Class Diagram

Lecture # 16,17, 18
5, 7, 10 Oct

Rubab Jaffar
rubab.jaffar@nu.edu.pk

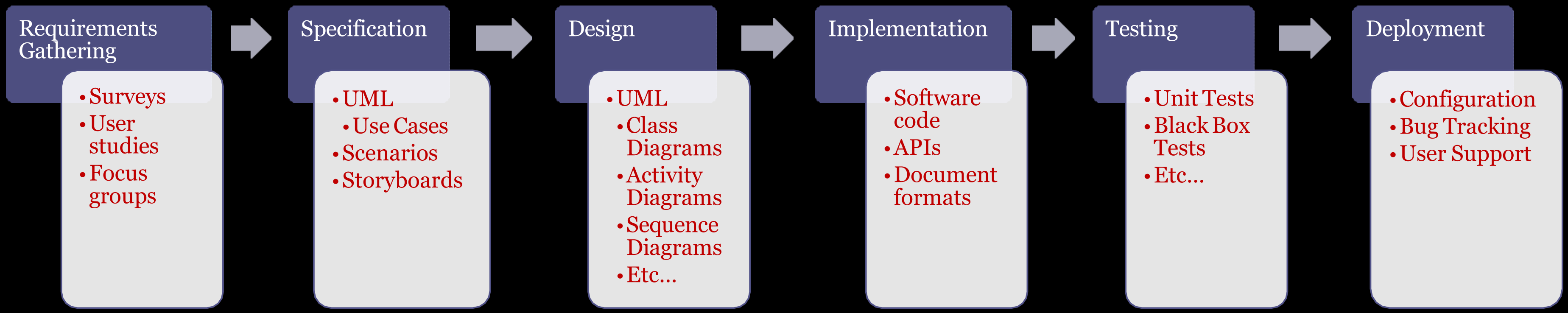
Software Design and Analysis

CS-324

Today's Outline

- Class Diagram
- Components of Class Diagram
- Relationships
- Exercises

OOAD: Big Picture



Requirements Gathering

- Surveys
- User studies
- Focus groups

Specification

- UML
  - Use Cases
- Scenarios
- Storyboards

Design

- UML
  - Class Diagrams
  - Activity Diagrams
  - Sequence Diagrams
  - Etc…

Implementation

- Software code
- APIs
- Document formats

Testing

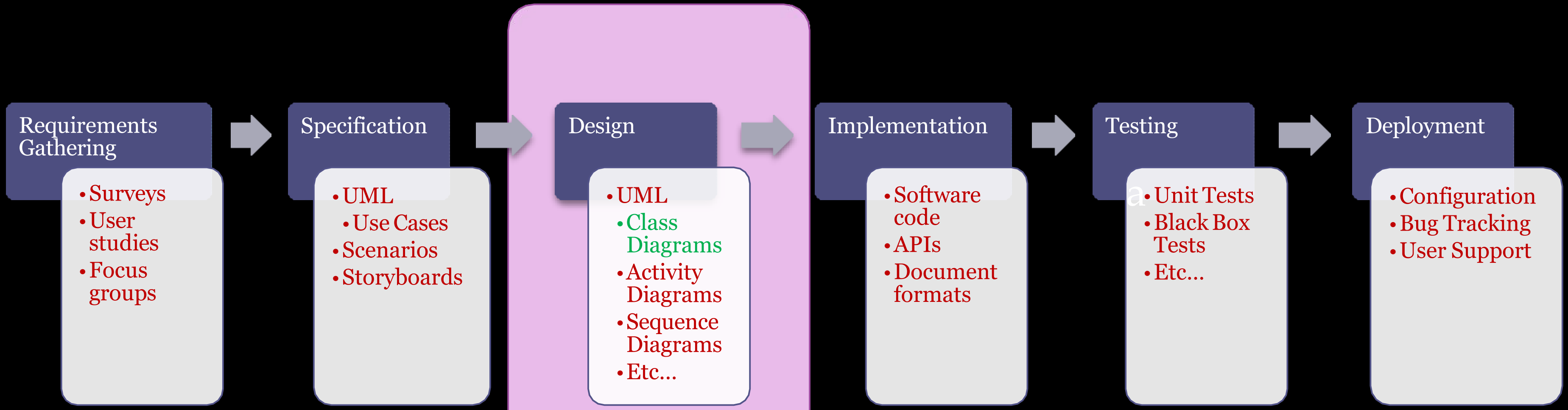
- Unit Tests
- Black Box Tests
- Etc…

Deployment

- Configuration
- Bug Tracking
- User Support

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

OOAD: Big Picture



Requirements Gathering
- Surveys
- User studies
- Focus groups

Specification
- UML
  - Use Cases
- Scenarios
- Storyboards

Design
- UML
  - Class Diagrams
  - Activity Diagrams
  - Sequence Diagrams
  - Etc…

Implementation
- Software code
- APIs
- Document formats

Testing
- Unit Tests
- Black Box Tests
- Etc…

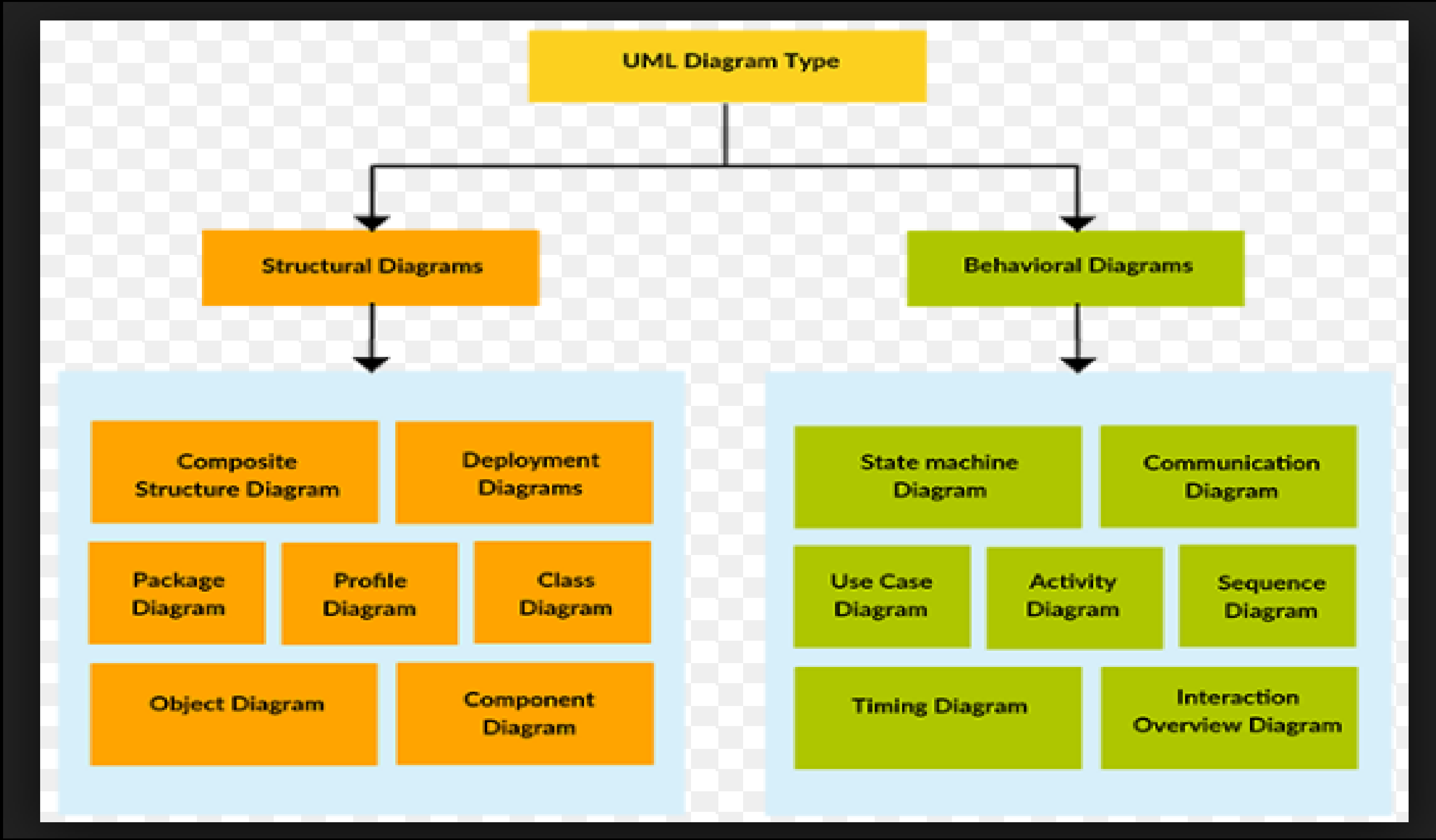
Deployment
- Configuration
- Bug Tracking
- User Support

Designing a solution

Specifying Data, Objects and Relationships

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

Basics of UML Class Diagrams

- A software application is comprised of classes and a diagram depicting the relationship between each of these classes would be the class diagram.
- A class diagram is a pictorial representation of the detailed system design
- The purpose of class diagram is to model the static view of an application.
- Class diagrams are the only diagrams which can be directly mapped with object-oriented languages.
- Widely used at the time of construction.

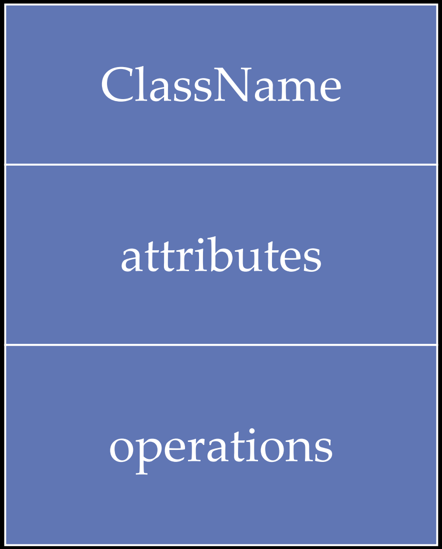
Relationship between Class Diagram and Use Cases

- 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



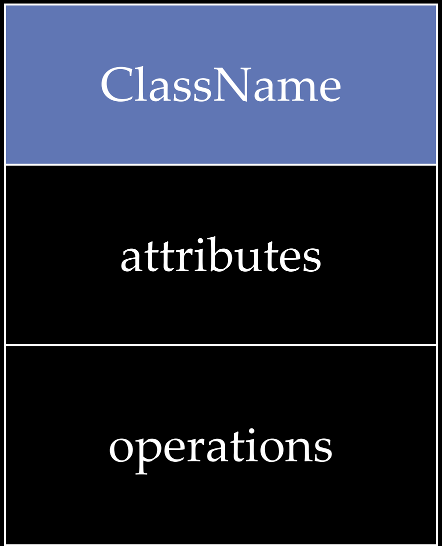
ClassName

attributes

operations

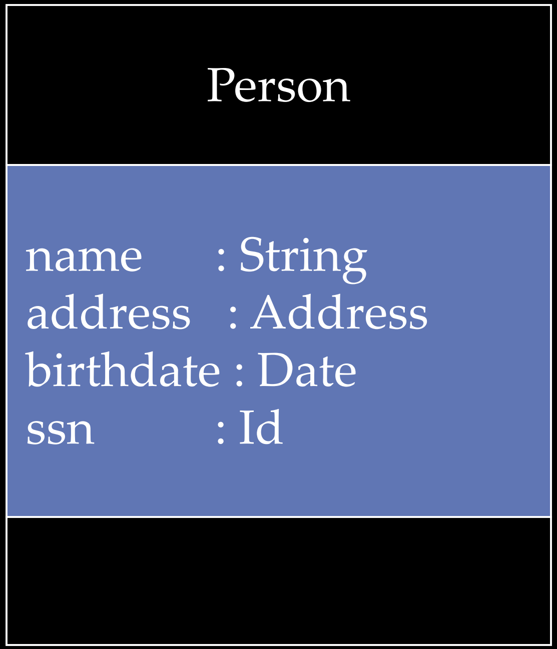
- A class represents an entity of a given system that provides an encapsulated implementation of certain functionality of a given entity. These are exposed by the class to other classes as methods. Apart from functionality, a class also has properties that reflect unique features of a class. The properties 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



Person

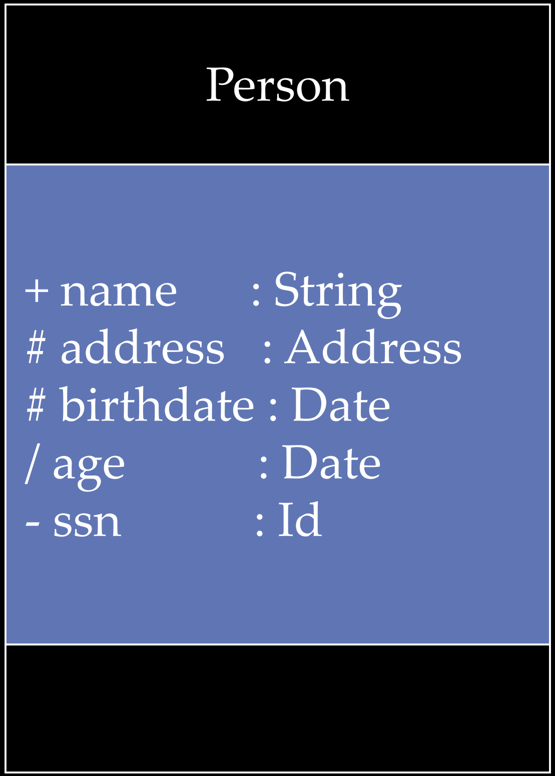
name : String
address : Address
birthdate : Date
ssn : Id

- 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)



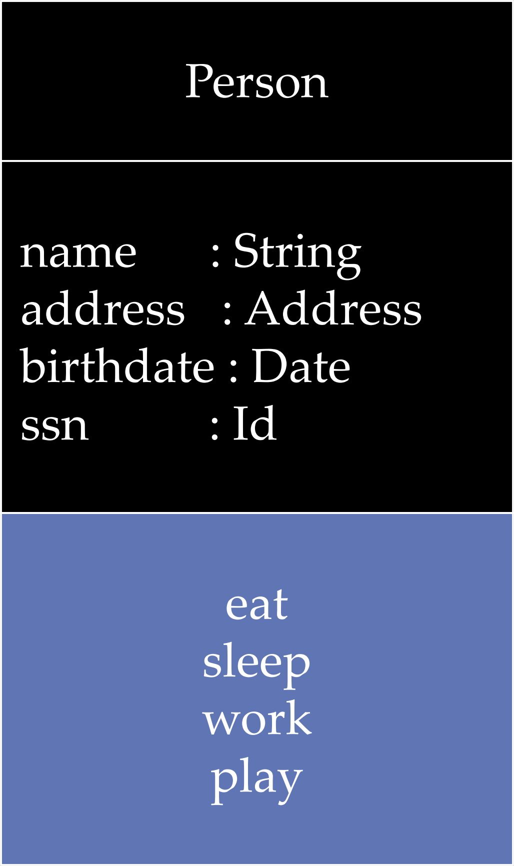
Person

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

Attributes can be:

+ public
# protected
- private
~ package
/ derived

Class Operations



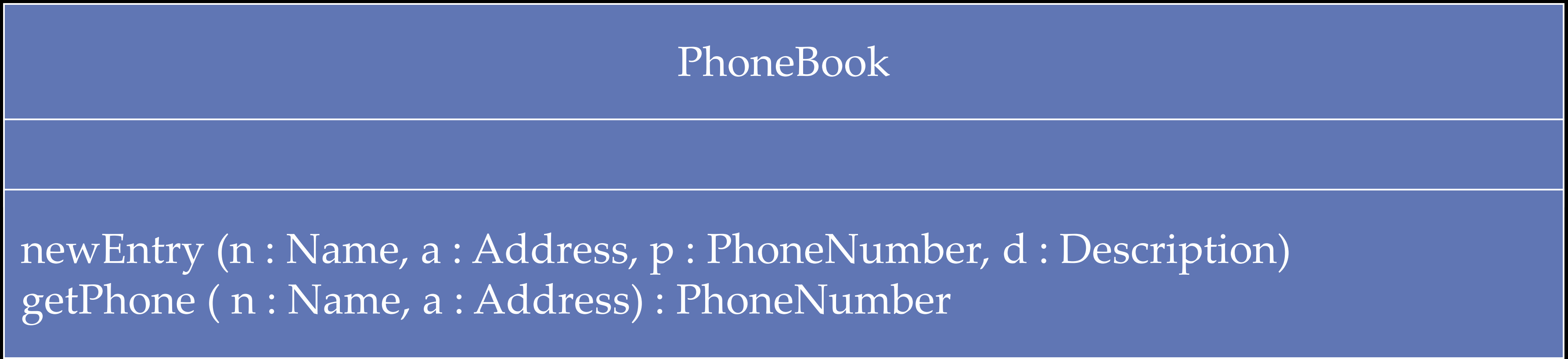
Person

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

eat
sleep
work
play

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

Class Operations (Cont'd)



PhoneBook

newEntry (n : Name, a : Address, p : PhoneNumber, d : Description)
getPhone (n : Name, a : Address) : PhoneNumber

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

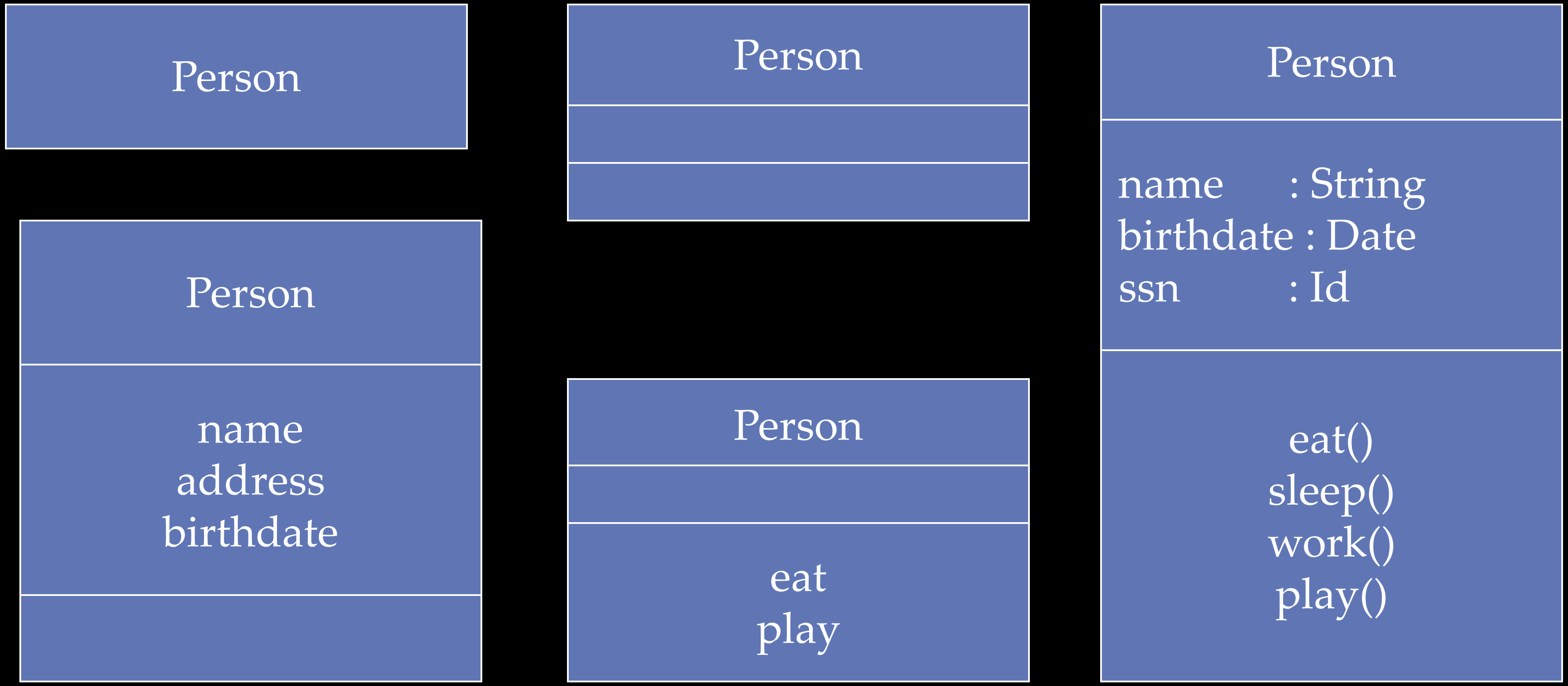
flightNumber : Integer

The full UML syntax for operations is

visibility name (parameter-list) : return-type

Depicting Classes

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



Person

Person

Person

name : String
birthdate : Date
ssn : Id

eat()
sleep()
work()
play()

Person

name
address
birthdate

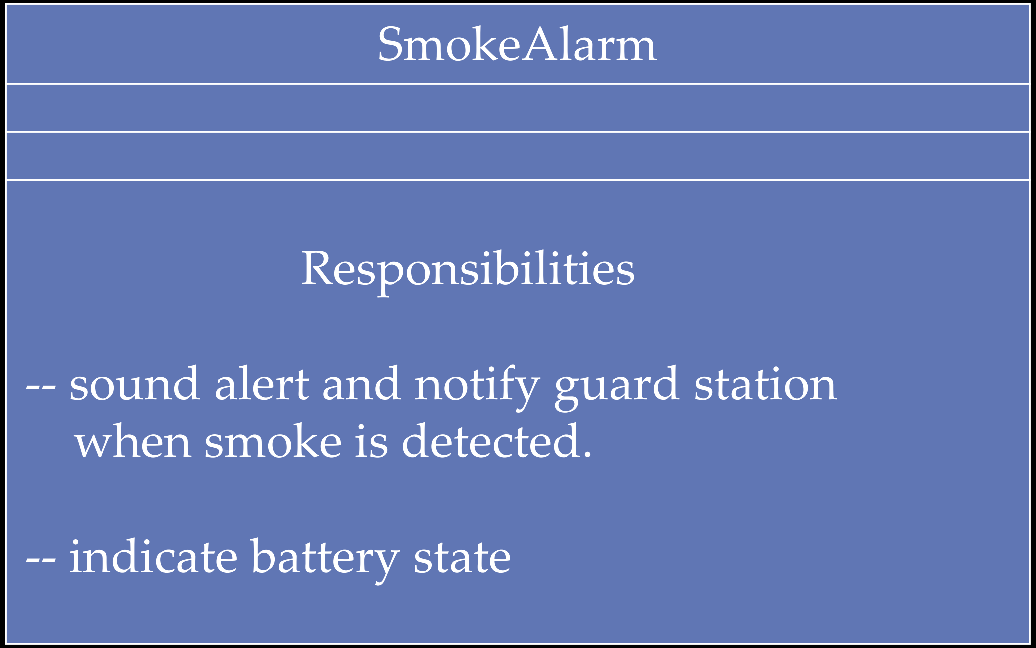
Person

eat
play

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.



SmokeAlarm

Responsibilities

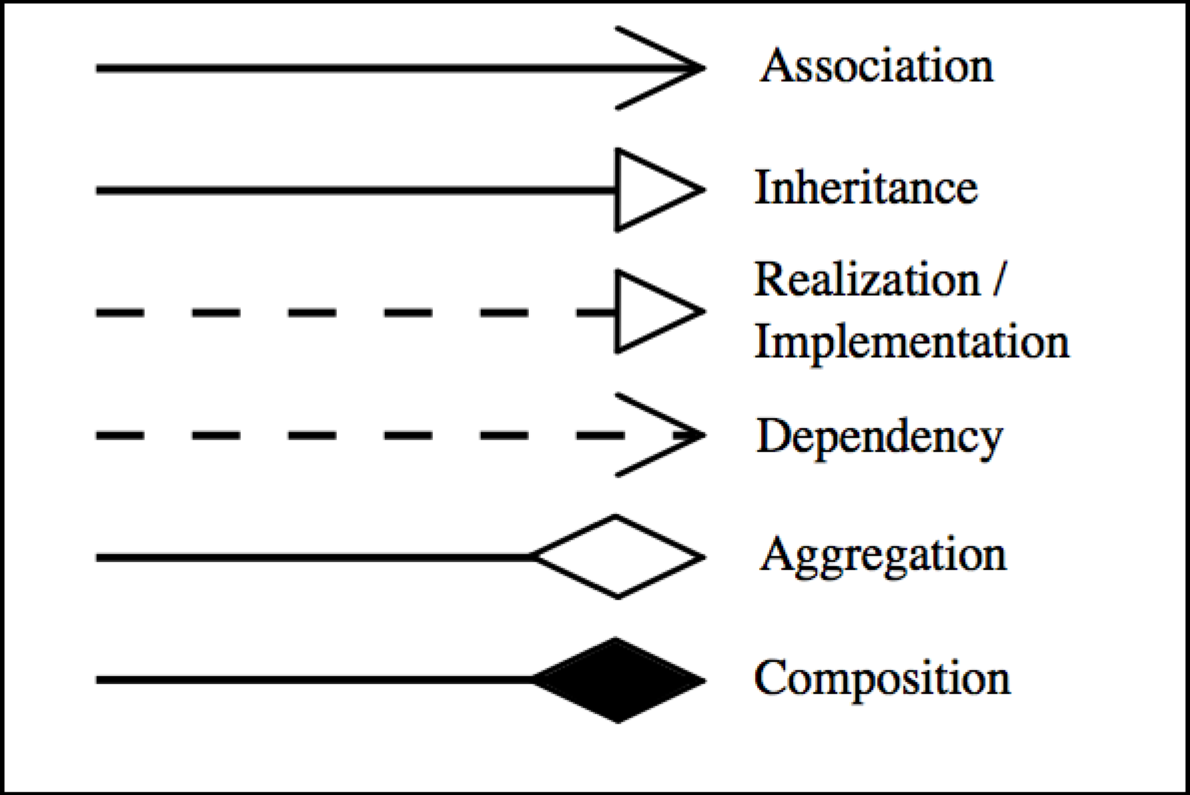
-- sound alert and notify guard station
when smoke is detected.

-- indicate battery state

Relationships

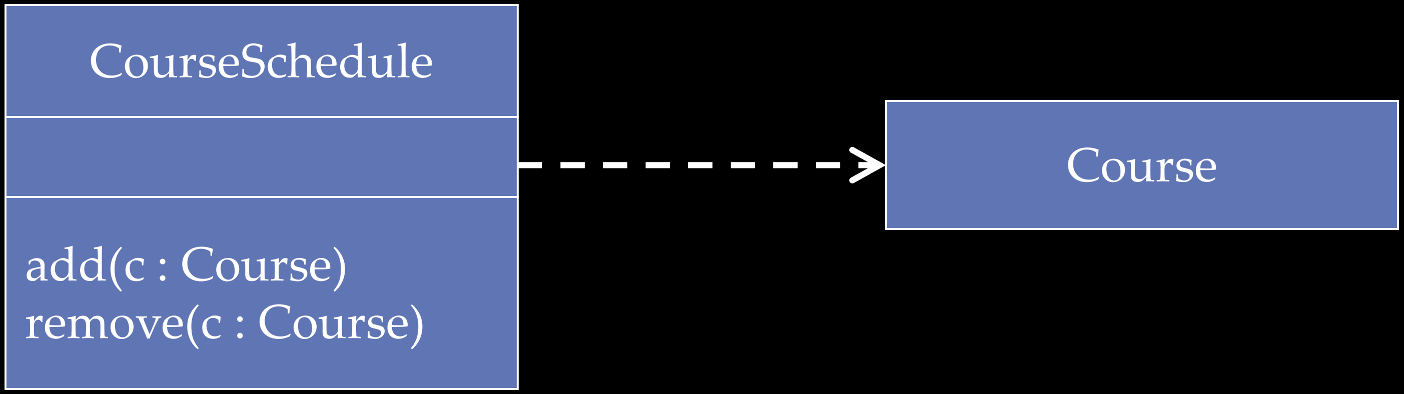
In UML, object interconnections (logical or physical), are modeled as relationships.

There are six kinds of relationships in UML:



Dependency 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*.

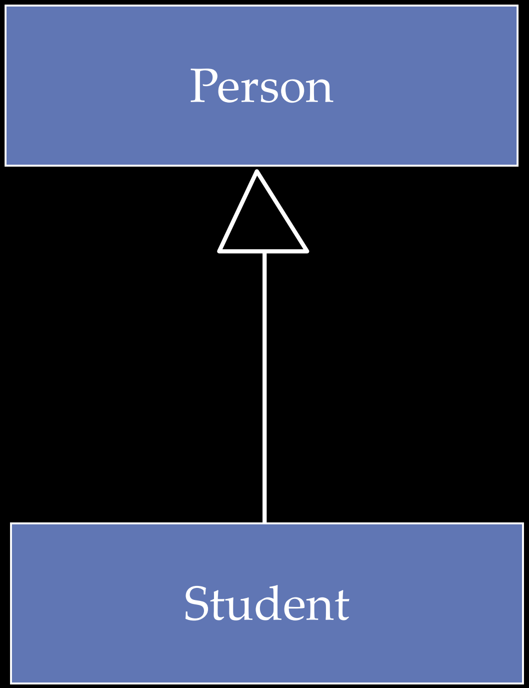


Dependency

- 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



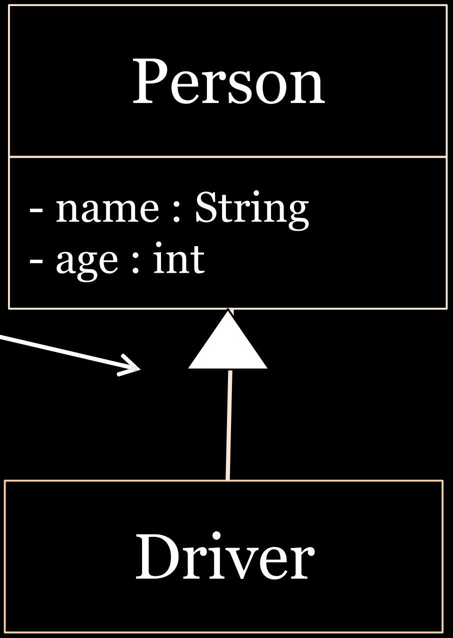
A *generalization* connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

```
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



Person

- name : String
- age : int

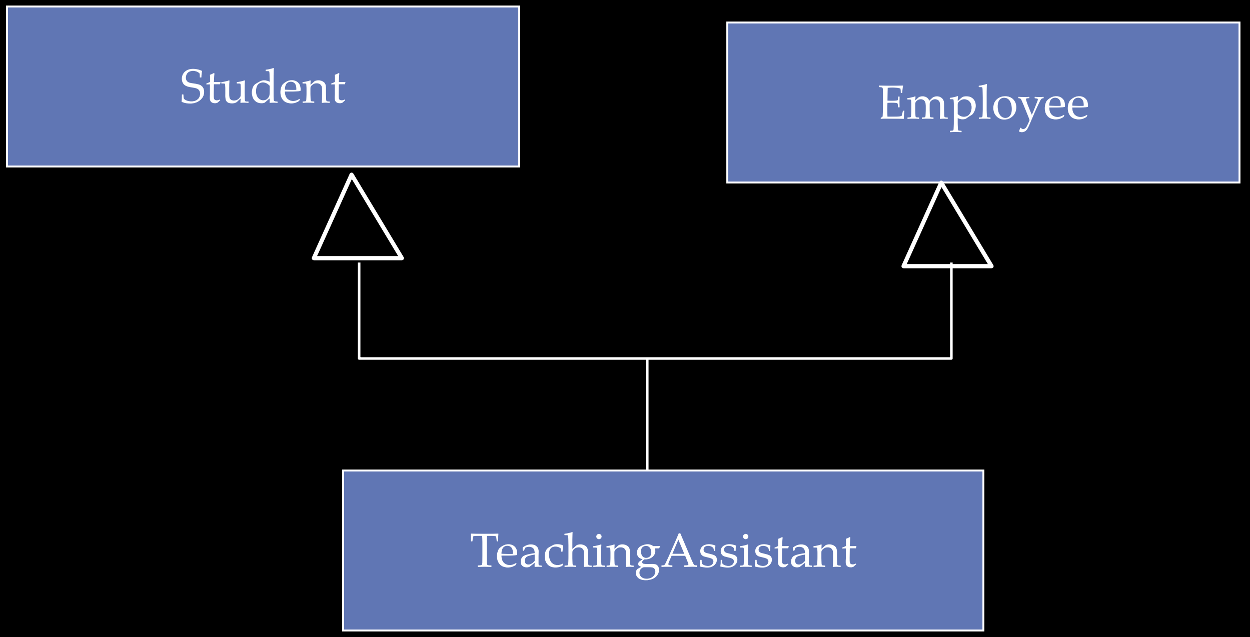
Driver

```
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)

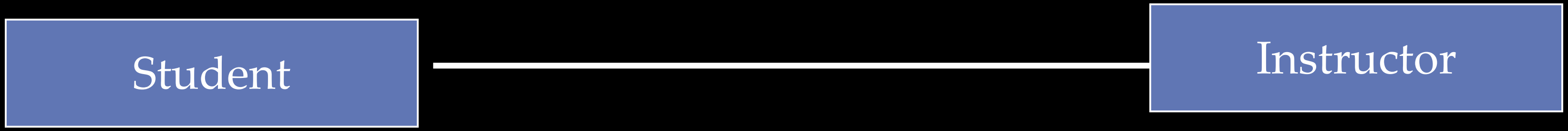
Generalization Relationships (Cont'd)

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



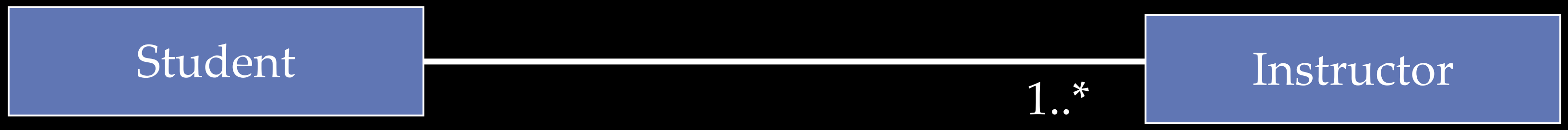
Multiplicity

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

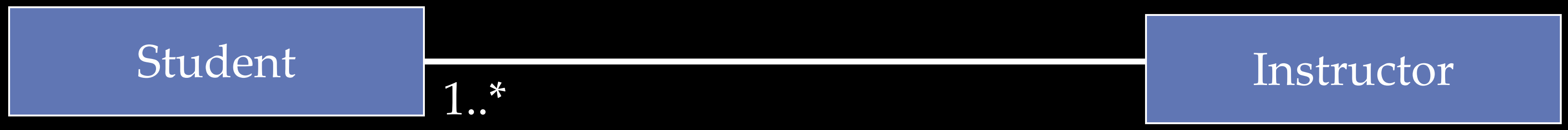
- Multiplicity (how many are used) □
- * ⇒ 0, 1, or more
- 1 ⇒ 1 exactly □
- 2..4 ⇒ between 2 and 4,
- inclusive □3..* ⇒ 3 or more

Association Relationships (Cont'd)

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

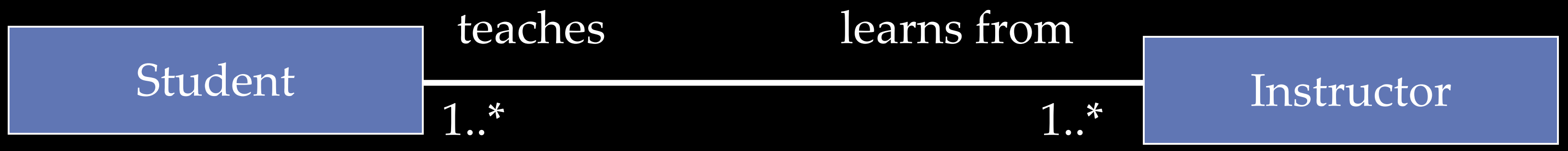


The example indicates that every *Instructor* has one or more *Students*:

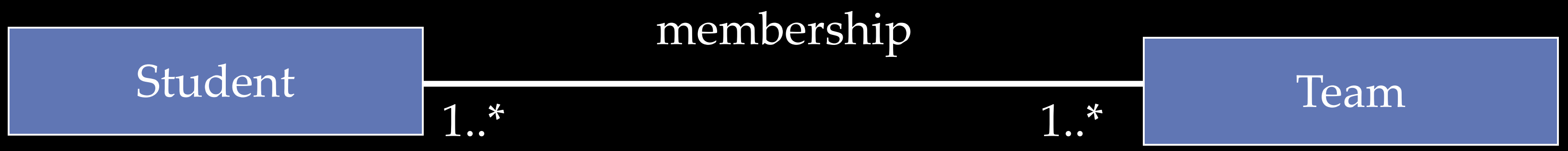


Association Relationships (Cont'd)

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



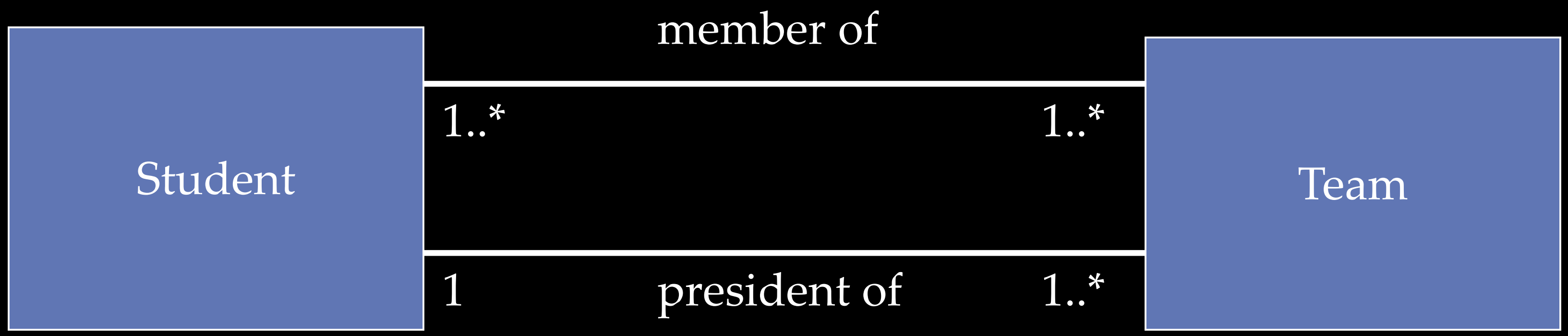
We can also name the association.



Association Relationships

(Cont'd)

We can specify dual associations.



Kinds of Association

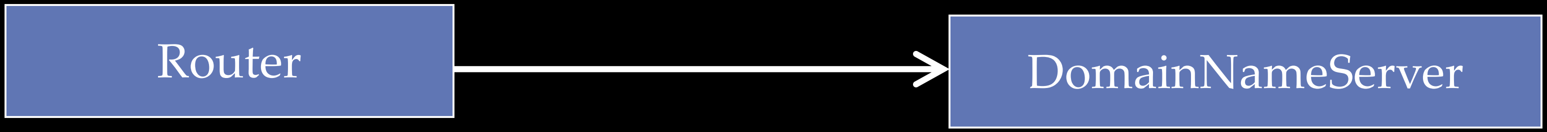
- Object Association
  - Simple Association: Is simply called as “association”
  - Composition
  - Aggregation

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*.



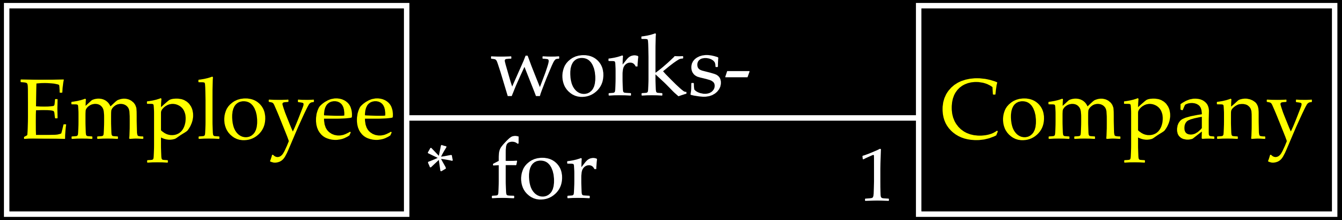
One way Association-Person-Address

```
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

*

1

Company

- Employee works for company
- Company employs employees

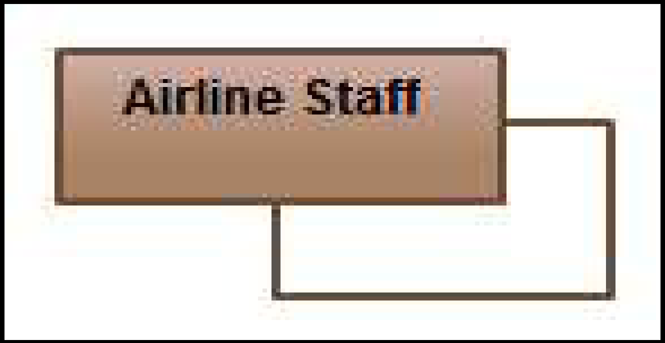
Two way Association-Contractor-Project

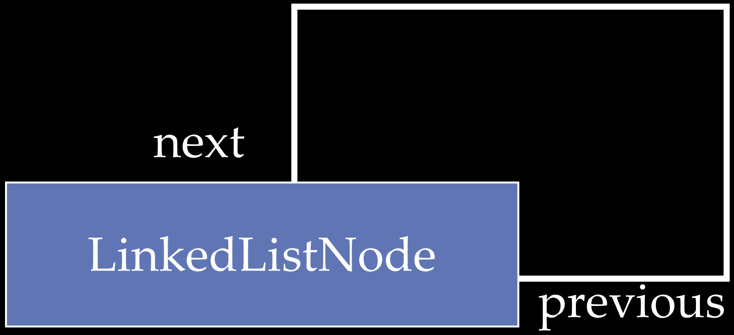
```
class Contractor
{
private:
string Name;
Project *MyProject;
…
};
```

```
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

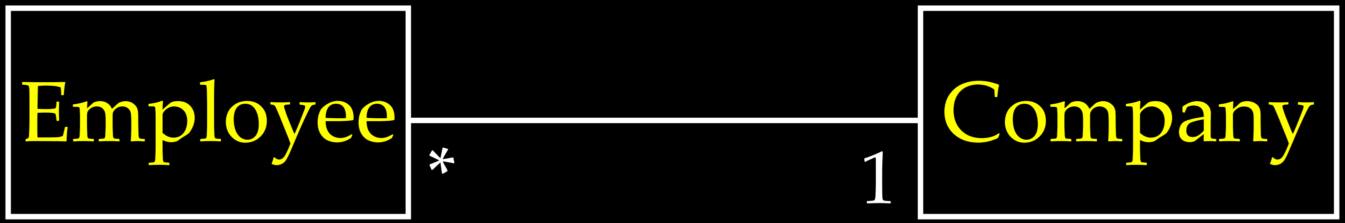
```
class Course
{
private:
    std::string m_name;
    Course *m_prerequisite;

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

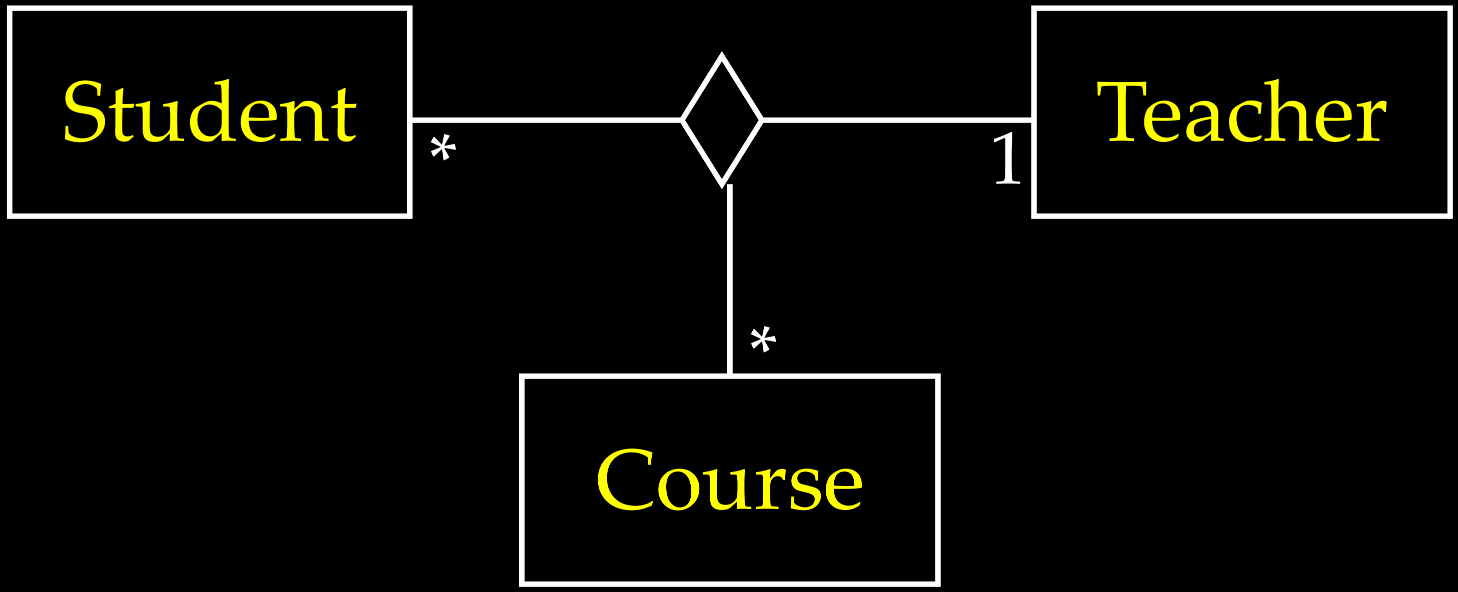
- Denoted by a line, or an arrow between the associated objects



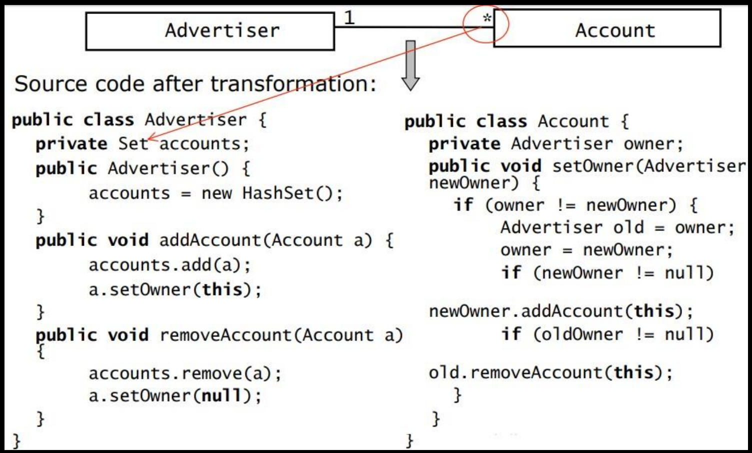
- Association "works-for" associates objects of exactly two classes

Ternary Association

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



Bidirectional Association

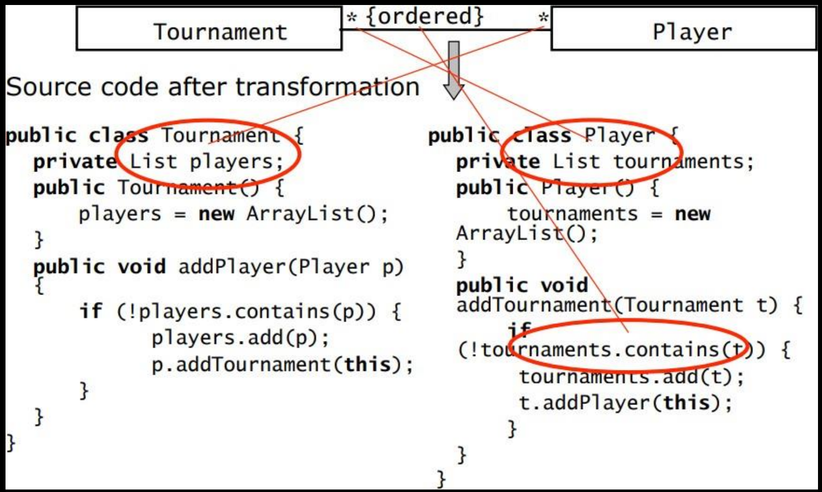


Source code after transformation:

```
public class Advertiser {
  private Set accounts;
  public Advertiser() {
       accounts = new HashSet();
  }
  public void addAccount(Account a) {
       accounts.add(a);
       a.setOwner(this);
  }
  public void removeAccount(Account a)
  {
       accounts.remove(a);
       a.setOwner(null);
  }
}
```

```
public class Account {
  private Advertiser owner;
  public void setOwner(Advertiser
  newOwner) {
    if (owner != newOwner) {
         Advertiser old = owner;
         owner = newOwner;
         if (newOwner != null)

  newOwner.addAccount(this);
         if (oldOwner != null)

  old.removeAccount(this);
    }
  }
}
```

One to many

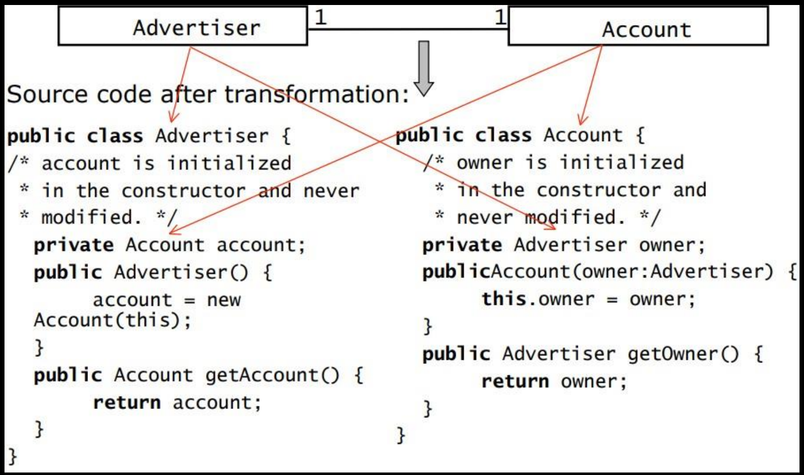


Source code after transformation

```
public class Tournament {
  private List players;
  public Tournament() {
      players = new ArrayList();
  }
  public void addPlayer(Player p)
  {
      if (!players.contains(p)) {
            players.add(p);
            p.addTournament(this);
      }
  }
}
```

```
public class Player {
  private List tournaments;
  public Player() {
      tournaments = new
  ArrayList();
  }
  public void
  addTournament(Tournament t) {
        if
  (!tournaments.contains(t)) {
         tournaments.add(t);
         t.addPlayer(this);
        }
  }
}
```

many to many



Source code after transformation:

```
public class Advertiser {
/* account is initialized
 * in the constructor and never
 * modified. */
  private Account account;
  public Advertiser() {
       account = new
  Account(this);
  }
  public Account getAccount() {
       return account;
  }
}
```

```
public class Account {
/* owner is initialized
 * in the constructor and
 * never modified. */
  private Advertiser owner;
  publicAccount(owner:Advertiser) {
       this.owner = owner;
  }
  public Advertiser getOwner() {
       return owner;
  }
}
```

One to one

N-ary Association

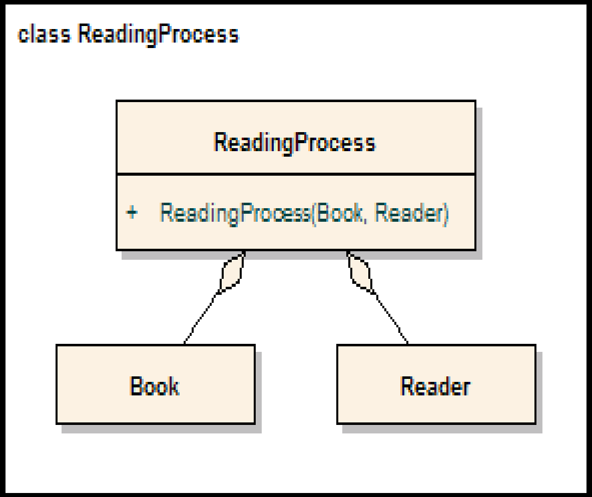
- An association between 3 or more classes
- Practical examples are very rare

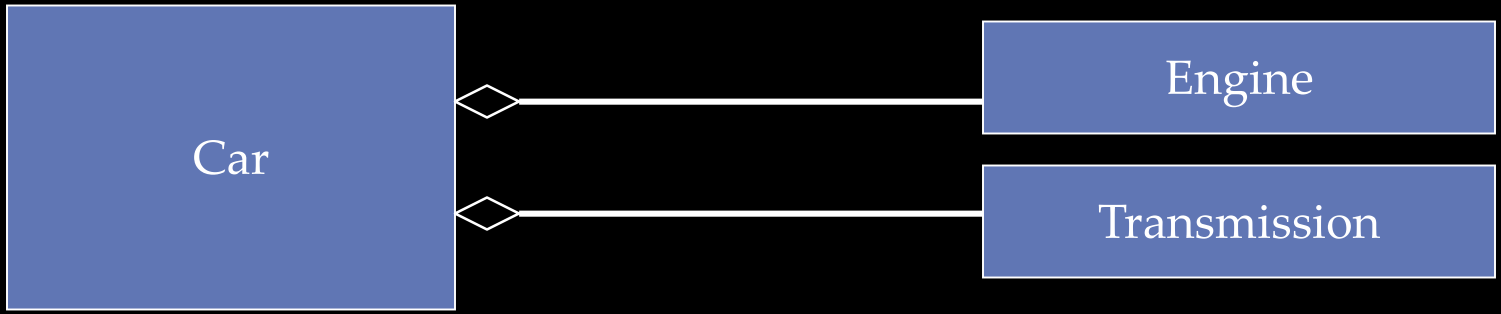
Association Relationships (Cont'd)

Aggregation:

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

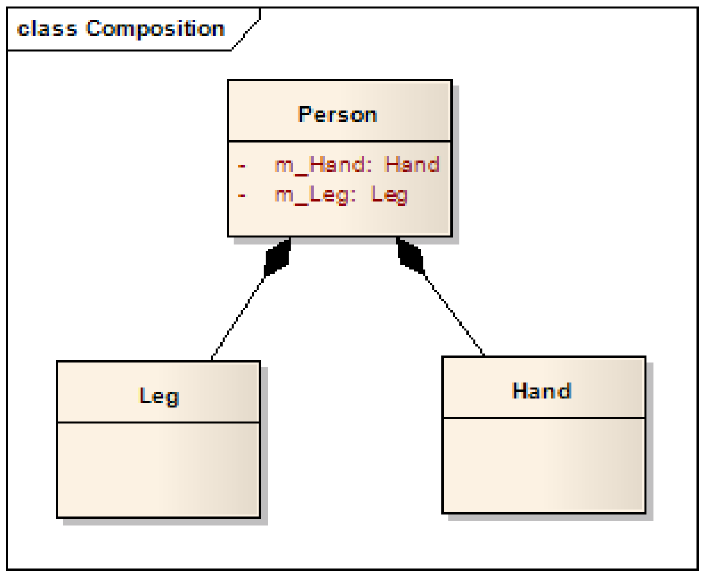
An ***aggregation*** specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.

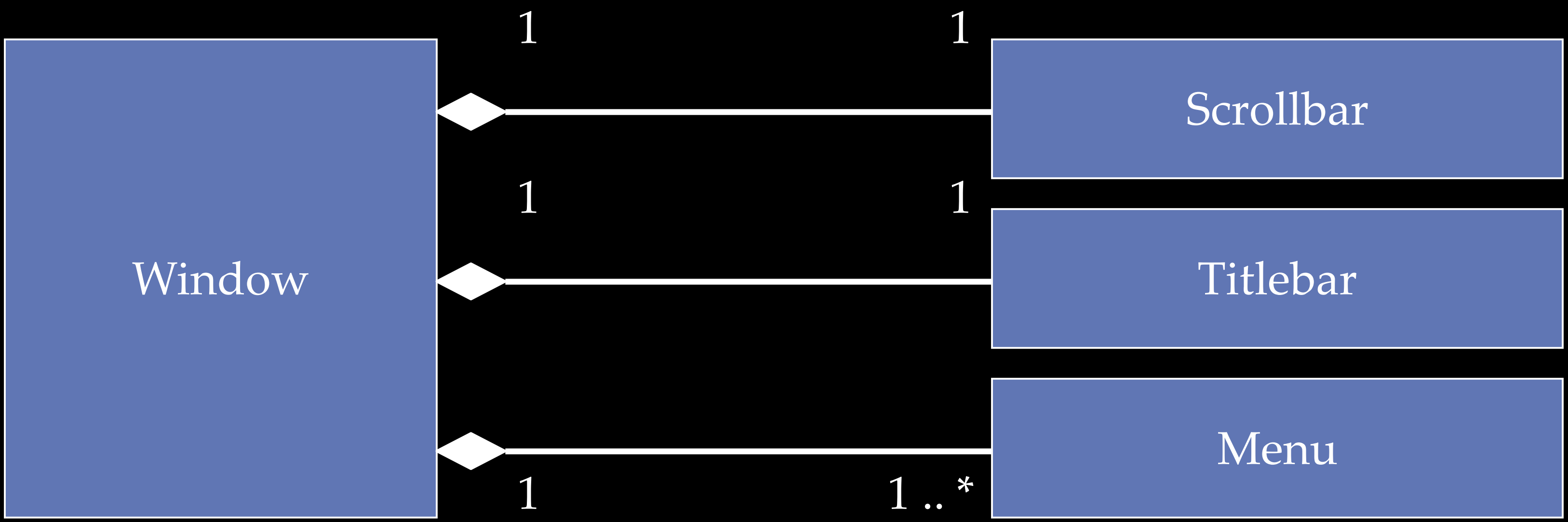




Association Relationships (Cont'd)

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.





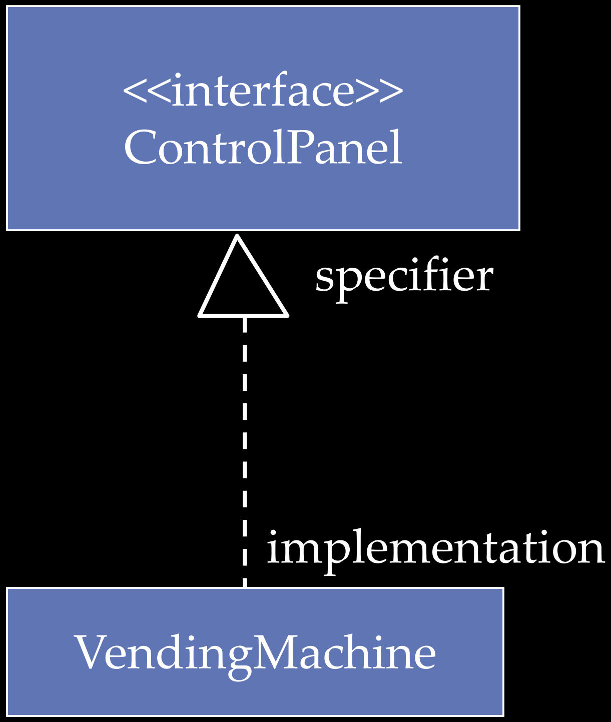
Window

1 1 Scrollbar

1 1 Titlebar

1 1 .. * Menu

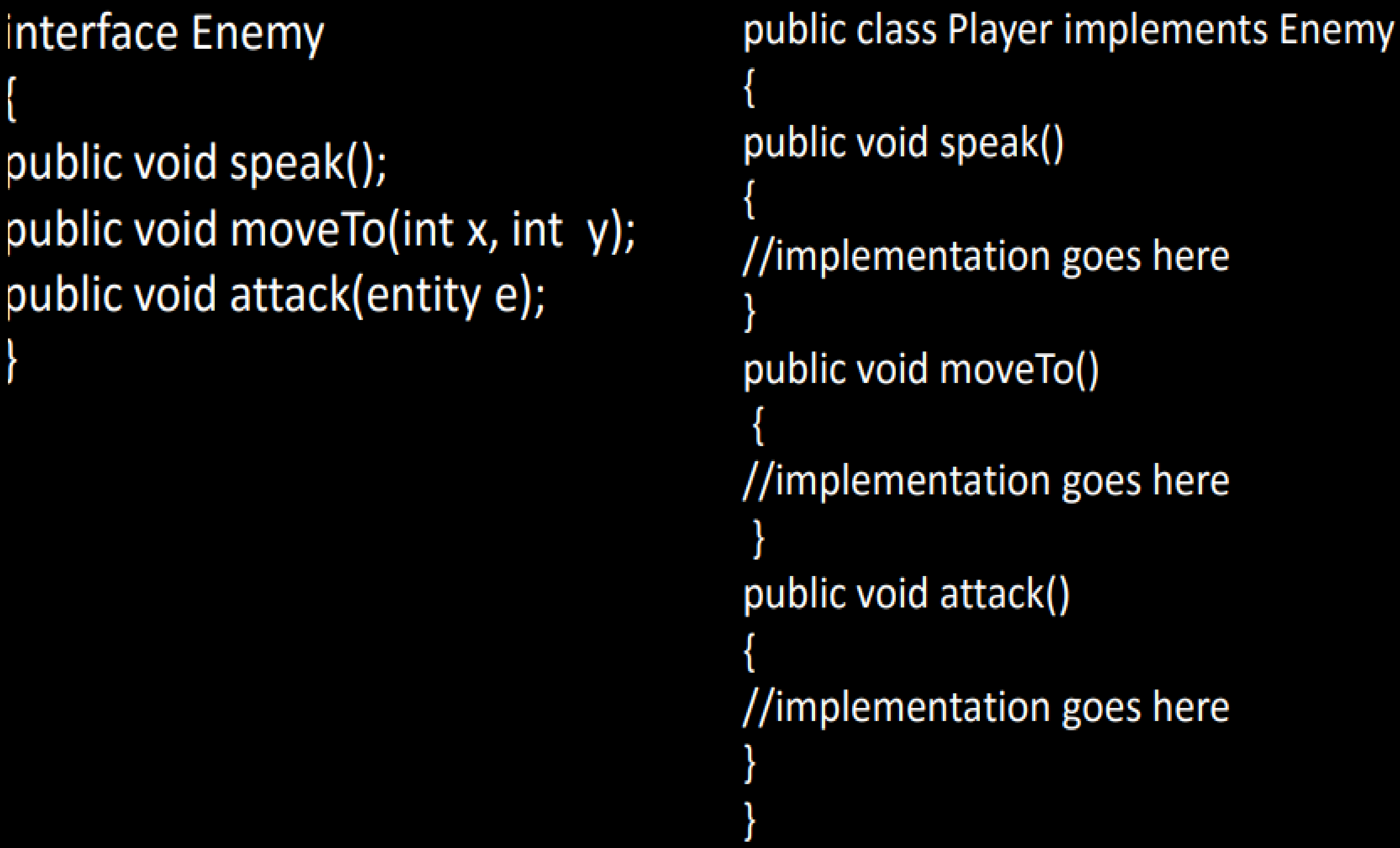
Interface Realization Relationship



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
```

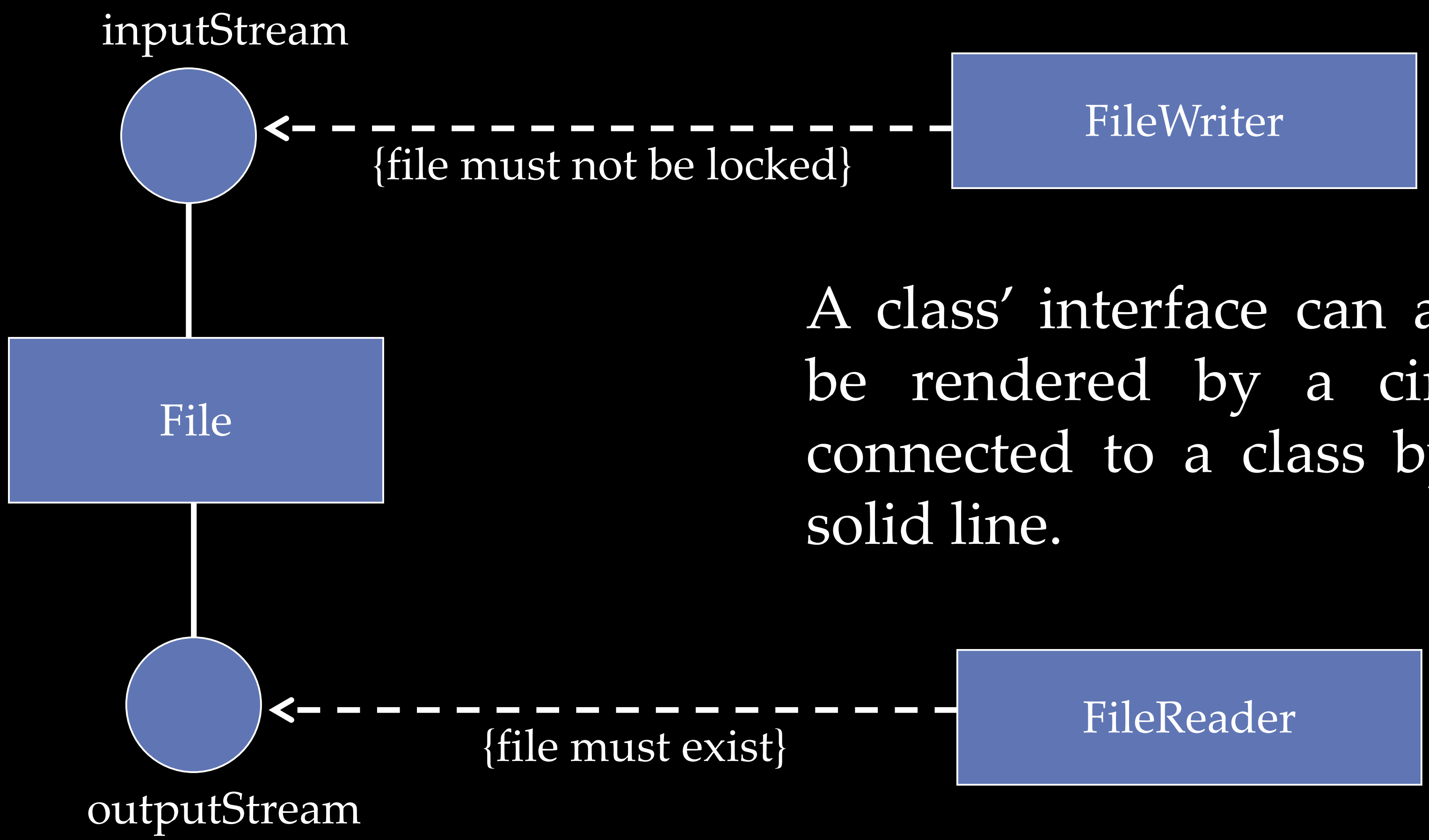
Realization/Interface Implementation



```
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
}
}
```

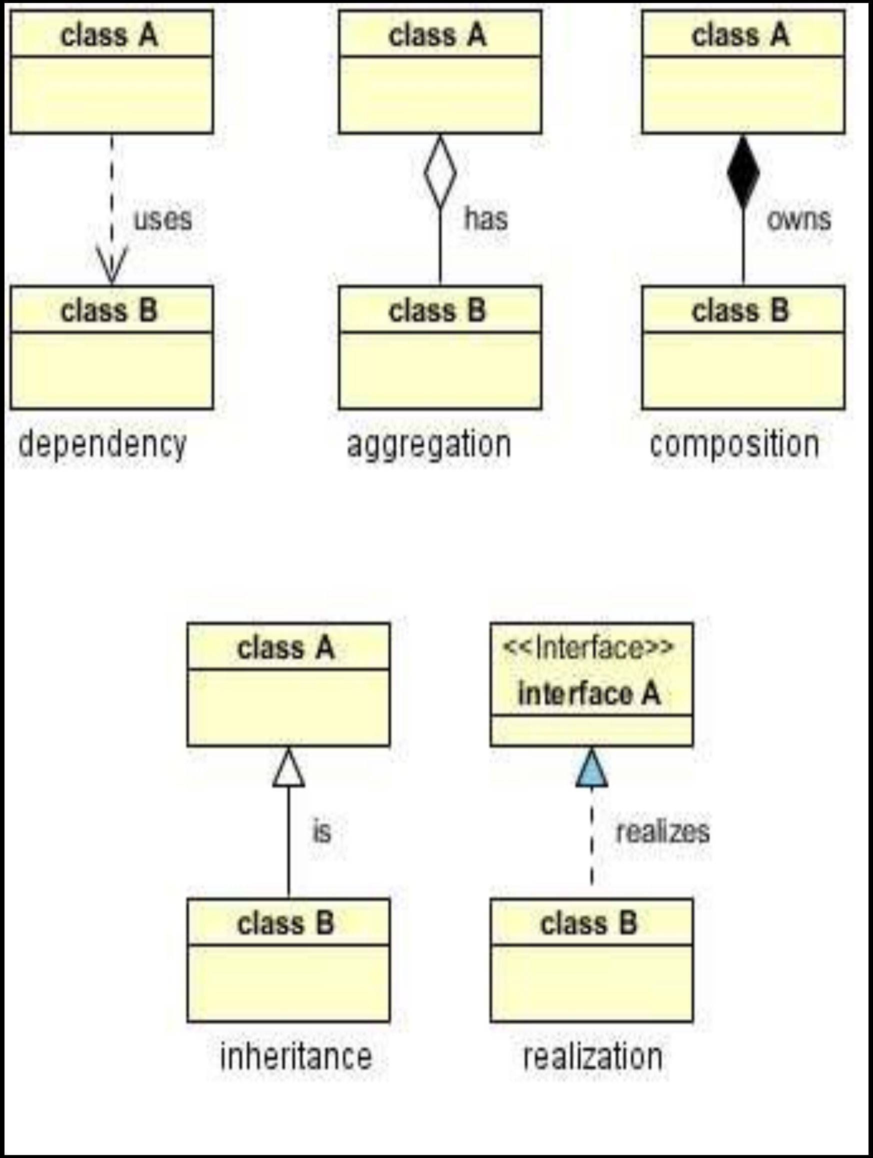
Interfaces



A class’ interface can also be rendered by a circle connected to a class by a solid line.

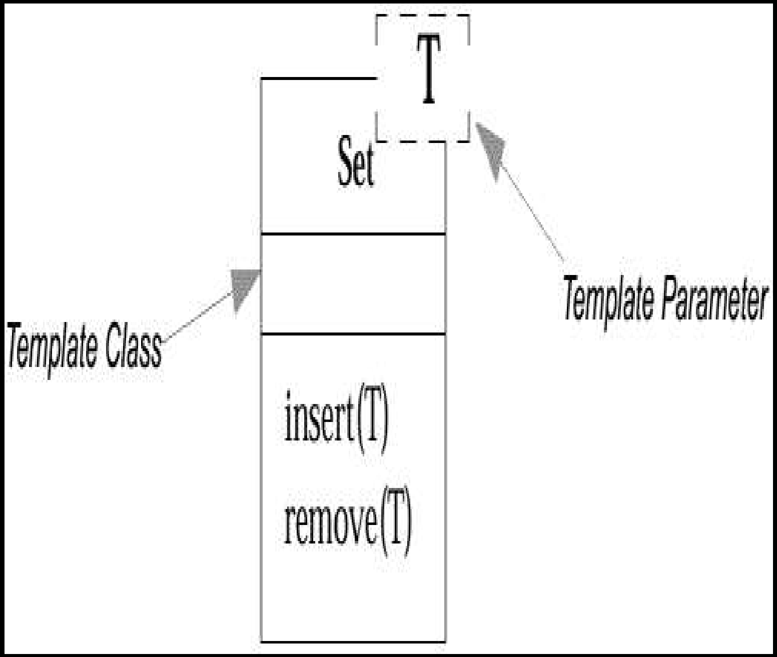
Relationships in a NutShell

- 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)



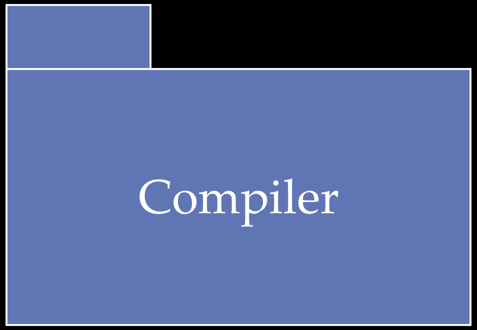
Parameterized Class

- 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

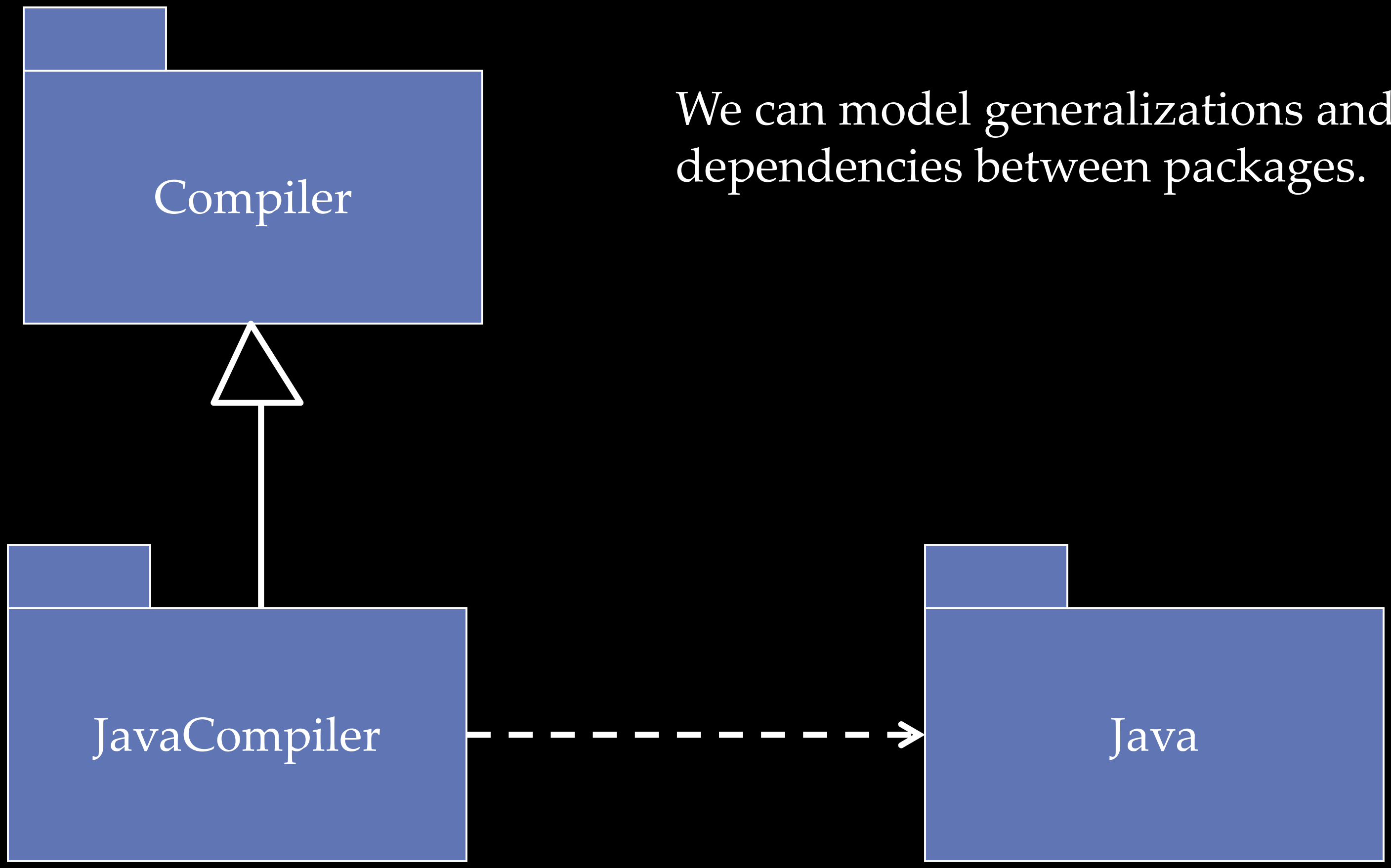


A *package* is a container-like element for organizing other elements into groups.

A package can contain classes and other packages.

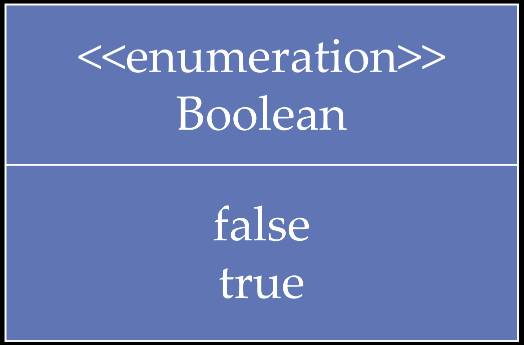
Packages can be used to provide controlled access between classes in different packages.

Packages (Cont'd)



We can model generalizations and dependencies between packages.

Enumeration

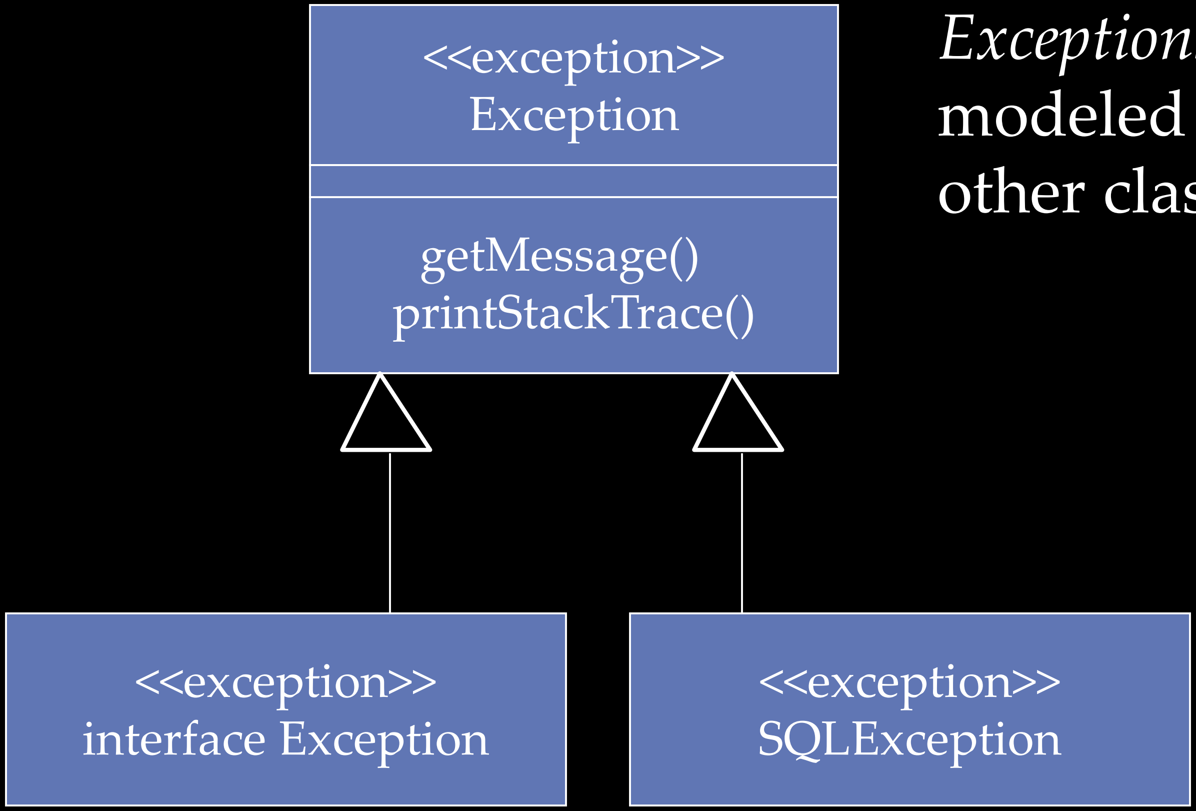


<<enumeration>>
Boolean

false
true

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.

That is all