SE-2002
SOFTWARE DESIGN AND ARCHITECTURE

RUBAB JAFFAR
RUBAB.JAFFAR@NU.EDU.PK

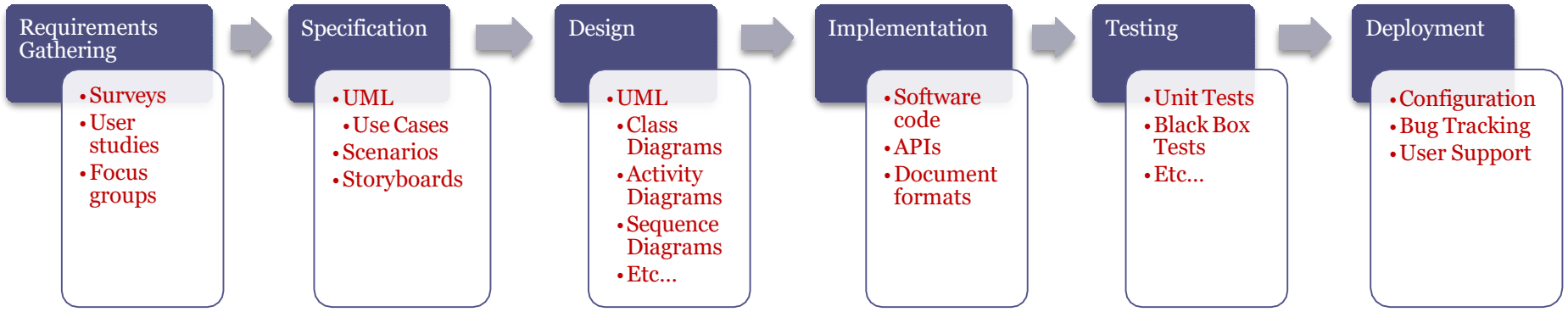
Class Diagram

Lecture # 10,11,12

TODAY'S OUTLINE

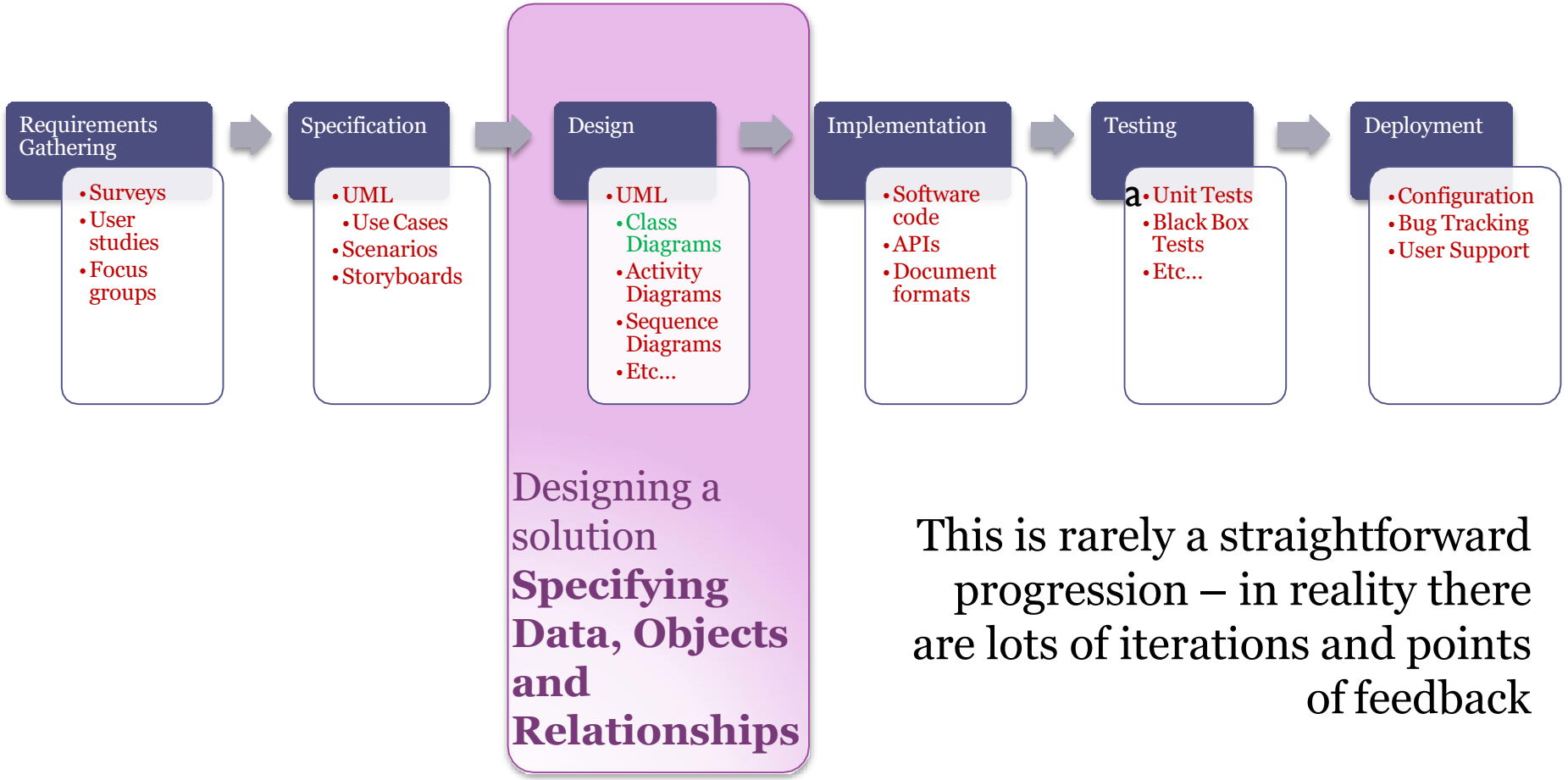
- Class Diagram
- Components of Class Diagram
- Relationships
- Exercises
- Use-case Realization
- Analysis Classes

OOAD: BIG PICTURE



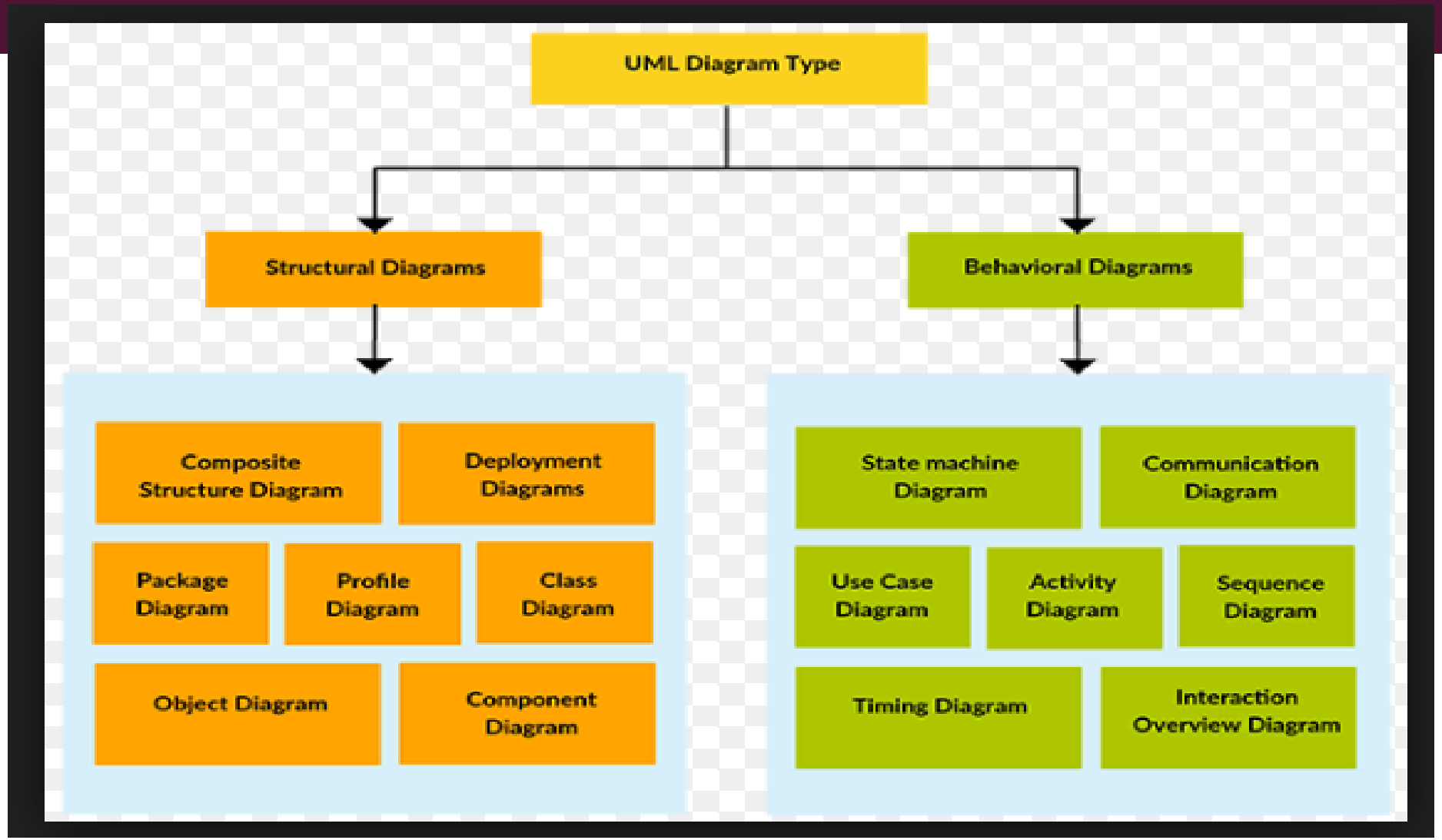
This is rarely a straightforward progression – in reality there are lots of iterations and points of feedback

OOAD: BIG PICTURE



This is rarely a straightforward progression – in reality there are lots of iterations and points of feedback

TYPES OF UML DIAGRAMS



TYPES OF DIAGRAMS

- 2 types of diagrams
  - Structure Diagrams
    - Provide a way for representing the data and static relationships that are in an information system
    - You are connecting different parts together to get the final design.
  - Behavior Diagrams
    - Behavioral modeling refers to a way to model the system based on its functionality.

WHAT IS UML CLASS DIAGRAM?

- What is a UML class diagram?
- Imagine you were given the task of drawing a family tree. The steps you would take would be:

  - ❖ Identify the main members of the family
  - ❖ Identify how they are related to each other
  - ❖ Find the characteristics of each family member
  - ❖ Determine relations among family members
  - ❖ Decide the inheritance of personal traits and characters

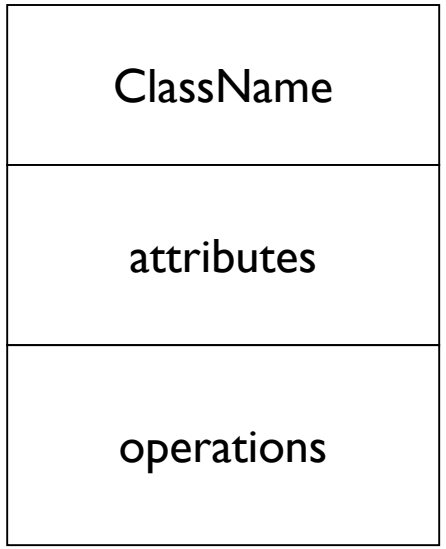
RELATIONSHIP BETWEEN CLASS DIAGRAM AND USE CASES

- How does a class diagram relate to the use case diagrams that that we learned before?
- When you designed the use cases, you must have realized that the use cases talk about "what are the requirements" of a system.
- The aim of designing classes is to convert this "what" to a "how" for each requirement
- Each use case is further analyzed and broken up that form the basis for the classes that need to be designed

ELEMENTS OF A CLASS DIAGRAM

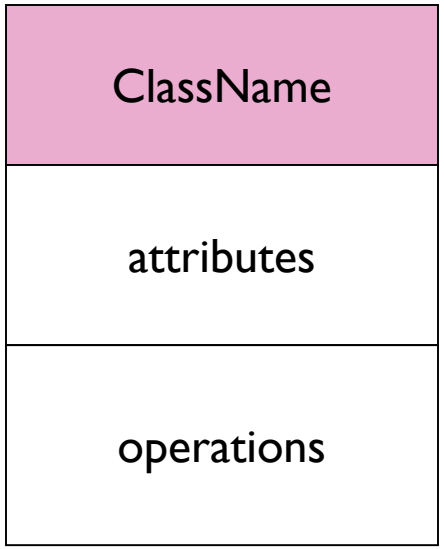
- A class diagram is composed primarily of the following elements that represent the system's business entities:
  - Class
  - Class Relationships

CLASSES



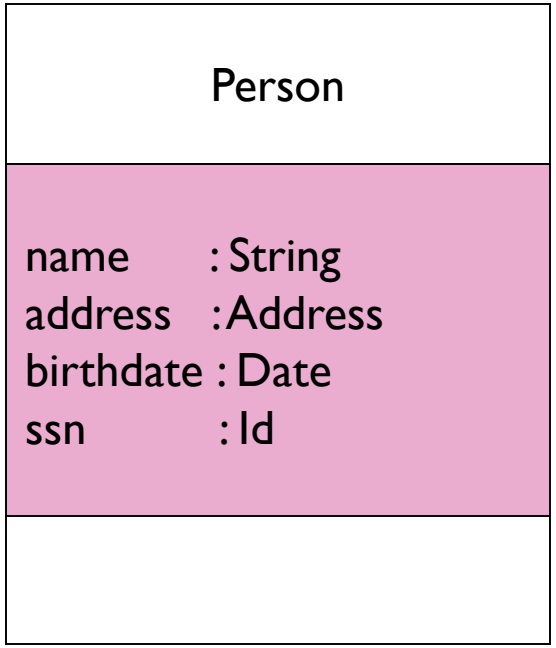
- A class represents an entity of a given system that provides an encapsulated implementation of certain functionality of a given entity. These are exposed by the class to other classes as methods. Apart from functionality, a class also has properties that reflect unique features of a class. The properties of a class are called attributes.
- Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

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

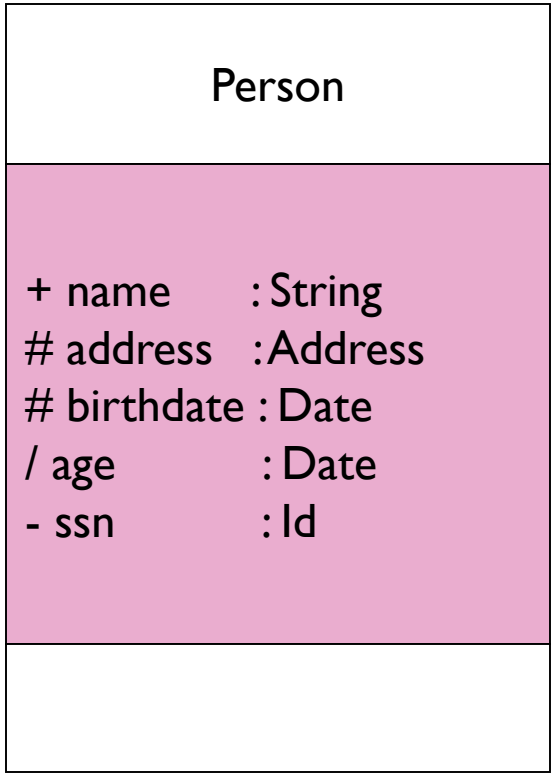


- An *attribute* is a named property of a class that describes the object being modeled.
- In the class diagram, attributes appear in the second compartment just below the name-compartment.

- Attributes are usually listed in the form:

attributeName : Type

CLASS ATTRIBUTES-VISIBILITY(ACCESS SPECIFIERS)



Attributes can be:

+ public

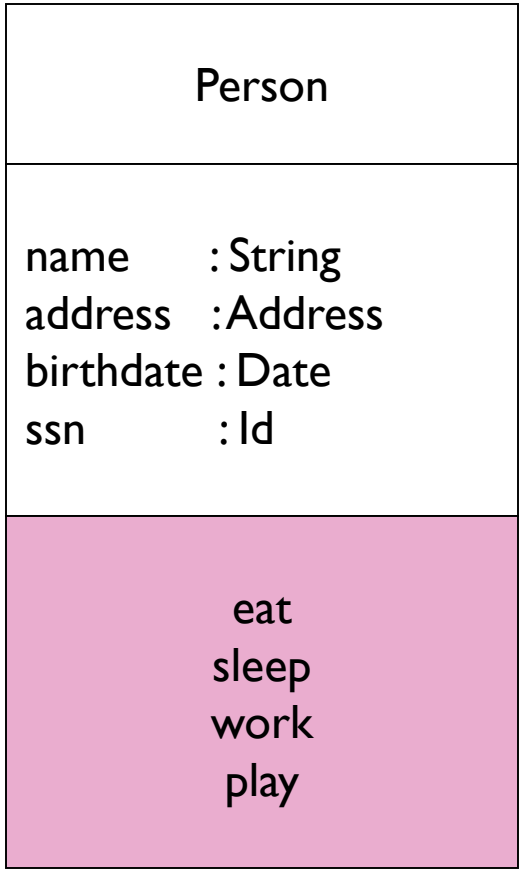
protected

- private

~ package

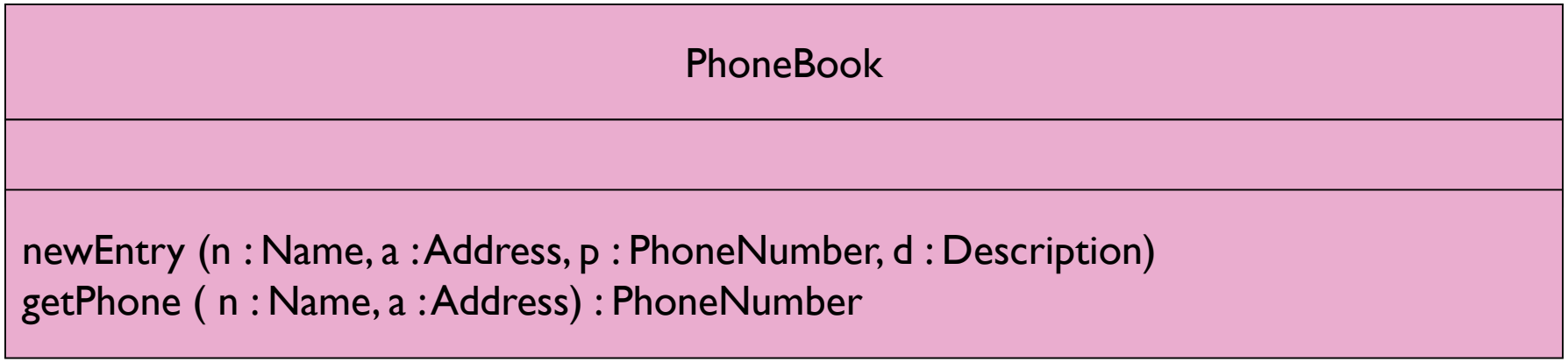
/ derived

OPERATIONS



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

CLASS OPERATIONS (CONT'D)



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.

The full UML syntax for attribute list is

name : attribute type

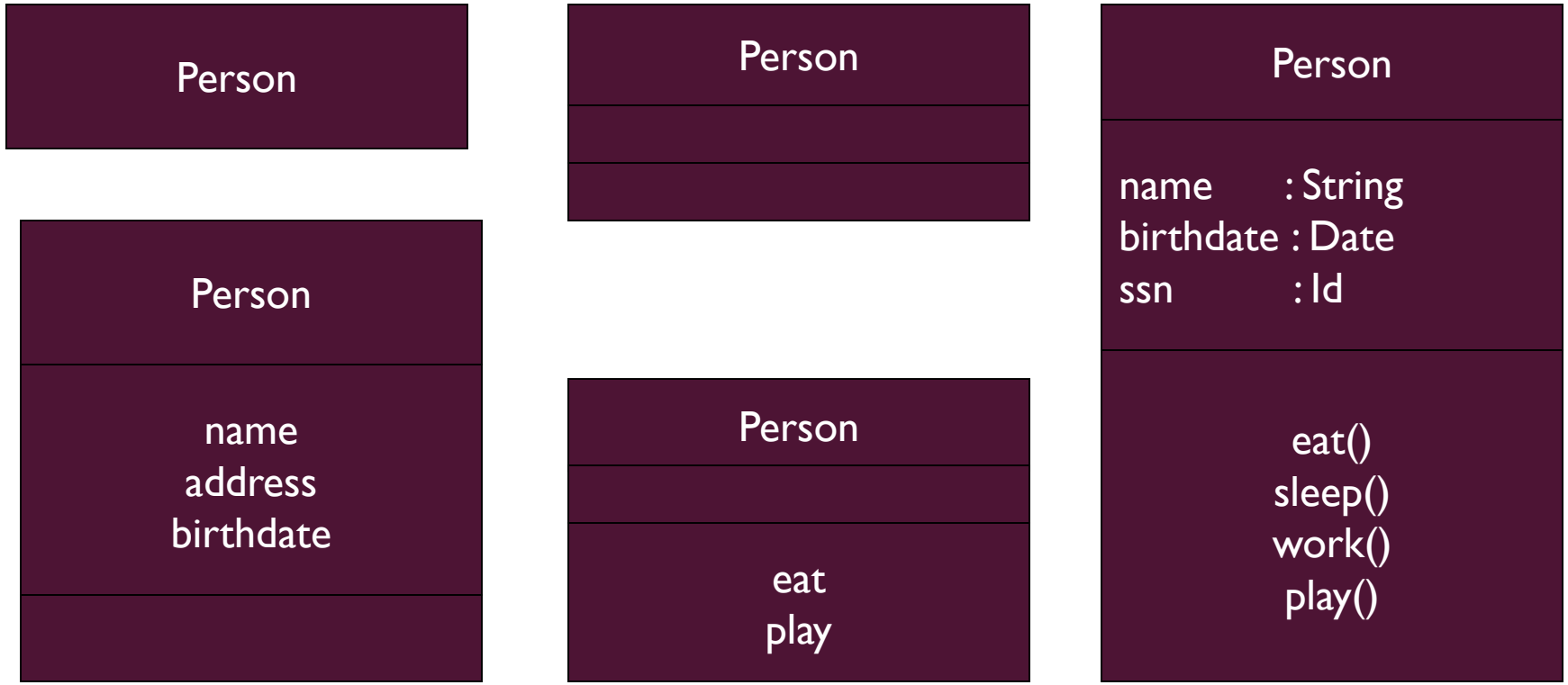
flightNumber : Integer

The full UML syntax for operations is

visibility name (parameter-list) : return-type

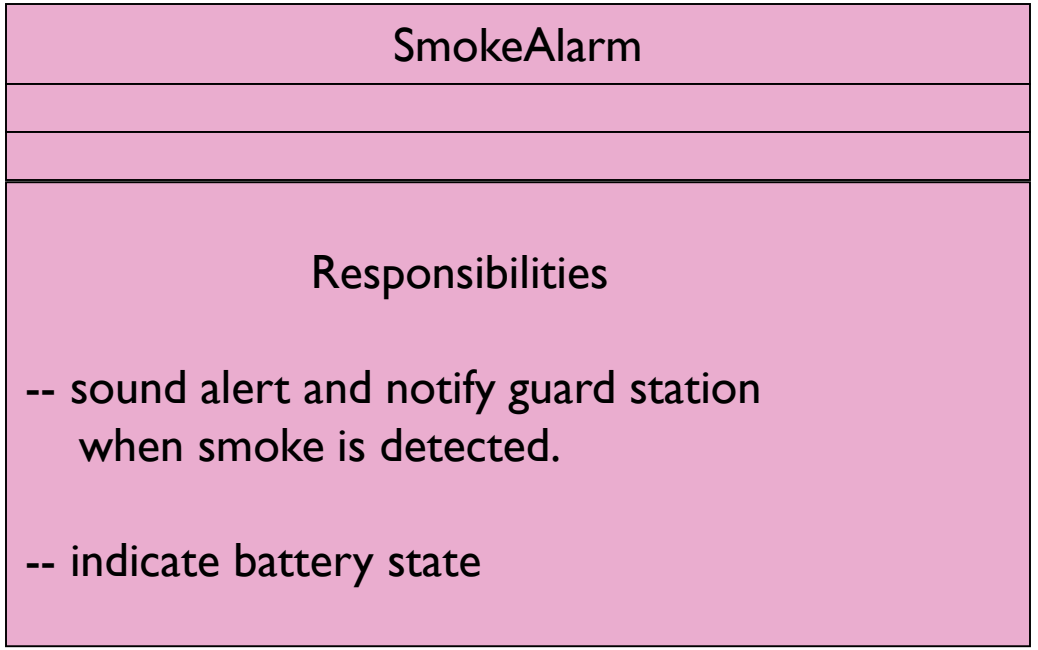
CLASSES

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



CLASS RESPONSIBILITIES

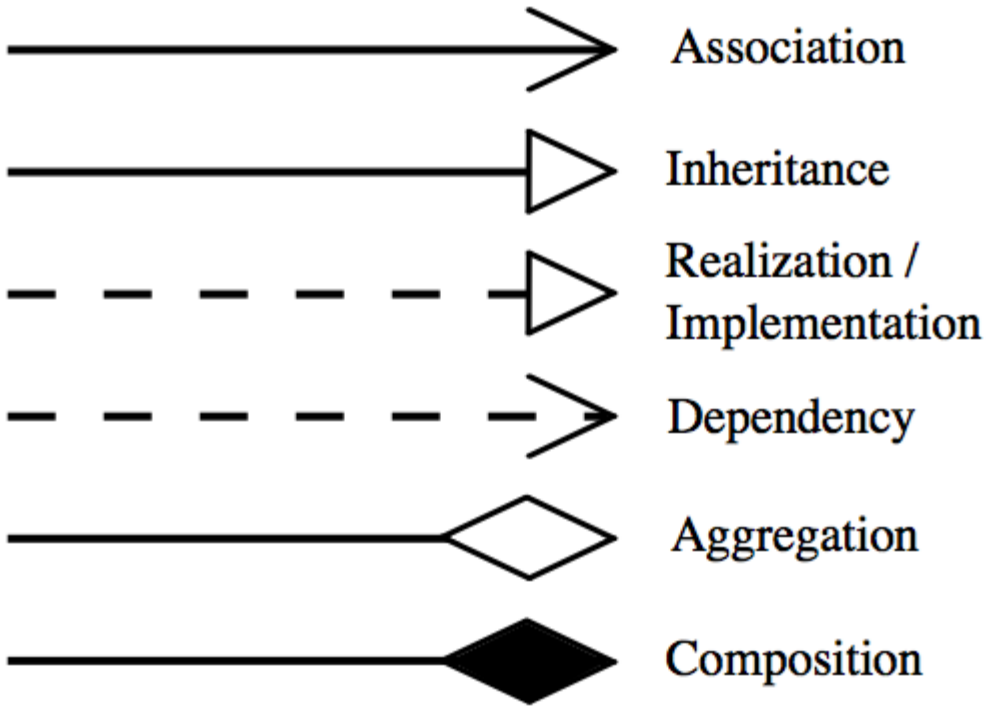
A class may also include its responsibilities in a class diagram.
A responsibility is a contract or obligation of a class to perform a particular service.



RELATIONSHIPS

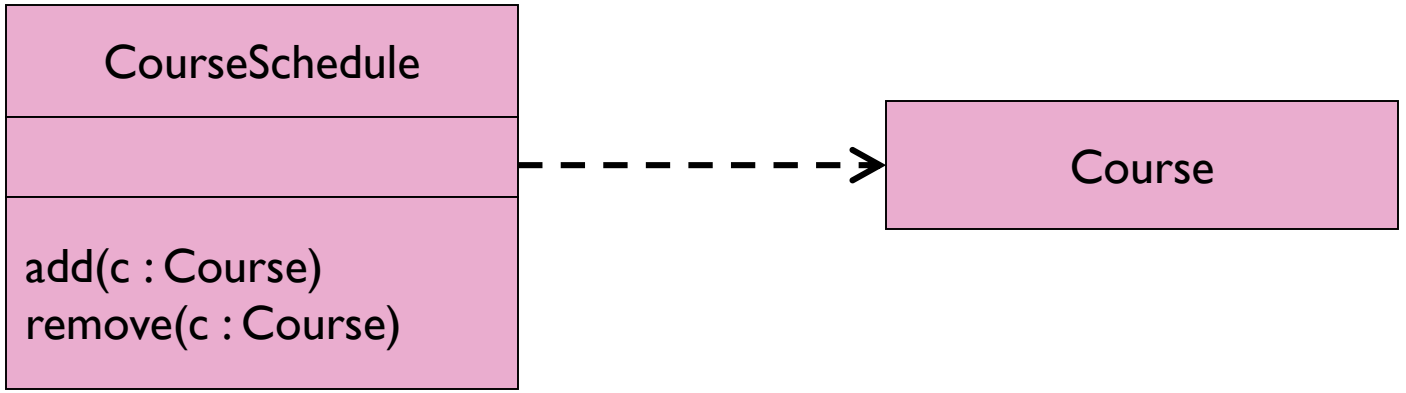
In UML, object interconnections (logical or physical), are modeled as relationships.

There are six kinds of relationships in UML:



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*.

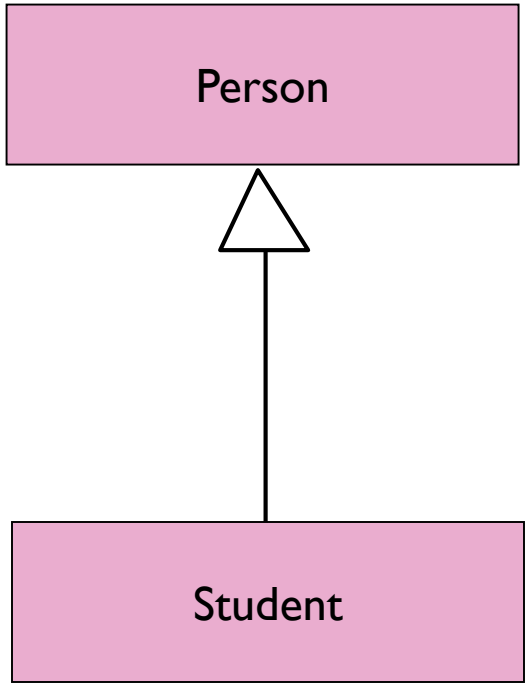


DEPENDENCY

- Dependency is represented when a reference to one class is passed in as a method parameter to another class.
- Dependency is a relationship between two things in which change in one element also affects the other.
- For example, an instance of class B is passed in to a method of class A:

```
Import class B
public class A {
    public void doSomething(B b) {
        }
```

GENERALIZATION RELATIONSHIPS



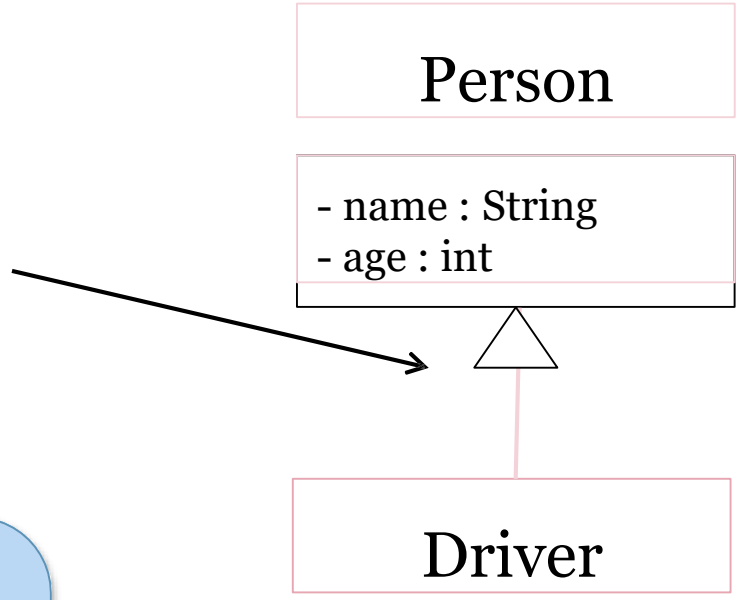
A *generalization* connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

```
public class Person {
...
} // class Person
public class Student extends
Person {
....
} // class Student
```

UML CLASS DIAGRAMS: GENERALIZATION

Drivers are a type of person. Every person has a name and an age.

Note: we use a special kind of arrowhead to represent generalization

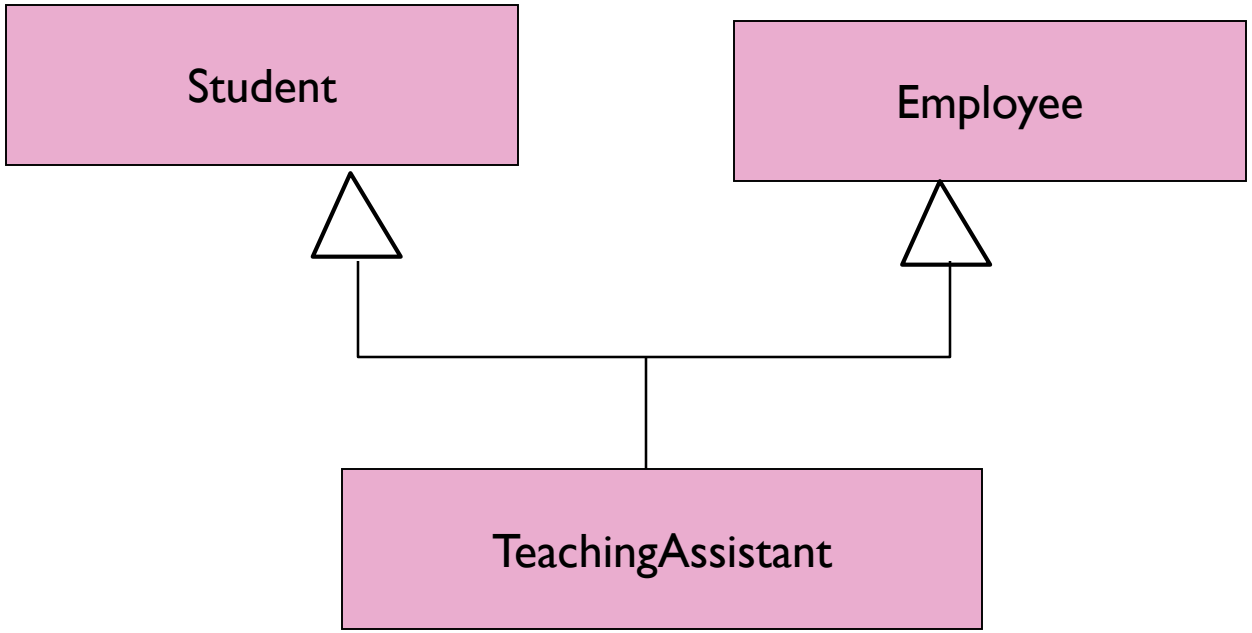



```
public Person {
...
} // class Person
public class Driver extends Person{
....
} // class Driver
```

We assume that Driver **inherits** all the properties and operations of a Person (as well as defining its own)

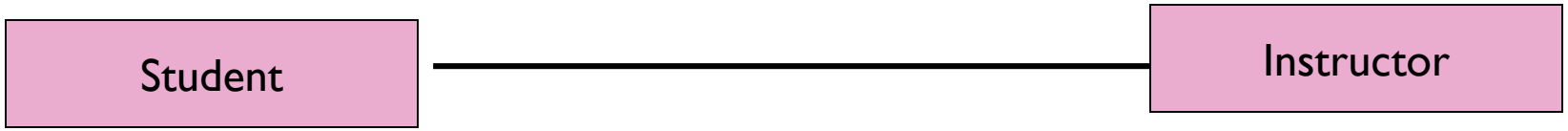
GENERALIZATION RELATIONSHIPS (CONT'D)

UML permits a class to inherit from multiple superclasses,



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.
- Usually an object provides services to several other objects
- An object keeps associations with other objects to delegate tasks



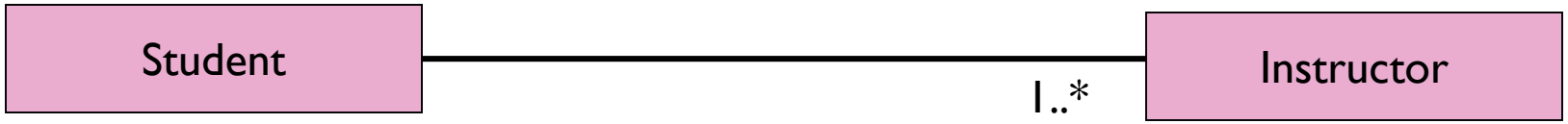
MULTIPLICITY

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

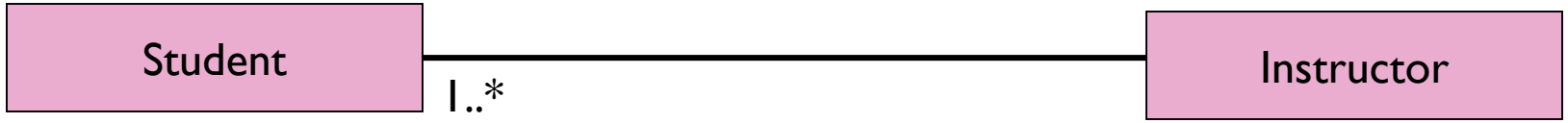
- Multiplicity (how many are used) □
- * ⇒ 0, 1, or more
- 1 ⇒ 1 exactly □
- 2..4 ⇒ between 2 and 4,
- inclusive □3□1.* ⇒ 3 or more

ASSOCIATION

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

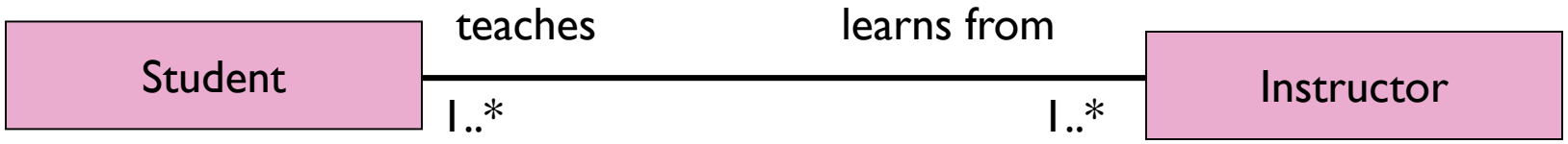


The example indicates that every *Instructor* has one or more *Students*:

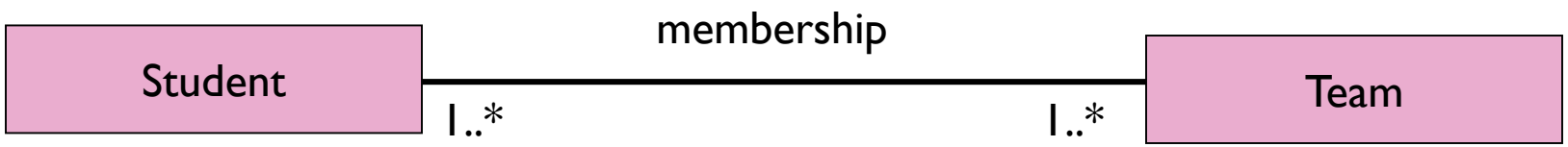


ASSOCIATION RELATIONSHIPS (CONT'D)

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

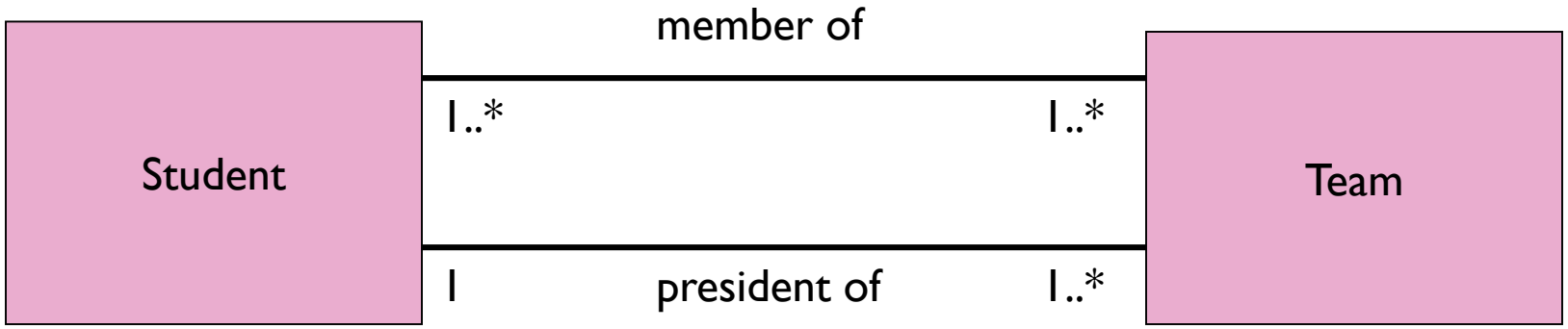


We can also name the association.



ASSOCIATION RELATIONSHIPS (CONT'D)

We can specify dual associations.



KINDS OF ASSOCIATION

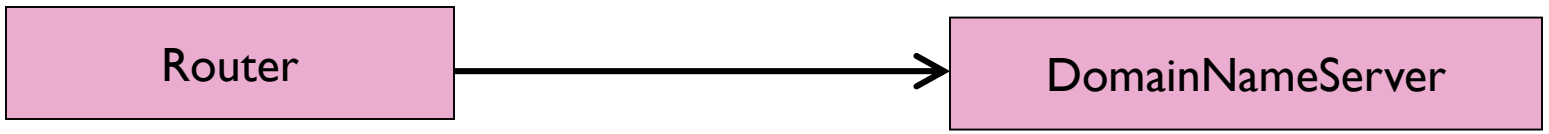
- Object Association
  - Simple Association: Is simply called as “association”
  - Composition
  - Aggregation

KINDS OF SIMPLE ASSOCIATION

- w.r.t navigation
  - One-way Association
  - Two-way Association
  - Self association
- w.r.t number of objects
  - Binary Association
  - Ternary Association
  - N-ary Association

ONE-WAY ASSOCIATION

- We can constrain the association relationship by defining the *navigability* of the association.
- In one way association, We can navigate along a single direction only
- Denoted by an arrow towards the server object
- 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*.



ONE WAY ASSOCIATION-PERSON-ADDRESS

```
class Person {
string Name;
Address *addr;
int Age;
public:
Person(){..}
~Person{..}
void setAddress(Address*
a)
{
addr = a; //shallow copy
}
};
```

```
class Address {

string Street;
long postalCode;
string Area;
…..
}
```

TWO-WAY ASSOCIATION

- We can navigate in both directions

- Denoted by a line between the associated objects

  - Employee works for company
  - Company employs employees

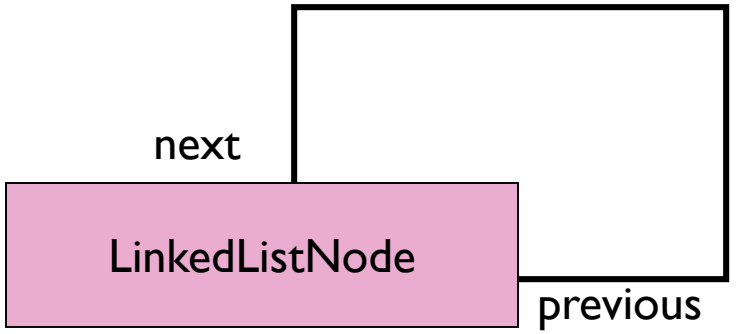
TWO WAY ASSOCIATION-CONTRACTOR-PROJECT

```
class Contractor
{
private:
string Name;
Project *MyProject;
…
};
```

```
class Project
{
string Name;
Contractor *person;
….
};
```

SELF ASSOCIATION

A class can have a *self association/ reflexive Association.*



Two instances of the same class:
Pilot
Aviation engineer

SELF ASSOCIATION

```
class Course
{
private:
    std::string m_name;
    Course *m_prerequisite;

public:
    Course(std::string &name, Course *prerequisite=nullptr):
        m_name(name), m_prerequisite(prerequisite)
    {
    }

};
```

W.R.T OBJECTS
BINARY ASSOCIATION

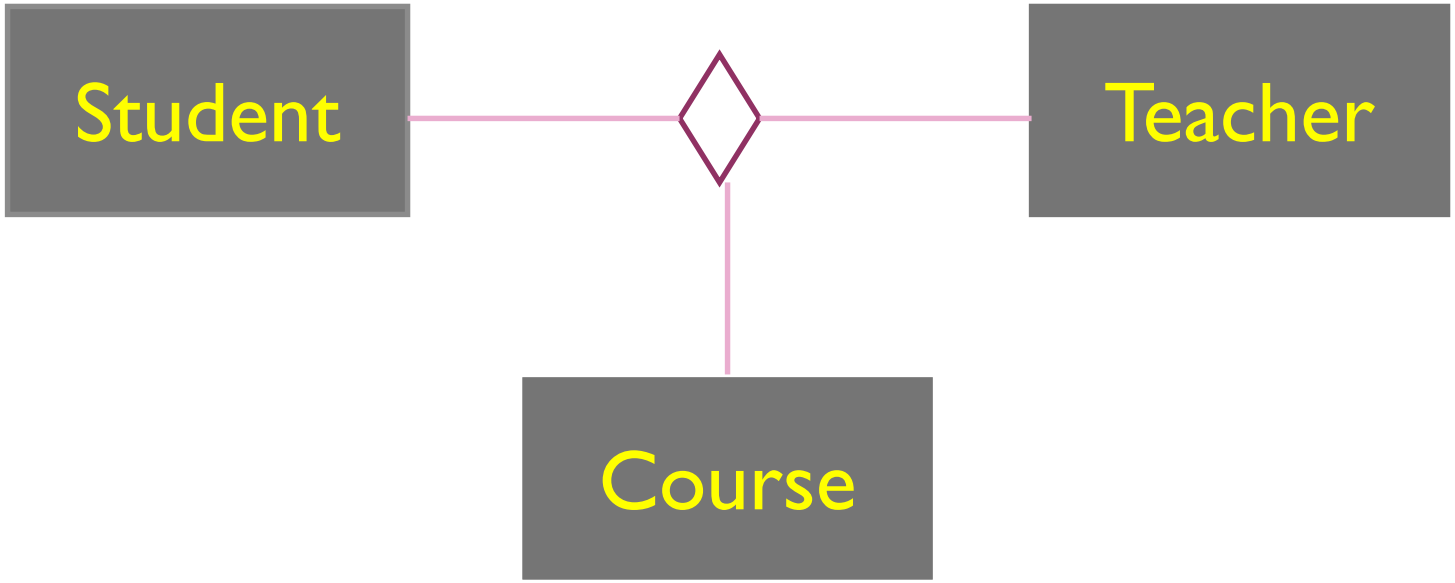
- Associates objects of exactly two classes

- Denoted by a line, or an arrow between the associated objects

- Association "works-for" associates objects of exactly two classes

TERNARY ASSOCIATION

- Associates objects of exactly three classes.
- Denoted by a diamond with lines connected to associated objects.



N-ARY ASSOCIATION

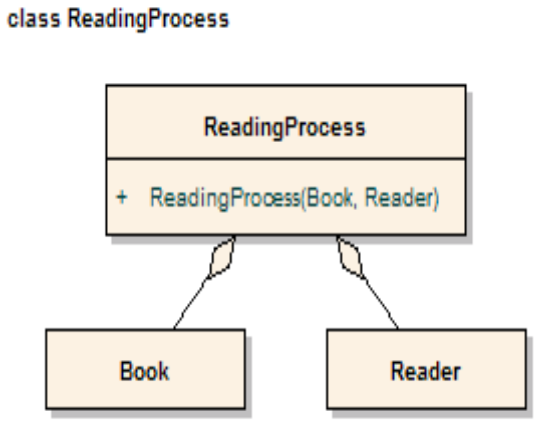
- An association between 3 or more classes
- Practical examples are very rare

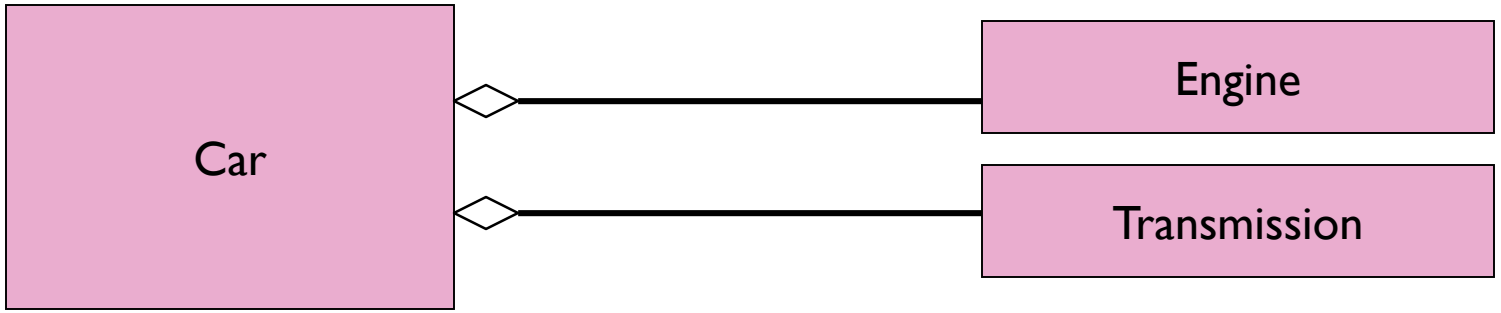
ASSOCIATION RELATIONSHIPS (CONT'D)

Aggregation:

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

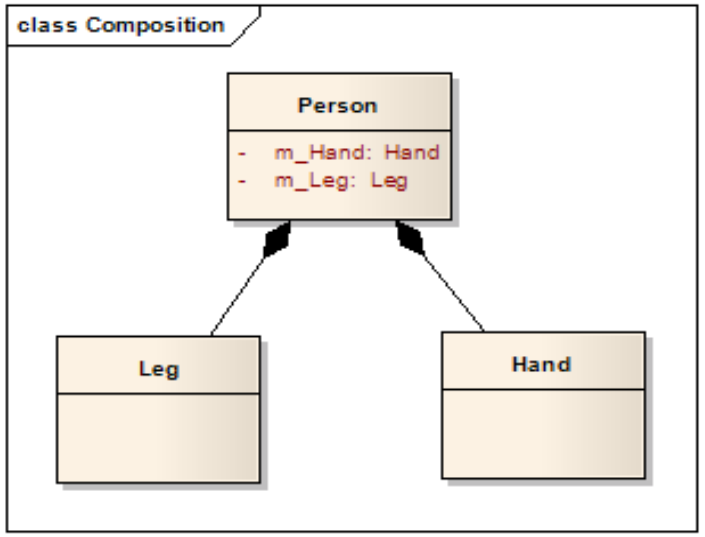
An ***aggregation*** specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.

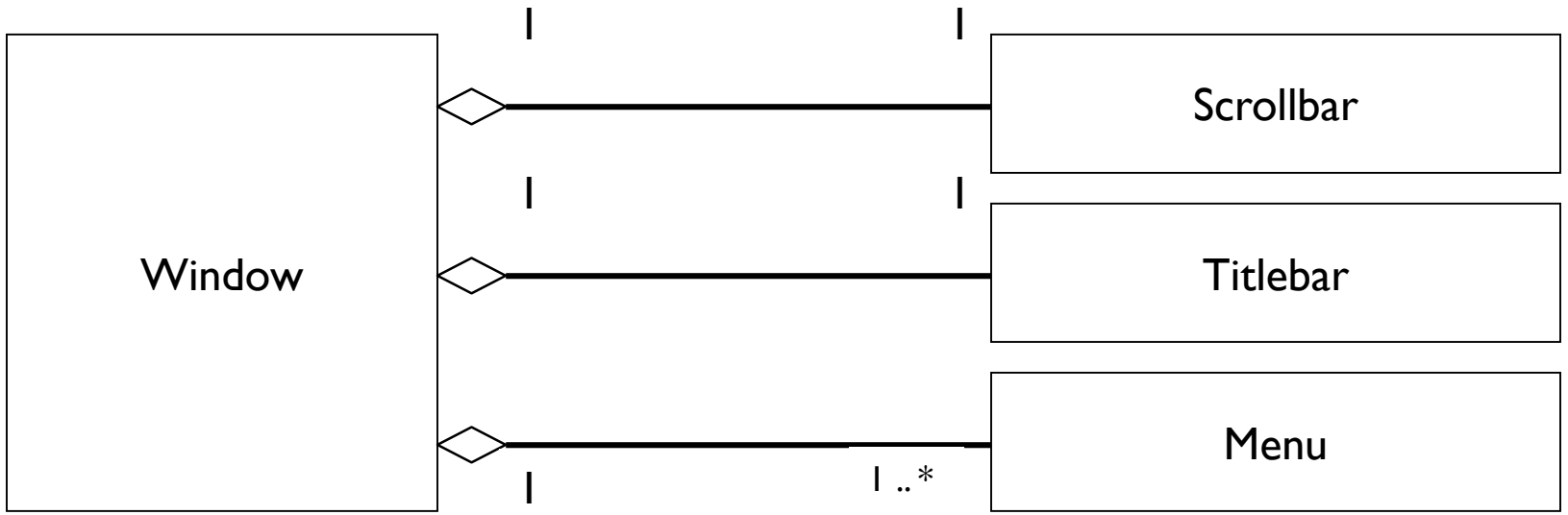




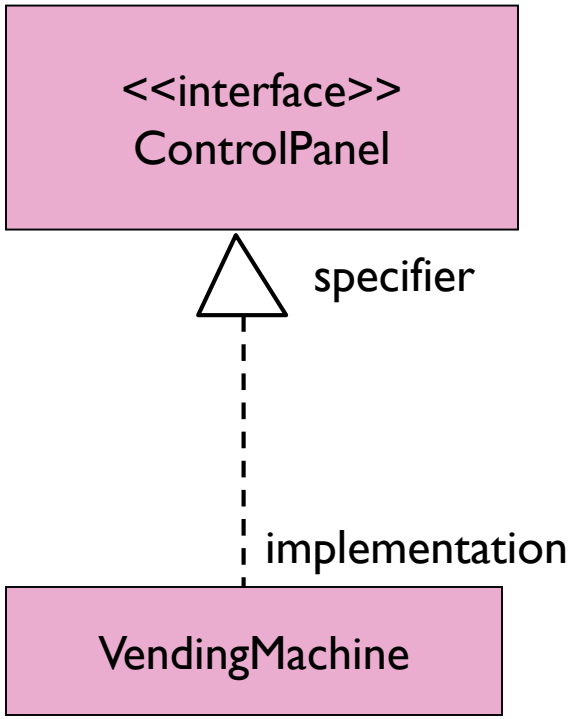
ASSOCIATION RELATIONSHIPS (CONT'D)

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.





INTERFACE REALIZATION RELATIONSHIP



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.

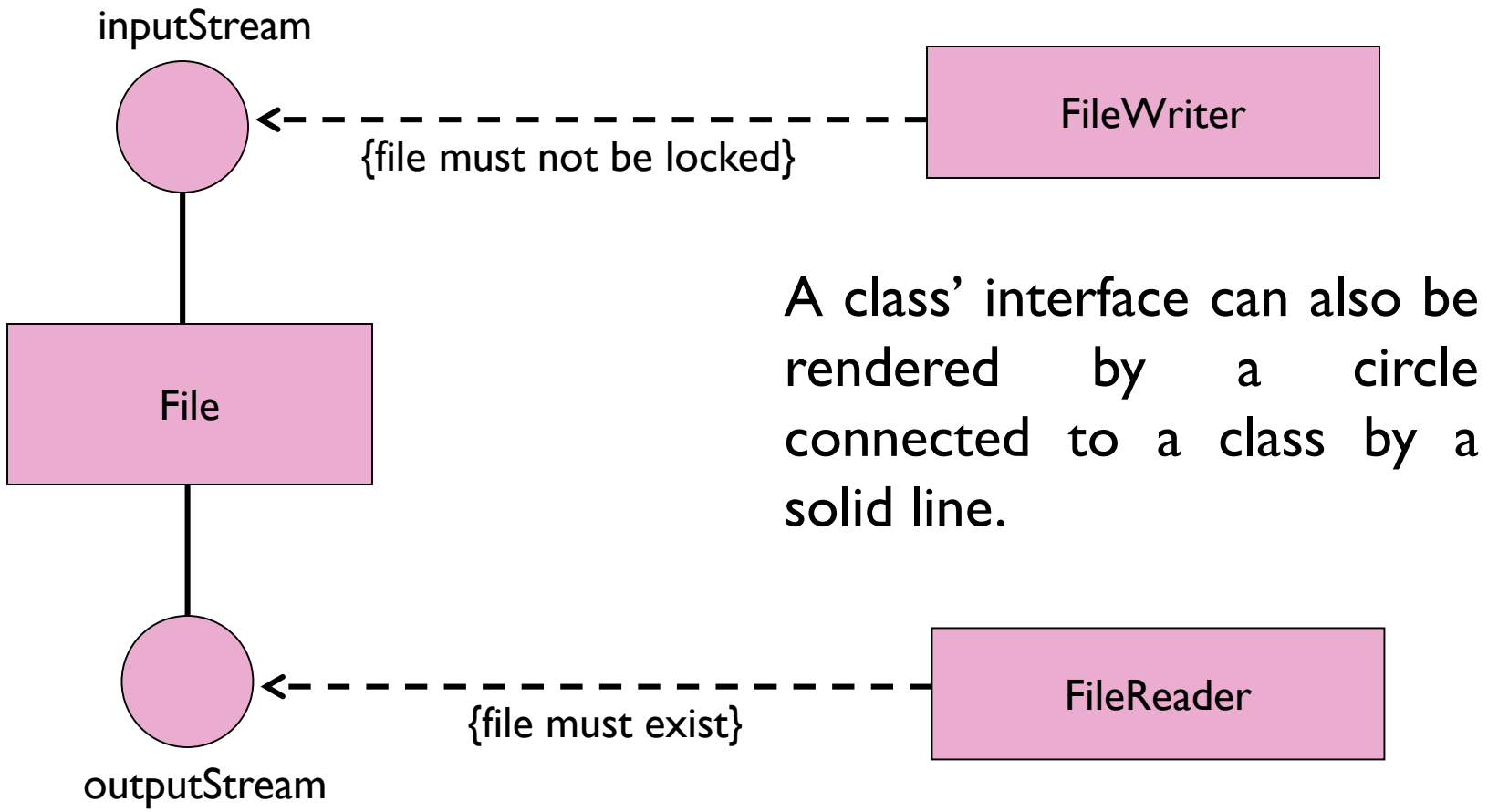
```
public interface A {
...
} // interface A
public class B implements A {
...
} // class B
```

Realization/Interface Implementation

```
interface Enemy
{
public void speak();
public void moveTo(int x, int  y);
public void attack(entity e);
}
```

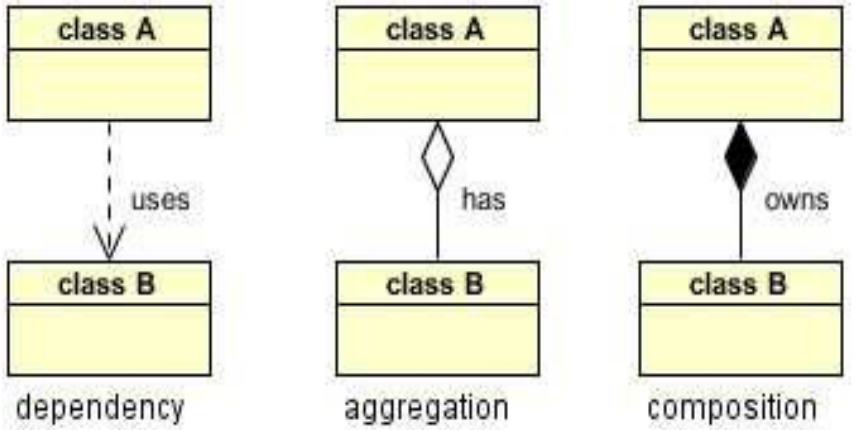
```
public class Player implements Enemy
{
public void speak()
{
//implementation goes here
}
public void moveTo()
 {
//implementation goes here
 }
public void attack()
{
//implementation goes here
}
}
```

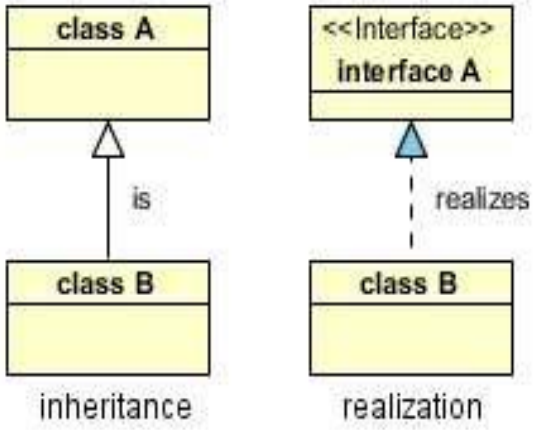
INTERFACES


inputStream
FileWriter
{file must not be locked}
File
A class' interface can also be rendered by a circle connected to a class by a solid line.
FileReader
{file must exist}
outputStream

RELATIONSHIPS IN A NUTSHELL

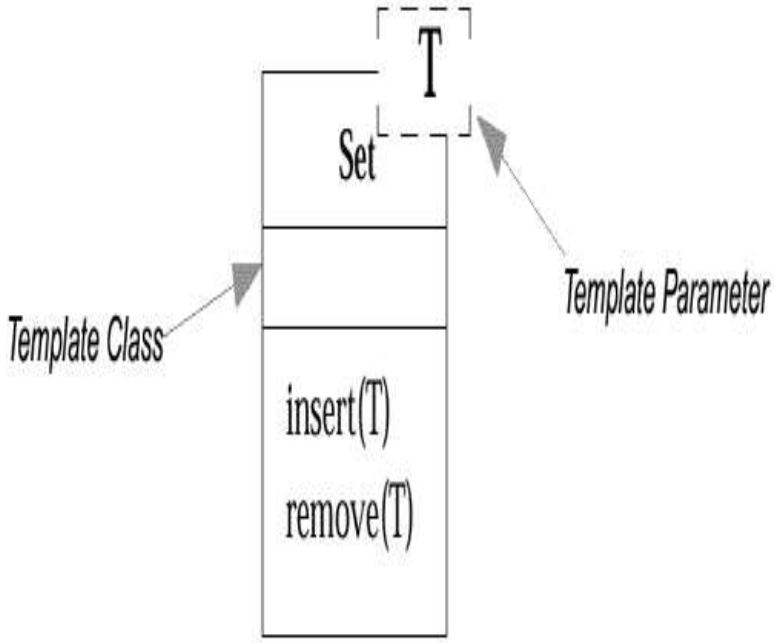
- Dependency : class A uses class B
- Aggregation : class A has a class B
- Composition : class A owns a class B
- Inheritance : class B is a Class A (or class A is extended by class B)
- Realization : class B realizes Class A (or class A is realized by class B)





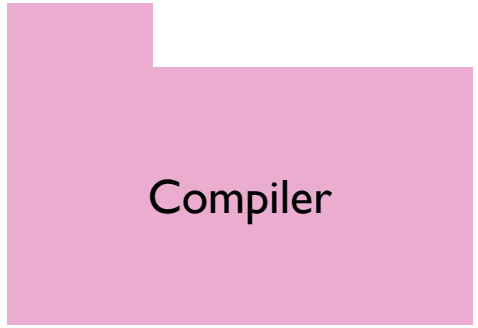
PARAMETERIZED CLASS

- 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.




```
class Set <T> {
void insert (T newElement);
void remove (T anElement);
```

PACKAGES

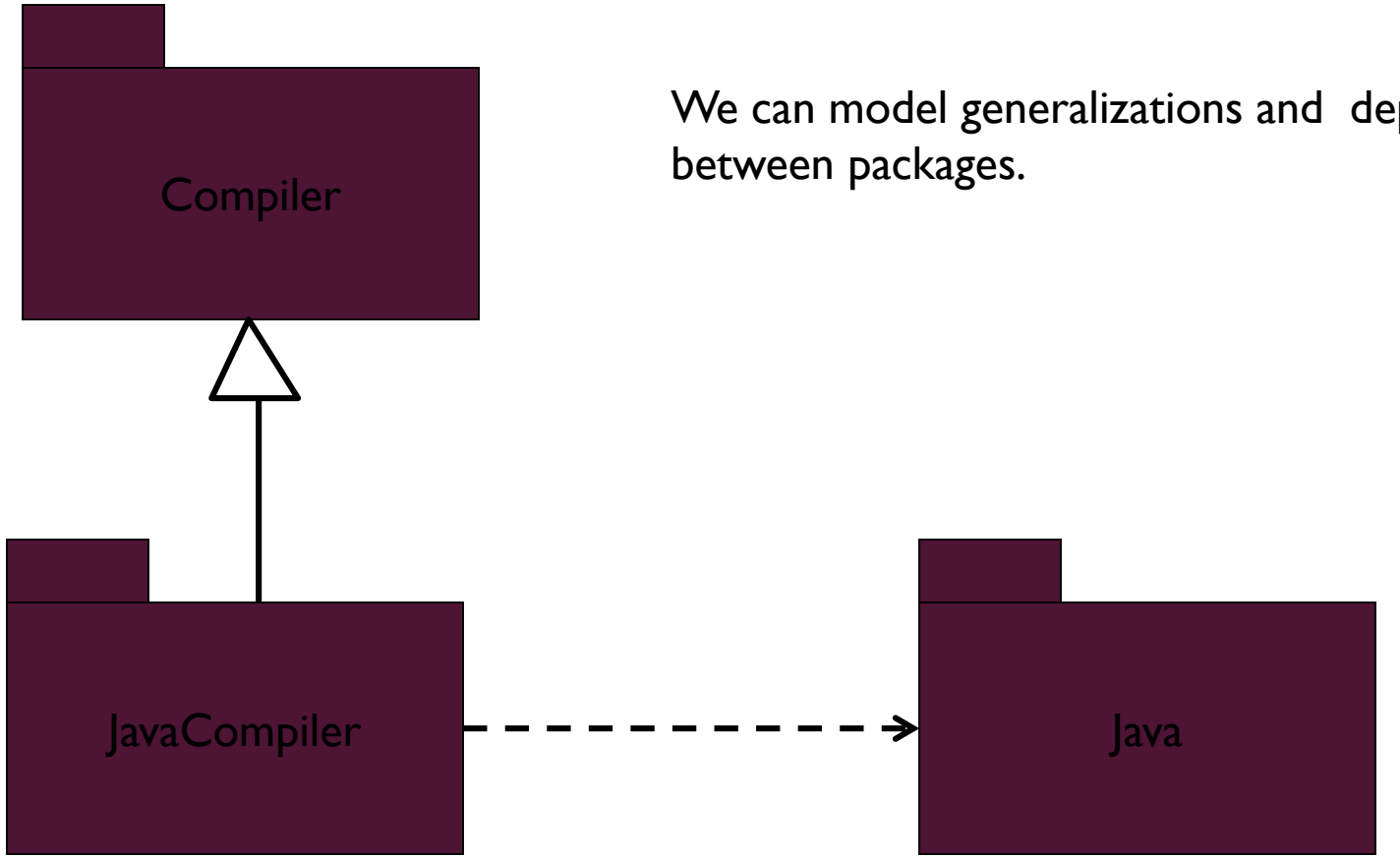


A *package* is a container-like element for organizing other elements into groups.

A package can contain classes and other packages.

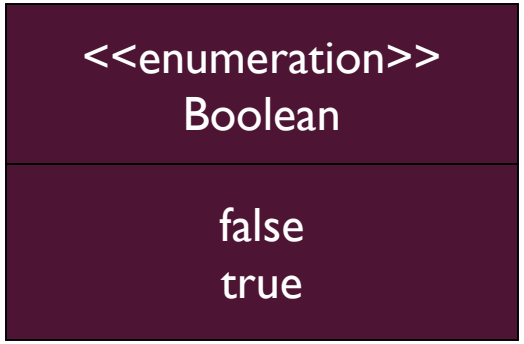
Packages can be used to provide controlled access between classes in different packages.

PACKAGES (CONT'D)



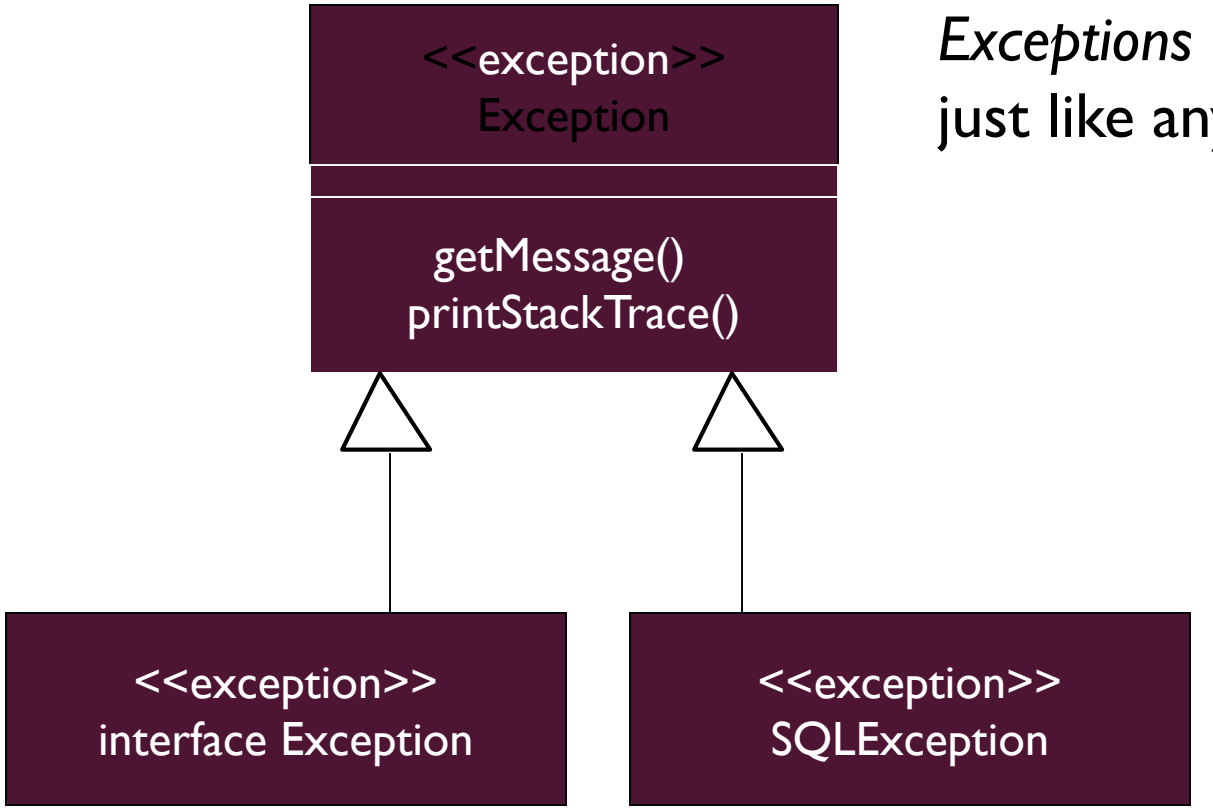
We can model generalizations and dependencies between packages.

ENUMERATION



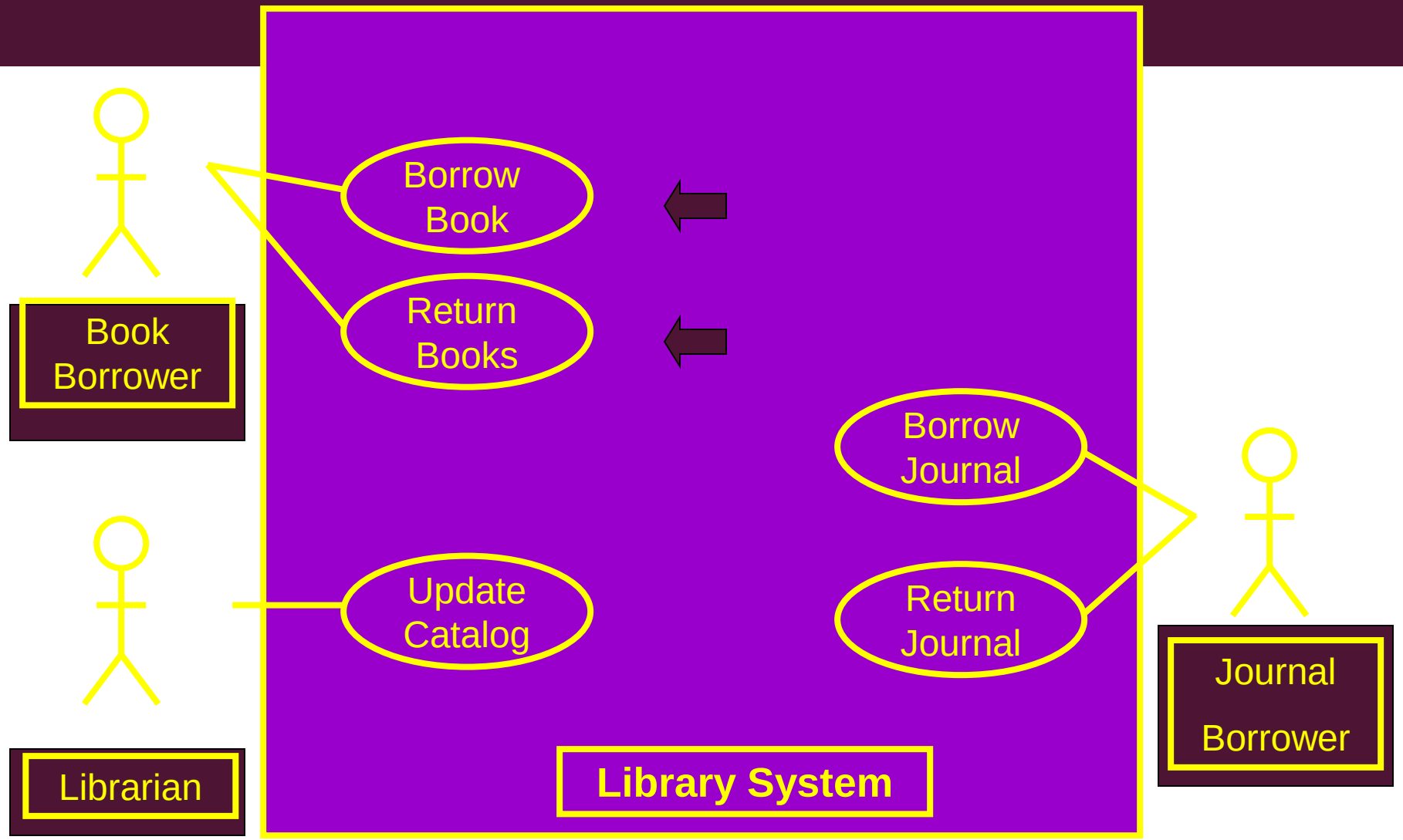
An *enumeration* is a user-defined data type that consists of a name and an ordered list of enumeration literals.

EXCEPTIONS



Exceptions can be modeled just like any other class.

USE DIAGRAM FOR A LIBRARY SYSTEM



EXERCISE

- Draw a class diagram of a campus library management system. Attributes of library include name, phone number. Library contains books and journals that can be added or removed form the library.
- Each book and journal has an id, name, author name and publisher.
- Library member can issue and return the book.
- Library member can be student or staff. Students can issue 4 books at a time and staff can have 8 books.
- Journals are available for staff only.

EXERCISE

- We have to develop an application that model different kinds of vehicles such as bicycles, motor bike and cars. All Vehicles have some common attributes (speed and colour) and common behavior (turnLeft, turnRight). Bicycle and MotorVehicle are both kinds of Vehicle. MotorVehicles have engines and license plates. MotorVehicles includes two types i.e. MotorBike and Car.

EXERCISE

University Team Management

- In the SDA course offered at Fast University, students make up project/ assignment teams.
- Each team has 2 or 3 members.
- Each team completes 0 to 3 assignments.
- Each student takes exactly two midterm test.
- Computer Science students have a single account on Coding Development facility , while each engineering student has an account on the Engineering facility.
- Each assignment and midterm is assigned a mark.

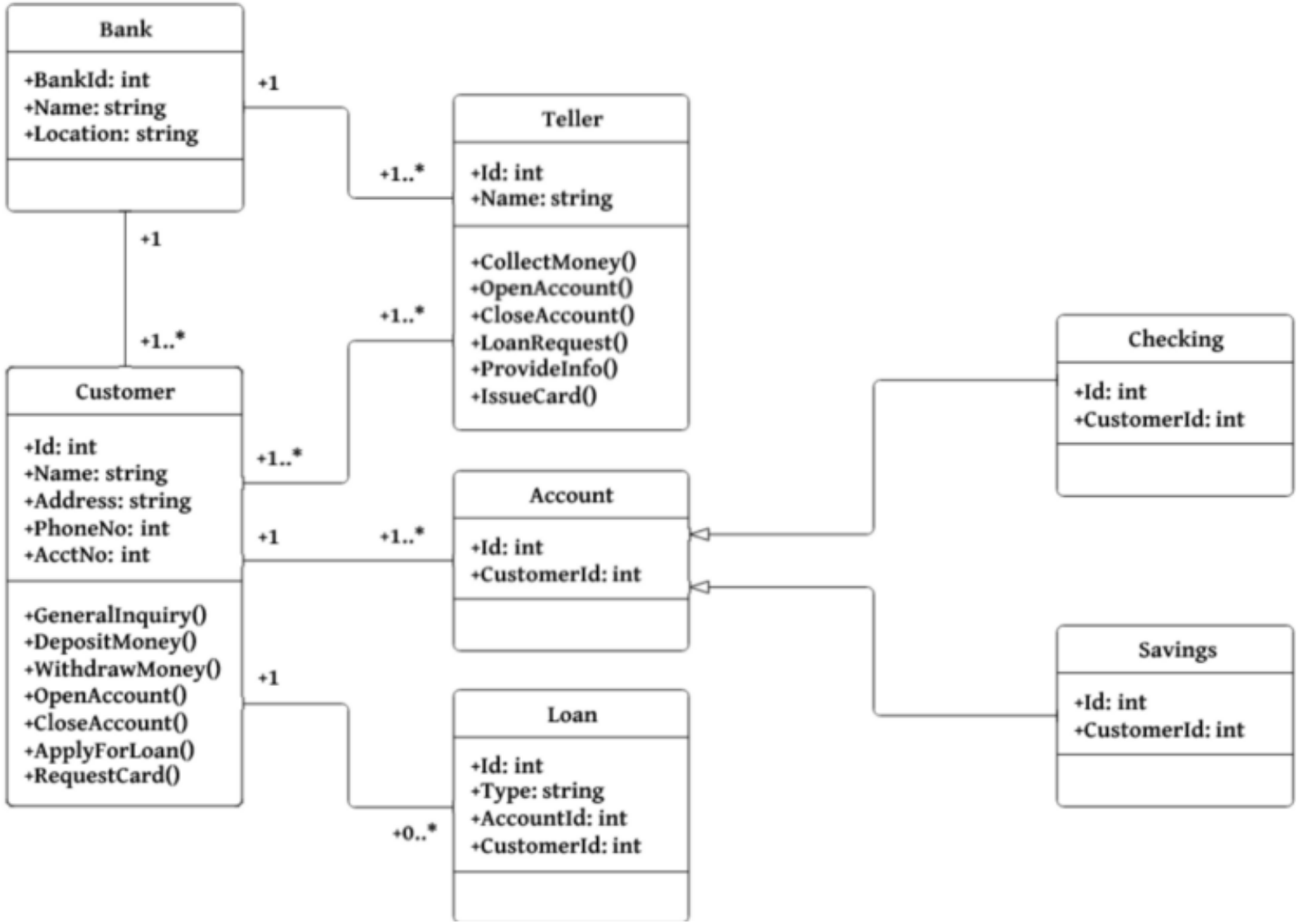
EXERCISE

University System:

- FAST university offers degrees to students.
- The university consists of faculties each of which consists of one or more departments.
- Each degree is administered by a single department.
- Each student is studying towards a single degree.
- Each degree requires one to 20 courses.
- A student enrolls in 1-5 courses (per term).
- A course cab be either graduate or undergraduate, but not both.
- Likewise, students are graduates or undergraduates but not both.

EXERCISE

- We have to develop a banking system application which provides many services to the customers like opening and closing accounts, balance enquiry, deposit money, cash withdrawal, and taking cards. Customer can open two types of accounts i.e. saving and current account. Bank also has an ATM machine which provides the services related to balance. Customer can take loan from the bank against his/her account. One customer can take only one loan at a time.


Bank
+BankId: int
+Name: string
+Location: string
+1
+1..*
Teller
+Id: int
+Name: string
+CollectMoney()
+OpenAccount()
+CloseAccount()
+LoanRequest()
+ProvideInfo()
+IssueCard()
+1
+1..*
+1..*
Customer
+Id: int
+Name: string
+Address: string
+PhoneNo: int
+AcctNo: int
+GeneralInquiry()
+DepositMoney()
+WithdrawMoney()
+OpenAccount()
+CloseAccount()
+ApplyForLoan()
+RequestCard()
+1..*
+1
+1..*
Account
+Id: int
+CustomerId: int
+1
+0..*
Loan
+Id: int
+Type: string
+AccountId: int
+CustomerId: int
Checking
+Id: int
+CustomerId: int
Savings
+Id: int
+CustomerId: int

- We want to develop a car rental system. When any customer/user wants to rent a car, he/she must be registered with the system. The system must also create the list of cars according to their category (i.e. Gomini, Go, Goplus). When a user requests booking of a car of certain category, start day of rental is the current day or a day after the current day; end day of rental lies after the start day. Customer can also cancel his or her booking. When the customer requests a pickup, a suitable car must be found among the currently available cars. When the trip is completed, booking closes and car must be returned.

That is all