Modeling with UML

Class and Object Diagrams

Dr. Eman Saleh eman.saleh@uga.edu

The Class Diagram

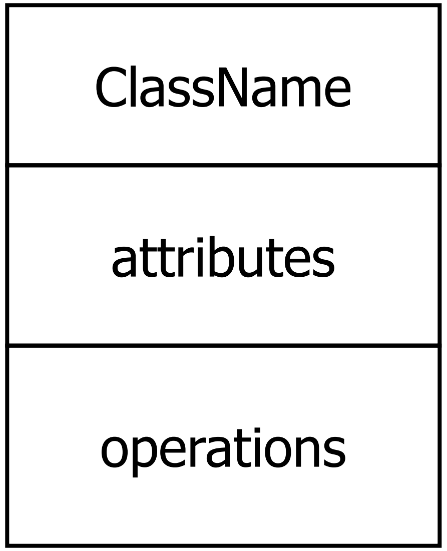
Class Diagram is a **static** structure representing:

- possible classes (object types), including their attributes and operations, and
- relationships among them

Object Diagram is a structure including

- objects (including their attribute values), and
- links (connections) among them

Classes



ClassName

attributes

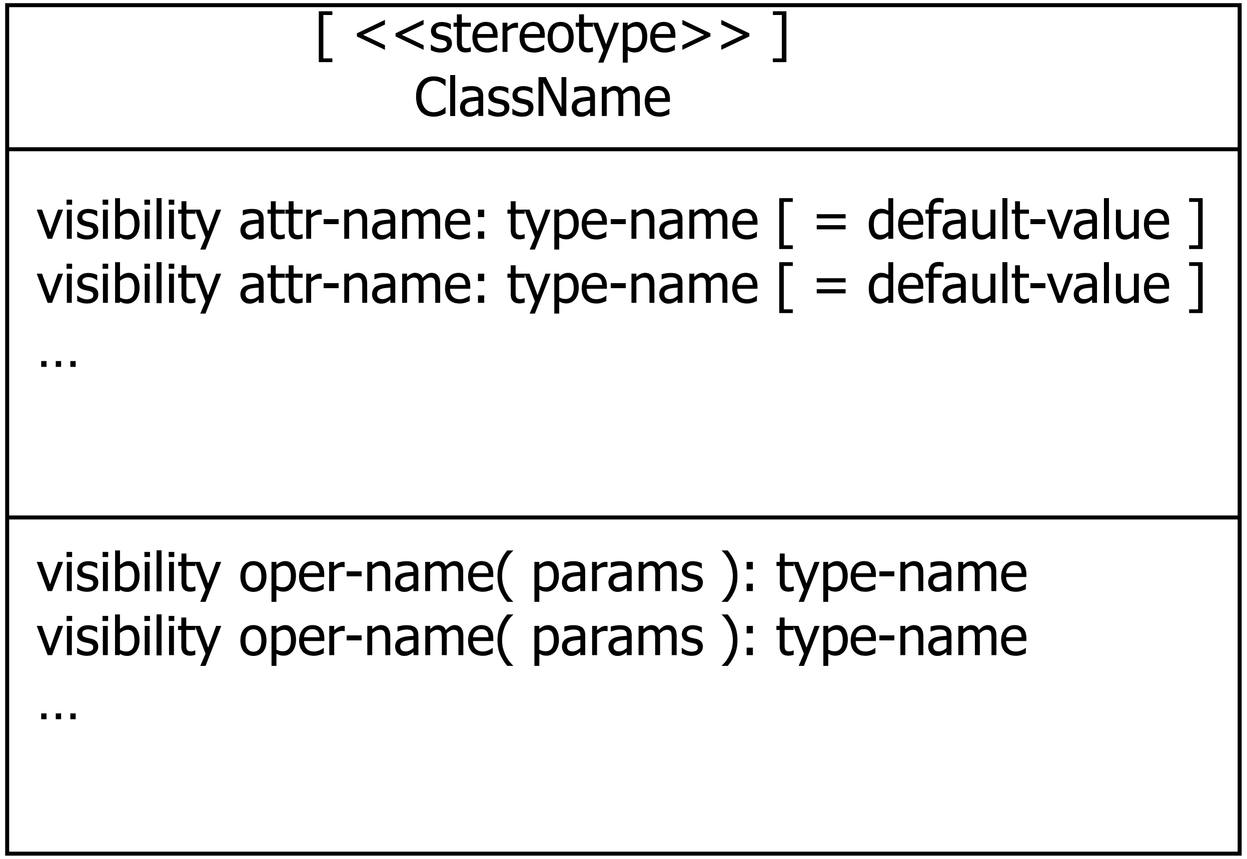
operations

A class is a description of a set of similar objects that share the same set of attributes, operations, relationships, and meaning

Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments

Classes

General format:

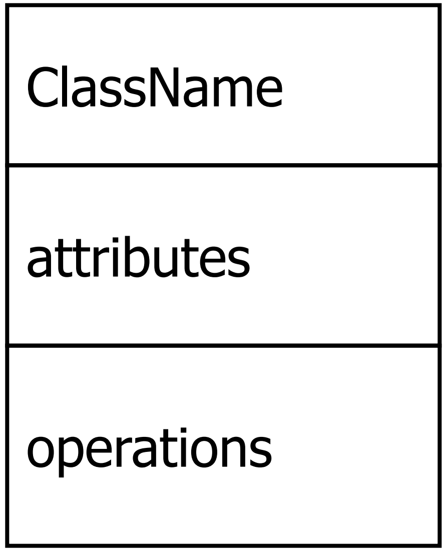


[<<stereotype>>]
ClassName

visibility attr-name: type-name [= default-value]
visibility attr-name: type-name [= default-value]
…

visibility oper-name(params): type-name
visibility oper-name(params): type-name
…

Class Representation



ClassName

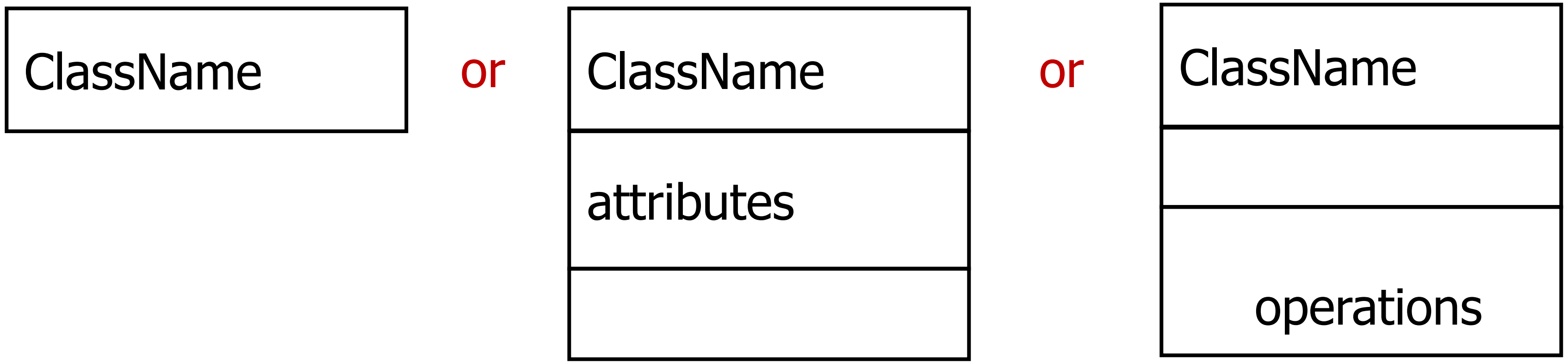
attributes

operations

The name of the class is the only required element. It always appears in the top-most compartment.

As always, use a noun (noun phrase) in singular form

Attributes and/or operations may be omitted.



ClassName

or

ClassName

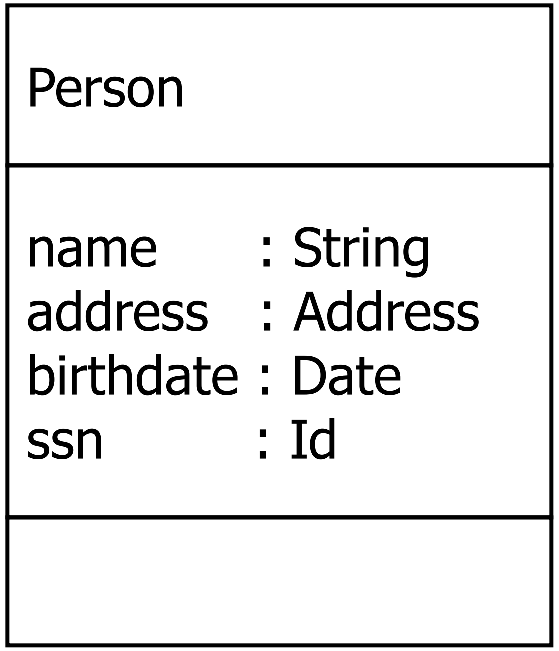
attributes

or

ClassName

operations

Class Attributes



Person

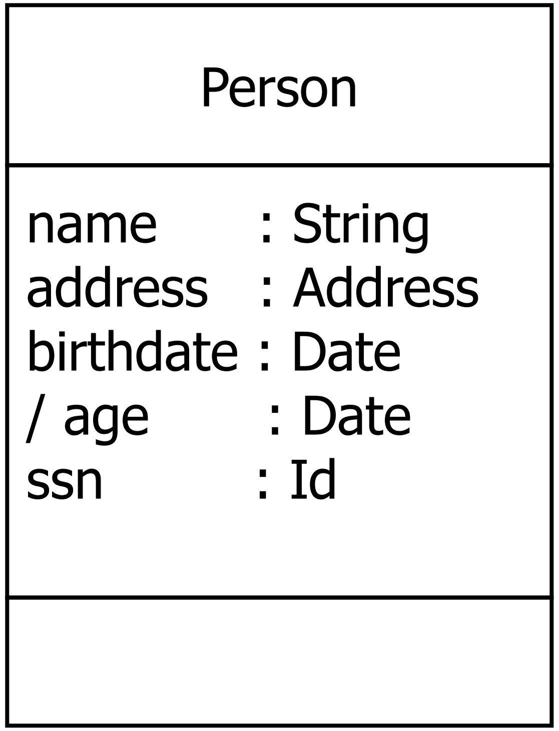
name : String
address : Address
birthdate : Date
ssn : Id

For example, Person, *NOT Persons*

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



Person

name : String
address : Address
birthdate : Date
/ age : Date
ssn : Id

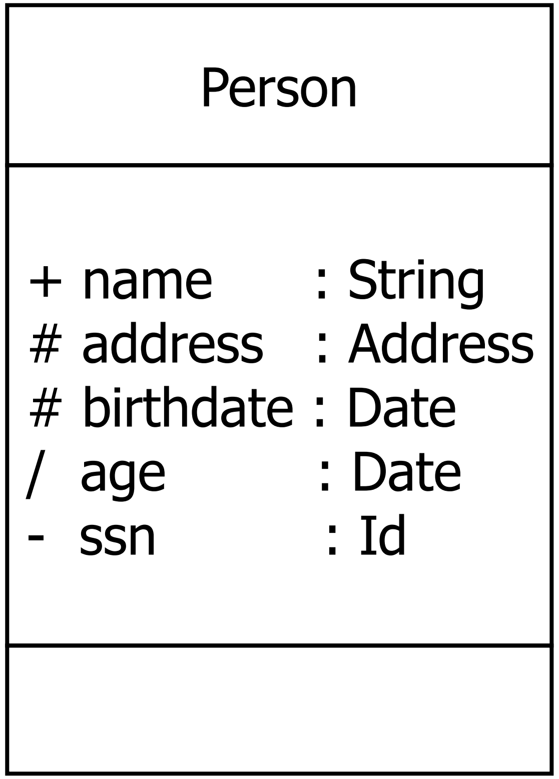
Attributes are usually listed in the form:

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



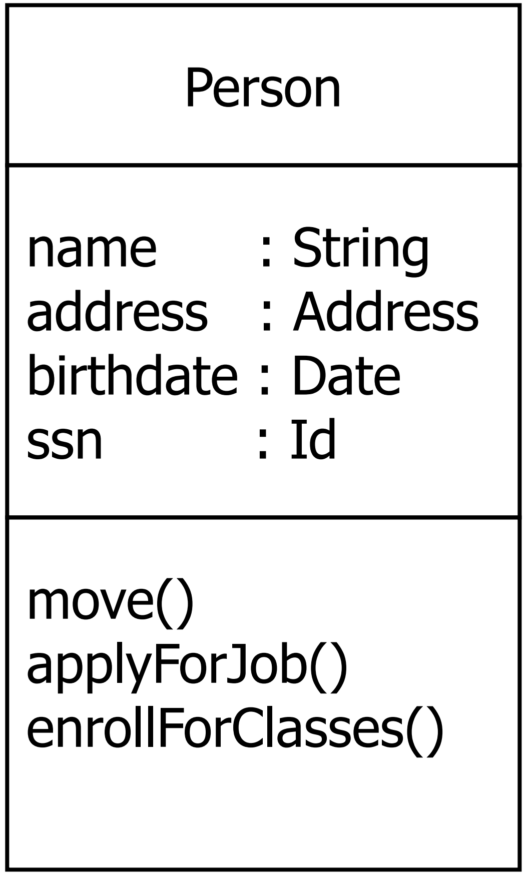
Person

+ name : String
# address : Address
# birthdate : Date
/ age : Date
- ssn : Id

Attribute visibility can be:

+ public
# protected
- private
~ package
/ derived

Class Operations



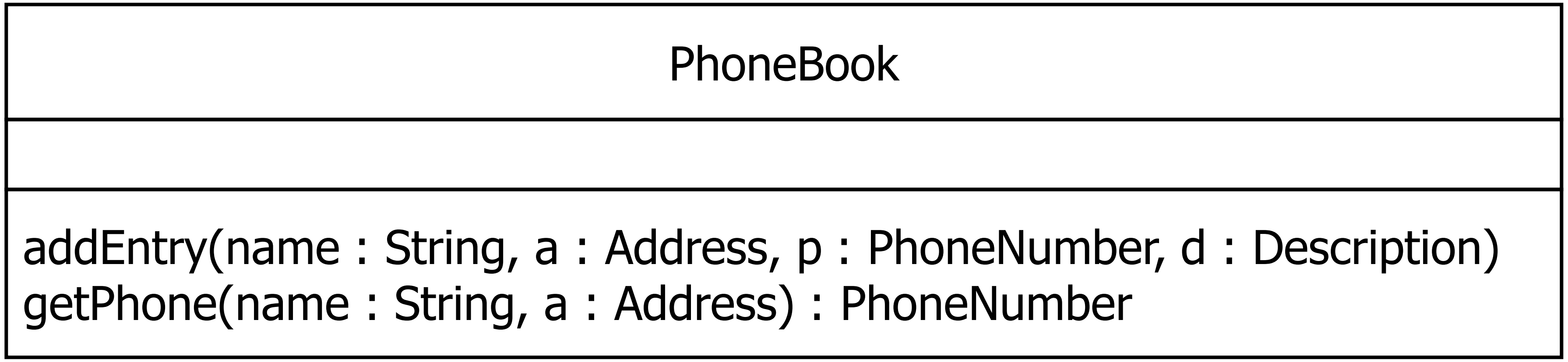
Person

name : String
address : Address
birthdate : Date
ssn : Id

move()
applyForJob()
enrollForClasses()

Operations describe the class behavior and appear in the third compartment.

Class Operations



PhoneBook

addEntry(name : String, a : Address, p : PhoneNumber, d : Description)
getPhone(name : String, a : Address) : PhoneNumber

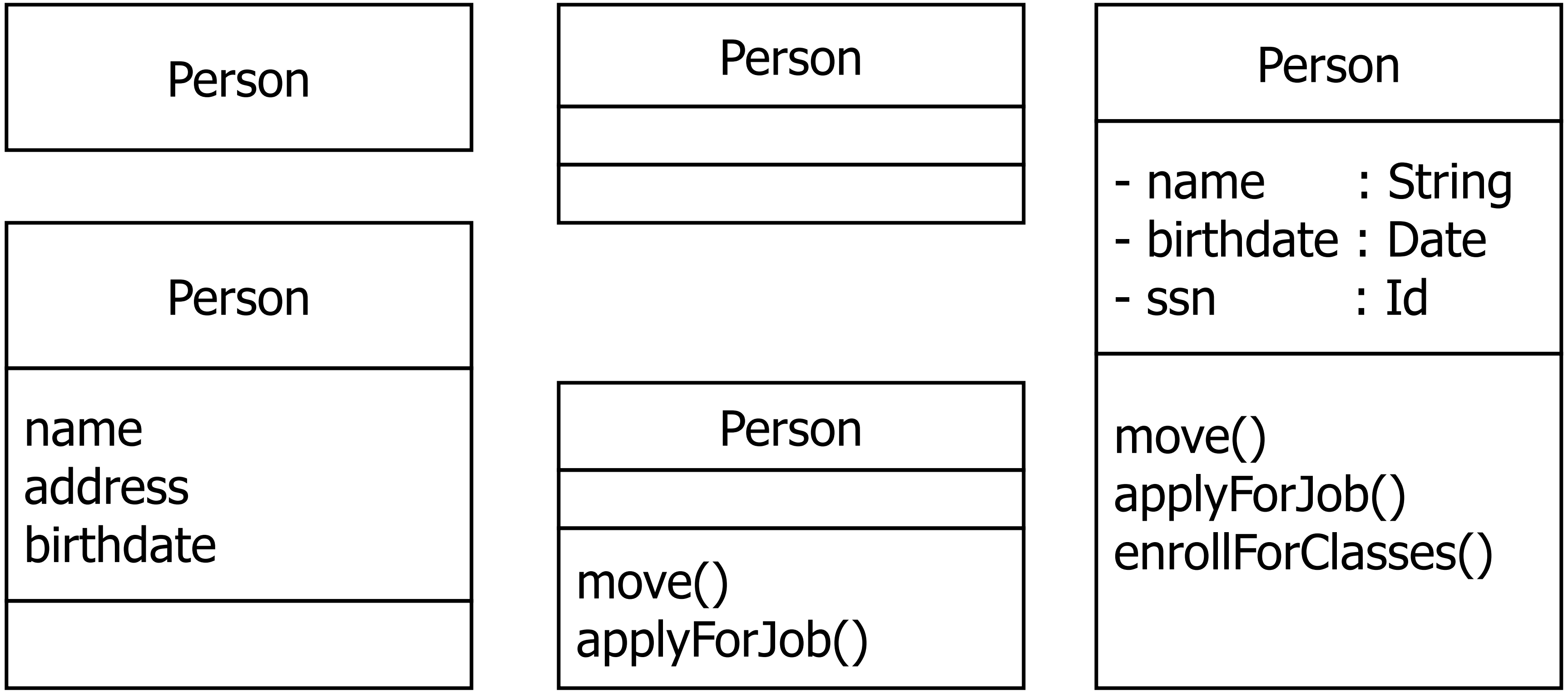
You can specify an operation by stating its signature:

- the name,
- type and default value of all parameters, and
- return type, for non-void operations (functions)

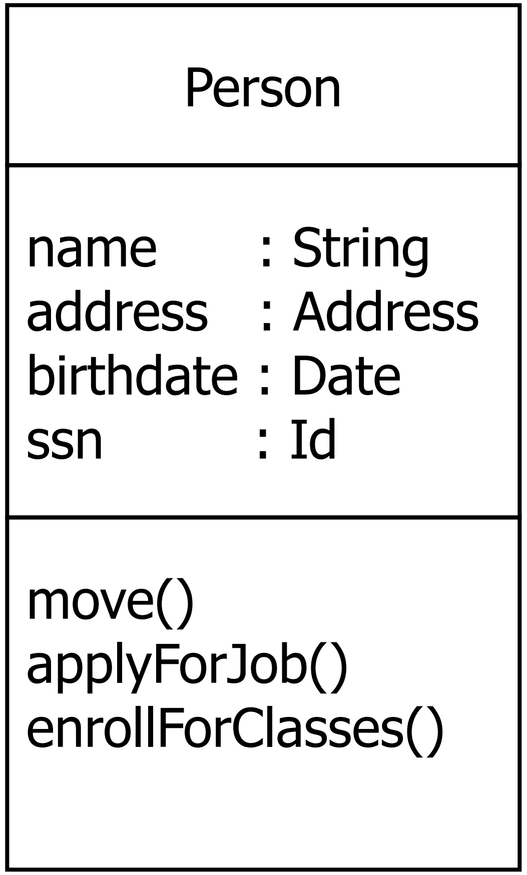
Note the order of the parameter name, colon, and type.

Class Representation

Class representations may be expanded incrementally, including more details, as they become available, or as needed.



Objects

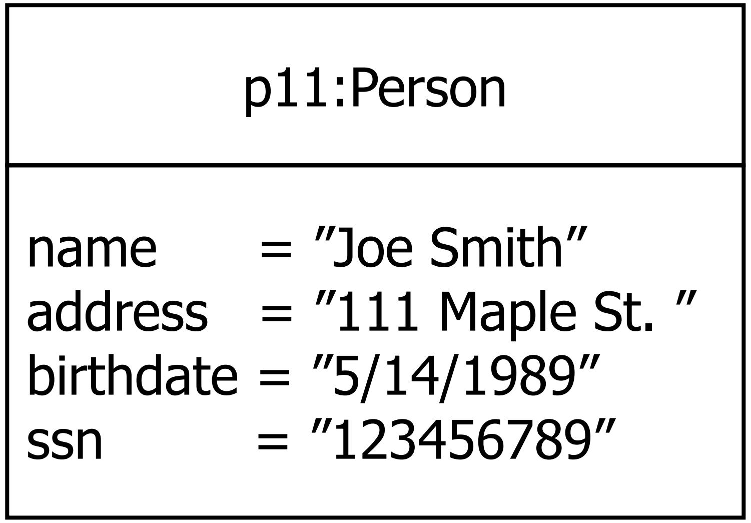


Person

name : String
address : Address
birthdate : Date
ssn : Id

move()
applyForJob()
enrollForClasses()

A Class



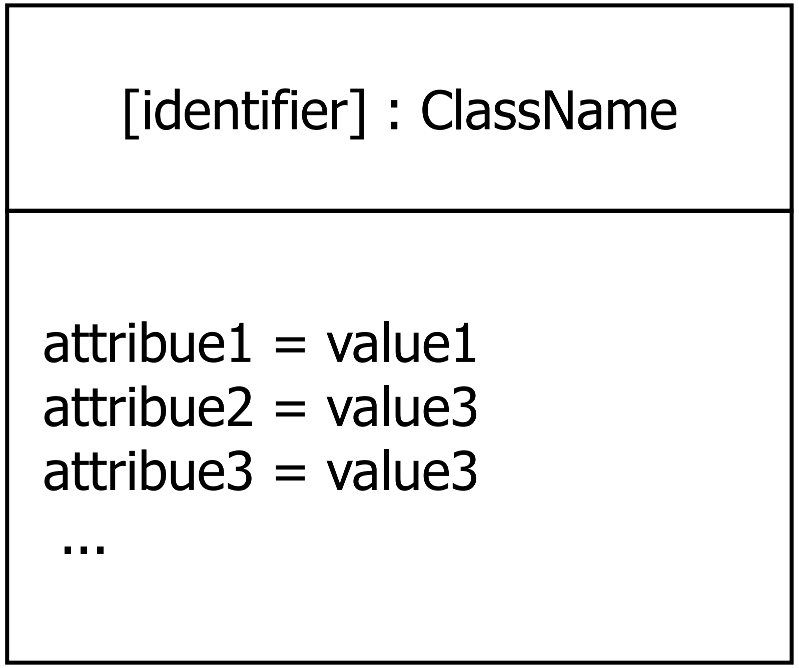
p11:Person

name = ”Joe Smith”
address = ”111 Maple St. ”
birthdate = ”5/14/1989”
ssn = ”123456789”

An object (instance)

Objects

General format of the object representation

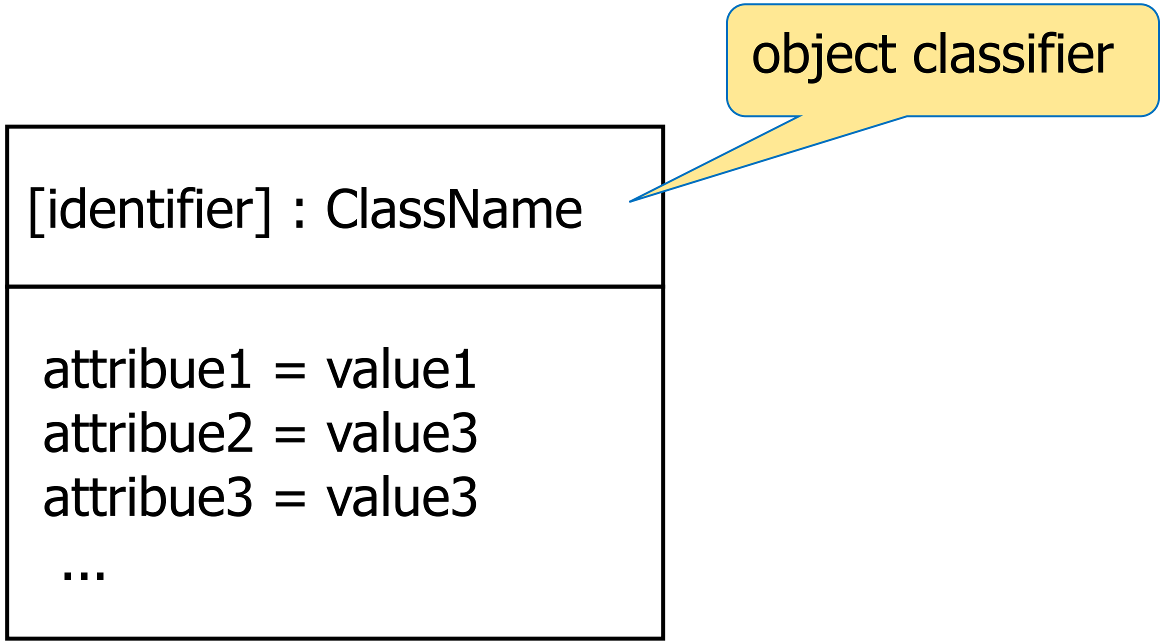


[identifier] : ClassName

attribue1 = value1
attribue2 = value3
attribue3 = value3
...

Objects

General format of the object representation



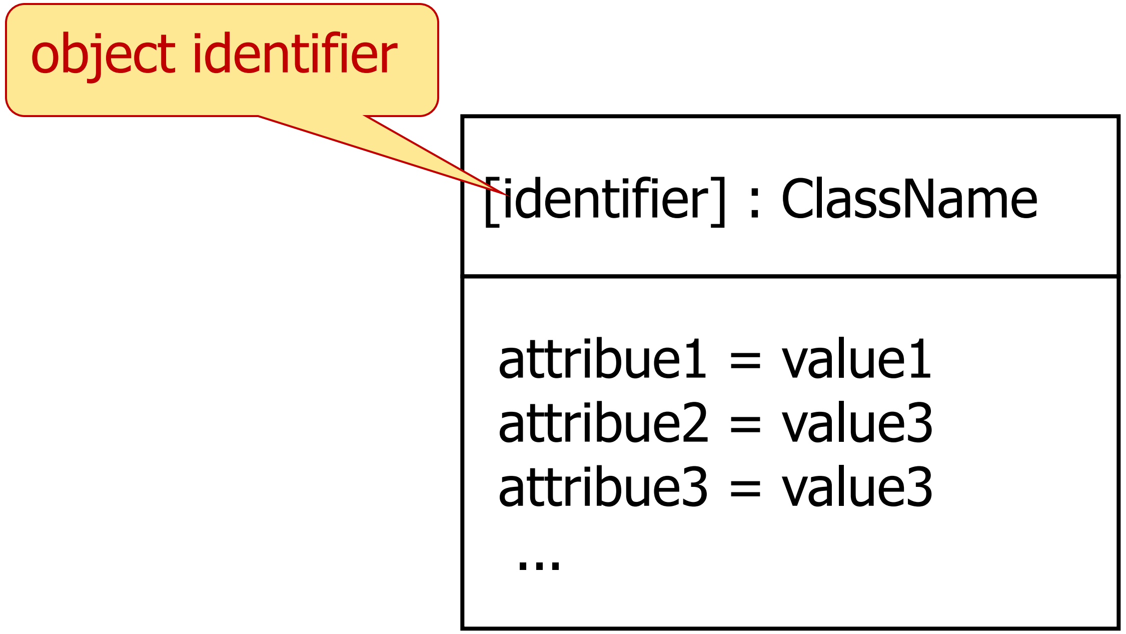
object classifier

[identifier] : ClassName

attribue1 = value1
attribue2 = value3
attribue3 = value3
...

Objects

General format of the object representation



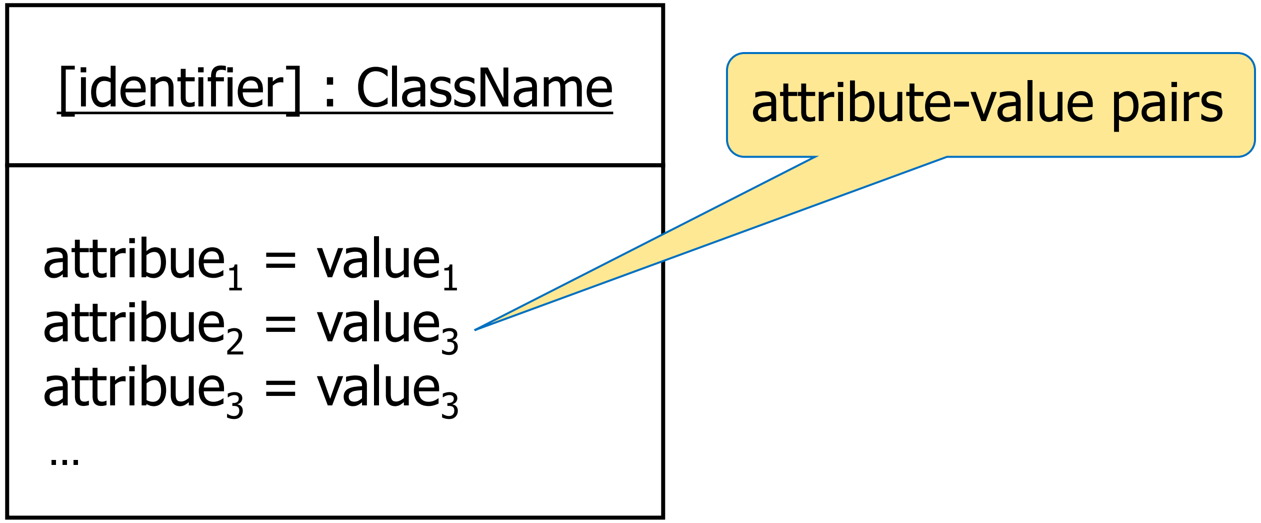
object identifier

[identifier] : ClassName

attribue1 = value1
attribue2 = value3
attribue3 = value3
...

Objects

General format of the object representation



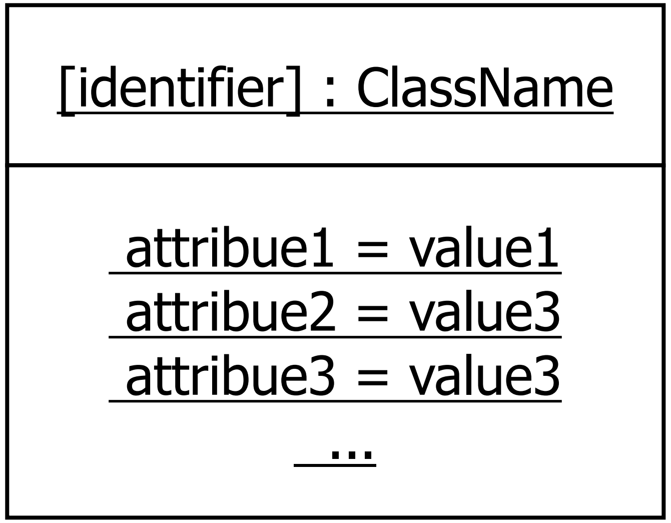
[identifier] : ClassName

attribue$_1$ = value$_1$
attribue$_2$ = value$_3$
attribue$_3$ = value$_3$
…

attribute-value pairs

Objects

General format of the object representation



[identifier] : ClassName

attribue1 = value1
attribue2 = value3
attribue3 = value3
...

NO OPERATIONS ARE LISTED!

Relationships

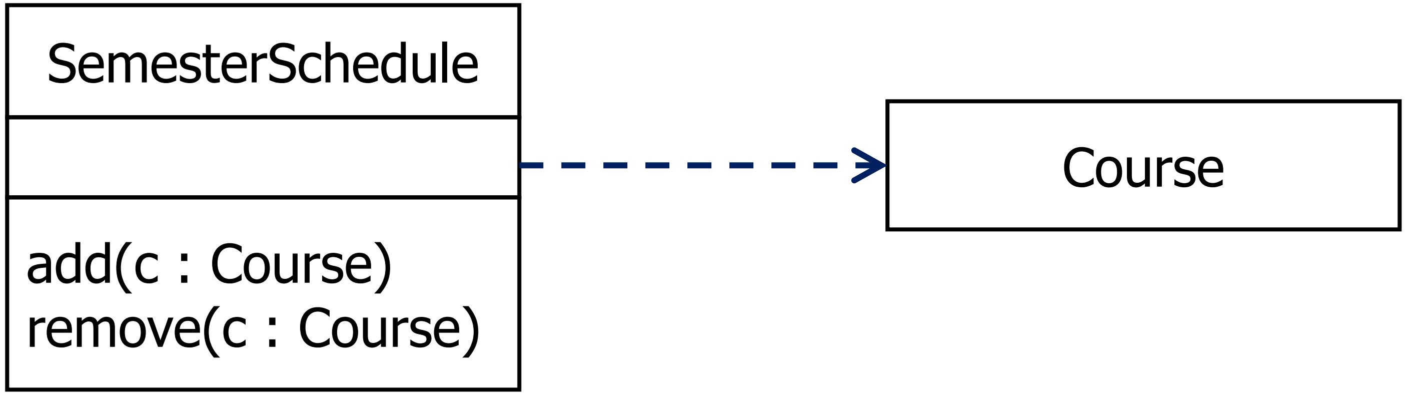
In UML, object interconnections (logical or physical), are modeled as relationships

There are three kinds of relationships in UML:

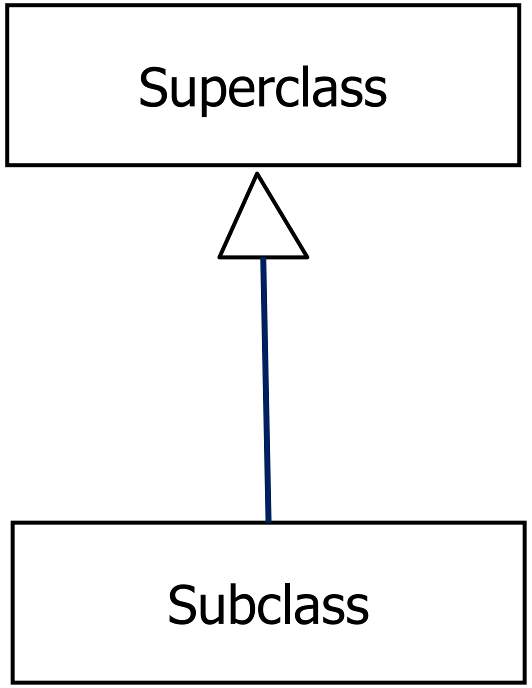
- dependencies
- associations
- generalizations

Dependency Relationship

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 Relationship



A generalization connects a subclass to its superclass

It denotes an inheritance of attributes, behavior, and relationships from the superclass to the subclass

Subclass is a specialization of the more general superclass

Generalization Relationship

- Generalization introduces a taxonomy

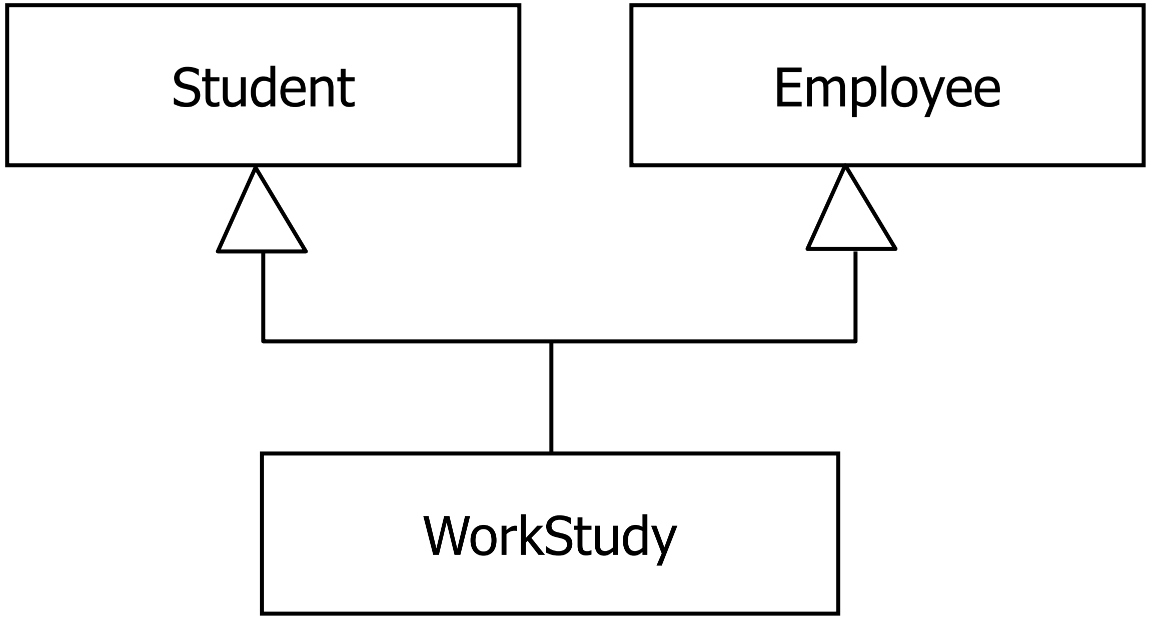
  ”A taxonomic (taxonomy is the science of classification) relationship between a more general element and a more specific element.”
- A subclass (a more specific classifier) is fully consistent with a superclass (a more general classifier)
- A more specific element usually contains more information
- Generalization is also known as inheritance
- Introduces the is-a relationship

Generalization Semantics

- A subclass inherits all attributes and operations from its superclass
- A subclass also inherits all associations in which its superclass participates
- Visibility rules apply:
  - private members are inherited, but are not visible to the subclass
  - protected members are inherited and visible

Generalization Relationships

- A class may have multiple parents (super classes)

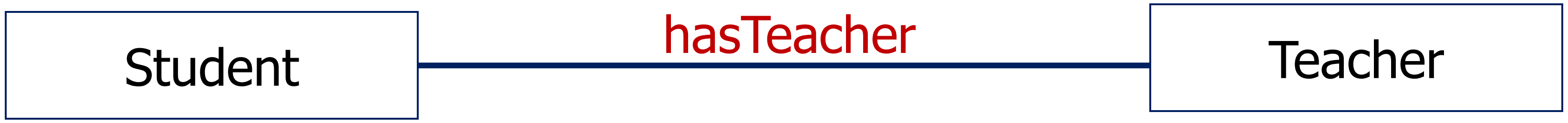


Association Relationships

If two classes in a model are related in some way, or need to communicate with each other, there must be a connection between them

An association denotes that connection

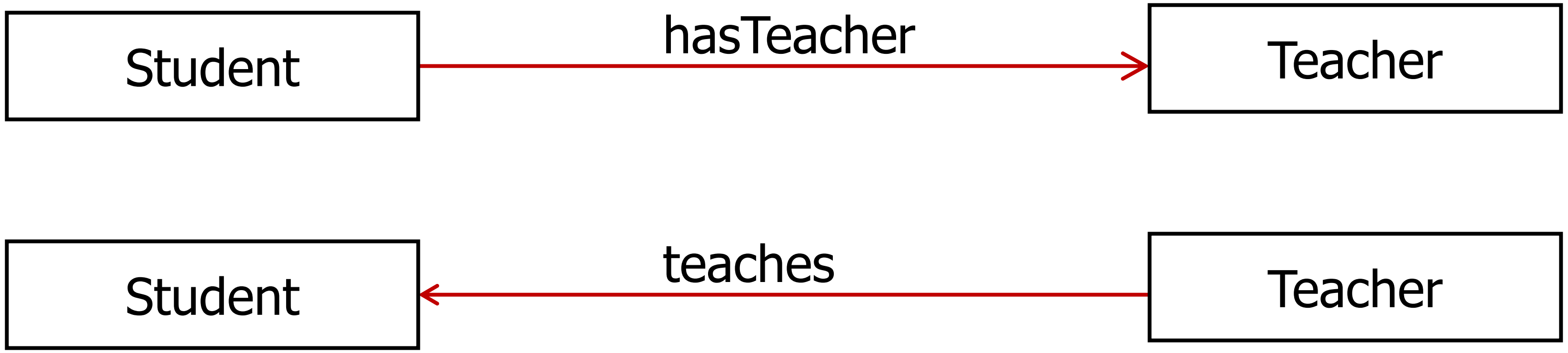
The name of an association represents its meaning
It should be a *verb* or a *verb phrase*



Association Relationships

An association may be navigable or non-navigable

A navigable association has an arrow indicating the direction of navigability

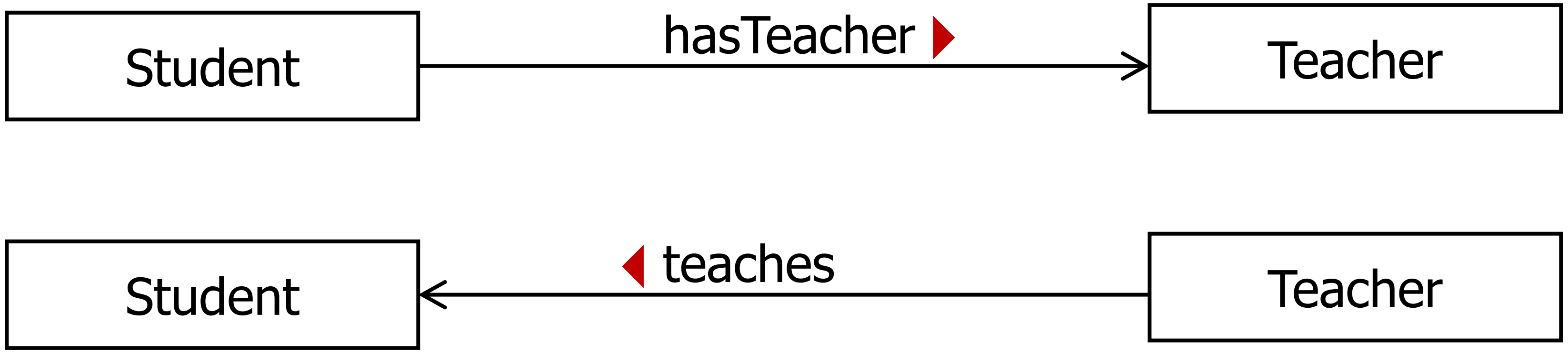


By default, an association is navigable in both directions and the arrows are omitted

Navigability does not have to be specified (no arrows)

Association Relationships

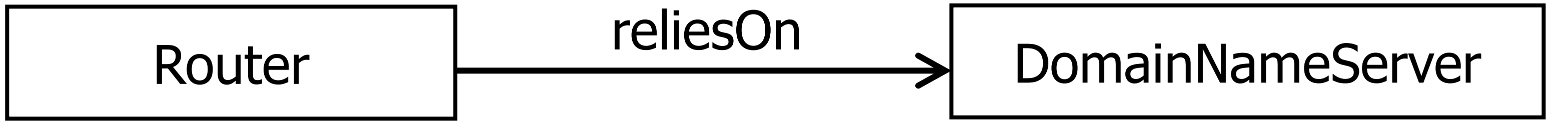
The name of an association may indicate its directionality, indicated by an arrow head next to the association name



Association Relationships

Here, a ***Router* object** requests services from a ***DNS*** object by sending messages to (invoking the operations of) the server.

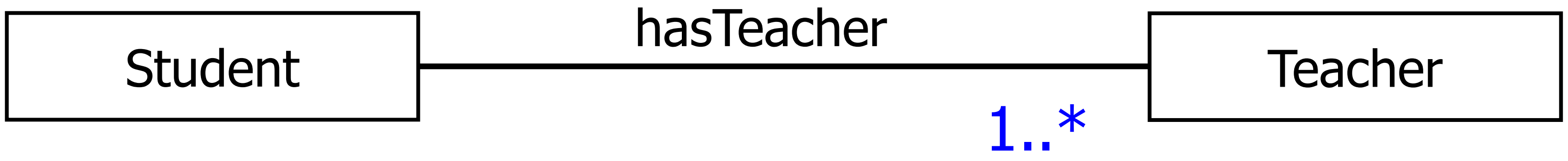
The direction of the association indicates that the server has no knowledge of the *Router*.



Association Relationships

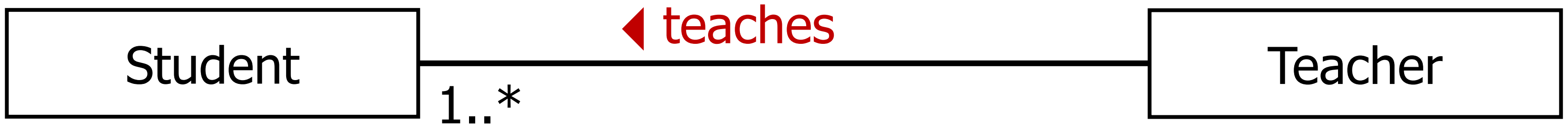
We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association.

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



Association Relationships

This example indicates that **every** Teacher teaches one or more *Students*:

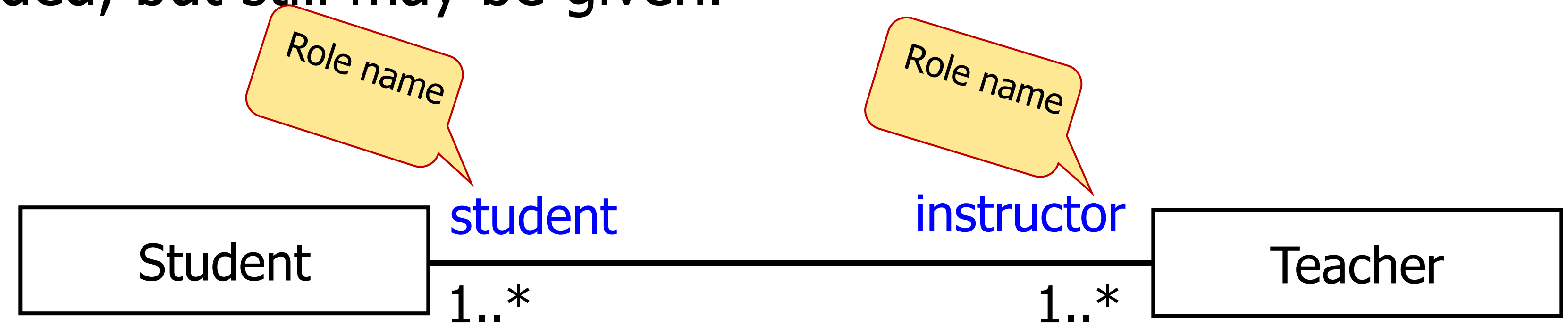


Association Relationships

We can also indicate how a class participates in an association by specifying its role in the association.

A role name is a name of an association’s endpoint.

If the role names are specified, the association name is usually not needed, but still may be given.



- Activity 11 is due tonight (2 attempts)
- Activity 12 will be done **In class** On Monday, prepare by reading the Library case study

Association Multiplicities

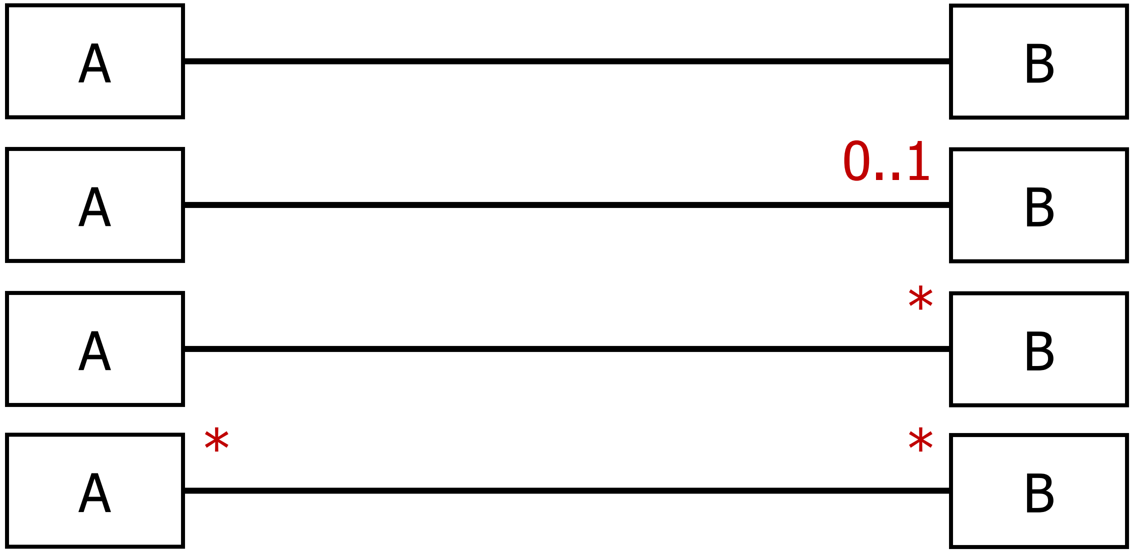
Multiplicity may take the general form of

M .. N, where M and N may be non-negative integers (M≤N) or *, to indicate any non-negative number

M .. M is usually written as just M

0 .. * is usually written as just *

1 .. 1 is just 1, or not provided at all (the default)



one-to-one

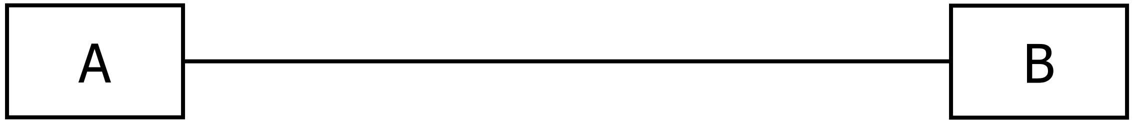
optional

one-to-many

many-to-many

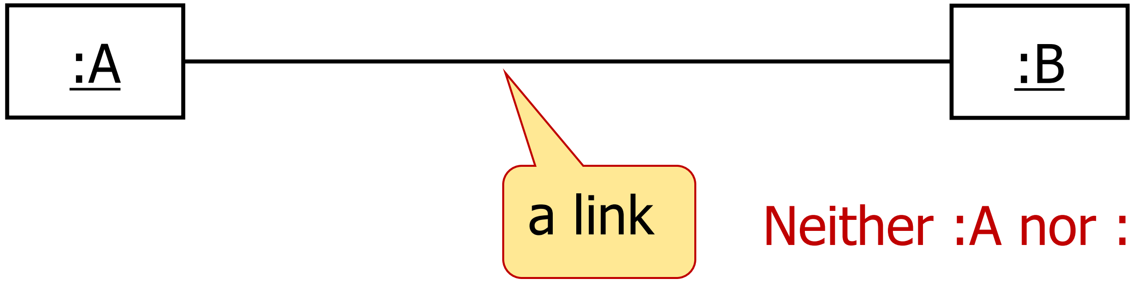
Association Multiplicities

Multiplicity should be thought of from the point of a single object on the other side of the association. For example,



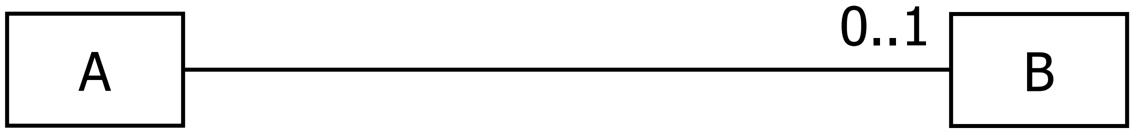
one-to-one

A single A object must be linked to exactly one B object, and vice versa.



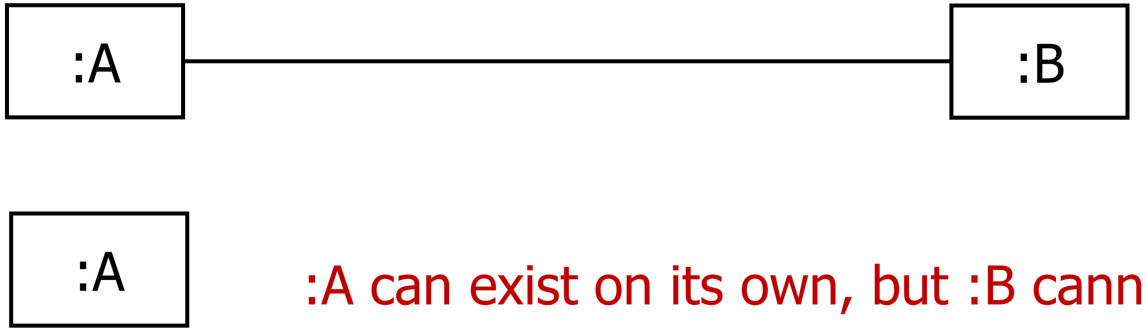
Neither :A nor :B can exist on its own

Association Multiplicities



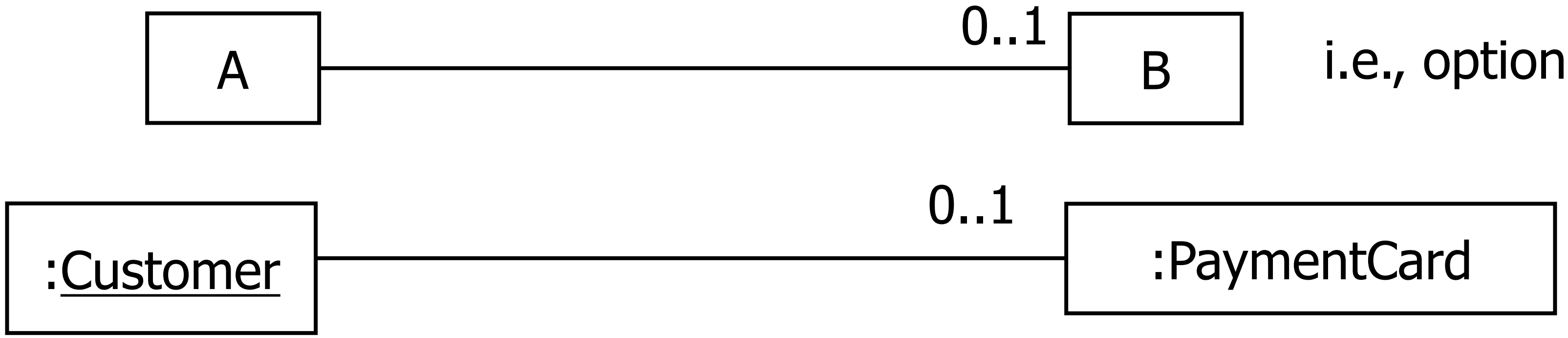
i.e., optional

A single A object may be linked to exactly one B object or none at all. But a B object must be linked to exactly one A object.



:A can exist on its own, but :B cannot

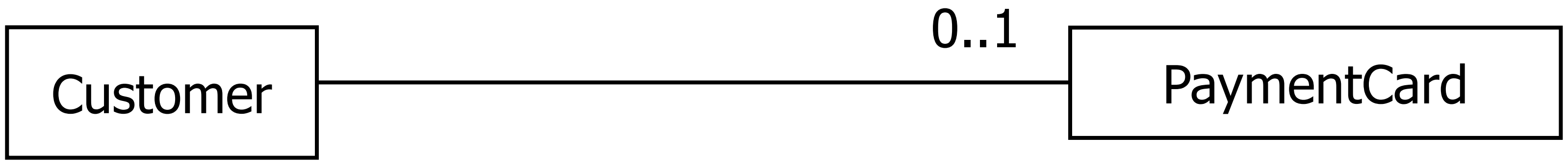
Example



i.e., optional

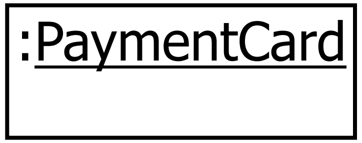
A customer object can exist by its own
A payment card object can exist on its own

Exercise:

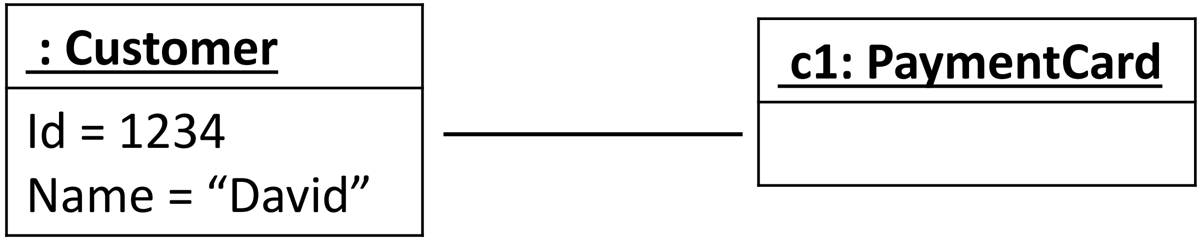


Which of the following object diagrams are consistent with the above class diagram

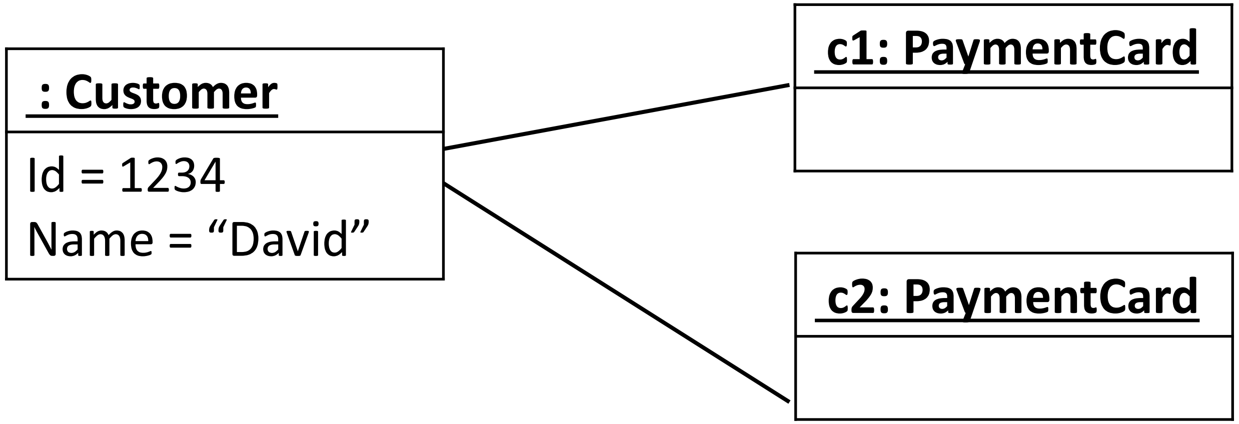
1)



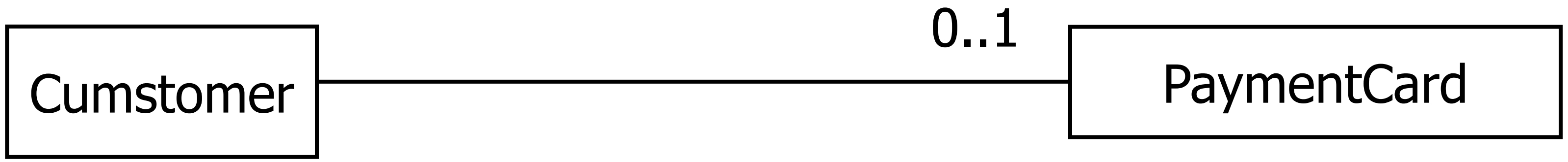
2)



3)



Exercise:

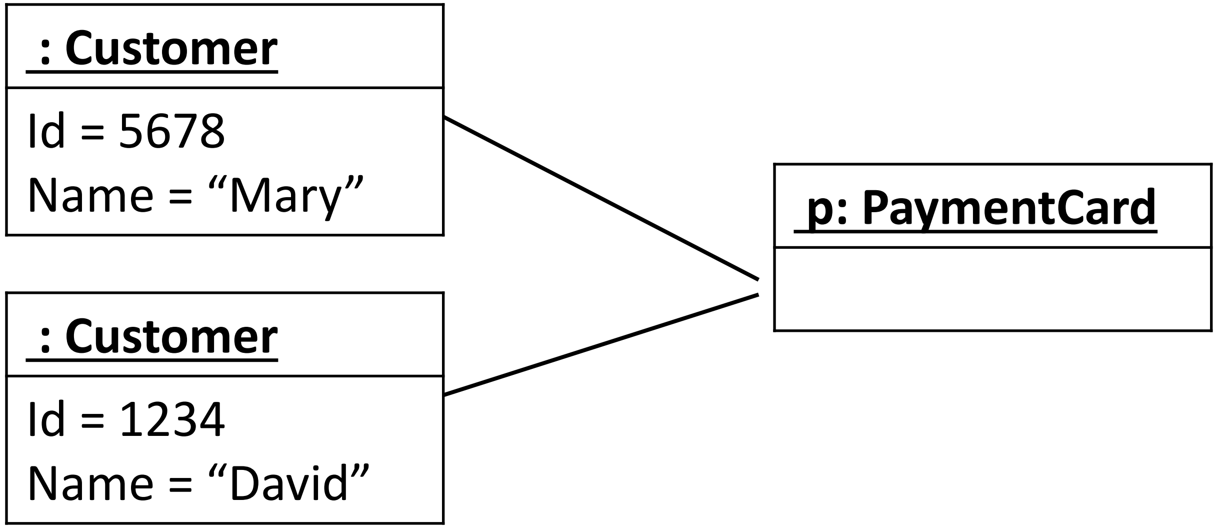

Cumstomer

0..1

PaymentCard

Which of the following object diagrams are consistent with the above class diagram

4)


: Customer

Id = 5678

Name = “Mary”

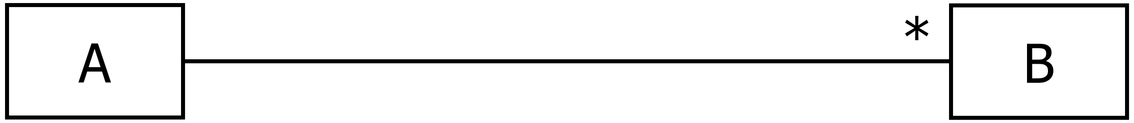
: Customer

Id = 1234

Name = “David”

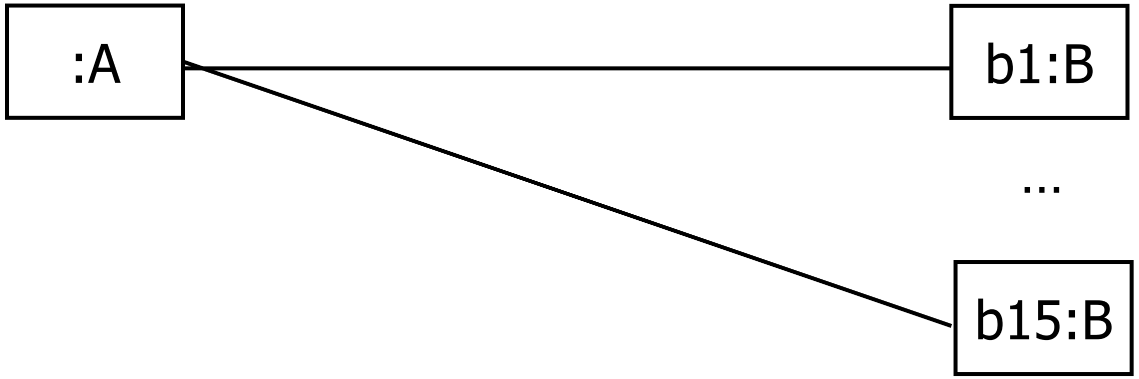
p: PaymentCard

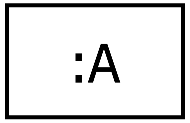
Association Relationships



one-to-many

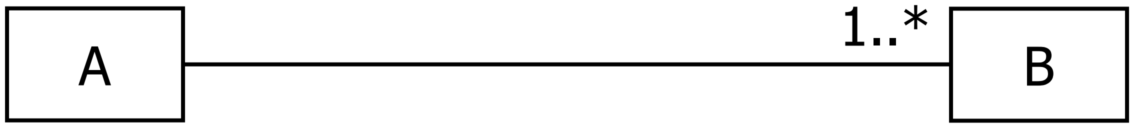
A single A object may be linked to zero or more B objects, while a B object must be linked to exactly one A object.





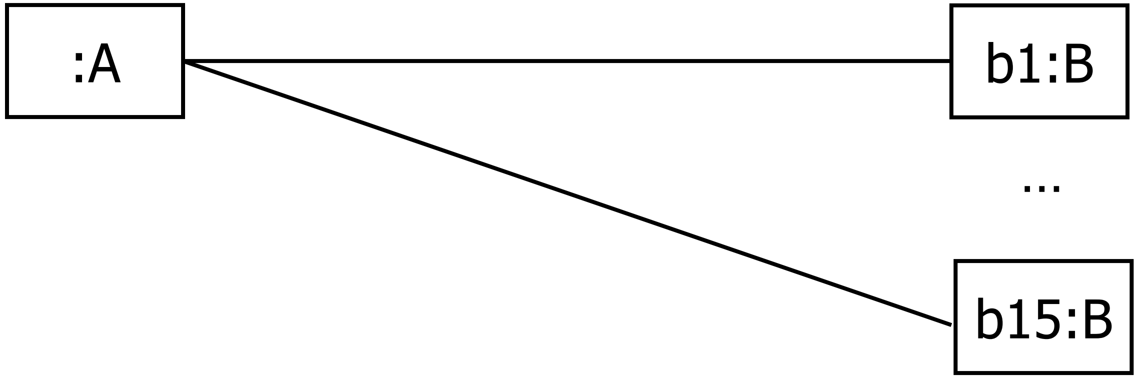
:A can exist on its own; :B objects cannot

Association Relationships



one-to-many

A single A object must be linked to one or more B objects. A B object must be linked to exactly one A object.

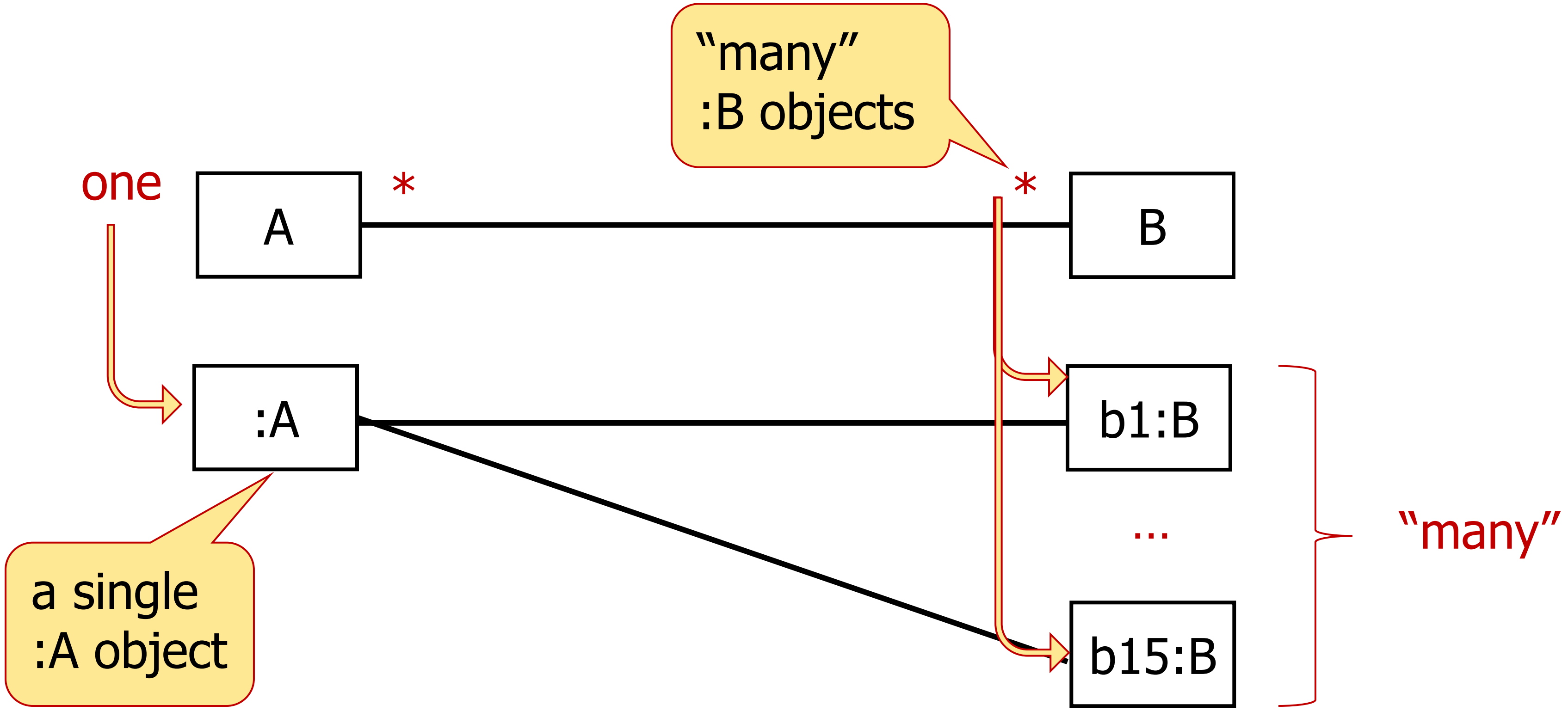


Neither :A nor :B objects can exist on their own

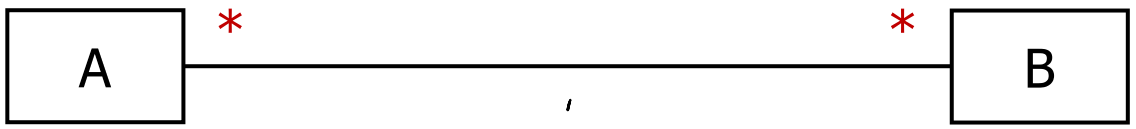
Cardinality of an Association

An association is a set of links that can exist among objects of the associated classes.

The multiplicity at one endpoint restricts the cardinality of the set of links that connect to a single object at the other endpoint.

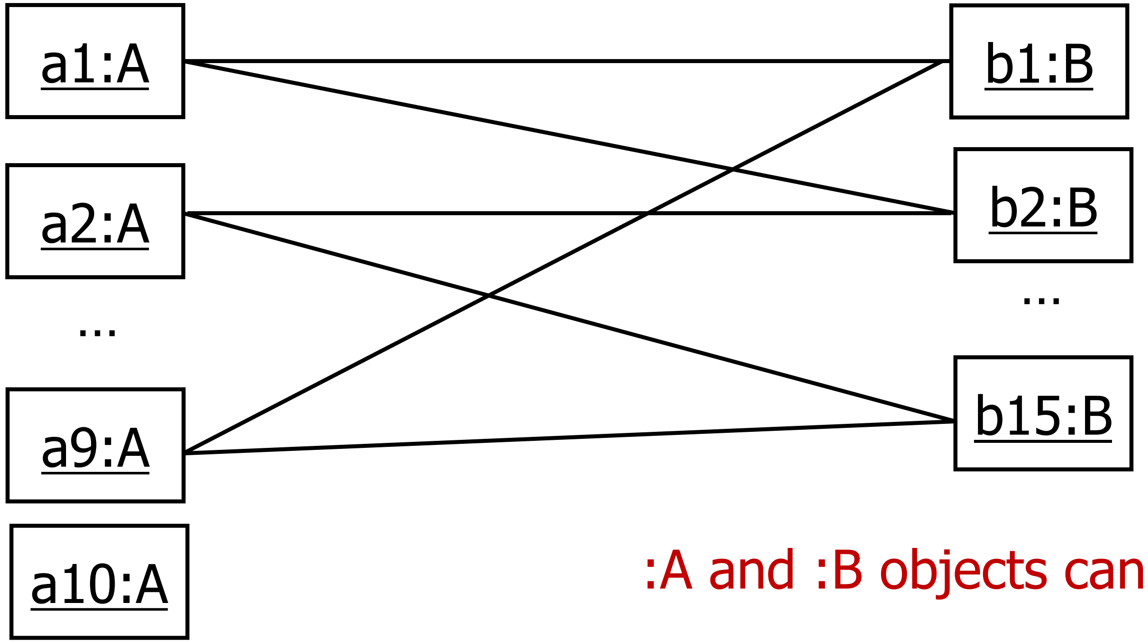


Association Relationships



many-to-many

A single A object may be linked to zero or more B objects, and vice versa.



:A and :B objects can exist on their own

Business Rules:

Draw a class diagram that represents each of the following

1. A customer must store at least one payment card.

2. A customer may store none or at most 4 payment cards.

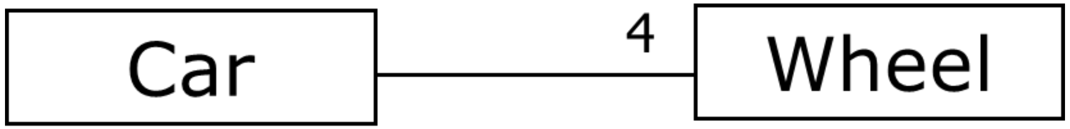
M..M or M..N

A car has exactly 4 wheels.

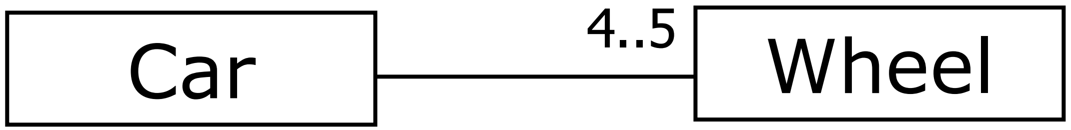
A car may have 4 or 5 wheels

M..M and M..N

A car has exactly 4 wheels.

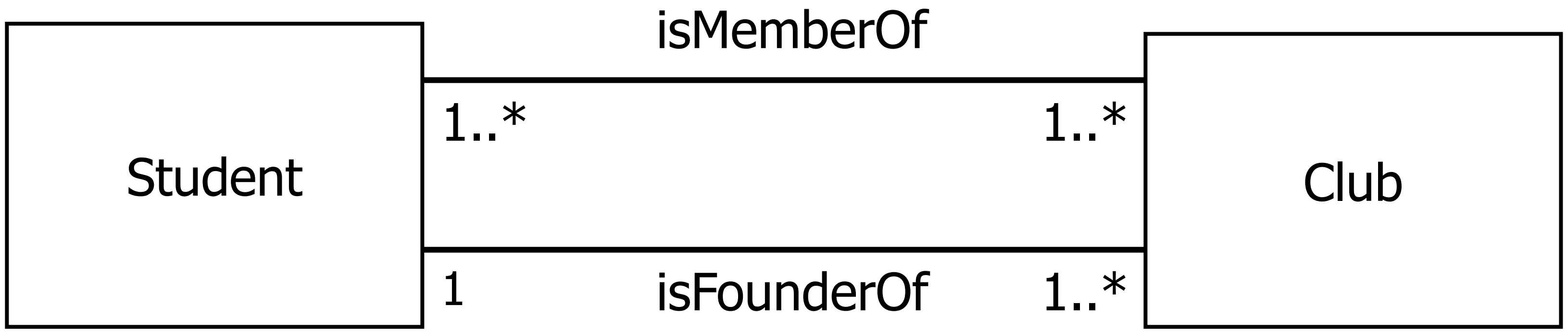


A car may have 4 or 5 wheels

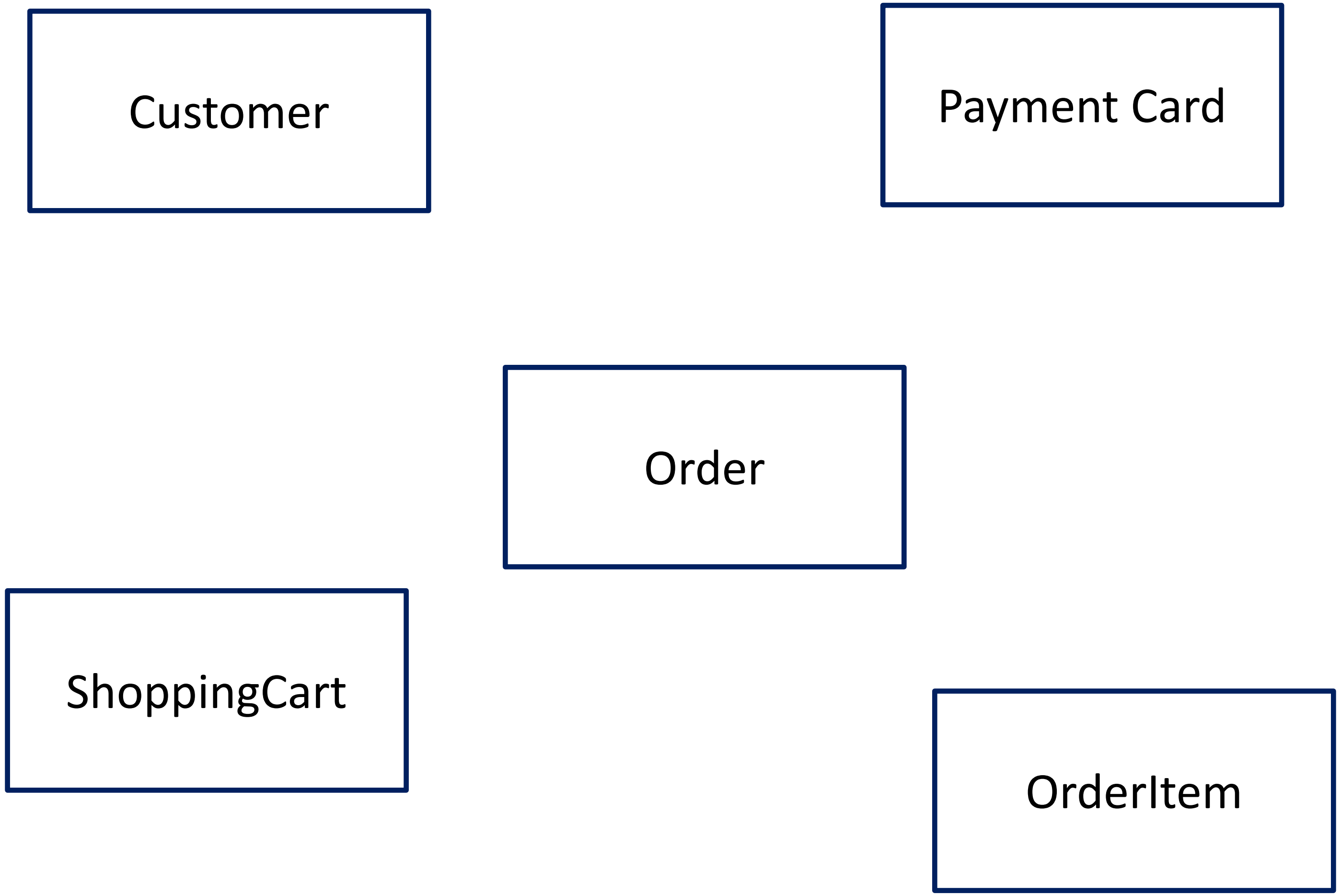


Association Relationships

Two classes may be connected by *multiple associations* as long all of them have different meanings

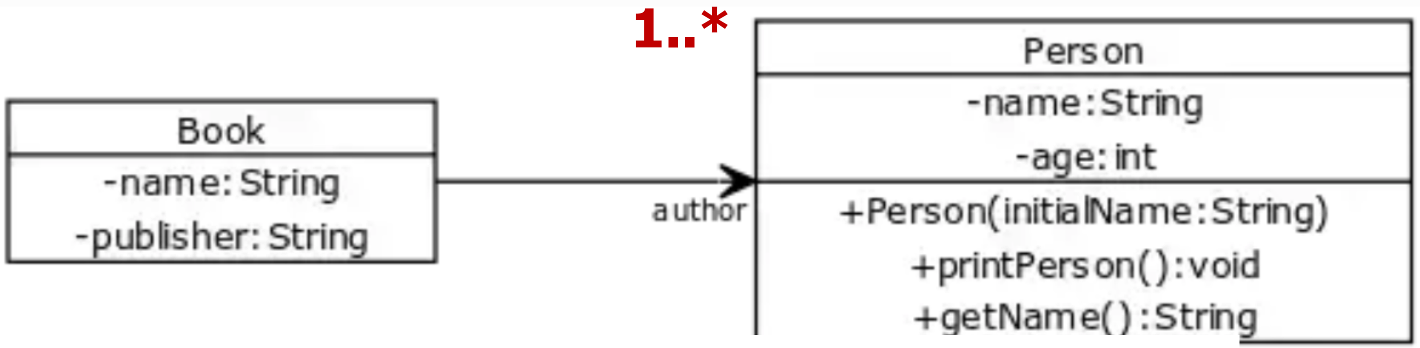


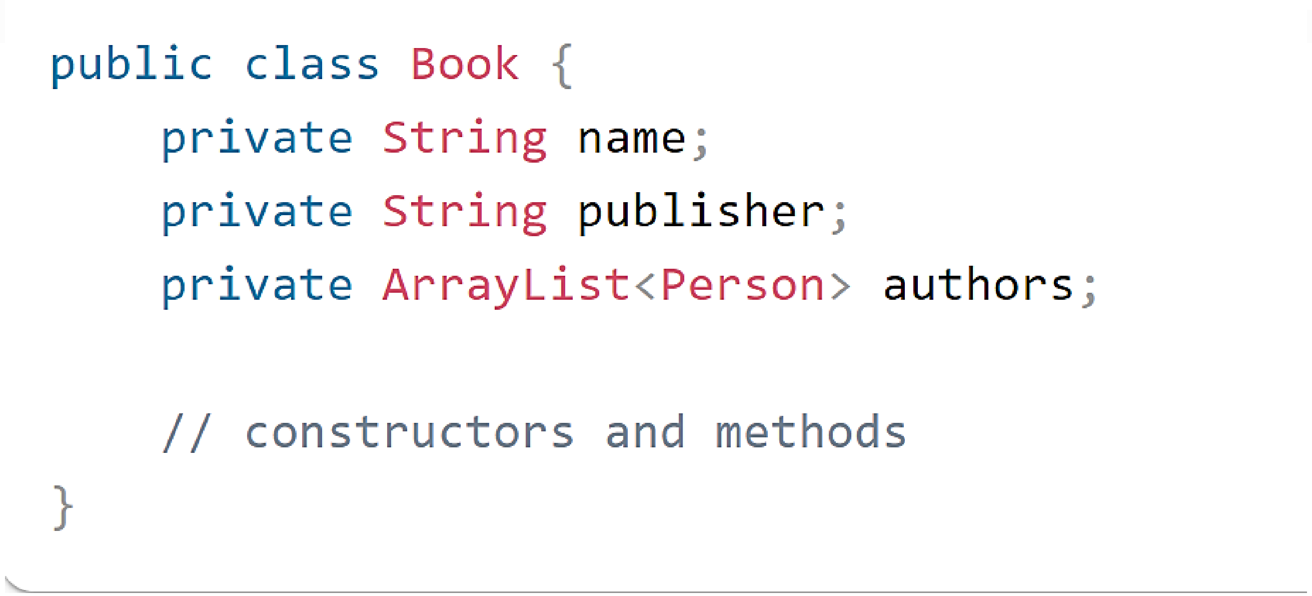
Example: Draw the multiplicities between the classes for an online shopping system.



Forward Engineering

Write Java code that implements the following class diagram





```
public class Book {
    private String name;
    private String publisher;
    private ArrayList<Person> authors;

    // constructors and methods
}
```

Reverse Engineering

- Draw a class diagram that represents the relationship between the classes.

```
public class LinearLinkedList {
private SLinkedListNode node;
private int nodeCounter;
public LinearLinkedList() {
//...
}
public void add(Object data) {
//..
}
public void delete(Object data) {
//..
}
//..
};
```

```
public class SLinkedListNode {
  private Object data;
  private SLinkedListNode next;
//
  public Object getData() {
    return data;
  }
  public void setData(Object data) {
     this.data = data;
  }
//..

};
```

Reverse Engineering

- Draw a class diagram that represents the relationship between the classes.

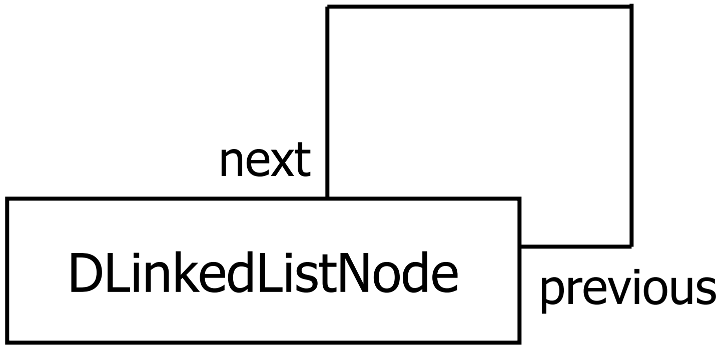
```
public class DoublyLinkedListNode {
   private DoublyLinkedListNode next;
   private DoublyLinkedListNode previous;
   private Object data;

   //constructors and methods
//..

};
```

See next slide for solution

Association Relationships

A class can have an association to itself, called a self association (or a recursive association)

Role names are usually necessary here

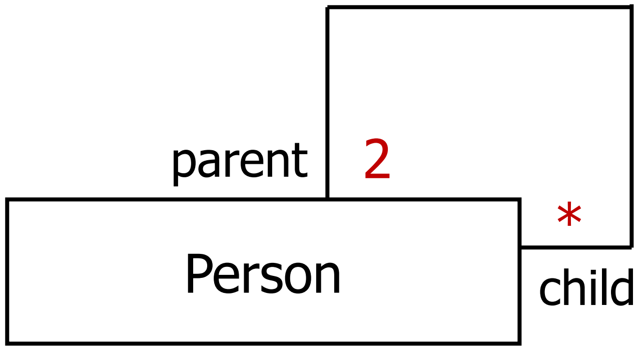


Your turn: Draw a domain class diagram

A person has exactly 2 parents.
A person can have many or no children.

Self / reflexive Association

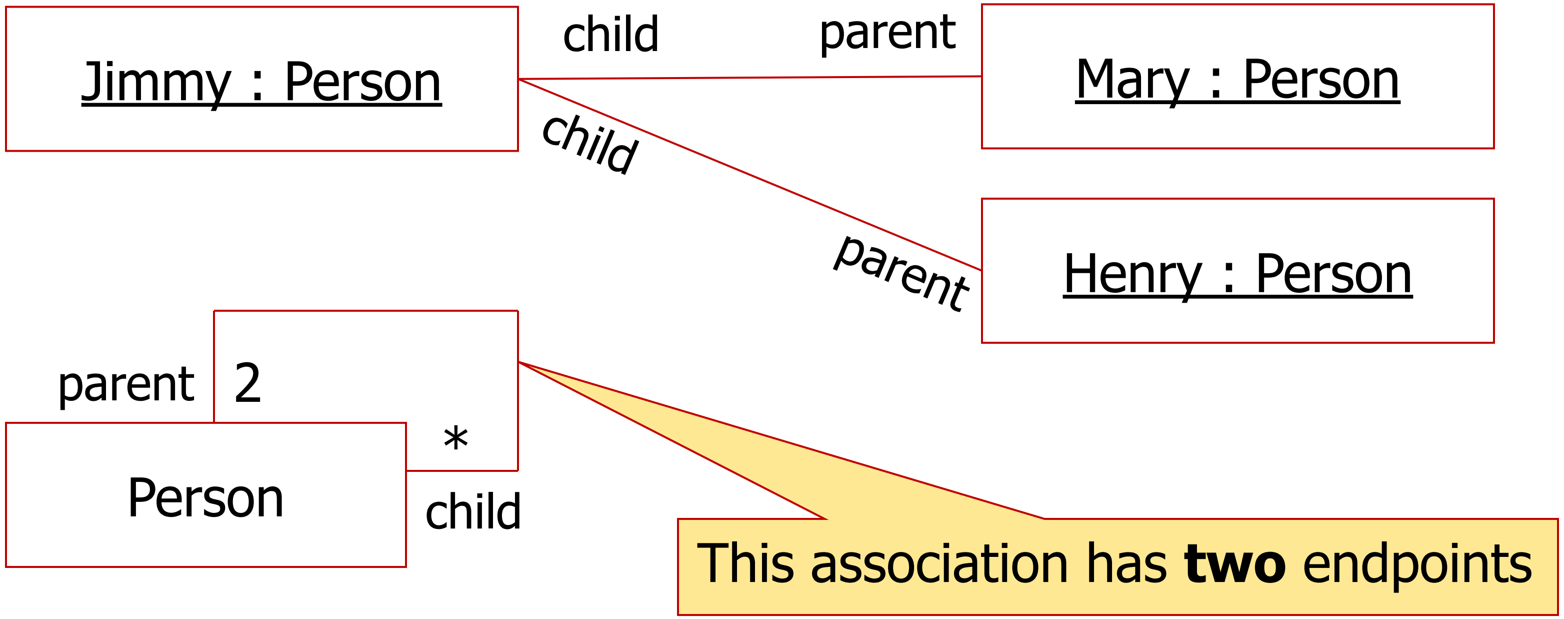
A person has 2 parents.
A person can have many or no children.



Association Relationships

Note that self associations are *binary*, even though there is only one class involved.

Recall that an association represents a collection of links connection object instances

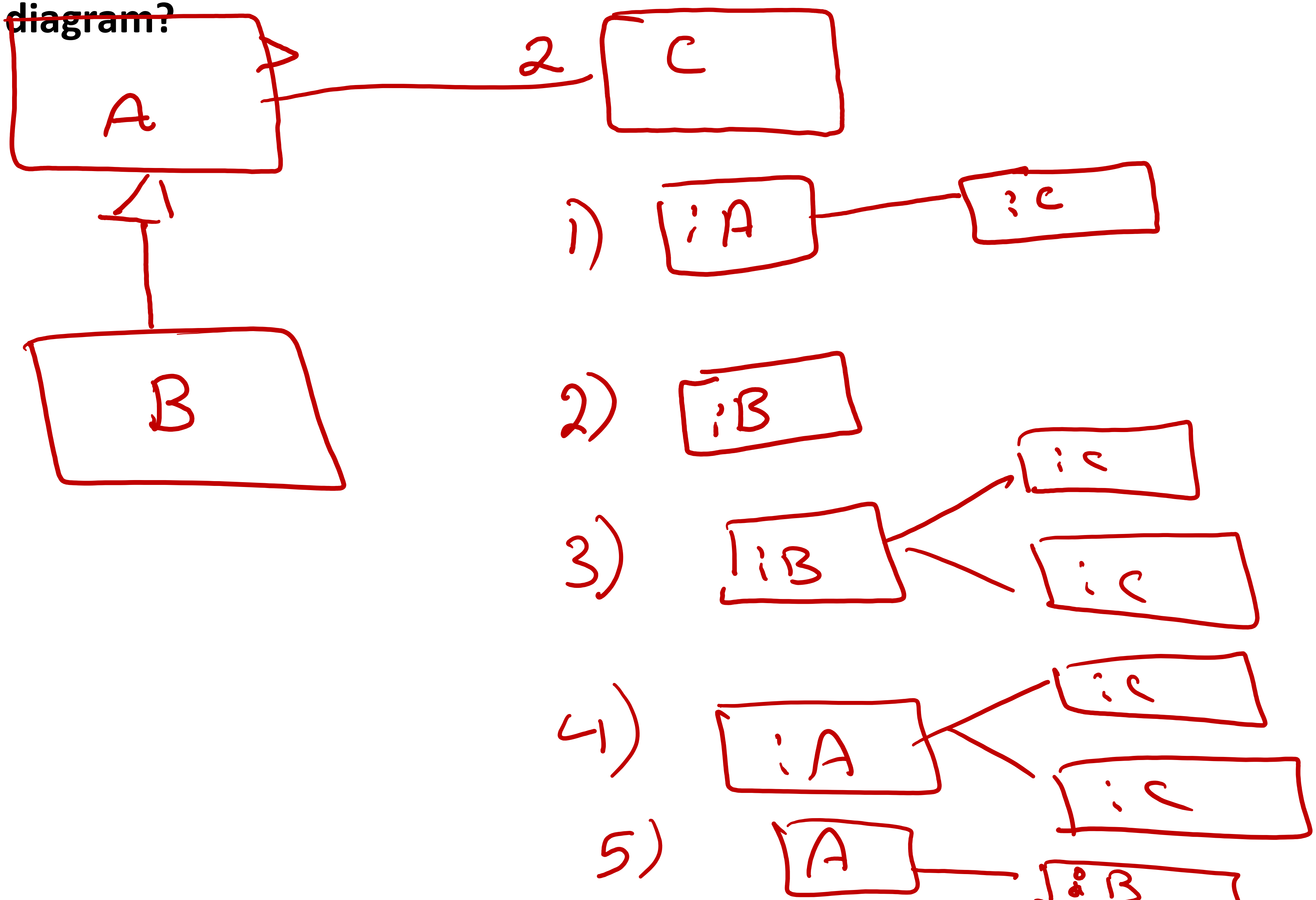


This association has **two** endpoints

```
// Parent could be an array of size 2.
public class Person {
   Private Person[] parent;
   Private Person [] child;
//constructor
Public Person(Person [] parents, Person[] child)
  {
      if ( (parent == null) ||  (parent.length != 2) )
            throw new TMException(“Invalid No. of prents"); }
     else
           this.parent = parent;
     this. Child= child;
//rest of code
```

```
public class Person {
   private Person mother;
   private Person father;
   Private Person [] children;
   // other attributes
   //possible constructor
   public Person (Person father, Person mother, Person[] kids)
     {
         if ( (father == null) || (mother == null) )
             throw new TMException(“Invalid No. of prents");
         else
         {
              this.father = father;
             this.mother = mother;
         }
         this. Children= children;
     }
   // ... // rest of your code // ... }
```

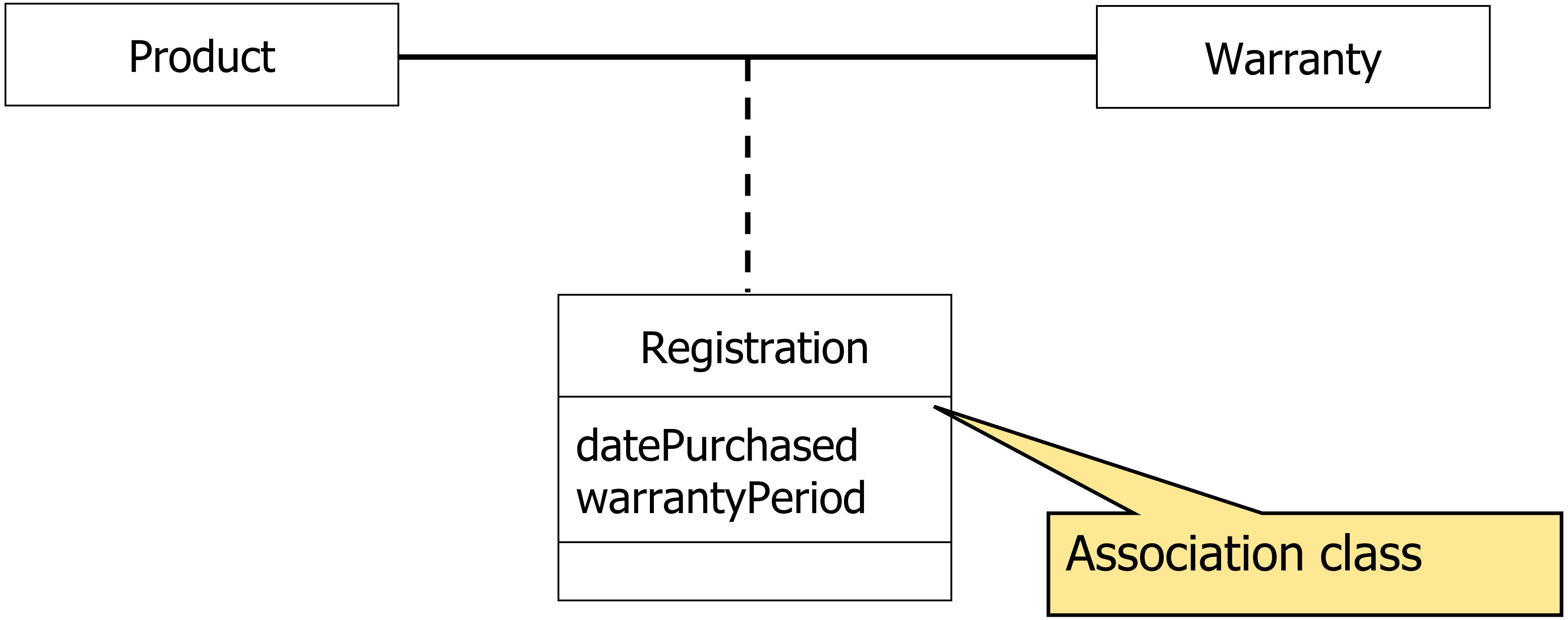
Exercise: Which of the following object diagrams are consistent with the class diagram?



Association Classes

Sometimes, it is necessary to represent properties of an association (link) itself.

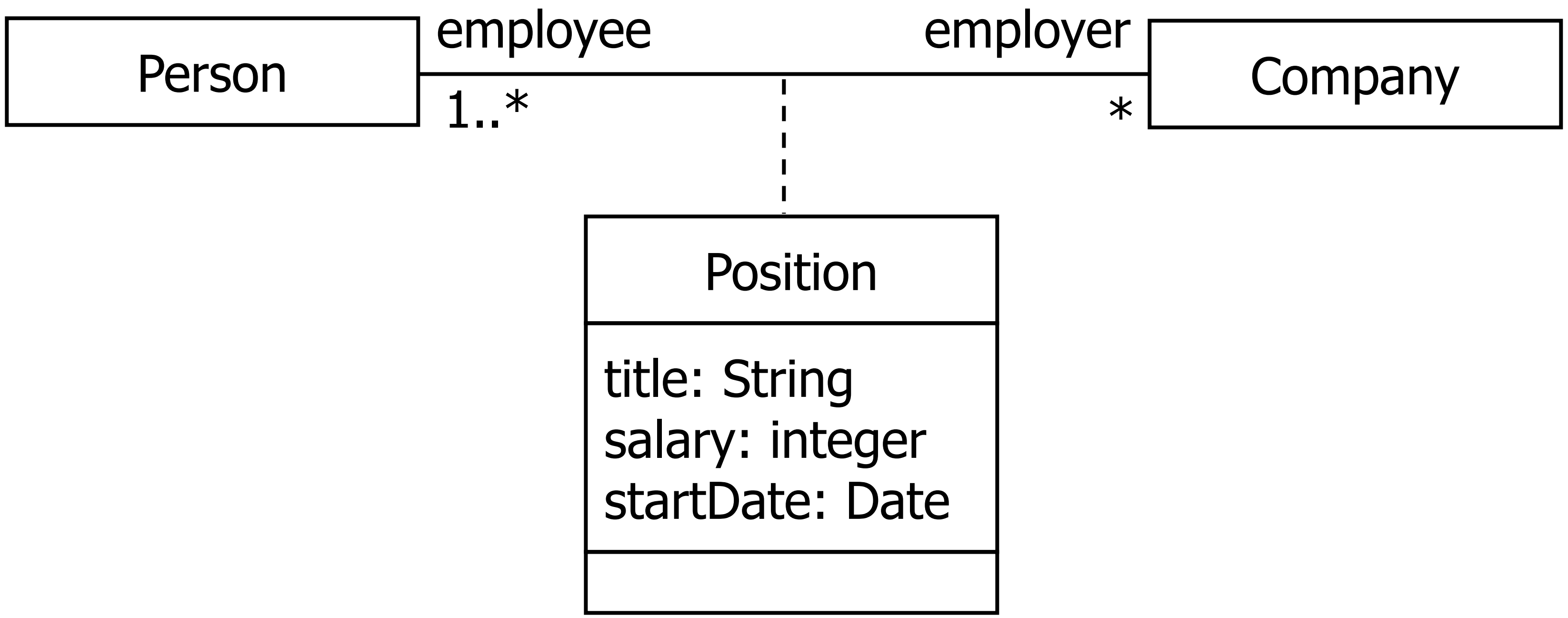
We do this using an association class.



Association Classes

The attributes title, salary, and startDate describe the employee-employer relationship between a Person and a Company

Note that they are not properties of either of the two related classes; they exist *only* if a Person has an employer

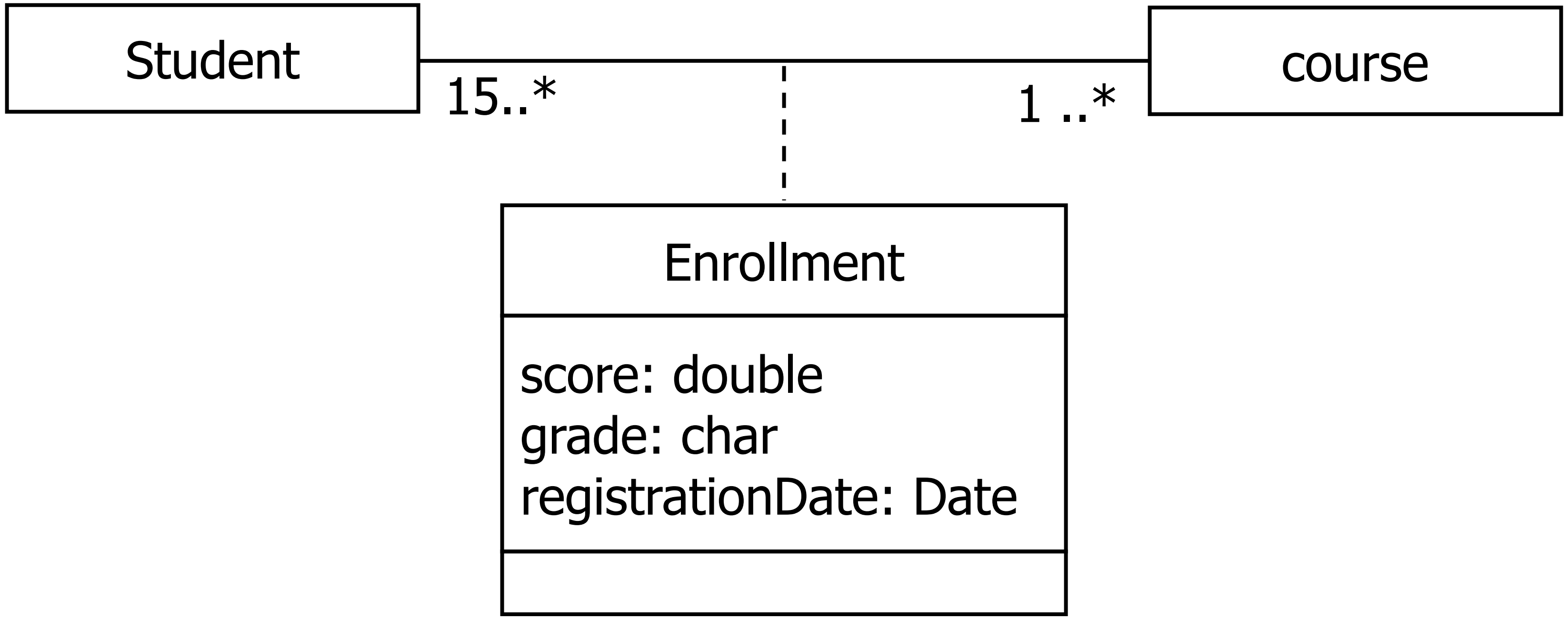


Draw a domain diagram

A student can register for one or more courses. A course can have at least 15 students.

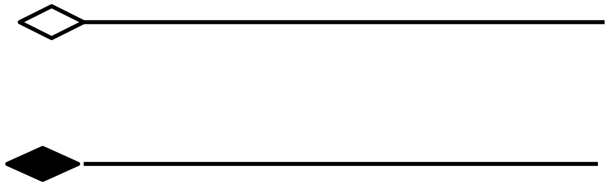
We need to store the student’s grade in each course.

Association Classes



Whole-Part relationship

- Aggregation
- Composition



These are two sub-types of Association relationships.

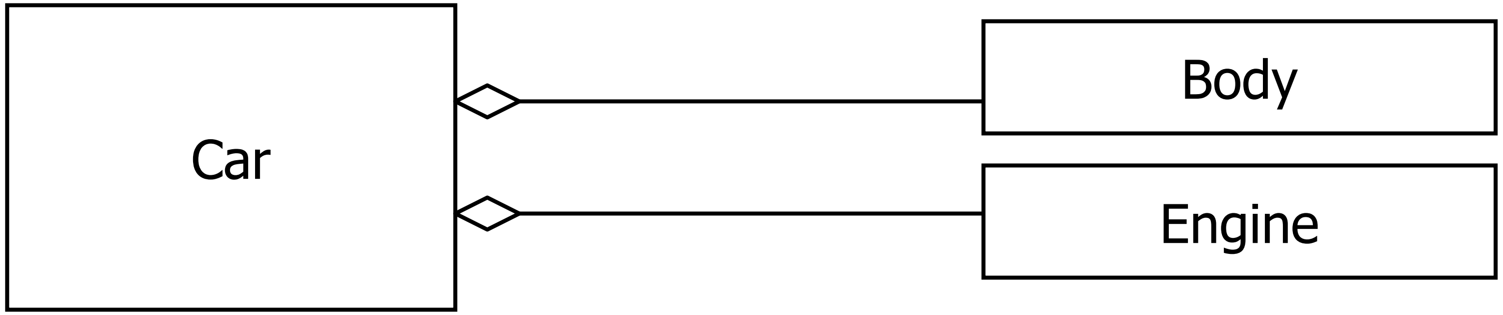
What’s the difference between these two?

- ClassA contains ClassB as an attribute, or
- Instances of ClassB are constructed inside ClassA

Aggregations

We can model objects that contain other objects by way of special associations called aggregations and compositions. Both are types of associations.

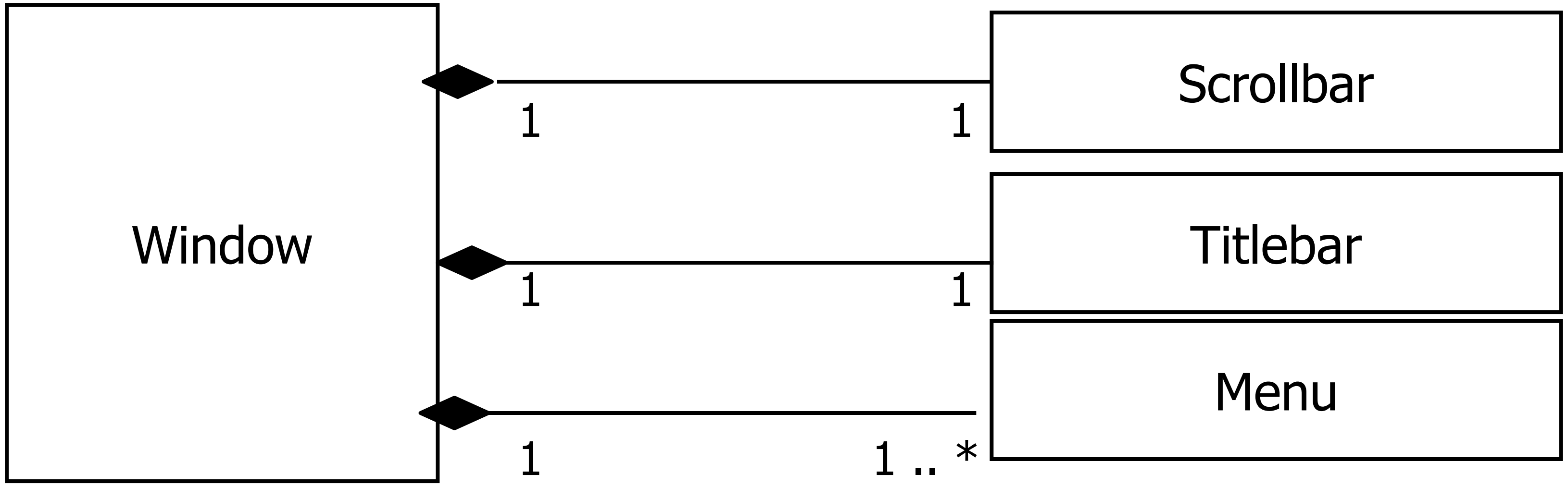
An aggregation specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.



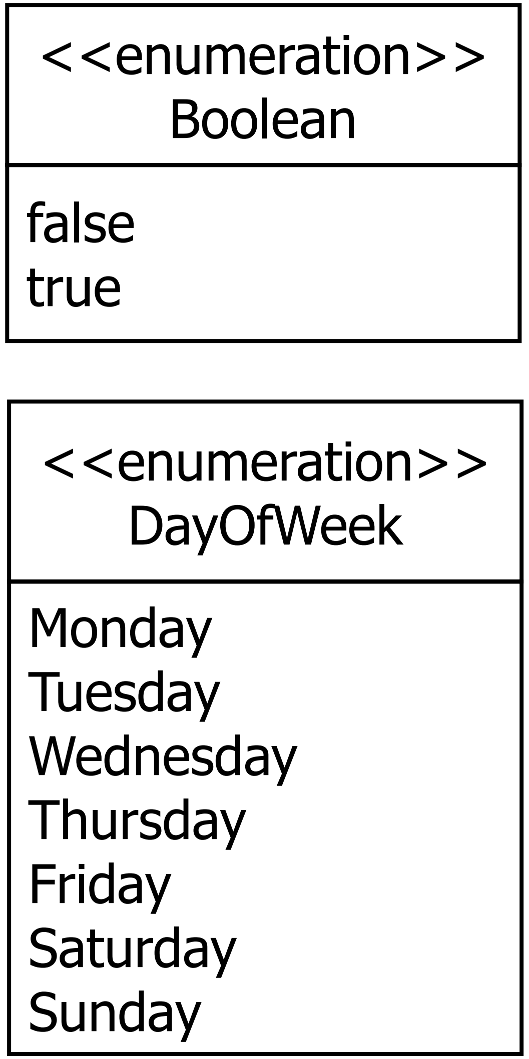
composition

A composition indicates a *strong ownership* and coincident lifetime of parts by the whole (created and destroyed at the same time).

Compositions are denoted by a filled-diamond adornment on the association.



Enumeration



<<enumeration>>
Boolean

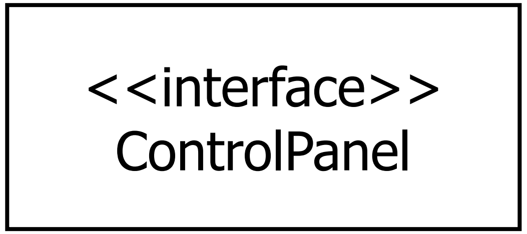
false
true

<<enumeration>>
DayOfWeek

Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday

An enumeration is a user-defined data type that consists of a name and an ordered list of enumeration literals.

Interfaces

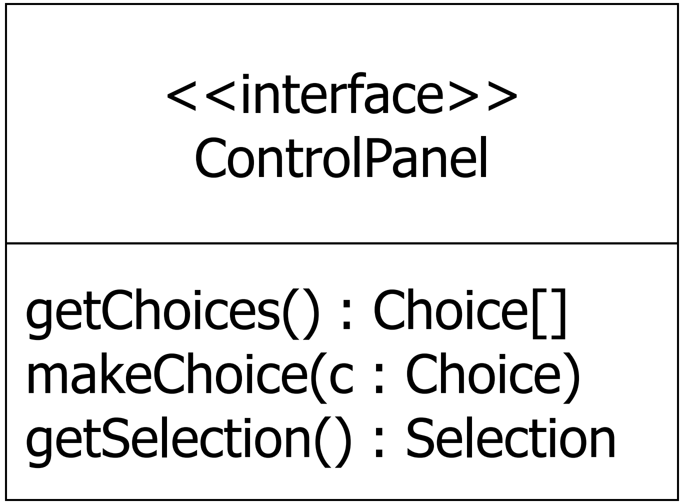


<<interface>>
ControlPanel

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 Services



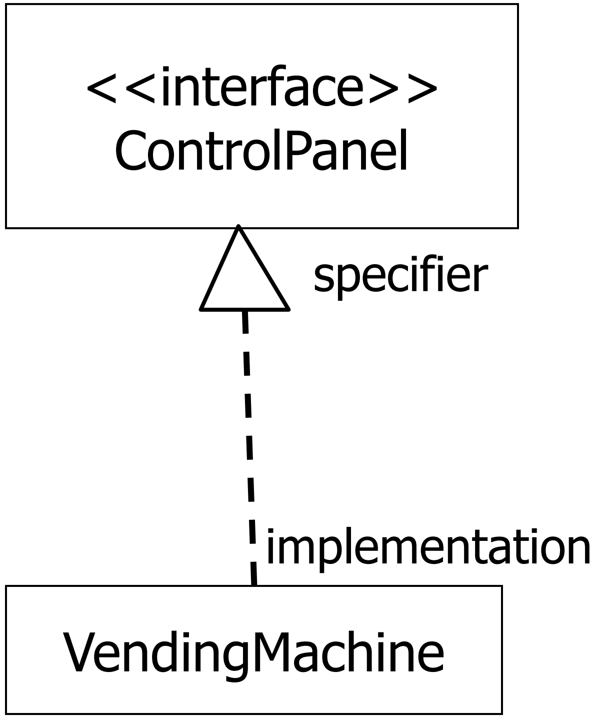
<<interface>>
ControlPanel

getChoices() : Choice[]
makeChoice(c : Choice)
getSelection() : Selection

Interfaces do not get instantiated. They have no attributes or state

Rather, they specify the services offered by a related class

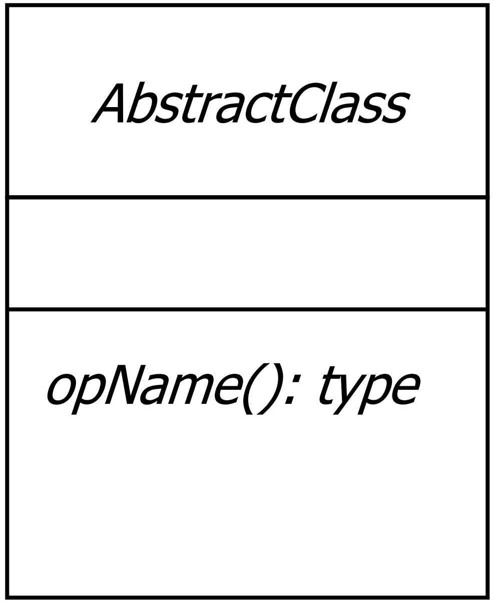
Interface Realization



A realization relationship connects a class with an interface that supplies its behavioral specification.

It is rendered by a dashed line with a hollow triangle towards the specifier.

Abstract Class

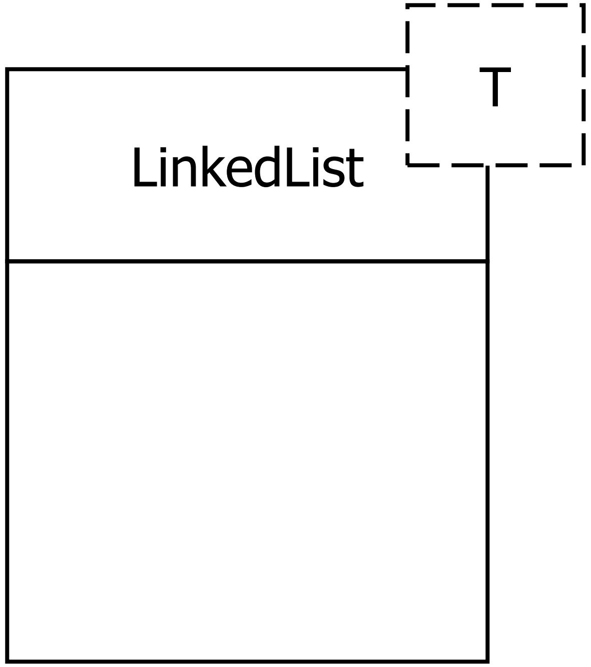


A class may be abstract

An abstract class has the name written in *italics*

Similarly, an abstract operation is represented in italics

Parameterized Class

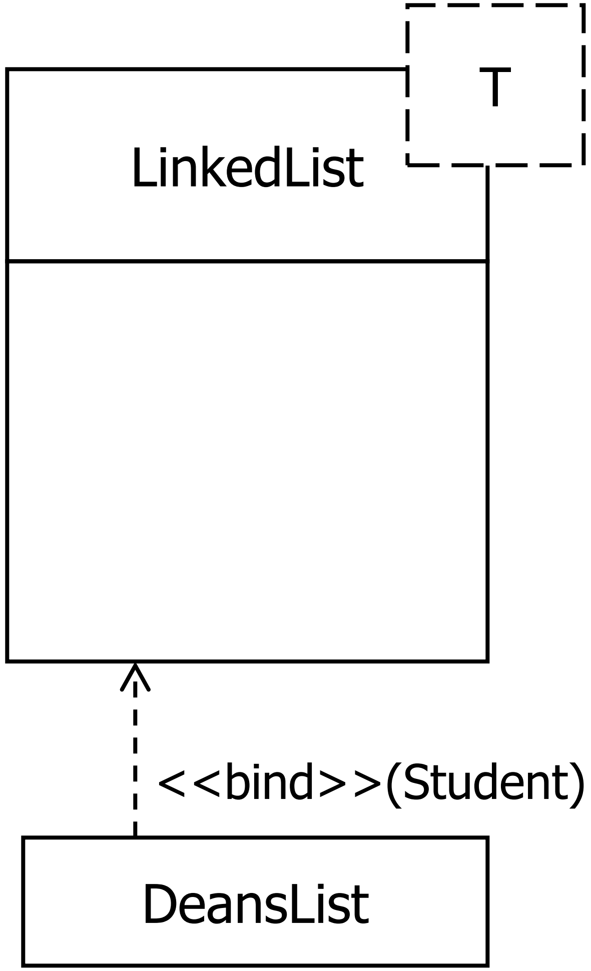


A parameterized class (or template) defines a family of potential elements

A template is represented by a small dashed rectangle in the upper-right corner of the class.

The dashed rectangle contains a list of type parameters for the class

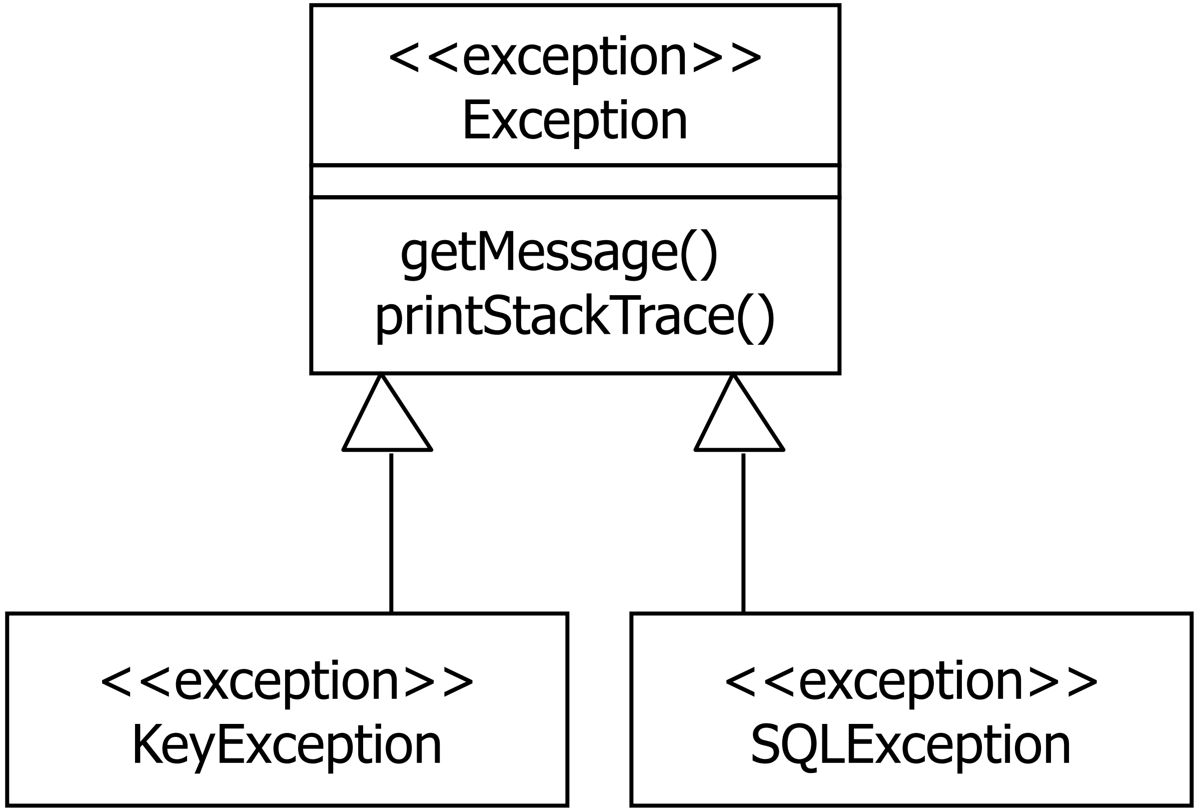
Parameterized Class



Binding is done with the <<bind>> stereotype and a parameter to supply to the template. They are displayed along the dashed arrow denoting the realization relationship.

DeansList is a LinkedList of Students (each element is a Student)

Exceptions



Exceptions can be modeled just like any other class.

Use the <<exception>> stereotype in the name compartment

More on Attributes and Operations

- Attributes and operations may have class-scope
- A class-scope attribute logically belongs to the class, not to individual instances (similar to static variables in Java)
- A class-scope operation is applied to the class, not individual instances
- Class-scope attributes and operations are underlined

More on Attributes and Operations

Attributes may have multiplicity specifications, as in the case of associations

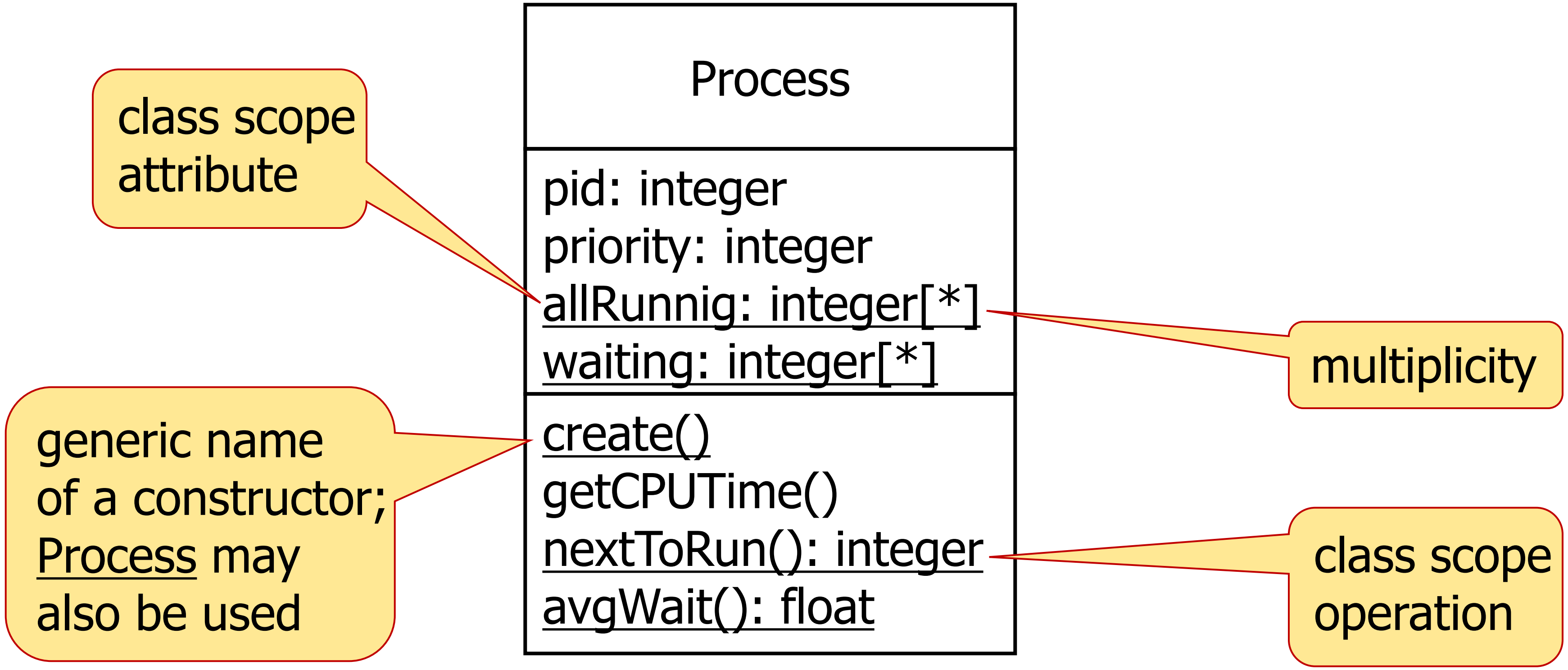
name : type [M..N]

The multiplicity specifies the cardinality of the set of values the attribute can have

For example:

vertex : node [1..*] -- one or more values
coefficient : float [1..100] -- up to a 100 floats
state : sink [0..1] -- one value or null

More on Attributes and Operations



class scope attribute

Process

pid: integer
priority: integer
allRunnig: integer[*]
waiting: integer[*]

create()
getCPUTime()
nextToRun(): integer
avgWait(): float

multiplicity

generic name of a constructor; Process may also be used

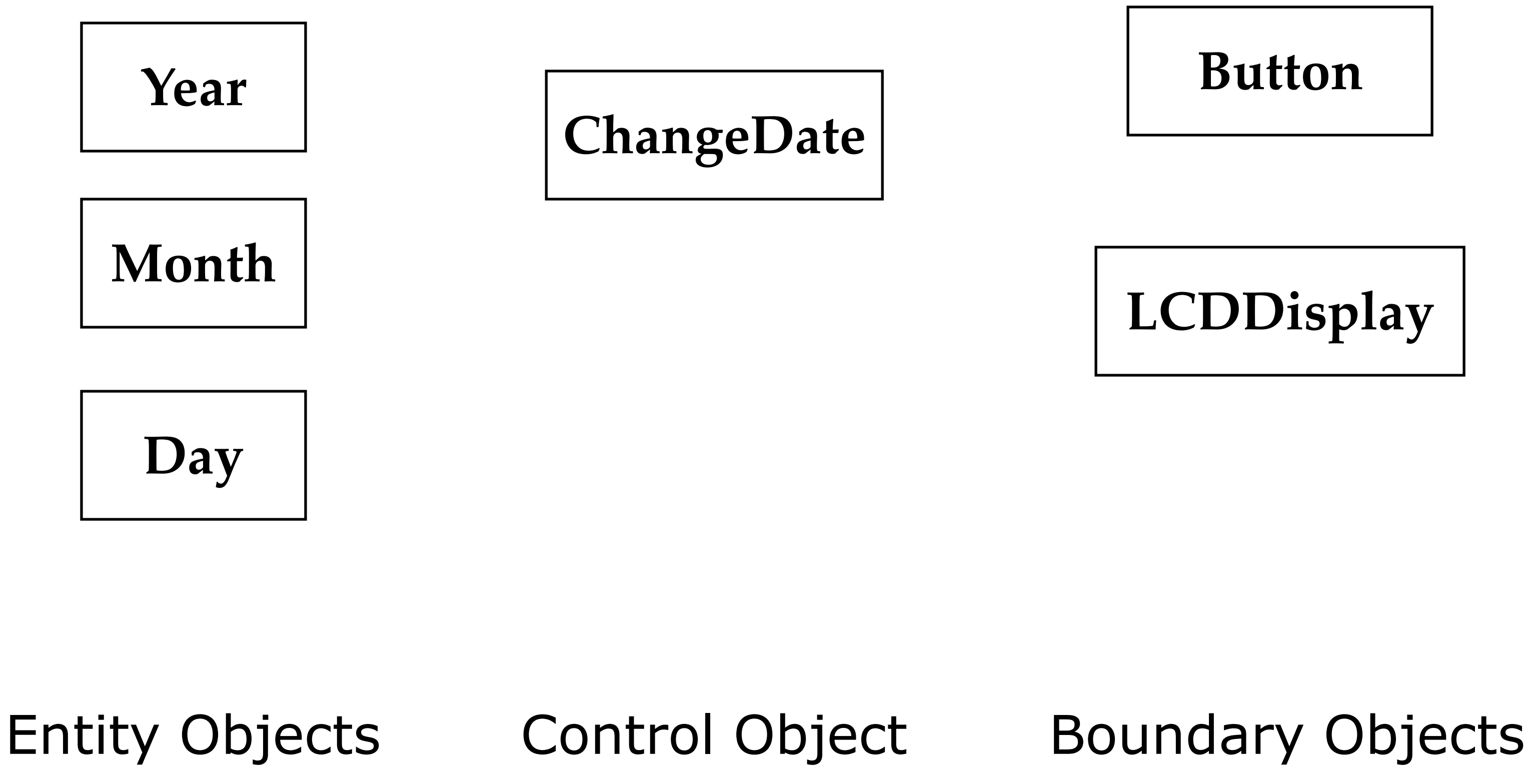
class scope operation

Class-scope attributes and operations are frequently used for aggregate-type values

There are different types of Objects

- Entity Objects
  - Represent the persistent information tracked by the system (Application domain objects, also called "Business objects")
- Boundary Objects
  - Represent the interaction between the user and the system
- Control Objects
  - Represent the control tasks performed by the system.

Example: 2B Watch Modeling



Year

Month

Day

ChangeDate

Button

LCDDisplay

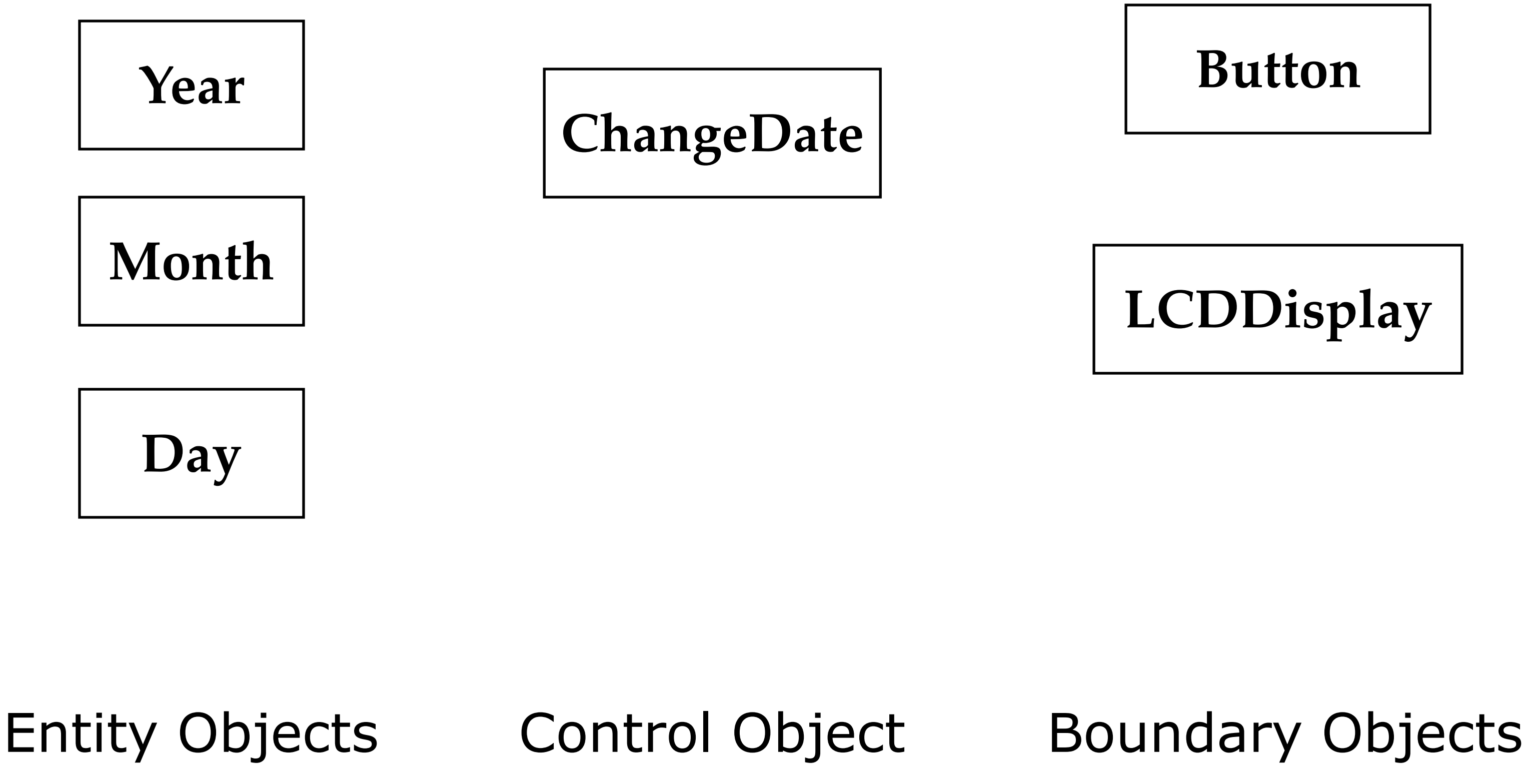
Entity Objects

Control Object

Boundary Objects

Example: 2B Watch Modeling

To distinguish different object types
in a model we can use the
UML Stereotype mechanism



Year

Month

Day

ChangeDate

Button

LCDDisplay

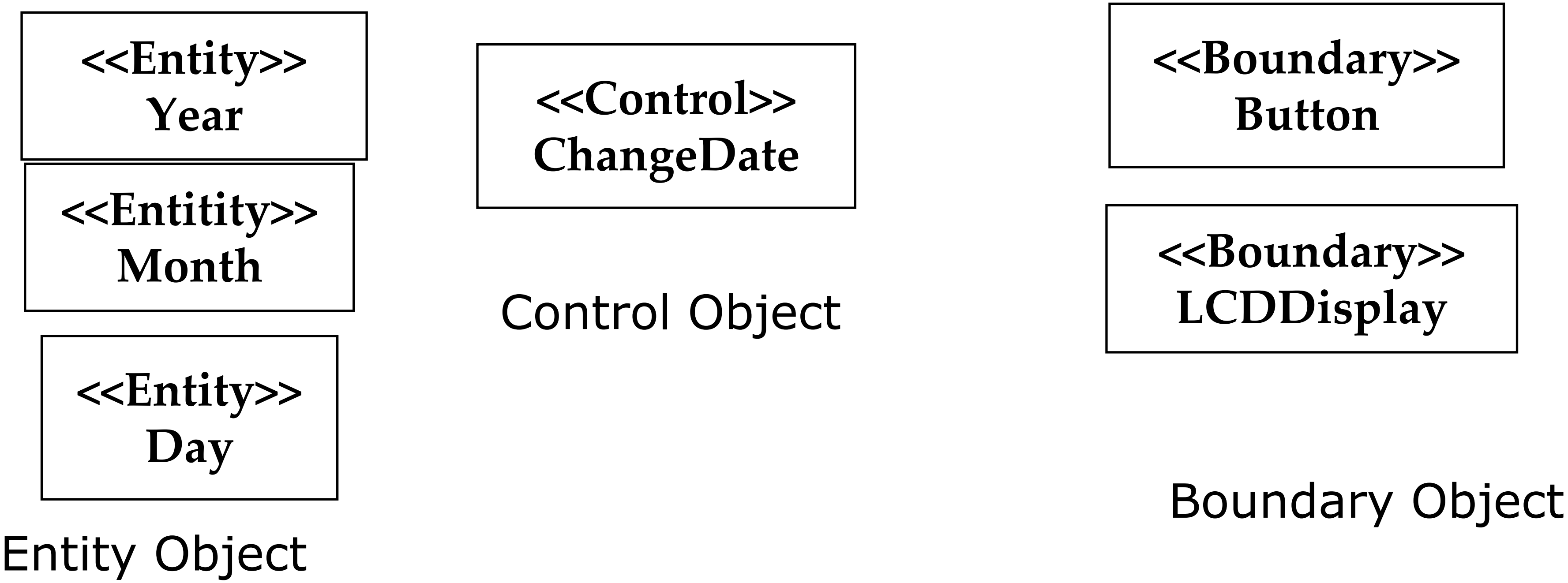
Entity Objects

Control Object

Boundary Objects

Naming Object Types in UML

- UML provides the stereotype mechanism to introduce new types of modeling elements
  - A stereotype is drawn as a name enclosed by angled double-quotes ("guillemets") (<<, >>) and placed before the name of a UML element (class, method, attribute, ….)
  - Notation: <<String>>Name



<<Entity>> Year

<<Entitity>> Month

<<Entity>> Day

Entity Object

<<Control>> ChangeDate

Control Object

<<Boundary>> Button

<<Boundary>> LCDDisplay

Boundary Object

Finding Participating Objects in Use Cases

- Pick a use case and look at flow of events
- Do a textual analysis (noun-verb analysis)
  - Nouns are candidates for objects/classes
  - Verbs are candidates for operations
  - This is also called Abbott’s Technique

- After objects/classes are found, identify their types
  - Identify real world entities that the system needs to keep track of (FieldOfficer □ Entity Object)
  - Identify real world procedures that the system needs to keep track of (EmergencyPlan □ Control Object)
  - Identify interface artifacts (PoliceStation □ Boundary Object).

Example for using the Technique

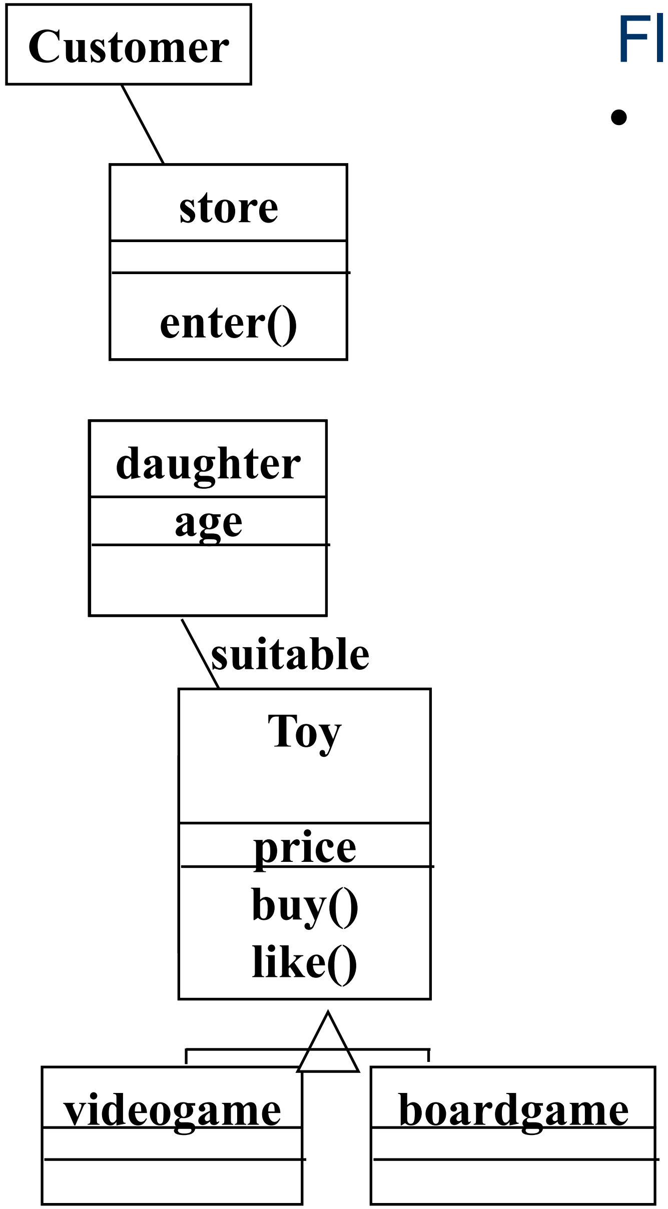
Flow of Events:

- The customer enters the store to buy a toy.
- It has to be a toy that his daughter likes and it must cost less than 50 Euro.
- He tries a videogame, which uses a data glove and a head-mounted display. He likes it.
- An assistant helps him.
- The suitability of the game depends on the age of the child.
- His daughter is only 3 years old.
- The assistant recommends another type of toy, namely the boardgame “Monopoly".

Mapping parts of speech to model components (Abbot's Technique)

Example	*Part of speech*	*UML model component*
"Monopoly"	Proper noun	object
Toy	Improper noun	class
Buy, recommend	Doing verb	operation
is-a	being verb	inheritance
has an	having verb	aggregation
must be	modal verb	constraint
dangerous	adjective	attribute
enter	transitive verb	operation
depends on	intransitive verb	Constraint, class, association

Generating a Class Diagram from Flow of Events



Flow of events:

- The customer enters the store to buy a toy. It has to be a toy that his daughter likes and it must cost less than 50 Euro. He tries a videogame, which uses a data glove and a head-mounted display. He likes it.

  An assistant helps him. The suitability of the game depends on the age of the child. His daughter is only 3 years old. The assistant recommends another type of toy, namely a boardgame. The customer buy the game and leaves the store

Ways to find Objects

- Syntactical investigation with Abbot‘s technique:
  - Flow of events in use cases
  - Problem statement
- Use other knowledge sources:
  - Application knowledge: End users and experts know the abstractions of the application domain
  - Solution knowledge: Abstractions in the solution domain
  - General world knowledge: Your generic knowledge and intution

Order of Activities for Object Identification

1. Formulate a few scenarios with help from an end user or application domain expert
2. Extract the use cases from the scenarios, with the help of an application domain expert
3. Then proceed in parallel with the following:
   - Analyse the flow of events in each use case using Abbot's textual analysis technique
   - Generate the UML class diagram.

Steps in Generating Class Diagrams

1. Class identification (textual analysis, domain expert)
2. Identification of attributes and operations (sometimes before the classes are found!)
3. Identification of associations between classes
4. Identification of multiplicities
5. Identification of roles
6. Identification of inheritance

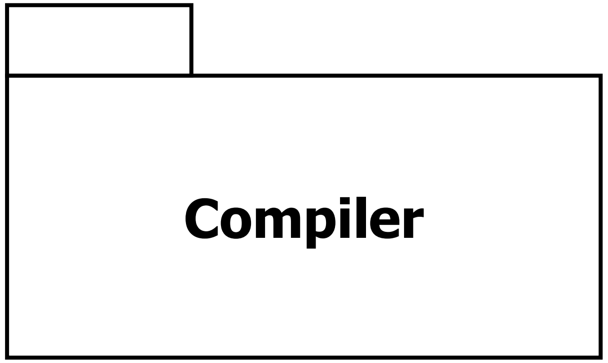
More on Generalizations

Generalizations can be:

- complete – all subclasses have been specified, i.e. no additional subclasses may be created

  (similar to a *final* class in Java)
- incomplete – subclasses may be created in the future
  - **this is the default**

Packages



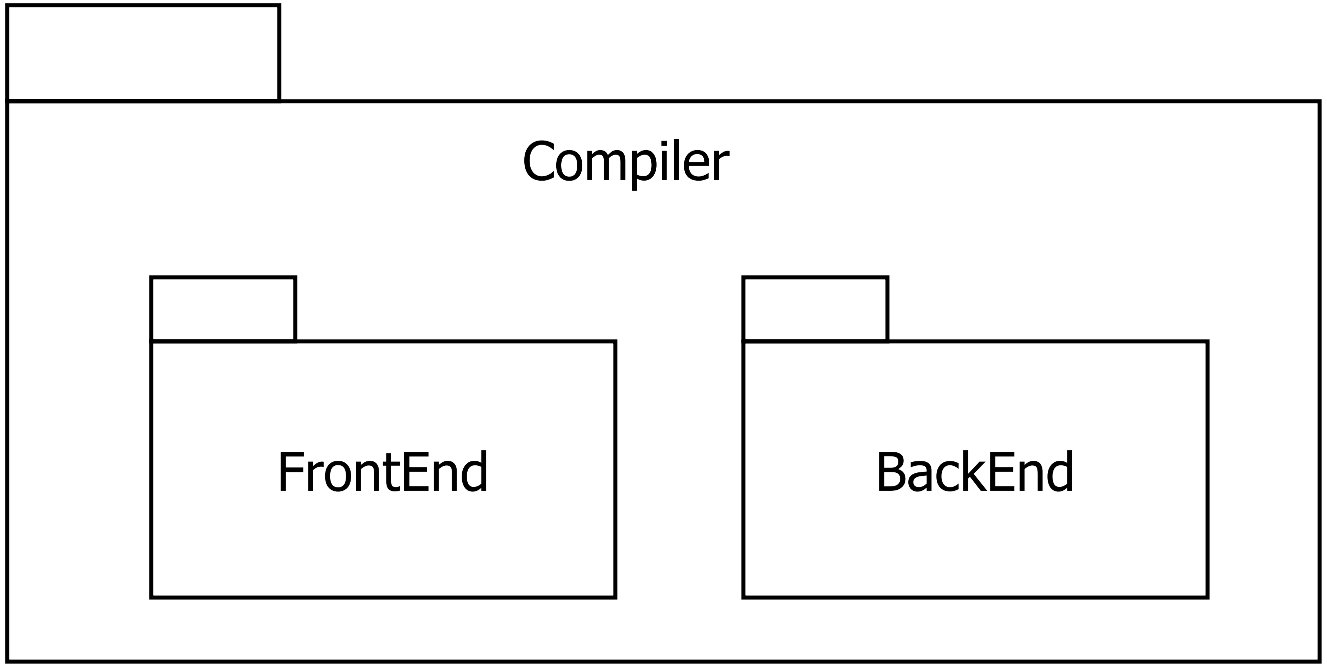
A package is a container-like element for organizing other elements into groups

A package can contain classes, other packages and diagrams

Packages can be used to provide controlled access between classes in different packages

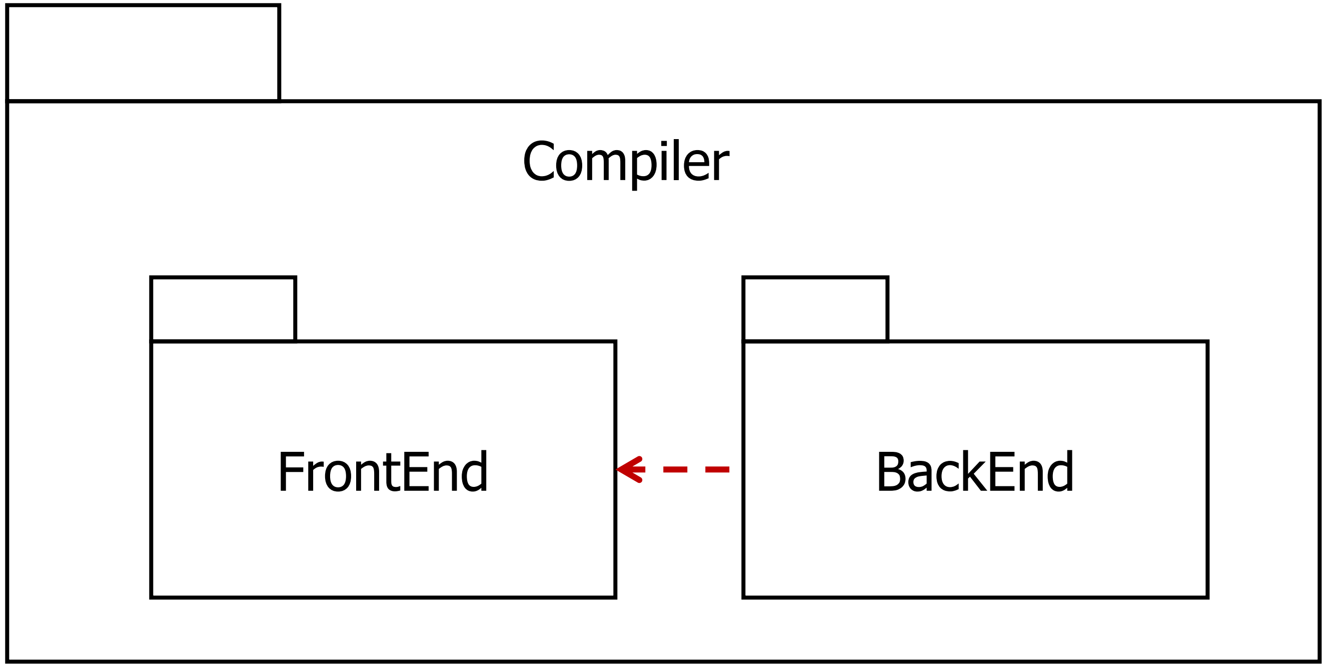
Packages

Classes in the *FrontEnd* package and classes in the *BackEnd* package cannot access each other in this diagram.

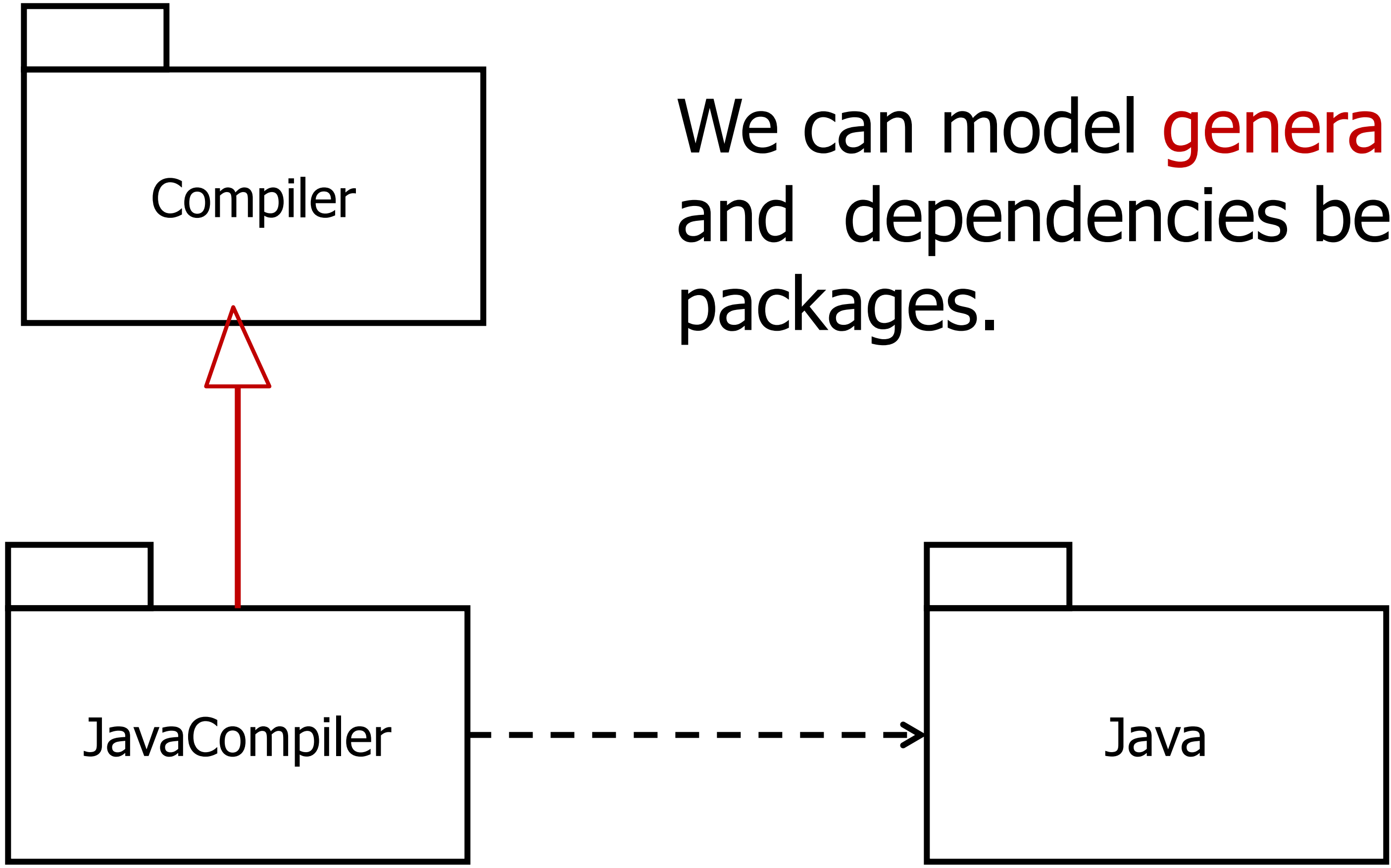


Packages

Classes in the *BackEnd* package now have access to the classes in the *FrontEnd* package.



Packages

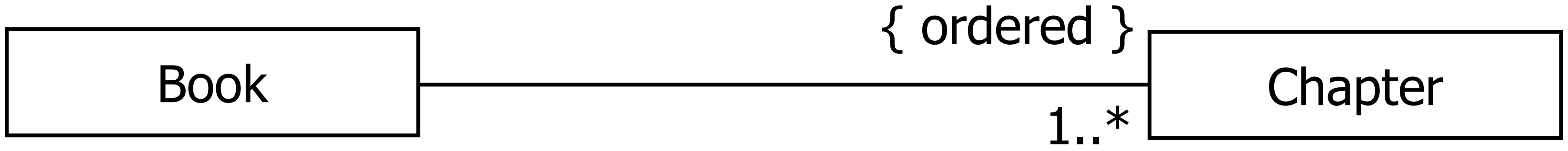


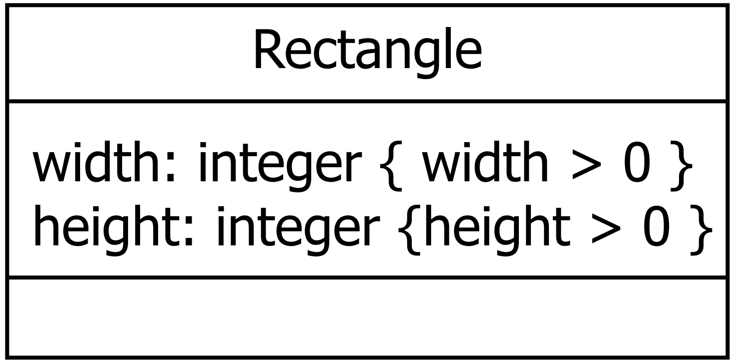
We can model generalizations and dependencies between packages.

Constraints

A class diagram may include constraints

Constraints are enclosed within the curly braces { … }





Constraints

Other types of typical constraints include:

- unordered,
- unique,
- nonunique

More complicated constraints can be expressed with the use of the Object Constraint Language (OCL).

We will study OCL later…

The Rest …

We will revisit class diagrams during the Object Design phase and add a few more modeling constructs