CSc 133 Lecture Notes

3 - OOP Concepts

Computer Science Department

California State University, Sacramento

Announcement

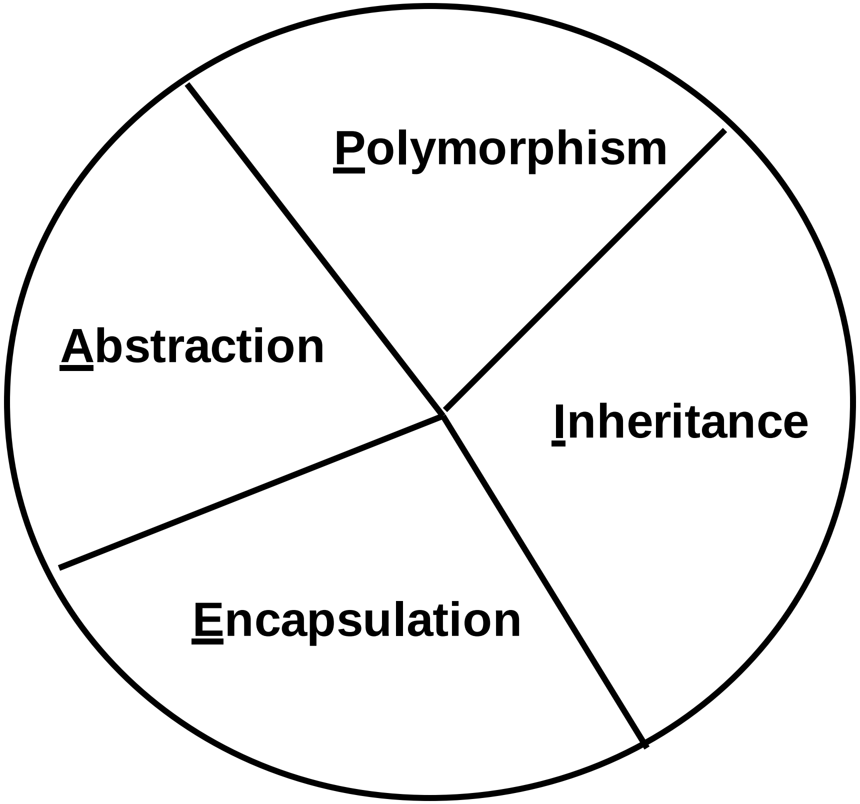
I am moving to room RVR 3006 (new office)

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

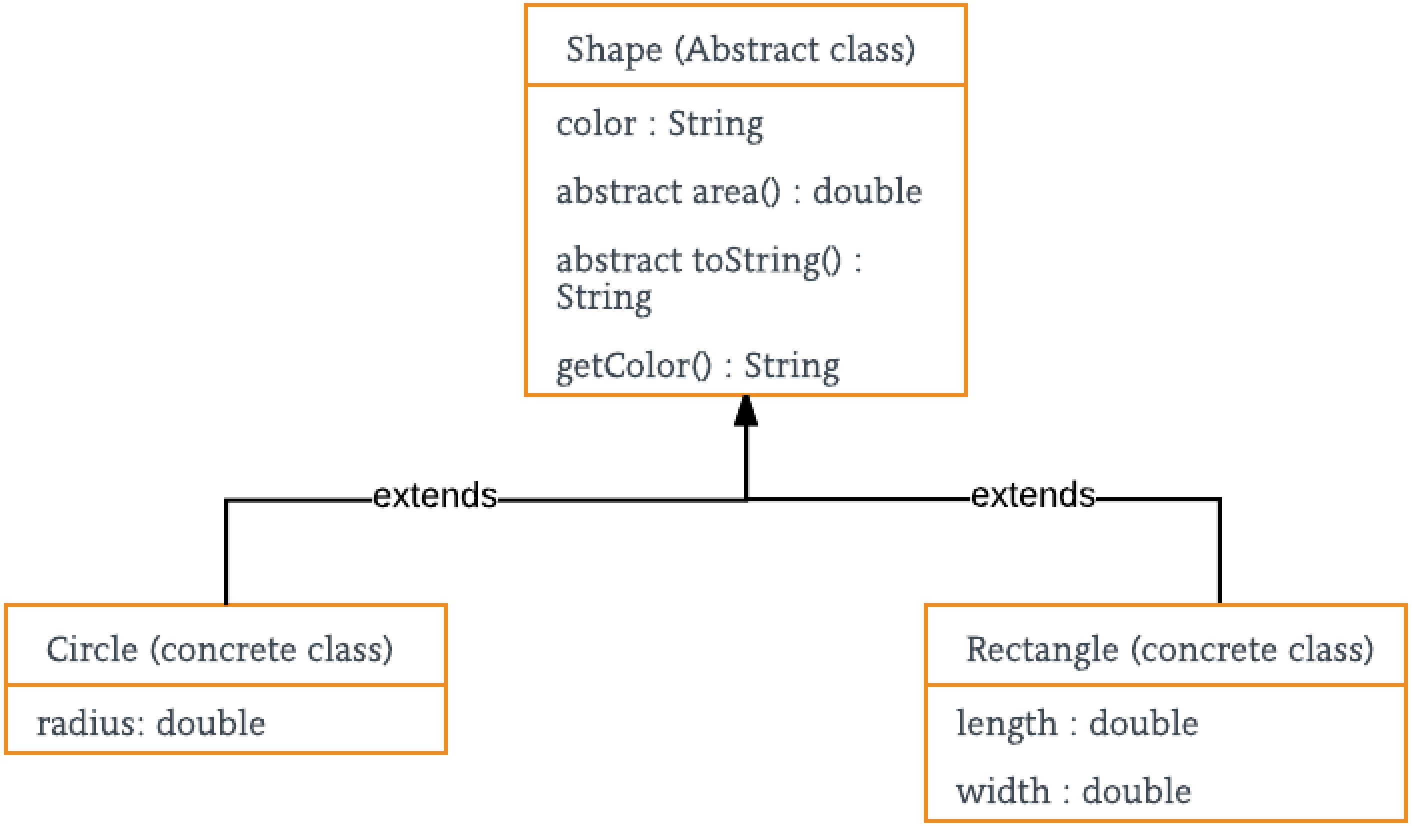
Abstraction is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics.

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)

Abstraction Example



Shape (Abstract class)

color : String

abstract area() : double

abstract toString() : String

getColor() : String

Circle (concrete class)

radius: double

Rectangle (concrete class)

length : double

width : double

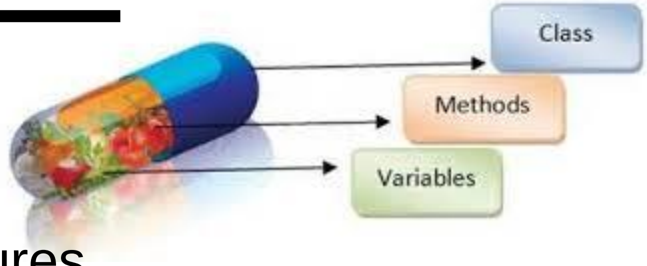
The base type is “shape” and each shape has a color, size and so on. From this, specific types of shapes are derived(inherited)-circle, square, triangle and so on – each of which may have additional characteristics and behaviors.

Source: Geeks for Geeks Abstraction in Java

Encapsulation

Encapsulation refers to the bundling of data with the methods that operate on that data.

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

Questions: (1) Name the Constructor ?
(2) Usage ?

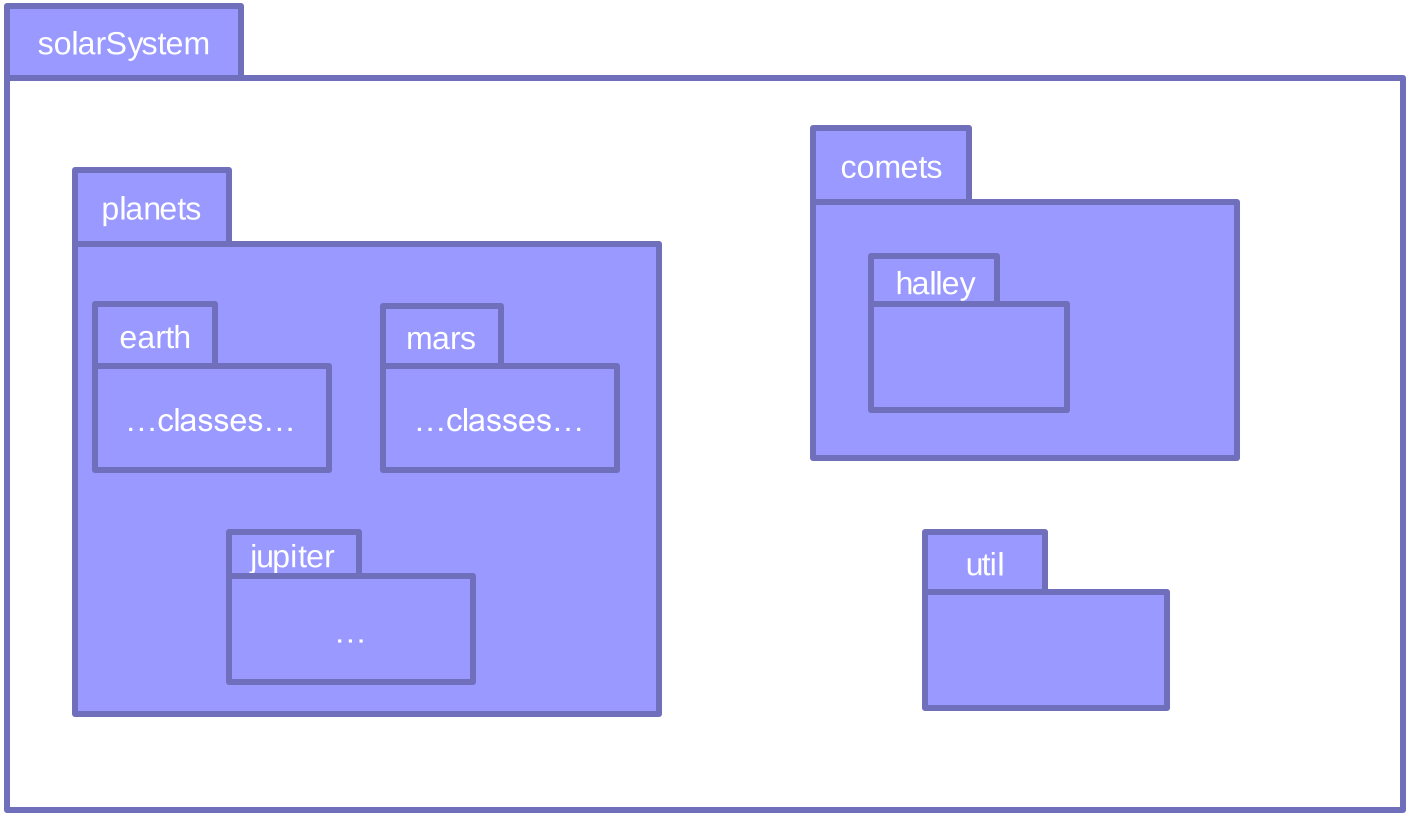
Access (Visibility) Modifiers

	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.)

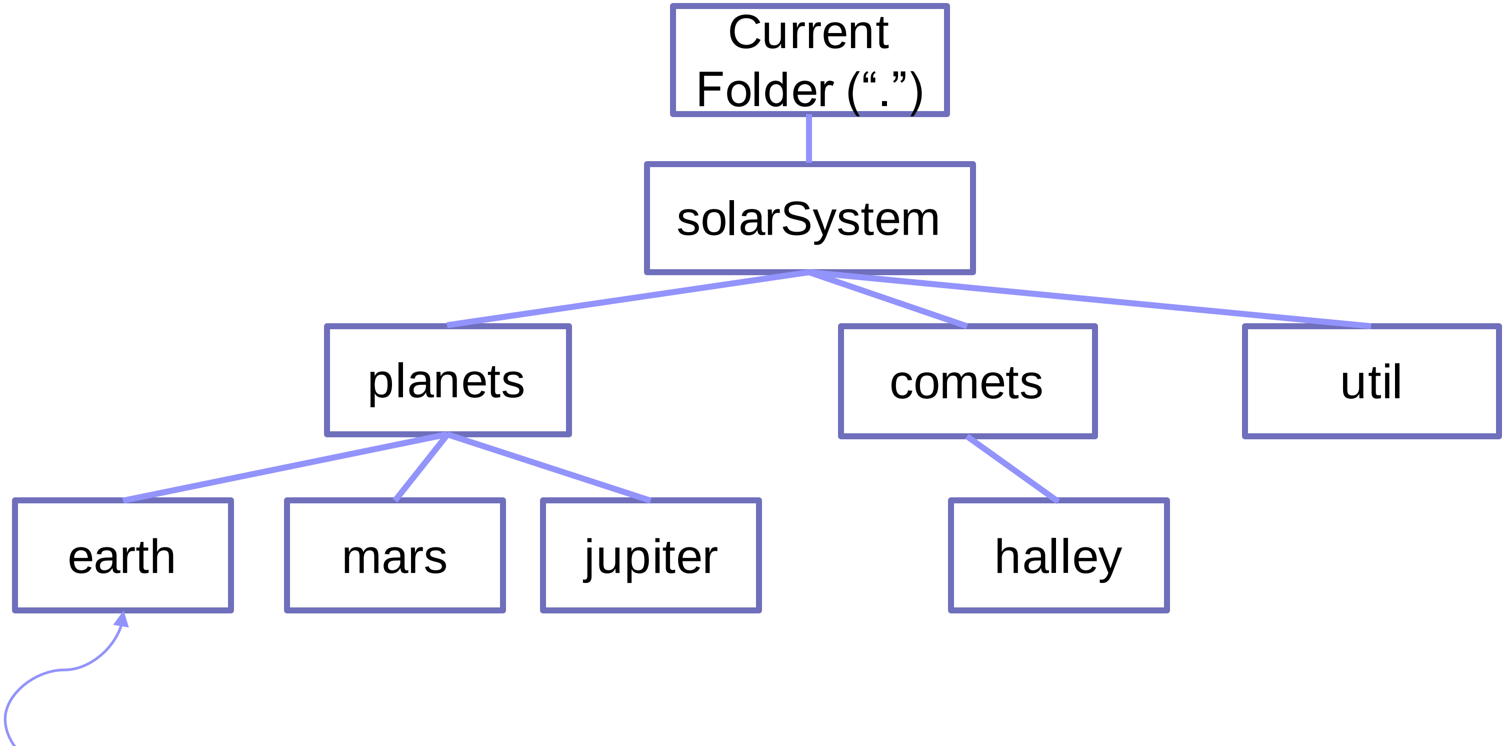
- Packages are *named* using the concatenation of the enclosing package names
- Types (e.g. classes) must declare what package they belong to
  - Otherwise they are placed in the "default" (unnamed) package
- Package names become part of the class name; the following class has the full name

  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:



Current Folder (".")

solarSystem

planets

comets

util

earth

mars

jupiter

halley

Class "Human" (inside folder "earth")

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

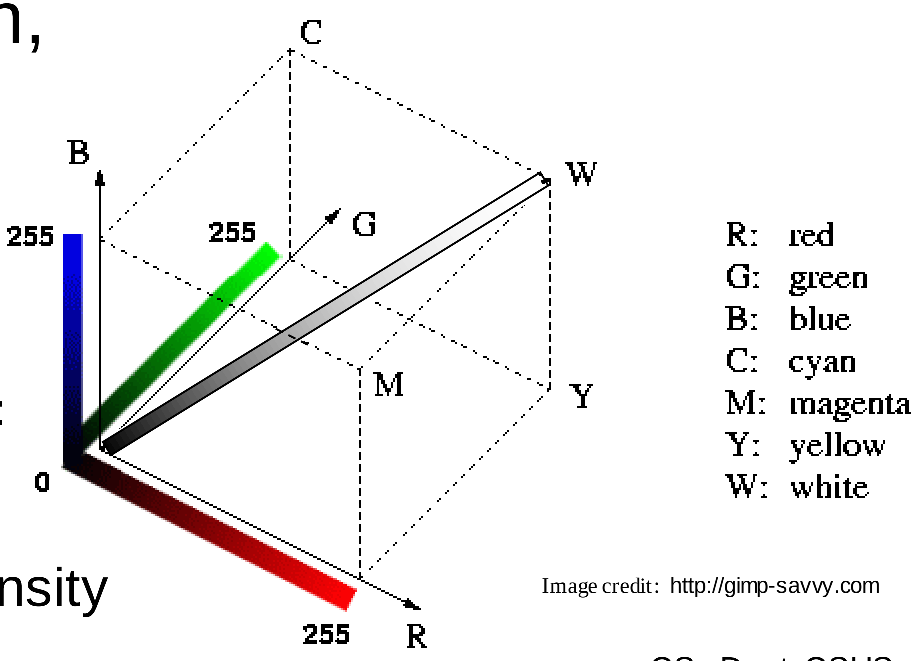


Image credit: http://gimp-savvy.com

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]
```

Questions: (1) Class name ?
(2) Methods invocation ?
(3) Capitalized GREEN ?

Breaking Encapsulations

- **The wrong way, with public data:**

```
public class Point {
  public double x, y;

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

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

BAD!

Breaking Encapsulations (cont.)

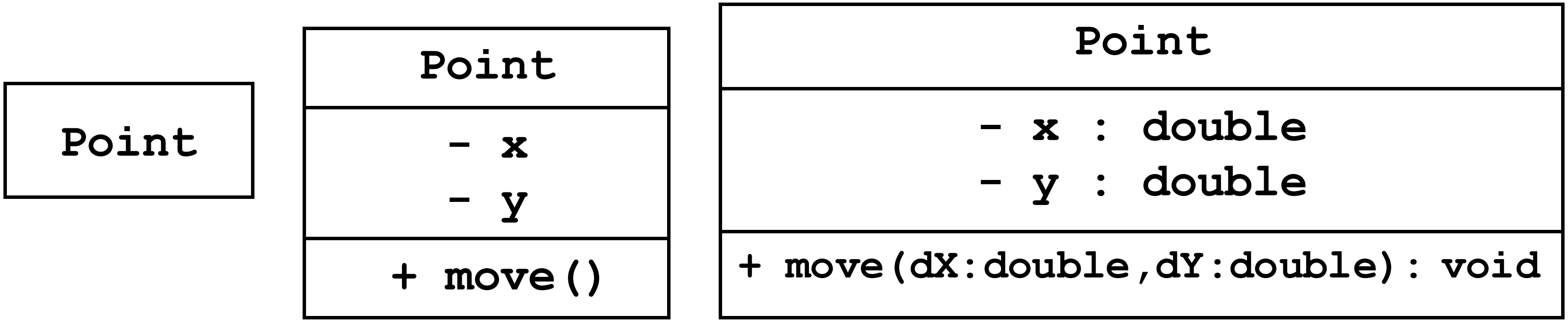
- **The correct way, with "Accessors":**

```
public class Point {

    private double x, y ;

    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.
}
```

Note

UML “Class Diagrams”

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



```
Point
```

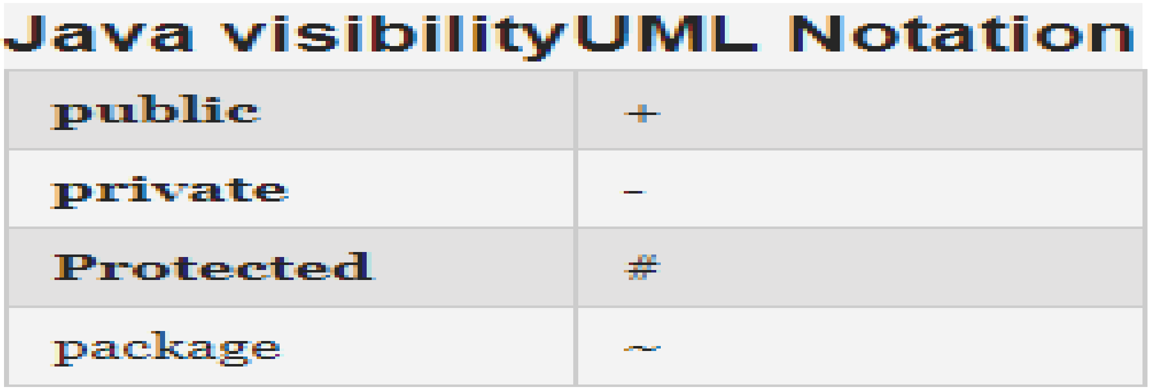
```
Point
- x
- y
+ move()
```

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

Class Name, Attributes, Methods notation

+, -, # , ~ ?

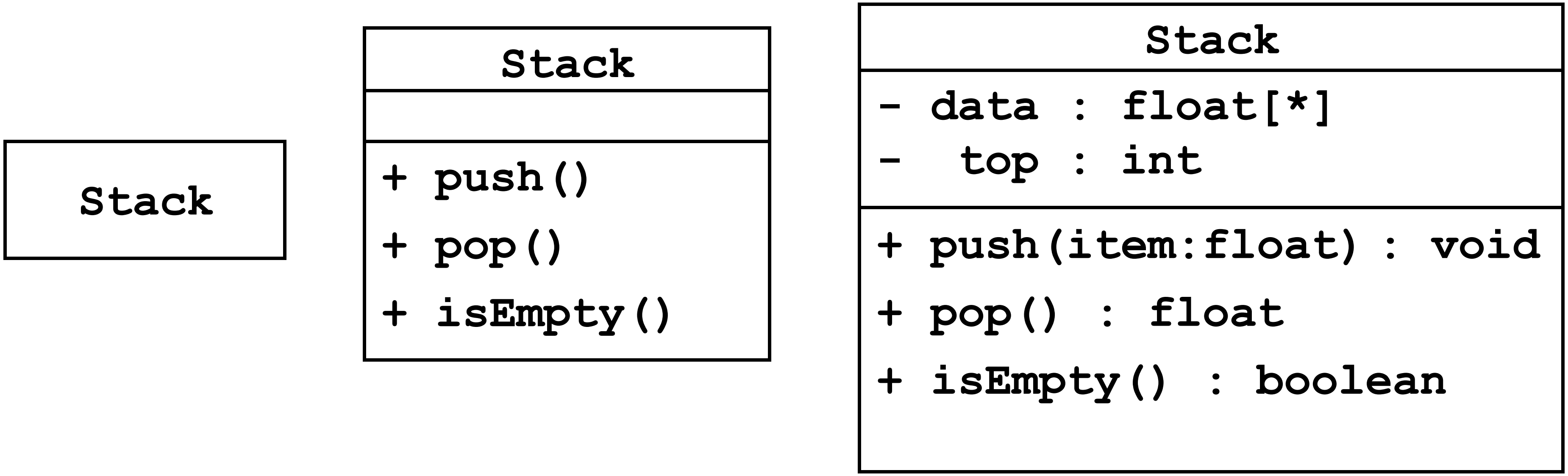
Java Visibility UML Notation



Java visibility	UML Notation
public	+
private	–
Protected	#
package	~

UML "Class Diagrams" (cont.)

- UML for the "**Stack**" class:



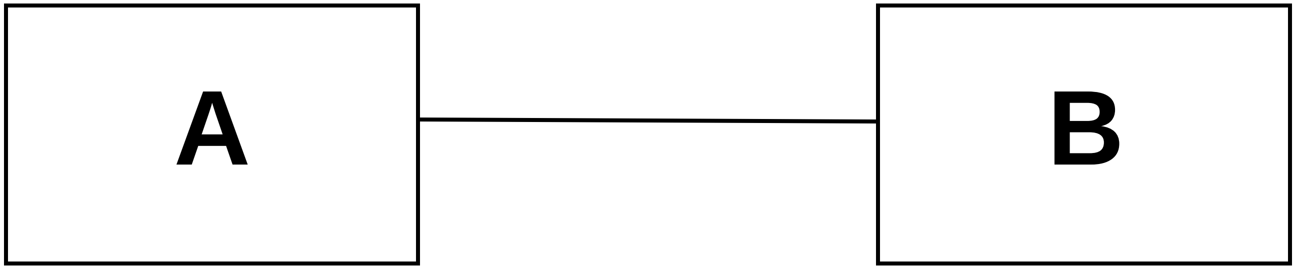
```
Stack
```

```
Stack

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

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

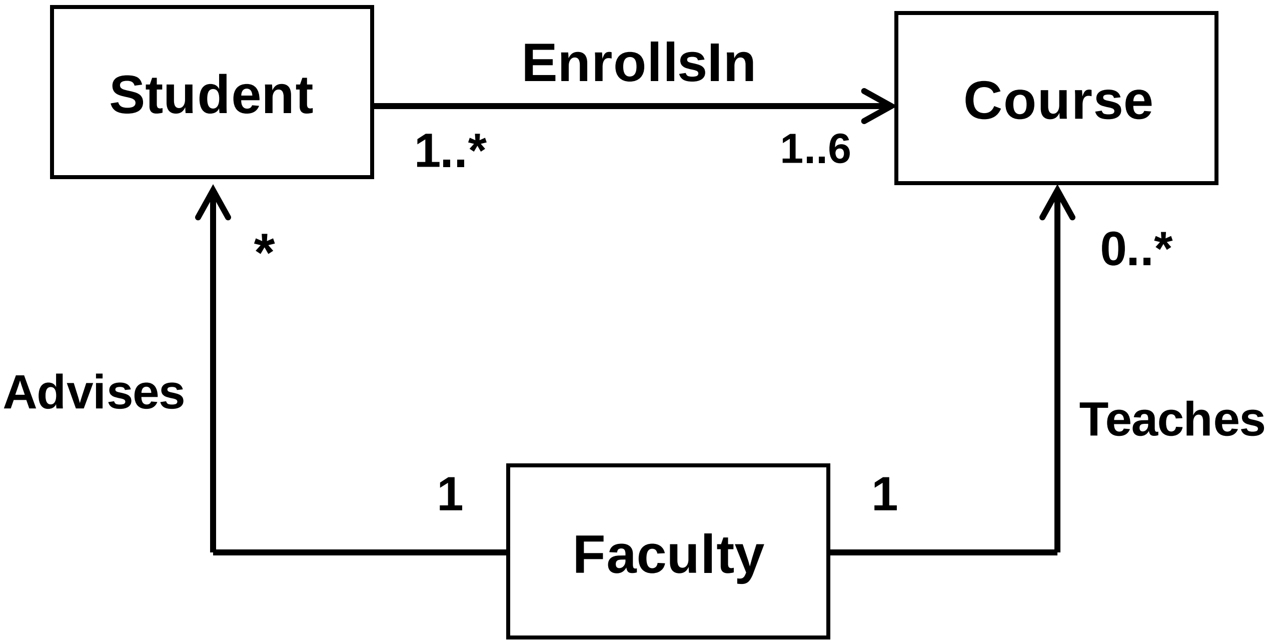
Associations

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

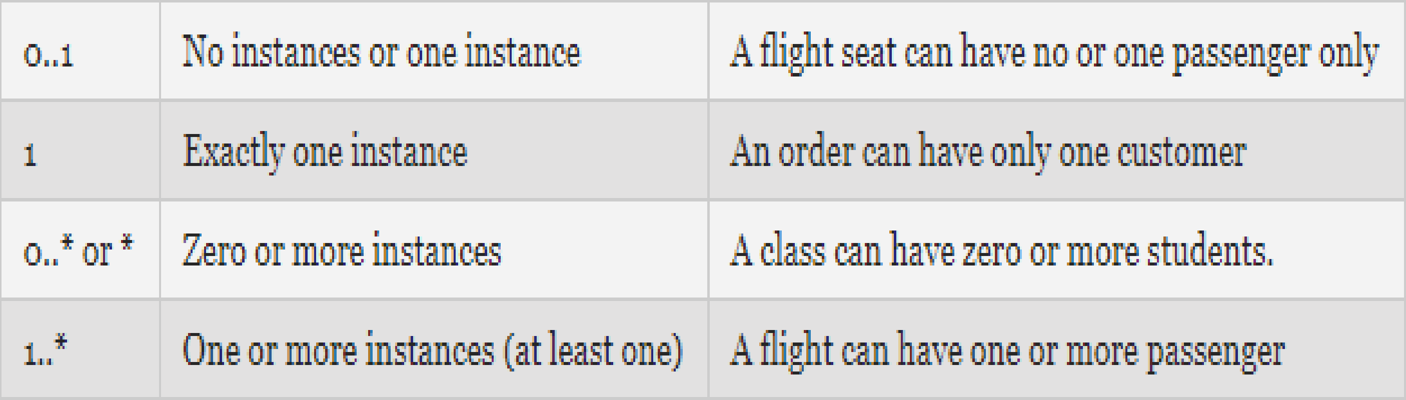


Associations (cont.)

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



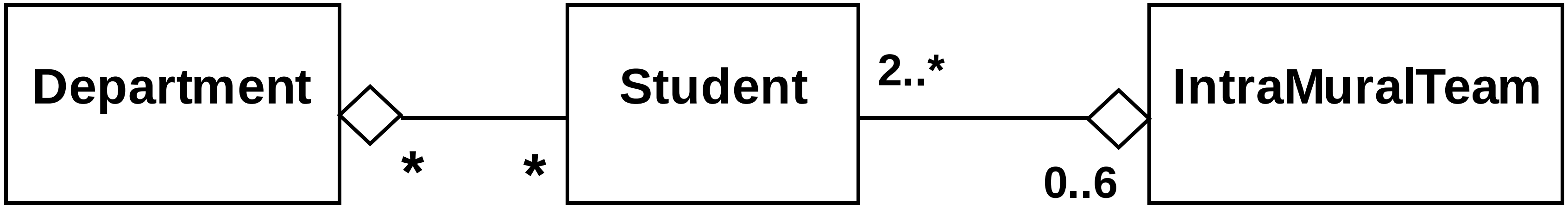
Multiplicity



0..1	No instances or one instance	A flight seat can have no or one passenger only
1	Exactly one instance	An order can have only one customer
0..* or *	Zero or more instances	A class can have zero or more students.
1..*	One or more instances (at least one)	A flight can have one or more passenger

Special Kinds Of Associations

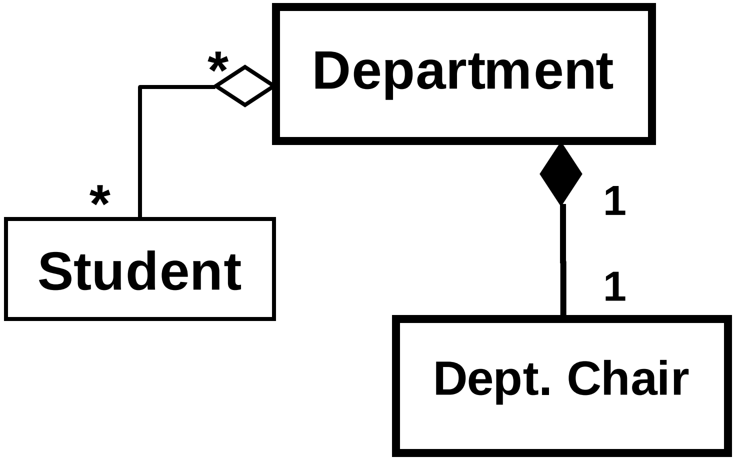
- **Aggregation**
  - Represents ***“has-a”*** or ***“is-Part-Of”***



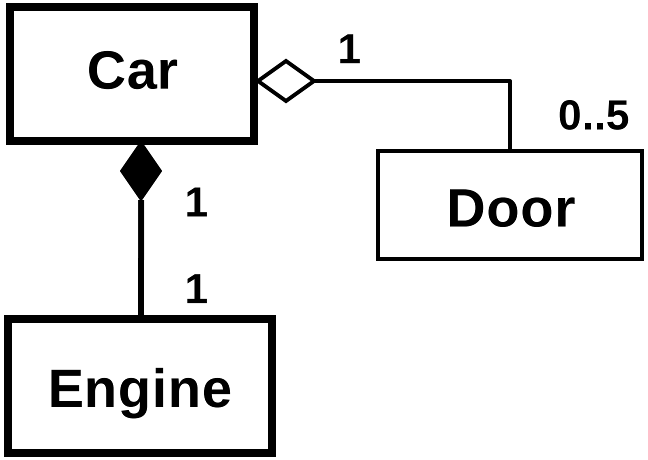
- **An IntraMuralTeam is an aggregate of *(has)* 2 or more Students**
- **A Student *is-a-part-of* at most six Teams**
- **A Department has any number of Students**
- **A Student can belong to any number of Departments (e.g. double major)**

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**)



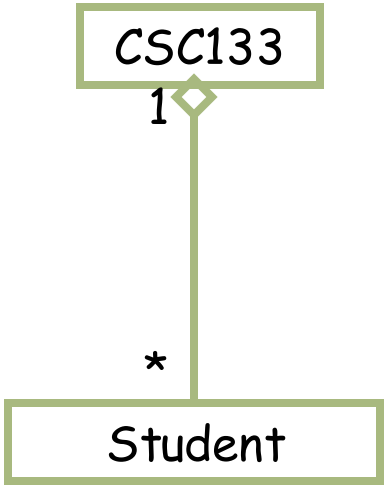
Exclusive ownership



Required ownership

Example: Aggregation vs. Composition

Aggregation

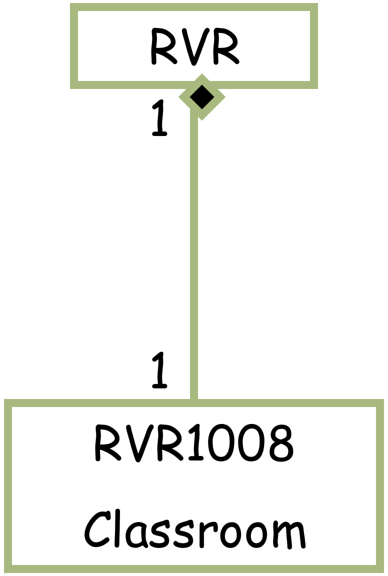


An association in which one class belongs to a collection

- Shared: An object can exist in more than one collections
- No ownership implied

Denoted by <u>hollow</u> diamond on the “contains” side

Composition



RVR =
Riverside Hall
ECS Building

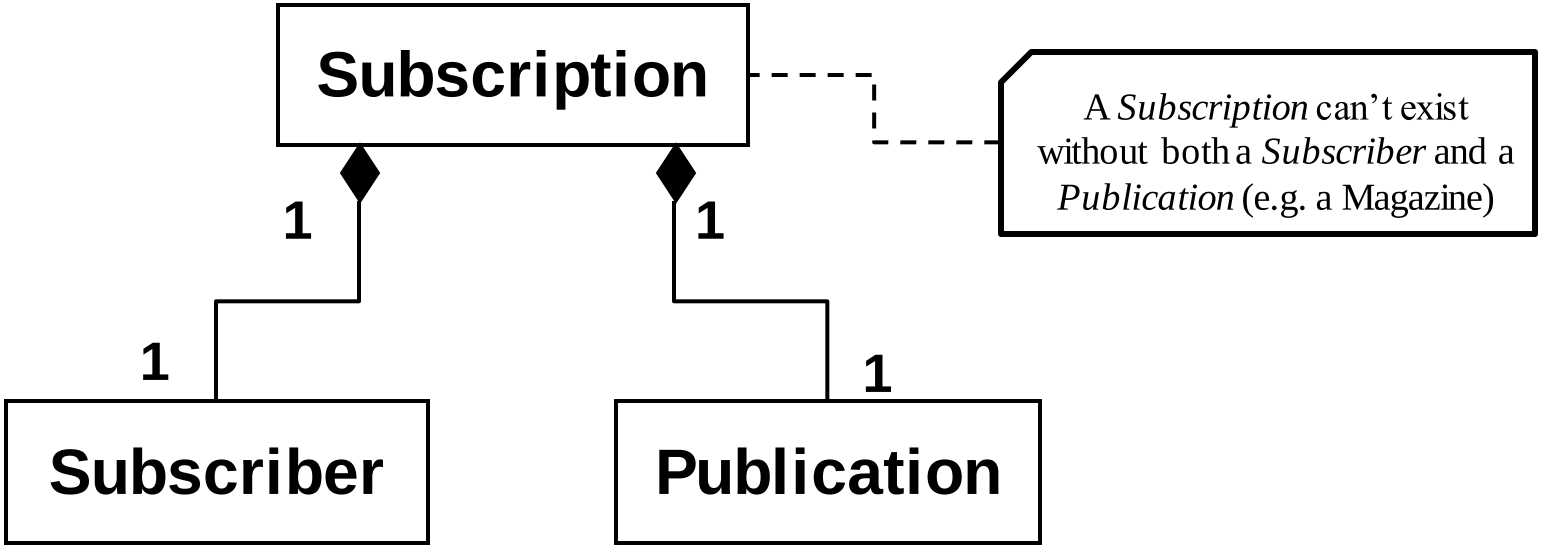
An association in which one class belongs to a collection

- No Sharing: An object cannot exist in more than one collections
- Strong “has a” relationship
- Ownership

Denoted by <u>filled</u> diamond on the “contains” side

Special Kinds Of Associations (cont.)

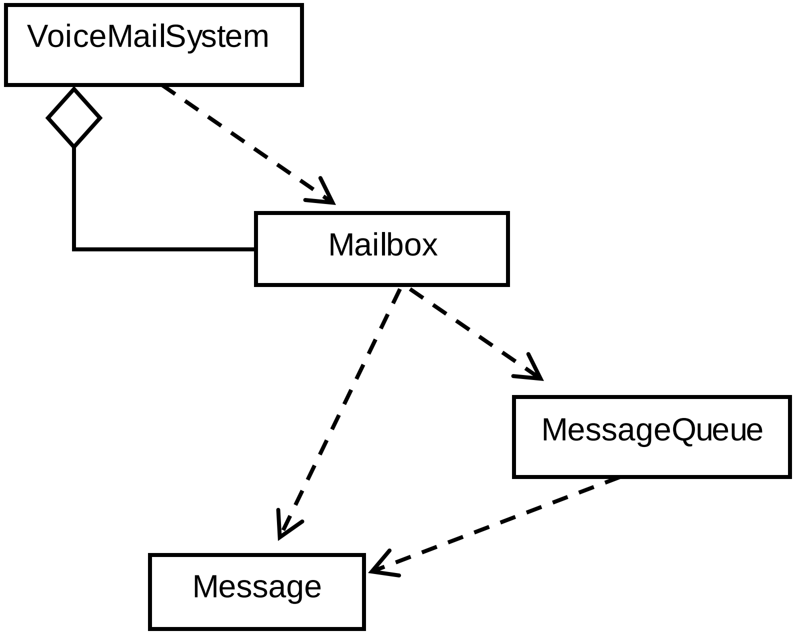
- Composition (another example)


Subscription
1
1
1
1
Subscriber
Publication
A *Subscription* can't exist without both a *Subscriber* and a *Publication* (e.g. a Magazine)

Special Kinds Of Associations (cont.)

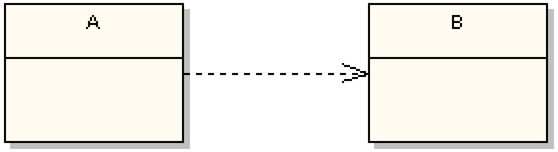
- Dependency
  - **Represents “uses” (or “knows about”)**

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



More on Dependency (cont.)

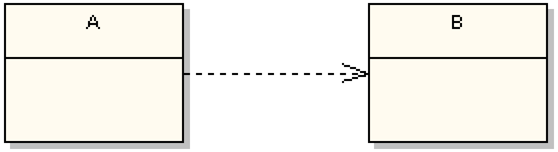
- Dependency



  - **Represents "<u>uses</u>" (or "knows about")**
  - It means that the class at the source end of the relationship has some sort of dependency on the class at the target (arrowhead) end of the relationship.
  - Class A uses class B, but that class A <u>does not</u> contain an instance of class B <u>as part of its own state.</u>

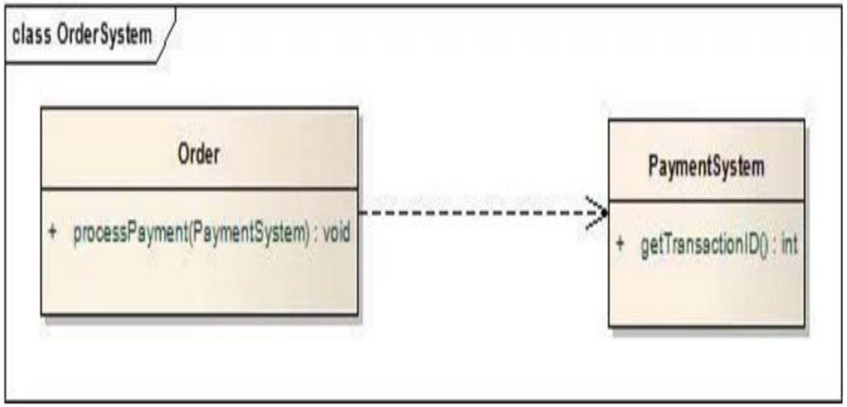
Examples Dependency (cont.)

```
class A {
   void foo(){
      b object= new B();
      object.baar();
   }
}
class B {
   void baar(){
   }
}
```

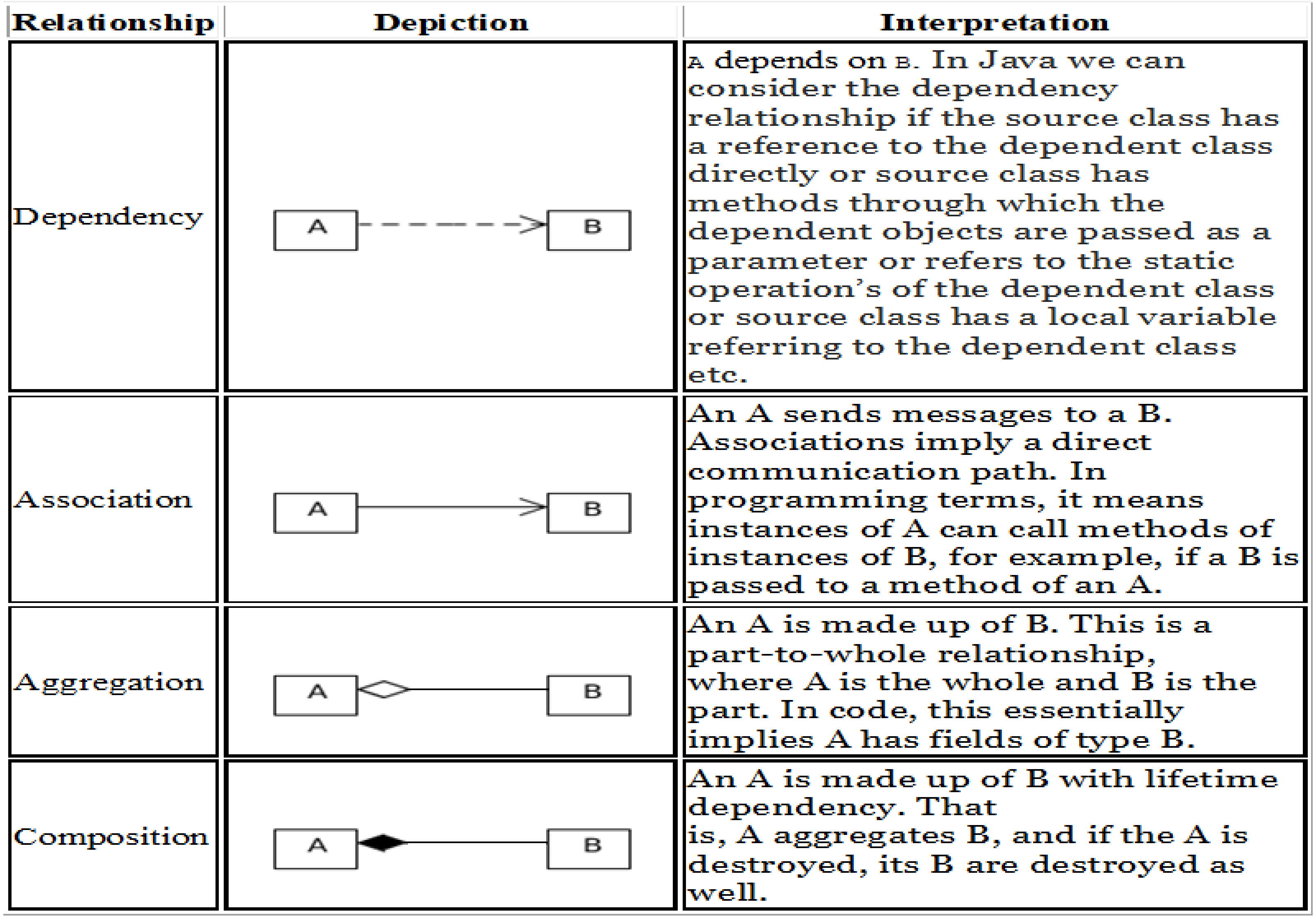


Class A uses class B. Therefore class A has a dependency on class B.

```
public class PaymentSystem {

}

public class Order {
  public void   processPayment(PaymentSystem ps){

  }
}
```

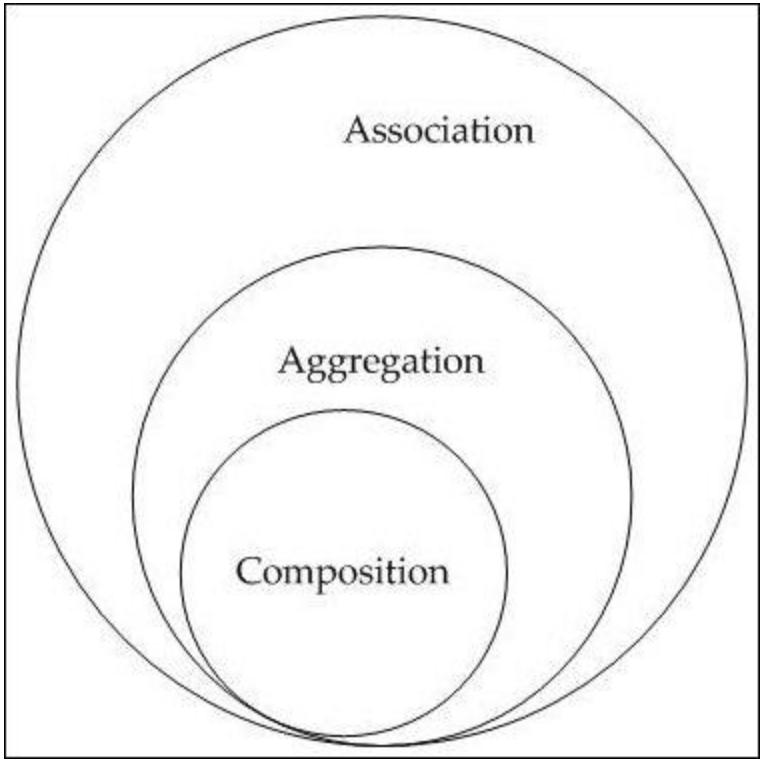

class OrderSystem
Order
+ processPayment(PaymentSystem) : void
PaymentSystem
+ getTransactionID() : int

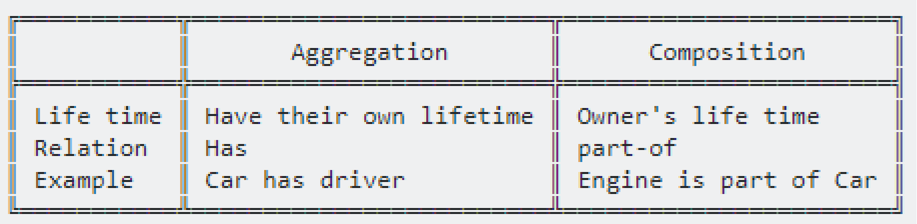
Recap 1



Relationship	Depiction	Interpretation
Dependency	A B	A depends on B. In Java we can consider the dependency relationship if the source class has a reference to the dependent class directly or source class has methods through which the dependent objects are passed as a parameter or refers to the static operation's of the dependent class or source class has a local variable referring to the dependent class etc.
Association	A B	An A sends messages to a B. Associations imply a direct communication path. In programming terms, it means instances of A can call methods of instances of B, for example, if a B is passed to a method of an A.
Aggregation	A B	An A is made up of B. This is a part-to-whole relationship, where A is the whole and B is the part. In code, this essentially implies A has fields of type B.
Composition	A B	An A is made up of B with lifetime dependency. That is, A aggregates B, and if the A is destroyed, its B are destroyed as well.

Recap 2





	Aggregation	Composition
Life time	Have their own lifetime	Owner's life time
Relation	Has	part-of
Example	Car has driver	Engine is part of Car

Sometimes, it can be a complicated process to decide if we should use association, aggregation, or composition. This difficulty is caused in part because **aggregation** and **composition** are subsets of **association**, meaning they are specific cases of association.

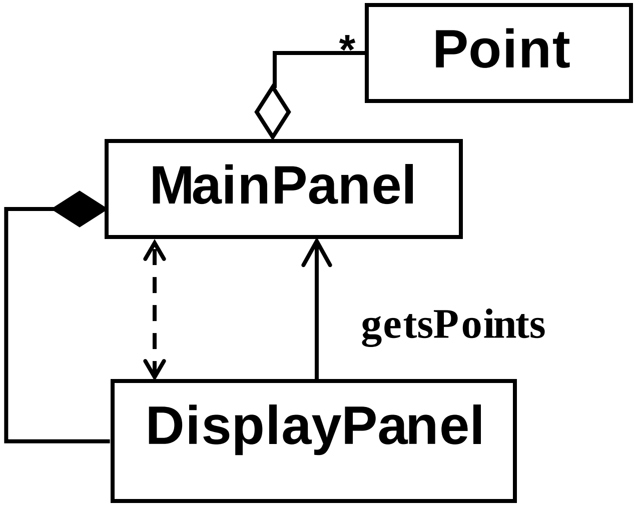
Source: https://softwareengineering.stackexchange.com/questions/61376/aggregation-vs-composition

Implementing Associations

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

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

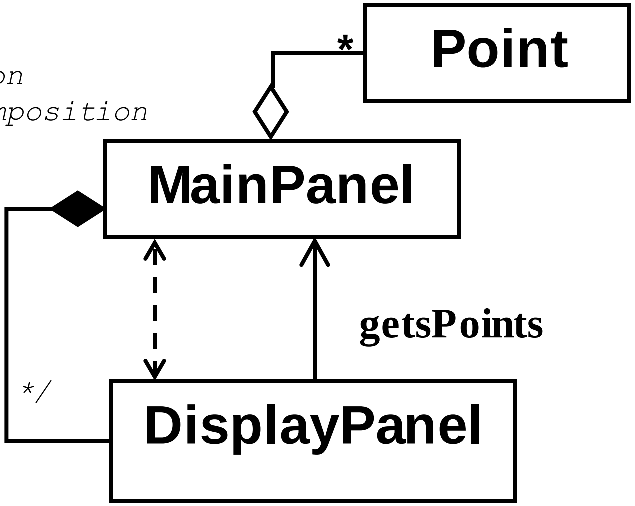
```
public class DisplayPanel {
   private MainPanel myMainPanel ;

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



Question: Code reference to diagram ?

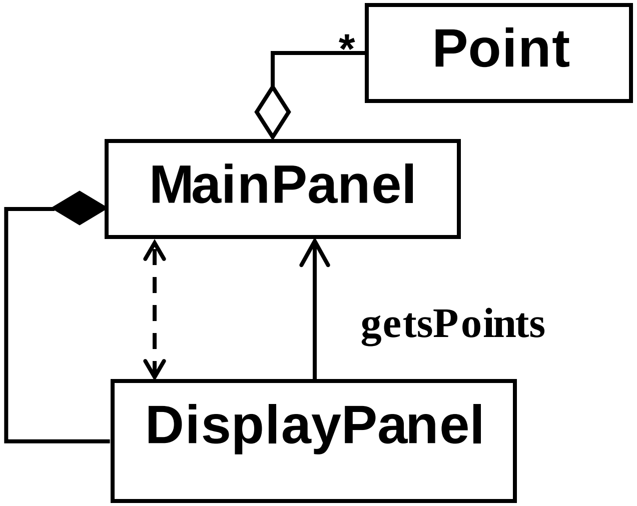
Implementing Associations (cont.)

```
/**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);
        }
    }
}
```


*
Point
MainPanel
getsPoints
DisplayPanel