Introduction to UML

UML

- UML is a visual modelling Language
- **visual** --- UML documents are a diagrams.
- **modelling** --- UML is for describing systems
- **systems** --- may be software systems or domains (e.g. business systems), etc.
- It is **semi-formal**
  - The UML definition tries to give a reasonably well defined meaning to each construct.

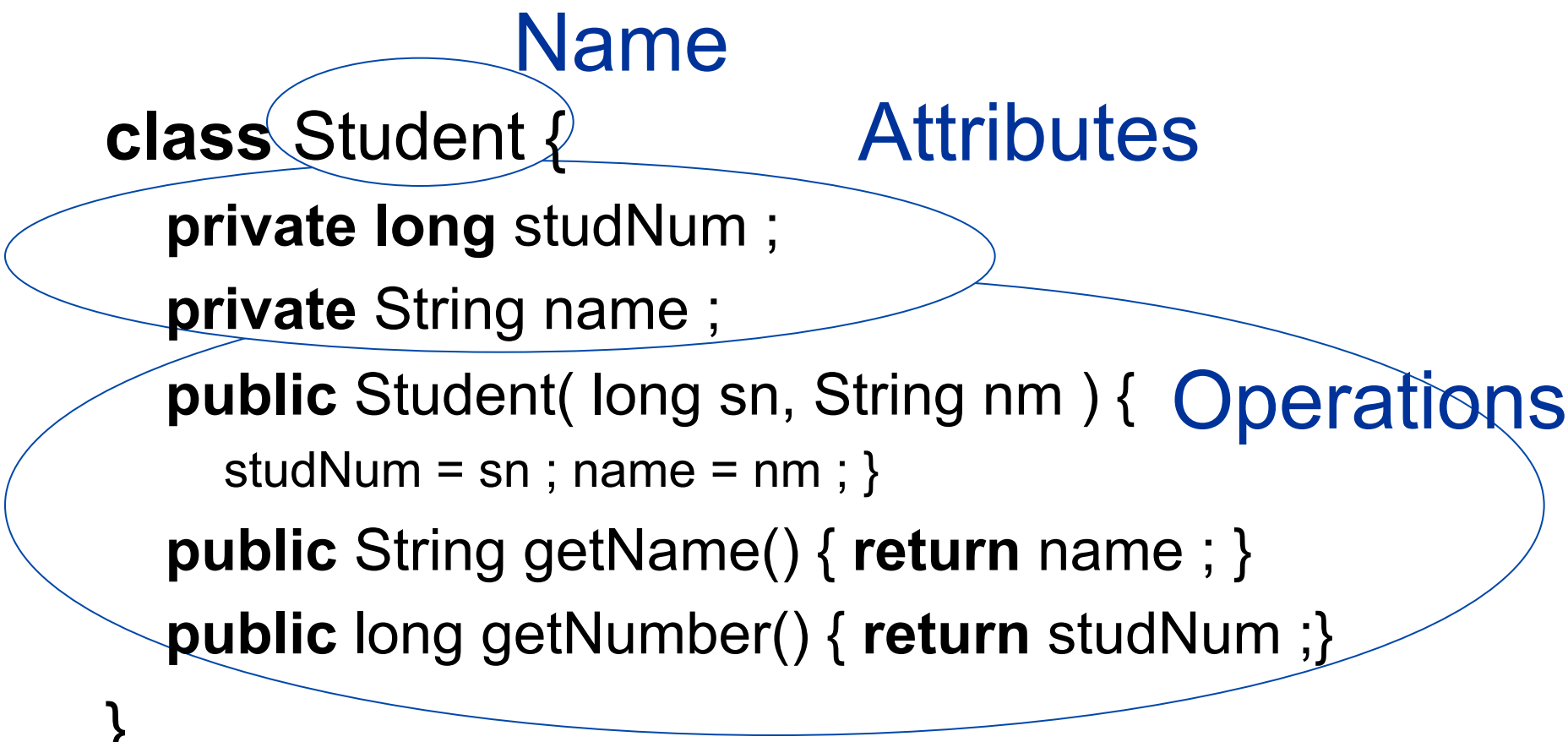
Classes and Class Diagrams

- Classes
- Fields and Operations
- Association
- Composition
- Generalization
- Interfaces and Abstract Classes

Classes

- Classes are specifications for objects
- Consist of (in the main)
  - A name
  - A set of *attributes* (aka *fields*)
  - A set of *operations*
    - Constructors: initialize the object state
    - Accessors: report on the object state
    - Mutators: alter the object state
    - Destructors: clean up

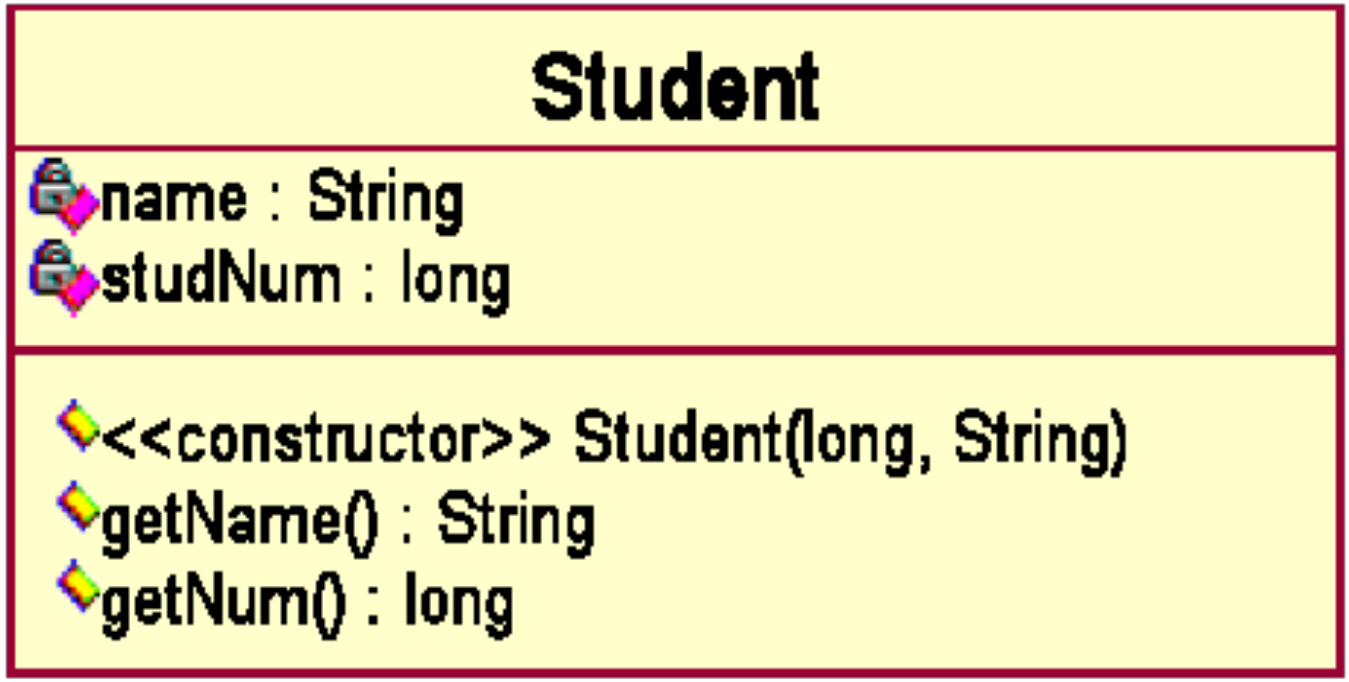
Java representation of a class




```
class Student {
    private long studNum ;
    private String name ;
    public Student( long sn, String nm ) {
        studNum = sn ; name = nm ; }
    public String getName() { return name ; }
    public long getNumber() { return studNum ;}
}
```

UML Representation of a class

Note: UML model may contain more info.



Classes in UML

UML can be used for many purposes.

- In *software design* UML classes usually correspond to classes in the code.
- But in *domain analysis* UML classes are typically classes of real objects (e.g. real students) rather than their software representations.

Usage of (software) classes

A class `C` can be used in 3 ways:

- **Instantiation**. You can use `C` to create new objects.
  - Example: `new C()`
- **Extension**. You can use `C` as the basis for implementing other classes
  - Example: `class D extends C { … }`
- **Type**. You can use `C` as a type
  - Examples: `C func( C p ) { C q ;… }`

Relationships Between Classes

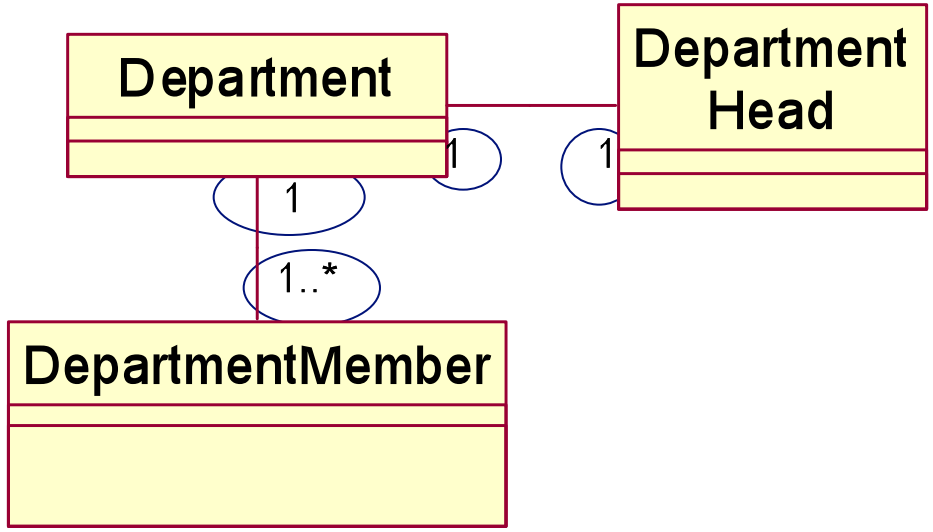
- Association
- Aggregation
- Composition
- Dependence
- Generalization

Association Relationships

- Two classes are “associated” if each instance of one may be associated with instances of the other.
- Associations are typically named.
- Associations are often implemented with pointers

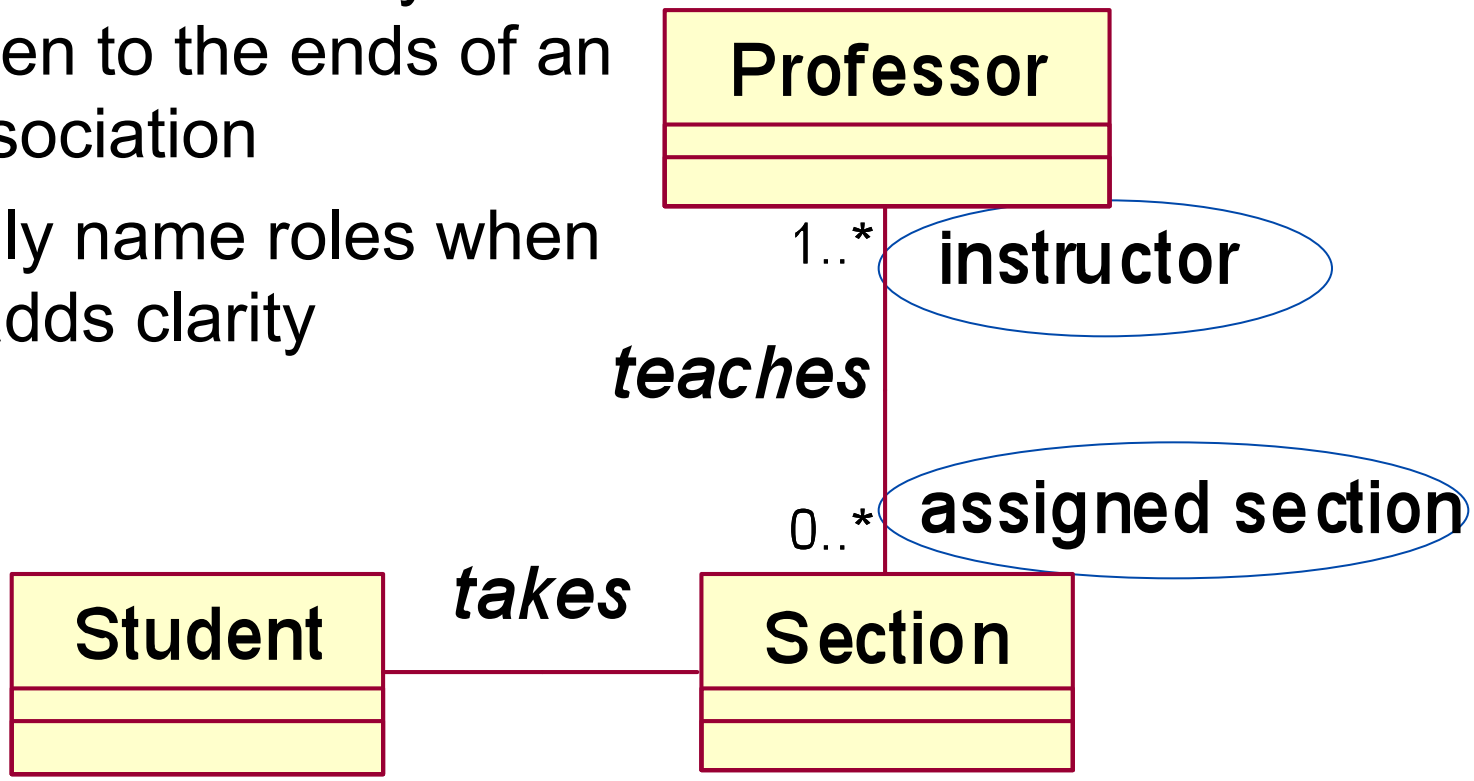
Multiplicity Constraints

- Each Department is associated with one DepartmentHead and at least one DepartmentMember
- Each DepartmentHead and DepartmentMember is associated with one Department
- No constraint means multiplicity is unspecified



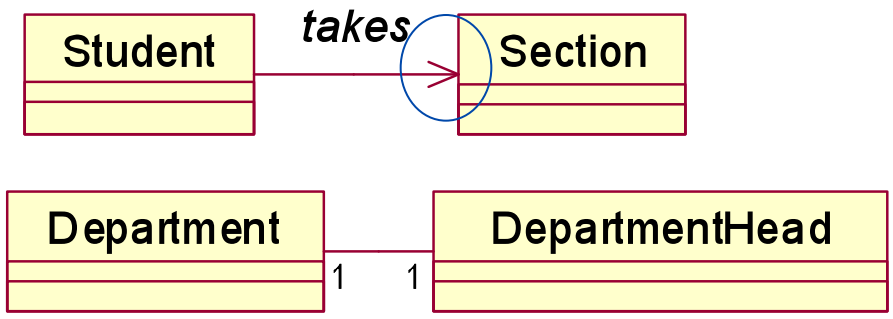
Role names

- Role names may be given to the ends of an association
- Only name roles when it adds clarity



Navagability

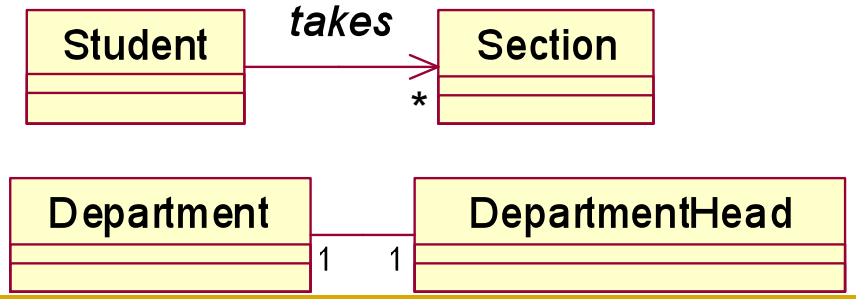
- An arrow-head indicates the direction of navigability.
- E.g. Given a student object, we can easily find all Sections the student is taking.
- ⚠ No arrow-head: means navigability in both directions.



Implementing navigable associations

Usually implemented with fields

```
class Student {
  private List<Section> sections ; … }
class Department {
  private DepartmentHead deptHead ;  … }
```



Implementing associations indirectly

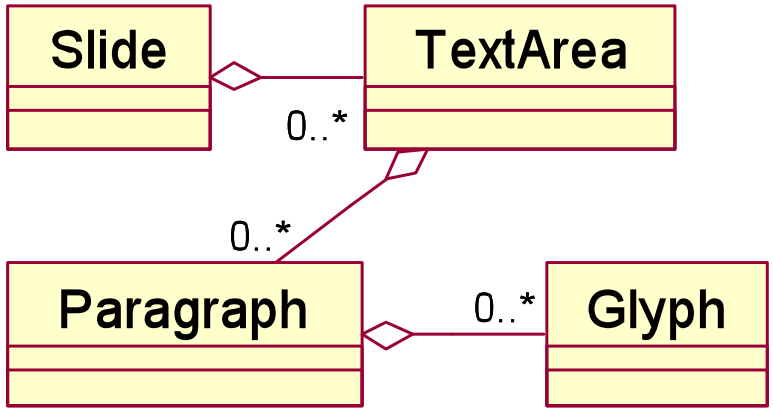
- An association between objects might also be stored outside of the objects

```
class Department {
  private static
    Map<Department,DepartmentHead> heads =
    new<Department,DepartmentHead> HashMap();

  DepartmentHead getHead() {
    return  heads.get(this) ;  }
  …
}
```

Aggregation

- Aggregation is a special case of association.
- It is used when there is a “whole-part” relationship between objects.

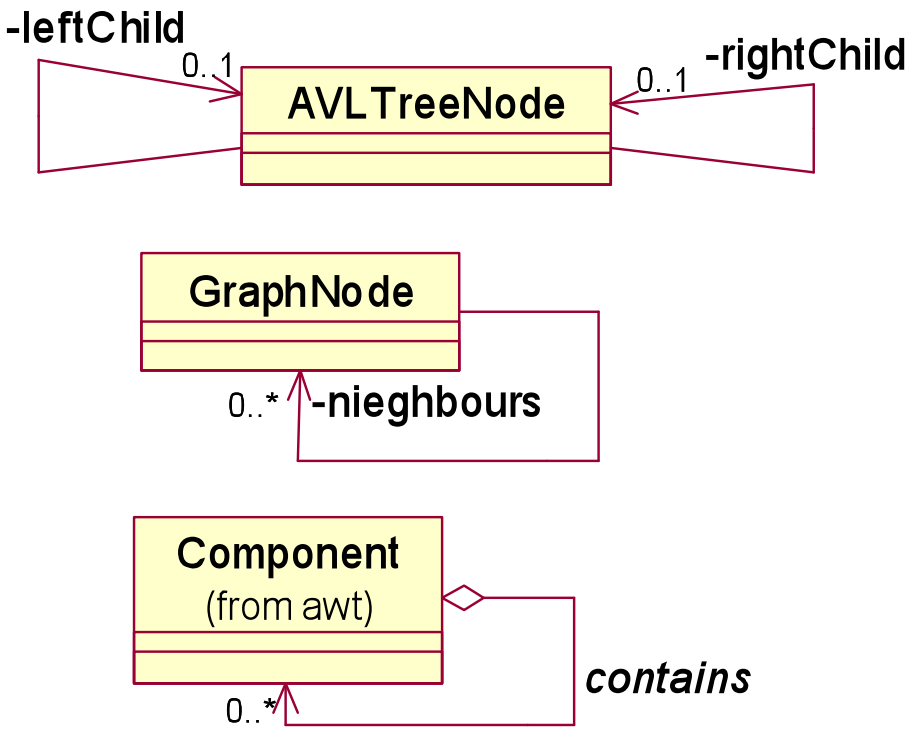


Composition

- Composition is a special case of aggregation.
- Composition is appropriate when
  - each part is a part of one whole
  - the lifetime of the whole and the part are the same.
- Graphically it uses a solid diamond

Recursive associations

- Associations may relate an class to itself.
- The objects of the class may or may not be associated with themselves.
- (For example, the left and right children of a node would not be that node. But a GraphNode object might be its own neighbour.)



Associations vs. attributes

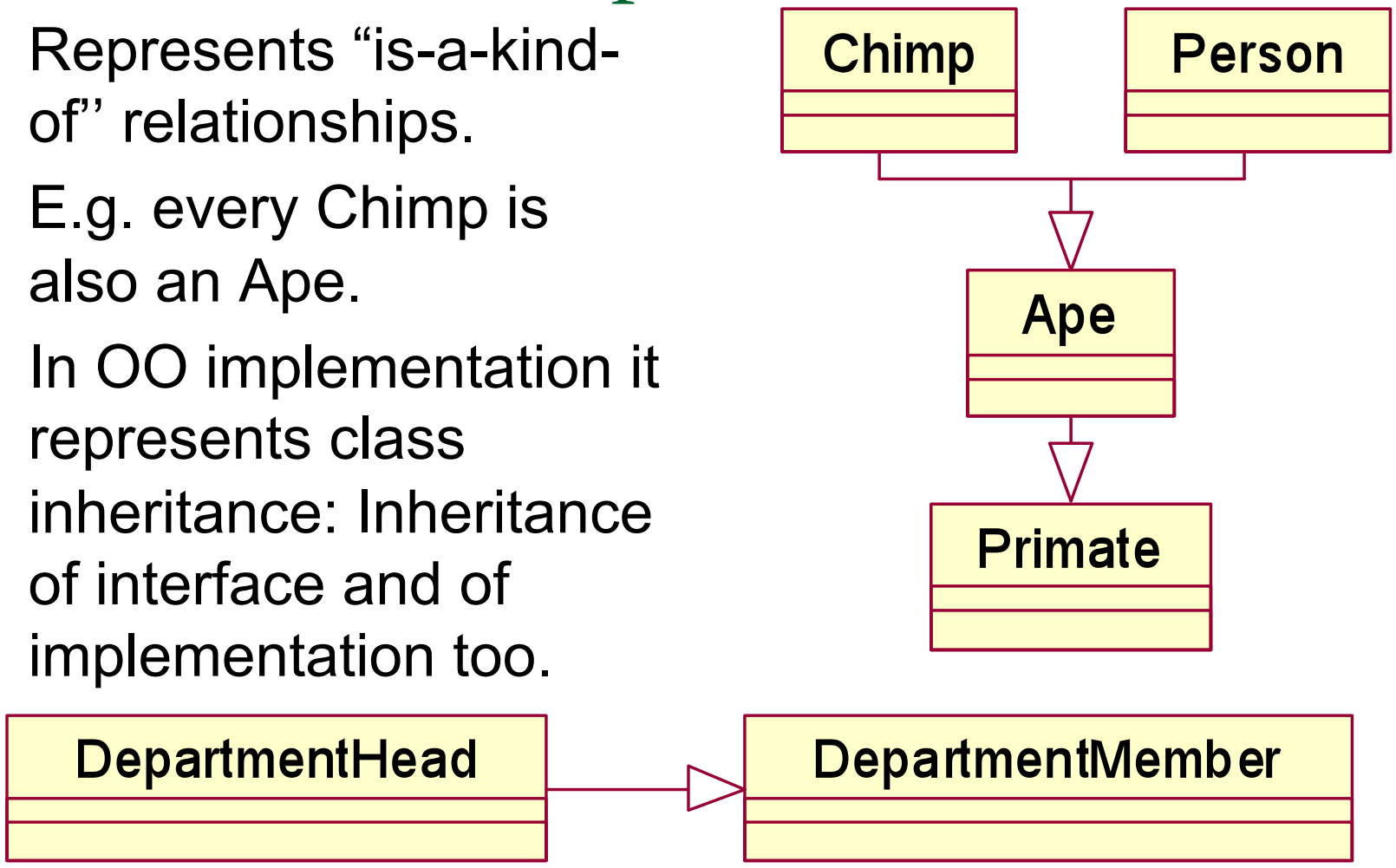
- Both are usually implemented by fields (a.k.a. data members).
- Use attributes for primitive types such as integer, boolean, char, etc, and pointers to such.
- Use association (or aggregation) for pointers that point to classes or interfaces.
- Use composition for data members that are classes. (Not possible in Java).
- Use composition if the life time of the part is identical to or contained in the lifetime of the whole.

Degrees of belonging

- Attribute. Lifetime of attribute equals life time of object that contains it.
- Composition. Lifetime of the part equals or is, by design, nested within the lifetime of the whole.
- Aggregation. Whole-part relationship, but parts could be parts of several wholes, or could migrate from one container to another.
- Association. Relationship is not part/whole.

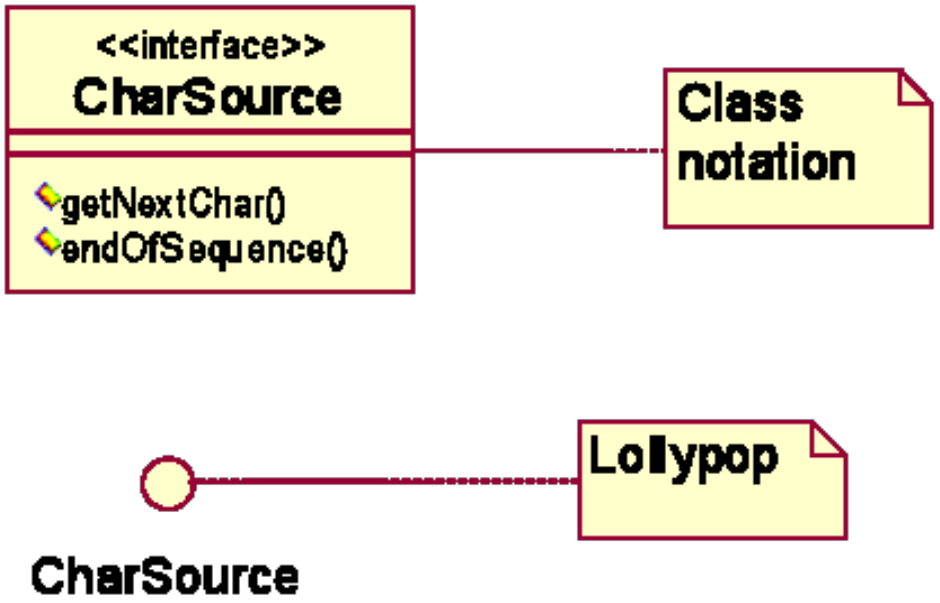
Generalization/Specialization

- Represents “is-a-kind-of” relationships.
- E.g. every Chimp is also an Ape.
- In OO implementation it represents class inheritance: Inheritance of interface and of implementation too.



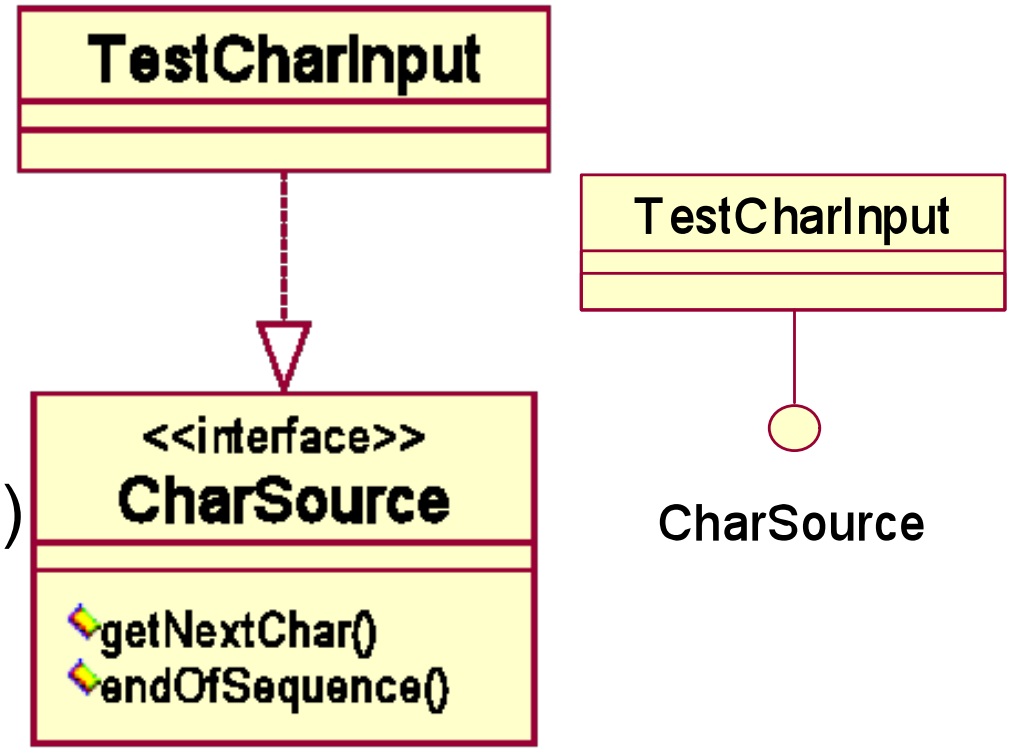
Interfaces

- Interfaces are classes that have no associated implementation.
- I.e.
  - no attributes,
  - no implementations for any operations
- In UML use either stereotype to indicate an interface, or "lollypop"



Realization

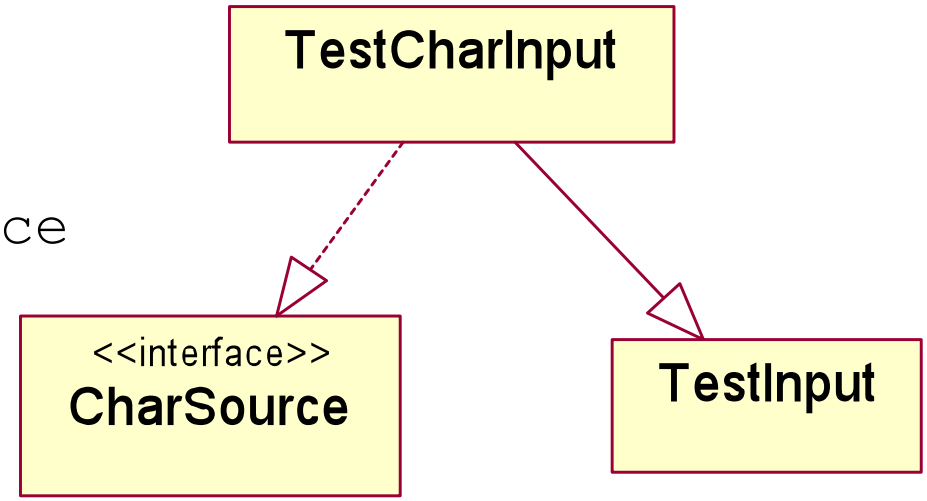
- Classes “specialize” classes, but “realize” interfaces. Similar concept, similar notation. (Note dashes)
- Choice of notations. Diagrams at right are equivalent.



Generalization/Specialization and Realization in Java

UML terminology	Java terminology
C specializes D	C extends D
C realizes D	C implements D

```
class TestCharInput
    extends TestInput
    implements CharSource
{
    …
}
```



The Substitution Principle

- Suppose class C specializes class D

or class C realizes interface D

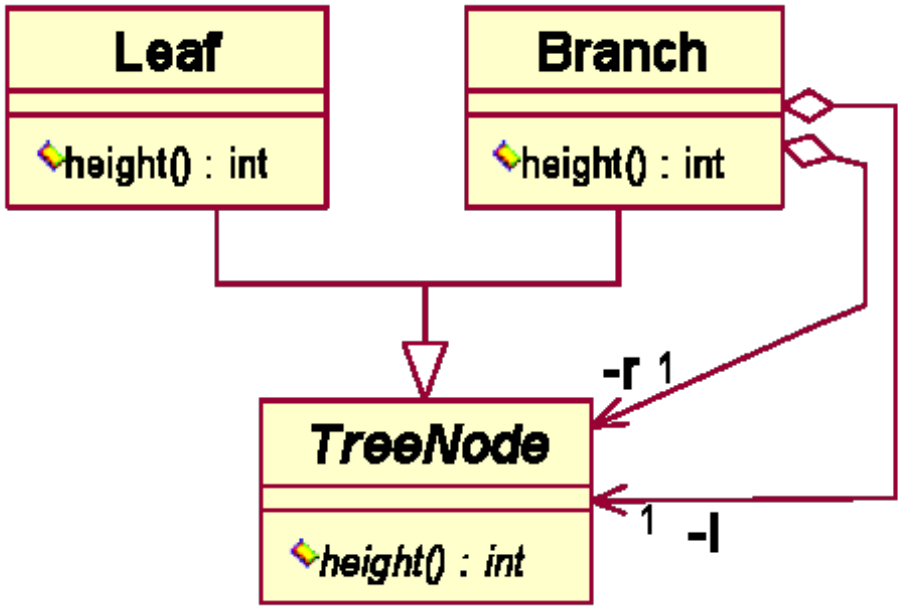
- Then any properties that should hold true of all D objects should hold true of all C objects.
- Question:
  - Should Rectangle specialize Square, or
  - should Square specialize Rectangle, or
  - neither should specialize the other?
- More on this later.

Abstract operations

- An operation O is "abstract" in class C if it does not have an implementation in class C.
- The implementation of the operation will be filled in in specializations of C.
-
```
abstract class TreeNode {
  abstract int height() ; … }
class Leaf extends TreeNode {
  int height() { return 1 ; } … }
class Branch extends TreeNode {
  TreeNode r, l ;
  int height(){return Math.max( l.height(),
                                r.height ) ; } … }
```

Abstract in VP

- In VP classes are made abstract with a checkbox in the specification.
- Likewise for operations (class must be abstract first).
- Italics or slanted text indicate abstractness



Abstract and Concrete classes

- Classes that have abstract operations can not be instantiated --- since this would mean that there is no implementation associated with one of the object's operations
- Classes that can not be instantiated are called ***abstract classes***.
- Classes that can be are called ***concrete***
- In UML use the <<abstract>> stereotype for abstract classes and operations.
  - Alternatively: The name of the abstract class or operation is in italics.

Dependence

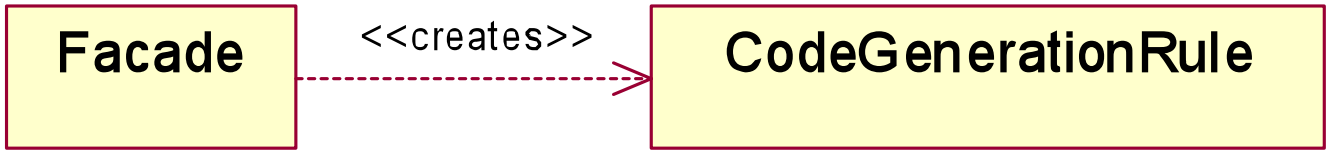
A class C depends on class D if the implementation or interface of C mentions D.

- C extends D or implements D
- C has a field of type D or pointer to D or array of D
- C creates a new D
- C has an operation that has a
  - parameter
  - local variable
  - return type

  of type D of a pointer to D or an array of D etc.

Dependence

- Often dependence is implicit in generalization or association relationships.
- When it is not, you may want to indicate dependence explicitly.
- Stereotypes and documentation can add detail.



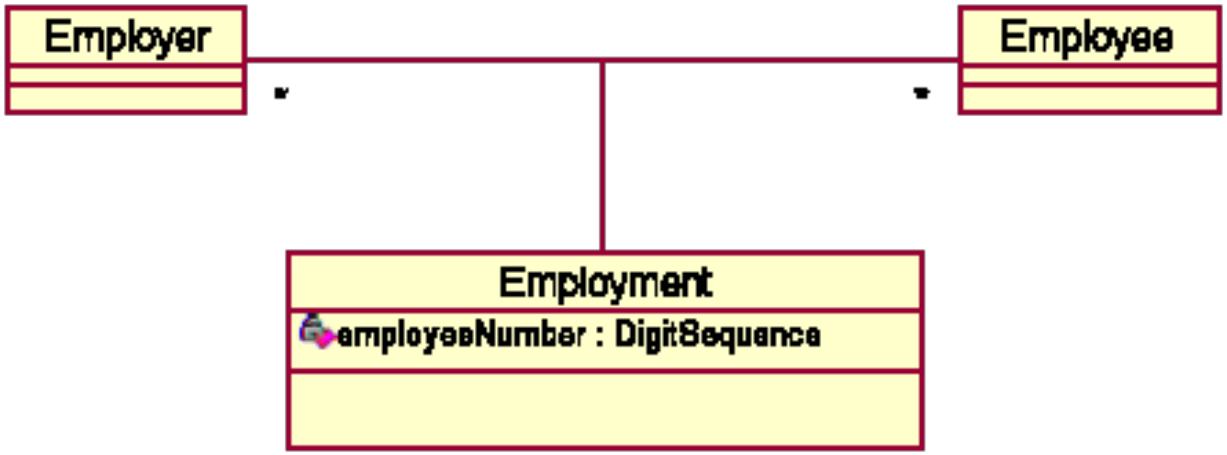
Dependence

- Dependence relations are important to note because unneeded dependence makes components
  - harder to reuse in another context
  - harder to isolate for testing
  - harder to write/understand/maintain, as the depended on classes must also be understood
- It is better to depend on an interface than on a class.
- More on this later.

More on Associations

Association Classes

- Sometimes associations need attributes themselves.
- An *employment* relationship between an employer and employee might have a employee number associated with it.



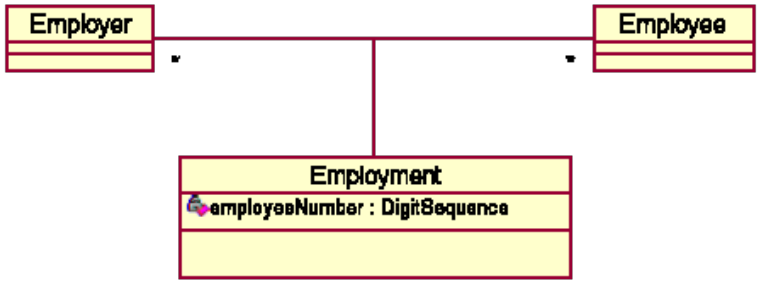
set, bag, ordered, and sequence

- In math a
  - A **set** is an unordered collection without duplication
  - A **bag** is an unordered collection with possible duplication
  - An **ordered set** is an ordered collection without duplication
  - A **sequence** is an ordered collection with possible duplication

	duplication	order
set		
bag	√	
ordered set		√
sequence	√	√

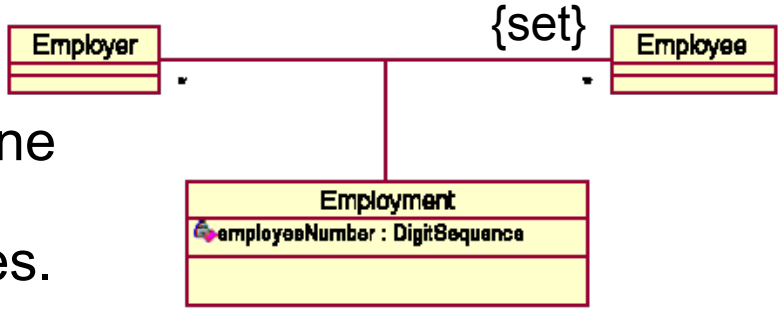
set, bag, ordered, and sequence

- By default, pairs are associated only once.
- Thus



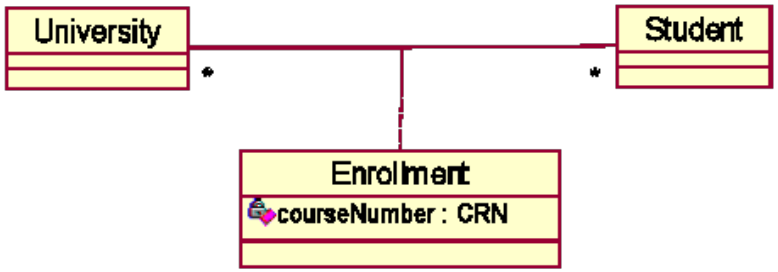
means that, while each employer can have many employees, each employee has only one employment link with each employer.

- Thus each employee can only have one employee number.
- Each employer has a ***set*** of employees. We can emphasise this point with an annotation:



set, bag, ordered, and sequence

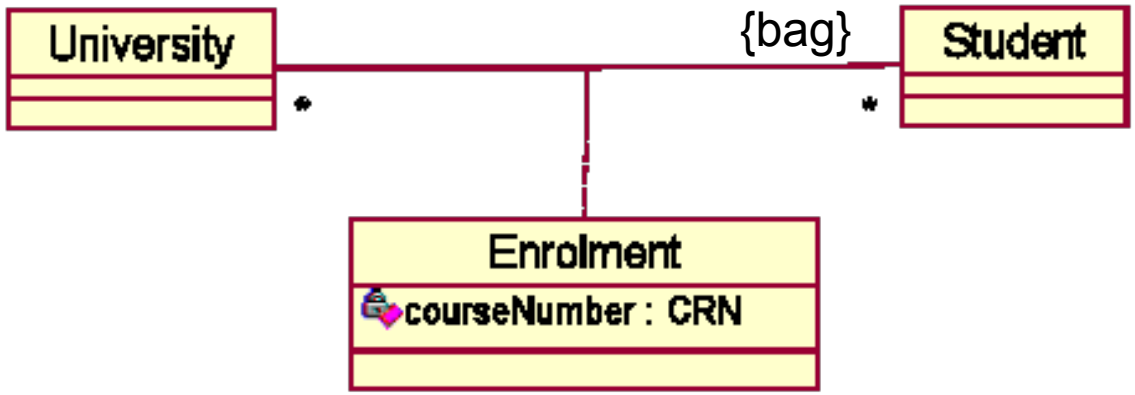
- Now consider this association



- It implies (incorrectly) that a student can only have one enrolment per university!.
- (Remember the {set} is the default.)

set, bag, ordered, and sequence

- We need a special annotation to say that the same (University/Student) pair can have multiple Enrolment links



set, bag, ordered, and sequence

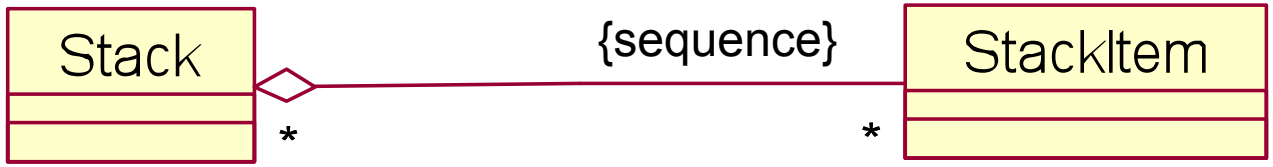
- Consider a OS's representation of a display screen. It has a *set* of windows, but furthermore the set is ordered.
- Use the **ordered** annotation for ordered sets

set, bag, ordered, and sequence

- Now consider a Stack of StackItems. The diagram

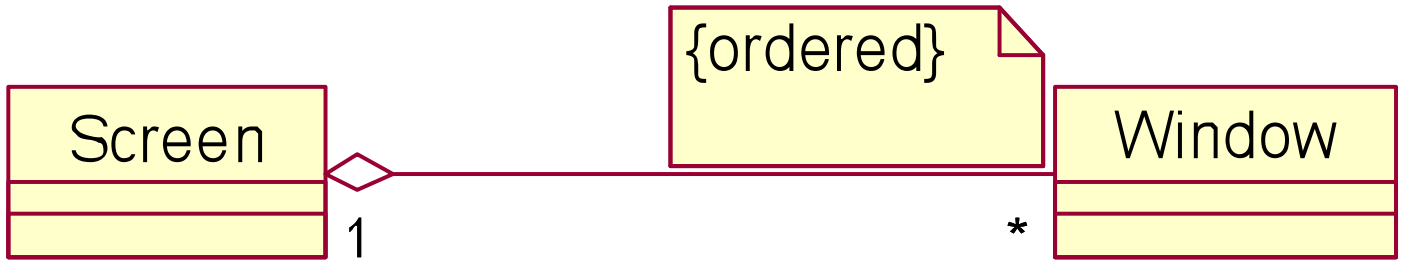
correctly shows that the same StackItem object may appear on the same stack more than once.

- But we may want to further indicate that the items on each stack are ordered



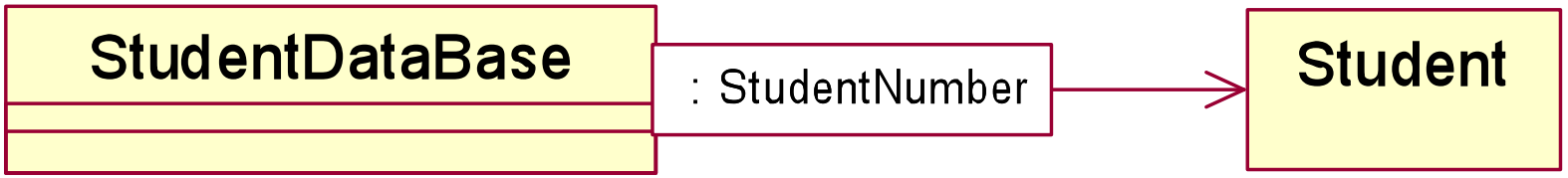
set, bag, ordered, and sequence

- Unfortunately the version of Rose we currently have does not support these annotations. As a work-around, use UML *notes*.



Qualified Associations

- An association may require other information. For example, given a StudentDataBase, one can find an associated Student “given a student number”
- Could be implemented by an array or a some kind of map structure (search tree or hash table).



n-ary Associations

- Normally associations are binary, but we can have *n*-ary associations for $n > 2$.
- Multiplicities are given assuming all other objects are fixed.
- Example: In a genogram application we might have