# Object Oriented Analysis and Design with Java

**Prof. J Ruby Dinakar**
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

# Object Oriented Analysis and Design with Java

## Class Modelling: OO-Relationships

**Prof. J Ruby Dinakar**

Department of Computer Science and Engineering

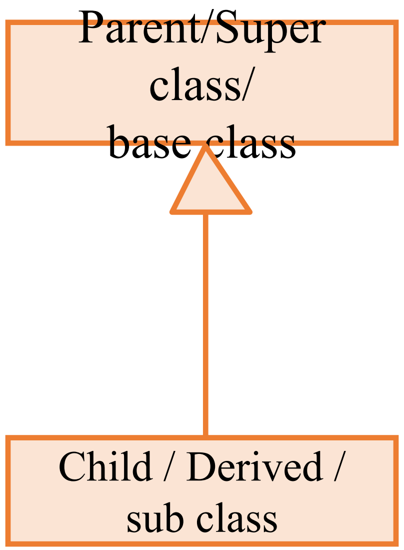
## OO-Relationships

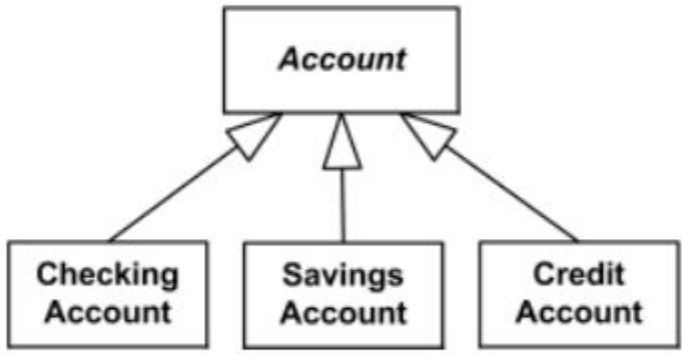
In UML, object interconnections (logical or physical), are modeled as relationships. The type of relationships are listed below:

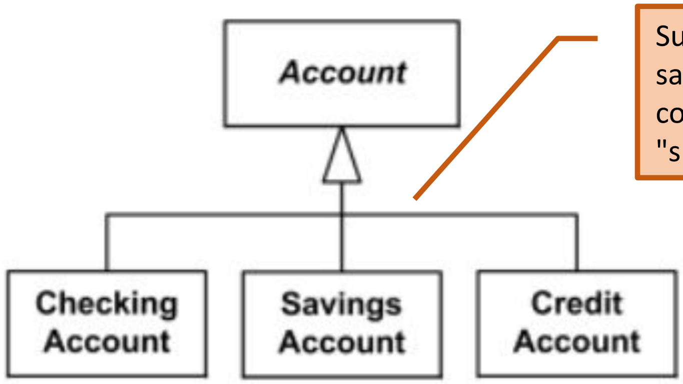
- Generalization- an inheritance  relationship
- Abstraction
- Realization - interface implementation
- Dependency
- Association – using relationship
    - Aggregation
    - Composition

## Generalization

- Also known as Inheritance.
- It represents "is a" or "is a kind of" relationship since the child class is a type of the parent class.
- It is used to showcase reusable elements in the class diagram.
- The child classes "inherit" the common functionality(attributes and methods) defined in the parent class.
- A generalization is shown as a line with a hollow triangle as an arrowhead.
- The arrowhead points to the super class



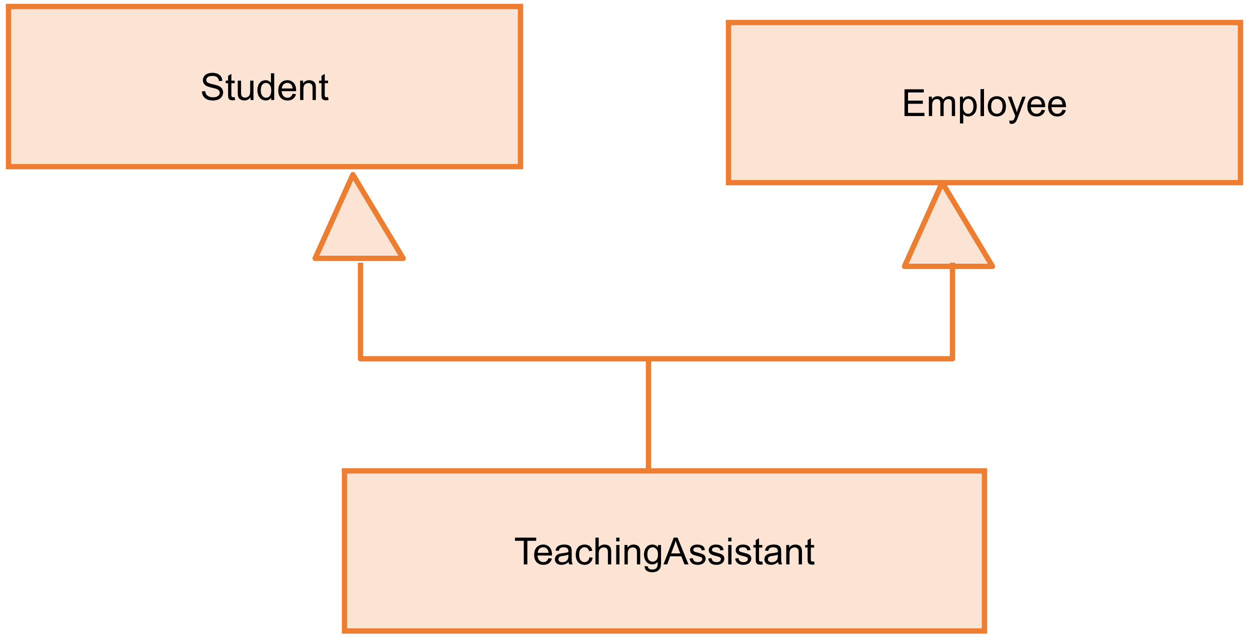




Sub classes that reference the same base class can also be connected together in the "shared target style."

## Generalization Relationships (Cont’d)

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

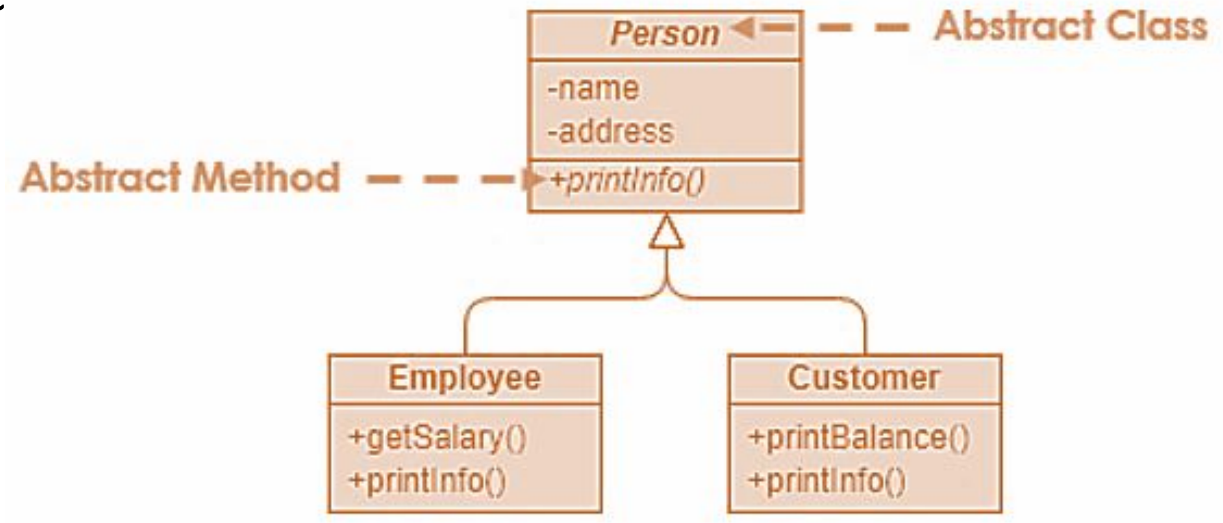


## Abstraction

In an inheritance hierarchy, subclasses implement specific details, whereas the parent class defines the framework its subclasses. The parent class also serves as a template for common methods that will be implemented by its subclasses.

The name of an abstract Class is typically shown in italics; alternatively, an abstract Class may be shown using the textual annotation, also called stereotype {abstract} after or below its name.

An abstract method is a method that do not have implementation. In order to create an abstract method, create a operation and make it italic

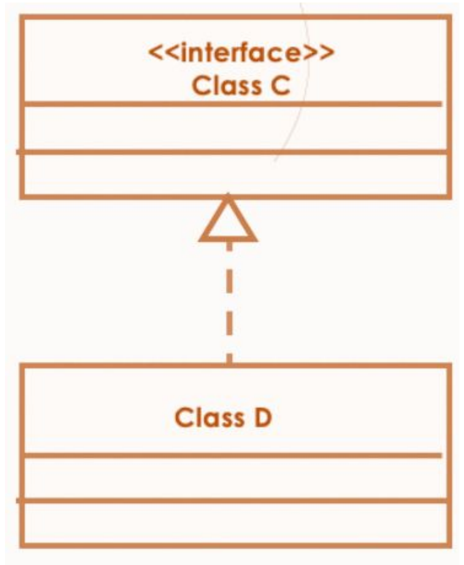


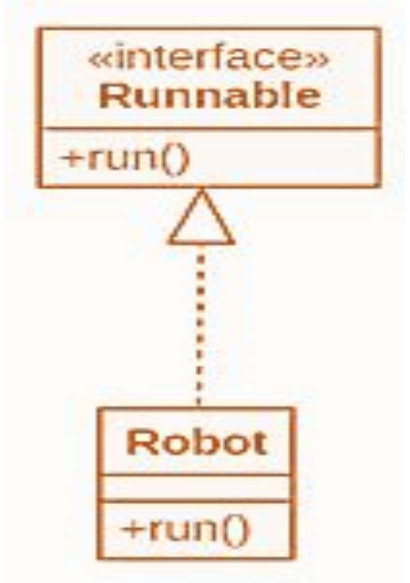
## Realization / Interfaces

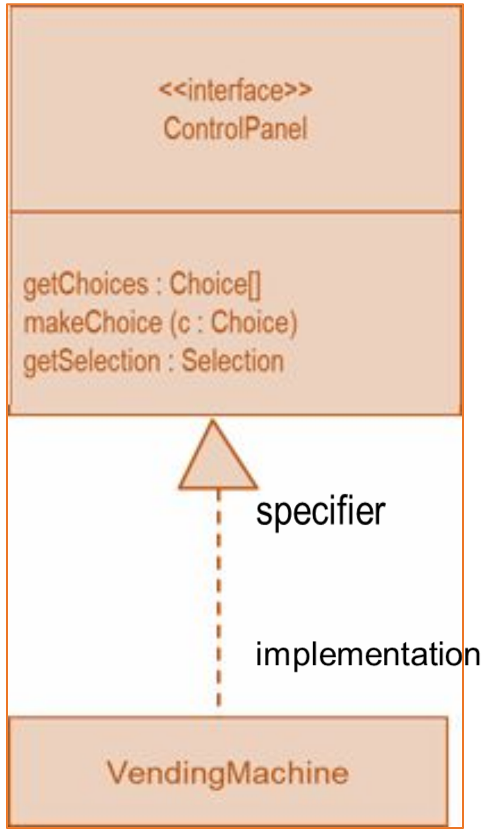
Realization is a specialized abstraction relationship between two things where one thing (an interface) specifies a contract that another thing (a class) guarantees to carry out by implementing the operations specified in that contract.

Interface defines a set of functionalities as a contract and the other entity "realizes" the contract by implementing the functionality defined in the contract.

Realization is shown as a dashed directed line with an open arrowhead pointing to the interface.

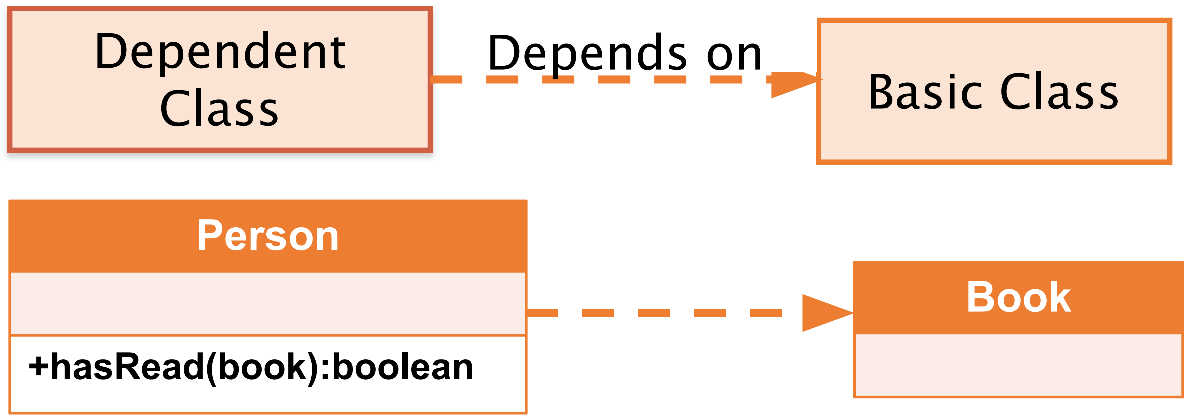






## Dependency

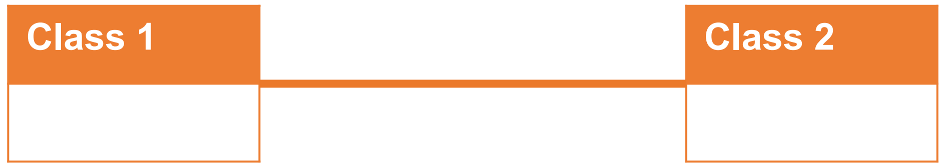
- Dependency means “**One class uses the other**”
- A dependency relationship indicates that a **change** in **one class** may **affect** the **dependent class**, but not necessarily the reverse.
- A dependency relationship is often used to show that a **method has object** of a class **as arguments**.

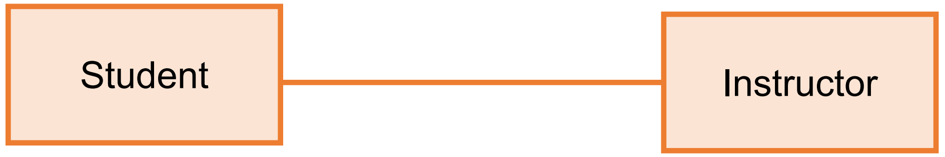


An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship. For example, the Person class might have a hasRead method with Book as a parameter that returns true if the person has read the book.

## 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.



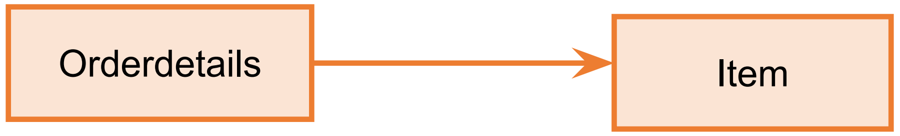


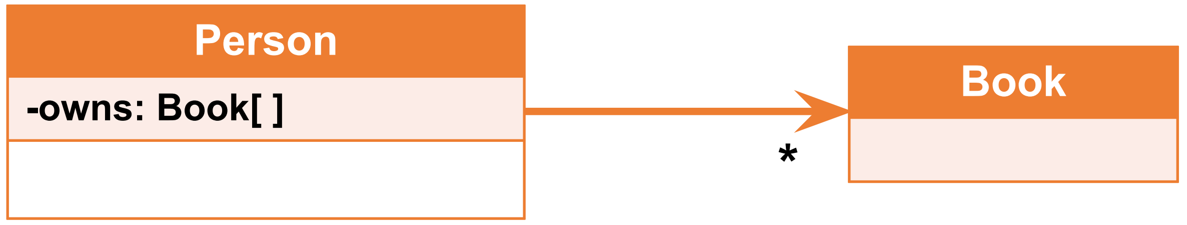
We can constrain the association relationship by defining the *navigability* of the association.
Association is of two types.
They are
- Unidirectional Association
- Bidirectional Association

## Object Oriented Analysis and Design with Java

## Association Relationships (Cont'd)

- In a uni-directional association, two classes are related, but only one class "knows" that the relationship exists.
- A uni-directional association is drawn as a solid line with an open arrowhead pointing to the known class.
- Uni-directional association includes a role name and a multiplicity value, but only for the known class.
- In this example Orderdetails class only knows that it has a relationship with Item class but Item class does not know about their relationship.





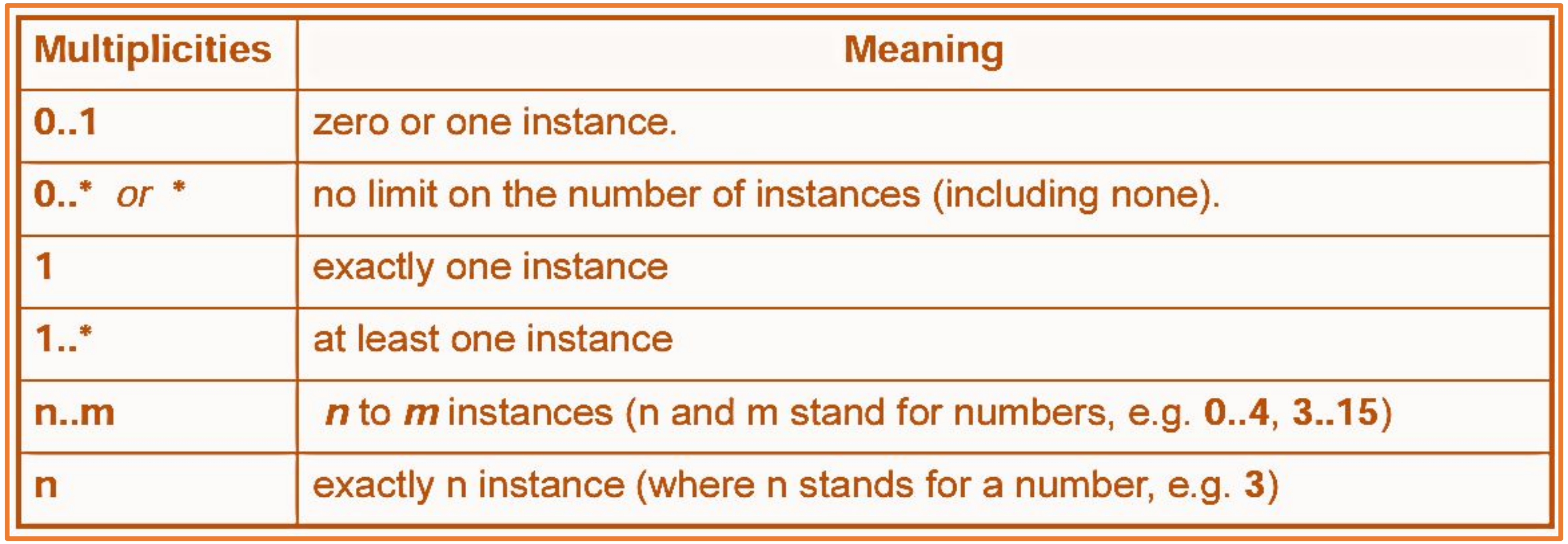
An object might store another object in a field. For example, people own books, which might be modeled by an owns field in Person objects. However, a book might be owned by a very large number of people, so the reverse direction might not be modeled. The * in the figure indicate that a person can own any number of books.

## Association Relationships (Cont'd)

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

The multiplicity of an association is the number of possible instances of the class associated with a single instance of the other end.

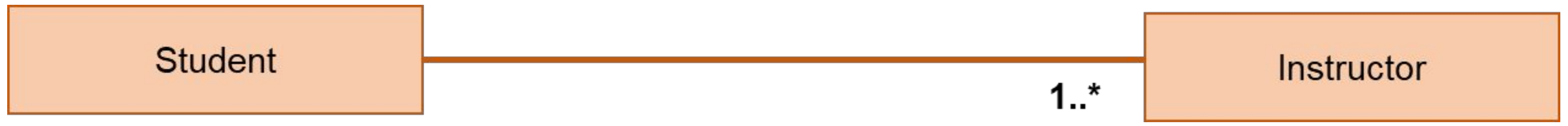
Multiplicities are single number or ranges of numbers.

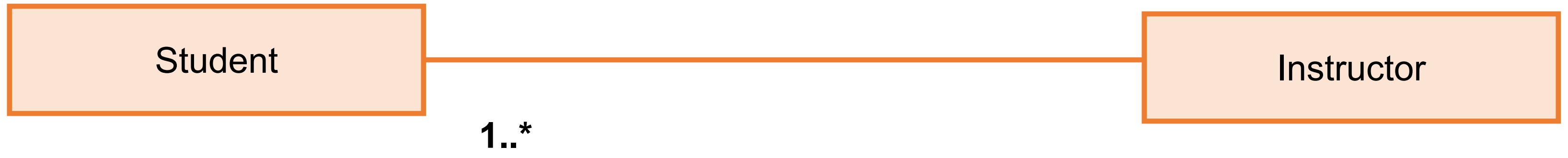


| **Multiplicities** | **Meaning** |
|---|---|
| **0..1** | zero or one instance. |
| **0..*** *or* ***** | no limit on the number of instances (including none). |
| **1** | exactly one instance |
| **1..*** | at least one instance |
| **n..m** | ***n*** to ***m*** instances (n and m stand for numbers, e.g. **0..4**, **3..15**) |
| **n** | exactly n instance (where n stands for a number, e.g. **3**) |

## Association Relationships (Cont’d)

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

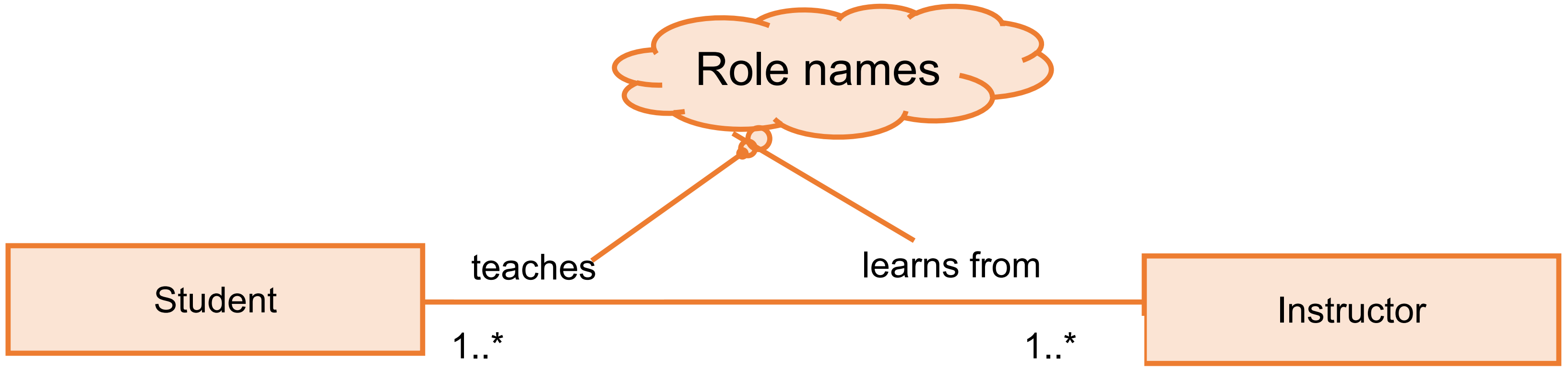




The example indicates that every *Instructor* has one or more *Students*:

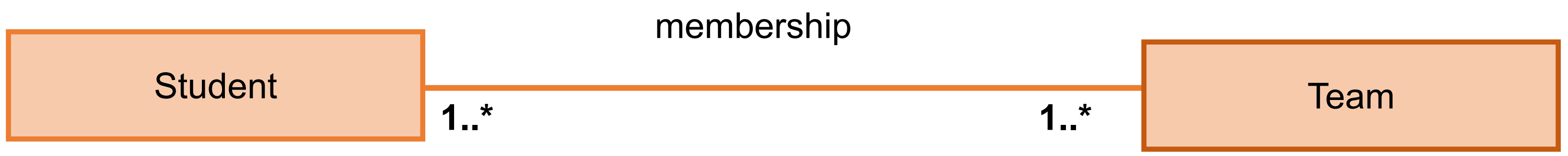
## Association Relationships (Cont'd)

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



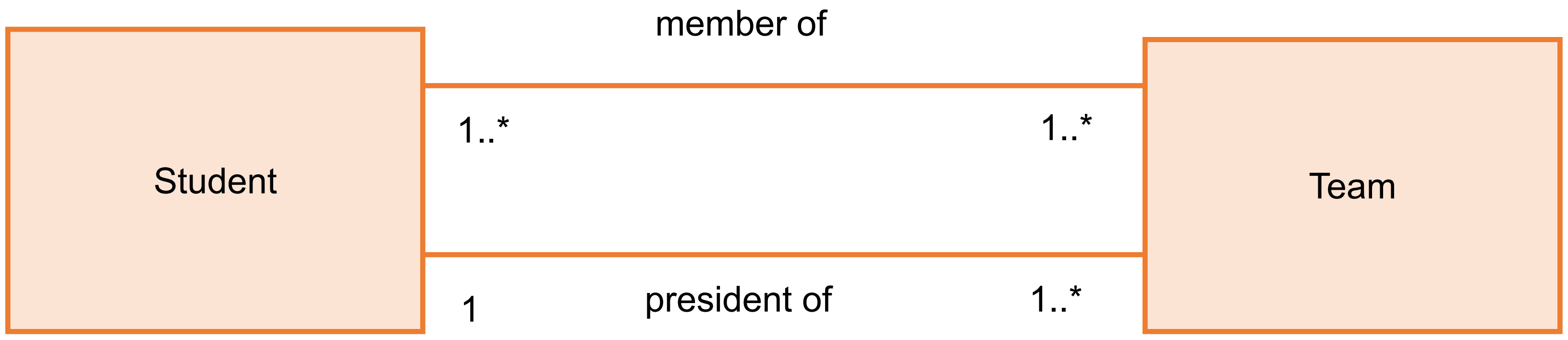
## Association Relationships (Cont'd)

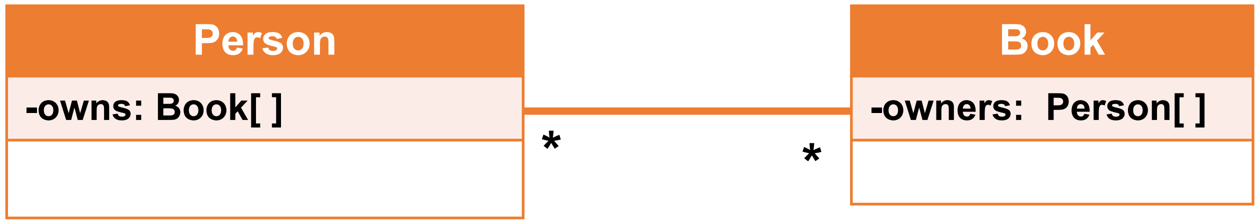
We can also name the association.



## Association Relationships (Cont'd)

We can specify dual associations using bidirectional association

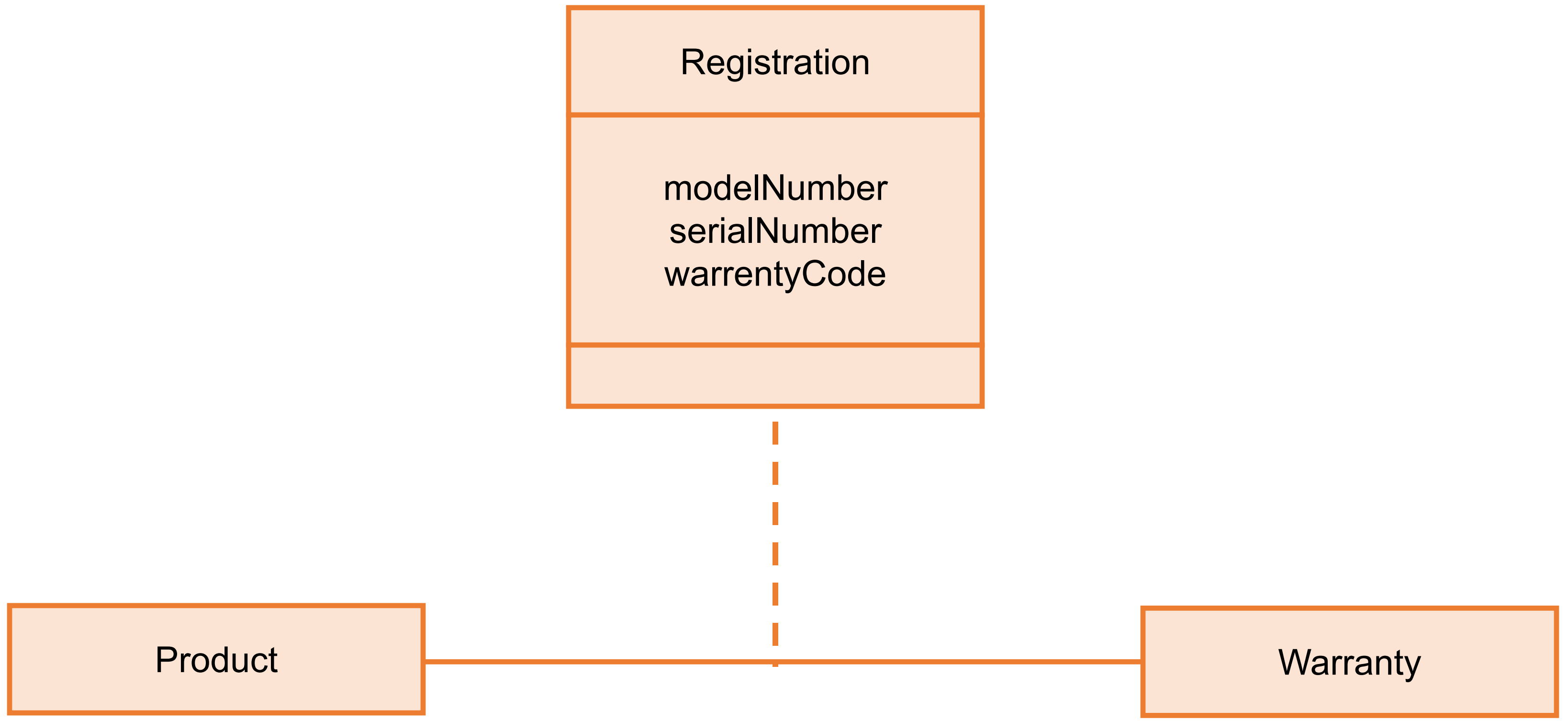




Two objects might store each other in fields. For example, in addition to a Person object listing all the books that the person owns, a Book object might list all the people that own it

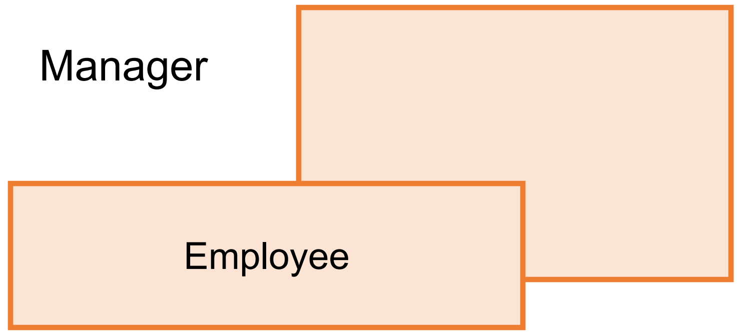
## Association Relationships (Cont'd)

Associations can also be objects themselves, called *link classes* or an *association classes.* An association class is represented like a normal class, but it is linked to an association line with a dotted line.



## Association Relationships (Cont'd)

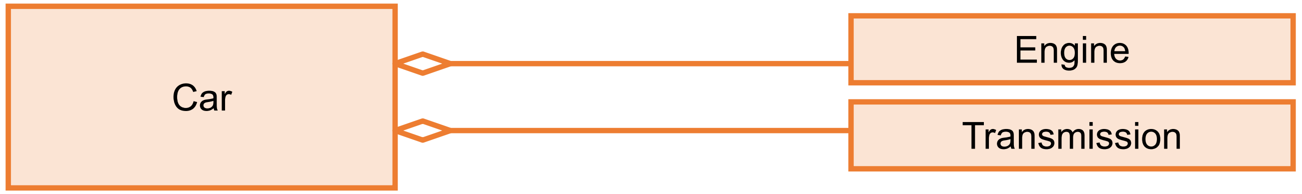
A class can have a *self association*.

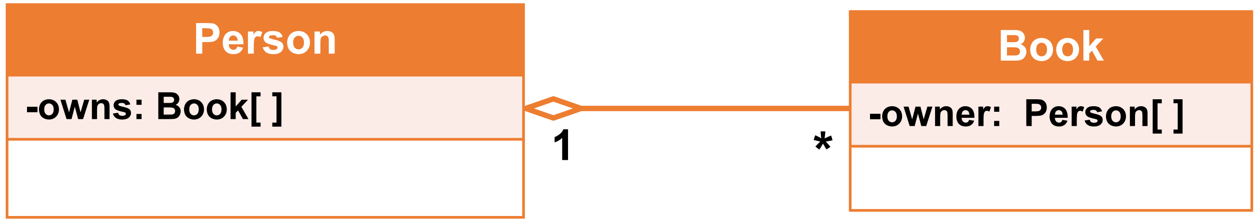


## Aggregation

We can model objects that contain other objects by way of special associations called *aggregations* and *compositions. It is also known as "has a" relationship.*

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.

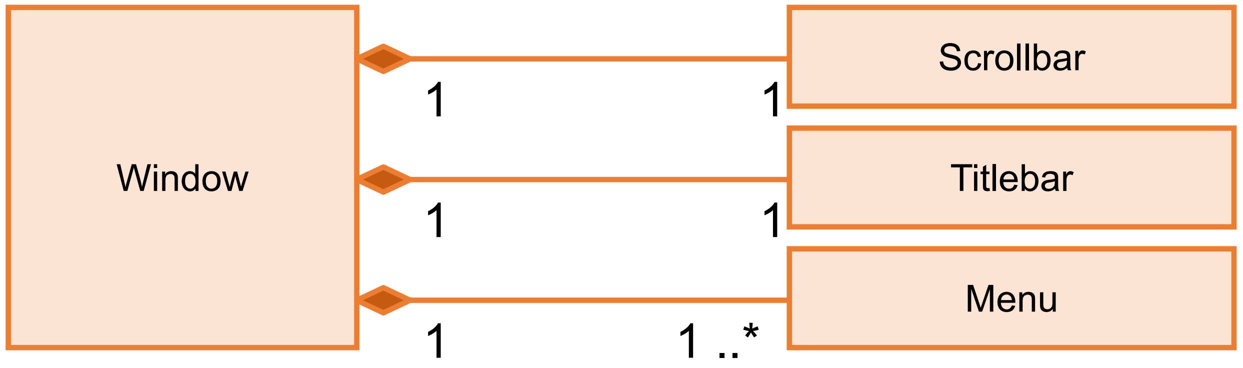


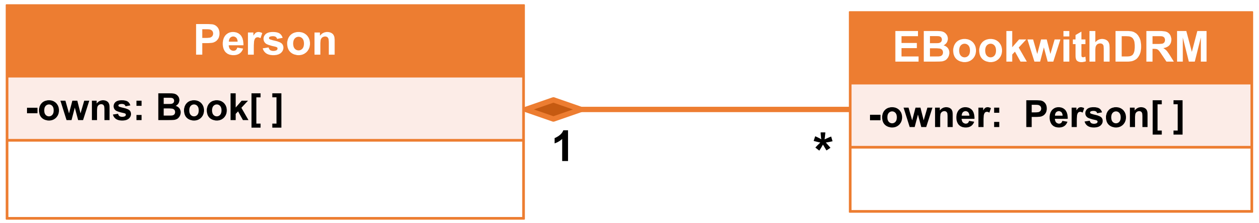


One object A has or owns another object B, and/or B is part of A. For example, suppose there are different Book objects for different physical copies. Then the Person object has/owns the Book object, and, while the book is not really part of the person, the book is part of the person's property. In this case, each book will (usually) have one owner. Of course, a person might own any number of books.

## Composition

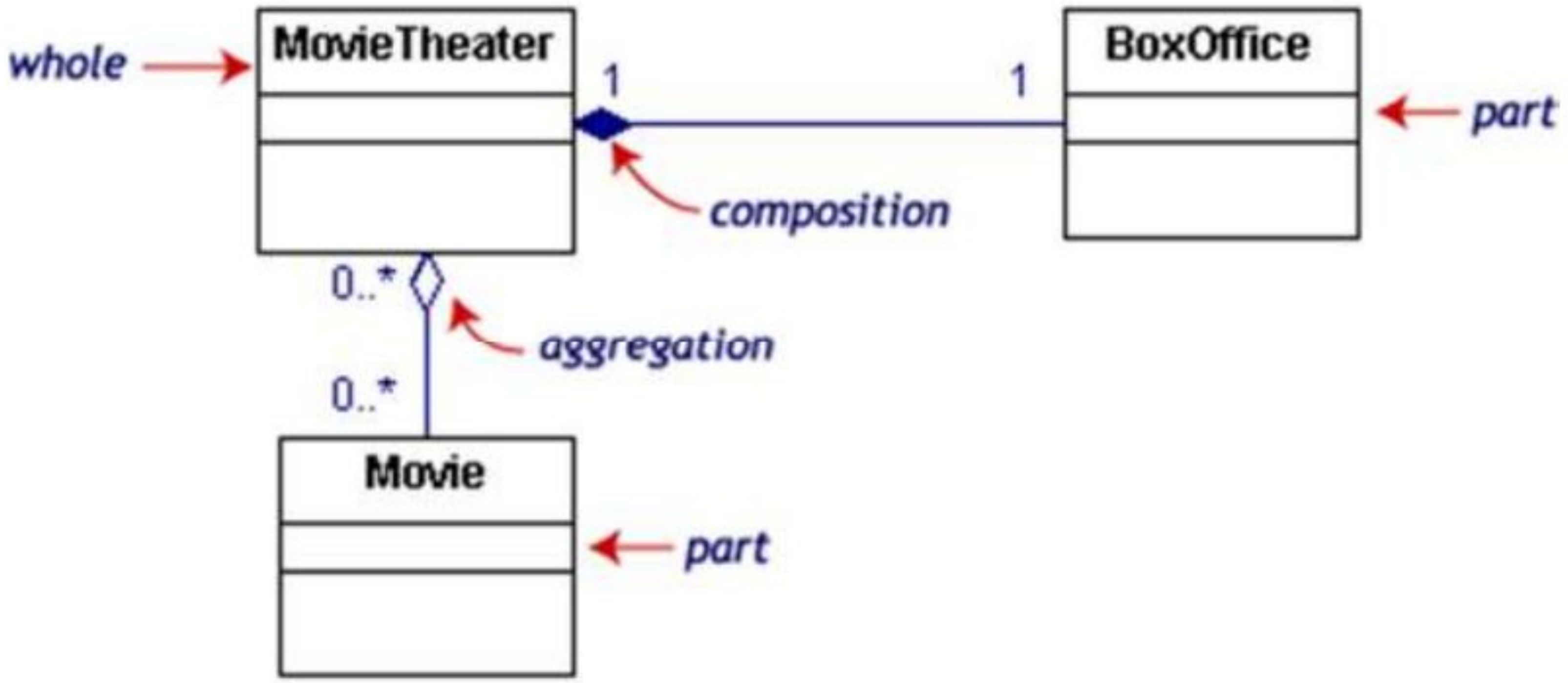
A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.,* they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.





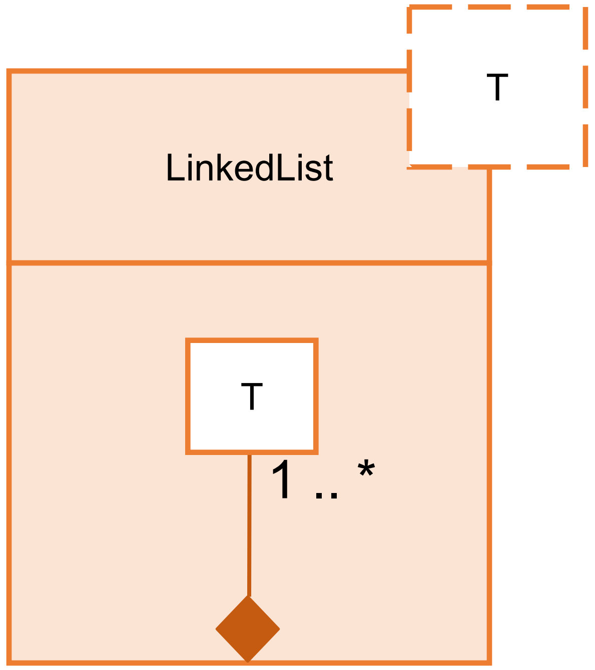
In addition to an aggregation relationship, the lifetimes of the objects might be identical, or near For example, in an idealized world of electronic books with DRM (Digital Rights Management), a person can own an ebook, but cannot sell it. After the person dies, no one else can access the ebook. [This is idealized, but might be considered less than ideal.]

## Composition / Aggregation Example



If the movie theater goes away
so does the box office => composition
but movies may still exist => aggregation

## Parameterized Class

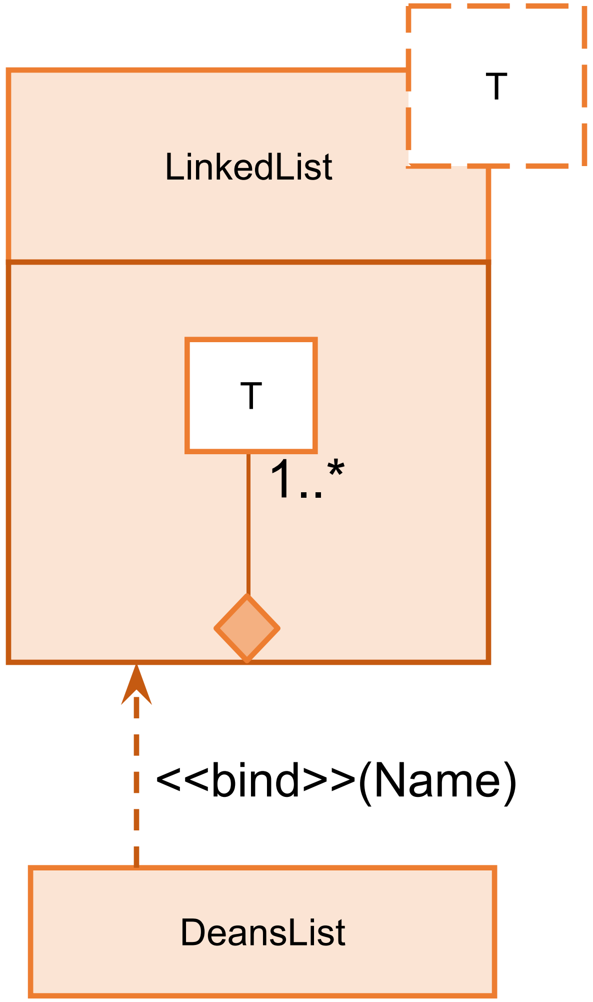


A *parameterized class* or *template* defines a family of potential elements.

To use it, the parameter must be bound.

A *template* is rendered by a small dashed rectangle superimposed on the upper-right corner of the class rectangle. The dashed rectangle contains a list of formal parameters for the class.

## Parameterized Class (Cont’d)



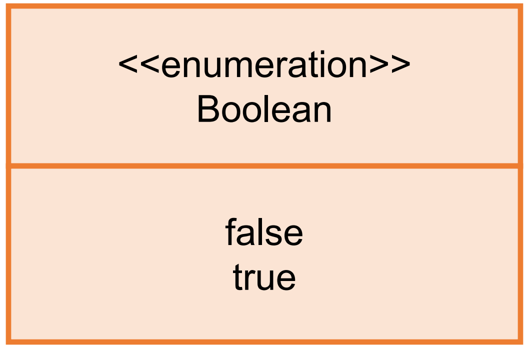
*Binding* is done with the <<bind>> stereotype and a parameter to supply to the template. These are adornments to the dashed arrow denoting the realization relationship.

Here we create a linked-list of names for the Dean’s List.

## Object Oriented Analysis and Design with Java

### Enumeration

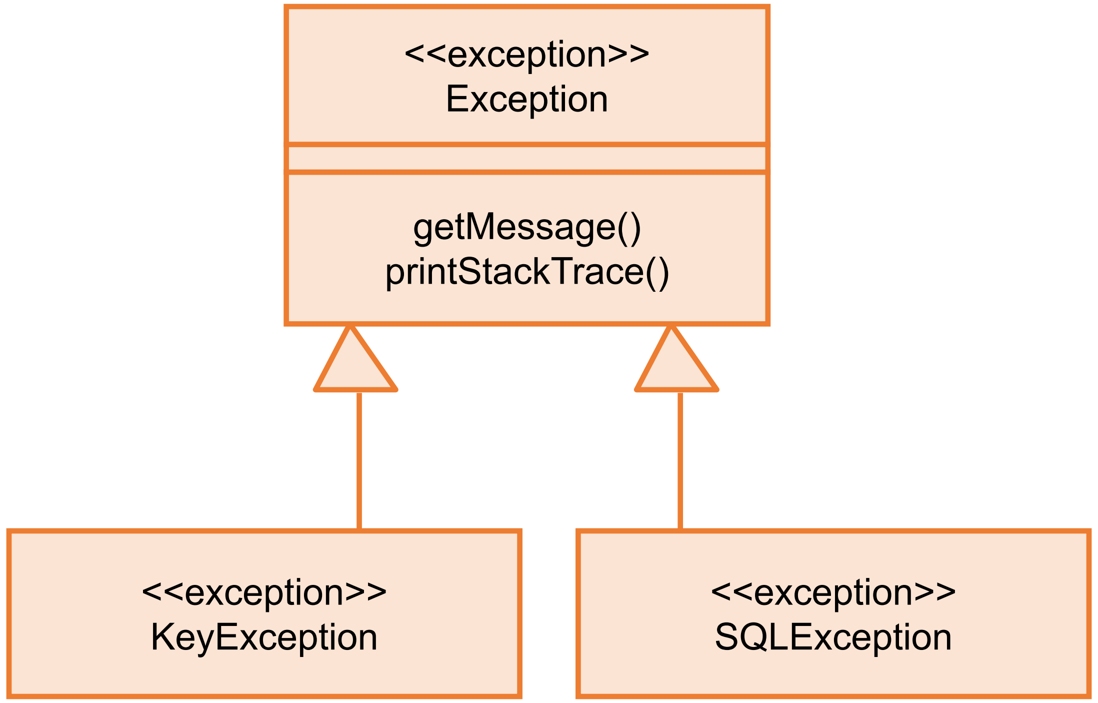
An *enumeration* is a user-defined data type that consists of a name and an ordered list of enumeration literals.



## Exceptions

*Exceptions* can be modeled just like any other class.

Notice the <<exception>> stereotype in the name compartment.



## How to make a class diagram

1.Identify all the classes participating in the software solution.

2. Draw them in a class diagram.

3. Identify the attributes.

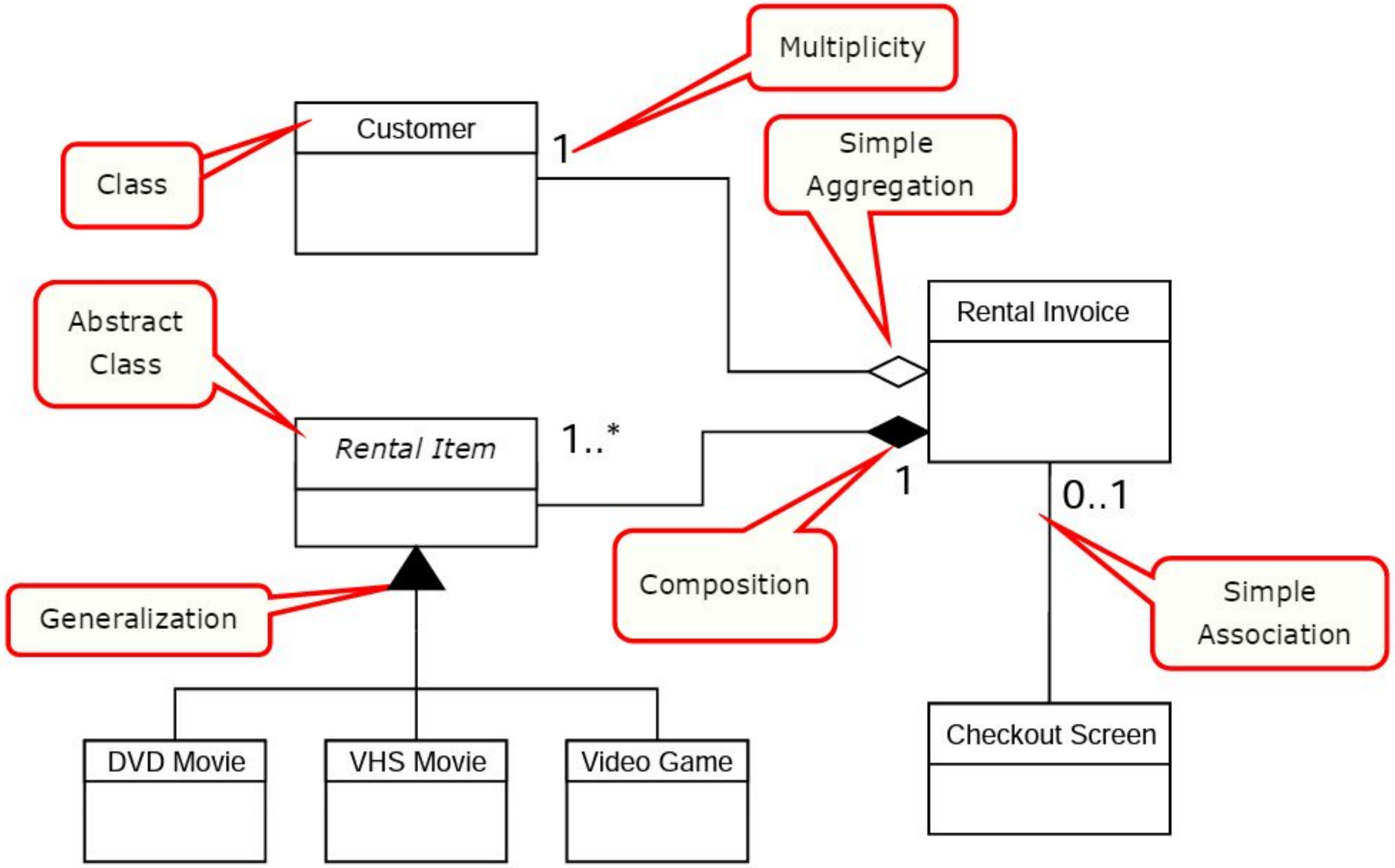
4. Identify the methods.

5. Add associations, generalizations, aggregations and dependencies.

6. Add other stuff (roles, constraints, ...)

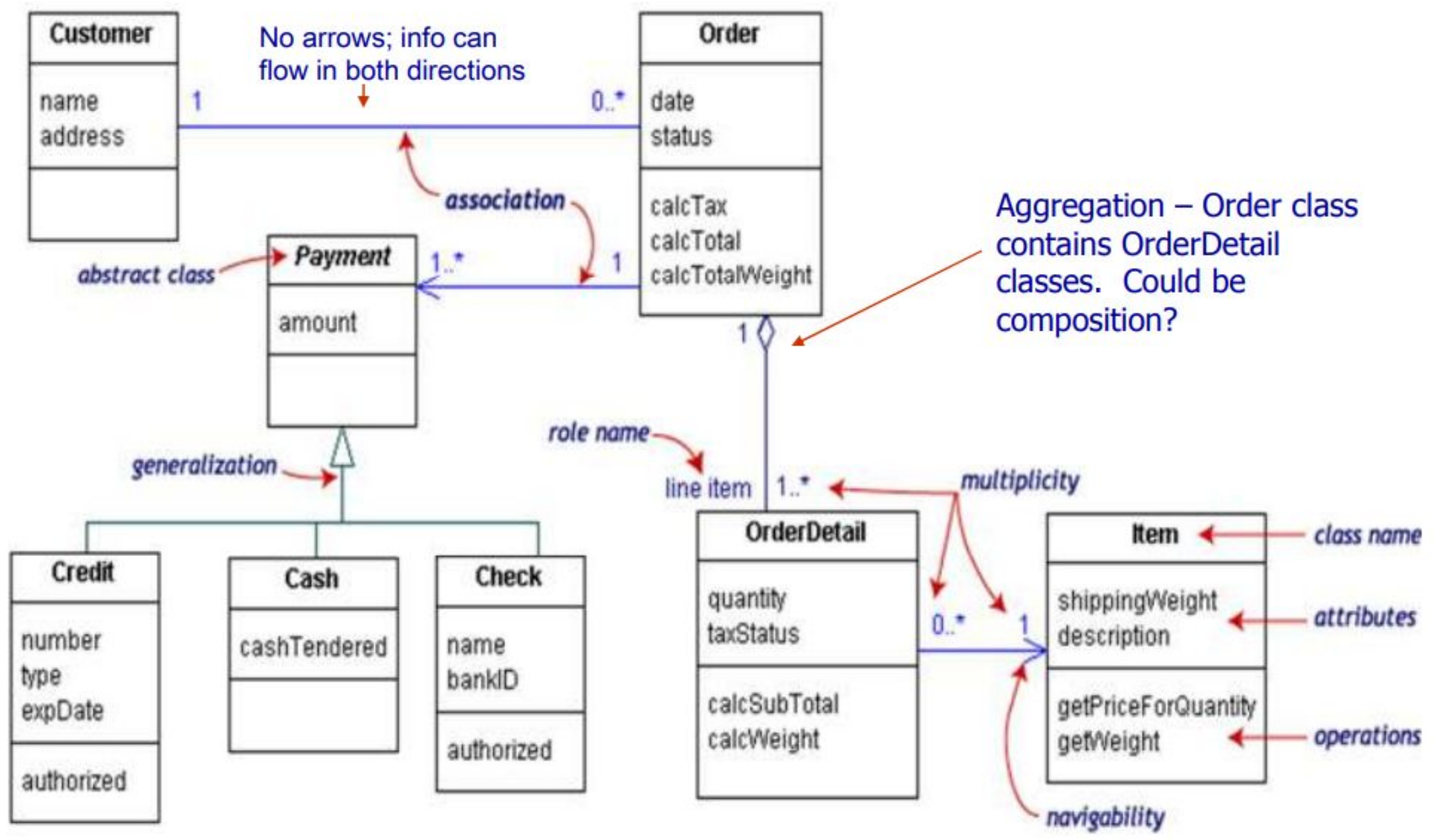
# Object Oriented Analysis and Design with Java

## Example class diagram –Video Store



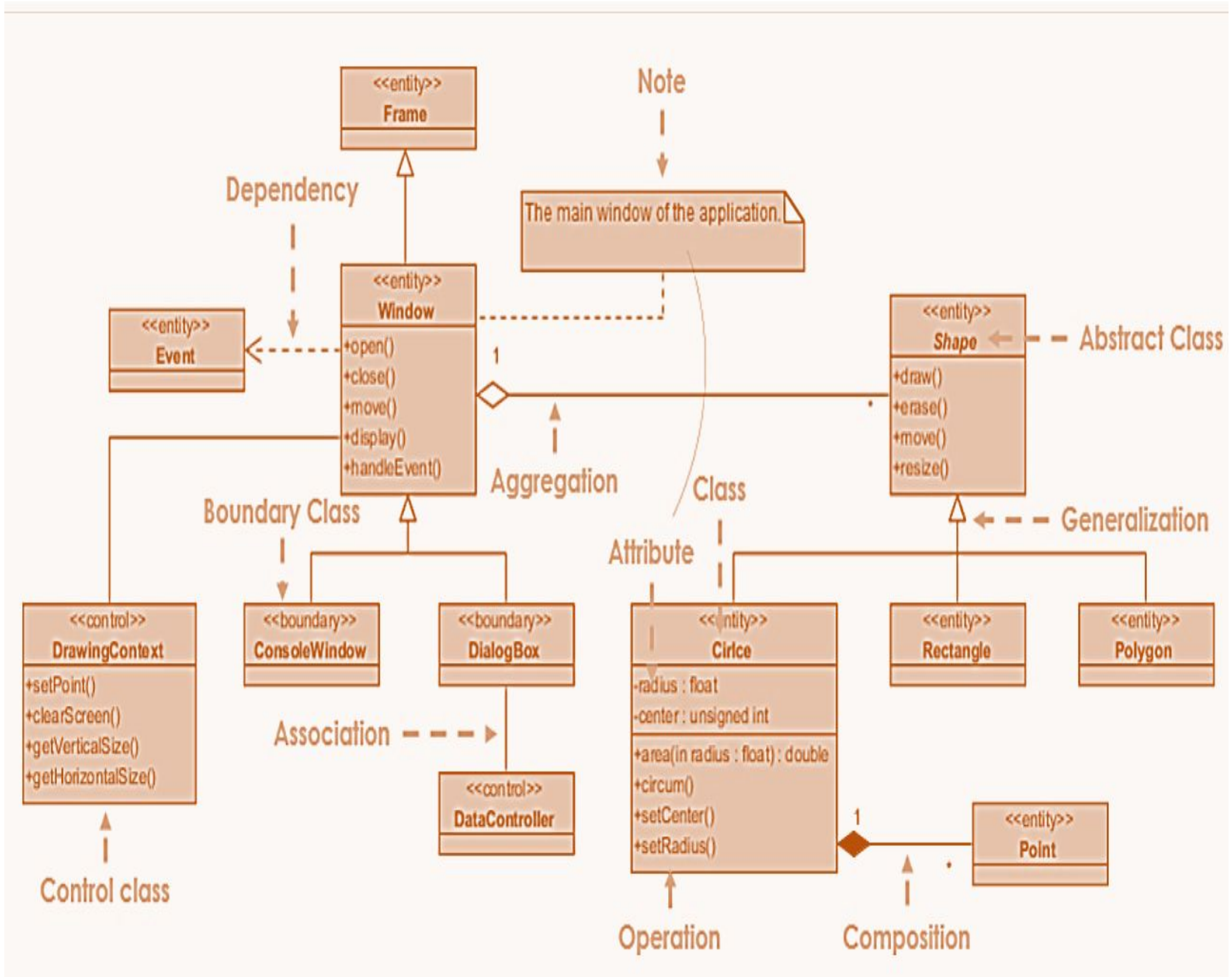
# Object Oriented Analysis and Design with Java

## Example class diagram –Video Store



# Object Oriented Analysis and Design with Java

## Example class diagram



Shape is an abstract class. It is shown in Italics.
Shape is a superclass. Circle, Rectangle and Polygon are derived from Shape. This is a generalization / inheritance relationship.
There is an association between DialogBox and DataController.
Shape is part-of Window. This is an aggregation relationship. Shape can exist without Window.
Point is part-of Circle. This is a composition relationship. Point cannot exist without a Circle.
Window is dependent on Event. However, Event is not dependent on Window.
The attributes of Circle are radius and center. This is an entity class.
The method names of Circle are area(), circum(), setCenter() and setRadius().
The parameter radius in Circle is an in parameter of type float.
The method area() of class Circle returns a value of type double.
The attributes and method names of Rectangle are hidden.
Some other classes in the diagram also have their attributes

## THANK YOU

**Prof J.Ruby Dinakar**

Department of Computer Science and Engineering

**rubydinakar@pes.edu**