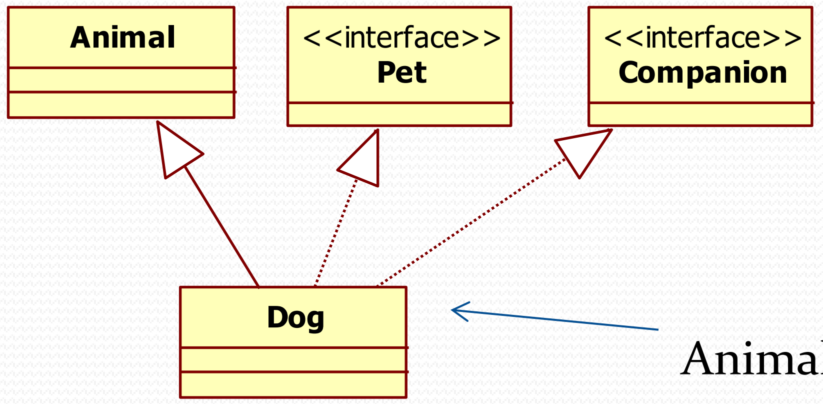


Which version of `method()` Does D inherit?

  - Note there is no conflict among the types (interfaces) declared by the different classes

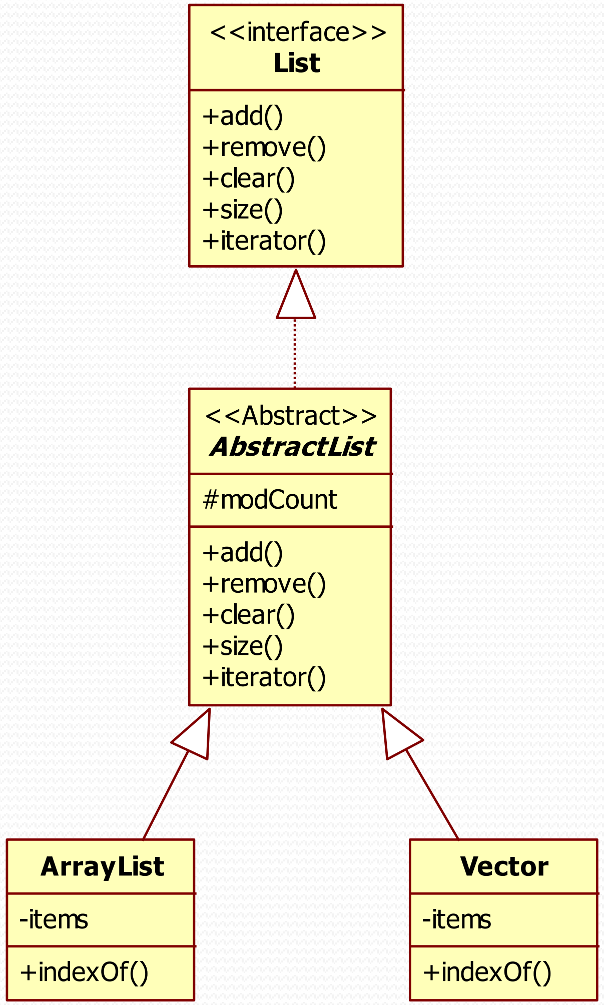
# Java’s Answer (pre - Java SE 8)

- *Implementation* can only be ‘inherited’ / extended once
- *Types* can be ‘inherited’ / implemented multiple times
  - No limit on the amount of interfaces you can implement
  - A single interface can extend multiple other interfaces



Dog ‘is a’:
Animal, Pet, and Companion

## Interface vs. Abstract Class: Pre-Java 8



**Interface has no implementation**

• Important types should always be interfaces to allow for 'multiple' inheritance

**Interface takes abstraction one step further.**

• Abstract class is an abstraction of its subclasses – provide common implementation.

• Interface is an abstraction of its (abstract) subclasses – provides common type.

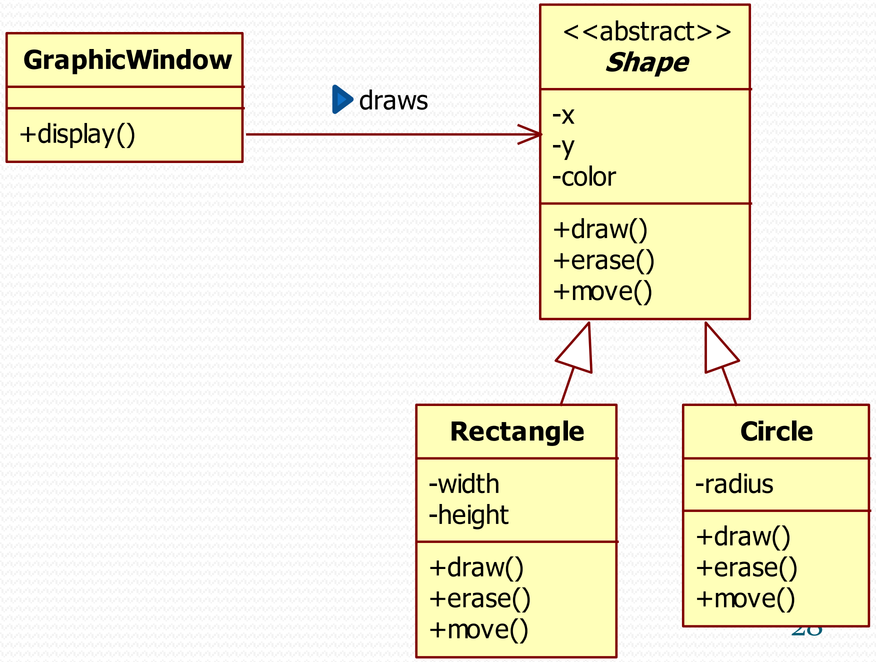
## Some Advantages of Interfaces

- They support the safe part of multiple inheritance
- They enforce information hiding and encapsulation.
  - Remember encapsulation is about grouping data and methods together for ease of use. Information hiding hides the implementation from the public ‘interface’.
- They support change – implementation can be changed behind the interface
- They support development of code in parallel – each team can rely on other teams interfaces even before they are implemented.

## Flexibility of Interfaces – In class exercise

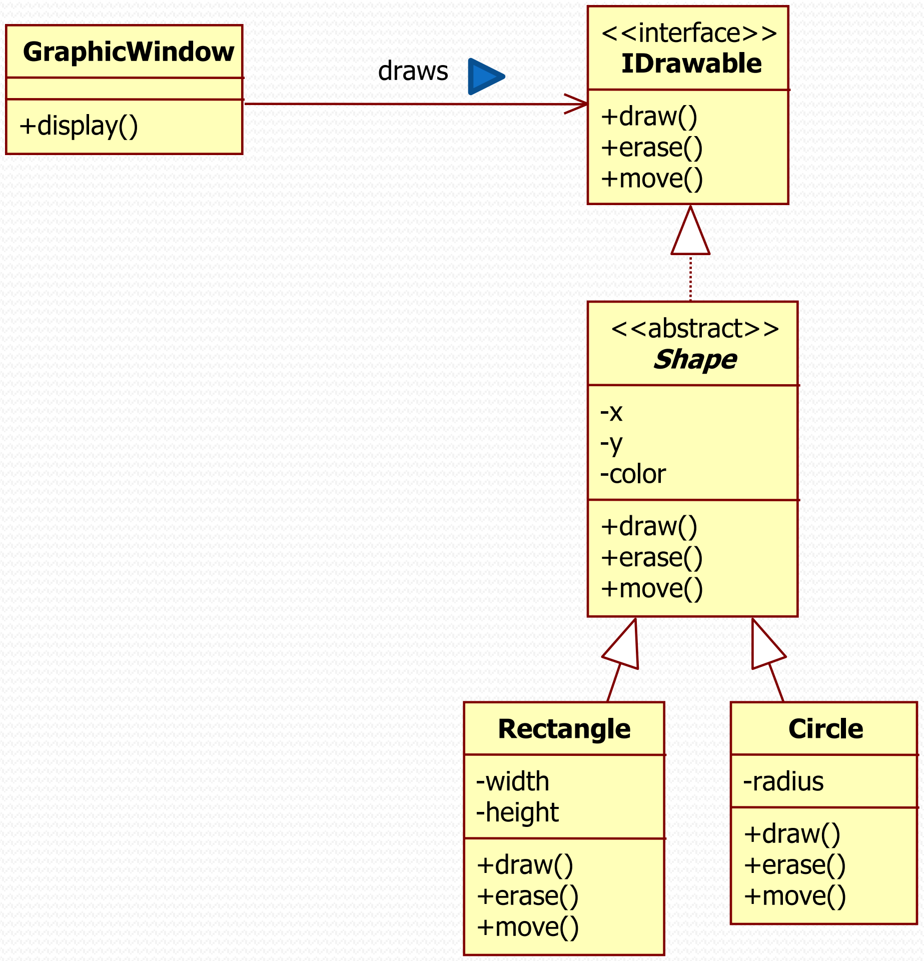
- Interfaces let you take greater advantage of polymorphism in your designs, which in turn helps you make your software more flexible.
- In your small groups modify this class hierarchy so it supports display of images.

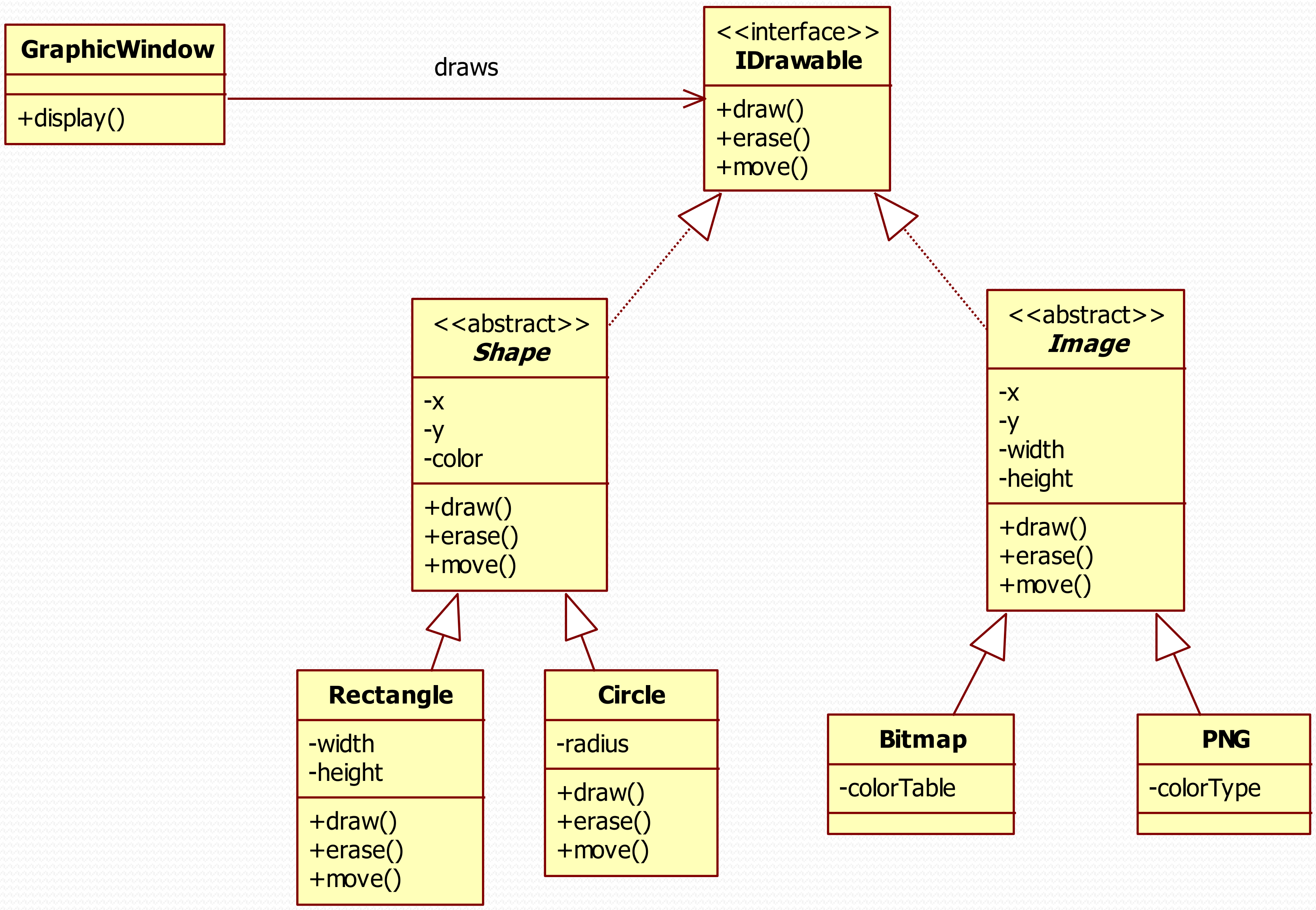
[Use an interface to create greater abstraction.]



# A Solution

- With an interface, we can easily add a new class hierarchy

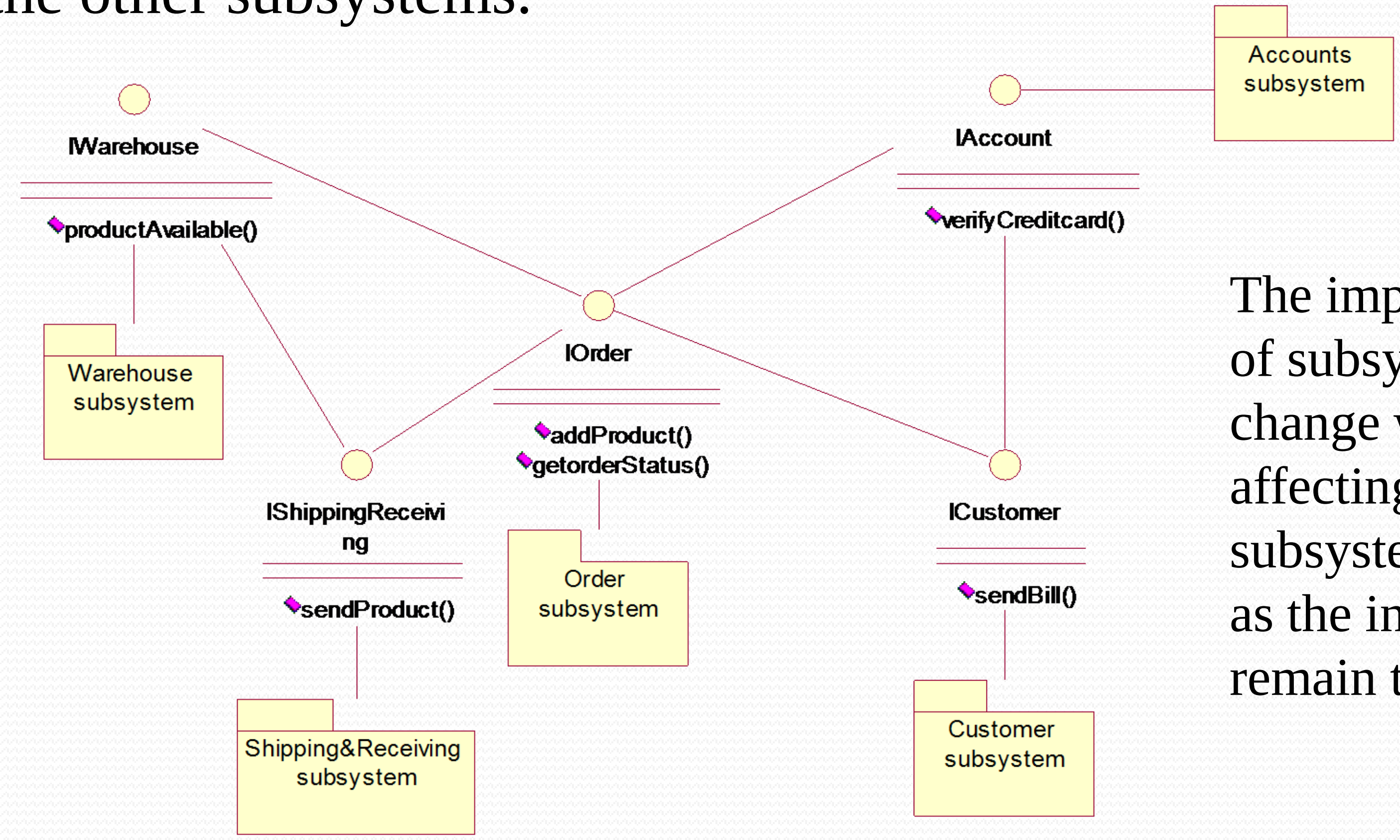




GraphicWindow
+display()
draws
<<interface>>
IDrawable
+draw()
+erase()
+move()
<<abstract>>
Shape
-x
-y
-color
+draw()
+erase()
+move()
<<abstract>>
Image
-x
-y
-width
-height
+draw()
+erase()
+move()
Rectangle
-width
-height
+draw()
+erase()
+move()
Circle
-radius
+draw()
+erase()
+move()
Bitmap
-colorTable
PNG
-colorType

# Interfaces Support Team Development

First define the interfaces for all subsystems, then every programmer can program one subsystem by using the interfaces of the other subsystems.



The implementation of subsystems may change without affecting all other subsystems, as long as the interfaces remain the same.

## Interfaces in System Development

If you have a subsystem that represents an abstraction that may have multiple implementations, whether the subsystem is a single object, a group of objects, an entire Java applet or application, you should define Java interfaces through which the rest of the world communicates with that subsystem.

When you use interfaces in this way, you decouple the parts of your system from each other and generate code that is more flexible: more easily changed, extended, and customized. (encapsulation and abstraction)

**Program to Interface (P2I), rather than implementation.**

## Best Practices: When to Use Interfaces?

1. Always use interfaces for subsystem development
2. Prefer interfaces over multiple levels of abstract classes
3. If an abstract class will provide all the abstraction you need, then do not add an interface.
4. Code will be *read* many more hours than it will take to *write* it. Make it as simple, elegant, and clear as possible.

   "**as simple as possible, but no simpler**" (Einstein)

# The Evolving API Problem

**<u>Problem</u>**: You have created a library of Java classes and you have a substantial clientele who make use of your library. Your library contains numerous interfaces, for which you have implementations (in some cases, multiple implementations) in your library code.

Suppose you now want to add new functionality to your library. In many cases, you will need to add new methods to some of your interfaces. You think "I have to be careful not to change the signature of my interface methods, but adding new methods should not create a problem for my users." You add some methods, and distribute a new release.

A few days later you get hundreds of complaints that your new code has broken the code of your clients who were using your library. What went wrong?

**Explanation:**

Clients created their own implementations of your interfaces, in earlier versions of your code. When you add new methods to those interfaces, their code breaks because they do not have implementations of the new methods.

New features of interfaces in Java 8 provide a solution to this and other issues concerning interfaces.

## Main Point 2

Abstract classes and Interfaces are both strongly related to the concept of Inheritance.

The interface is the most abstract entity in the class diagram, and by pro-gramming to interfaces, we generate more flexible code.

Greater abstraction holds the possibility of greater potential; this priniciple is especially evident in the case of the unified field.

## Summary

Today we looked at modeling Abstractions through Inheritance, Abstract classes and Interfaces. We also looked at the design decisions that go along with using them:

- Abstract classes and Interfaces contain less and less implementation details, and instead focuses more on general abstract parts (like types)
- With Java it is important to always Program to Interface - P2I.
- Use interfaces for ‘multiple’ inheritance, encapsulation, flexibility, and team work.

## Connecting the Parts of Knowledge With the Wholeness of Knowledge

### Abstraction in the Form of Abstract Classes and Interfaces

1. A concrete class embodies a set of concrete behaviors on a set of data whereas an abstract class embodies some concrete behaviors and at the same time gives expression to new, unimplemented behaviors in the form of *abstract methods*.

2. Interfaces (in pre-Java 8) give expression to abstract "unmanifest" behaviors, "pure possibilities," which can be realized in an endless number of ways by implementing classes.

---

3. *Transcendental Consciousness* is a field of all possibilities.

4. *Impulses Within the Transcendental Field.* Pure consciousness, as it prepares to manifest, is a "wide angle lens" making use of every possibility for creative ends.

5. *Wholeness Moving Within Itself.* In Unity Consciousness, awareness is flexible enough to give expression to any possibility that is needed at the time.