Object Oriented Programming Methodology

Module 5: Class Diagram

Faculty In-charge:

Pragya Gupta

E-mail: pragya.g@somaiya.edu

Contents

- Class Diagram

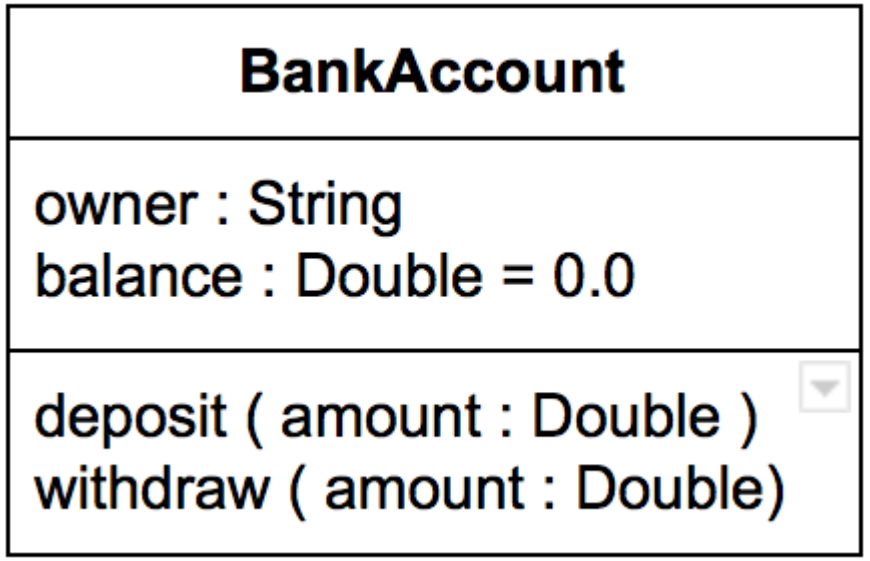
What is a Class Diagram?

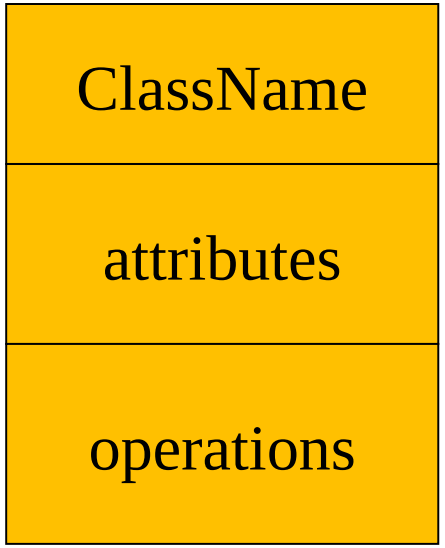
- Suppose you have to design a system. Before implementing a bunch of classes, you'll want to have a conceptual understanding of the system
  - What classes do I need?
  - What functionality and information will these classes have?
  - How do they interact with one another?
  - Who can see these classes?
- That's where class diagrams come in
- Class diagrams are a neat way of visualizing the classes in your system *before* you actually start coding them up
- They're a static representation of your system structure

Class Representation in UML

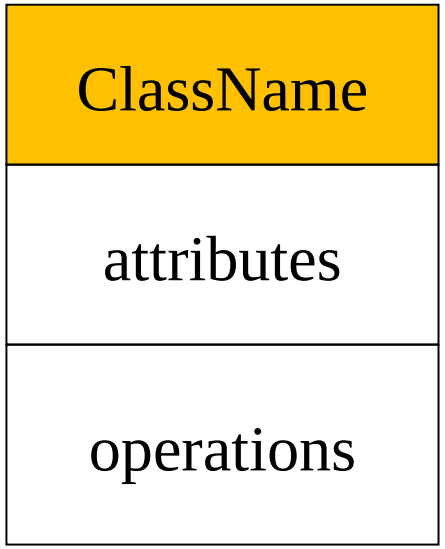
- A class is represented as a box with 3 compartments.

- The uppermost one contains the class name
- The middle one contains the class attributes
- The last one contains the class methods



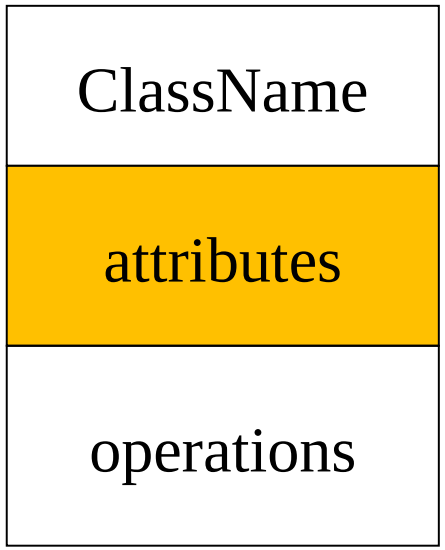


Class Names

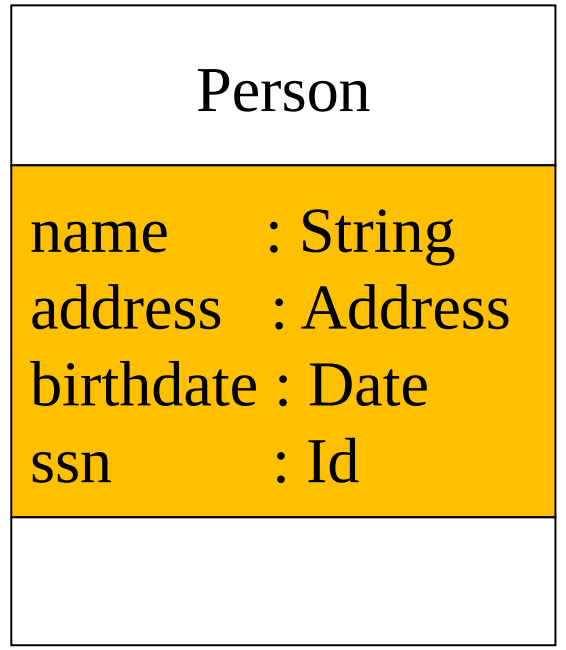


- The name of the class is the only required tag in the graphical representation of a class
- It always appears in the top-most compartment

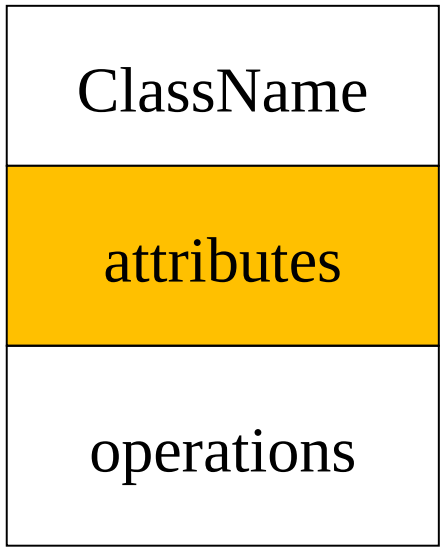
Class attributes



- An *attribute* is a named property of a class that describes the object being modeled
- In the class diagram, attributes appear in the second compartment just below the name-compartment



Class attributes (Contd.)

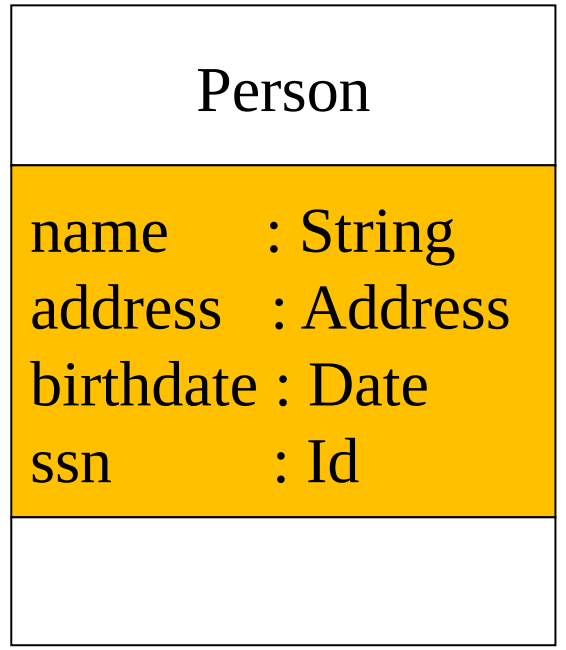


Attributes are usually listed in the form:

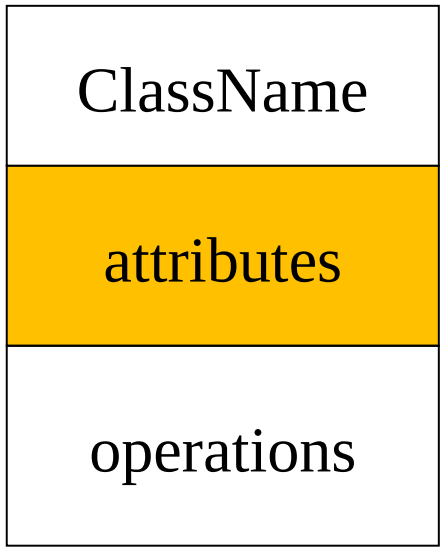
attributeName : Type

- A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist.
- For example, a Person's age can be computed from his birth date
- A derived attribute is designated by a preceding '/' as in:

/ age : Date

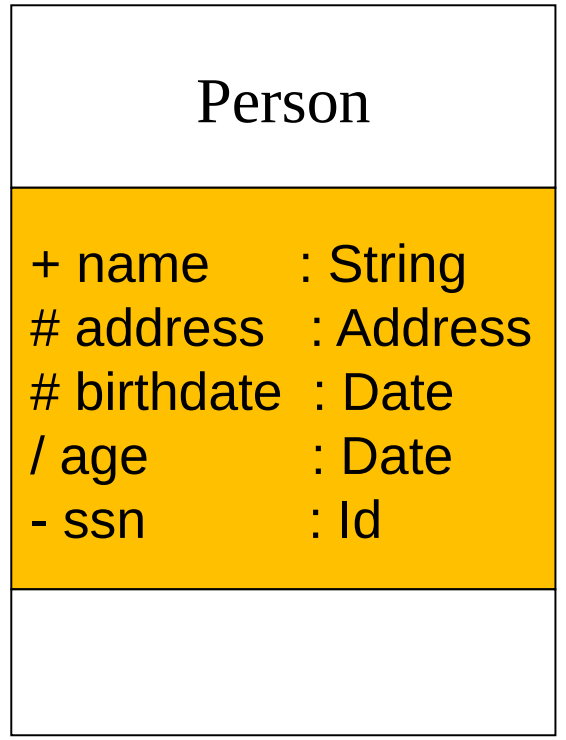


Class attributes (Contd.)

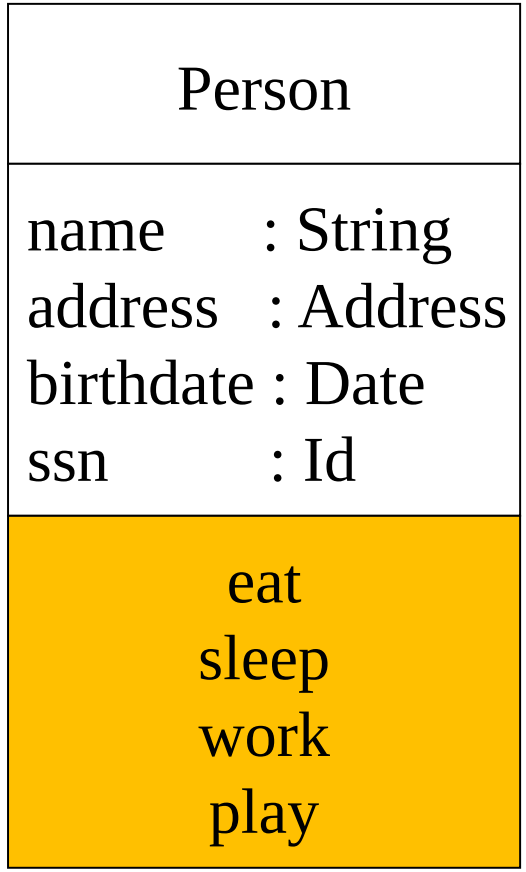


Attributes can be:

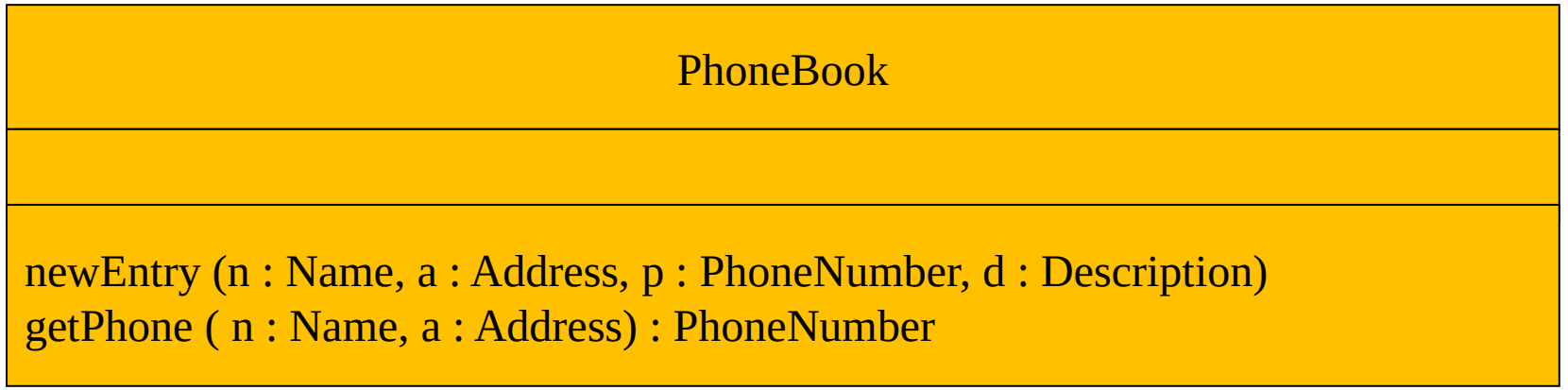
+ public
# protected
- private
/ derived



Class Operations

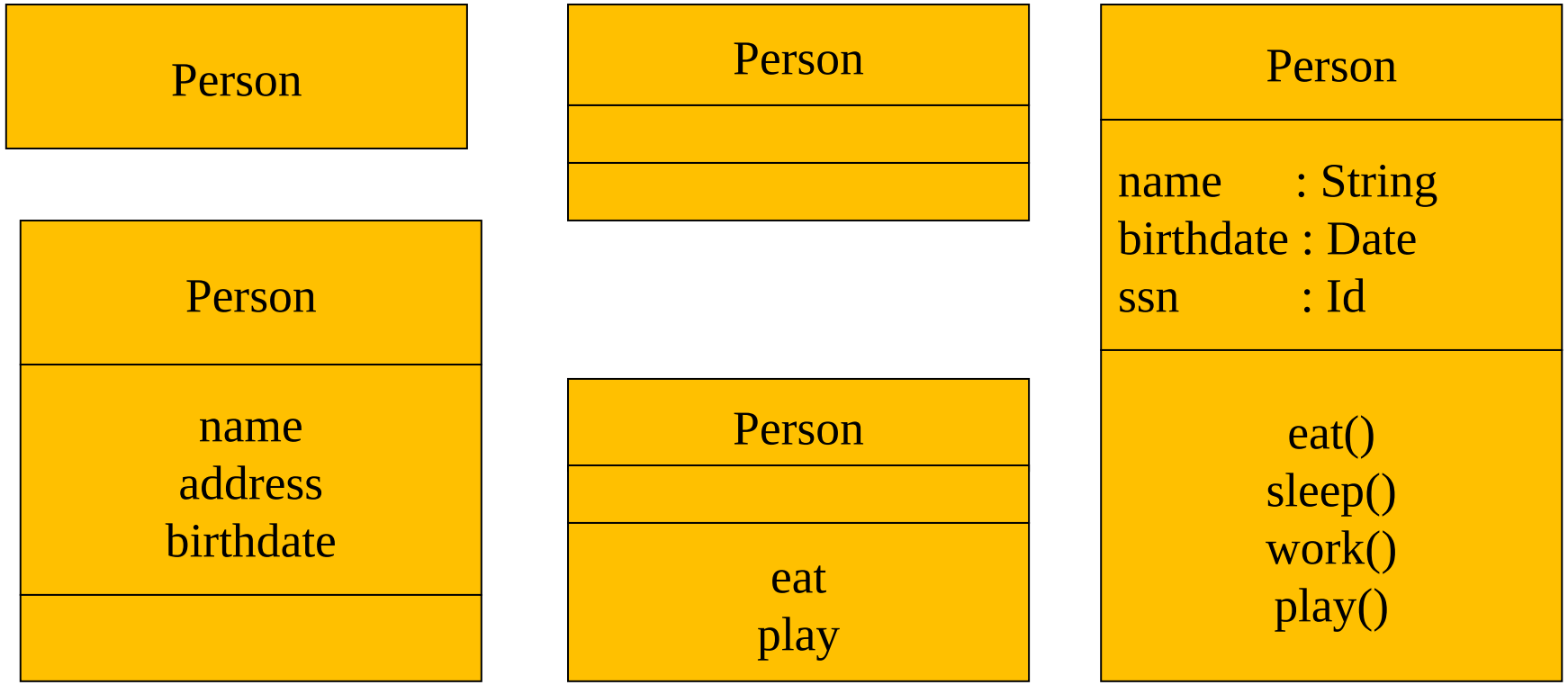


- *Operations* describe the class behavior and appear in the third compartment
- You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type



Depicting Classes

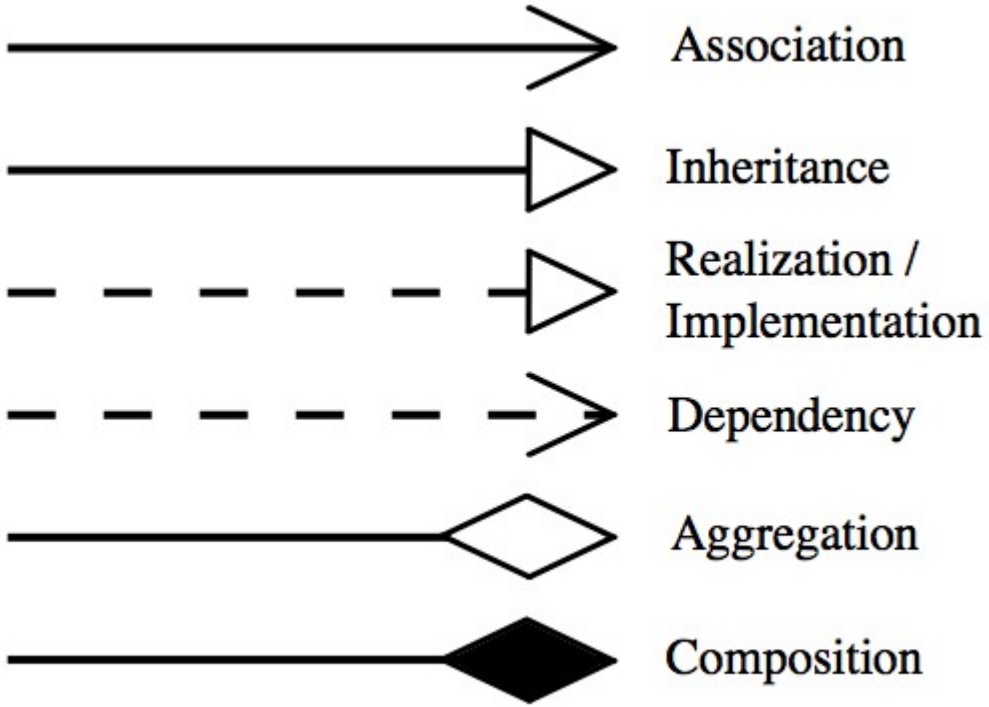
When drawing a class, you needn't show attributes and operation in every diagram



Relationships

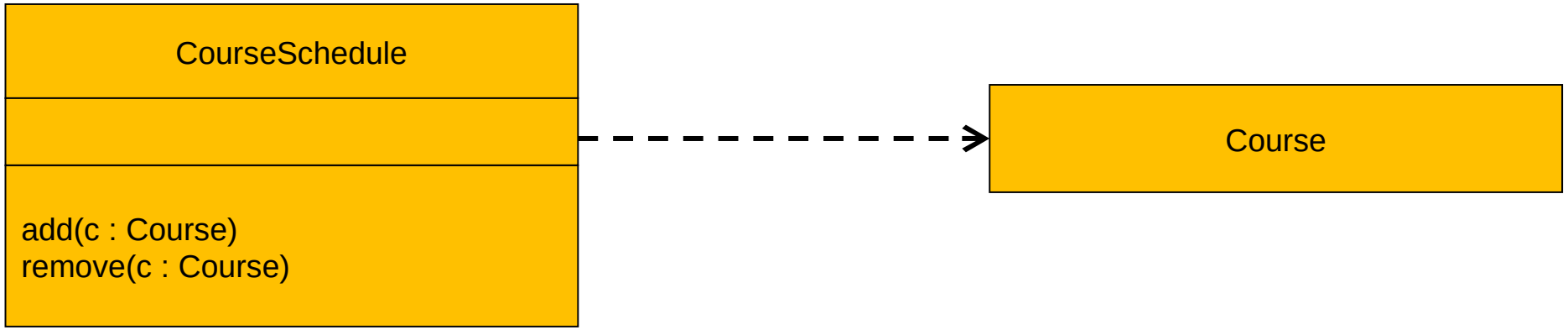
- In UML, object interconnections (logical or physical), are modeled as relationships
- There are three kinds of relationships in UML:
  - dependencies
  - generalizations
  - associations

Relationships

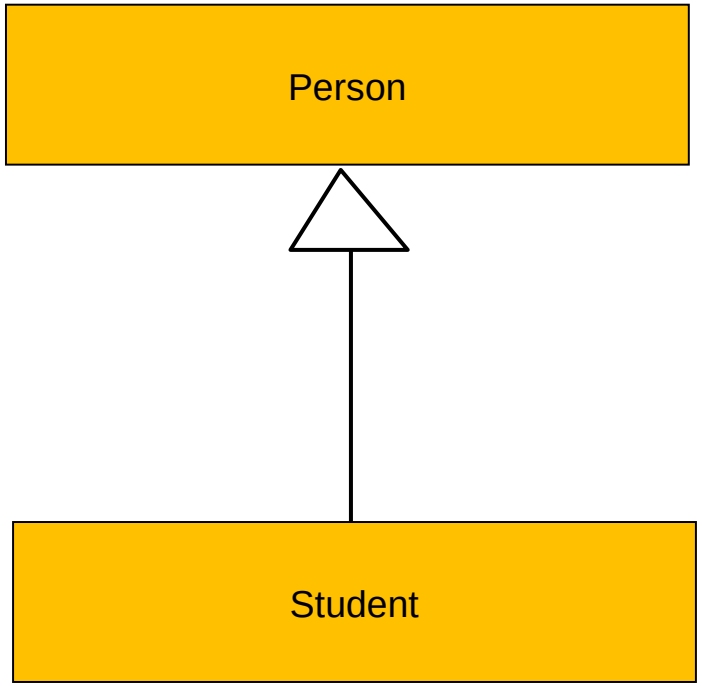


Dependency Relationships

- A *dependency* indicates a semantic relationship between two or more elements
- The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*



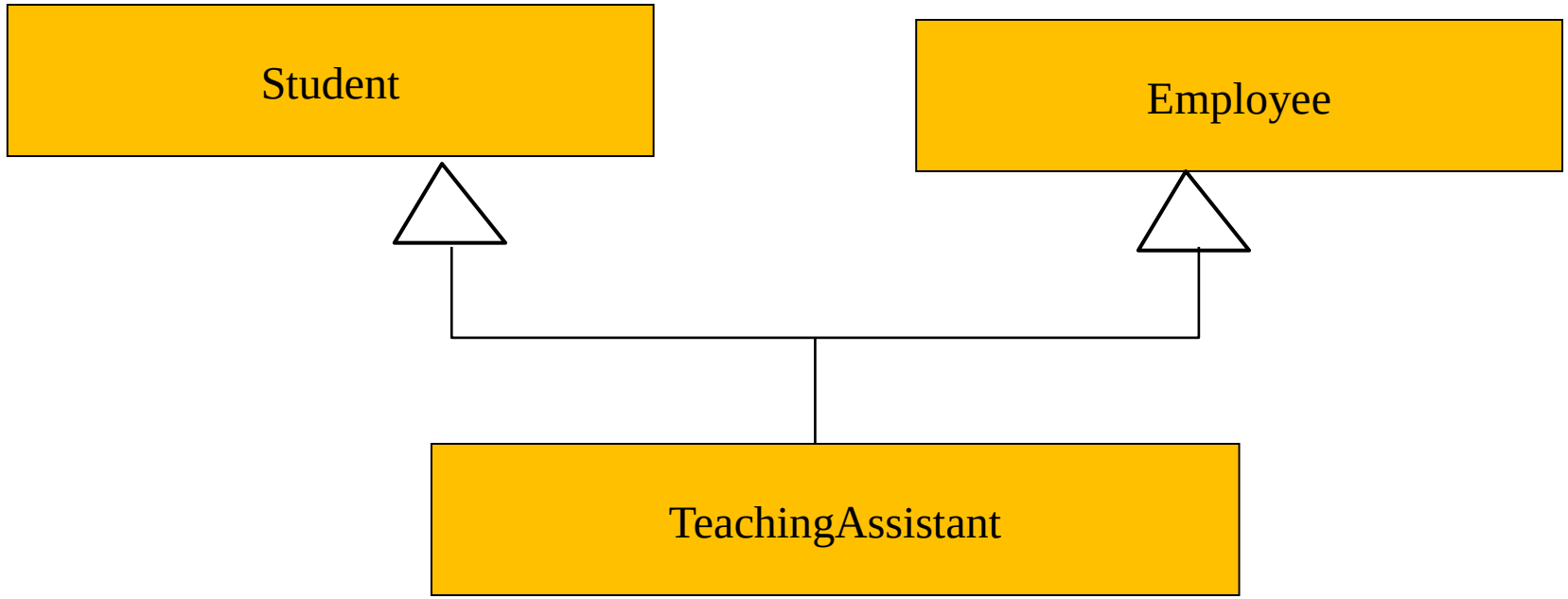
Generalization Relationships



- A *generalization* connects a subclass to its superclass
- It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass

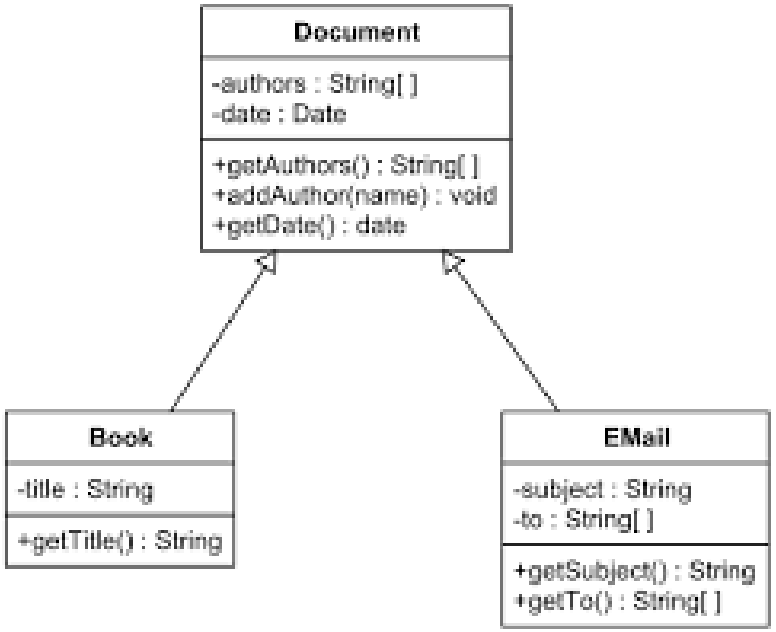
Generalization Relationships (Contd.)

UML permits a class to inherit from multiple superclasses, although some programming languages (*e.g.*, Java) do not permit multiple inheritance



Inheritance

- Indicates that child (subclass) is considered to be a specialized form of the parent (super class).
- For example consider the following:



Association

- An association is a relationship between two separate classes
- It joins two entirely separate entities.
- There are four different types of association:
    - Bi-directional
    - Uni-directional
    - Aggregation (includes composition aggregation)
    - Reflexive
- Bi-directional and uni-directional associations are the most common ones
- This can be specified using multiplicity (one to one, one to many, many to many, etc.)
- A typical implementation in Java is through the use of an instance field
- The relationship can be bi-directional with each class holding a reference to the other
- Associations that have the same class at both ends are known as reflexive associations

Association Relationships

- If two classes in a model need to communicate with each other, there must be link between them
- An *association* denotes that link

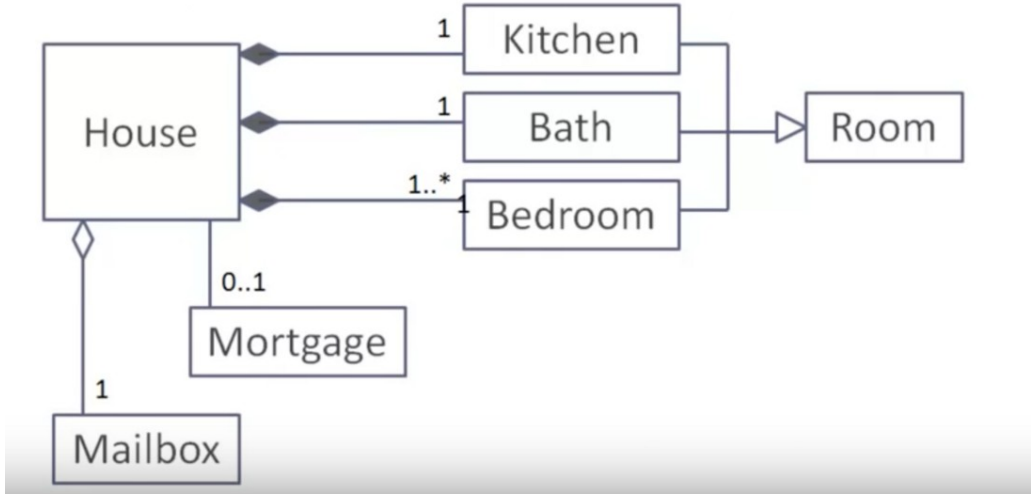
Association Relationships (Contd.)

We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association

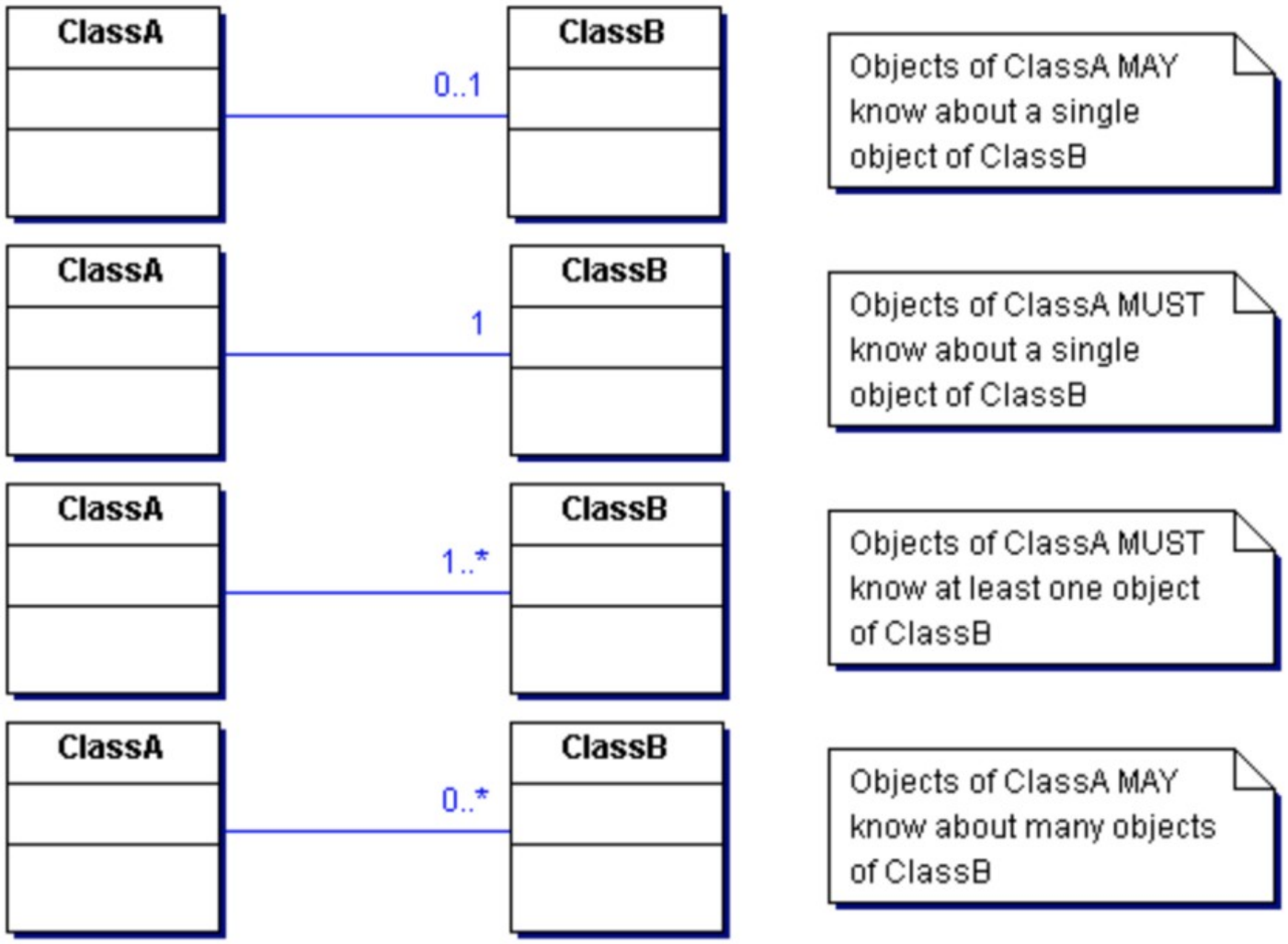
The example indicates that a *Student* has one or more *Instructors*:

Multiplicity

- After specifying the type of association relationship by connecting the classes, you can also declare the cardinality between the associated entities
- For example:
- The below UML diagram shows that a house has exactly one kitchen, exactly one bath, at least one bedroom (can have many), exactly one mailbox, and at most one mortgage (zero or one)



Multiplicity



Association Relationships (Contd.)

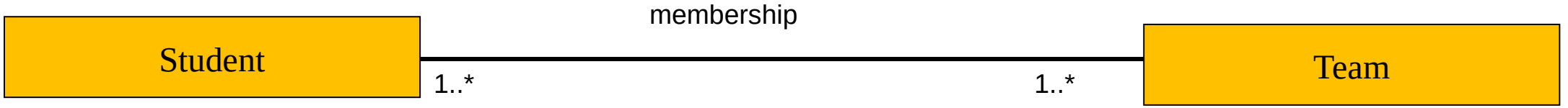
The example indicates that every *Instructor* has one or more *Students*:

Association Relationships (Contd.)

We can also indicate the behavior of an object in an association (*i.e.*, the *role* of an object) using *rolename*

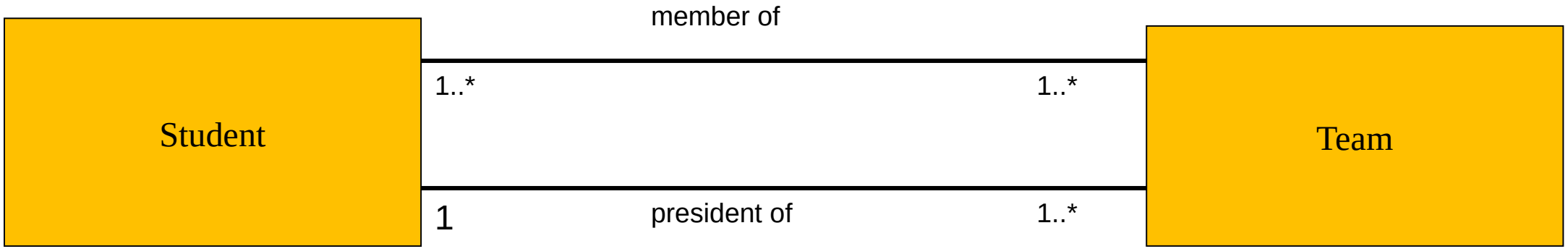
Association Relationships (Contd.)

We can also name the association.



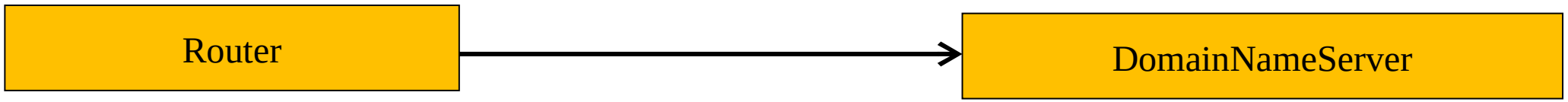
Association Relationships (Contd.)

We can specify dual associations



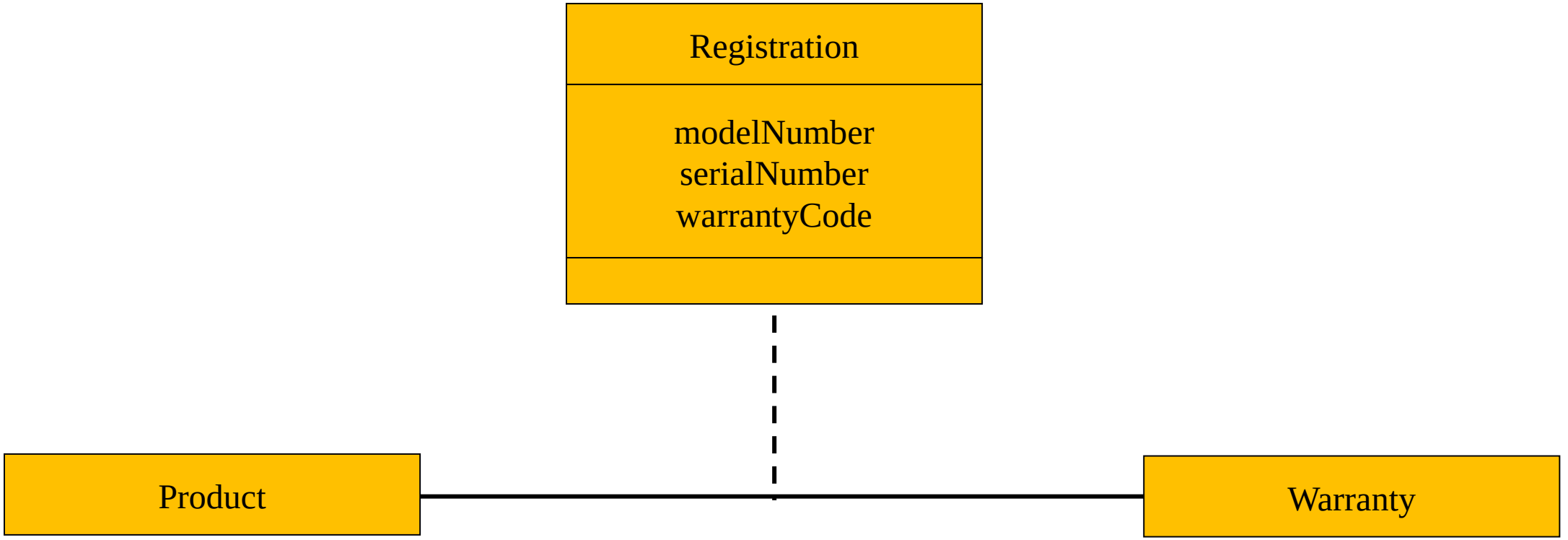
Association Relationships (Contd.)

- We can constrain the association relationship by defining the *navigability* of the association
- Here, a *Router* object requests services from a *DNS* object by sending messages to (invoking the operations of) the server
- The direction of the association indicates that the server has no knowledge of the *Router*



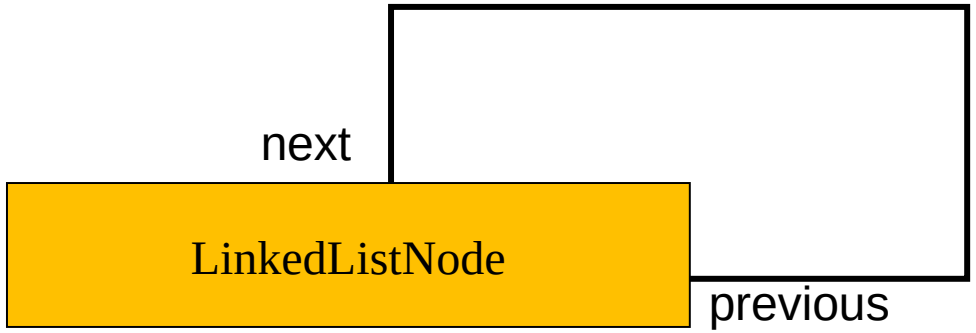
Association Relationships (Contd.)

Associations can also be objects themselves, called *link classes* or an *association classes*



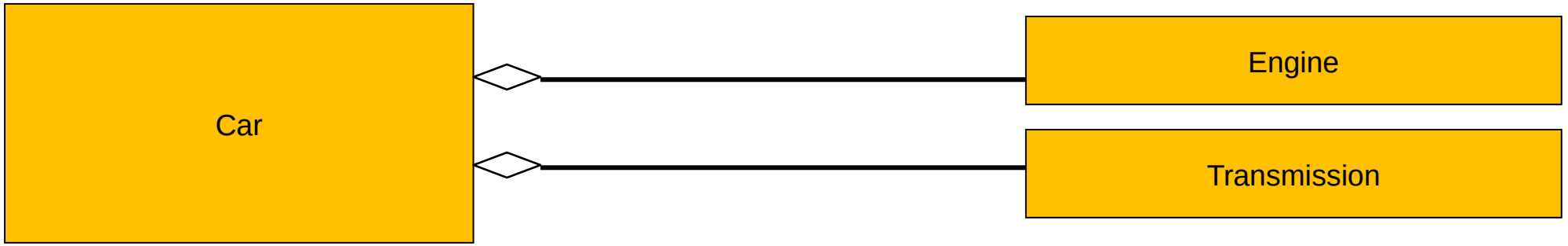
Association Relationships (Contd.)

A class can have a *self association*



Association Relationships (Contd.)

- We can model objects that contain other objects by way of special associations called *aggregations* and *compositions*
- An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate
- Aggregations are denoted by a hollow-diamond adornment on the association



Dependency - Aggregation

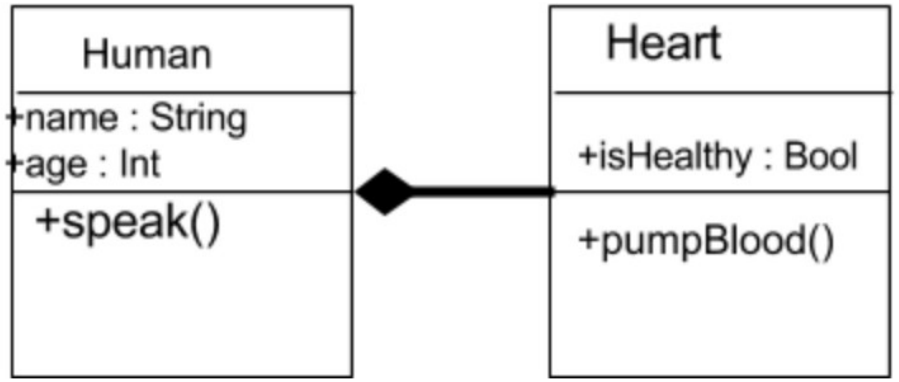
- A special form of association which is a unidirectional (a.k.a one way) relationship between classes
- The best way to understand this relationship is to call it a “has a” or “is part of” relationship
- For example, consider the two classes: Wallet and Money
- A wallet “has” money, but money doesn’t necessarily need to have a wallet so it’s a one directional relationship

Association Relationships (Contd.)

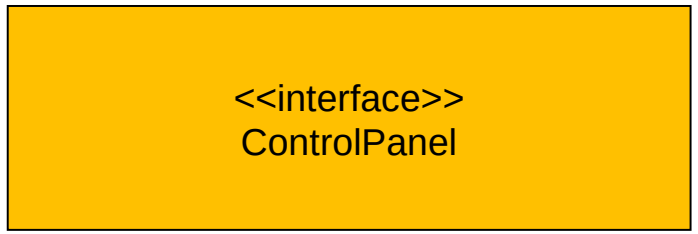
- A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole)
- Compositions are denoted by a filled-diamond adornment on the association

Composition

- A restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other
- A human needs a heart to live and a heart needs a human body to function on
- In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one does too) then it’s a composition



Interfaces



- An *interface* is a named set of operations that specifies the behavior of objects without showing their inner structure
- It can be rendered in the model by a one- or two-compartment rectangle, with the *stereotype* <<interface>> above the interface name

Interface

A relationship between two model elements, in which one model element implements/executes the behavior that the other model element specifies