CSC 133

Object-Oriented Computer Graphics Programming

OOP Concepts I - Overview

Dr. Kin Chung Kwan

Spring 2023

Computer Science Department
California State University, Sacramento

SACRAMENTO STATE

DOUBLE - O

Object-Oriented Programming Concept

Object-Oriented Programming

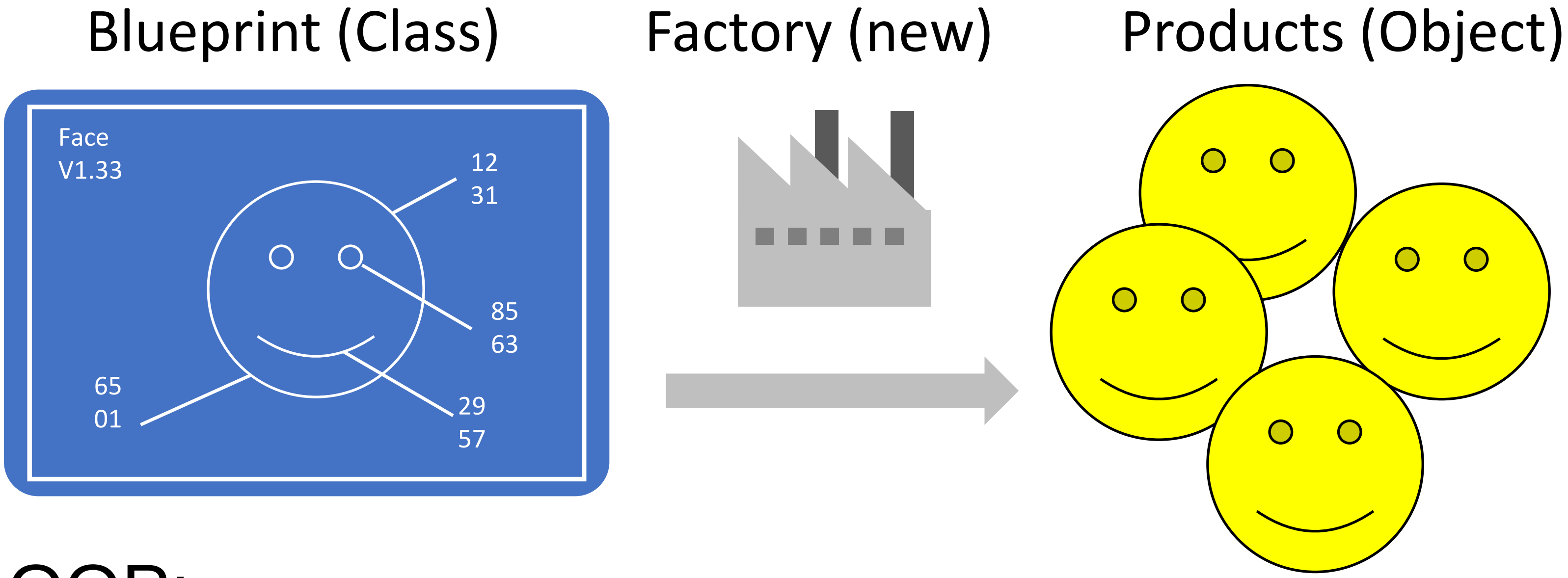
In view of application users:

- No difference

In the view of programmer

- Affecting the code design
- Reuse existing code

Class and Object



Blueprint (Class)

Factory (new)

Products (Object)

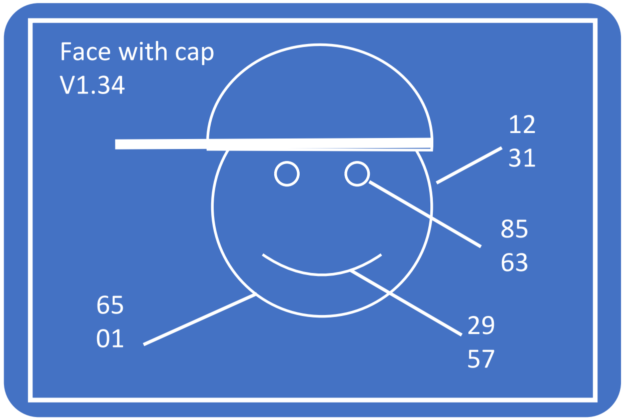
OOP:

- Design the blueprint and use the products
- Idea vs physical

Problem

What if I want to create a new blueprint

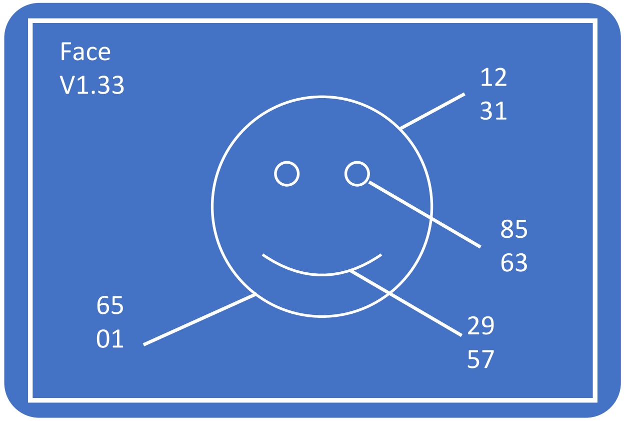
- But with varieties

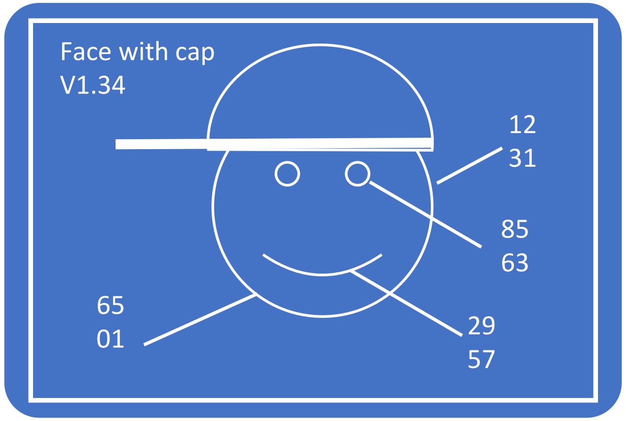


Draw on the original one?

- Not good as you may need the original one
- E.g., original taste vs new taste

Create a new one?



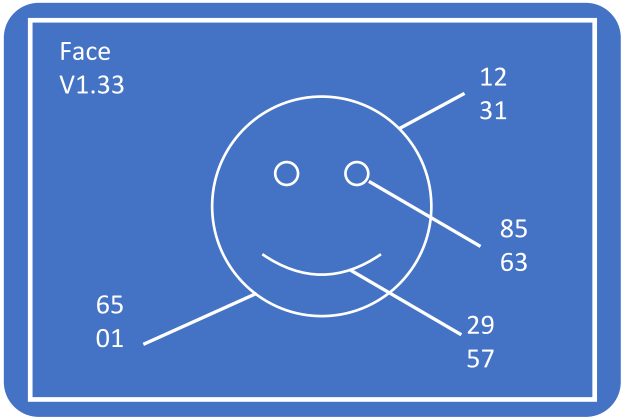


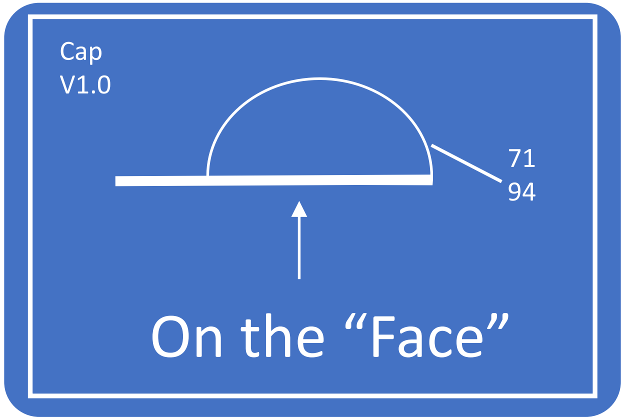
Recreate everything every time?

- And then add a pair of earphone, add a …..
- Or you want to fix the face design

Reuse the Resources

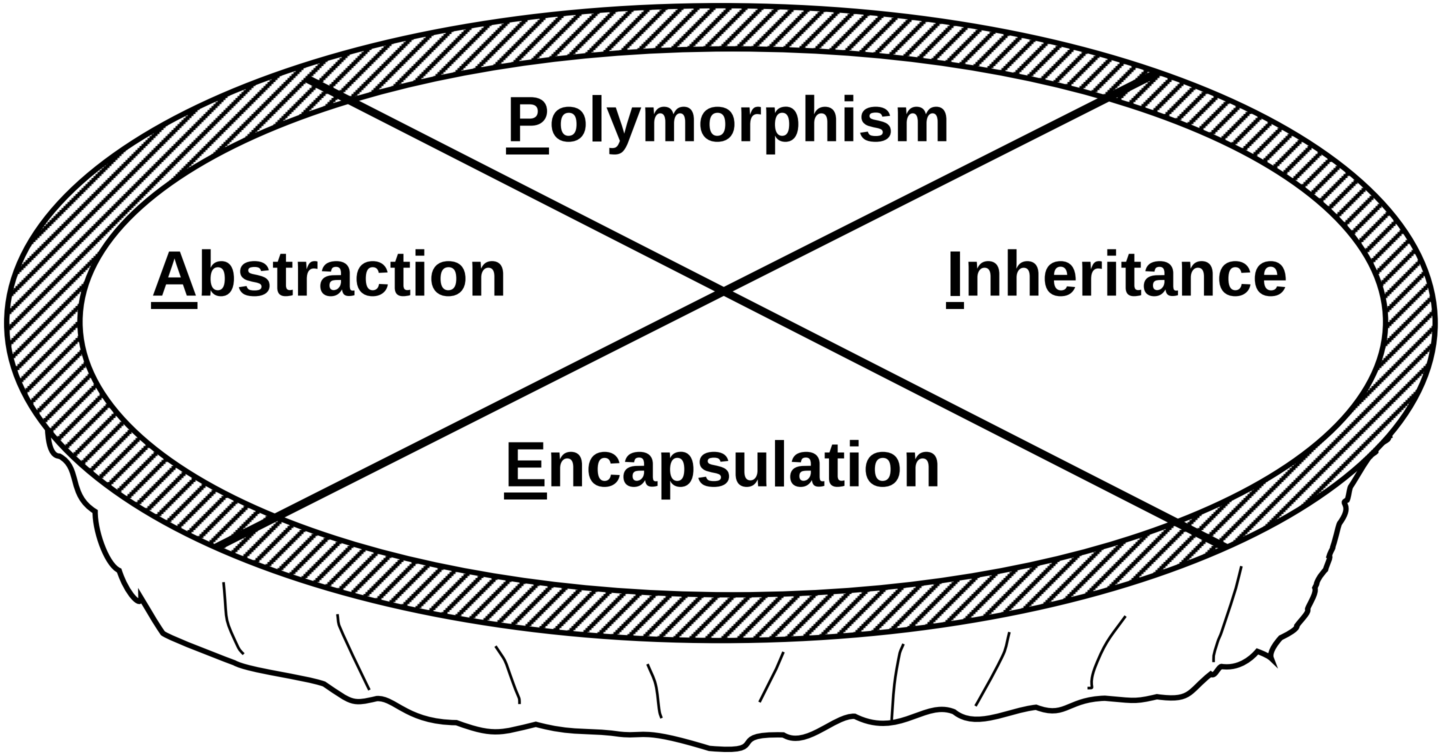
Only need to specific relationship of two classes





“A Pie”

Four distinct OOP Concepts (or Pillars)



A

Abstraction

Major idea of class

- Keep the minimum essential characteristics of an entity

Two abstraction types in Java:

1. Procedural abstraction
2. Data abstraction

Procedural Abstraction

Some objects can help you to do

- Without understanding its underlining

Siri, I cannot understand. Call the teacher to complaint!

```
phone.call(teacher);
```

Data Abstraction

No need to know how do the data stored

- Add new data?

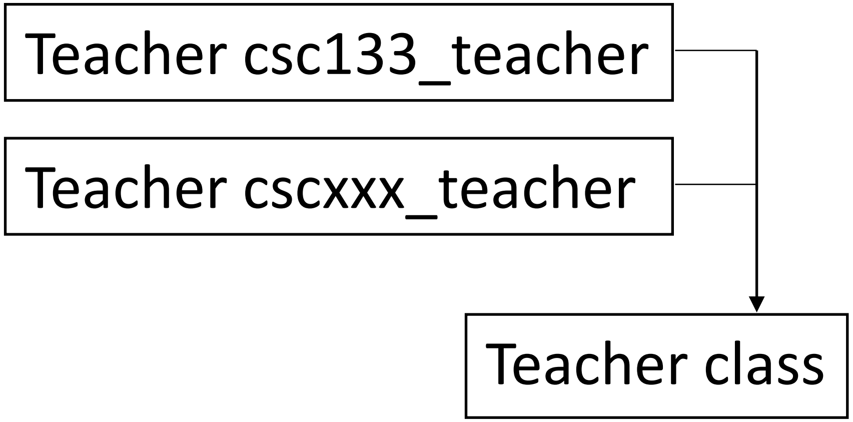
Without OO

string csc133_teacher_name
string csc133_teacher_age
string csc133_teacher_gender
string csc133_teacher_att

string cscxxx_teacher_name
string cscxxx_teacher_age
string cscxxx_teacher_gender
string cscxxx_teacher_att

Add for every class

With OO



Just add one here!

Abstraction

Advantage

- Use them without knowing the detail
- Easy to code and modify

Good for

- Collaboration
- Large and complex systems

Disadvantage

- Loss of content

P

Polymorphism

Literally: from the Greek

poly *("many")* + ***morphos*** *("forms")*

Examples in nature:

- Carbon: graphite or diamond
- H_2O: water, ice, or steam
- Blood: A, B, AB, or O type

Polymorphism Example

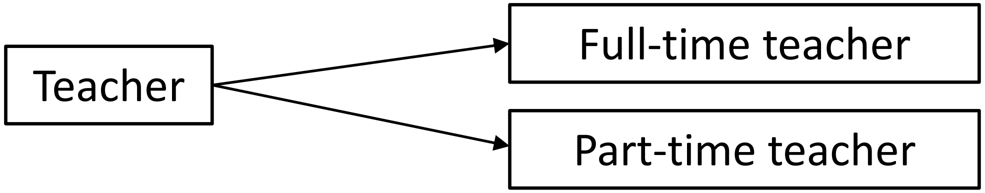
Same operation for various types of objects

kc.learnFrom(teacher)   *vs*   *kc.learnFrom(student)*

Same operation in a variety of ways

kc.teachCSC133( )   *vs*   *other.teachCSC133( )*

A reference to different types



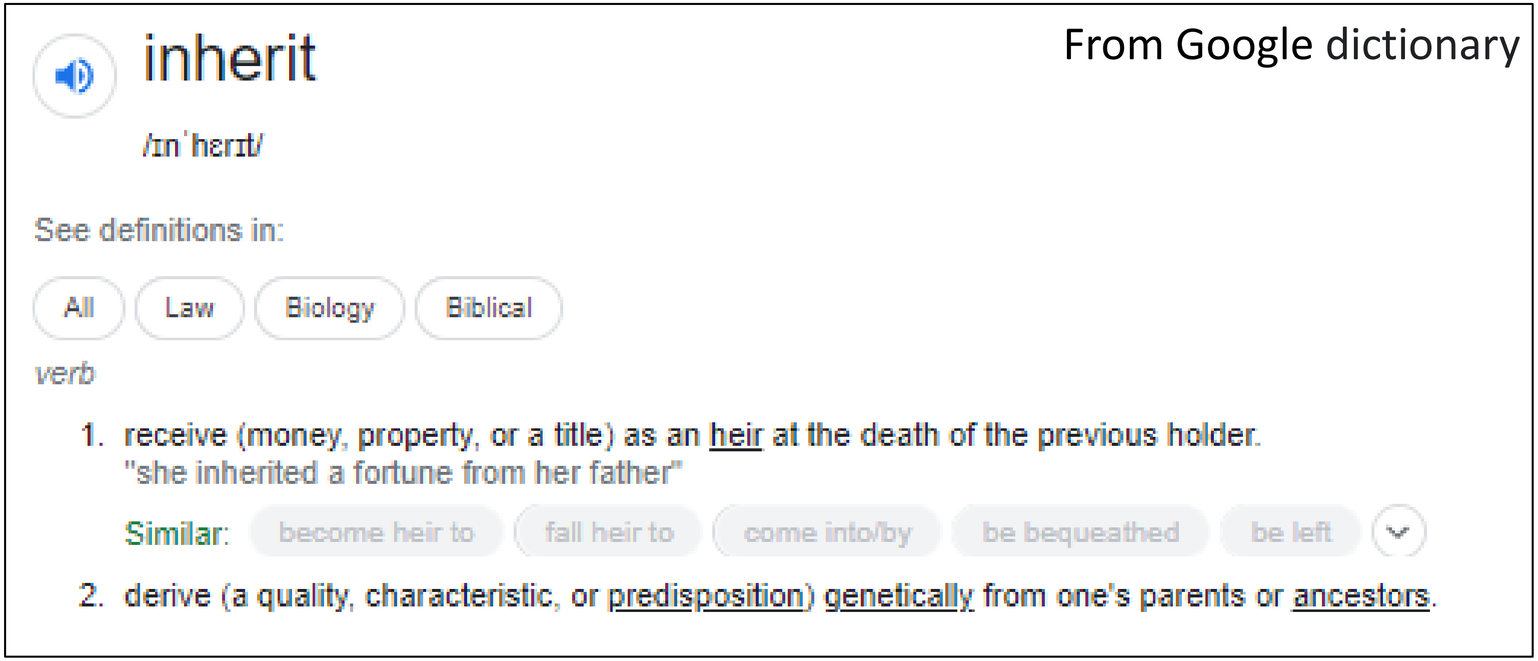
Teacher

Full-time teacher

Part-time teacher

I

Inheritance



From Google dictionary

inherit

/ɪnˈhɛrɪt/

See definitions in:

verb

1. receive (money, property, or a title) as an heir at the death of the previous holder.
   "she inherited a fortune from her father"

   Similar: become heir to | fall heir to | come into/by | be bequeathed | be left

2. derive (a quality, characteristic, or predisposition) genetically from one's parents or ancestors.

Parent-child

- Child has the same ability of the parent

Inheritance Example

In OO, this is “is-a” relationship.

Teacher can teach (well or poorly☺)

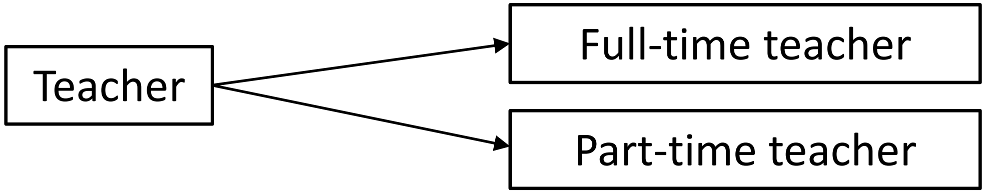
- “Full-time teacher” **is a** teacher
- Can teach

Teacher	Full-time teacher
Parent class	Child class
Is a class	is a teacher
Can teach	Can teach

Polymorphism VS Inheritance

Polymorphism:

- If you are a teacher, you can be either full-time or part-time

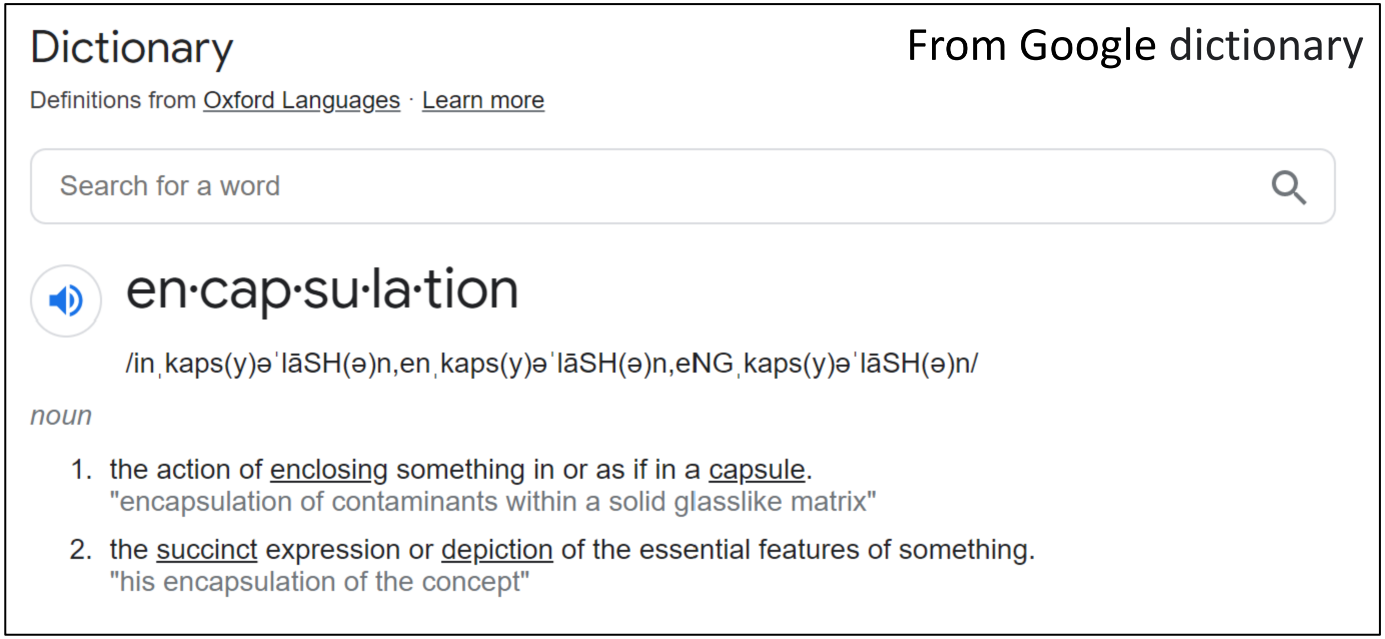


Inheritance:

- If you are a full-time teacher, you can do what teacher do.

E

Encapsulation



From Google dictionary

Dictionary

Definitions from Oxford Languages · Learn more

en·cap·su·la·tion

/inˌkaps(y)əˈlāSH(ə)n,enˌkaps(y)əˈlāSH(ə)n,eNGˌkaps(y)əˈlāSH(ə)n/

noun

1. the action of enclosing something in or as if in a capsule.
   "encapsulation of contaminants within a solid glasslike matrix"
2. the succinct expression or depiction of the essential features of something.
   "his encapsulation of the concept"

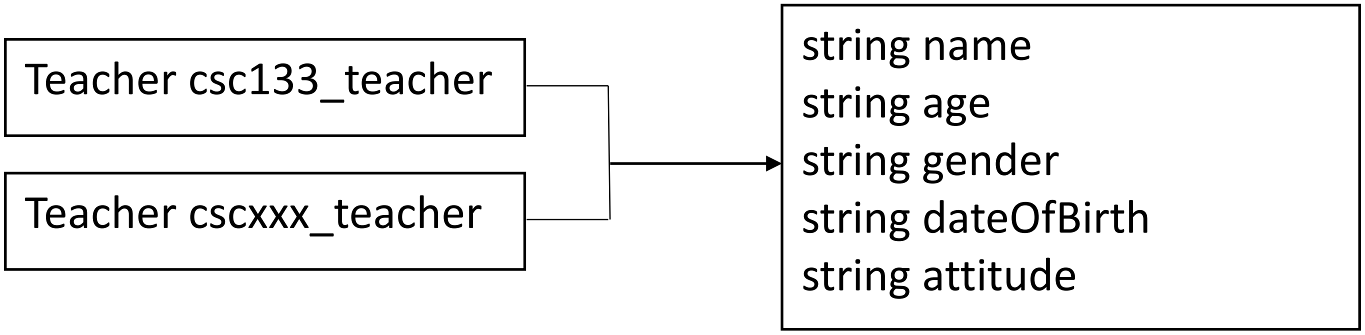
Enclosing something

Encapsulation

Implementing a class = doing encapsulation.

“Bundling”

- Class has fields (data) and methods (procedures)



Teacher csc133_teacher

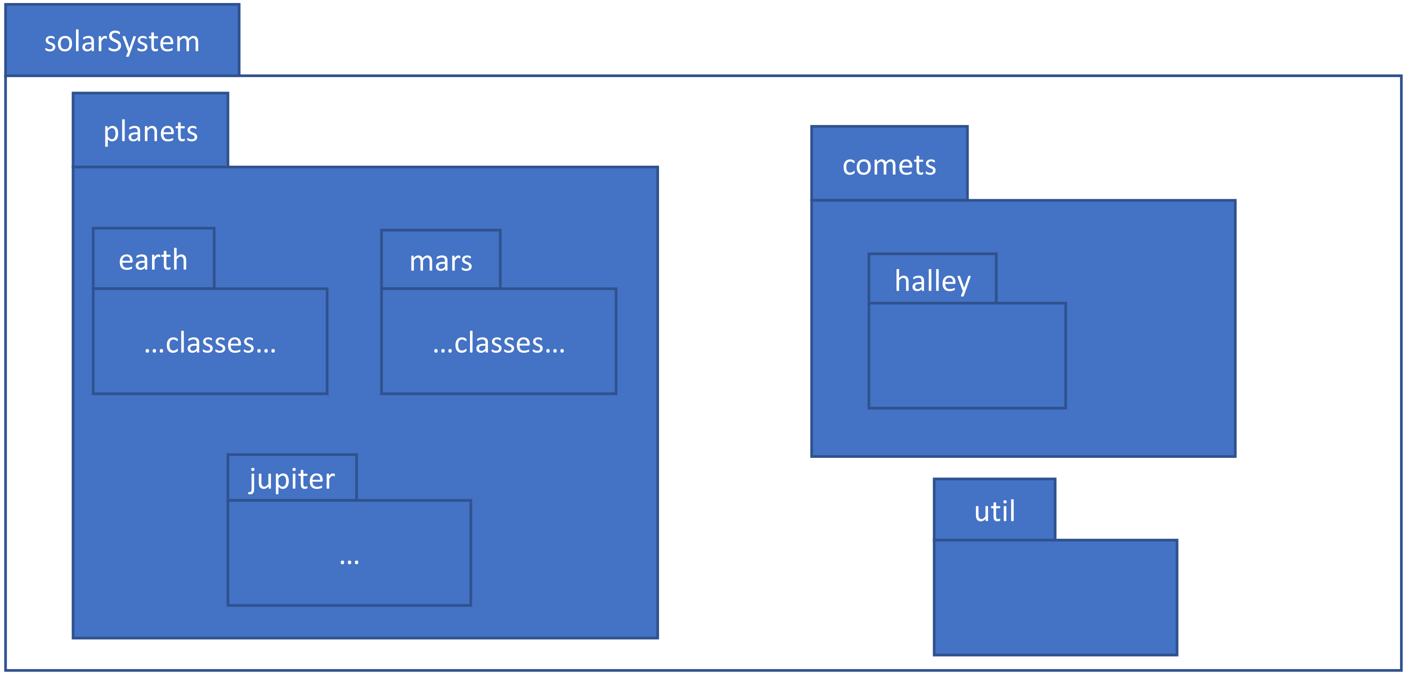
Teacher cscxxx_teacher

string name
string age
string gender
string dateOfBirth
string attitude

Java Packages

Java package collect a group of classes

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



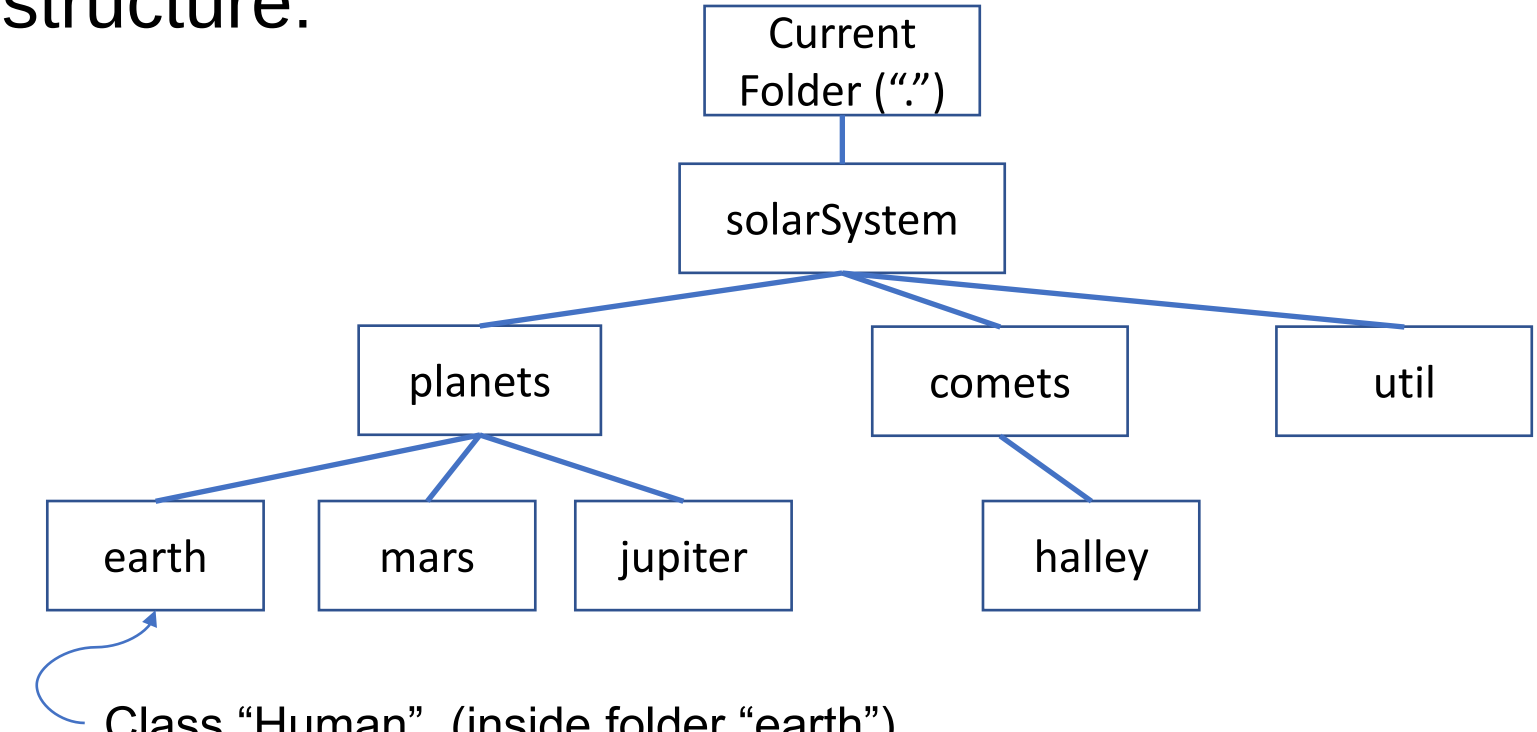
Java Packages

- Named using the concatenation of the enclosing package names
- Classes can declare what package they belong to
  - Default placed in the “default” (unnamed) package
- Package names is added to the class name;
  - Full name of this class: *solarSystem.planets.earth.Human*

```
package solarSystem.planets.earth ;

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

}
```

Packages and Folders

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



Class "Human" (inside folder "earth")

Example: CN1 `ColorUtil` Class

- An *encapsulated* example
- **`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]
```

Encapsulation

“Information Hiding”

- Prevents certain aspects of the abstraction from being accessible to its clients
- E.g., access the variable only with a certain way

KC’s Bank Account

Balance = -999,999

```
kc.bank.balance +=
9999999;
```

Good?

```
kc.deposit(10)
```

Better

Encapsulation

Visibility modifiers:

- Public: everyone can access
- Private: the object itself can access
- Protected: its child can access

Good way

- Keep all data **private**
- Use accessors (Get & Set)

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

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.
}
```

Access (Visibility) Modifiers

Java:

Modifier	Access Allowed By			
	Class	**Package**	**Subclass**	**World**
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"*

UML

UML Class Diagrams

Unified Modeling Language is a “graphical notation” for classes

For documentation

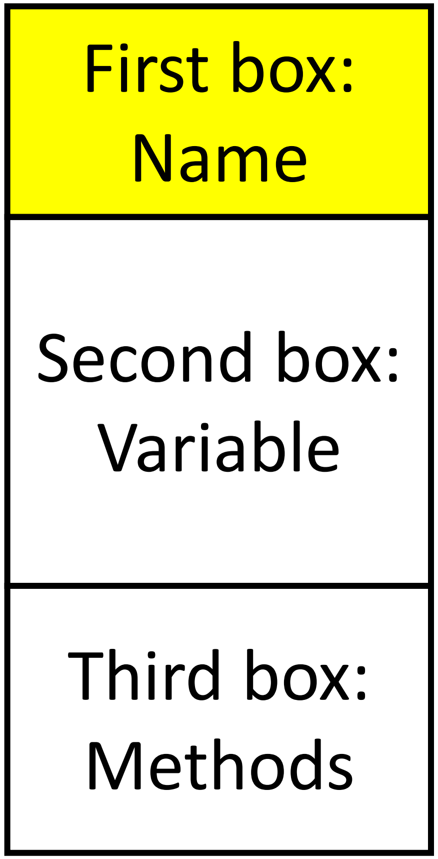
- the relationship of classes
- the data of classes
- the method of classes

UML Structure (Name)

The whole boxes represent one class

- 1 - 3 sub-boxes

First box: Class name



First box: Name

Second box: Variable

Third box: Methods

UML Structure (Variable)

+ name : type

- First part is a visibility

  + (public)

  - (private)

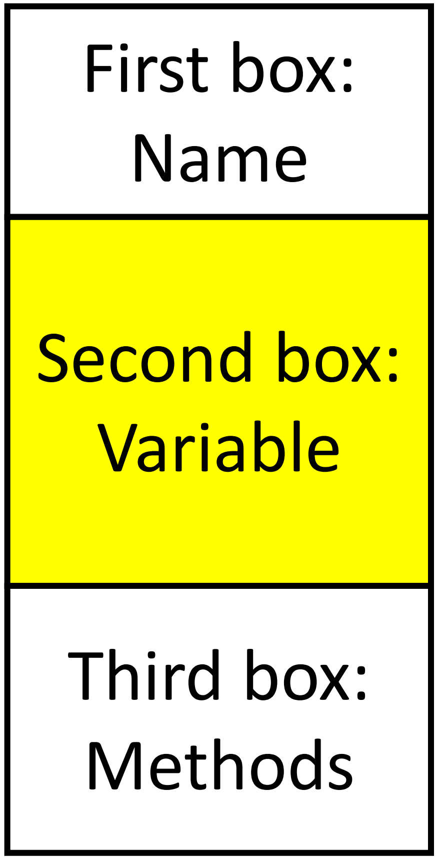
  # (protected)

- Middle is the variable name
- Last is the type after a colon

  : int

  : float

Last one is optional



First box:
Name

Second box:
Variable

Third box:
Methods

UML Structure (Methods)

- op (inout name: type): type

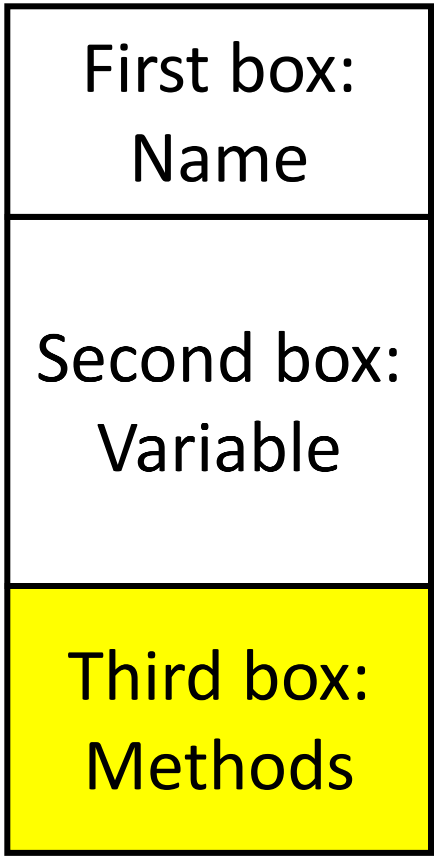
- First part is a visibility
- Second is the method name
- Third is parameter details

  in (input only)

  out (output only)

  inout (for input and output)
- Last is the return type

Third and last one is optional

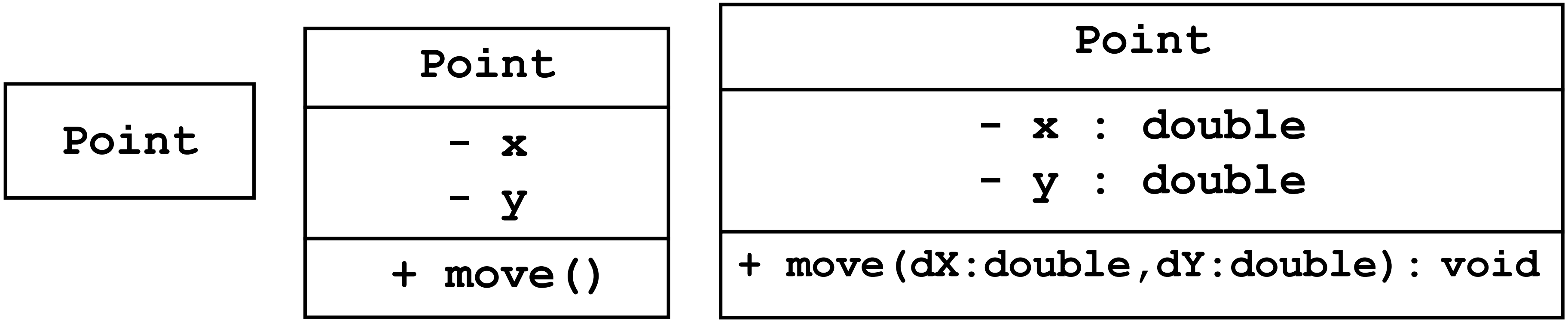


First box: Name

Second box: Variable

Third box: Methods

Example UML for Point

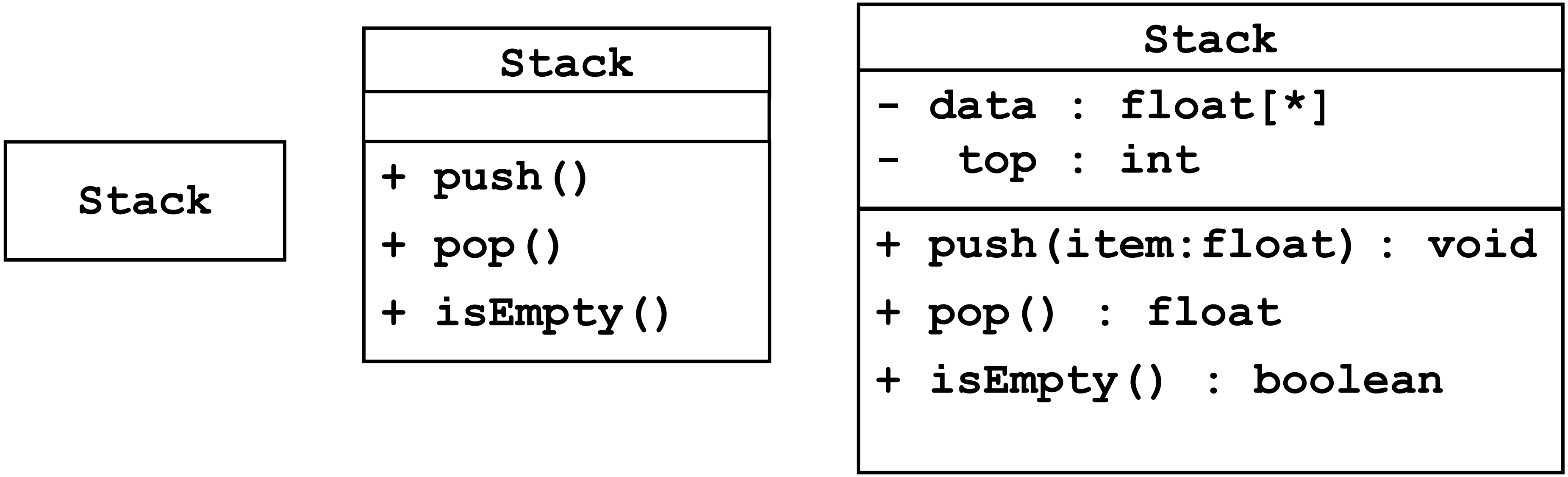


```
Point
```

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

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

Example UML for Stack



Stack

Stack

+ push()

+ pop()

+ isEmpty()

Stack

- data : float[*]
- top : int

+ push(item:float) : void

+ pop() : float

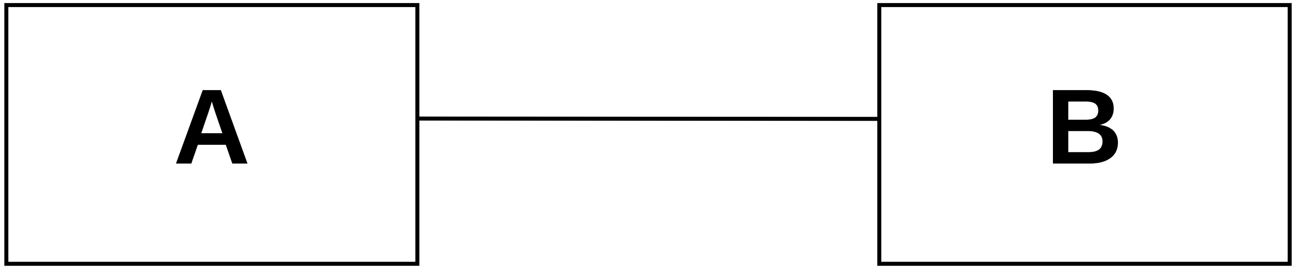
+ isEmpty() : boolean

Associations

Definition:

A relationship between two classes A and B if instances can send or receive messages (make method calls) between each other.

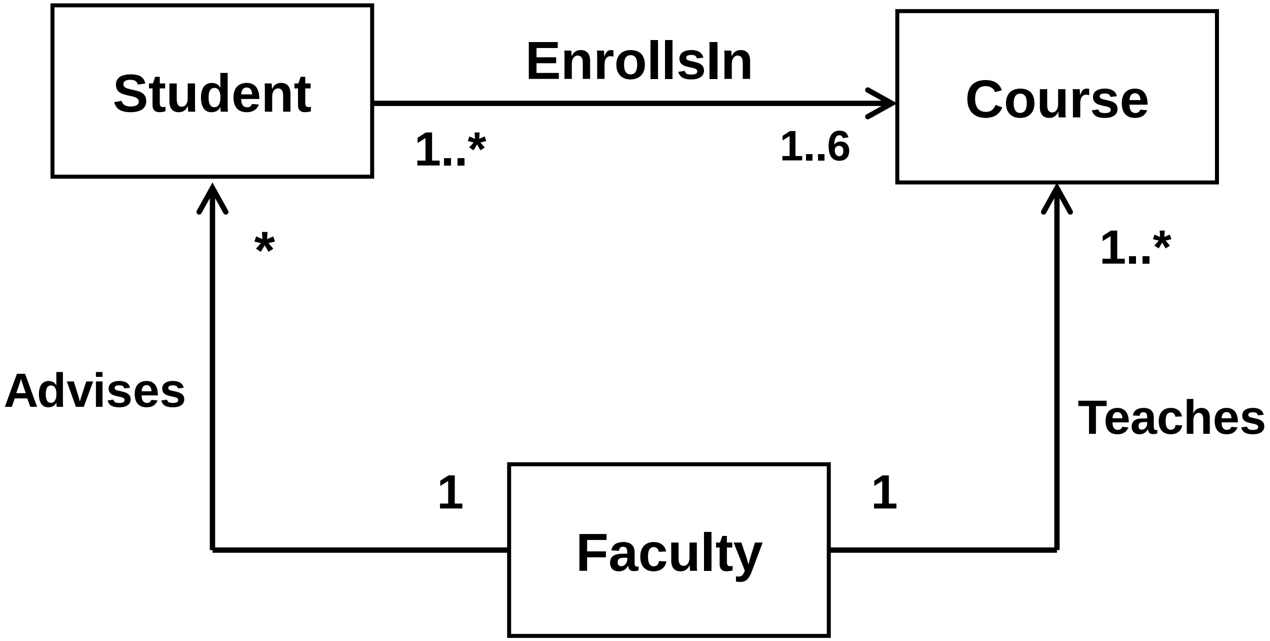
Link the classes to represent their associations in UML.



Properties of Associations

Associations can have properties:

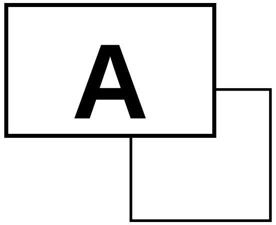
- **Cardinality**
  - Number of objects. (* means any number >=0)
- **Direction**
  - message
- **Label**
  - Name



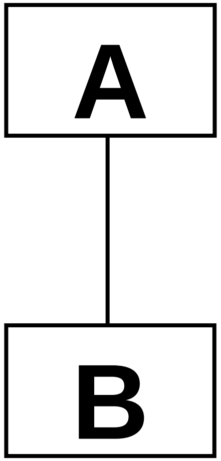
Properties of Associations

Associations can be N-ary

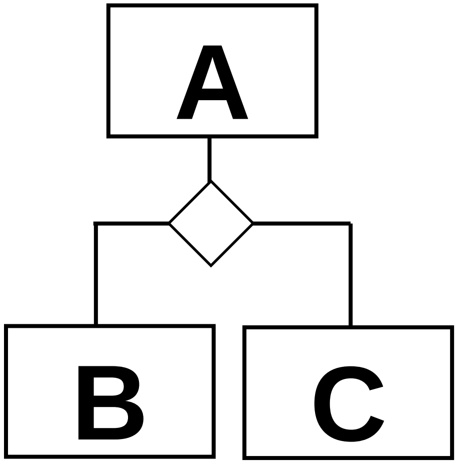
Unary (1)



Binary (2)

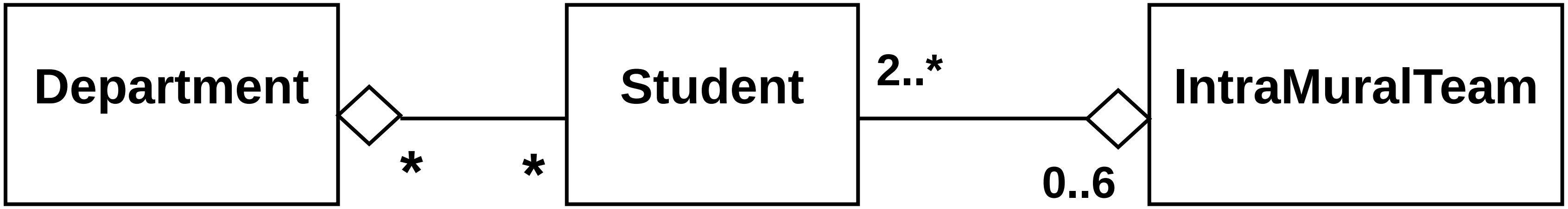


Ternary (3)



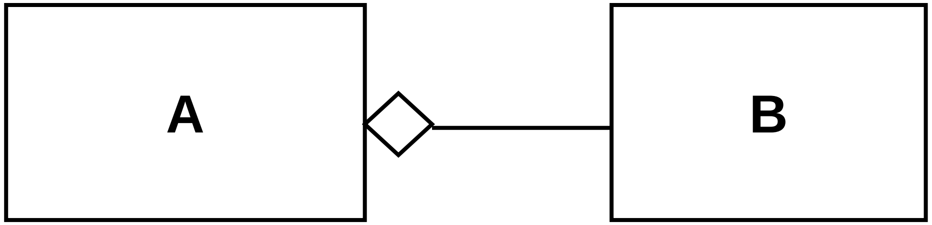
Aggregation

*“**has-a**”* (*“**is-Part-Of**” in opposite direction***)**

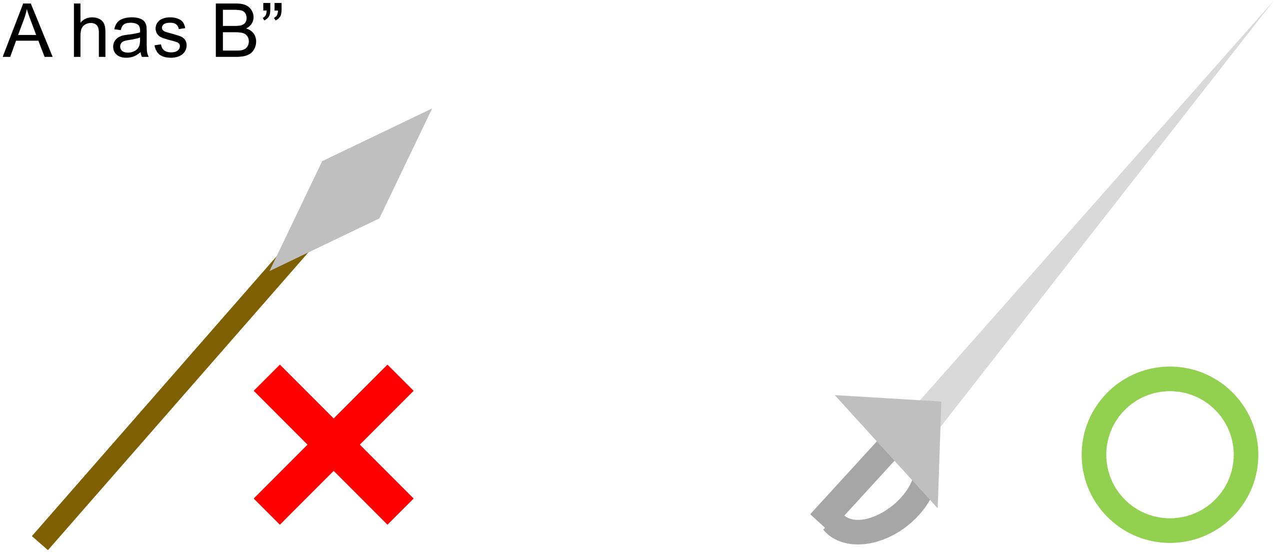


- An IntraMuralTeam is an aggregate of *(has)* 2 or more Students
- A Student *is-a-part-of* at most six Teams
- A Department has any number of Students
- A Student can belong to any number of Departments (e.g. double major)

Direction



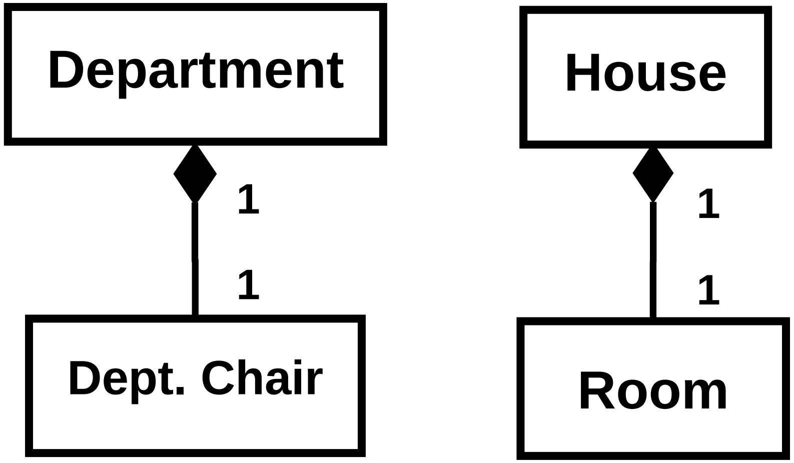
It is “A has B”



Composition

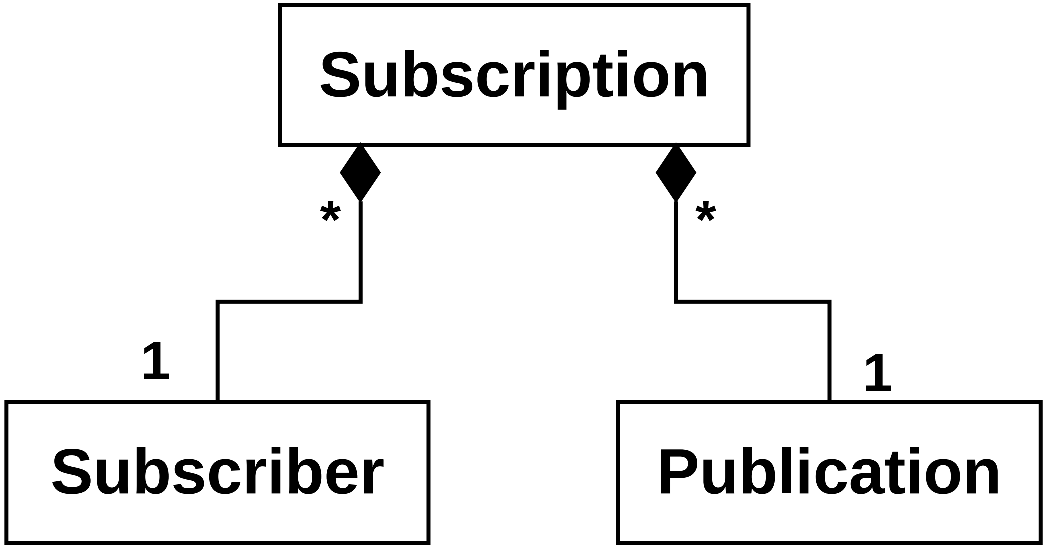
“Require” relationship

- without whole, the part can’t exist
- B live and die with A



Composition Example

A subscription can’t exist without both a Subscriber and a Publication (e.g., a Magazine)



Aggregation vs Composition

Aggregation: child exist independently

Composition: child cannot exist without parent

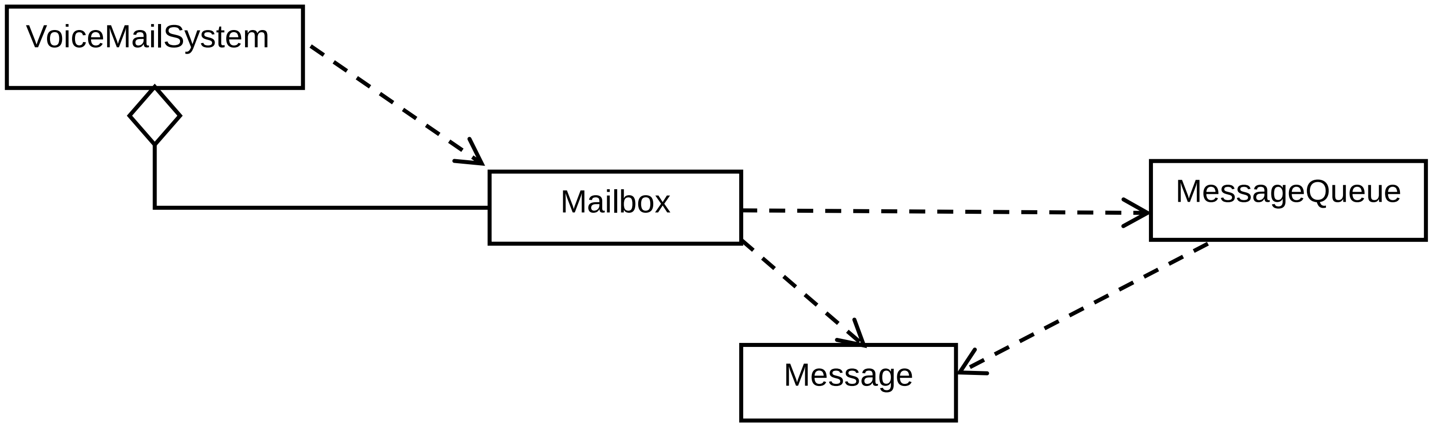
Note:

- Sometime no need to match common sense
- It depends on your program design
- Better to match common sense

Dependency

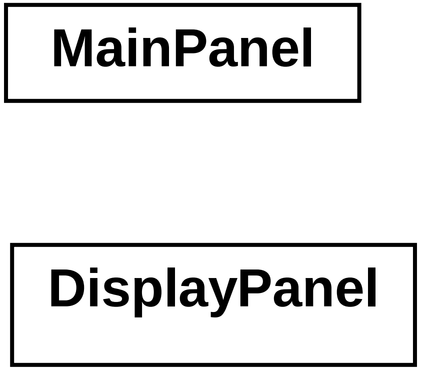
“**Uses**” (or “**knows about**”)

- Indicates coupling using dotted arrow
- Any relationships imply dependency
- Desirable to minimize dependencies



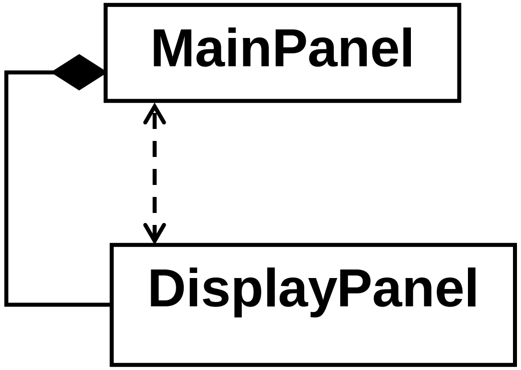
UML Example

```
public class MainPanel {
     ...
}

public class DisplayPanel {
     ...
}
```



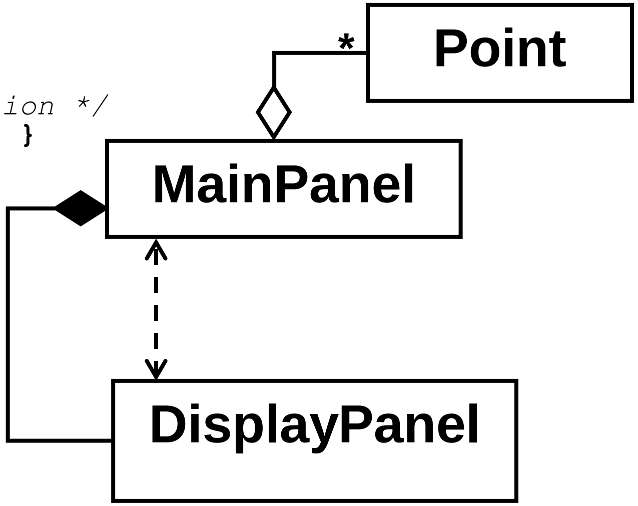
UML Example

```
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 ;
   }
   ...
}
```



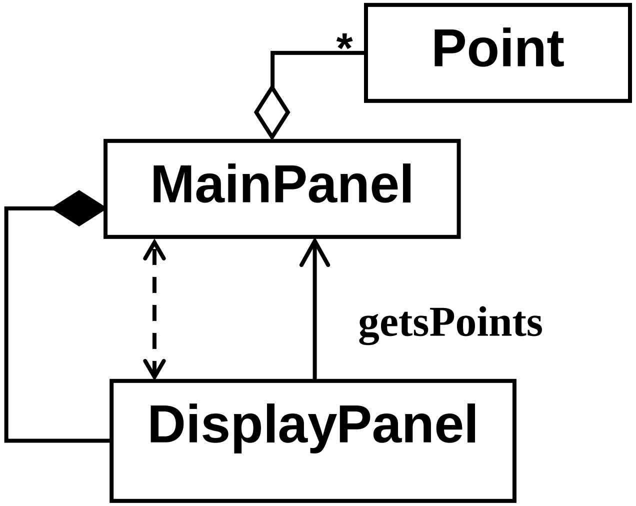
UML Example

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

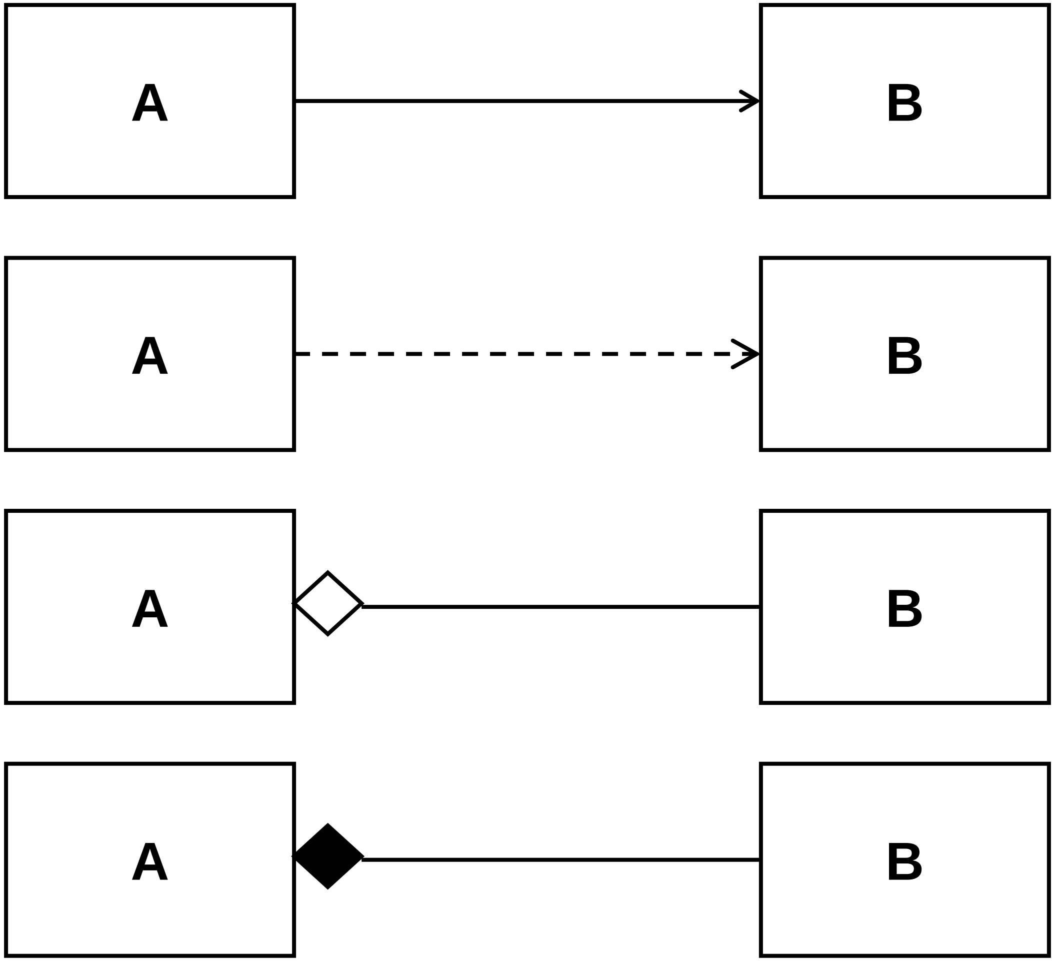


UML Example

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



Summary



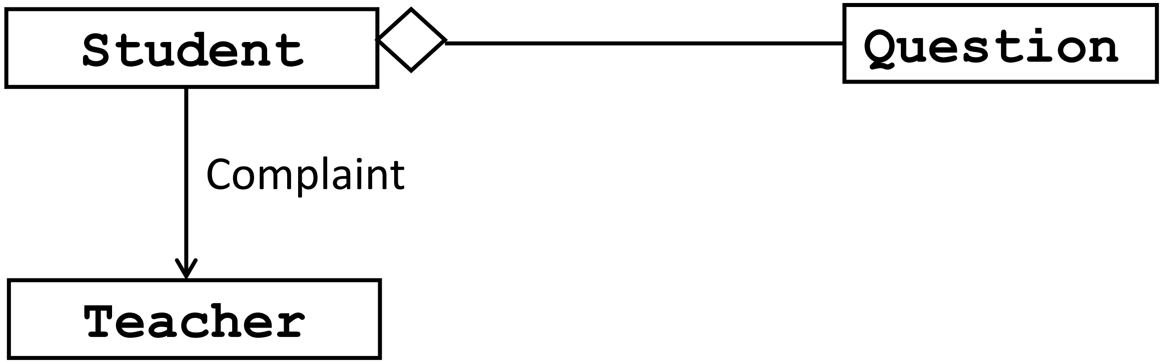
Association
Can be 2-direction
Message

Dependency
Can be 2-direction
Know/used

Aggregation
Single direction
Has a

Composition
Single direction
requires

Questions?



Free to Go!