MAHARISHI UNIVERSITY of MANAGEMENT

Engaging the Managing Intelligence of Nature

Computer Science Department

CS401 Modern Programming Practices (MPP)

Professor Paul Corazza
and
Professor Ankhtuya Ochirbat

Object Diagrams

- An **object diagram** is a snapshot of the objects in a system at a point in time.
- Because it shows instances rather than classes, an object diagram is often called an *instance diagram*.

Object diagrams look superficially similar to class diagrams however the boxes represent specific instances of objects.

Boxes are titled with :-

objectName : ClassName

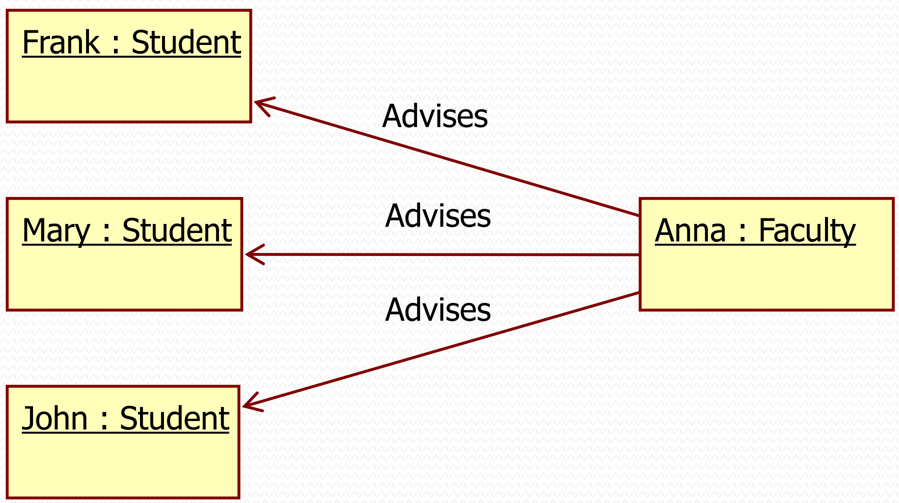
As each box describes a particular object at a specific moment in time the box contains attributes and their values (at that moment in time).

attribute = value

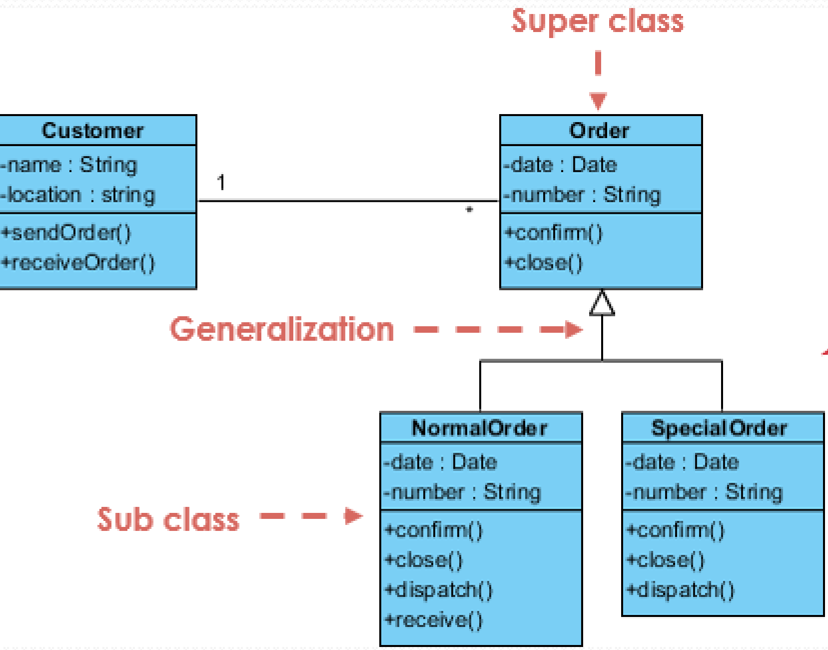
These diagrams are useful for illustrating particular 'snapshot' scenarios during design.

Object Diagram Syntax

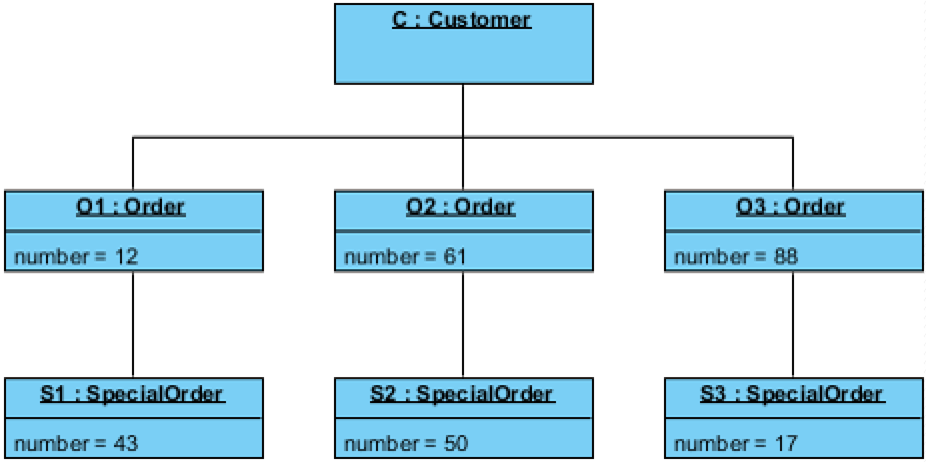
- Underlining indicates it's an object
- Usually shows colon separated name and type
- Associations in object diagrams don't have multiplicities
- Associations may or may not display an arrowhead



Example



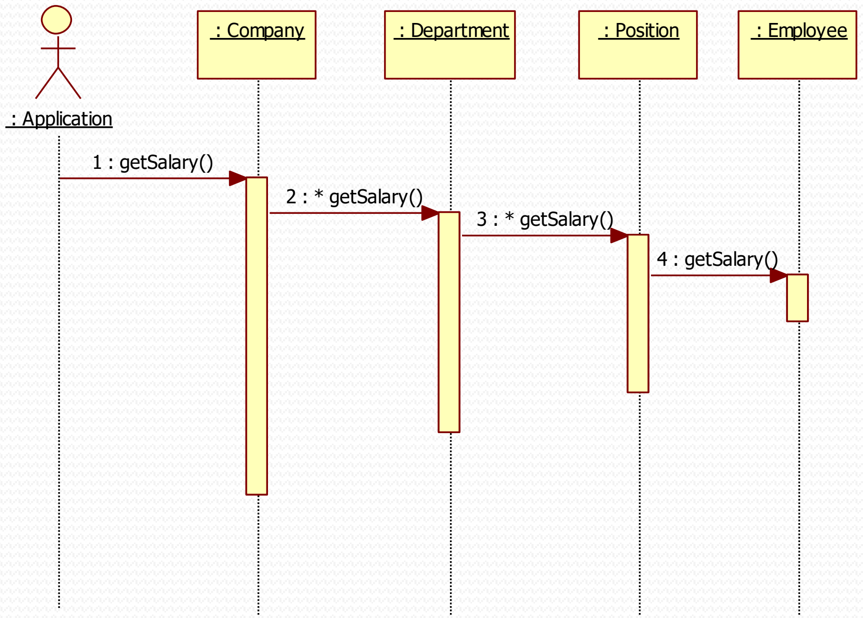
Super class
Customer
-name : String
-location : string
+sendOrder()
+receiveOrder()
1
*
Order
-date : Date
-number : String
+confirm()
+close()
Generalization
NormalOrder
-date : Date
-number : String
+confirm()
+close()
+dispatch()
+receive()
SpecialOrder
-date : Date
-number : String
+confirm()
+close()
+dispatch()
Sub class



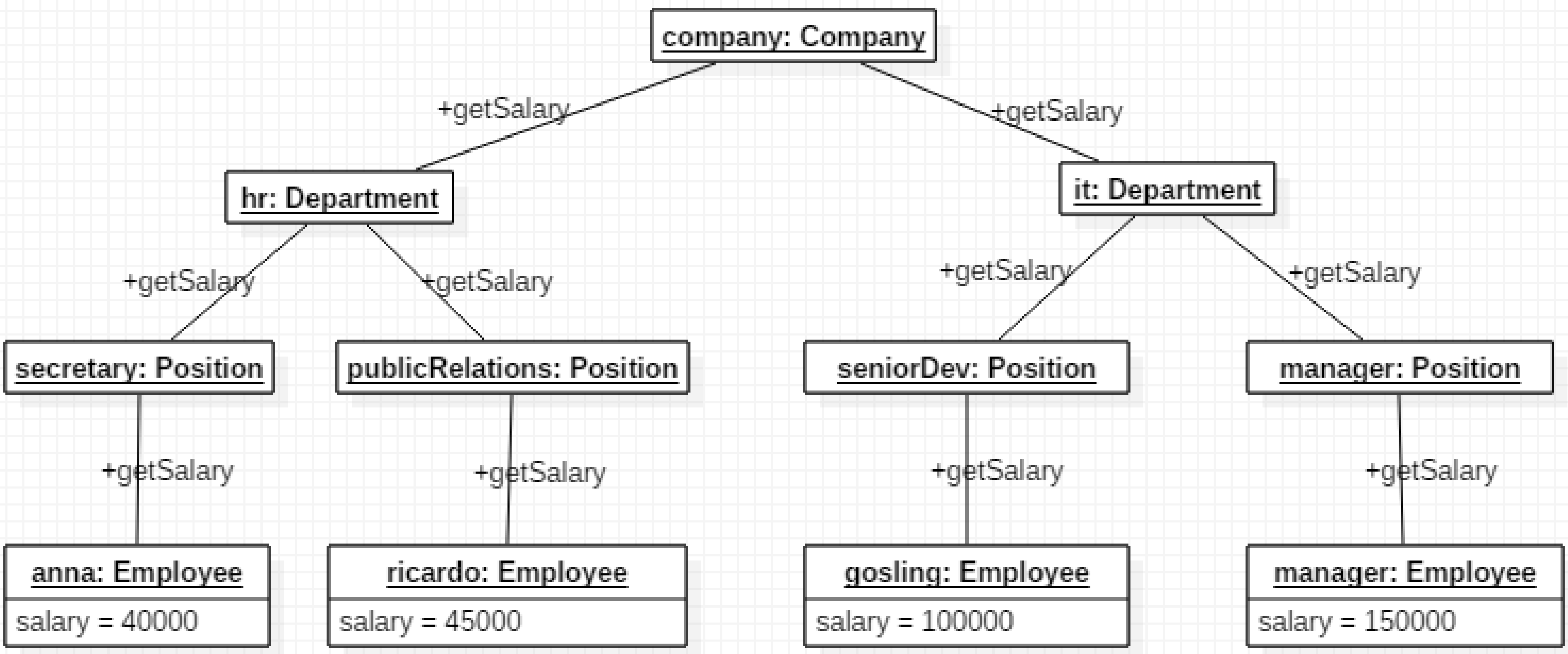
C : Customer
O1 : Order
number = 12
O2 : Order
number = 61
O3 : Order
number = 88
S1 : SpecialOrder
number = 43
S2 : SpecialOrder
number = 50
S3 : SpecialOrder
number = 17

Exercise

- Create an object diagram that captures a scenario for the computeSalary problem.
- Assume there is one instance of Company, which has two Departments, each with two positions, each filled with one employee. Invent salaries for the employees.



Solution



company: Company
+getSalary
+getSalary
hr: Department
it: Department
+getSalary
+getSalary
+getSalary
+getSalary
secretary: Position
publicRelations: Position
seniorDev: Position
manager: Position
+getSalary
+getSalary
+getSalary
+getSalary
anna: Employee
salary = 40000
ricardo: Employee
salary = 45000
gosling: Employee
salary = 100000
manager: Employee
salary = 150000

Main Point

Object Diagrams show the relationships between objects, where each object is an instance of a class, and each reference is represented by a single arrow or line.

This phenomenon illustrates the principle that *the whole is greater than the sum of the parts*: The objects (parts) on their own are not the important focus for an object diagram. What is important is how the objects relate; together, objects and their relationships form a whole that is more than just the sum of individual objects collected together.

Lecture 5: Abstract Classes and Interfaces

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. 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.

Rules About Abstract Classes

- 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

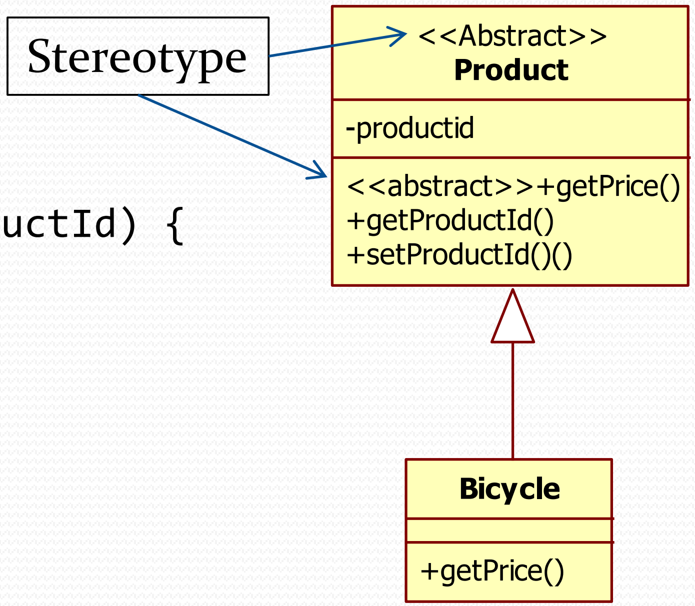
```java
public abstract class Product {
    private String productId;

    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.
  <u>Example</u>: The `getSalary` method of `Employee`. (Lesson 3)

```
//from Employee
public double getSalary() {
    return salary;
}
```

```
//from Manager
public double getSalary() {
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```

- *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.
  <u>Example</u>: The `computeStipend` method of `StaffPerson` (Lesson 3)

```
public abstract class StaffPerson {
    abstract public double computeStipend();
```

```
public class Faculty extends StaffPerson {
    public double computeStipend() {
        return 4000.0;
    }
}
```

Pre-Java 8 Interfaces

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

- 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:

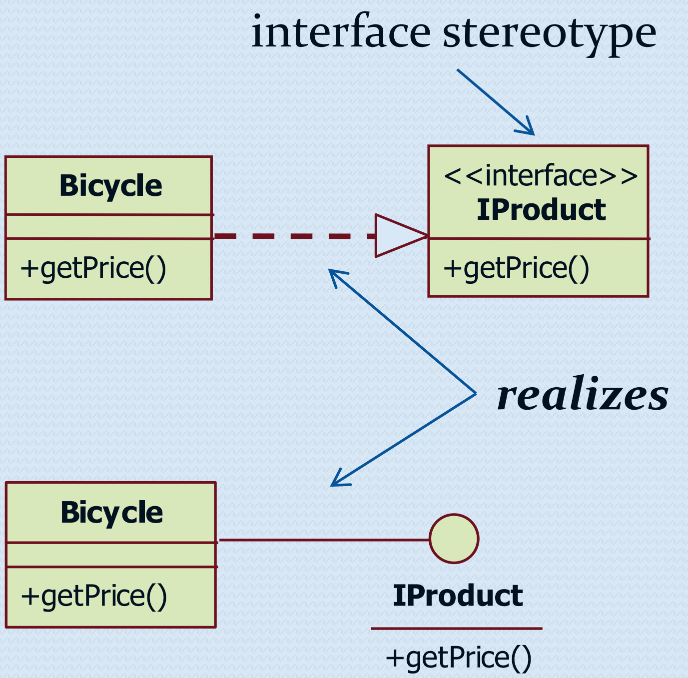
- One interface can extend another. Example: `List` extends `Collection`
- In many cases, when an abstract class is used for polymorphism, an interface could be used instead.
- All methods in an interface are automatically public and abstract

Interface Example

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

public class Bicycle implements IProduct {
    @Override
    public double getPrice() {
        return 230.45;
    }
}
```

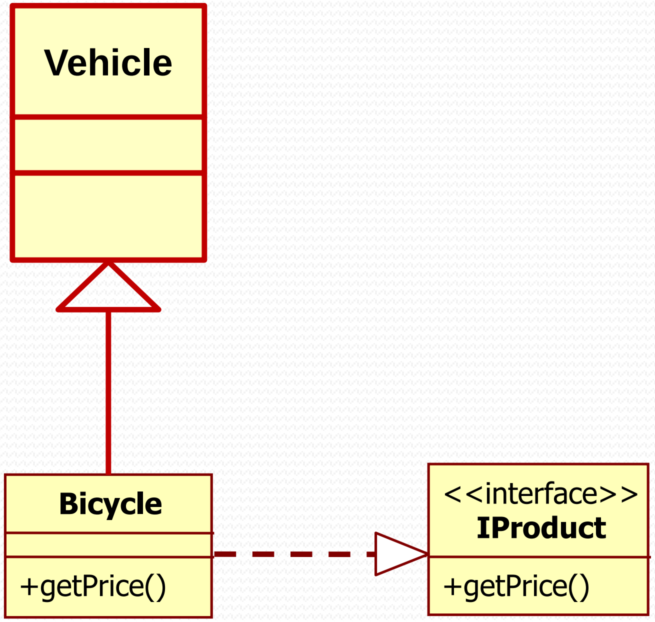
UML has two different notations for Interfaces



Interface Example

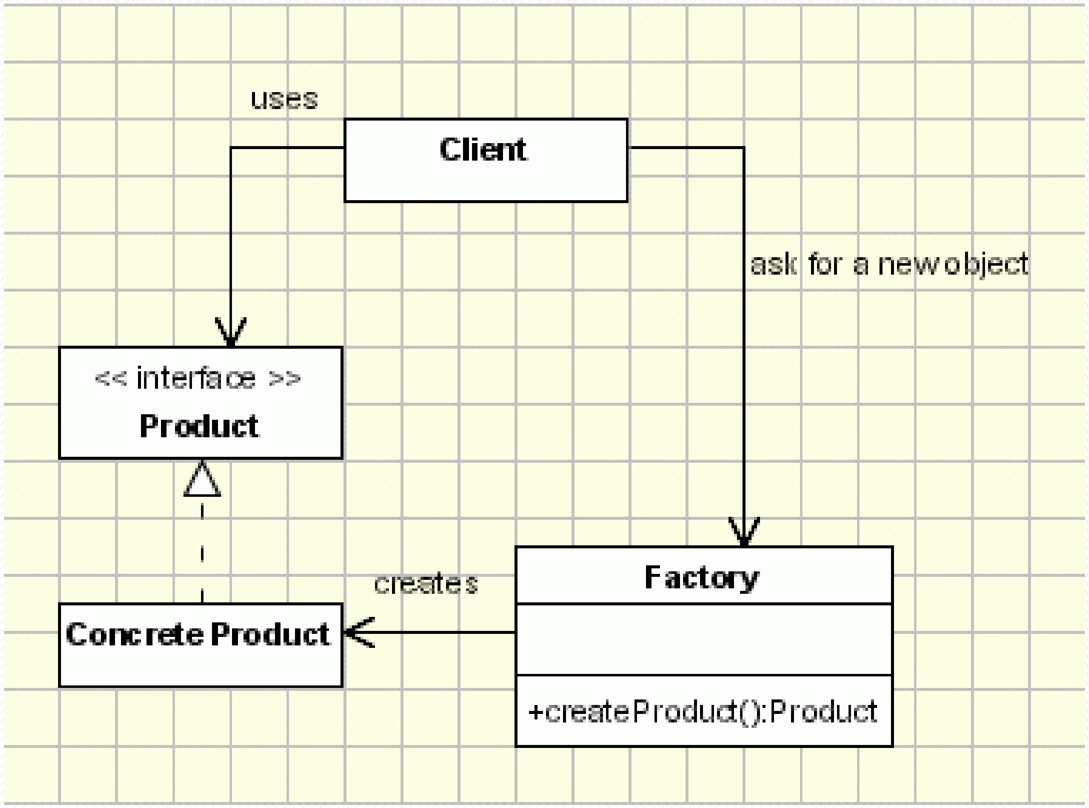
```java
public interface IProduct {
    public abstract double getPrice();
}

public class Bicycle extends Vehicle
      implements IProduct {
    @Override
    public double getPrice() {
        return 230.45;
    }
}
```



<u>One benefit</u>: **Bicycle can be treated as a subclass of Vehicle at the same time as it *implements* IProduct (as does every product in our system).**

Application of Interfaces: Object Creation Factory



Examples of Object-Creation Pattern

- In Java's `Collections` class, there are 32 static factory methods.

<u>Examples</u>:

```
public static <T> List<T> unmodifiableList(List<T> list)

public static <T> List<T> synchronizedList(List<T> list)
```

Exercise 5.1

The following lines of code make use of one of the factory methods in Collections. Compare these with the Object Creation Factory pattern diagram, and answer the following:

1. Which class plays the role of Factory?
2. Which class plays the role of Client?
3. Which interface plays the role of Product?
4. Which class plays the role of Concrete Product?

```java
public class MyClass {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Joe", "Bill", "Tom");
        List<String> unList = Collections.unmodifiableList(list);
    }
}
```

Solution

- Factory – Collections
- Client – MyClass
- Product – List<String>
- Concrete Product – Anonymous implementor of List

Advantages of Using Object-Creating Factory Methods in Place of Constructors

- Factory methods have a name – easier to understand what is being requested, and to distinguish between different kinds of invocations on an object.
- Can control access to instances. (See the Student/GradeReport and DataAccess examples, upcoming)
- Solves the problem that a class can have only one constructor with a given signature. (See Triangle example, upcoming)
- Unlike constructors, factory methods are not required to create a new instance every time they are invoked. (See Singleton creation example.)
- Unlike constructors, factory methods can return a subtype of the requested type, or an implementation of an interface type.

Simple Factory Methods

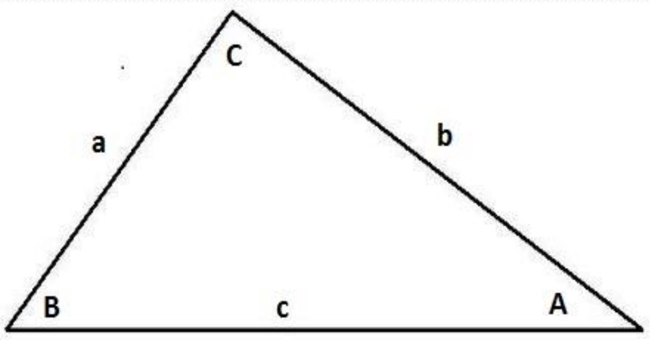
- Much of the benefit of having a separate object creation factory to create instances of objects (like the Collections class to produce different kinds of Lists) can be realized by just having a special "create" function inside a class to provide instances of the class – also called a *factory method*
- **Example**: In this Singleton implementation, you control how many instances are created by using a factory method to provide the instance

```
public class MySingleton {
    private static MySingleton instance = new MySingleton();
    private MySingleton() {}
    public static MySingleton getInstance() {
        return instance;
    }
}
```

Application: Problem of Multiple Constructors with Same Signature

Sometimes you may have a class that should provide two (or more) constructors that do different things, accept different input arguments, but the arguments are all of the same type. Java does not allow you to overload constructors with identical signatures. Using factory methods solves this problem

Triangle Example. By the laws of geometry, we can specify a triangle by specifying three of its sides, or by specifying two sides and the included angle. See demo: lesson5.lecture.factorymethods5.triangle



This is not allowed:

```
class Triangle {
    Triangle(double s1, double s2, double s3) {
        side1 = s1; side2 = s2; side3 =s3;
    }
    Triangle(double s1, double s2, double inclAngle){
        side1 = s1; side2 = s2; angle3 =inclAngle;
    }
}
```

Application: Controlling Access to Instances

<u>Student/GradeReport Example</u>. We wish to maintain a bidirectional 1:1 relationship between Student and GradeReport when data for these is read from a database. We wish to guarantee that instances of these classes are created in just the right way. After instances of each are created, we want to make sure that classes are read-only (i.e. immutable). To do this we:

1. Create instances using a Factory
2. Keep all classes related to GradeReport and Student in the same package
3. Provide only package level access for all constructors and setters
4. Declare all classes in the package as final

See lesson5.lecture.factorymethods6

Issues When Using Factory Methods

- Classes without public, protected, or package-level constructors cannot be subclassed (when factory methods are used, usually the constructor is private)
- The factory method name 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
        - valueOf [Integer.valueOf(int)]
        - of [LocalDate.of(year, month, day)]

Interfaces as *Types*

- Primitives (int, float, etc) are examples of simple types
- Classes provide an ‘interface’ and an implementation
  - In this context the interface is ‘The publicly exposed methods’ – the services provided by the class.
  - This ‘interface’ is therefore a way of specifying the type

- A Java Interface provides a pure ‘type’ – an abstraction of a class.
  - Just specifies what you can do with an implementer of the interface

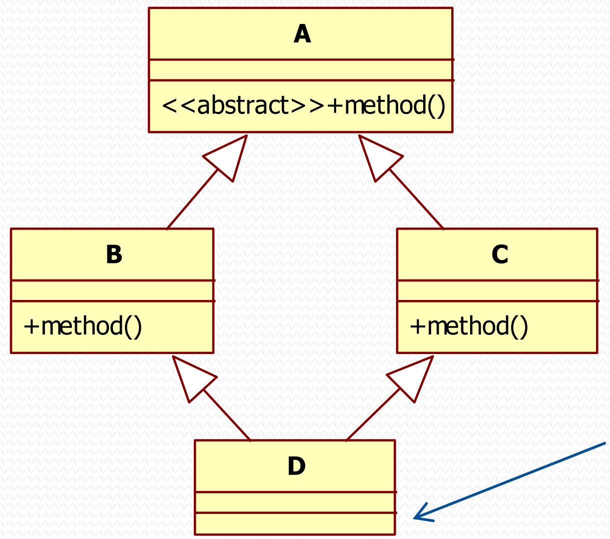
Interfaces and Polymorphism

- Since interfaces are types like classes, they can be used in the same polymorphic ways that classes can be used. [For these examples, 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?

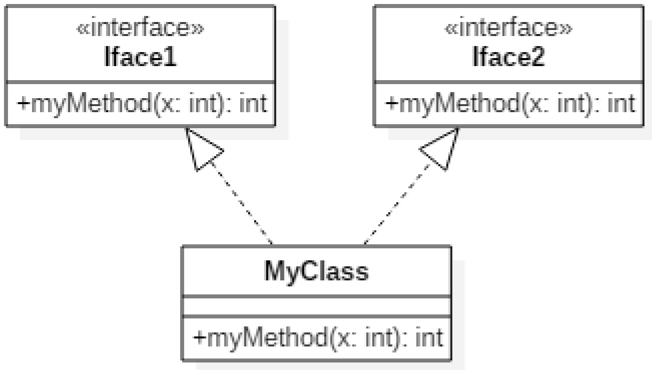
  - Note there is no conflict if A, B, C are understood to be interfaces

Exercise 5.2

Does the following code compile and run? Explain

```
public interface Iface1 {
    int myMethod(int x);
}
```

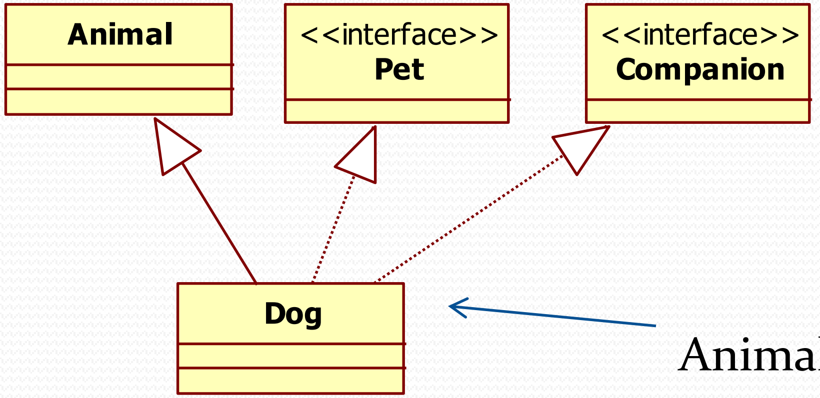
```
public interface Iface2 {
    int myMethod(int x);
}
```




```
public class MyClass implements Iface1, Iface2 {
    public int myMethod(int x) {
        return x + 1;
    }
}
```

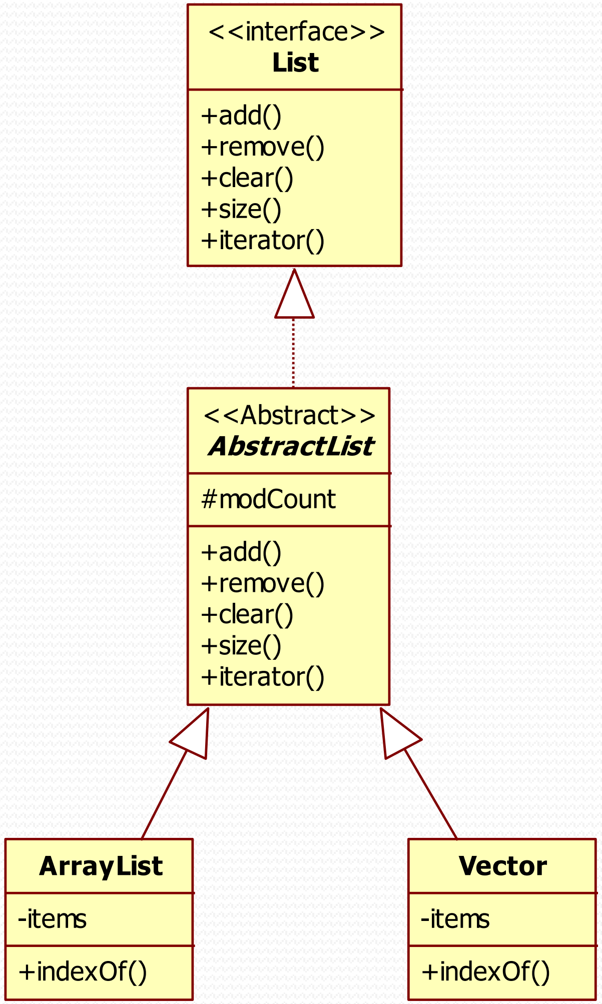
"Multiple Inheritance" **Before Java SE 8**

- *Implementation* can be 'inherited' / extended *only once*
- *Types* can be 'inherited' / implemented *multiple times*
  - No limit on the number of interfaces you can implement
  - A single interface can extend 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.

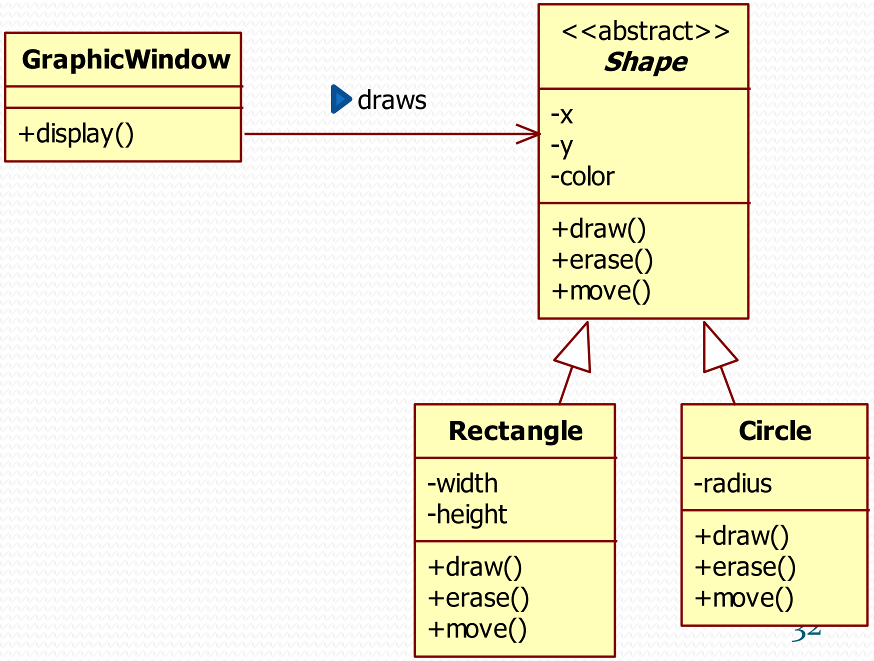
Some Advantages of Interfaces

- 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

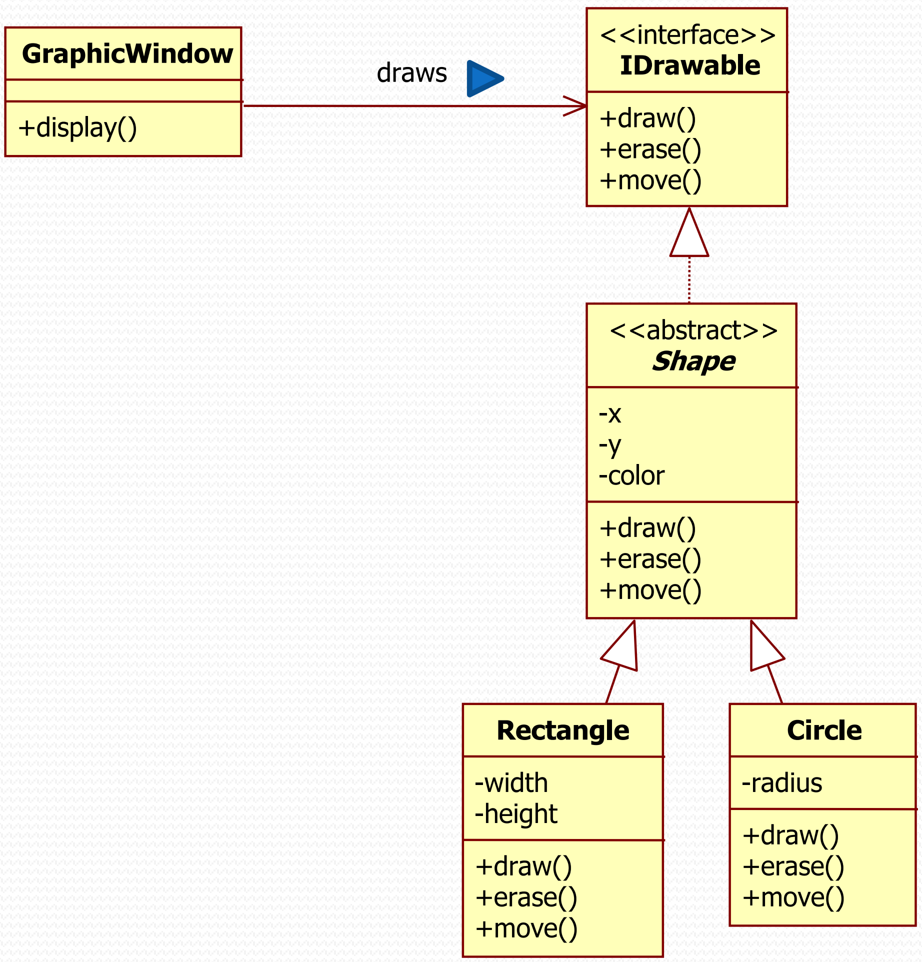
- Interfaces let you take greater advantage of polymorphism in your designs, which in turn helps you make your software more flexible.
- We can modify this class hierarchy so it supports display of images (like bitmaps and png's).

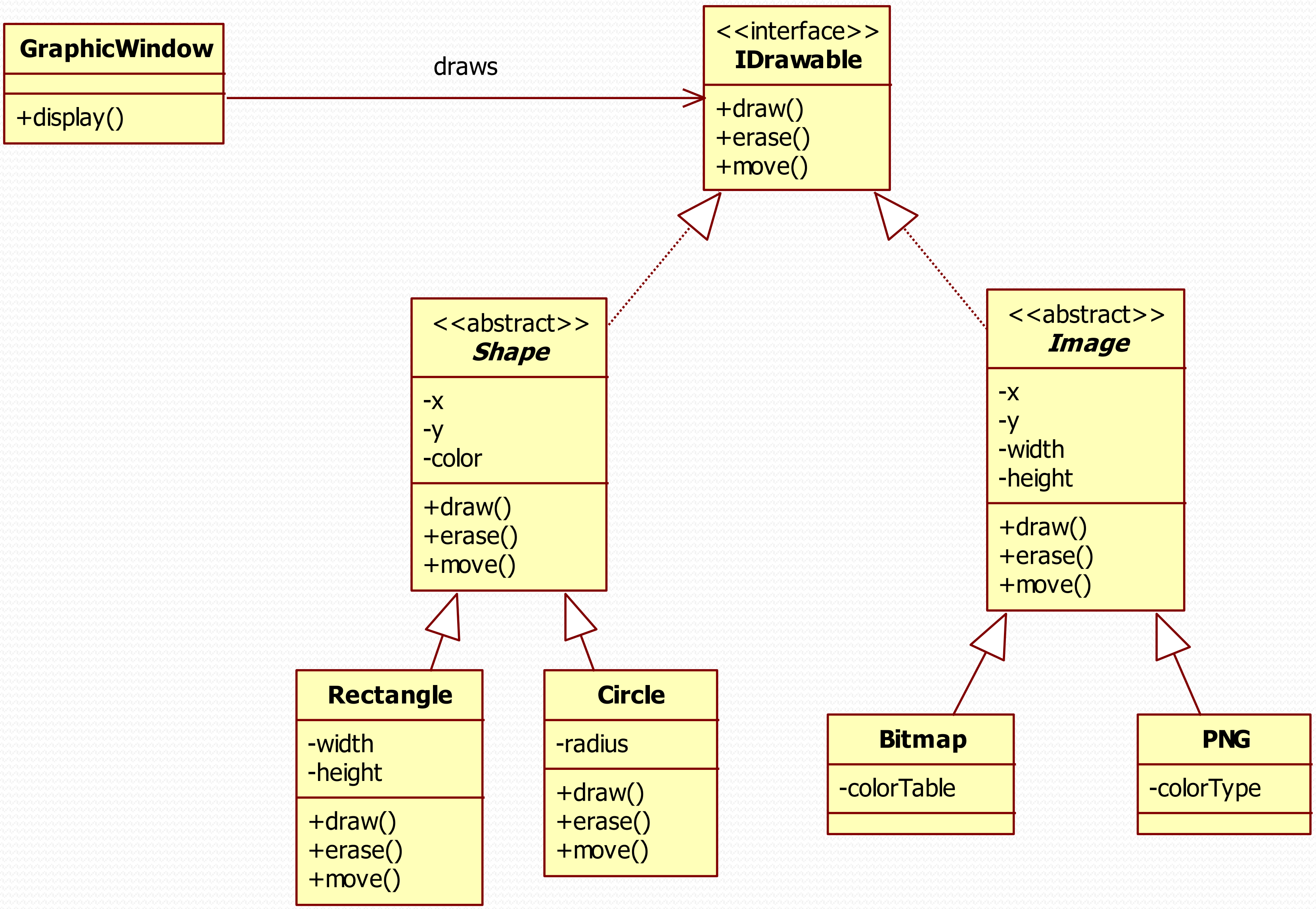
[Use an interface to create greater abstraction.]



(continued)

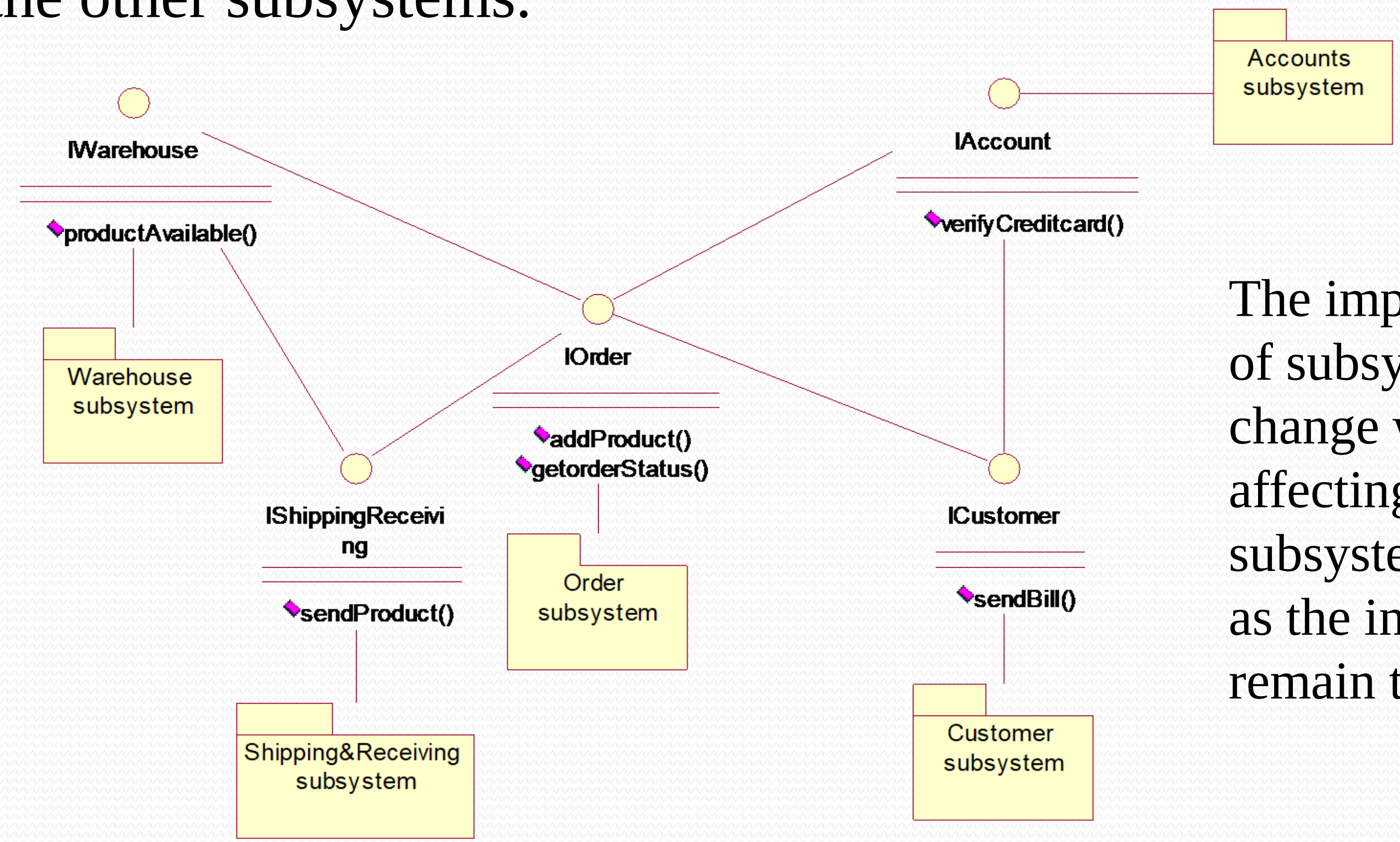
- With an interface, we can easily add a new class hierarchy




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.

Best Practices: When to Use Interfaces?

1. <u>Always</u> use interfaces for subsystem development
2. <u>Prefer</u> 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 *<u>read</u>* many more hours than it will take to *<u>write</u>* it. Make it as simple, elegant, and clear as possible.

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

The Evolving API Problem

Problem: 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

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 programming to interfaces, we generate more flexible code.

Greater abstraction holds the possibility of greater potential; this principle 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 focus more on general abstract parts (like types)
- With Java it is important to always “Program to Interface”.
- Use interfaces for ‘multiple’ inheritance, encapsulation, flexibility, and parallel development.

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.