COMP 3111
SOFTWARE ENGINEERING

LECTURE 2
MODELING SOFTWARE SYSTEMS
USING UML

LEARNING OBJECTIVES

1. Understand what is the UML and how the UML can be used to model software systems.

2. Appreciate that the UML is a modeling language and *not* a software development methodology.

3. Understand the basic modeling components of UML class diagrams: class, association and generalization.

MODELING SOFTWARE SYSTEMS USING UML: OUTLINE

UML and Object-oriented Modeling

- Overview of the UML
- Object-oriented Modeling

Class

- Attribute
- Operation

Association

- Multiplicity
- Aggregation and Composition

Association Class

Generalization

- Inheritance
- Coverage

Constraints

WHAT IS THE UML*?

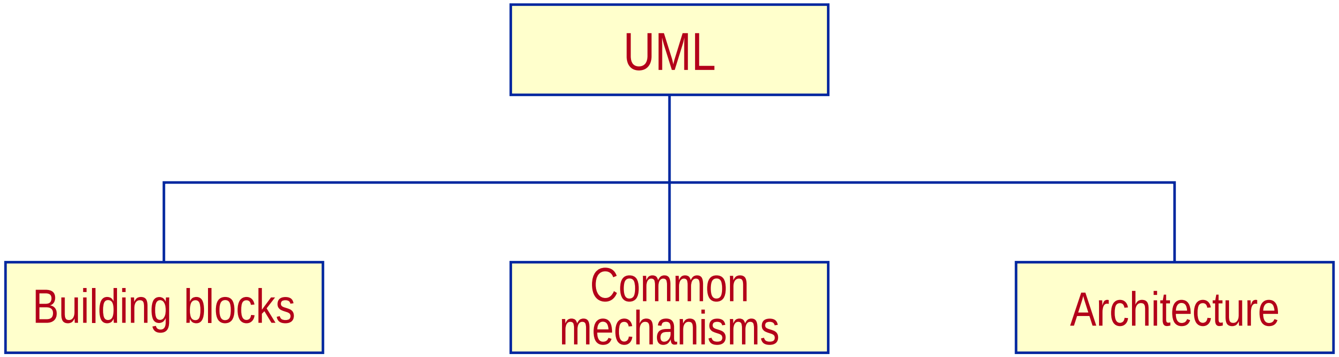
- General purpose *visual modeling language* for systems.
- *Incorporates current best practices* in OO modeling techniques.
- *Software development methodology/process* *neutral*.
- *Industry standard OO modeling language for modeling systems* (but can also be used for non-OO systems).

Basic Premise of the UML

A software system can be modeled as a collection of collaborating objects.

* Unified Modeling Language

UML STRUCTURE



- **Building blocks**
  - things
  - relationships
  - diagrams
- **Common mechanisms**
  - specifications
  - adornments
  - common divisions
  - extensibility mechanisms
- **Architecture**
  - use-case view
  - logical view
  - implementation view
  - process view
  - deployment view

WHY BUILD MODELS?

How the customer explained it.

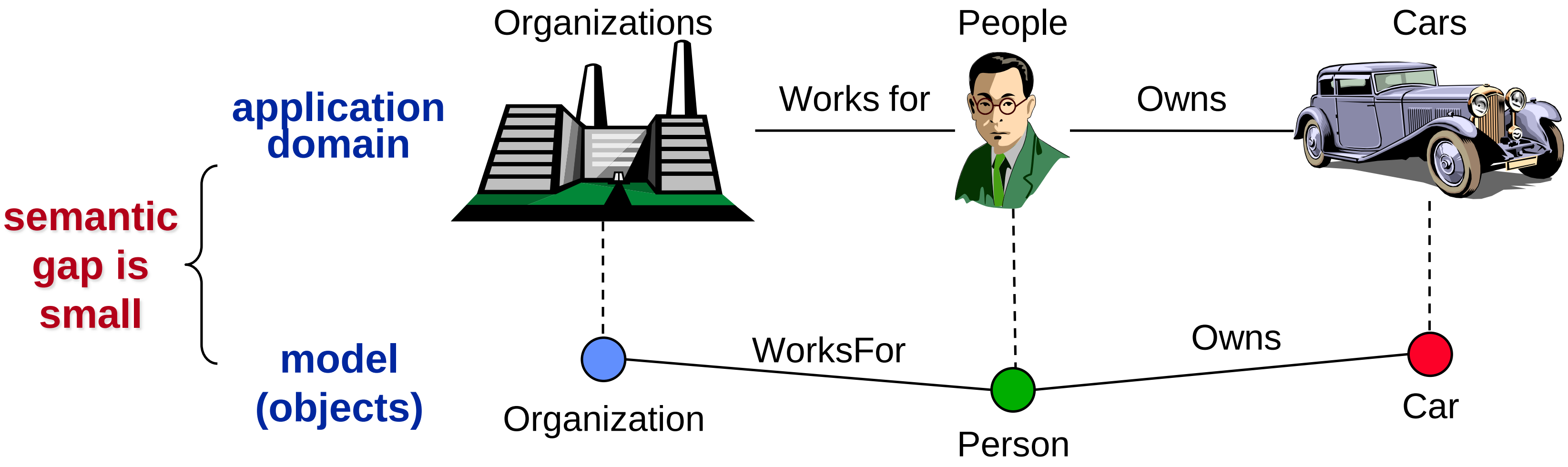
What do you think is the problem here?

Why?

WHY BUILD MODELS?

- Models succinctly describe reality (i.e., they abstract reality).
  - They show essential details and filter out non-essential details.
- For software development, this allows us to focus on the “big picture”,
  - i.e., programming-in-the-large.
- Such a focus allows us to better deal with the *complexity* of software development,
  - i.e., with human limitations in understanding complex things.
- The result is better understanding of requirements, cleaner designs, and more maintainable systems.

WHY OBJECT-ORIENTED MODELING?



☞ Allows direct representation of “things” in an application domain.

☞ Reduces the “semantic gap” between the application domain and the model.

☞ Better represents how people think about reality.

An application domain is modeled as a collection of objects.

OO MODELING & LEVELS OF ABSTRACTION

Requirements level → We construct a *requirements model.*

- We do not consider any aspects of the implementation of objects.

☞ **Focus: identifying objects (concepts) in the application domain.**

Analysis & Design level → We construct a *solution model.*

- We consider interfaces of objects (but no internal aspects).

☞ **Focus: how objects interact in the solution.**

Implementation level → We implement the *solution model.*

- We consider all details of objects (external and internal).

☞ **Focus: how to code objects.**

The same OO concepts can be used at all levels.

MODELING SOFTWARE SYSTEMS USING UML: OUTLINE

✓ UML and Object-oriented Modeling
- Overview of the UML
- Object-oriented Modeling

➔ **Class**
- **Attribute**
- **Operation**

Association
- Multiplicity
- Aggregation and Composition

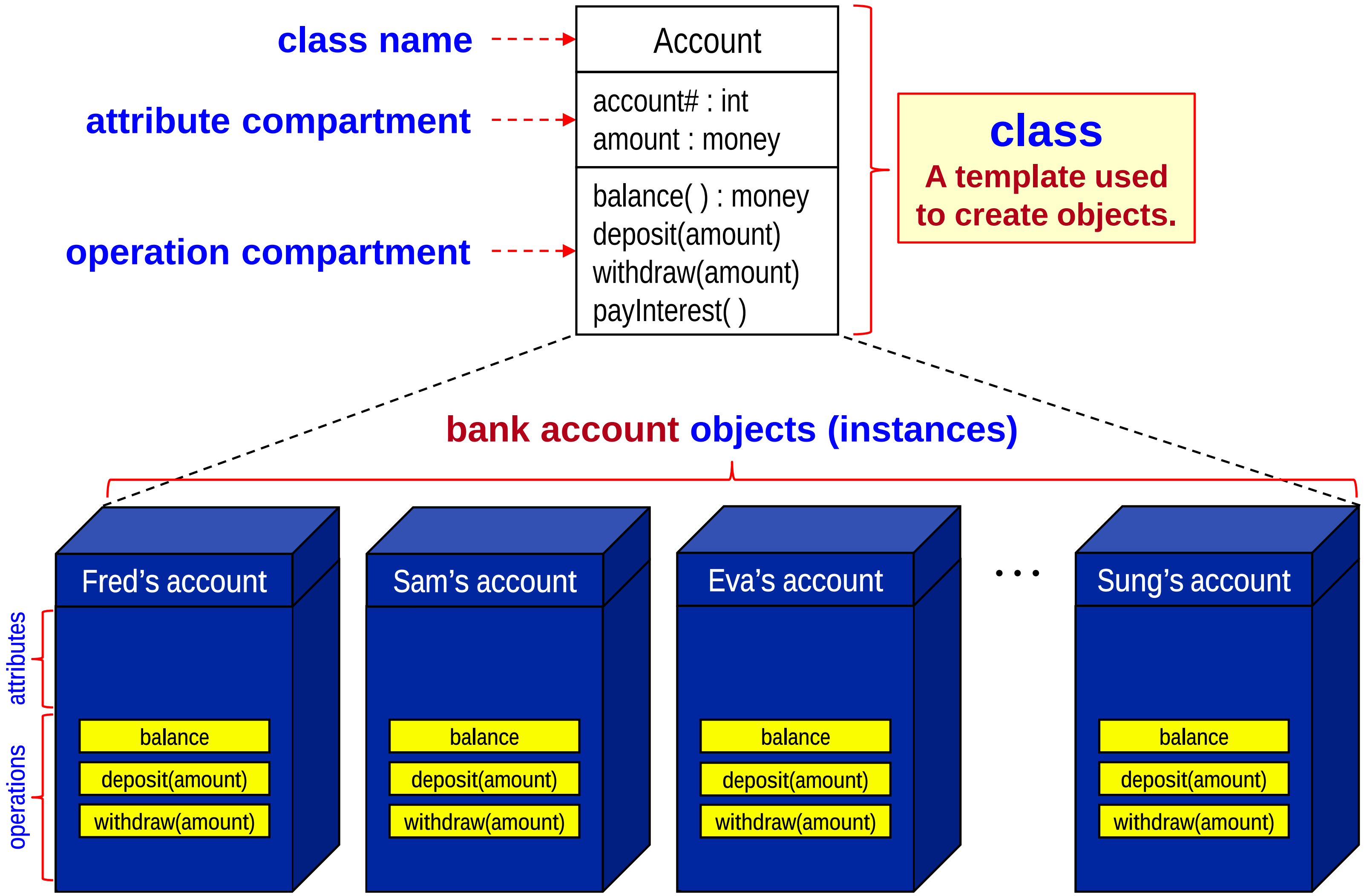
Association Class

Generalization
- Inheritance
- Coverage

Constraints

2.3.3 2.4.2

CLASS



class name

attribute compartment

operation compartment

Account

account# : int
amount : money

balance( ) : money
deposit(amount)
withdraw(amount)
payInterest( )

class

A template used to create objects.

bank account objects (instances)

attributes

operations

Fred's account: balance, deposit(amount), withdraw(amount)

Sam's account: balance, deposit(amount), withdraw(amount)

Eva's account: balance, deposit(amount), withdraw(amount)

• • •

Sung's account: balance, deposit(amount), withdraw(amount)

CLASS

A *class* describes a collection of objects having common:

– semantics – attributes – operations – relationships

☞ A class is a classifier; an object is an instance.

- A class is a “factory” for creating objects.
- A good class should capture one and only one abstraction.

  ☞ It should have one major theme.

- A class should be named using the vocabulary of the application domain (class names must be unique).

  ☞ So that it is meaningful and traceable from the application domain to the model.

2.3.3 5.4.8

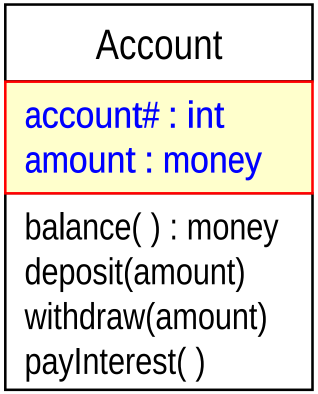
CLASS: ATTRIBUTE

An *attribute* describes the data values held by objects in a class.

- Attribute properties:
  - name: unique within a class, but not across classes.
  - type: the domain of values – string, integer, money, etc.

For modeling, name and type should always be specified.

  - visibility: who can access the attribute's values.
    public (+), private (–), protected (#), package (~)
  - initial value [optional]: the attribute's initial value.
  - multiplicity [optional]: the number of simultaneous values.
  - changeability: whether the value can be changed.
    unspecified *(default)* readOnly


Account
account# : int
amount : money
balance( ) : money
deposit(amount)
withdraw(amount)
payInterest( )

2.3.3 2.4.2

CLASS: OPERATION

An *operation* describes a function or transformation that may be applied to or by objects in a class.

- Operation properties:
  - operation signature

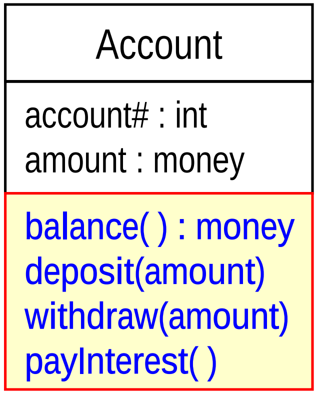
    operation name

    parameter names

    result type

    For modeling, all should always be specified.
  - visibility

    public (+), private (–), protected (#), package (~)


Account

account# : int
amount : money

balance( ) : money
deposit(amount)
withdraw(amount)
payInterest( )

- An operation instance (its implementation) is called a method.

☞ An operation can have several methods that implement it (polymorphic operation).

WHY CLASSES FOR MODELING SYSTEMS?

By abstracting a collection of objects and representing them as a class, ***the complexity of developing a system is reduced*** since it becomes easier to:

- **understand** the system → We need to understand only the classes, not the individual objects.
- **specify** the system → Classes provide a place to define and store common definitions only once.

Choosing appropriate classes is an

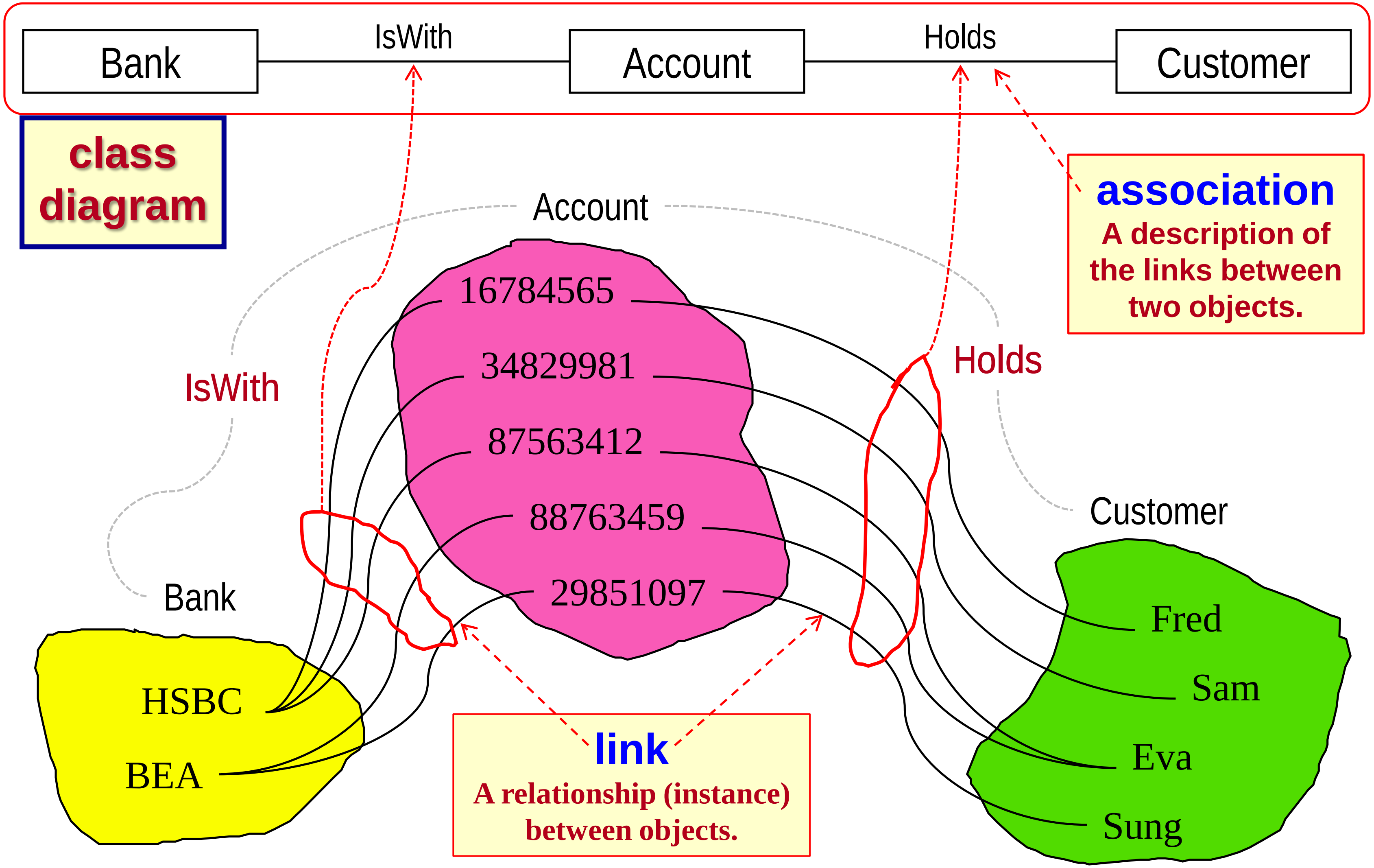
IMPORTANT DESIGN DECISION

that *helps promote modular development*.

MODELING SOFTWARE SYSTEMS USING UML: OUTLINE

- ✓ UML and Object-oriented Modeling
  - Overview of the UML
  - Object-oriented Modeling
- ✓ Class
  - Attribute
  - Operation
- ➔ **Association**
  - **Multiplicity**
  - **Aggregation and Composition**
- Association Class
- Generalization
  - Inheritance
  - Coverage
- Constraints

ASSOCIATION



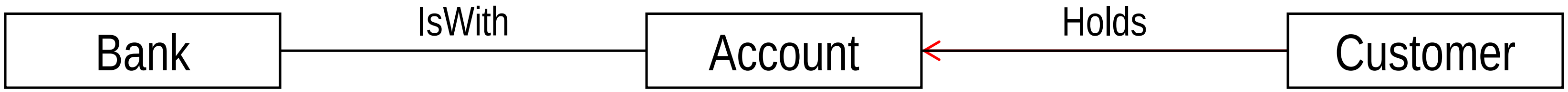
class diagram

association
A description of the links between two objects.

link
A relationship (instance) between objects.

ASSOCIATION

An *association* describes a collection of links with common semantics.



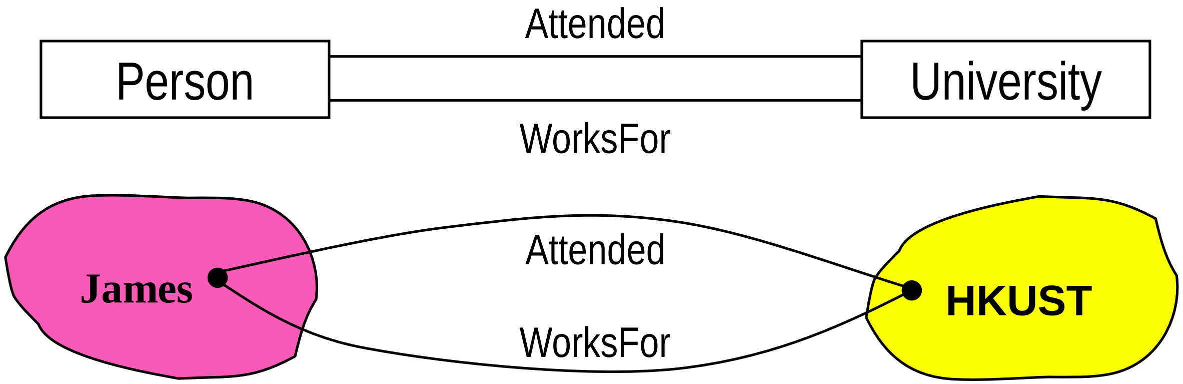
☞ An association is a classifier; a link is an instance.

☞ Conceptually, associations are inherently bi-directional.

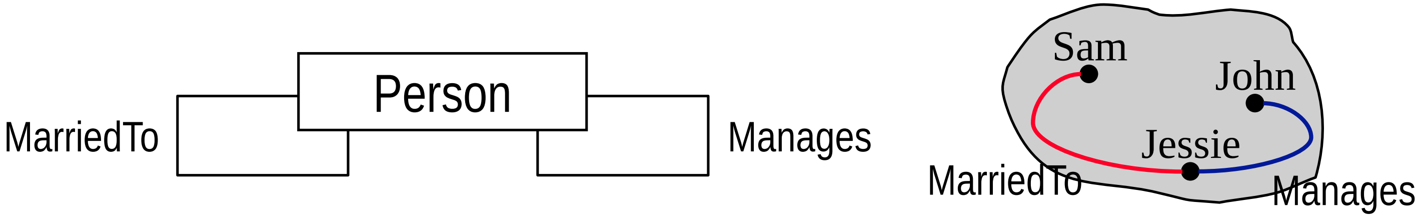
☞ Can show navigability of associations with an arrowhead.
(Implies that the source object has a reference to the target object.)

ASSOCIATIONS AND CLASSES

- Two different classes can be related by several associations.



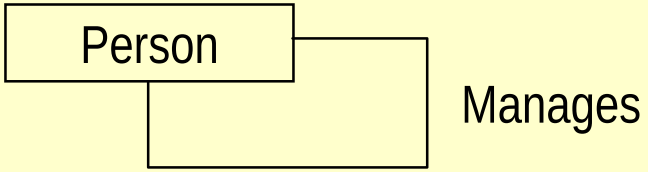
- The same class can be related by several associations.



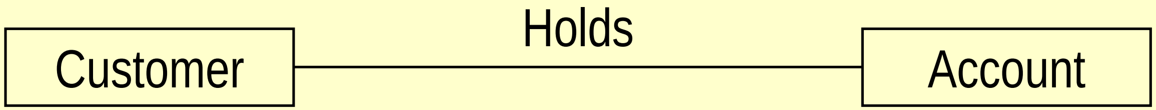
The collection of class and association names must be unique.

ASSOCIATION: DEGREE

- **unary** (reflexive)
  relates *one* class to itself

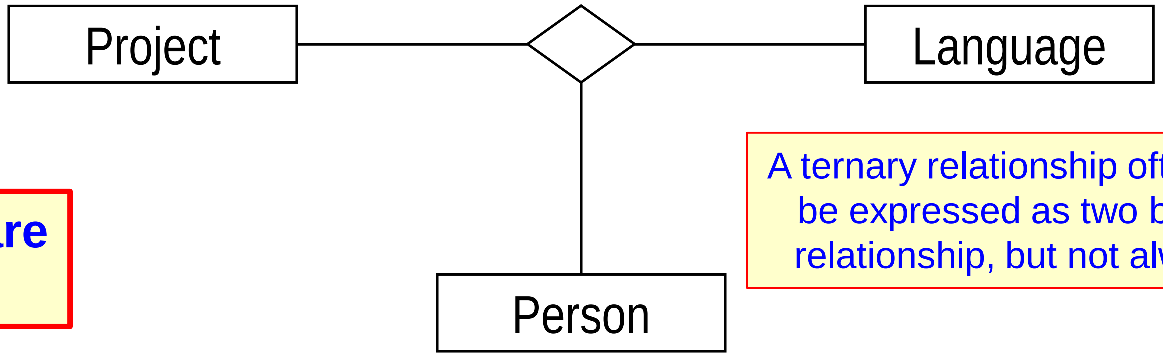

Person
Manages

- **binary**
  relates *two* classes


Customer
Holds
Account

We will use only unary and binary associations in this course.

- **ternary**
  relates *three* classes


Project
Language
Person

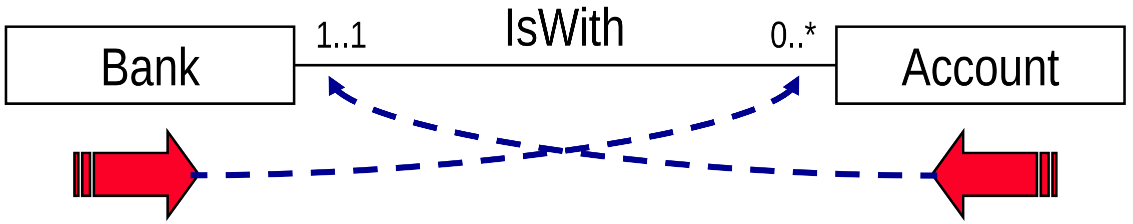
A ternary relationship often can be expressed as two binary relationship, but not always.

Higher degrees are extremely rare!

In practice, the majority of associations are binary!

ASSOCIATION: MULTIPLICITY

Multiplicity **specifies a restriction on the number of objects in a class that may be related to an object in another class.**



For a given bank, how many accounts can it have?

- ☞ A bank may have no accounts or it may have many accounts.

For a given account, how many banks can it be with?

- ☞ An account must be with exactly one bank.

Multiplicity is an *application domain constraint*!

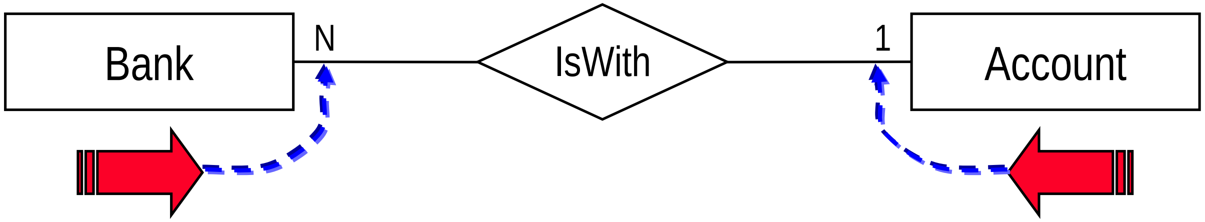
ASSOCIATION: MULTIPLICITY (cont'd)

A NOTE FOR COMP 3311 STUDENTS

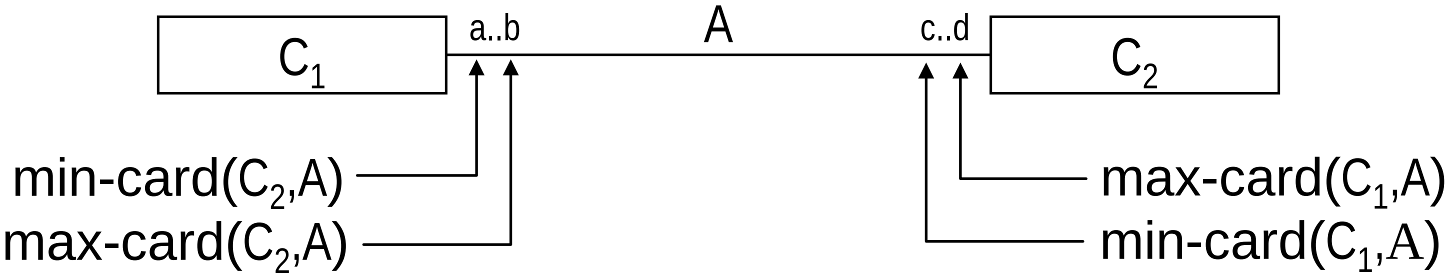
Both the ER model and the UML can represent the data requirements of a system.

However, placement of the multiplicity in the ER model used in COMP 3311 *is different* than that of the UML.

CAUTION: BE CAREFUL NOT TO MIX UP NOTATIONS!



ASSOCIATION: MULTIPLICITY (cont'd)



minimum cardinality (min-card)

min-card(C_1,A): the *minimum number of links* in which each object of C_1 can participate in association A

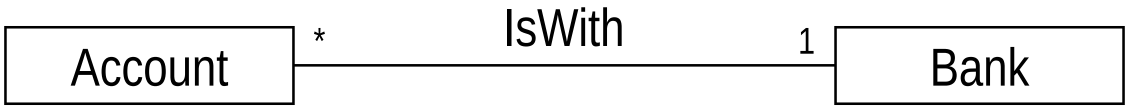
min-card(C_1,A) = 0 → optional participation *(may not be related)*

min-card(C_1,A) > 0 → mandatory participation *(must be related)*

maximum cardinality (max-card)

max-card(C_1,A): the *maximum number of links* in which each object of C_1 can participate in association A

ASSOCIATION: MULTIPLICITY (cont'd)



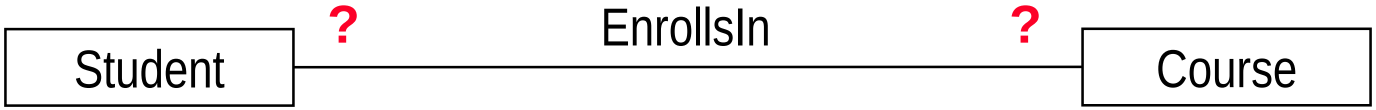
special cardinalities:

max-card = ***** → an unlimited upper bound (∞)

min-card = 1 and max-card = 1 → can use 1 by itself

min-card = 0 and max-card = ***** → can use ***** by itself

MULTIPLICITY EXAMPLE

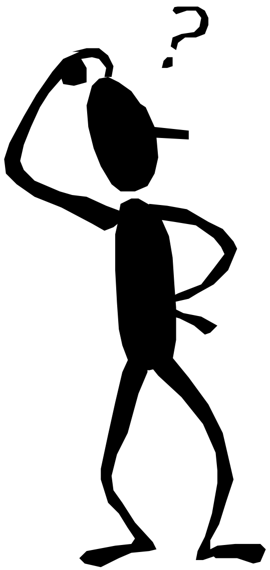


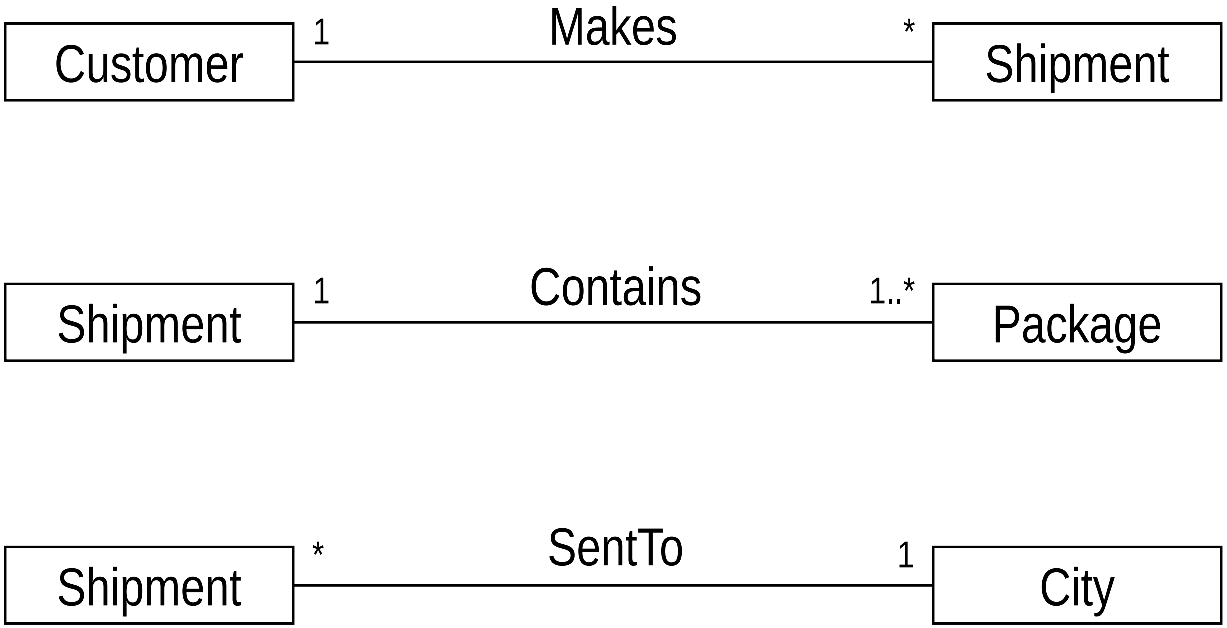
- A student must enroll in at least one course and can enroll in at most five courses

- A course must have at least ten students enrolled in it and cannot have more than forty-five students enrolled in it.

SINEX — COURSE PROJECT QUESTION

What is the most likely multiplicity of the following associations?





Customer 1 Makes * Shipment

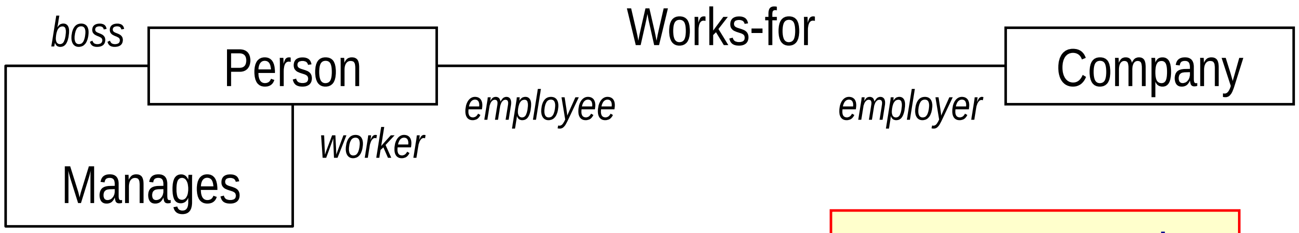
Shipment 1 Contains 1..* Package

Shipment * SentTo 1 City

2.4.2 5.4.6

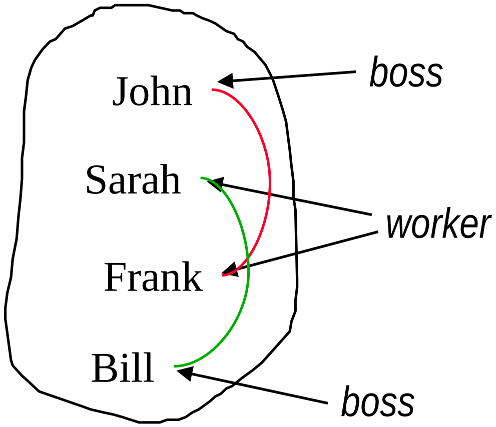
ASSOCIATION: ROLE

A *role* is one end of an association.



For unary and binary associations there are two roles.

Who manages who?

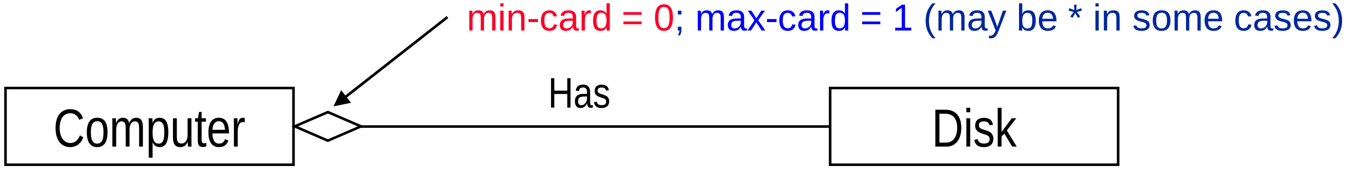


It is necessary to use role names when an association relates objects from the same class.

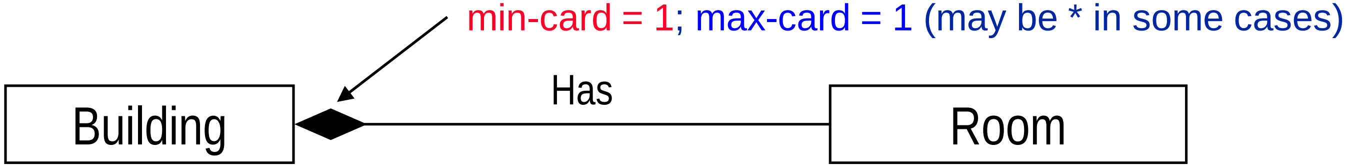
AGGREGATION/COMPOSITION ASSOCIATION

- A special type of association in which there is a “part-of” relationship between one class and another class.

☞ A component may exist independent of the aggregate object of which it is a part → aggregation. [◇ adornment]



☞ A component may not exist independent of the aggregate object of which it is a part → composition. [◆ adornment]



WHEN TO USE AGGREGATION/COMPOSITION?

- Would you use the phrase “part of” to describe the association or name it “Has”?

  ☞ **BUT BE CAREFUL!** Not all “Has” associations are aggregations.

- Is there an intrinsic asymmetry to the association where one object class is subordinate to the other(s)?

- Are operations on the whole automatically applied to the part(s)? → composition

The decision to use aggregation is a matter of *judgment*.
It is a design decision.

It is not wrong to use association rather than aggregation!
(In a real project, when in doubt, use association!)

MODELING SOFTWARE SYSTEMS USING UML: OUTLINE

✓ UML and Object-oriented Modeling
- Overview of the UML
- Object-oriented Modeling

✓ Class
- Attribute
- Operation

✓ Association
- Multiplicity
- Aggregation and Composition

➔ **Association Class**

Generalization
- Inheritance
- Coverage

Constraints

MODELING SOFTWARE SYSTEMS USING UML

EXERCISE