Chapter 2: Relationship between classes using UML

Dr. Safwan Qasem

King Saud University
College of Computer and Information Sciences
Computer Science Department

What is UML?

The OMG(*) specification states:

"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.

The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

(*) Object Management Group (http://www.omg.org)

Goals of UML ?

The primary goals in the design of the UML were:

- Provide users with a ready-to-use, expressive visual (graphical) modeling language so they can develop **and exchange** unambiguous meaningful models.
- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language (defined semantics for each graphical symbol).
- Encourage the growth of the OO tools market.

UML Diagrams

- UML defines several diagrams. Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction.
- UML diagrams commonly created in visual modeling tools include:
  - **Class Diagram** models class structure, relationships and contents
  - **Use Case Diagram** displays the relationship among actors and use cases.
  - **Interaction Diagrams**
    - **Sequence Diagram** displays the time sequence of the objects.
    - **Collaboration Diagram** shows the sequence of messages between objects.
  - **State Diagram** displays the sequences of states that an object of an interaction goes through.
  - **Activity Diagram**
  - **Physical Diagrams**

Some UML tutorials

- http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/index.htm
- http://www.sparxsystems.com.au/uml-tutorial.html

Chapter Outline

1. UML Object Models: Classes
2. Association
3. Composition
4. Aggregation
5. Examples

UML Object Model : Classes

Class Name

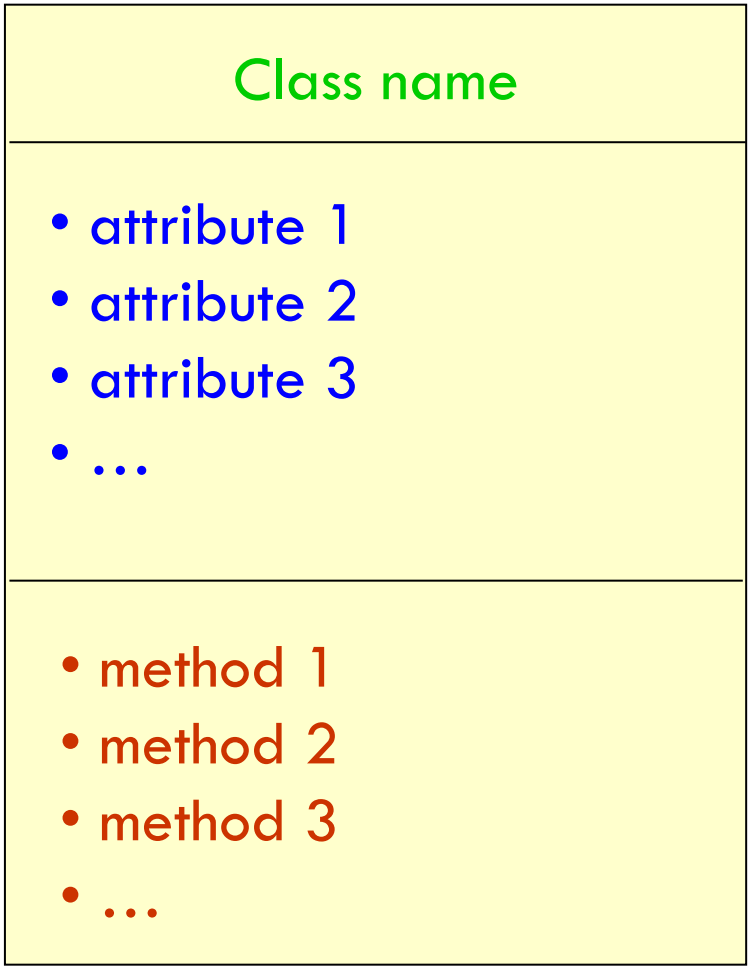
Should be descriptive of the class and capitalized in the first letter

Attributes

The named properties of the class. Can be typed, possibly with default values

Methods

Services offered by the class. Methods can be typed e.g. parameter types and return types specified.

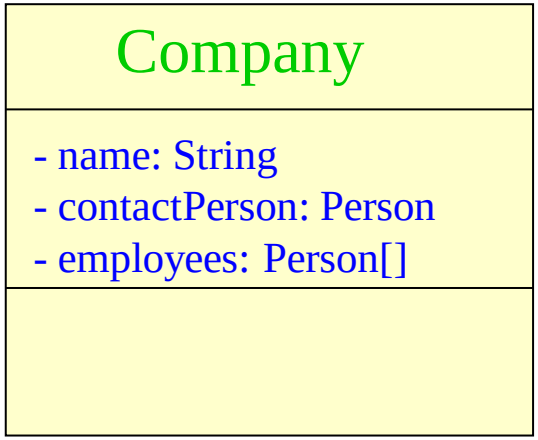


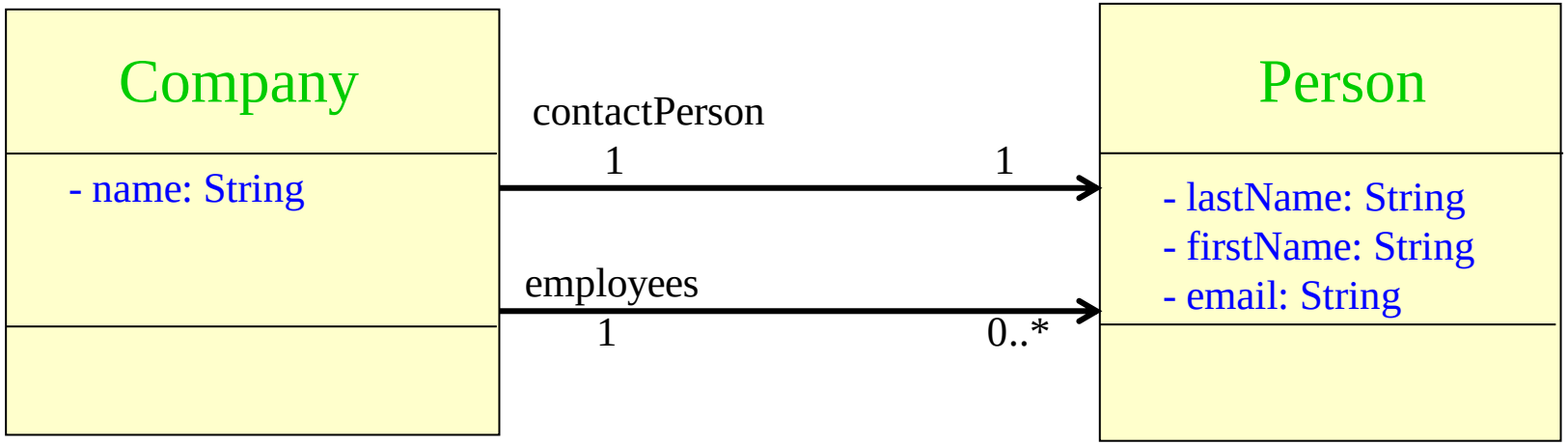
UML Object Model : Association

- UML diagrams show a collection of named boxes - indicating classes or types of object.
- The boxes have lines connecting them called links.
- Each link is called an **Association**.
- An association models some relationship or connection between the classes.

UML Object Model : Association

Classes can contain references to each other. The *Company* class has two attributes that reference the Person class. Although this is perfectly correct, it is more expressive to show the attributes as associations.





UML Object Model: Association

- The two associations have the same meaning as the attributes in the first version of the *Contact* class.
- The first association (the top one) represents the old *contactPerson* attribute. There is one contact person in a single Company.
- The ***multiplicity*** of the association is one to one meaning, that for every *Company there* is one and only one *contactPerson* and for each *contactPerson* there is one Company.
- In the bottom association there are zero or many employees for each company. Which means that the array *employees* may be empty.

UML Object Model: Association

Multiplicities can be anything you specify. Some examples are shown:

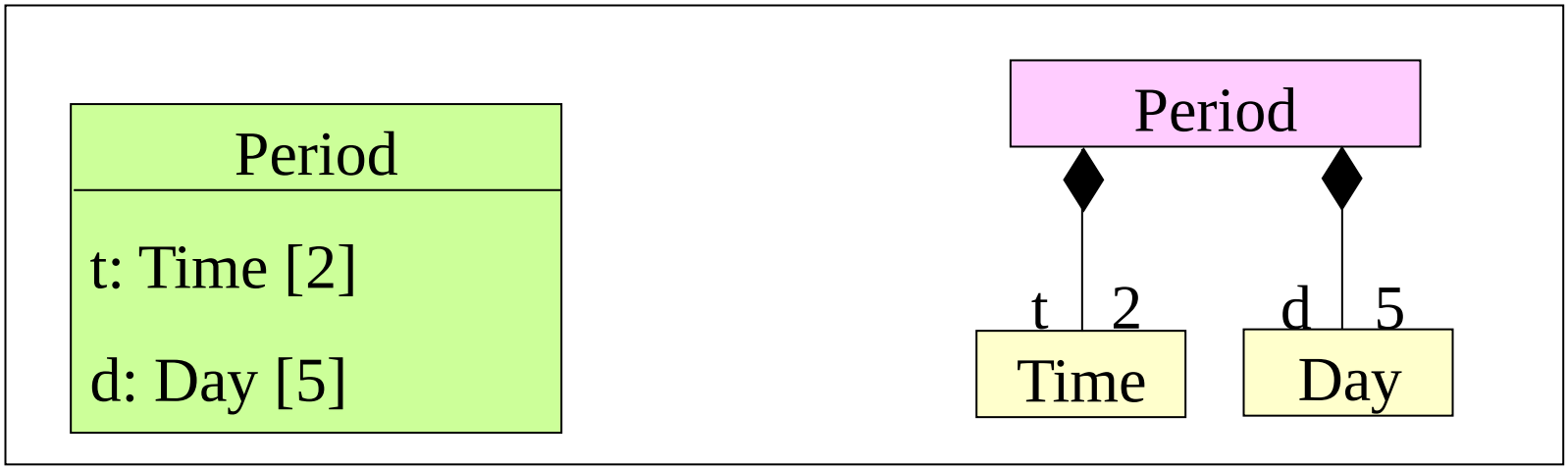
0	zero
1, 5	One or five
0..4	Zero, one, two, three, or four
1..*	one or many
1..2, 10..*	one, two or ten and above but **not** three through nine

Aggregation and Composition

- The association between two classes can have different meanings:
  - **Composition**: The class that contains an object as one of its component can possess that object and be allowed to decide if such an object should exist or not
    - **Example:** One course can have several sections. If the course is deleted, all its sections should be deleted.
  - **Aggregation**: The object contained in a class can have an independent existence, and not dependant on the container class.
    - **Example:** Students are part of a section, but if the section is deleted, the students continues to exist as they may be also part of other sections.
- UML provides several notations to express more details about the type of association between two classes.

Composition

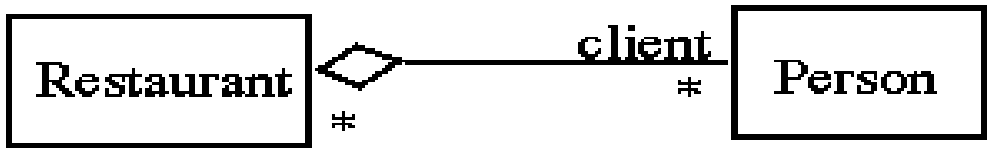
- *UML* provides several notations that can express the physical construction of a class. The filled in diamond is often used when a class contain other objects within them as parts or components. The *composition* association is represented by the solid diamond.
- Here are two examples: Period is *composed* of *Time and Day*.



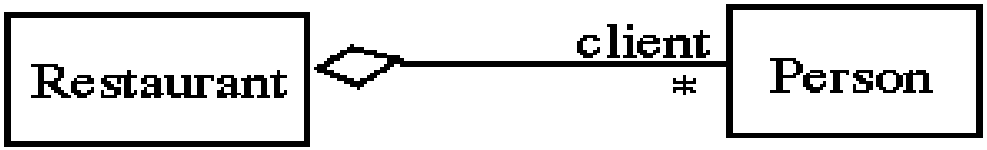
- We can use the dark diamond to indicate that the class possesses the components in the sense of controlling whether they exist of not. The filled in diamond indicates that the deletion of an object may delete its components as well.

Aggregation

- We can also show that a class *has* some parts and yet they have an independent existence. Example: In the computer world a page on the world Wide Web can use a hypertext reference to point to another resource -- deleting the page does not effect the other page. This association is called aggregation and is represented by the **hollow diamond**.
- In this example, a Restaurant will have a number of clients who are People and the clients exist whether or not they are clients of the Restaurant:



Each Person eats at any number of Restaurents

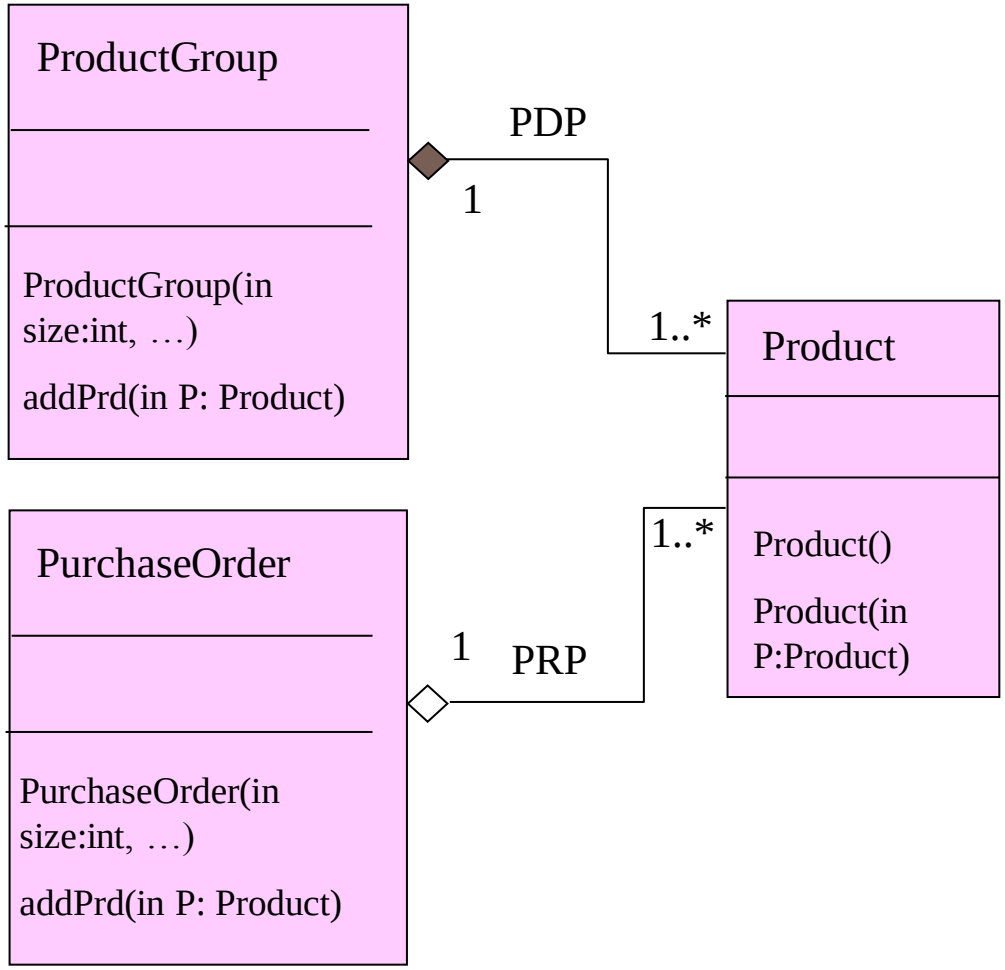


Each Person eats at one Restaurent.

Aggregation / Composition Example 1

ProductGroup is *composed* of *Products*. This means that if a *ProductGroup* is destroyed, the *Products* within the group are destroyed as well.

PurchaseOrder is an *aggregate* of *Products*. If a *PurchaseOrder* is destroyed, the *Products* still exist.



Aggregation / Composition Example 1

How to Implement Aggregation?

```
public class PurchaseOrder          {
    private Product PRP [];
    // number of current product in the array.
    private int nprp;
    …..
    public PurchaseOrder (int size, …)
    {
            PRP = new Product[size];
            nprp=0;
            ….
    }
     ……
}
```

How to Implement Composition?

```
public class ProductGroup          {
    private Product PDP [];
    // number of current product in the array.
    private int npdp;
    …..
    public ProductGroup (int size, …)
    {
            PDP = new Product[size];
            npdp=0;
            ….
    }
     ……
}
```

Aggregation / Composition Example 1

Aggregation:

How to add a new product?

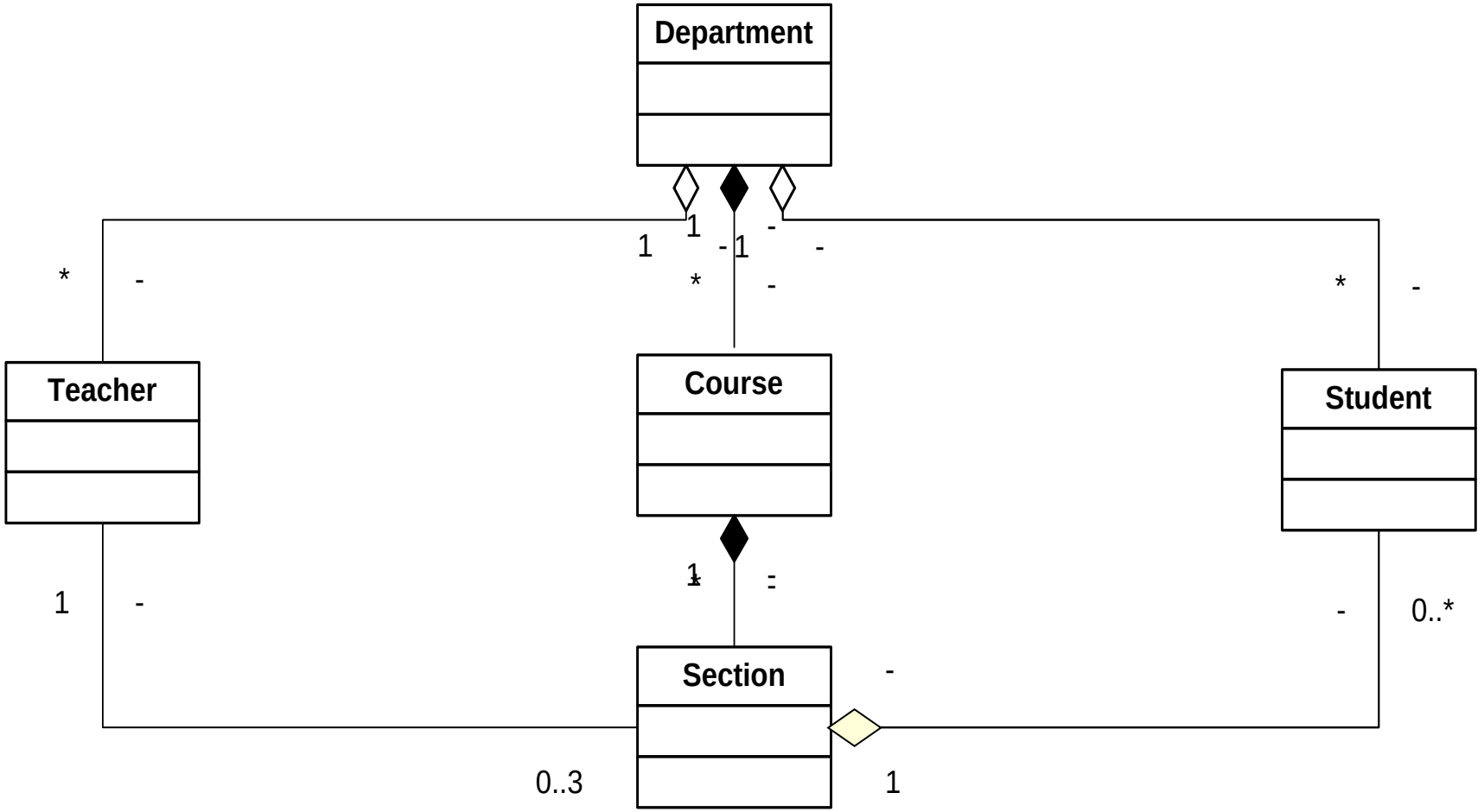
```
public class PurchaseOrder    {
    private Product PRP [];
    private int nprp; // number of current
                      product in the array.
    …..

    public void addPrd (Product P)  {
            PRP[nprp] = P;
            nprp++;
    }
    ……
}
```

Composition:

How to add a new product?

```
public class ProductGroup      {
    private Product PDP [];
    private int npdp; // number of current
                      product in the array.
    …..

    public void addPrd (Product P)  {
            PDP[npdp] = new Product(P);
            nprp++;
    }
    ……
}
```

Aggregation / Composition Example 2



Aggregation / Composition Example 2

The following Java code shows just how the links between the different objects can be implemented in Java. Note that this code just shows the links. It does not show constructors, or any other methods what would be required to actually use these objects.

```
/*   * Student.java -    */

public class Student
{
    private String name;
    private String id;

    public void copyStudent(Student st)
    {
            name= st.name;
            id= st.id ;
    }
    // ...
}
```

```
/*   * Section.java -     */
public class Section
{
    private String sectionName;
    private int capacity;
    private int currentNbStudents;
    private Student[ ] stud;
    ….
    public void addStudent(Student s)
    {
            stud[currentNbStudents]=s;
            currentNbStudents ++;
    }
    // ...
}
```

Aggregation / Composition Example 2

/* Course.java */

```
public class Course
{
    private String
    courseName;
    private int nbSection;
    private Section[ ] sect;

    // ...
}
```

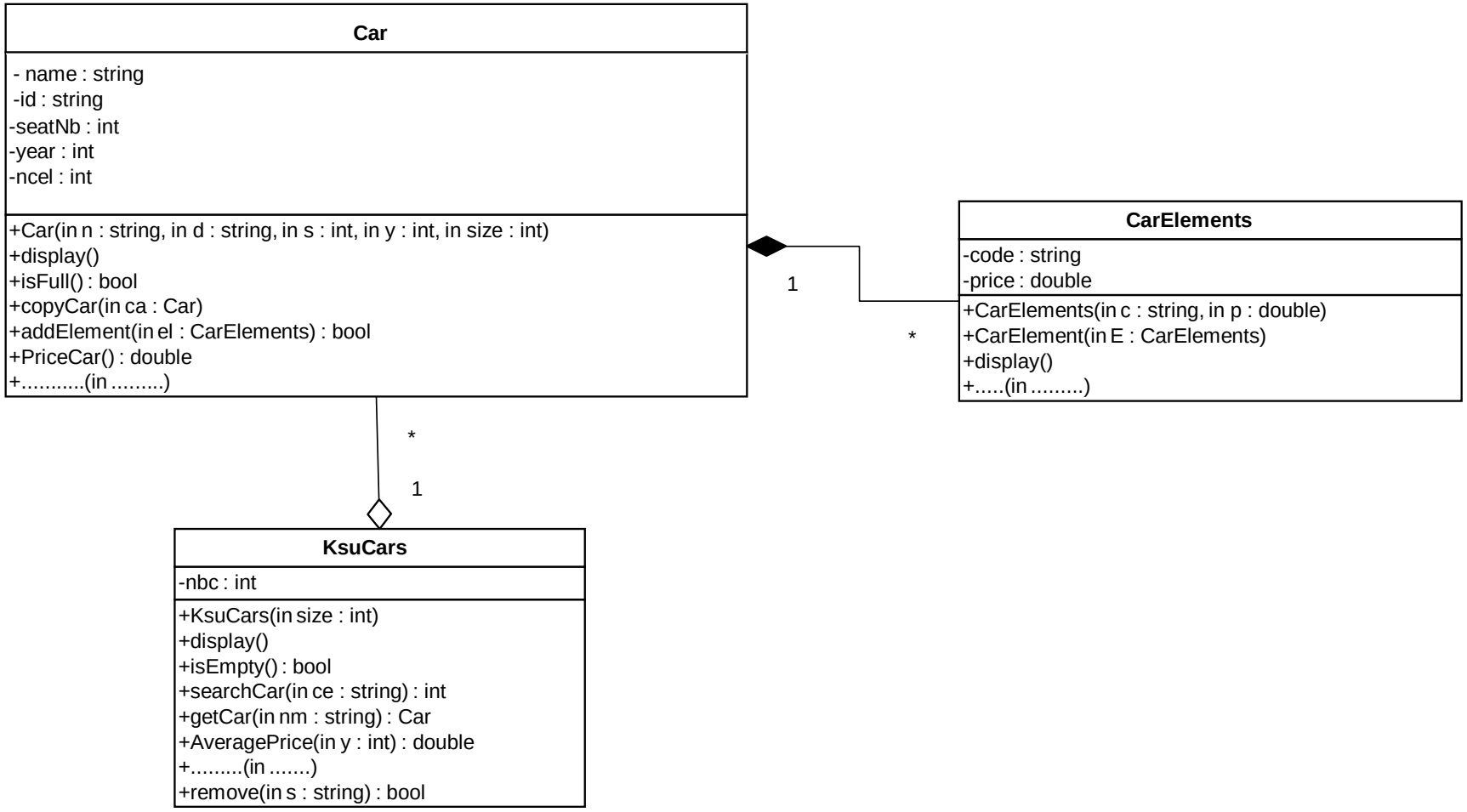
/* Teacher.java */

```
public class Teacher
{
    private String
    teacherName;
    private String Id;
    private Section[3] sect;

    // ...
}
```

/* Department.java */

```
public class Department
{
    private String
    departName;
    private Student[ ] stud;
    private Course[ ] csc;
    private Teacher[ ] teach;

    // ...
}
```

Aggregation / Composition Example 3



Question: **Implement all the classes with all their methods using the following descriptions.**

Description of the different classes:

Class CarElements:

- ✓ *The method* **display ()** displays the code and the price.
- ✓ **+ …….. (in ……..)** : if you need an other methods in this class you can add it.

You can't add another constructor.

Class Car:

- name
- id
- seatNb : *Number of seats*
- year : *Production year of car*
- ncel : *number of CarElements object currently in an object of the class Car.*
- ***And other attribute(s) deduced from the UML diagram.***

- ✓ **display ():** Displays all the attributes of an object Car.
- ✓ **addElement (CarElements el)**: This method receives a CarElements object and adds it to the Car object.
- ✓ **priceCar()**: Returns the sum of the CarElements price in an object of the class Car.

+…….. (in ……..) : if you *need an other methods in this class you can add it.*

Class KsuCars:

- nbc : *number of Car currently in an object of the class KsuCar.*
- ***And other attribute(s) deduced from the UML diagram.***

- ✓ **display ():** Displays all the attributes of an object KsuCars.
- ✓ **search (String ce):** This method receives a String representing the *name* of a Car object and returns the array index of the car object.
- ✓ **getCar (String nm):** This method receives a String representing the *id* of a Car object and returns the Car object if it's exist.
- ✓ **removeCar (String s)**: Removes a Car according to its name. It will return a value *true* if the operation has been completed successfully, or *false* if not.
- ✓ **AveragePrice(int y):** Calculates the average price of all car in an object of class KsuCars that produced after the year **y**.
- ✓ *+…….. (in ……..) : if* you *need an other methods in this class you can add it.*