CSc 133 Lecture Notes

3 - OOP Concepts

Computer Science Department

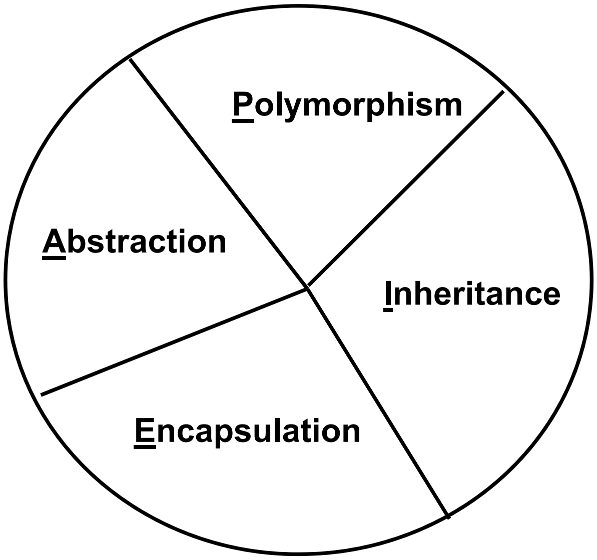
California State University, Sacramento

Overview

- The OOP “A PIE”
- Abstraction
- Encapsulation: Bundling, Information Hiding, Implementing Encapsulation, Accessors & Visibility
- UML Class Diagrams
- Class Associations: Aggregation, Composition, Dependency, Implementing Associations

The OOP "A Pie"

- Four distinct OOP Concepts make "A PIE"



Abstraction

- Identification of the minimum essential characteristics of an entity
- Essential for specifying (and simplifying) large, complex systems
- OOP supports:
    - *Procedural* abstraction
    - *Data* abstraction

(clients do not need to know about implementation details of identified procedures and data types, e.g. Stack)

Encapsulation

In Java encapsulation is done via classes.

"Bundling"

- Collecting together the data and procedures associated with an abstraction
- Class has fields (data) and methods (procedures)

"Information Hiding"

- Prevents certain aspects of the abstraction from being accessible to its clients
- Visibility modifiers: public vs. protected vs. private
- Correct way: keep all data **private** and use accessors (Getters/Selectors vs. Setters/Mutators*)*

Implementing Encapsulation

```
public class Point {

  private double x, y;
  private int moveCount = 0;

  public Point (double xVal, double yVal) {
    x = xVal;   y = yVal;
  }

  public void move (double dX, double dY) {
    x = x + dX;
    y = y + dY;
    incrementMoveCount();
  }

  private void incrementMoveCount() {
    moveCount ++ ;
  }
}
```

bundled, hidden data

bundled, exposed operations

bundled, hidden operations

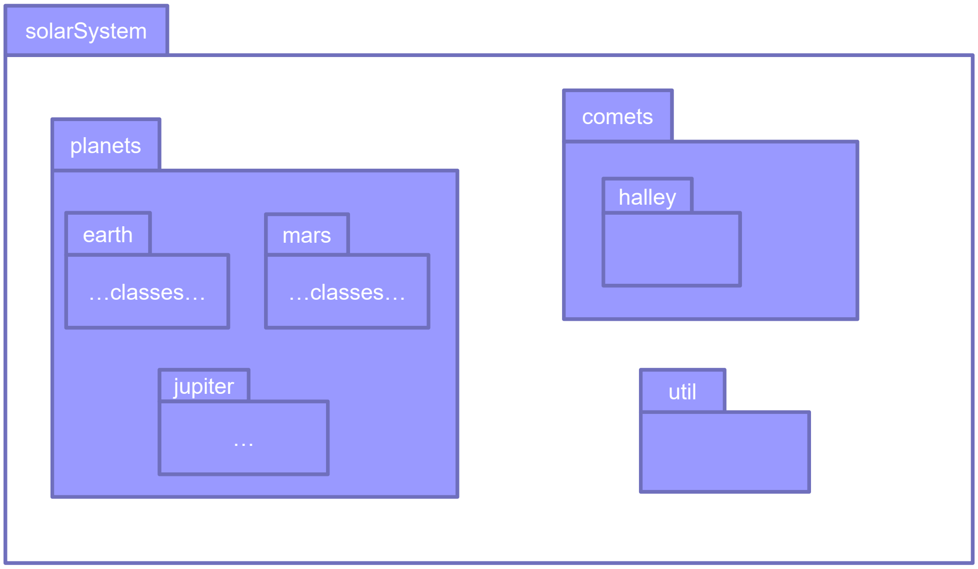
Access (Visibility) Modifiers

	Modifier	Access Allowed By			
		Class	Package	Subclass	World
Java:	public	Y	Y	Y	Y
	protected	Y	Y	Y	N
	<none>	Y	Y*	N	N
	private	Y	N	N	N
C++:	public	Y	<n/a>	Y	Y
	protected	Y	<n/a>	Y	N
	<none>	Y	<n/a>*	N	N
	private	Y	<n/a>	N	N

*In C++, omitting any visibility specifier is the same as declaring it *private*, whereas in Java this allows *"package access"*

Java Packages

- Used to group together classes belonging to the same category or providing similar functionality



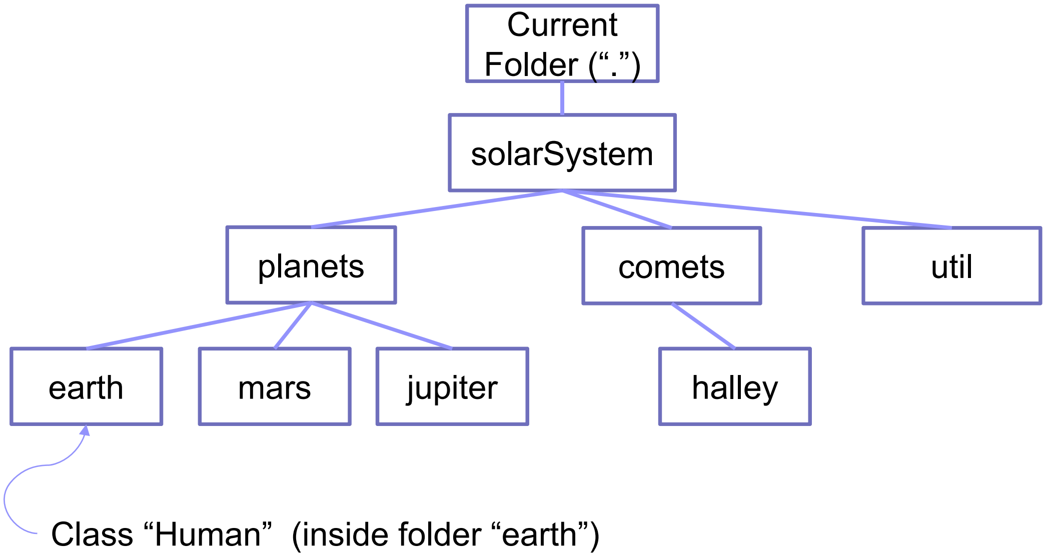
Java Packages (cont.)

- Packages are *named* using the concatenation of the enclosing package names
- Types (e.g. classes) must declare what package they belong to
  - Otherwise they are placed in the "default" (unnamed) package
- Package names become part of the class name; the following class has the full name
  solarSystem.planets.earth.Human

```
package solarSystem.planets.earth ;

//a class defining species originating on Earth
public class Human {

   // class declarations and methods here...
}
```

Packages and Folders

- Classes reside in (are compiled into) *folder hierarchies* which match the package name structure:



Abstraction example: Color

- We see colors at the visible portion of the electromagnetic spectrum.
  - Color can be represented by its wavelength.
  - Better approach: use abstraction and represent them with a color model (RGB, CMYK).
- Three axes: Red, Green, Blue
- Distance along axis = intensity (0 to 255)
- Locations within cube = different colors
  - Values of equal RGB intensity are grey

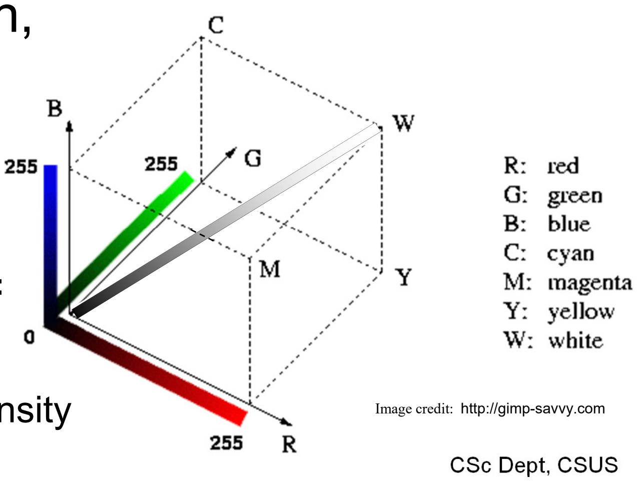


Image credit: http://gimp-savvy.com

Example: CN1 `ColorUtil` Class

- An *encapsulated abstraction*
- Uses “RGB color model”
- **`ColorUtil`** is in:
  - **`com.codename1.charts.util`**
- Has static functions to set color and get color, and static *constants* for many colors:

```
import com.codename1.charts.util.ColorUtil;
int myColor = ColorUtil.rgb(255 , 255, 255);  //set color to white
myColor = ColorUtil.rgb(255, 0, 0);            //change the color to red
myColor = ColorUtil.BLACK;                     //same as ColorUtil.rgb(0 , 0, 0)
myColor = ColorUtil.GREEN;                     //same as ColorUtil.rgb(0 , 255, 0)
System.out.println ("myColor = " + "[" + ColorUtil.red(myColor) + "," +
                                         ColorUtil.green(myColor) + "," +
                                         ColorUtil.blue(myColor) + "]";
                                               //prints: myColor = [0, 255, 0]
```

Breaking Encapsulations

- The wrong way, with public data:

```
public class Point {
  public double x, y;        ←        BAD!

  public Point () {
    x = 0.0 ;   y = 0.0 ;
  }

  // other methods here...
 }
```

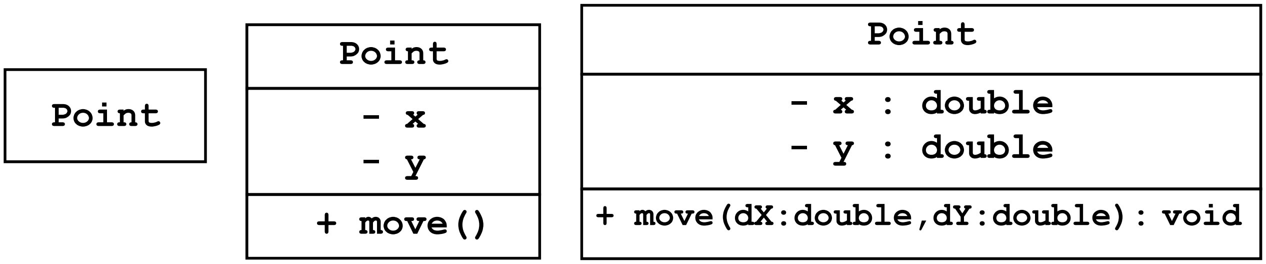
Breaking Encapsulations (cont.)

- The correct way, with “Accessors”:

```
public class Point {

   private double x, y ;      ←        Note

   public Point () {
     x = 0.0 ;    y = 0.0 ;
   }

   public double getX() {
     return x ;
   }

   public double getY() {
     return y ;
   }

   public void setX (double newX) {
     x = newX ;
   }

   public void setY (double newY) {
     y = newY ;
   }

   // etc.
}
```

UML “Class Diagrams”

- Unified Modeling Language defines a “graphical notation” for classes
  - UML for the “**Point**” class:



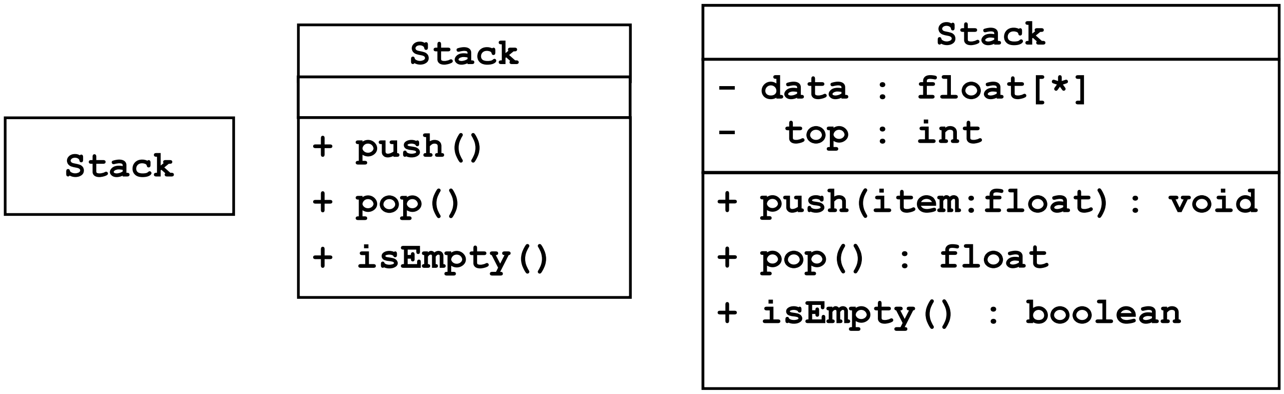
```
Point
```

```
Point
- x
- y
+ move()
```

```
Point
- x : double
- y : double
+ move(dX:double,dY:double): void
```

UML “Class Diagrams” (cont.)

- UML for the “**Stack**” class:



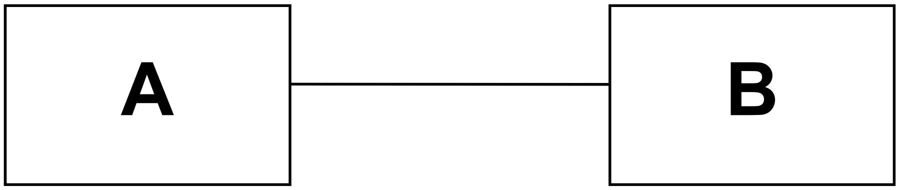
```
Stack
```

```
Stack

+ push()
+ pop()
+ isEmpty()
```

```
Stack
- data : float[*]
-  top : int
+ push(item:float) : void
+ pop() : float
+ isEmpty() : boolean
```

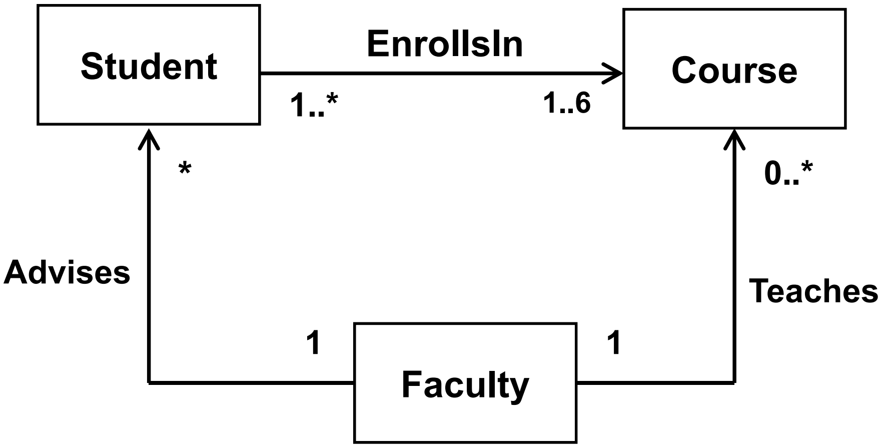
Associations

- Definition: An *association* exists between two classes A and B if instances can send or receive messages (make method calls) between each other.



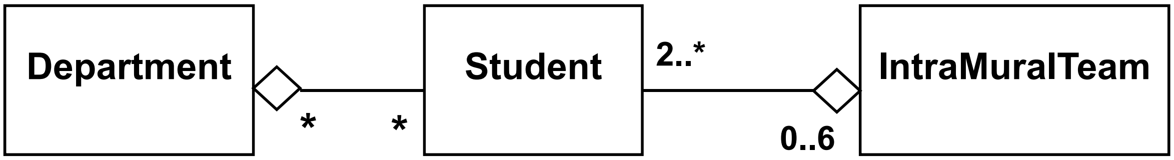
Associations (cont.)

- Associations can have properties:
  - Cardinality
  - Direction
  - Label (name)



Special Kinds Of Associations

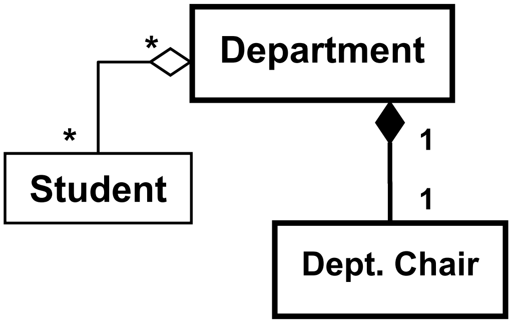
- **Aggregation**
  - Represents ***"has-a"*** or ***"is-Part-Of"***



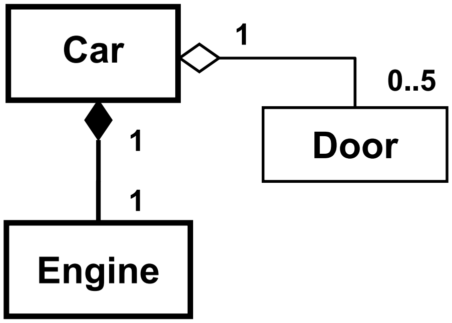
- **An IntraMuralTeam is an aggregate of *(has)* 2 or more Students**
- **A Student *is-a-part-of* at most six Teams**
- **A Department has any number of Students**
- **A Student can belong to any number of Departments (e.g. double major)**

Special Kinds Of Associations (cont.)

- Composition : a *special type* of *aggregation*
- Two forms:
  - "exclusive ownership" (without whole, the part can't exist)
  - "required ownership" (without part, the whole can't exist)



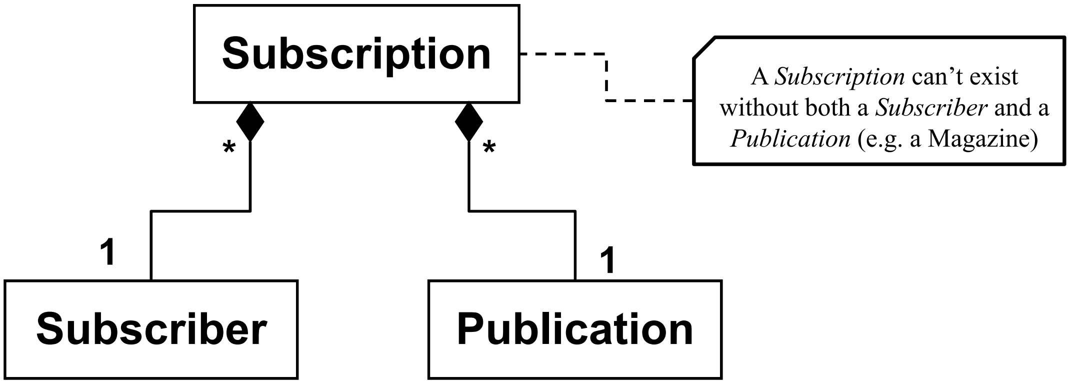
Exclusive ownership



Required ownership

Special Kinds Of Associations (cont.)

- Composition (another example)

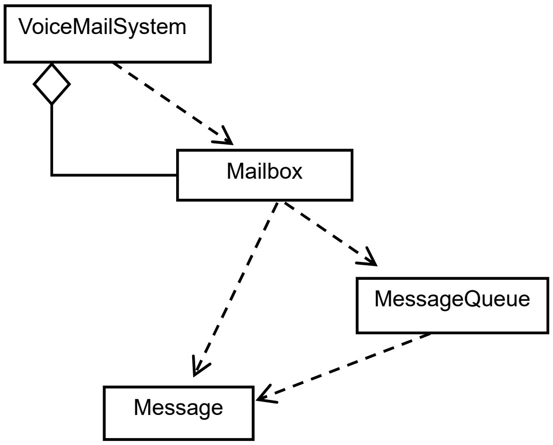


A *Subscription* can't exist without both a *Subscriber* and a *Publication* (e.g. a Magazine)

Special Kinds Of Associations (cont.)

- Dependency
  - **Represents "<u>uses</u>" (or "knows about")**

- **Indicates *coupling* between classes**
- **Desireable to *minimize* dependencies**
- **Other relationships (e.g. aggregation, inheritance) *imply dependency***

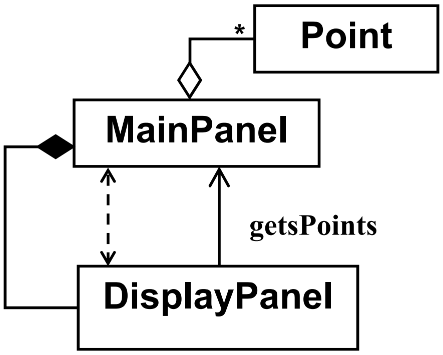


Implementing Associations

- Associations can be unary or binary
- Links are stored in private attributes

```
public class MainPanel {
  private DisplayPanel myDisPanel = new DisplayPanel (this) ;
  ...
}
```

```
public class DisplayPanel {
   private MainPanel myMainPanel ;

   //constructor receives and saves reference
   public DisplayPanel(MainPanel theMainPanel){
     myMainPanel = theMainPanel ;
   }
   ...
}
```



Implementing Associations (cont.)

```
/**This class defines a "MainPanel" with the following Class Associations:
 *  -- an aggregation of Points  -- a composition of a DisplayPanel.
 */
public class MainPanel {

    private ArrayList<Point> myPoints ;     //my Point aggregation
    private DisplayPanel myDisplayPanel;    //my DisplayPanel composition

    /** Construct a MainPanel containing a DisplayPanel and an
     *  (initially empty) aggregation of Points. */
    public MainPanel () {
        myDisplayPanel = new DisplayPanel(this);
    }

    /**Sets my aggregation of Points to the specified collection */
    public void setPoints(ArrayList<Point> p) { myPoints = p; }

    /** Return my aggregation of Points */
    public ArrayList<Point> getPoints() { return myPoints ; }

    /**Add a point to my aggregation of Points*/
    public void addPoint(Point p) {
        //first insure the aggregation is defined
        if (myPoints == null) {
            myPoints = new ArrayList<Point>();
        }
        myPoints.add(p);
    }
}
```

Implementing Associations (cont.)

```
/** This class defines a display panel which has a linkage to a main panel and
 *  provides a mechanism to display the main panel’s points.
 */
public class DisplayPanel {

    private MainPanel myMainPanel;

    public DisplayPanel(MainPanel m) {

        //establish linkage to my MainPanel
        myMainPanel = m ;
    }

    /**Display the Points in the MainPanel's aggregation */
    public void showPoints() {
        //get the points from the MainPanel
        ArrayList<Point> thePoints =  myMainPanel.getPoints();

        //display the points
        for (Point p : thePoints) {
            System.out.println("Point:" + p);
        }
    }
}
```