# Class Modeling

Please read Chapter 8

**Dr. Abdelkarim Erradi**

Dept. of Computer Science & Engineering

**QU**

# Software Design

# The Process of Design

- *Design* is a **problem-solving process** whose objective is to find and describe a way:
  - **To implement the system's *functional*** *requirements*...
  - While **meeting the non-functional** *requirements*...

    And constraints such as time and budget
  - And while **adhering to general principles of *good quality***

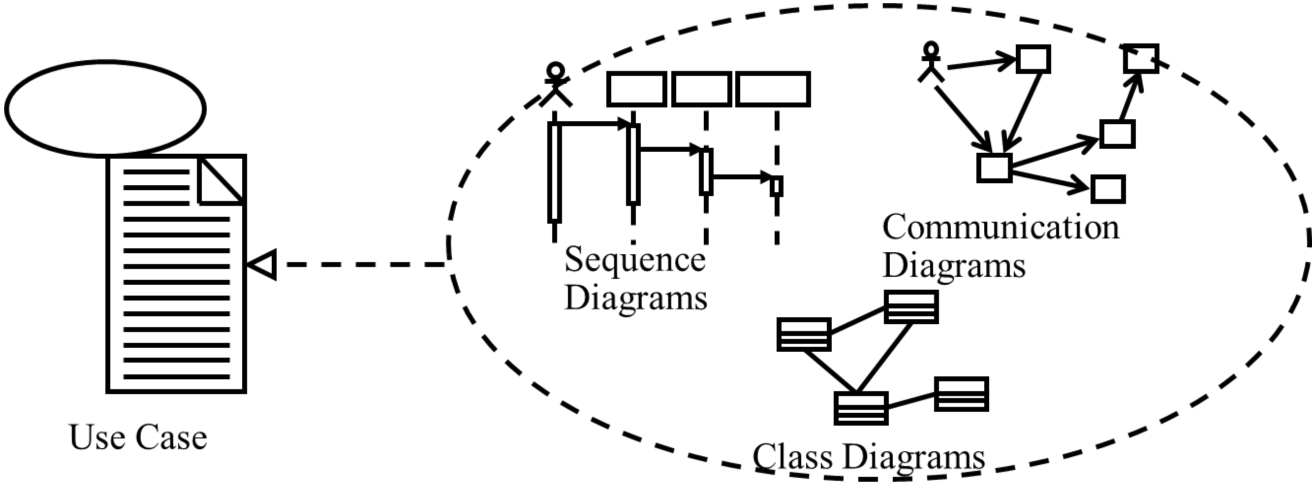
# OO Analysis vs. Design

- Object-Oriented Analysis
  - ***Domain Model:***
    - ➢ Important domain concepts or objects
    - ➢ Relationships

- Object-Oriented Design
  - Design of software objects = ***Design Model***:
    - ➢ Responsibilities
    - ➢ Collaborations
  - Apply Design patterns
  - Document the design rationale = the reasoning that went into making the decision

# Analysis and Design are Use-Case Driven

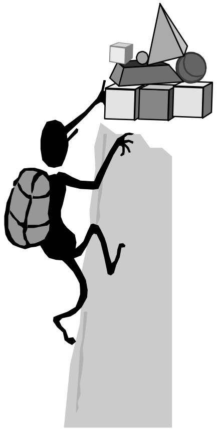
- Use cases defined for a system are the basis for the entire development process.
- Design aim to create the Use-Case Realization:
  - Allocate use-case responsibilities to analysis classes (documented in the domain model)
  - Model class interactions in Interaction diagrams



Provides traceability from Analysis and Design back to Requirements

# Different aspects of design

- ***Architecture design***: division into subsystems and components
  - How these will be connected
  - How they will interact
  - Their interfaces
- ***Class design***: Assign responsibilities to classes
- ***Interface design***: both User Interface and communication with other systems
- ***Algorithm design***: design of computational mechanisms
- ***Protocol design***: design of communications protocol
- **Database design**: design the database to persist objects data



Design objective =

create “good” classes

which are reusable and easy to maintain.

# Key Questions for Object-Oriented Design

**1. How should responsibilities be allocated to classes?**

**=> What classes should do what?**

**2. How should objects interact/collaborate to achieve a use case?**

**You can use to help you with the above:**

Responsibility-Driven Design Principles

Design patterns

# Well-known Pattern Families

- **GRASP** = General Responsibility Assignment Software ~~Patterns~~ Principles
  - Describe fundamental principles for **assigning responsibilities to classes** and for **designing interactions between classes**
  - GRASP try to formalize "common sense" in object oriented design.
- We will focus on the following GRASP principles:
  - Information Expert
  - Creator
  - Controller
  - Low Coupling
  - High Cohesion

# Class Diagram Review

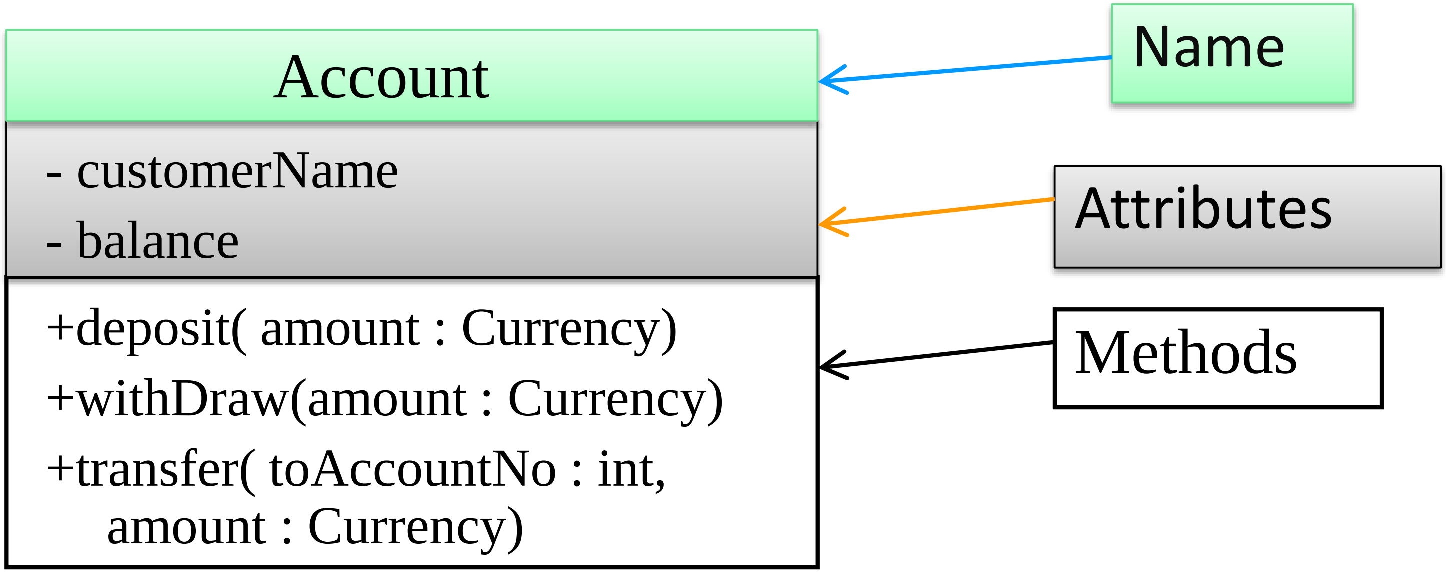
# Class diagram

- Static view of a system in terms of classes and relationships among the classes
- Modifiers are used to indicate visibility of attributes and methods.

‘+’ is used to denote *Public* visibility (visible to all)

‘#’ is used to denote *Protected* visibility (visible to derived classes)

‘-’ is used to denote *Private* visibility (only accessible within the class)



Account

- customerName
- balance

+deposit( amount : Currency)

+withDraw(amount : Currency)

+transfer( toAccountNo : int,
amount : Currency)

Name

Attributes

Methods

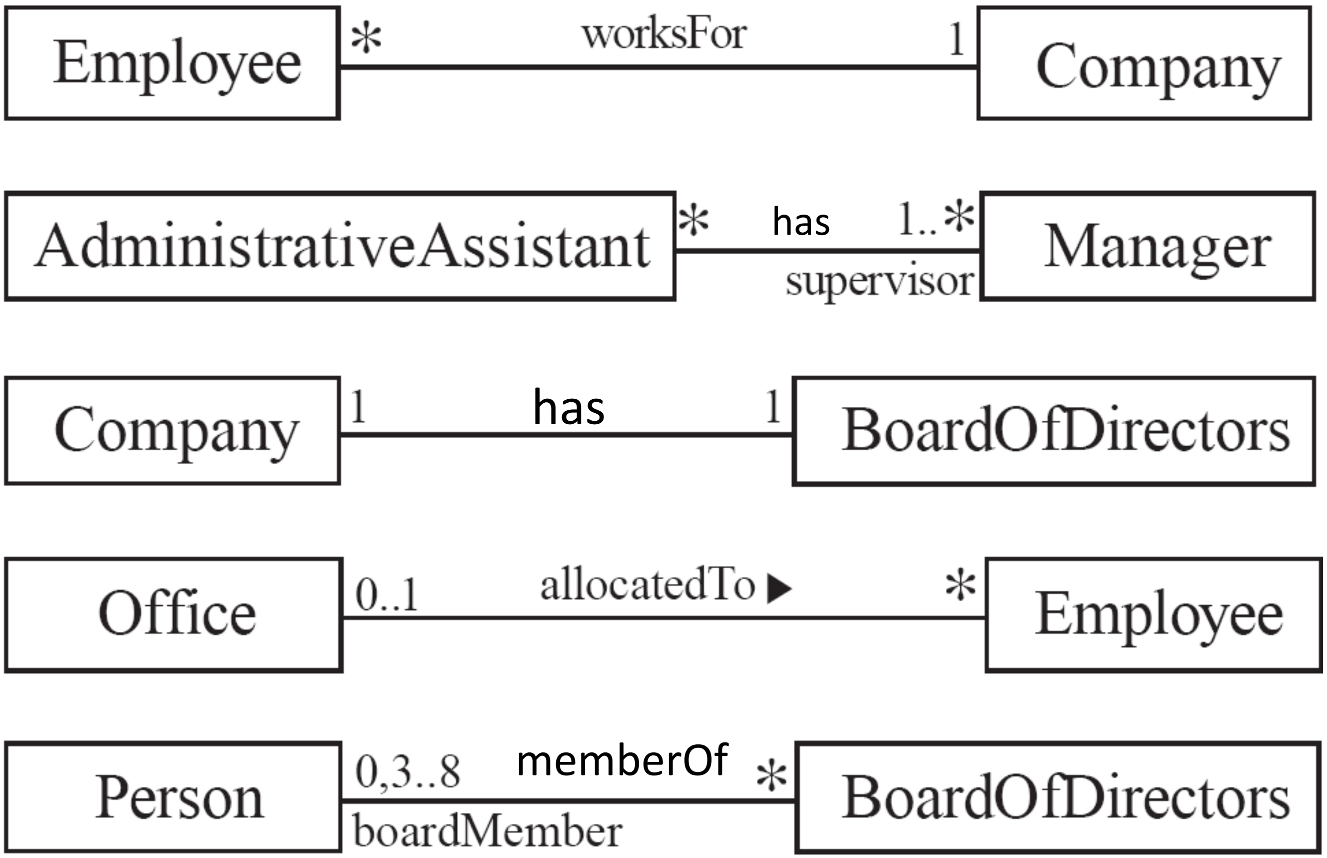
# Relationships between Classes

- There are two kinds of Relationships
  - Generalization (parent-child relationship)
  - Association (e.g., student enrolls in course)

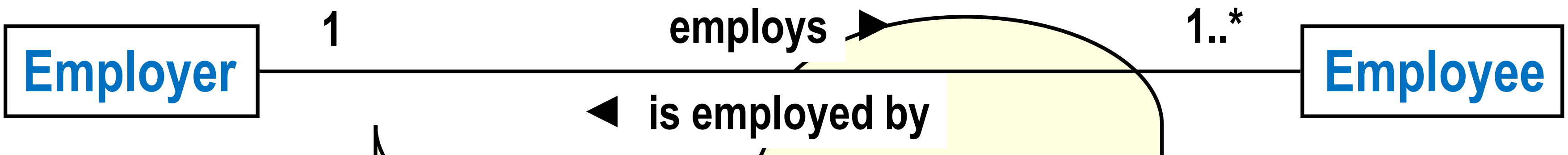
- Associations can be further classified as:
  - Aggregation
  - Composition

# Example associations

- Each association can be labelled, to make explicit the nature of the association



# Association : UML **Notation** and Typical Implementation

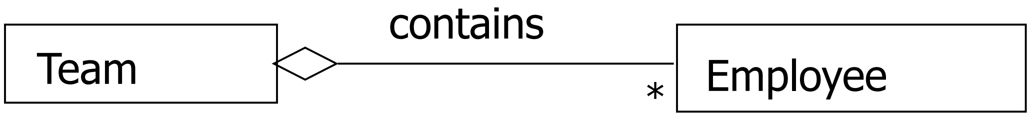


```
class Employer
{
    List<Employee> employees;
    . . . . .
}

class Employee
{
    Employer employer;
    . . . . .
}
```

# Aggregation

- Aggregation : (hollow diamond).
  Parts may ***exist independent of the whole***

e.g. **Employees may exist independent of the team**.

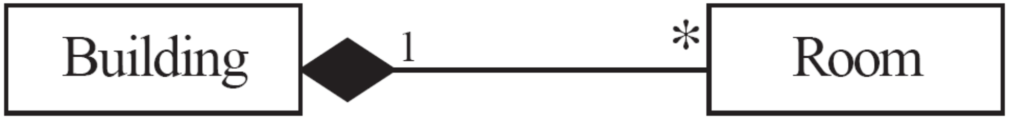


- Aggregation represents a relation “contains”, “is a part of”, “whole-part” relation.
  - Part instances can be added to and removed from the aggregate

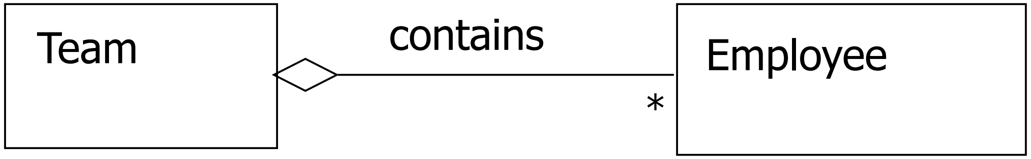
# Composition

- Composition : (filled diamond)
  ***Every part may belong to only one whole, and If the whole is deleted, so are the parts***
  - Stronger than an aggregate
  - Often involves a physical relationship between the whole and the parts, not just conceptual
  - the part objects are created, live, and die together with the whole: **the life cycle of the 'part' is controlled by the 'whole'.** Part cannot exist independent of the whole.

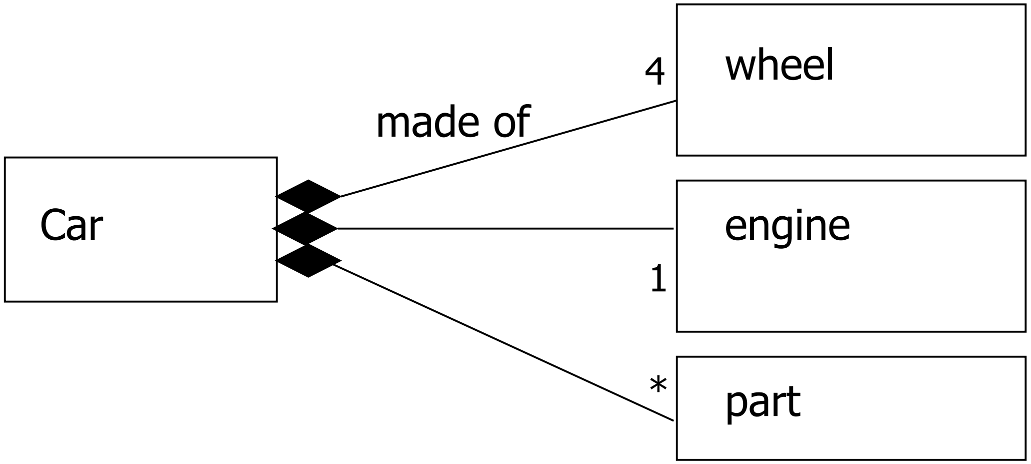
e.g. Each building has rooms that can not be shared with other building!



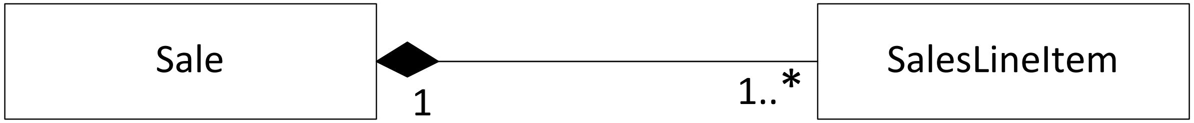
# Example: Aggregation vs. Composition

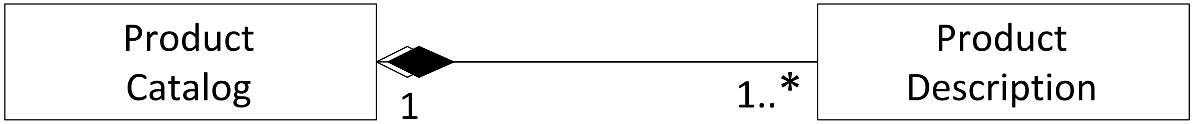


Aggregation: A team is made up of Many employees.



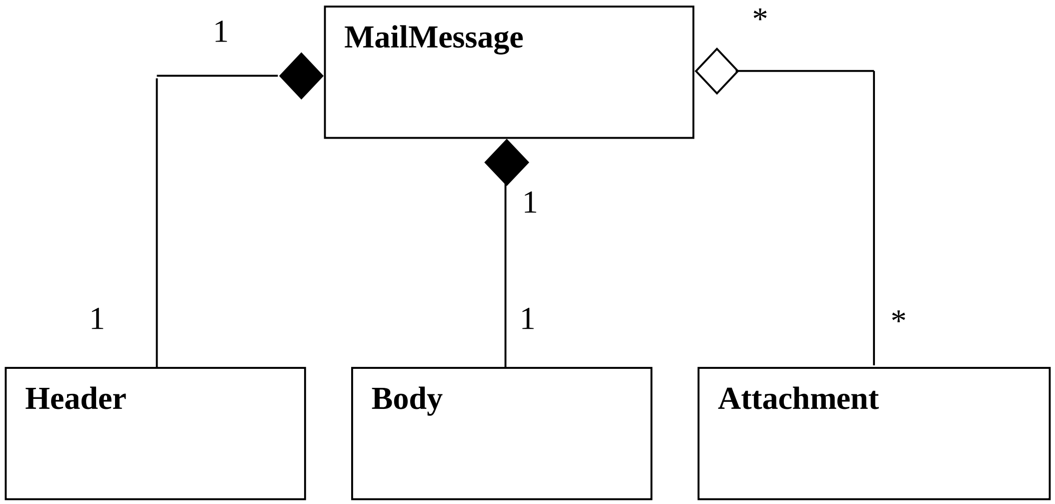
**Composition = Strong aggregation**





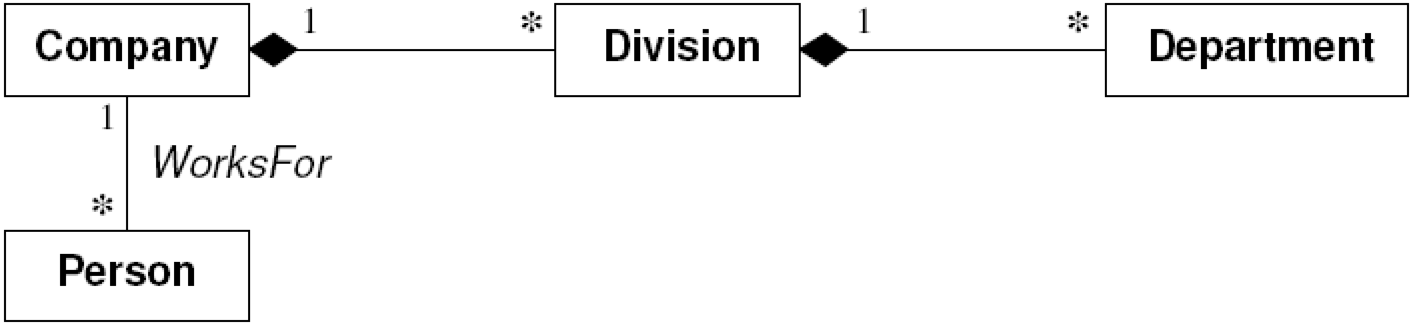
# Aggregation vs. Composition Example 1

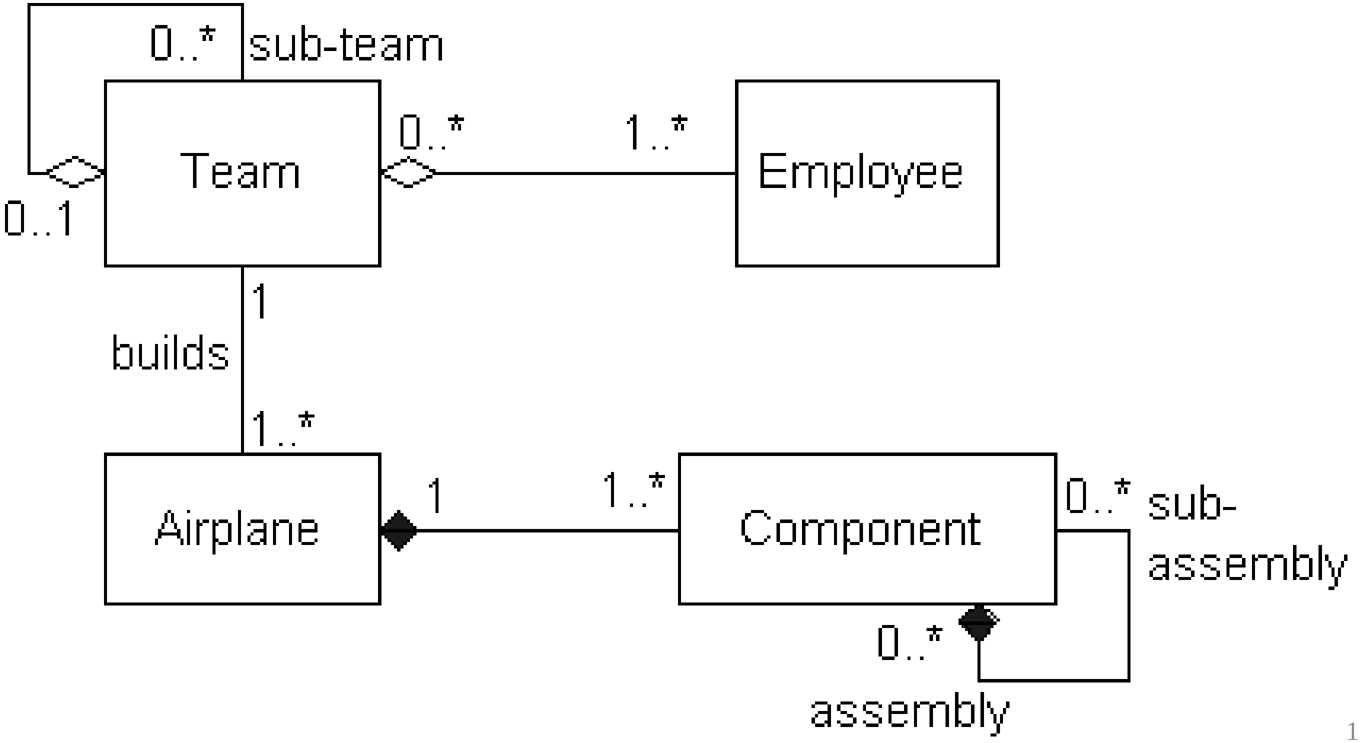
We could model the mail message example using composition and aggregation.



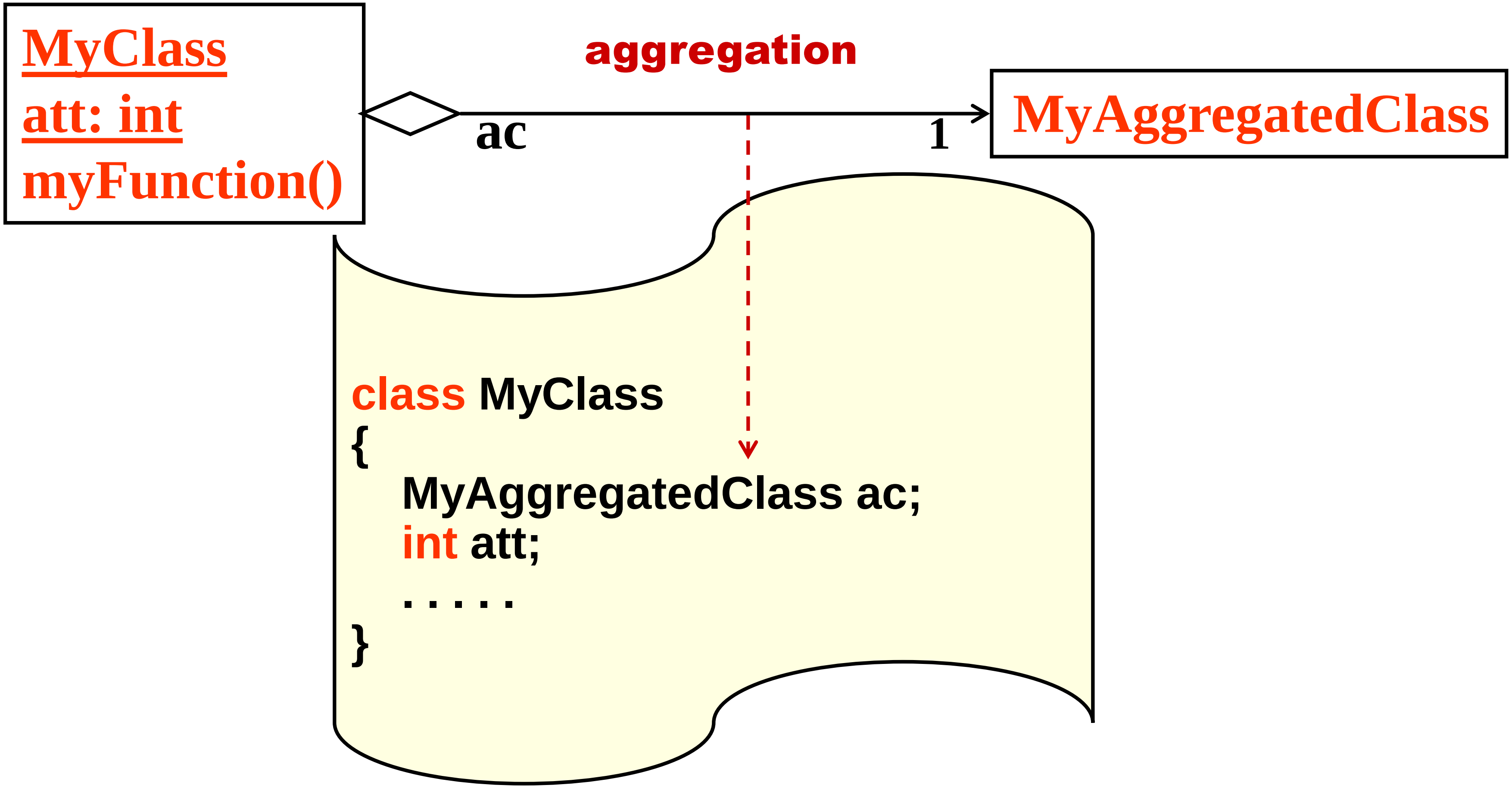
- When a MailMessage object is destroyed, so are the Header object and the Body object.
- The attachment object(s) are not destroyed with the MailMessage object, but still exist on their own.

# Aggregation vs. Composition Example 2



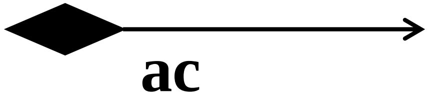


# Aggregation : UML Notation and Typical Implementation



```
class MyClass
{
    MyAggregatedClass ac;
    int att;
    . . . . .
}
```

**composition**

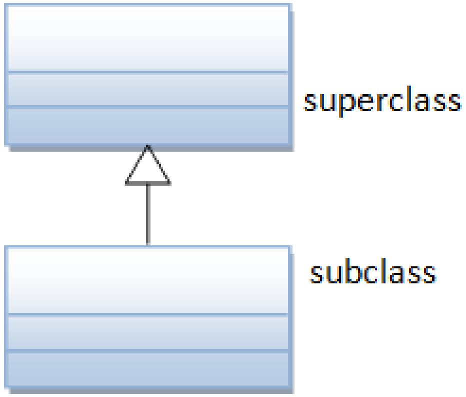


**Composed object exists only in the scope of owner object**

# Generalization

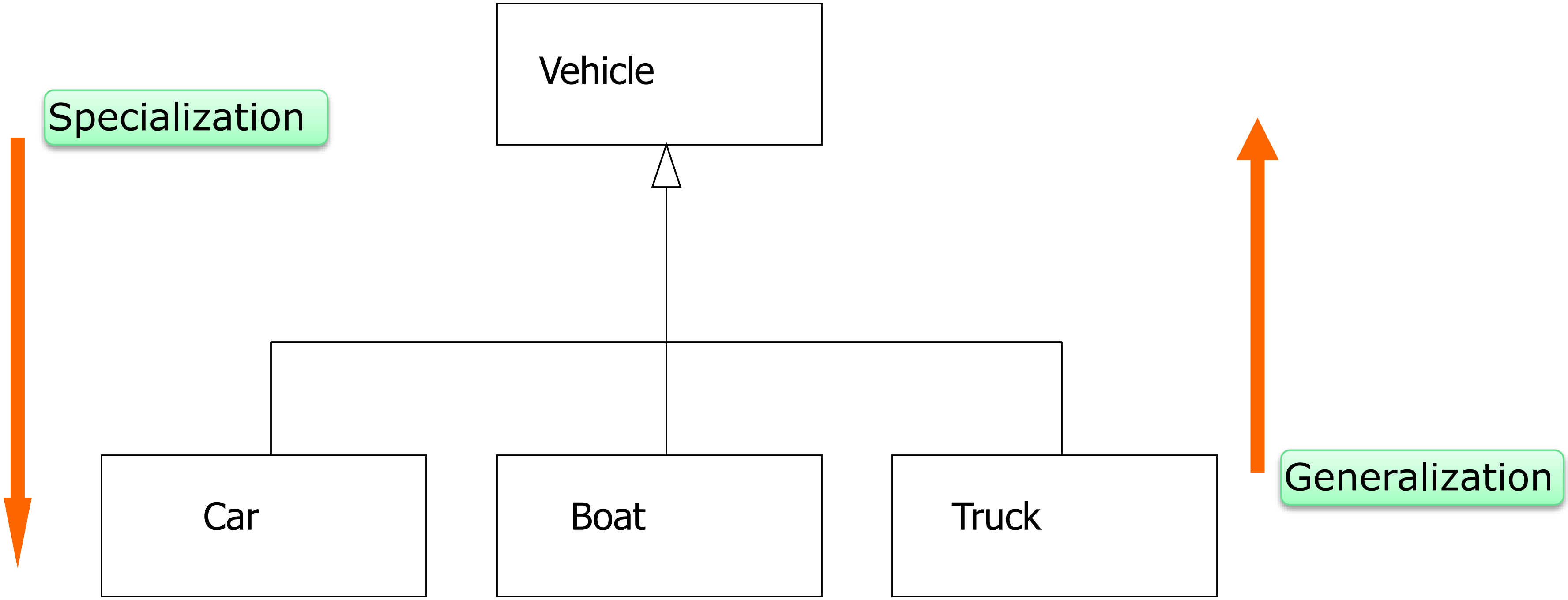
- Generalization is a relationship between a general and a specific class.
- The specific class called the subclass **inherits** from the general class, called the superclass.
- Public and protected properties (attributes) and behaviors (operations) are inherited.
- It represents “is a” relationship among classes Represented by a line with an hollow arrow head pointing to the superclass at the superclass end.

# Inheritance

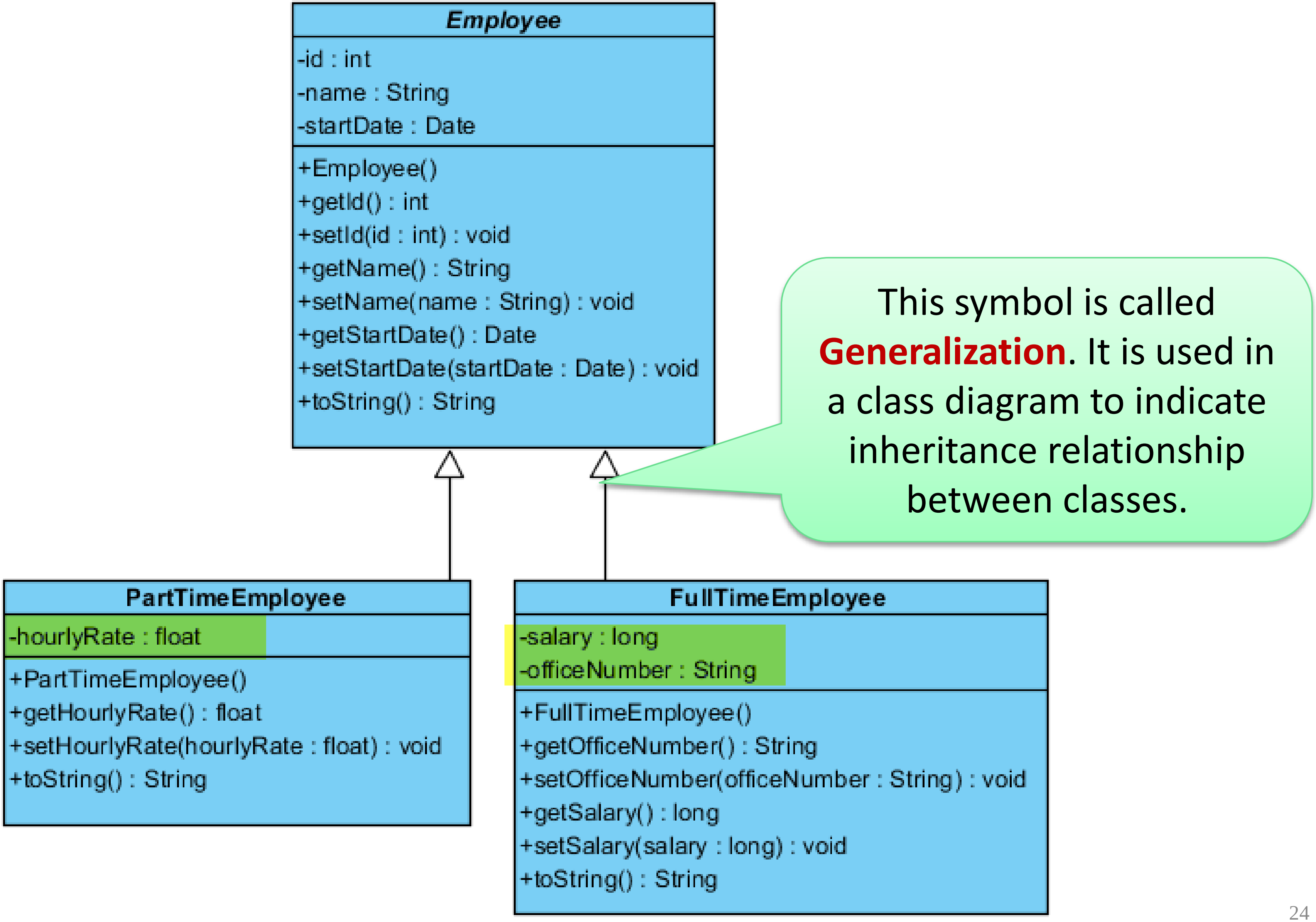


- **Ideas**
  - You can make a class that “inherits” characteristics of another class
    - The original class is called “parent class”, “super class”, or “base class”.
    - The new class is called “child class”, “subclass”, or “derived class”.
  - Subclass has access to all **non-private** (i.e, *public* and *protected*) **attributes and methods of the parent class**
  - Subclass can extend the base class by adding new attributes/methods and/or overriding the parent’s methods
- **Syntax**
  - public class ChildClass ***extends*** ParentClass { … }
- **Motivation**
  - Supports the key OOP goal of **code reuse** (i.e., don’t write the same code twice). Allow us to design **class hierarchies** so that **shared behavior is placed in a super class** then inherited to all classes that need it.

# Generalization Example 1



# Another Example - Employee Hierarchy



***Employee***

-id : int<br>
-name : String<br>
-startDate : Date

+Employee()<br>
+getId() : int<br>
+setId(id : int) : void<br>
+getName() : String<br>
+setName(name : String) : void<br>
+getStartDate() : Date<br>
+setStartDate(startDate : Date) : void<br>
+toString() : String

**PartTimeEmployee**

-hourlyRate : float

+PartTimeEmployee()<br>
+getHourlyRate() : float<br>
+setHourlyRate(hourlyRate : float) : void<br>
+toString() : String

**FullTimeEmployee**

-salary : long<br>
-officeNumber : String

+FullTimeEmployee()<br>
+getOfficeNumber() : String<br>
+setOfficeNumber(officeNumber : String) : void<br>
+getSalary() : long<br>
+setSalary(salary : long) : void<br>
+toString() : String

This symbol is called **Generalization**. It is used in a class diagram to indicate inheritance relationship between classes.

# Inheritance Rules

- The 100% Rule
  - All attributes and operations of the base class are applicable to the specialized class

- The ‘is-a-kind-of’ or ‘is-a’ Rule
  - The statement “<derived class> is a <base class>” should be true
  - Every instance of the <derived class> can be viewed as an instance of the <base class>

# is-a relationship vs. has-a relationship

- We distinguish between the is-a relationship and the has-a relationship
- ***Is-a*** represents inheritance
  - In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass
  - E.g., Student is a Person
- ***Has-a*** represents composition
  - In a *has-a* relationship, an object contains as members references to other objects
  - E.g., Student has a list of courses

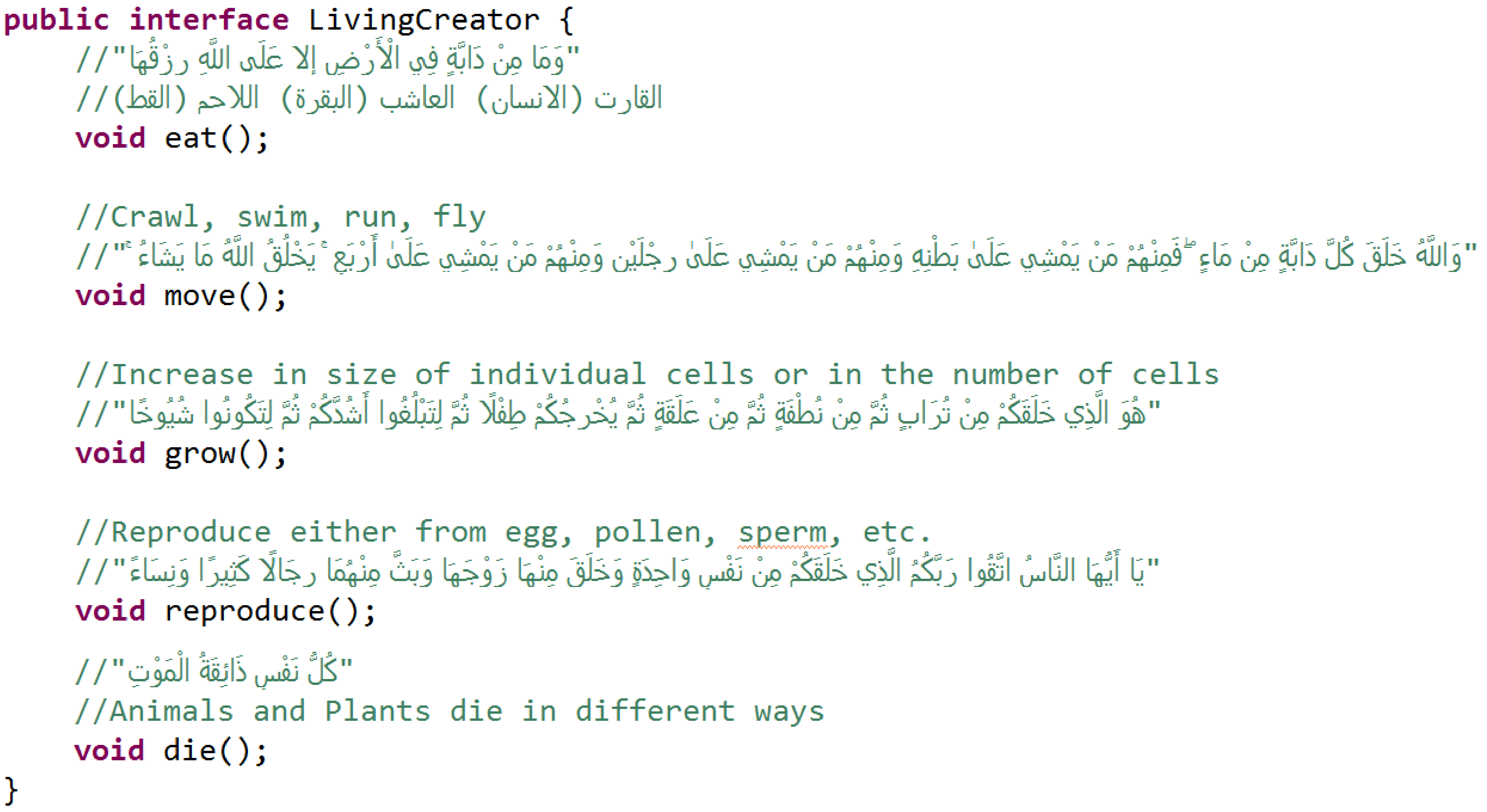
# Interfaces

- Idea
  - **Interfaces** are used to define  a set of common methods that must be implemented by possibly **unrelated classes**
  - The interface specifies ***what*** operations a class must perform but does not specify ***how*** they are performed
- Syntax

```
public interface SomeInterface {
  public SomeType method1(…);  // No body
  public SomeType method2(…);  // No body
}
public class SomeClass implements SomeInterface {
 // Real definitions of method1 and method 2
}
```

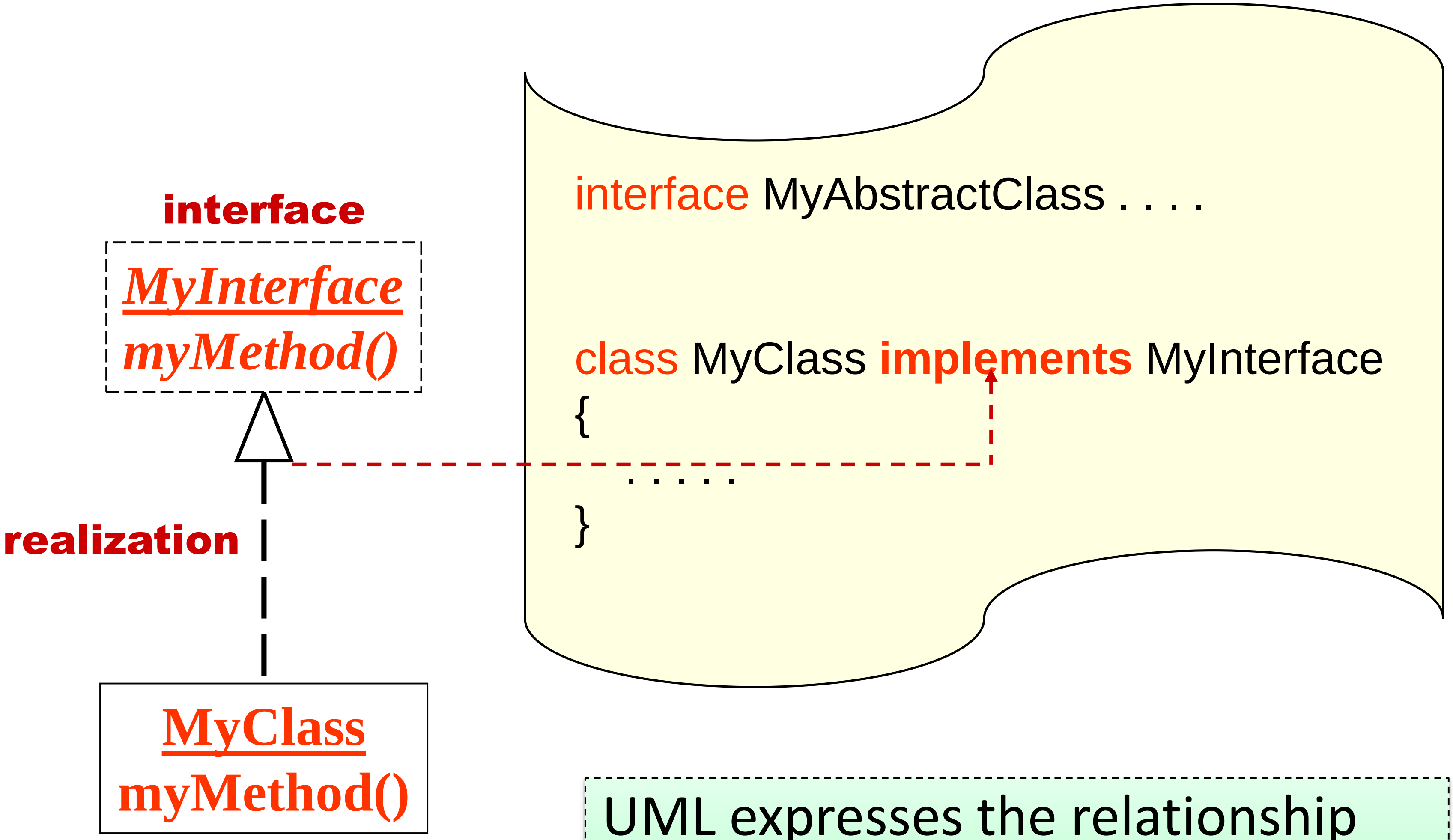
- Motivation
  - Interfaces are particularly useful for **assigning common functionality to possibly unrelated classes**

# Think of this *Interface!!!* implemented by ALL Living Creators (**Animals and Plants**) regardless of their inheritance hierarchy!



```
public interface LivingCreator {
    //"وَمَا مِنْ دَابَّةٍ فِي الْأَرْضِ إلا عَلَى اللَّهِ رِزْقُهَا"
    //القارت (الانسان)  العاشب (البقرة)  اللاحم (القط)
    void eat();

    //Crawl, swim, run, fly
    //"وَاللَّهُ خَلَقَ كُلَّ دَابَّةٍ مِنْ مَاءٍ ۖ فَمِنْهُمْ مَنْ يَمْشِي عَلَىٰ بَطْنِهِ وَمِنْهُمْ مَنْ يَمْشِي عَلَىٰ رِجْلَيْنِ وَمِنْهُمْ مَنْ يَمْشِي عَلَىٰ أَرْبَعٍ ۚ يَخْلُقُ اللَّهُ مَا يَشَاءُ ۚ"
    void move();

    //Increase in size of individual cells or in the number of cells
    //"هُوَ الَّذِي خَلَقَكُمْ مِنْ تُرَابٍ ثُمَّ مِنْ نُطْفَةٍ ثُمَّ مِنْ عَلَقَةٍ ثُمَّ يُخْرِجُكُمْ طِفْلًا ثُمَّ لِتَبْلُغُوا أَشُدَّكُمْ ثُمَّ لِتَكُونُوا شُيُوخًا"
    void grow();

    //Reproduce either from egg, pollen, sperm, etc.
    //"يَا أَيُّهَا النَّاسُ اتَّقُوا رَبَّكُمُ الَّذِي خَلَقَكُمْ مِنْ نَفْسٍ وَاحِدَةٍ وَخَلَقَ مِنْهَا زَوْجَهَا وَبَثَّ مِنْهُمَا رِجَالًا كَثِيرًا وَنِسَاءً"
    void reproduce();

    //"كُلُّ نَفْسٍ ذَائِقَةُ الْمَوْتِ"
    //Animals and Plants die in different ways
    void die();
}
```

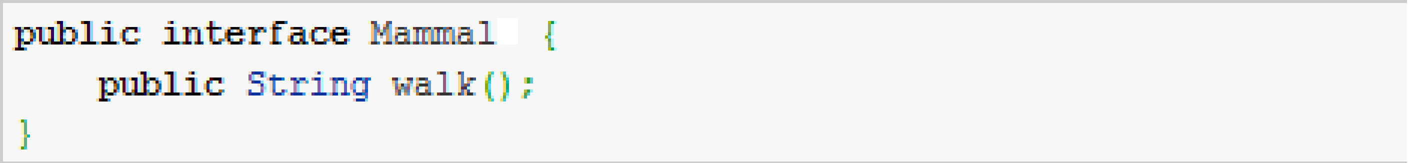
# Interfaces
## UML Notation …… Typical Java Implementation



```
interface MyAbstractClass . . . .


class MyClass implements MyInterface
{
      . . . . .
}
```

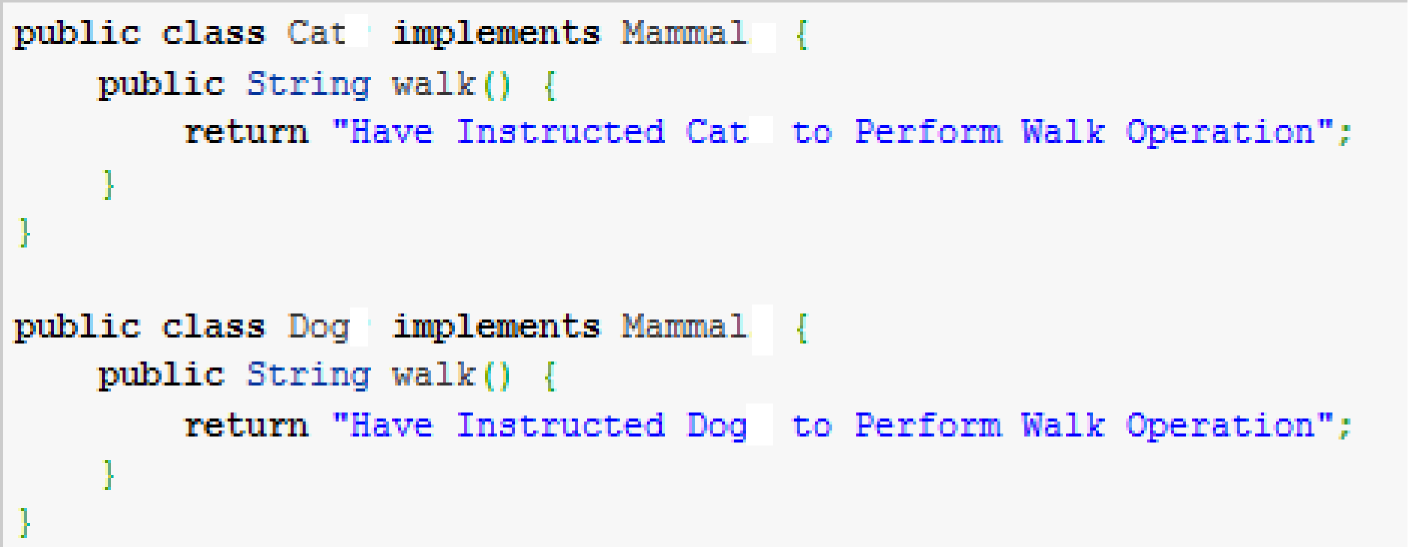
UML expresses the relationship between a class and an interface through a **realization**.

*Interface Code:*



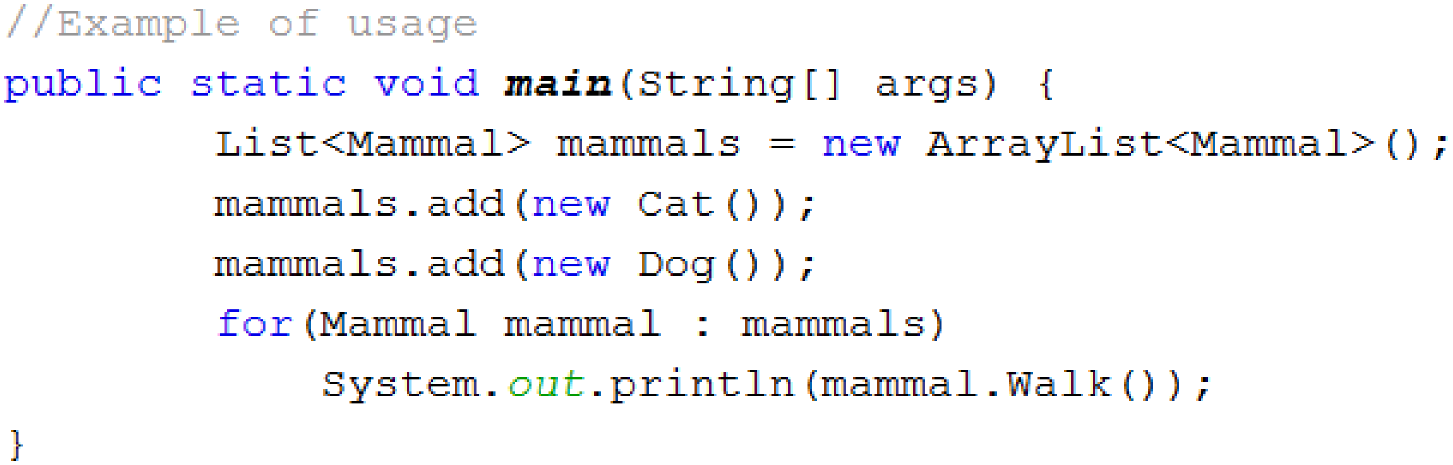
```java
public interface Mammal  {
    public String walk();
}
```

*Interface Implementation Class:*



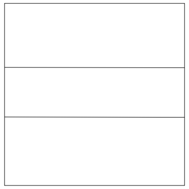
```java
public class Cat  implements Mammal  {
    public String walk() {
        return "Have Instructed Cat  to Perform Walk Operation";
    }
}

public class Dog  implements Mammal  {
    public String walk() {
        return "Have Instructed Dog  to Perform Walk Operation";
    }
}
```

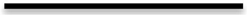
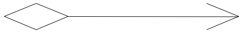
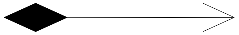
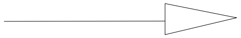
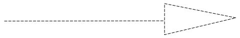
## Java code for the example shown in the previous slide.



```java
//Example of usage
public static void main(String[] args) {
        List<Mammal> mammals = new ArrayList<Mammal>();
        mammals.add(new Cat());
        mammals.add(new Dog());
        for(Mammal mammal : mammals)
            System.out.println(mammal.Walk());
}
```

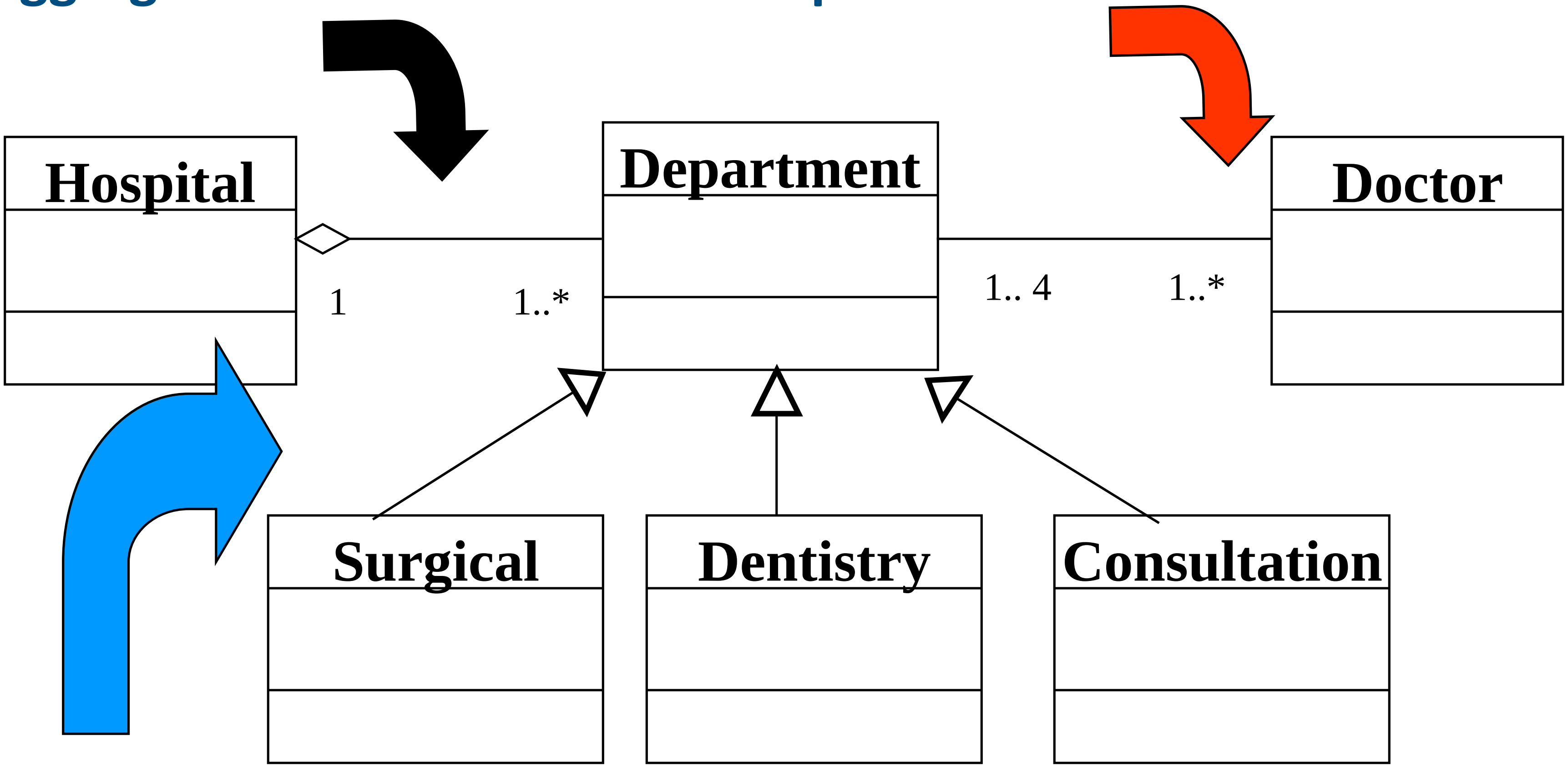
# Summary of Class Relationships

- Class 

- Association 
- Aggregation 
- Composition 
- Generalization 
- Realization 

Aggregation – “has a” relationship

Association “Uses”

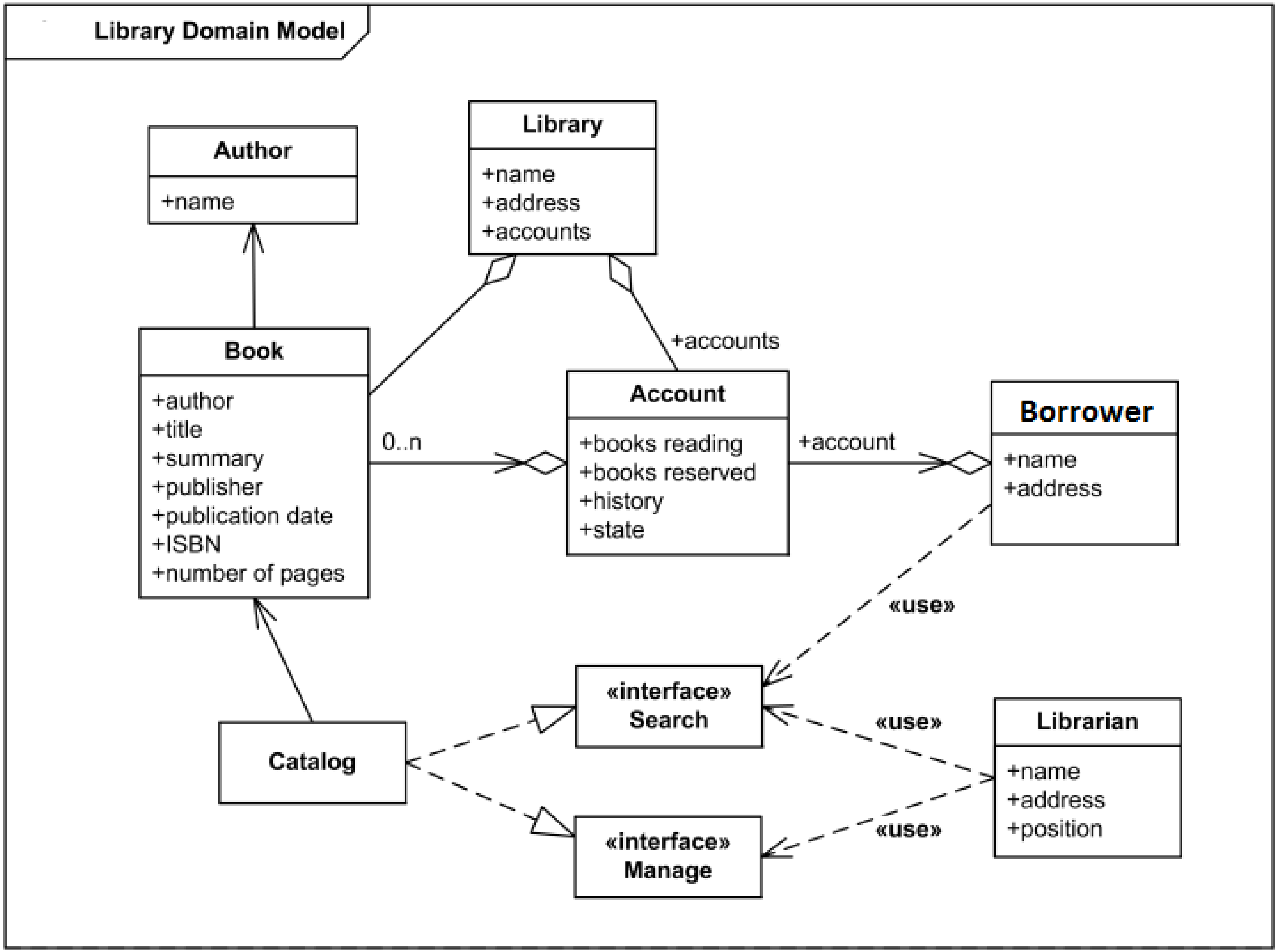


Inheritance “is a / is a kind of”

# Summary of relationships between classes

- "is a“ is inheritance
- "has a“ is composition / aggregation
- "uses a“ is association
- "looks like“ is interface
- Aggregation and Composition - both deal with **part-of** relationships
  - **Composition is stronger:** the composite object is responsible for the creation and destruction of the parts => part **cannot** exist on it's own
  - In an aggregation relationship, **the part may be independent of the whole** but the **whole requires the part**

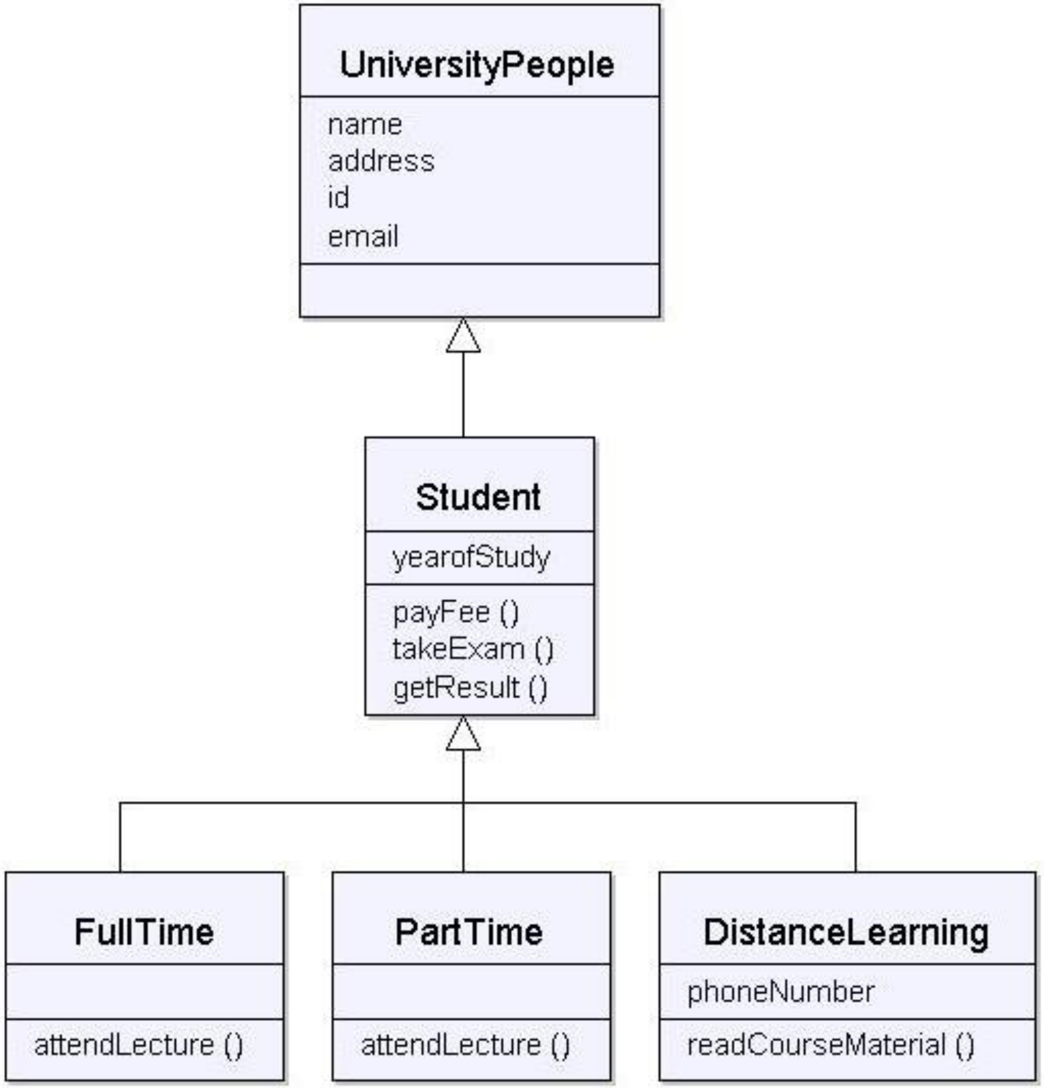
# Class Diagram Example - Library Domain Model



# Exercise 1 - Generalization

- Consider the following classes: UniversityPeople, Student, FullTime, PartTime and Distance Learning student. Draw a UML class diagram. Add properties and operations to the classes.

# Exercise 1 - Solution


UniversityPeople
name
address
id
email
Student
yearofStudy
payFee ()
takeExam ()
getResult ()
FullTime
attendLecture ()
PartTime
attendLecture ()
DistanceLearning
phoneNumber
readCourseMaterial ()