UCLA PIC 20A
Java Programming

- **Instructor:** Ivo Dinov,
  Asst. Prof. In Statistics, Neurology and Program in Computing
- **Teaching Assistant:** Yon Seo Kim, PIC

University of California, Los Angeles, Summer 2002
http://www.stat.ucla.edu/~dinov/

Chapter 6 – Interfaces & Packages

- Creating Interfaces
- Making Packages

Interfaces (recall)

- An interface is a device that unrelated objects use to interact with each other. An object can implement multiple interfaces.
- Ex. An inventory program doesn't care what class of items it manages, as long as each item provides certain information, such as price and tracking number, quantity, etc.
- Instead of forcing class relationships on otherwise unrelated items, the **inventory program sets up a protocol of communication** – a set of constant and method definitions contained within an interface. The **inventory interface would define, but not implement**, methods that set and get the retail price, assign a tracking number, and so on.

Interfaces vs. Abstract Classes

- An interface is simply a list of unimplemented, and therefore abstract, methods – **How an interface differs from an abstract class?**
  - An interface cannot implement any methods, whereas an abstract class can.
  - A class can implement many interfaces but can have only one superclass.
  - An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

Interface Example – StockMonitor
a class using the StockWatcher interface

- Watch stock prices coming over a data stream. This class allows other classes to register to be notified when the value of a particular stock changes.

```
public class StockMonitor  {
   public void watchStock ( StockWatcher watcher,
      String tickerSymbol, double delta)  //register for notification
        {  ...  }
}
```

- The StockWatcher is an interface that declares one method: valueChanged(). An object that wants to be notified of stock changes must be an instance of a class that implements this interface and thus implements the valueChanged method.
- The other arguments provide the symbol of the stock to watch and the amount of change that the watcher considers interesting enough to be notified of.

Interface StockWatcher

- When the StockMonitor class detects an interesting change, it calls the valueChanged method of the watcher.

```
public interface StockWatcher Extends <<List Of Interf's>>
      final String sunTicker ="SUNW";
      final String oracleTicker ="ORCL";
      final String ciscoTicker ="CSCO";
      void valueChanged ( String tickerSymbol,
            double newValue);
}
```

An interface declaration can have a list of superInterf's.
An interface can extend multiple other interfaces.

Interface Example - StockApplet

- Applet that implements the StockWatcher interface:

```
public class StockApplet extends Applet
        implements StockWatcher  {
   public void valueChanged (String
                        tickerSymbol,   double newValue)
   {    if (tickerSymbol.equals(sunTicker))
         {      ...      }
        else if (tickerSymbol.equals(oracleTicker))
         {      ...      }
        else if (tickerSymbol.equals(ciscoTicker))
         {      ...      }
}}
```


Interface Example - StockApplet

- Applet that implements the StockWatcher interface:

```
public class StockApplet extends Applet
        implements StockWatcher  {
   public void valueChanged (String
                tickerSymbol,   double newValue)
   {    if (tickerSymbol.equals( sunTicker ))
         {      ...      }
        ....
}}
```

Note that this class refers to each constant defined in StockWatcher, **sunTicker**, etc. Classes that implement an interface inherit the constants defined within that interface. Other classes use an interface's constants by: **StockWatcher.sunTicker**

Interface Example – StockMonitor
a class using the StockWatcher interface

- This class allows other classes to register to be notified when the value of a particular stock changes.

```
public class StockMonitor  {
   public void watchStock ( StockWatcher watcher,
      String tickerSymbol, double delta)  //register for notification
       {  ...  }
}
```

Only an **instance of a class** that implements the interface can be assigned to a reference variable whose type is an interface name. So only instances of a class that implements the StockWatcher interface can register to be notified of stock value changes.

Interface Can NOT Expand …

- To add some functionality to StockWatcher. **Ex.** add a method that reports the current stock price.
- If you make this change later, all classes that implement the old StockWatcher interface will break!

```
public interface StockWatcher {
        ....
   void currentValue ( String tickerSymbol,
                double newValue);
}
```

Try to anticipate all uses for your interface up front and specify it completely from the beginning. Otherwise, you need to create a **StockWatcher** sub-interface called **StockTracker** that declares the new method

Packages

A *package* is a collection, a bundle, of related classes and interfaces providing access protection and namespace management.

Ex: If you write a group of classes that represent a collection of graphic objects, such as circles, rectangles, lines, and points. You also write an interface, Draggable , that classes implement if they can be dragged with the mouse by the user:

Packages – Example

```
public abstract class Graphic
{       ...                    //in the Graphic.java file
}
public class Circle extends Graphic implements Draggable
{       ...                    //in the Circle.java file
}
public class Rectangle extends Graphic implements Draggable
{       ...                    //in the Rectangle.java file
}
public interface Draggable
{       ...                    //in the Draggable.java file
}
```

Why bundle classes in Packages?

- You and other programmers can easily determine that these classes and interfaces are related.
- You and other programmers know where to find classes and interfaces that provide **graphics-related functions**.
- The names of your classes won't conflict with class names in other packages, because the package creates a new namespace.
- You can allow classes within the package to have unrestricted access to one another yet still restrict access for classes outside the package.

Creating Packages

```
package graphics;

public class Circle extends
  Graphic implements Draggable
{
        ...
}
```


Creating Packages

```
package graphics;   <----
public class Circle extends Graphic
  implements Draggable
{        ...
}
----------Rectangle.java----------
package graphics;   <----
public class Rectangle extends Graphic
  implements Draggable
{        ...
}
```


Scope of Packages

- The scope of the package statement is the entire source file, so all classes and interfaces defined in Circle.java and Rectangle.java are also members of the graphics package.
- If you put multiple classes in a single source file, only one may be public, and it must share the name of the source file's base name. Only public package members are accessible from outside the package.

Naming Packages

- **By Convention:** Companies use their reversed Internet domain name in their package names, like this: com.company.package . Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name, for example, com.company.region.package.
- My packages look like:
  - edu.ucla.stat.dinov.MyNewPackage
  - edu.ucla.loni.LONI_Viz
  - edu.ucla.loni.BrainGraph

Using Packages

- Only public package members are accessible outside the package in which they are defined.
- To use a public package member from outside its package, you must do either of:
  - Refer to the member by its long (qualified) name:
    edu.ucla.stat.Viz.VizDisplayPanel()
  - Import the package member
    - import edu.ucla.stat.Viz.VizDisplayPanel;
      …
      VizDisplayPanel myPanel = new VizDisplayPanel();
  - Import the member's entire package
    - import edu.ucla.stat.Viz.*;
      …
      VizDisplayPanel myPanel = new VizDisplayPanel();

Using Packages

- Caution! When referring to a class be ware that a class with the same name can apper in several of the packages you import ….
  - Then an implicit localization of the required packages is necessary (compile error will be reported for you).

```
import edu.ucla.stat.Viz.Rectangle;
import java.Graphics.*;
Rectangle myREctangle = new Rectangle();
```

  - Which blue-print for a Rectangle object are you using?

Example of Using Packages

- C:\Ivo.dir\LONI_Viz\LONI_Viz_MAP_demo
  - Directory organization:
  - Package delimiters inside *.java files
  - Use of outside classes (external packages)
  - Compilation (make makefile; ant build.xml)
  - Debugging
  - Running (run.bat)