UML
Class Diagrams

Overview of Class Diagrams
&
Examples

UML Class Diagram

- A class diagram is a static model of a system (mainly software, but can include other types of components.
  - It shows the structure of the software in terms of the constituent classes and how each class is related to other classes.
  - It gives a static view of the system
    - As opposed to a dynamic view, which describes what the software does when it runs, a class diagram provides a static view, which describes the classes that make up the software.
- Main purpose: to *communicate* the structure of a software application.
  - Communicate the static structure of software to others for their review and understanding
  - Can be used to generate source code, in a limited way.
- Flexible: allows showing only pertinent information.
- Detailed *semantics*.
- *Extensible*
  - It can be *Extended* to show other similar components, such as an interface and a TCP connection between classes.

Class Diagram Components

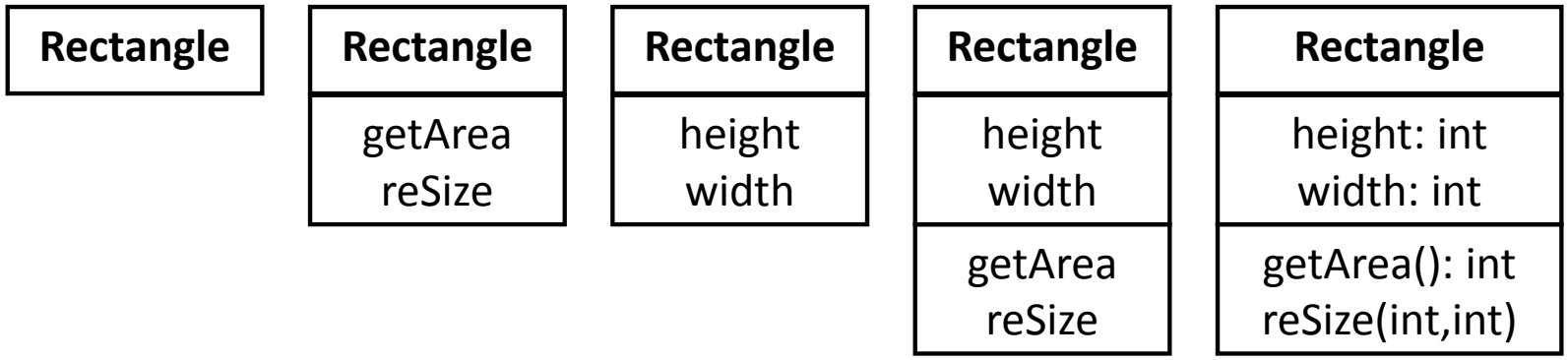
- The main symbols shown in class diagrams are:
  - Class
    - A class represents the blueprint (template) of its objects.
  - Fields (Attributes, Variables or Constants)
    - A field represents the state of the class and its instances.
  - Behaviour (Operations or Methods)
    - A behavior represents an operation performed by the class and its instances.
  - Associations
    - An association represents a relationship between two classes.
  - Generalizations
    - A generalization groups classes into an inheritance hierarchy.

UML Representation of Classes

- A class is simply represented as a box with the name of the class inside.
- The diagram may also show the attributes and/or operations (fields and behaviour).
- The complete signature of an operation is:

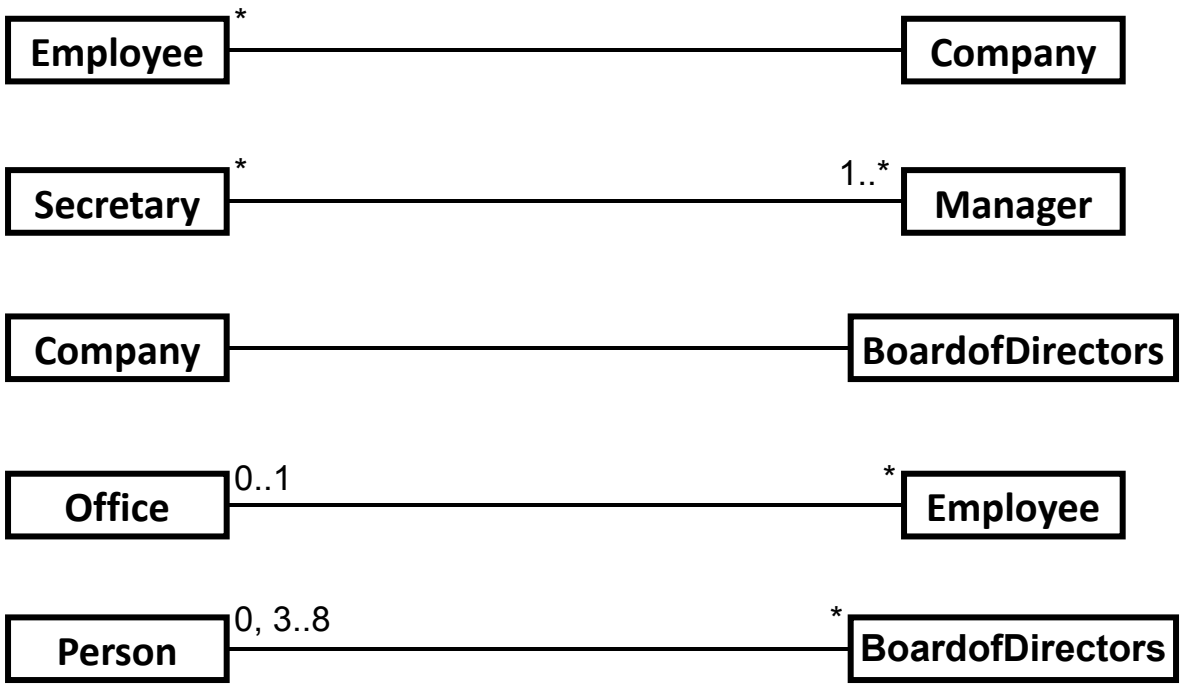
  operationName(parameterName: parameterType …): returnType

- Examples:



Associations and Multiplicity

- An *association* shows how classes are connected to each other:
  - Symbols indicating *multiplicity* are shown at each end of the association.



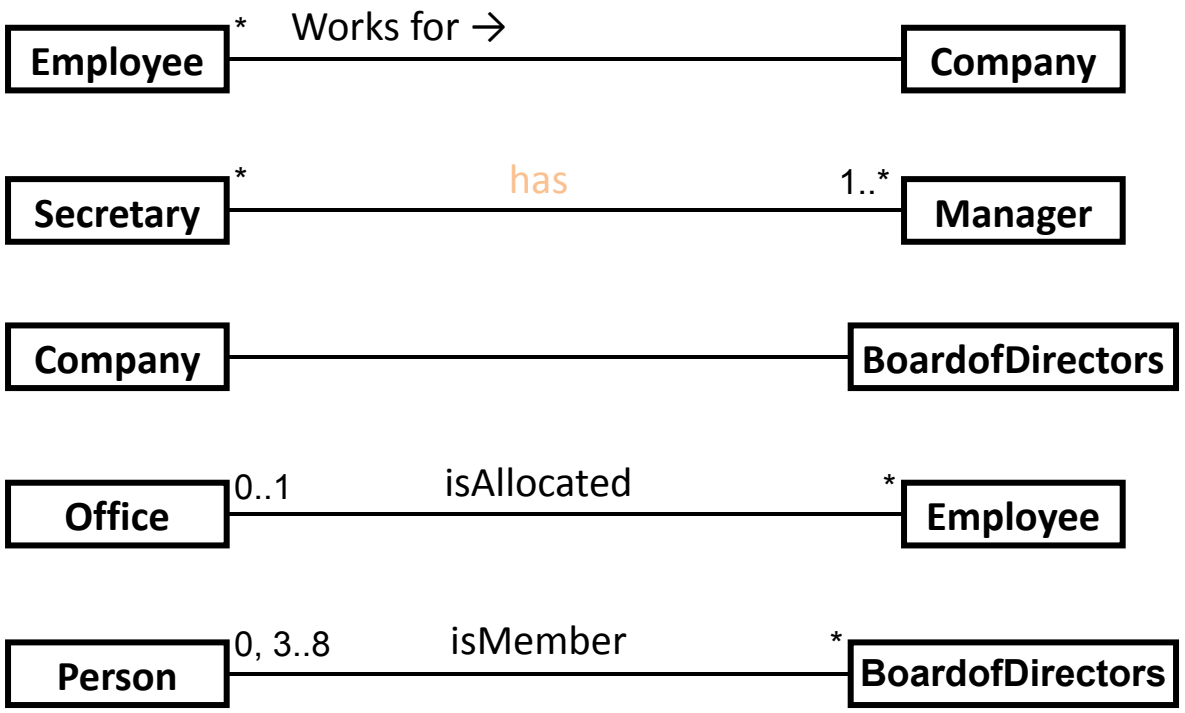
LEGEND

x..* (the range from x to many).
x..y (the range from x to y).

Labelling Associations

- Each association can be labelled, to make explicit the nature of the association:

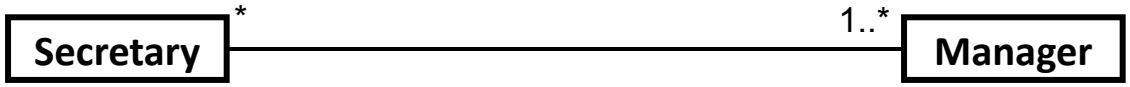
Labelling Associations



Analyzing & Validating Associations

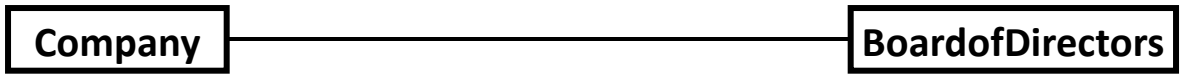
- **Many-to-one**
  - A company has many employees.
  - An employee can only work for one company.
    - This system is not capable of processing information about more than one company per employee.
  - A company can have zero employees.
    - E.g. a ‘shell’ company.
  - It is not possible to be an employee unless you work for a company.

Analyzing & Validating Associations



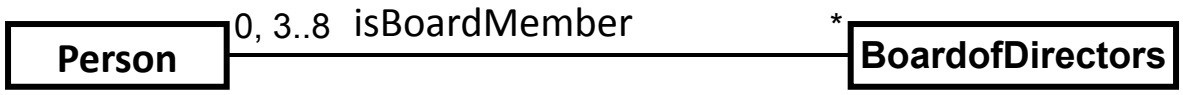
- **Many-to-many**
  - A secretary can work for many managers.
  - A manager can have many secretaries.
  - Secretaries can work in pools.
  - Managers can have a group of secretaries.
  - Some managers might have zero secretaries.
  - It is not possible for a secretary to have zero managers.

ANALYZING & VALIDATING ASSOCIATIONS



- **One-to-one**
  - For each company, there is exactly one board of directors.
  - A board is the board of only one company.
  - A company must always have a board.
  - A board must always be of some company.

ANALYZING & VALIDATING ASSOCIATIONS



- **Many-to-many**
  - There can be zero people on a board of directors.
  - There can be three to eight people on a board of directors.
  - A person plays the role of a “board member”.
  - A person can be a member of more than one board.
  - A person can exist without being a member of any board.

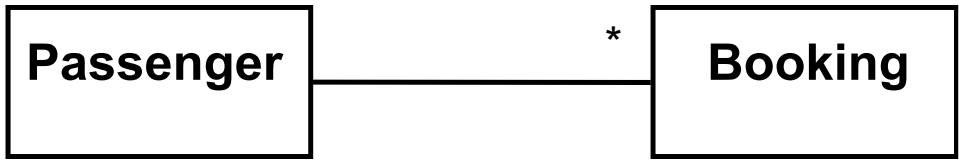
Ex: Passenger Reservation System (1)

- From the requirements of the problem, we are given:

- A Booking is always for exactly one passenger:
  - Cannot have a booking without a passenger.
  - A booking could never involve more than one passenger.

- A Passenger can have any number of Bookings:
  - A passenger could have no bookings at all.
  - A passenger could have more than one booking.

- Corresponding UML model:

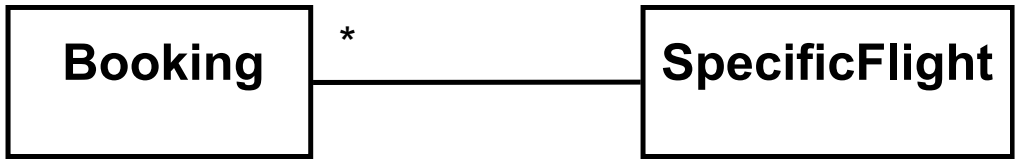


Ex: Passenger Reservation System (2)

- From the requirements of the problem, we are given:

- A Booking is always for exactly one SpecificFlight:
  - No booking with zero specific flights.
  - A booking could never involve more than one specific flight.

- A SpecificFlight can have any number of Bookings:
  - A specific flight could have no bookings at all.
  - A specific flight could have more than one booking.



Ex: Passenger Reservation System (3)

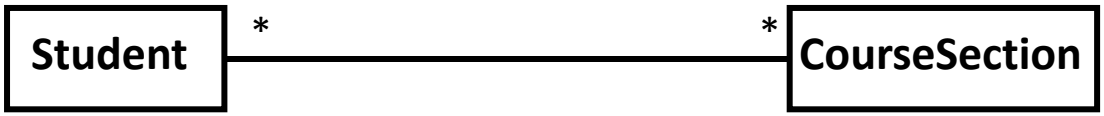
- Putting the previous two slides together:

- A Passenger can go on many different specific flights:
  - For every specific flight a passenger goes on, there is a unique booking associated with that specific flight.

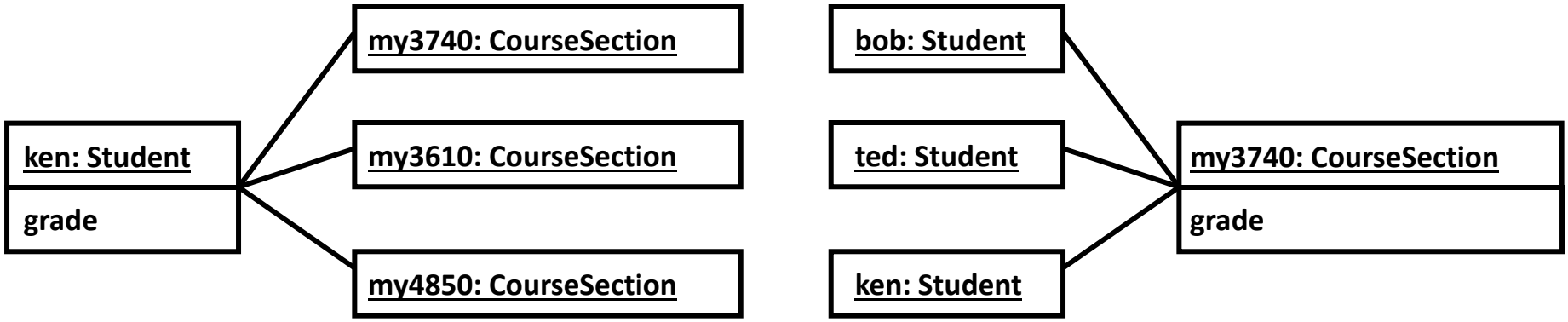
- A SpecificFlight can have any number of Passengers:
  - For every booking the specific flight has, there is a unique passenger associated with that booking.

Association Classes:
The Problem

- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes; for instance, in which class can grade be placed?



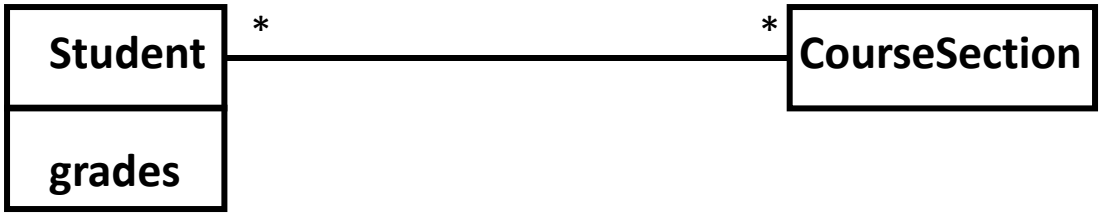
- Consider some objects, as depicted in the following instance diagrams:



- Conclusion: the grade attribute cannot be put in either of the above two classes.

Association Classes
(1st Attempt at a Solution: Not The Best Solution)

- The temptation is to create a grade array, such as the following:



- If implemented like this, the resulting code would be complex and difficult to read and understand:

```
Student ken = new Student();
. . .
ken.grades[2] = 'B';
```

  - Upon reading this code, it is not clear that the grade `ken.grades[2]` corresponds to the course section 4850.

Association Classes (The Best Solution)

- Solution: add another type of class, called an association class:

- A Student can have many registrations.
- Each registration is associated with one course section.
- A student may know the grade for each course by referring to the registration for that course.
- A course section can have many registrations.
- Each registration is associated with one student.
- A course section may know the grade for each student enrolled in the course by referring to the registration for that student.

Association Classes (The Best Solution)

- If implemented like this, the resulting code is easier to understand:

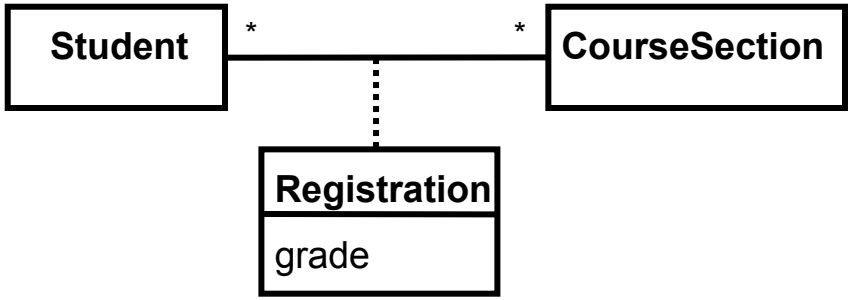
```
Student ken = new Student();
Registration kens3740Reg = new Registration();
. . .
ken.kens3740Reg.grade = 'A';
```

  - Reading this code, it is clear that `ken.kens3740Reg.grade` corresponds to course section 3740.

Association Classes
(Alternative Way of Drawing an Association Class)

- One way of depicting an association class:

- An equivalent way of depicting an association class:



- The equivalent way advantage: the association between Student and CourseSection is clearer.

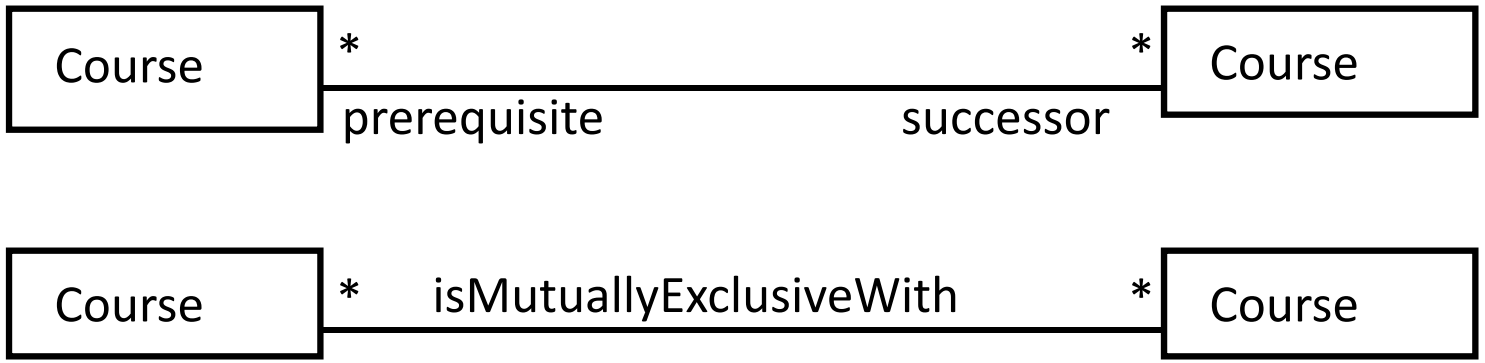
Reflexive Associations: The Problem

- How do you model a class that has relationships to other classes of the same type?
  - For example, a Course class may have prerequisite, successor, and mutually exclusive relationships to other Course classes.
  - Such as:
    - Course ECE 3730 has a prerequisite of COMP 1010 and ECE 3610.
    - Courses COMP 1010 and ECE 3610 have successor ECE 3730.
    - ECE 3730 cannot be taken with ECE 3740, i.e., they are mutually exclusive.

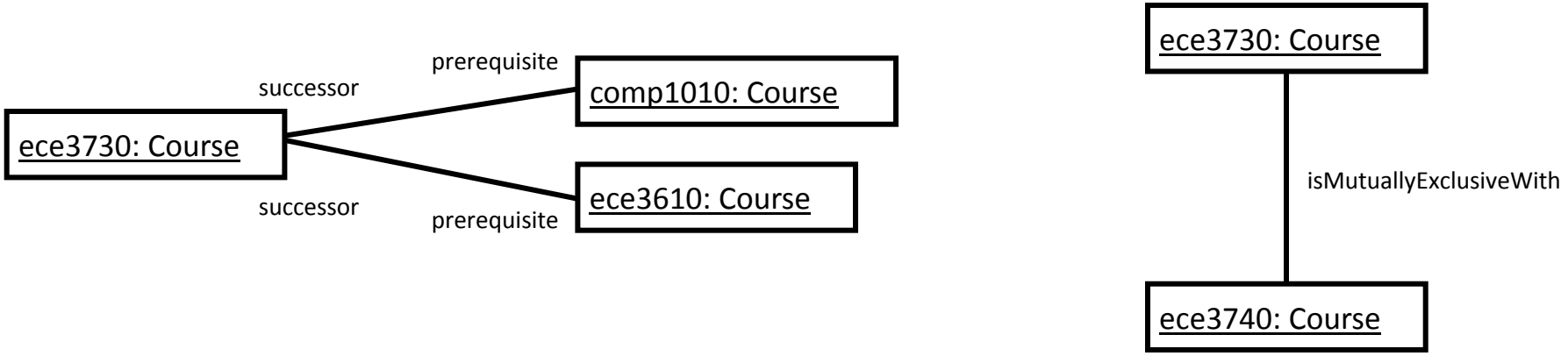
Reflexive Associations

(1st Attempt at a Solution: Incorrect Solution)

- Class diagrams:



- Possible instance diagrams:

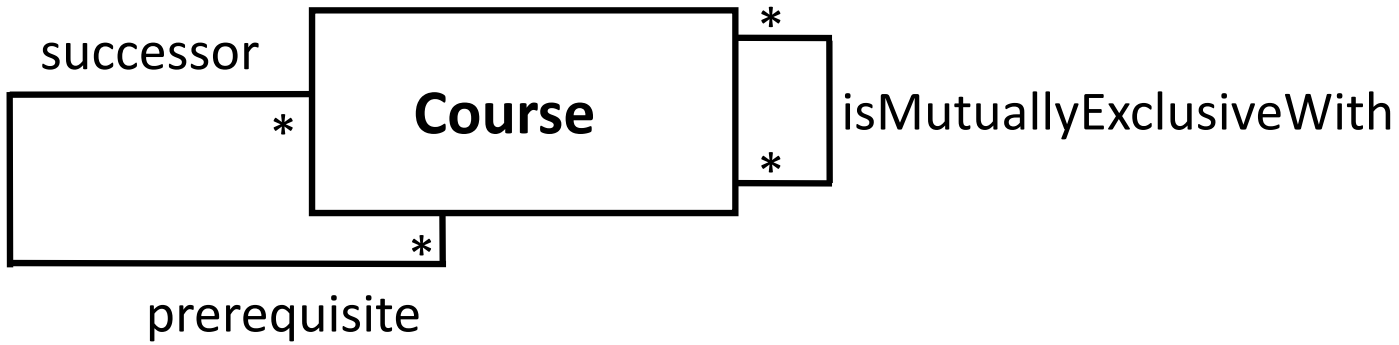


- **Incorrect** b/c the class diagram suggest there are four Course classes.

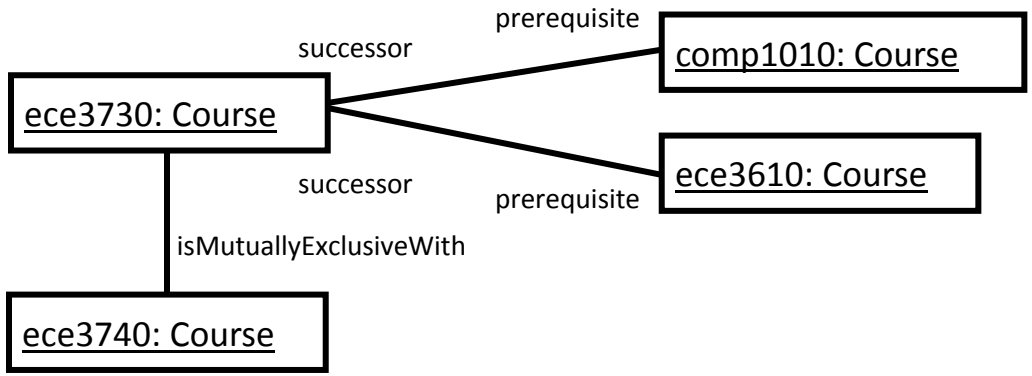
Reflexive Associations (The Solution)

- A reflexive association allows a class to connect to itself:

Class Diagram

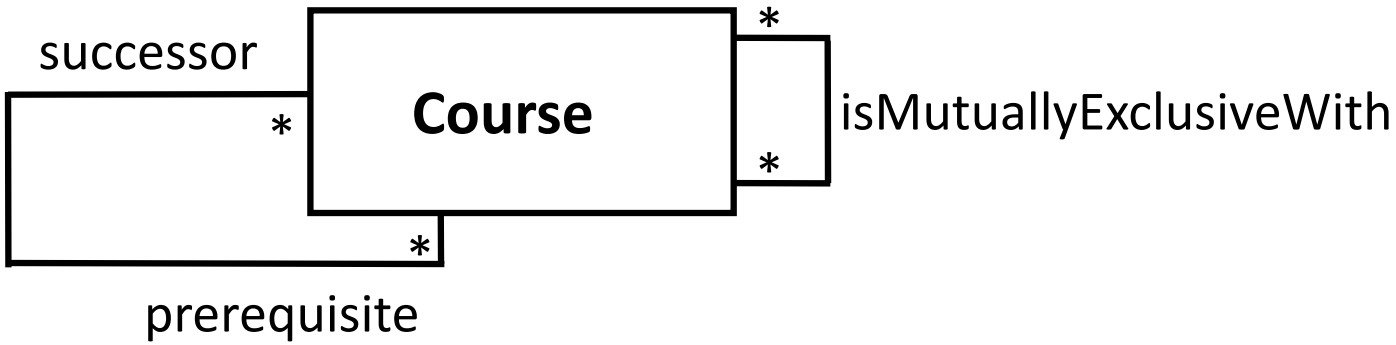


Instance Diagram



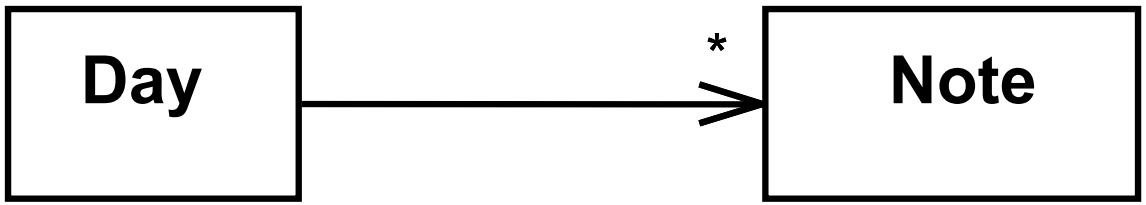
Reflexive Associations (The Solution)

- Exercise for students:
  - Where can the attributes prerequisite, successor, and isMutuallyExclusiveWith be stored?



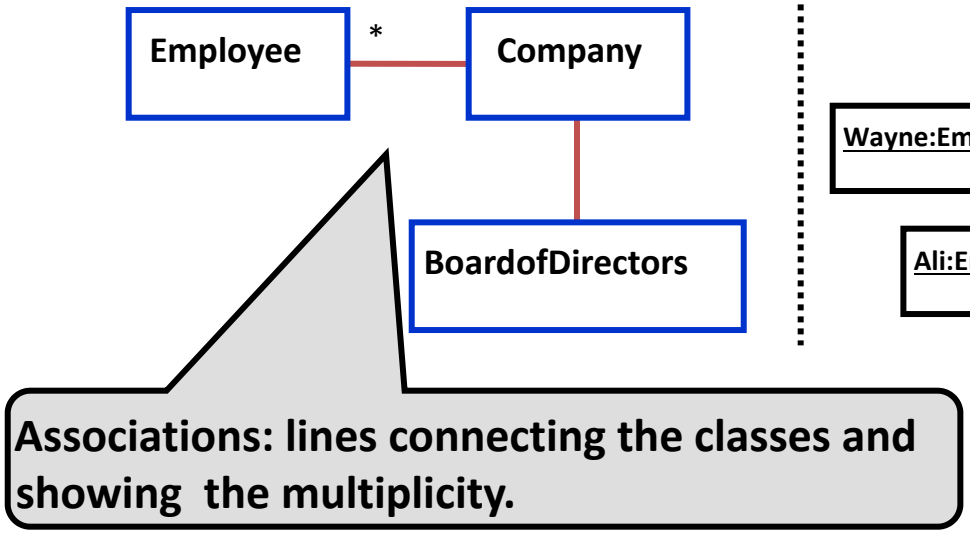
Directionality in Associations

- Associations are by default *bi-directional*.
- It is possible to limit the direction of an association by adding an arrow at one end.
- For example: the requirements for a “Day Planner” application state that the Day class needs a reference to a Note class, but the Note class does not need to reference the Day class.

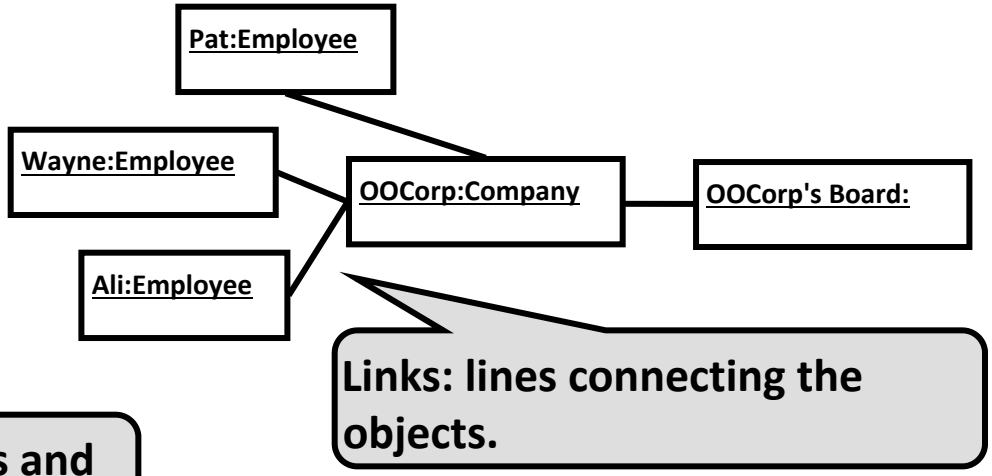


Instance Diagrams

Class Diagram



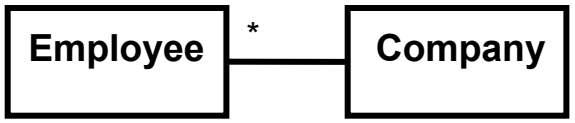
A possible Instance Diagram generated from the Class Diagram



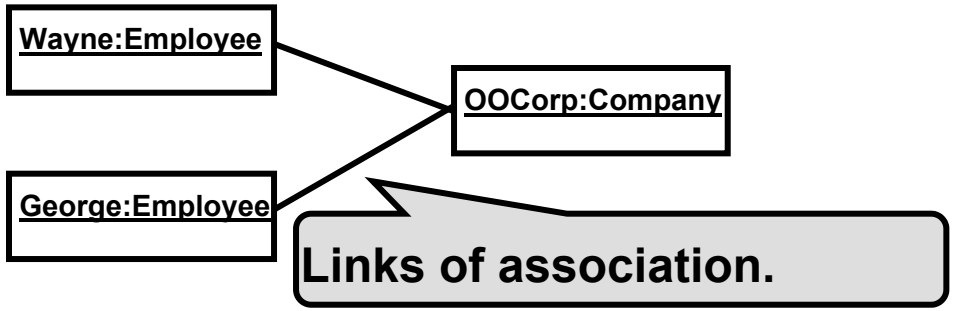
- An association is a blueprint that describes how objects can be linked with one another.
- A link is an instance of an association.
- An instance diagram never shows multiplicity.

Associations Versus Inheritance Hierarchies
(In Instance Diagrams)

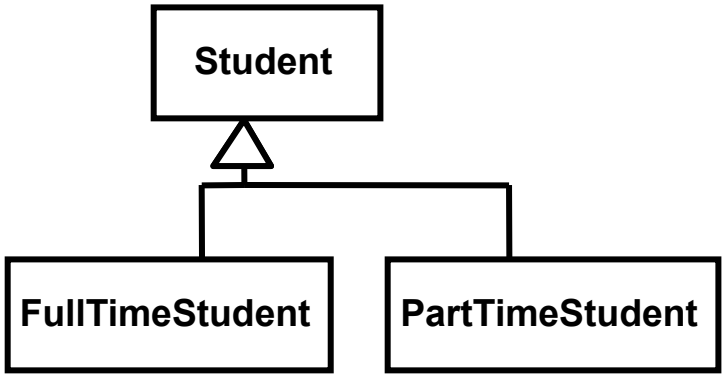
Class Diagram
(Showing Association)



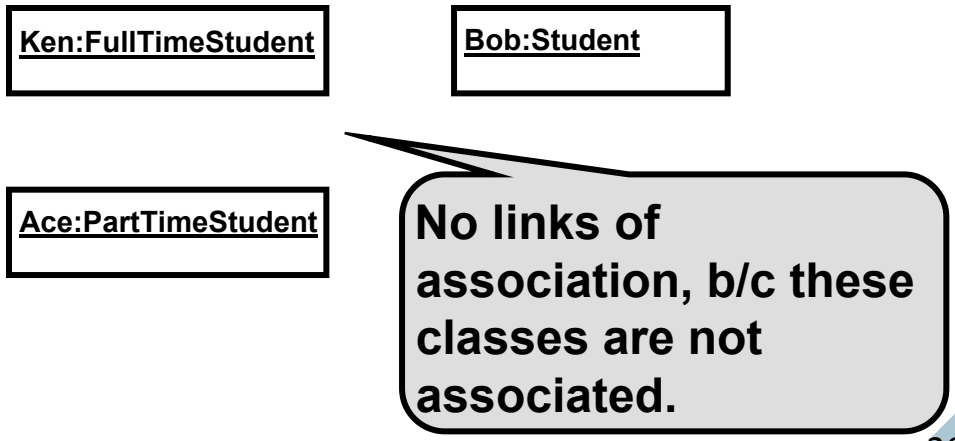
A Possible Instance Diagram



Class Diagram
(Showing Inheritance)



A Possible Instance Diagram



Associations Versus Inheritance Hierarchies
(In Instance Diagrams)

- Associations describe the relationships that will exist between *instances* at run time.
  - Associations appear in the form of links between objects in an instance diagram.

- Generalizations describe relationships between *classes* in class diagrams.
  - Generalization does not appear in instance diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class's superclasses .

TRANSLATING CLASS DIAGRAMS

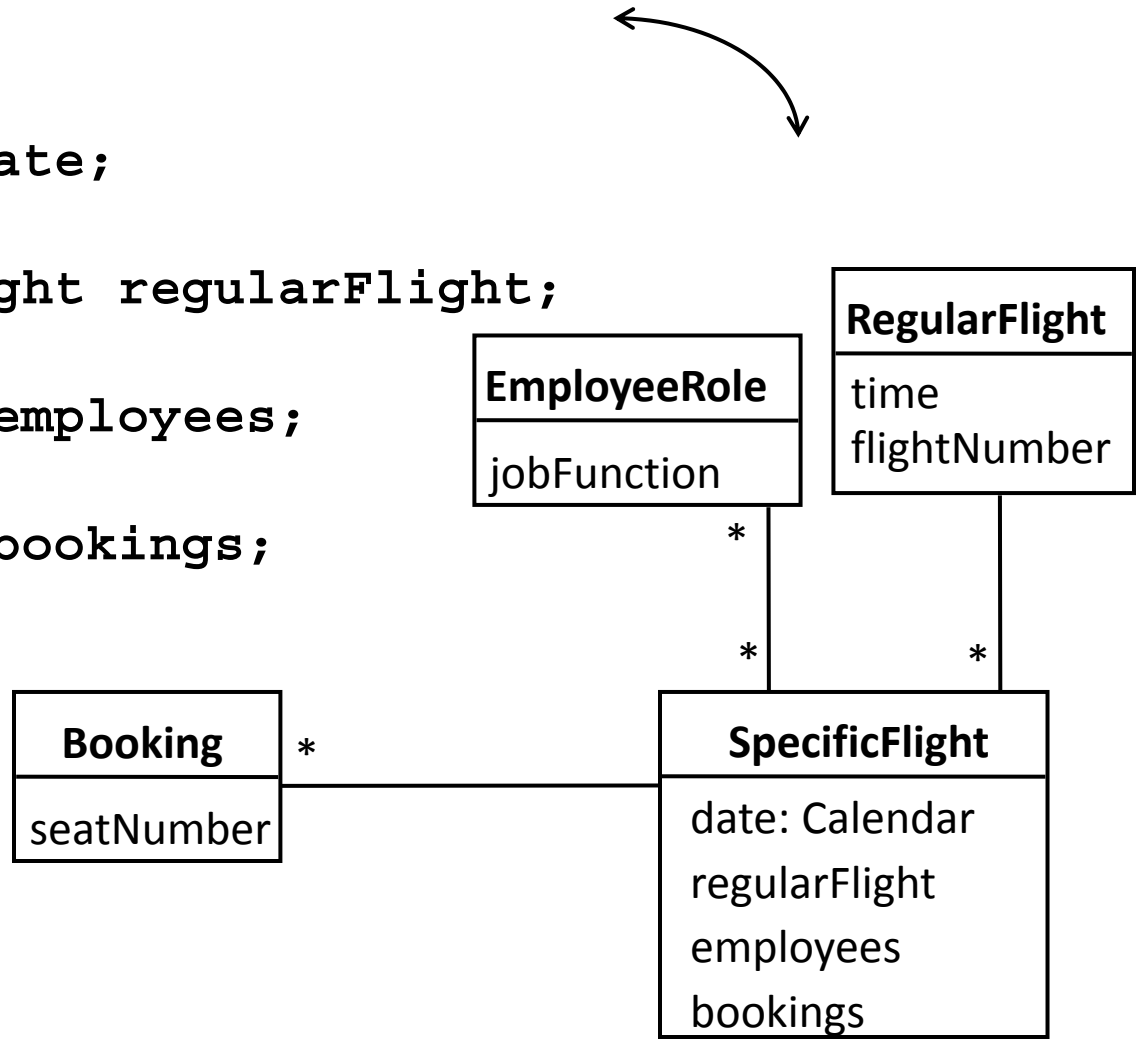
- Attributes are implemented as instance variables.
- Generalizations are implemented using `extends`.
- Interfaces are implemented using `implements`.
- Associations are implemented using instance (typically) variables:
  - Divide each two-way association into two one-way associations
    - so each associated class has an instance variable.
  - For a one-way association where the multiplicity at the other end is ‘one’ or ‘optional’
    - declare a variable of that class (a reference).
  - For a one-way association where the multiplicity at the other end is ‘many’:
    - use a collection class implementing `List`, such as `Vector`

SpecificFlight Attribute Translation

```
class SpecificFlight
{
  private Calendar date;

  private RegularFlight regularFlight;
  ...
  private ArrayList employees;

  private ArrayList bookings;
  ...
}
```



RegularFlight TRANSLATION

```
class RegularFlight
{
  private ArrayList specificFlights;
  ...



  public void addSpecificFlight(Calendar aDate)
  {
    SpecificFlight newSpecificFlight;
    newSpecificFlight = new SpecificFlight(aDate, this);
    specificFlights.add(newSpecificFlight);
  }
  ...
}
```