MAHARISHI UNIVERSITY of MANAGEMENT

Engaging the Managing Intelligence of Nature

Computer Science Department

CS401 Modern Programming Practices (MPP)
Professor Renuka Mohanraj

1

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.
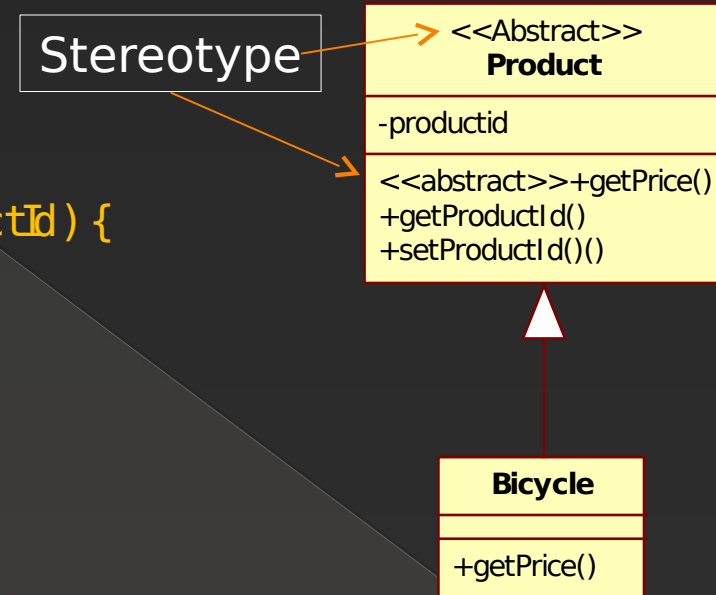
Abstract Classes and Methods

- 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

```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;
    }
}
```

Stereotype

<<Abstract>> **Product**
-productid
<>+getPrice()   +getProductId()   +setProductId()()

Bicycle
+getPrice()

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`.
  [Recall lesson3.lecture.polymorphism1]

- *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`
  [Recall: lesson3.lecture.polymorphism2]

Java Interfaces(Pre Java 8)

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.

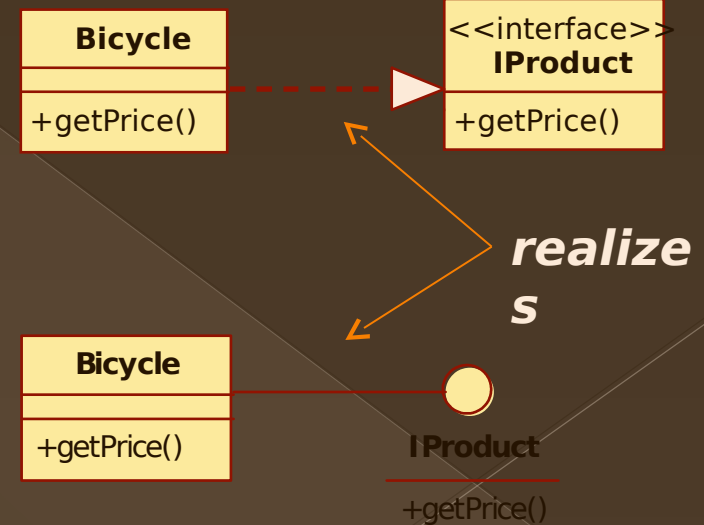- All methods in an interface are automatically public and abstract

Interface Example

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


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

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

UML has two different
notations for Interfaces

interface stereotype

Bicycle
+getPrice()

<<interface>> IProduct
+getPrice()

realizes

Bicycle
+getPrice()

IProduct
+getPrice()

Interface Example

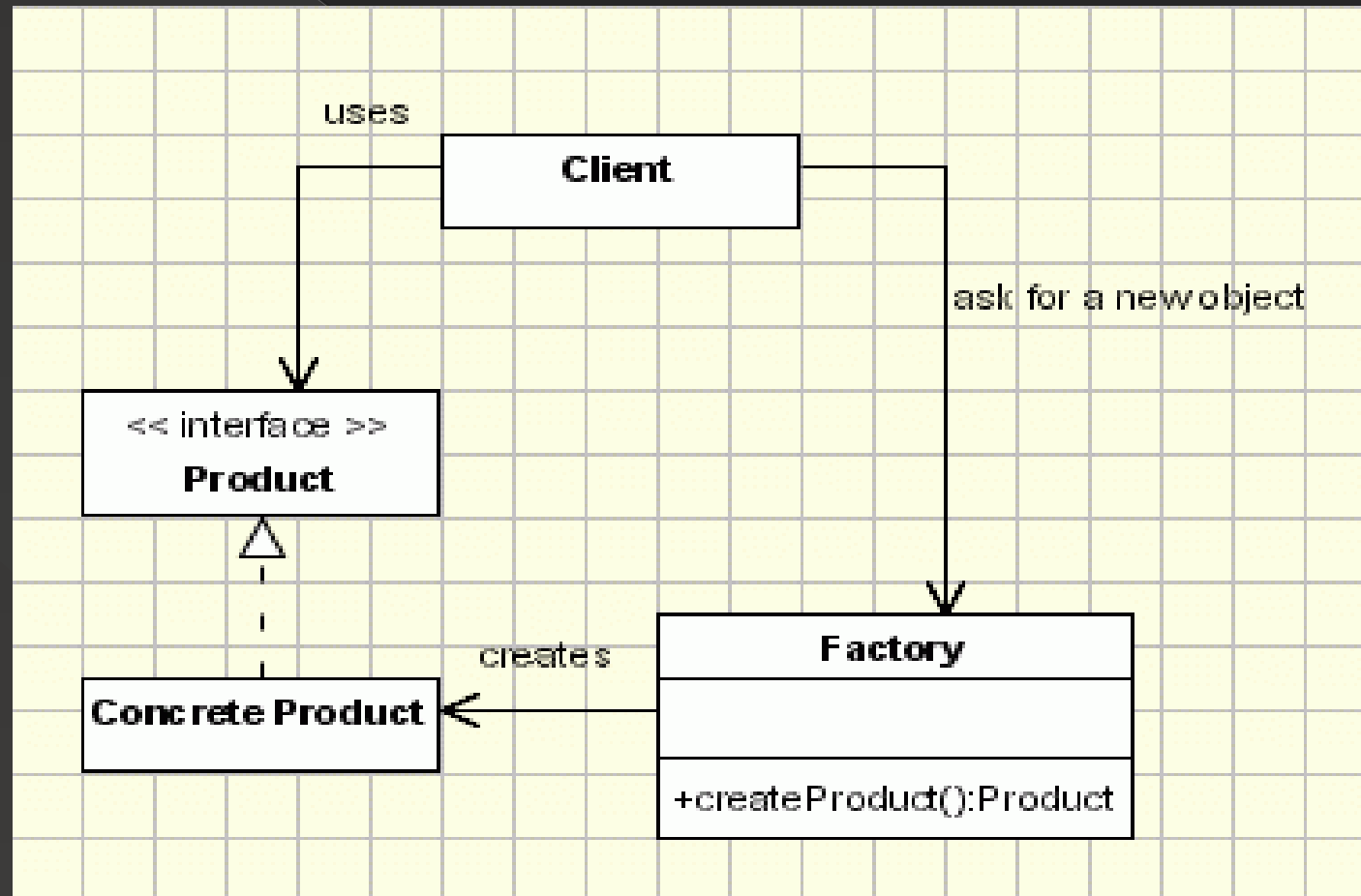```java
public interface IProduct {
    public abstract double getPrice();
}

public class Bicycle extends Vehicle
     implements IProduct {
   @Override
   public double getPrice() {
      return 230.45;
   }
}
```
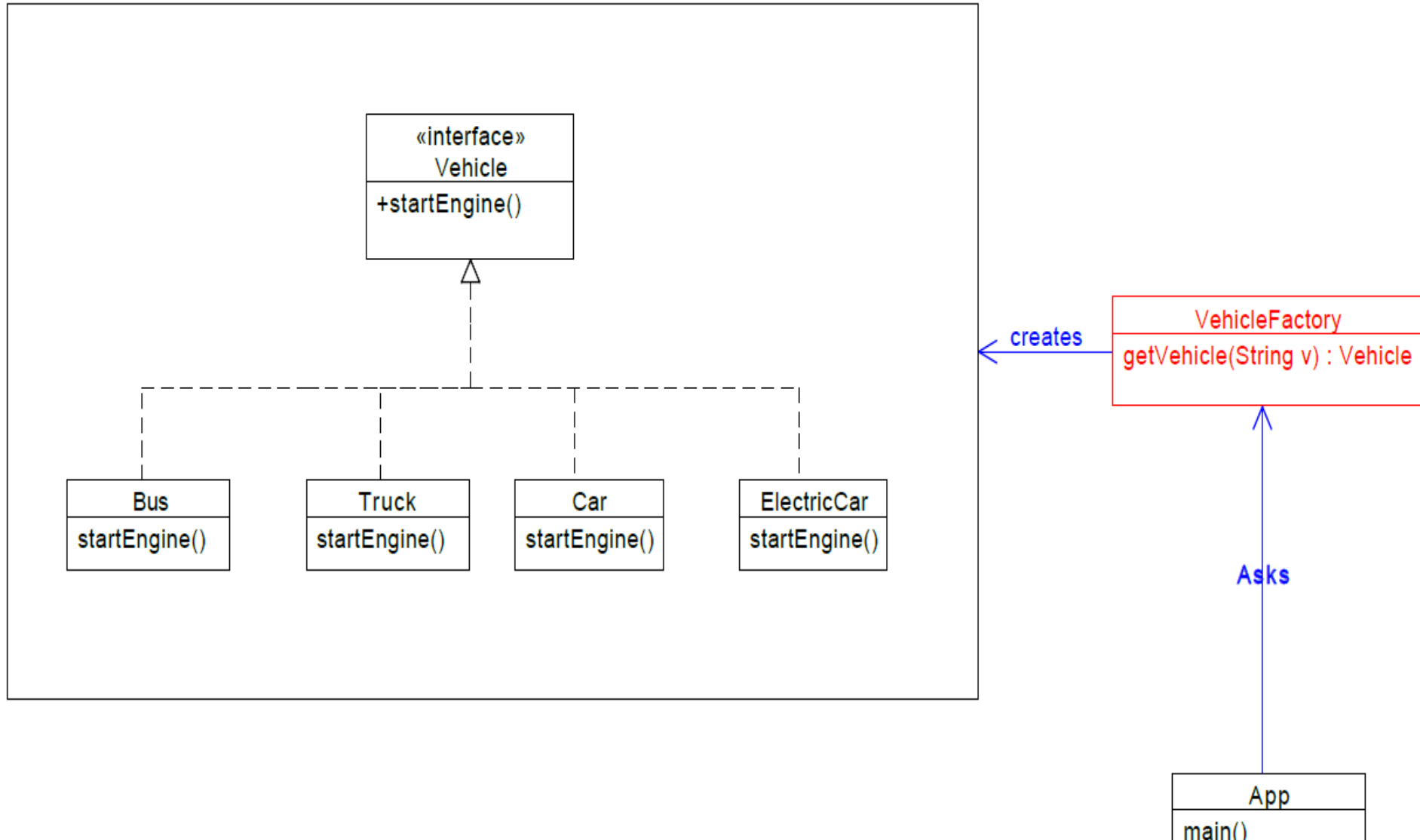
**One benefit: Bicycle can be treated as
a subclass of Vehicle at the same time as
it *implements* IProduct (as does every product in our system).**
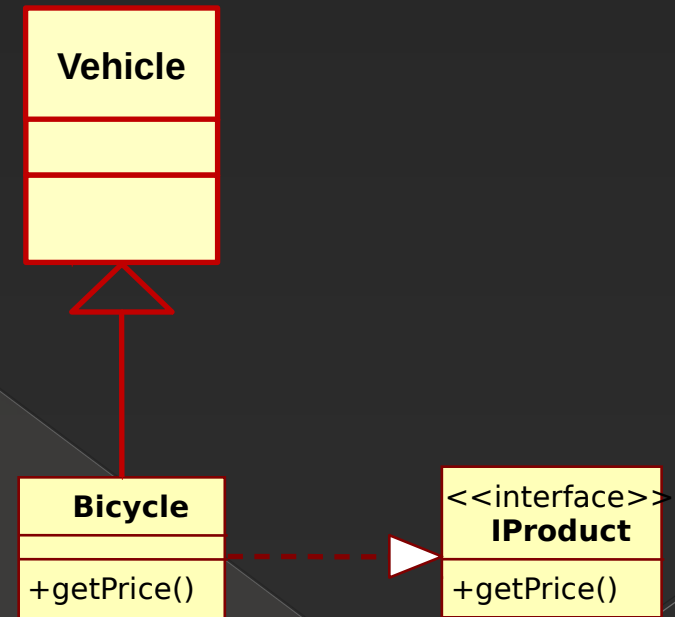
# Application of Interfaces: Object Creation Factory

# Factory Method Pattern

- The Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types.

- The Factory Method pattern encapsulates the functionality required to select and instantiate an appropriate class, inside a designated method referred to as a factory method.

- The Factory Method selects an appropriate class from a class hierarchy based on the application context and other influencing factors. It then instantiates the selected class and returns it as an instance of the parent class type.

- The advantage of this approach is that the application objects can make use of the factory method to get access to the appropriate class instance.

- This eliminates the need for an application object to deal with the varying class selection criteria.

# Factory Method Pattern

- This provides a pattern that describes the use of a factory class for constructing objects.  The methods on the factory need to return objects.

**Example :**

```
public class ConnectExample {
public static void main(String[] args) throws Exception {
URL oracle = new URL("http://www.oracle.com/");
URLConnection urlConn = oracle.openConnection(); // Factory method to return the instance of URLConnection class
BufferedReader in = new BufferedReader(new InputStreamReader(
urlConn.getInputStream()));
String inputLine;
while ((inputLine = in.readLine()) != null)
System.out.println(inputLine);
in.close();
}
}
```

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)

Advantages of Using Object-Creating Factory Methods

- Factory methods have a name – easier to understand what is being requested, and to distinguish between different kinds of invocations on an object. (Imagine a class that has 5 different constructors – how can you tell what each one does? A good example is the List creation methods in Collection.)

- Can control access to instances. (See the Student/GradeReport and DataAccess examples.)

- Solves the problem that a class can have only one constructor with a given signature. (See Triangle example)

- 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. (See RuleSet getRuleSet() in Rules Framework example – a subclass of RuleSet is returned)

- Simple example of Factory Pattern : lesson5.lecture.factorypattern.myexample.pizzas;

Simple Factory Methods

- Much of the benefit of having a separate 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

```java
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

<u>Triangle Example.</u> 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
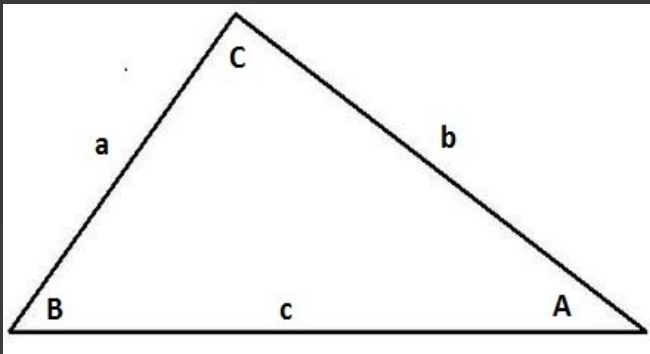


```
// produce compilation error
class Triangle {
      Triangle(double s1, double s2, double s3) {
            side1 = s1; side2 = s2; side3 =s3;
      }
      Triangle(double s1, double s2, double inclAngle){
            side1 = s1; side2 = s2; angle3 =inclAngle;
      }
   }
```

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 final

See lesson5.lecture.factorymethods6
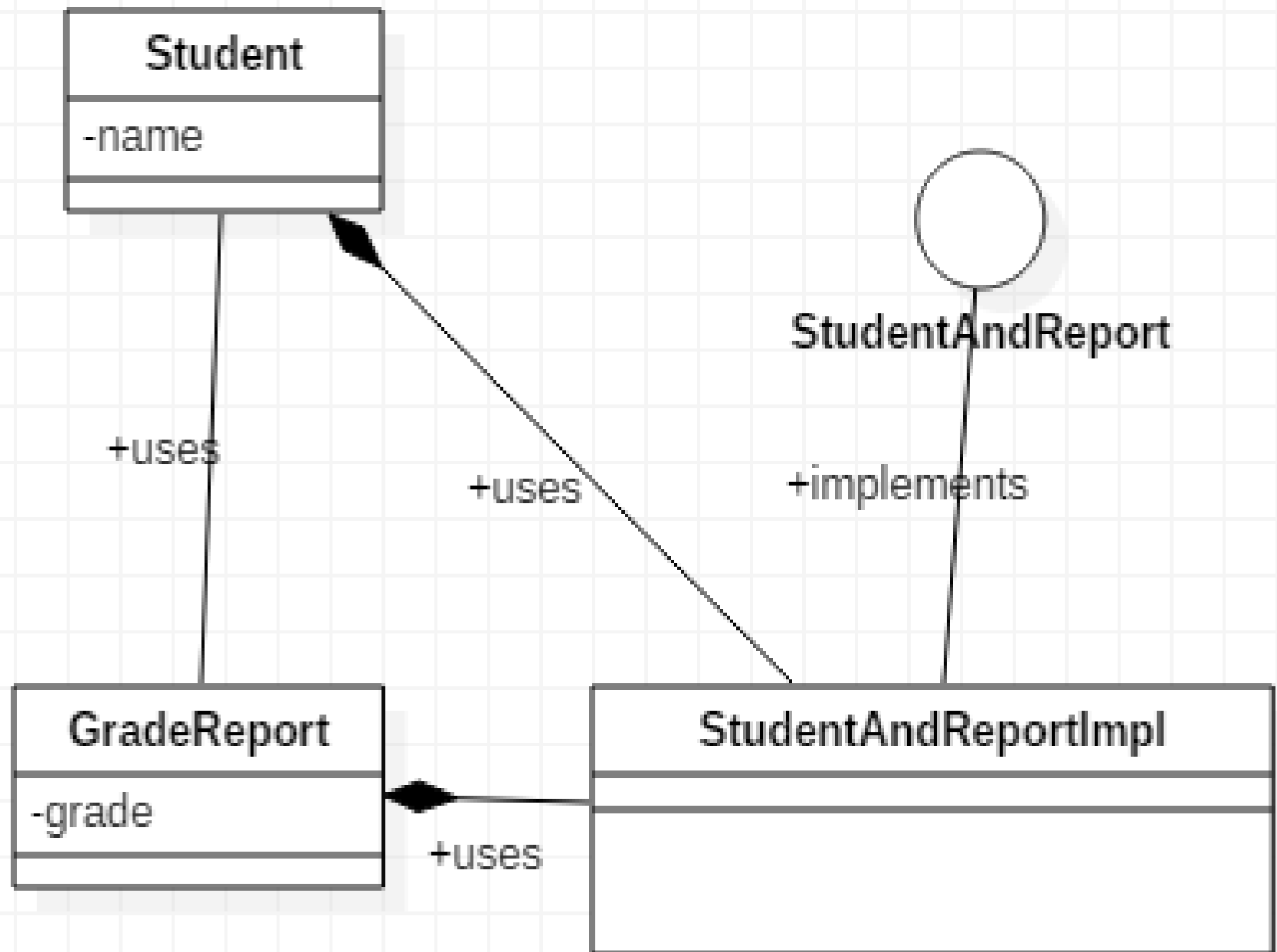
# Application: Controlling Access to Instances - continued

Data Access Example. We wish to provide restricted access to the database. Data access requests must come through a designated package. Special classes (called Daos) will be allowed to send in data to be saved and to make requests for data. To accomplish this we

1. Designate a dataaccess package
2. Provide a factory to produce instances of the DataAccessSystem, typed as an interface DataAccess
3. Make all setters and constructors of classes in the package have package level access
4. Require entity classes like Address or Person to make use of corresponding Dao classes, like AddressDao, PersonDao to communicate with the data access layer

See lesson5.lecture.factorymethods3.dataaccess

Parametrized Factory Methods

- Factory can produce different implementations of an interface based on input parameter. Example: Rules Framework see lesson5.lecture.factorymethods2
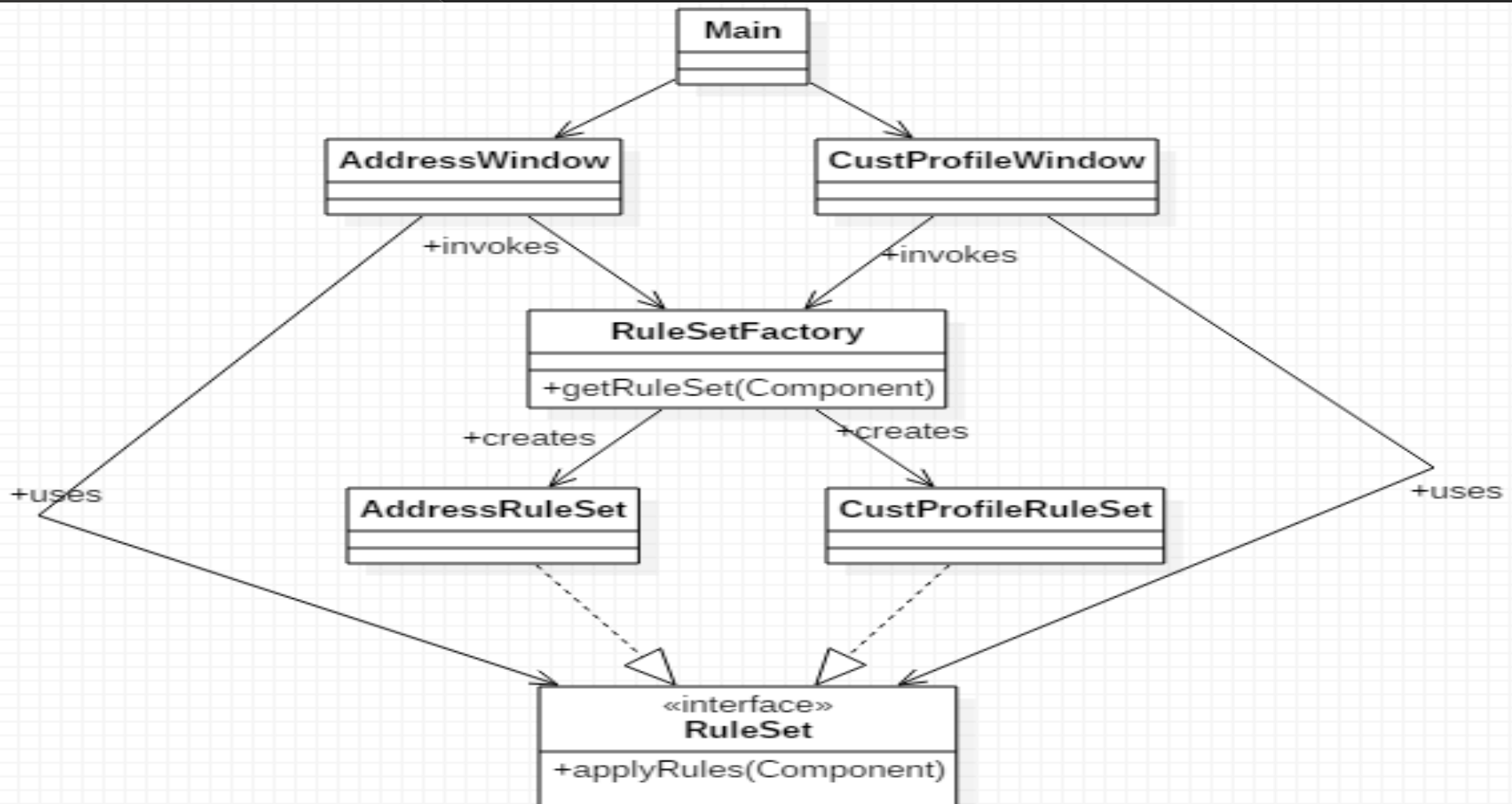
Issues When Using Factory Methods

- 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
    - openConnection [URLConnection example]3
    - valueOf [BigInteger.valueOf(long)]
    - of [LocalDate.of(year, month, day)]

23

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:
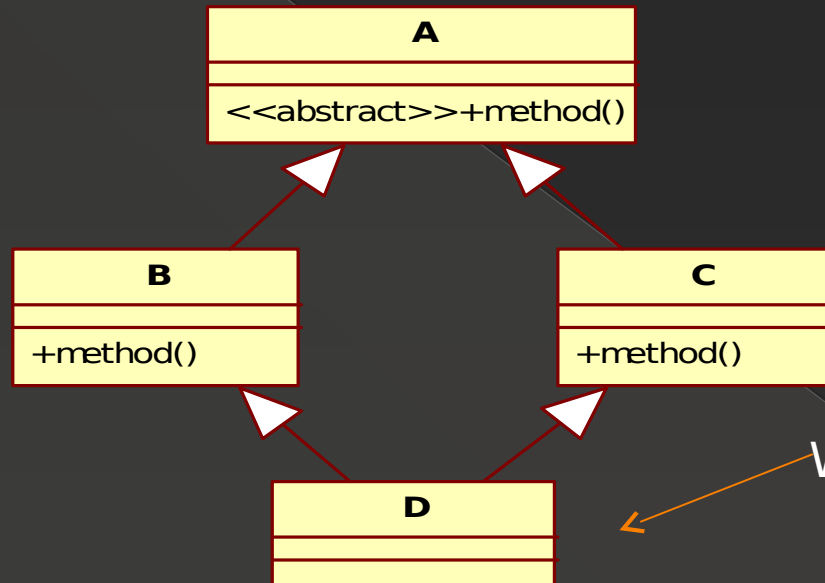 **public** List<Student> findStudents(String country)

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 among the types (interfaces) declared by the different classes

Java's Answer (pre - Java SE 8)

- *Implementation* can be 'inherited' / extended *only once*
- *Types* can be 'inherited'(one interface can extend another interface) / implemented *multiple interfaces in a class*
  - › No limit on the number of interfaces you can implement
  - › A single interface can extend other interfaces
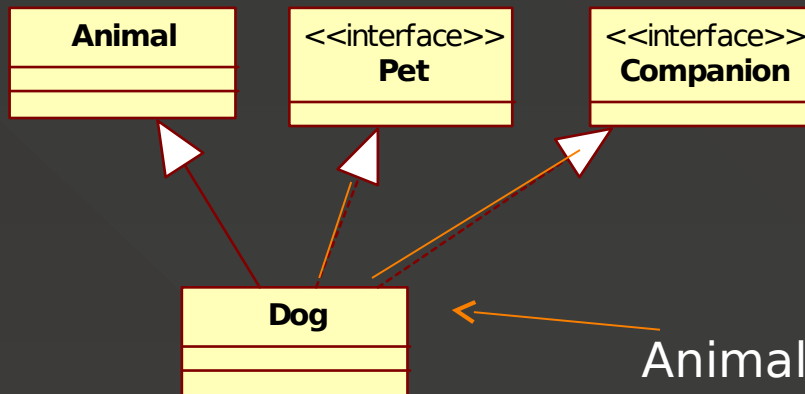
Dog 'is a':
Animal, Pet, and Companion

Interface vs. Abstract Class: Pre-Java 8

<<interface>>
List

+add()
+remove()
+clear()
+size()
+iterator()

<<Abstract>>
AbstractList

#modCount

+add()
+remove()
+clear()
+size()
+iterator()

ArrayList

-items

+indexOf()

Vector

-items
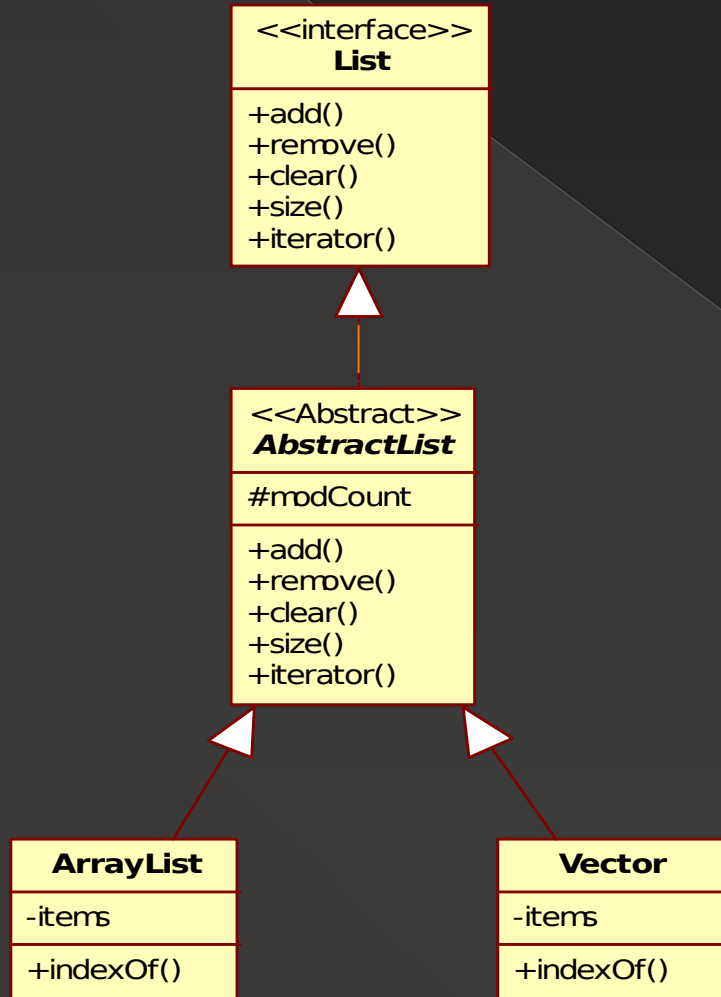
+indexOf()

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 - Interfaces are declared independently of classes, they are unaffected by changes to specific classes or to the class hierarchy as a whole.
- 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.

- In your small groups modify this class hierarchy so it supports display of images (like bitmaps and png's).

[Use an interface to create greater abstraction.]

GraphicWindow
+display()

draws →

<> *Shape*
-x -y -color
+draw() +erase() +move()

Rectangle
-width -height
+draw() +erase() +move()

Circle
-radius
+draw() +erase() +move()
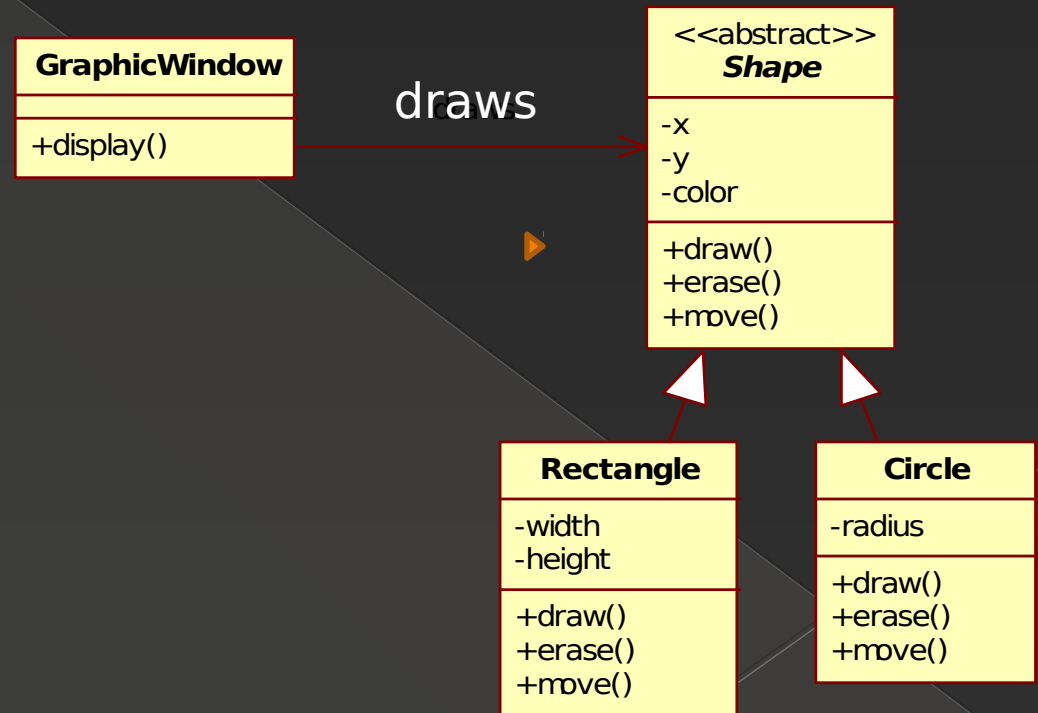
A Solution

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

GraphicWindow

+display()

draws

<<interface>>
IDrawable

+draw()
+erase()
+move()

<>
Shape

-x
-y
-color

+draw()
+erase()
+move()

Rectangle

-width
-height

+draw()
+erase()
+move()

Circle

-radius

+draw()
+erase()
+move()

GraphicWindow

+display()

draws

<<interface>>
IDrawable

+draw()
+erase()
+move()

<>
Shape

-x
-y
-color

+draw()
+erase()
+move()

<>
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.

Accounts subsystem

IWarehouse

◇productAvailable()

Warehouse subsystem

IAccount

◇verifyCreditcard()

IOrder

◇addProduct()
◇getorderStatus()
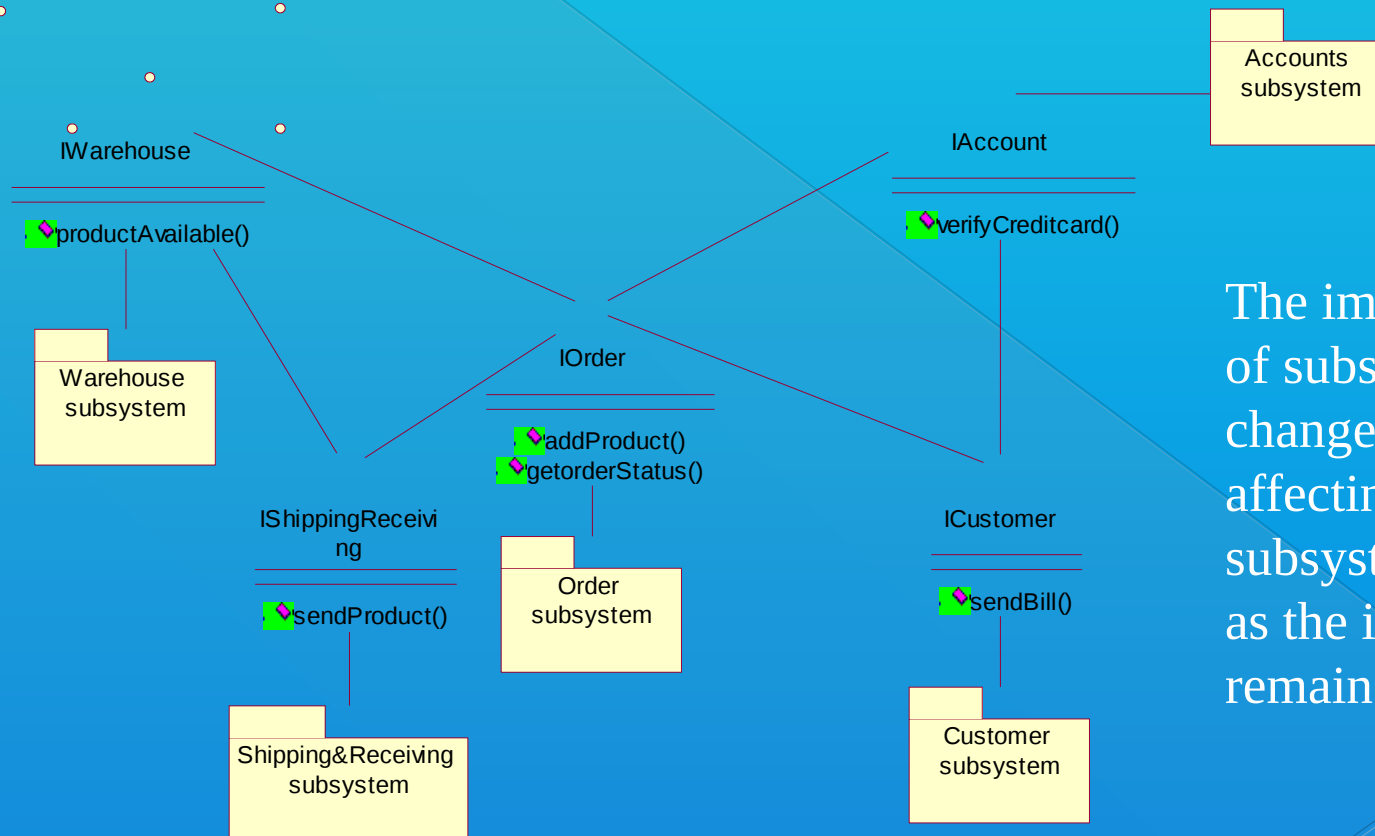
Order subsystem

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

IShippingReceiving

◇sendProduct()

Shipping&Receiving subsystem

ICustomer

◇sendBill()

Customer subsystem

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.

Program to Interface (P2I), rather than implementation.

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. Make code 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 pro-gramming 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.