Class Diagram – Part 01

Content

- Object Oriented Analysis
  - Identifying Classes
    - Noun and Verb Method
    - Class Responsibility and Collaboration (CRC) Cards.
- UML Class Diagrams

- “Do not begin an OO design until at least part of the use case model has been specified”
- The Use Case Model:
  - provides clues about the required classes and their functionality;
  - it will also hint at how the resulting objects should interact.
- The use case scenarios drive the OO design.

Identifying Classes

- Discover Classes for the requirements.

1. **Noun Analysis**
2. **Use Case Analysis**
3. **CRC Method**

Noun – Verb Analysis

Noun Verb Analysis

- A simple recipe
  - Step 1: We can identify objects in our problem statement by looking for ***nouns*** and ***noun phrases***.
  - Step 2: Each of these can be underlined and becomes a candidate for an object in our solution.
  - Step 3: We can eliminate some objects by some simple rules.

Rules For Rejecting Nouns

1. Redundant
2. Vague
3. An event or an operation
4. Outside scope of system
5. Meta-language
6. An attribute

Activity 1

- In a DVD rental store there are two types of users, a registered member can borrow up to 3 DVDs at a time. These members have already paid a deposit and only need to pay 50/= per DVD.
- Customers who are not registered can also borrow DVDs at the rate of 75/= per DVD. They are required to surrender their id card for this purpose.
- Members can keep the DVD for three days and when they are returned appropriate fines may be calculated.

Activity 1

- In a DVD rental store there are two types of users, a registered member can borrow up to 3 DVDs at a time. These members have already paid a deposit and only need to pay 50/= per DVD.
- Customers who are not registered can also borrow DVDs at the rate of 75/= per DVD. They are required to surrender their id card for this purpose.
- Members can keep the DVD for three days and when they are returned appropriate fines may be calculated.

Activity 1 Answer :

- DVD -class
- User
- Customer
- Member
- ID Card –
- Fine -
- Deposit –
- Store -
- Rate -

Redundant

Meta language

ID number is an attribute

Attribute

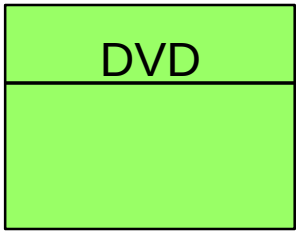
Attribute / operation

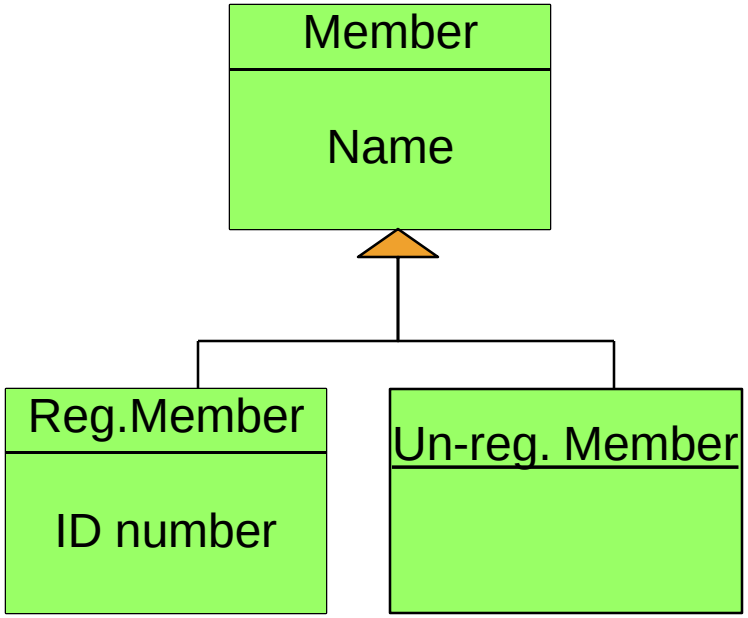
Out of scope

Attribute

Final Classes ?

- **DVD**
- **Reg. Member**
- **Un reg. Member**





Activity 2

Airline Reservation System

'Tour-Lanka Airlines' **runs sightseeing flights from Colombo, the capital of Sri Lanka. The reservation system keeps track of passengers who will be flying in specific seats on various flights, as well as people who will form the crew.**

For the crew, the system needs to track what everyone does, and who supervises whom. *Tour-Lanka Airlines* runs several daily numbered flights on a regular schedule. *Tour-Lanka Airlines* expects to expand in the future so that system needs to be flexible, in particular, it will be adding a frequent flyer plan.

ACTIVITY 2

AIRLINE RESERVATION SYSTEM

'Tour-Lanka Airlines' **runs sightseeing flights from Colombo, the capital of Sri Lanka. The reservation system keeps track of passengers who will be flying in specific seats on various flights, as well as people who will form the crew.**

For the crew, the system needs to track what everyone does, and who supervises whom. *Tour-Lanka Airlines* runs several daily numbered flights on a regular schedule. *Tour-Lanka Airlines* expects to expand in the future so that system needs to be flexible, in particular, it will be adding a frequent flier plan.

Other Nouns

- **Reservation system** – it is the system.
- **Sightseeing flight** –Better take „flight". its is more general and lead to flexibility
- **Seat** – attribute of flight
- **Crew** –They are employees (can have a sub class „Crewmember")
- **Schedule** – composite information bundle> can be managed with attributes of flight class
- **Future** – not part of current scope

Frequent flier plan –not part of current scope

Tour-Lanka Airlines – It is the system
Colombo – instance
Sri Lanka - instance

Classes

- **Flight**
  - **Sightseeing flight**
- **Passenger**
- **Employee**
  - **crewmember**

DISCOVERING CLASSES

2. USE CASE ANALYSIS

USE CASE ANALYSIS

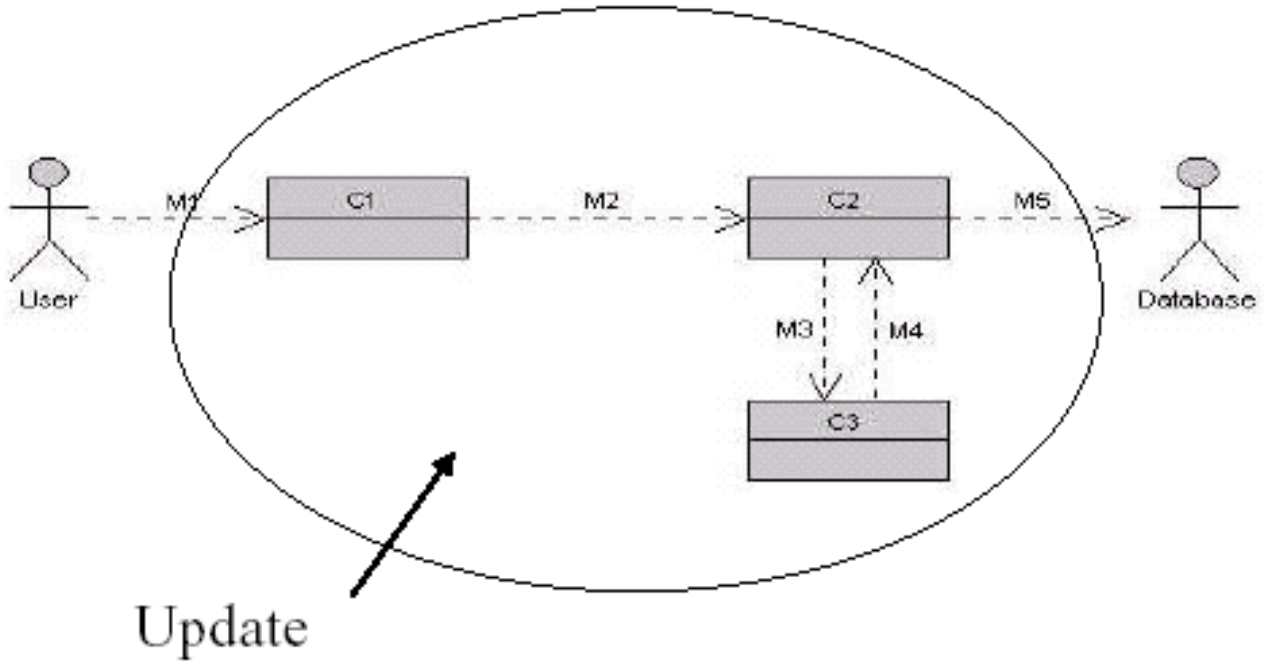
- Consider the above use case :

Update

- The user interacts with the system to perform an „update".
- The system interacts with the database.

USE CASE ANALYSIS

- Look into the use case further



There are some **classes**: C1, C2, C3
There are some **methods**: M1,M2,M3, etc…

USE CASE ANALYSIS

1. Determine **what classes** are needed, without accidentally picking:
   - there is an infinite number of classes to choose from.
2. Determine the **responsibilities** of the classes.
   - i.e. what they are supposed to do:
   - there are no limitations on the functionality of a class!
3. Define the **collaborations** between classes:
   - i.e. the relationships between them.

USE CASE ANALYSIS

- Once the basic scenarios have been developed for the system, it is time to identify candidate classes and indicate their responsibilities and collaborations.

- There are ways to make this easier.
  - Patterns.
  - Experience.
  - CRC Cards.

DISCOVERING CLASSES

3. CRC CARDS METHOD

What is CRC Cards Method?

- Class Responsibility and Collaboration (CRC) cards:
  - developed by Beck and Cunningham (1989).
- Used to explore class relationships between classes in a particular use-case scenario.

A CRC Card

- A CRC is a 4 × 6 inch card divided into three sections.
  - The **name** of the class on the **top** of the card.
  - The **responsibilities** of the class on the **left** of the card.
  - The class **collaborations** on the **right** of the card.

  (the list of other classes with which the class collaborates to realize its responsibilities)

Class name:	
Responsibilities:	**Collaborations:**

WHAT IS A RESPONSIBILITY?

- A responsibility is something that class has to
  - Know OR Do
- A card may have one or more responsibilities
- The list of responsibilities on the CRC card is **not** the list of methods in the class.
  - A responsibility may be realized by several methods.
  - A responsibility may require multiple collaborations.

What is a Collaboration ?

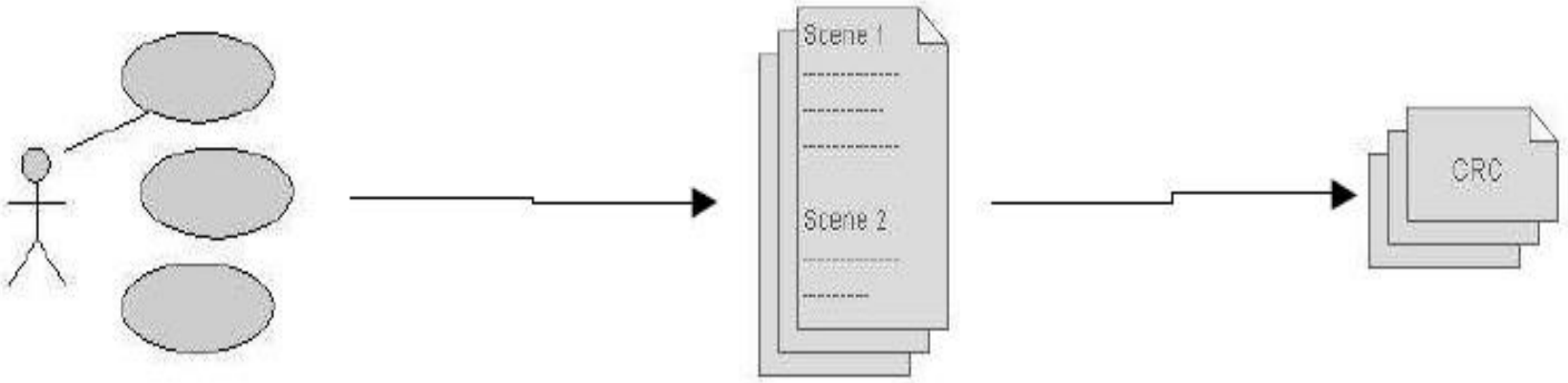
- A class may not be able to act upon its responsibility on its own.
- It may require some interaction with other classes.(need help)
  - these helper classes are the collaborators.

CRC Cards Method

- The easiest way to generate CRC cards is by using the scenarios in the use case model. No need to examine ALL scenarios, just a representative sample.

- The cards are used to "role play" each scenario

Use case ***Set of use case scenarios*** ***Set of CRCs***



CRC Cards Method

- Each scenario adds to the responsibilities and collaborations on several CRC cards.

- When you consider all scenarios for all use cases:
  - You have a set of CRC cards (indicating all the classes your system requires to realize the use cases).
  - For each CRC card, you will have the list of **all** responsibilities the class realizes.
  - For each responsibility, you will have the list of **all** collaborating classes.

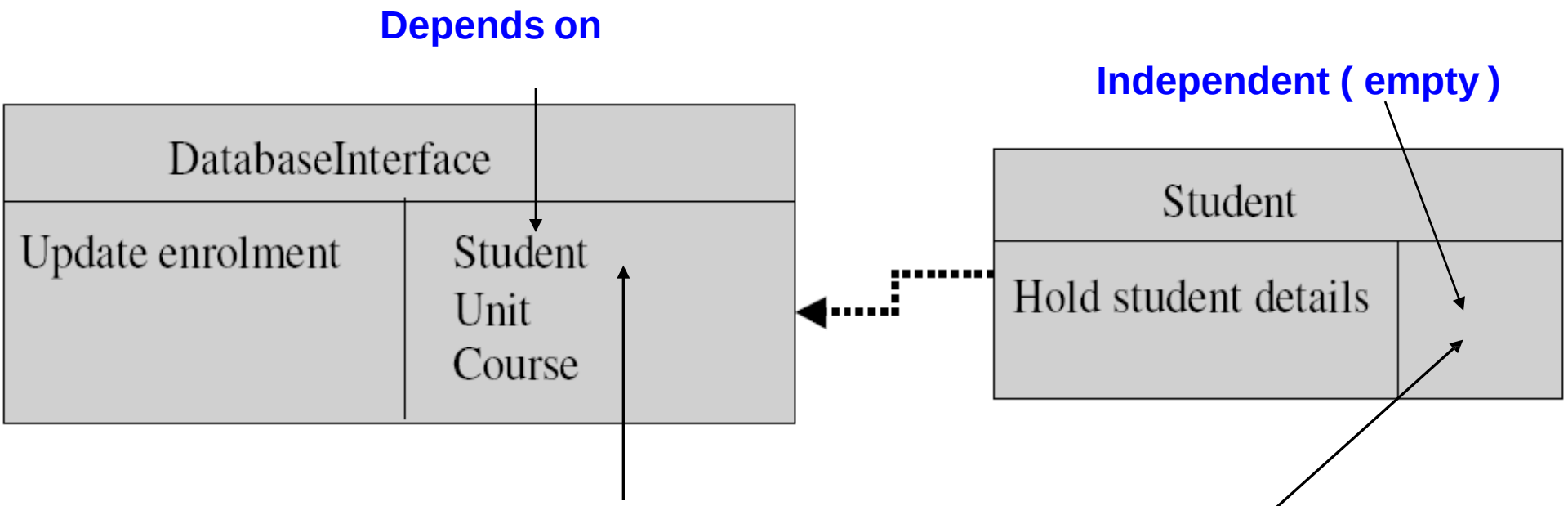
CRC CARDS – EXAMPLE

- An example of one responsibility requiring multiple collaborations.

DatabaseInterface	
Update enrolment	Student Unit Course

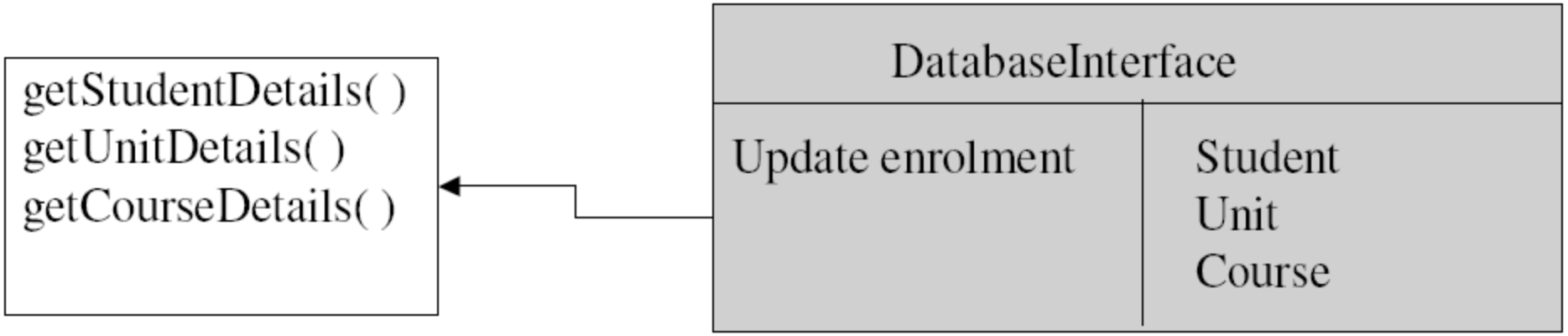
- The DatabaseInterface Class is **responsible for updating** the enrolment.
- It collaborates with the **Student, Unit and Course** classes to realize this responsibility.

CRC CARDS– EXAMPLE



- The Student class has a responsibility that is required by the DatabaseInterface class.
- Student class does notrequire DatabaseInterface to realize its responsibility.

CRC-EXAMPLE



- One responsibility may requires several methods to realize it.
- To "update enrolment" the DatabaseInterface class must call several methods in other classes.

Activity 3:

SCENARIO STEP :

"YOU (THE STUDENT) E-MAIL YOUR RESUME TO YOU'RE MY (LECTURER'S) BOSS "

Write few CRC cards for this

Steps

1. Take these as the *classes* for (the first) cards.

   Student, Resume , Boss, Lecturer

2. The only verb you need to make this scenario happen is
   "**send an e-mail with attachment**"

3. Add that as a *responsibility* to the “Student" card.
   However, Student can not do this activity alone.
   (because the card doesn't have enough information).

   So, Student needs two *collaborators/helpers* :

Have to attach **The resume** > student should have a Resume
Find **boss’s e-mail address**. > student should ask Lecturer

Activity 1: Basic CRC Cards

Student	
Responsibility	**Collaborators**
e-mail Resume to a given e-mail address. find e-mail address.	resume lecturer

Resume	
Responsibility	**Collaborators**
Know resume details	

Boss	
Responsibility	**Collaborators**
Know e-mail address	

Lecturer	
Responsibility	**Collaborators**
Get boss's e-mail address.	Boss

CRC CARDS – WHY BOTHER?

- CRC card generation usually results in **multiple** cards, each representing a class.

- From the set of cards, it makes sense to.
- Group classes with similar responsibilities.
  - Example: All the classes with database functionality.
- Group classes that collaborate together.
  - Example: Entity classes with data classes.
- The groupings are sometimes conflicting

CRC CARDS – WHY BOTHER?

- A collaboration graph can be constructed from the CRC cards.
- This helps to identify sub-systems - sets of classes that tend to collaborate with each other.
- CRC cards are the starting point for drawing class diagrams.

CRC CARDS – SOME IMPORTANT POINTS

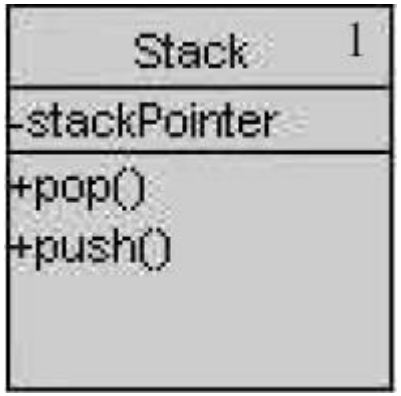
- Classes that collaborate with each other are candidates for an **association relationship.**
- Classes that share overlapping responsibilities are candidates for a **generalization relationship.**
- Classes that offer functionality using other classes are candidates for **composition /aggregation** relationships.

Class Diagrams

REPRESENTING CLASSES

CLASS STRUCTURES

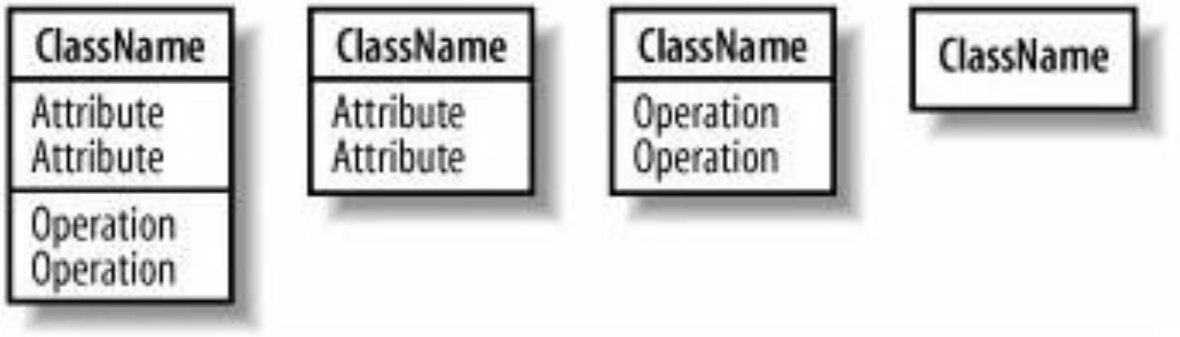
- A UML class structure describes several important properties of a class.
  - Name (optional multiplicity).
  - Attributes (optional).
  - Methods (optional).



- There is always exactly one (1) instance of this Stack class, with 1 attribute and 2 methods.

ORGANIZING ATTRIBUTES AND OPERATIONS

- Don't need to show every attribute and operation of a class.
- Choice depends on level of detail desired and the point of view.
- Ellipsis (...) explicitly shows additional elements.



CLASS ATTRIBUTES

- The attributes of a class can be fully specified as follows:

visibility **name** multiplicity : type = initial value {property}

VISIBILITY

Attribute visibility defines which classes can directly access the attribute (in their operations).

+	Public
#	Protected
-	Private

public (+): Visible to any class that can see the class containing the attribute.

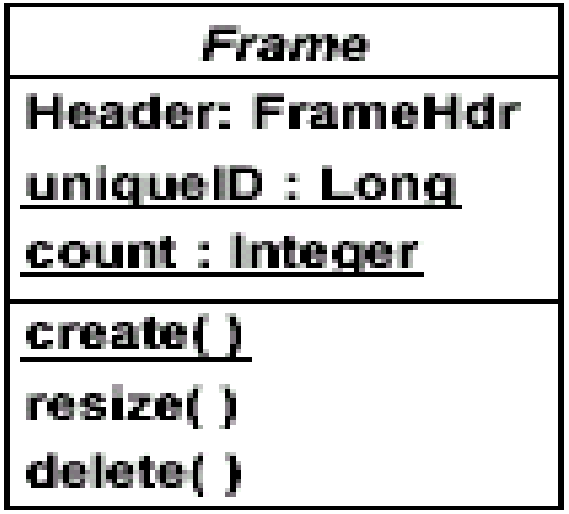
package (~): Visible to all classes of the package.

protected (#): Visible to the class itself and its subclasses.

private (-): Visible only to the class itself

SCOPE

- The scope of the attributes and methods can also be defined.
  - Instance scope (the default).
    - Each instance of the class effectively holds its own copy of the attribute or method.
  - Class scope.
    - A single copy of the attribute or method exists for all instances of the class
    - Class variables are declared as static
    - Denoted by underlining the name of the attribute or the method.



Protected String name;

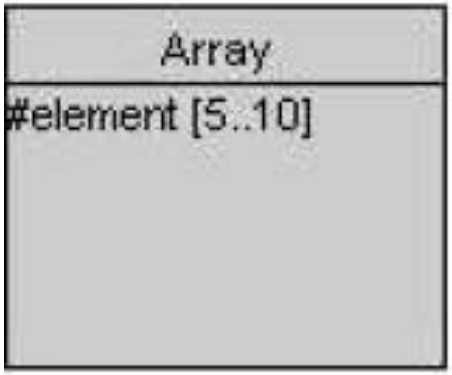
private static int count = 0;

 # name : String

- count : int = 0

MULTIPLICITY

- UML permits multiple-valued attributes, (sets or arrays)
- Multiplicity of a class attribute is defined within square brackets (the default is 1).



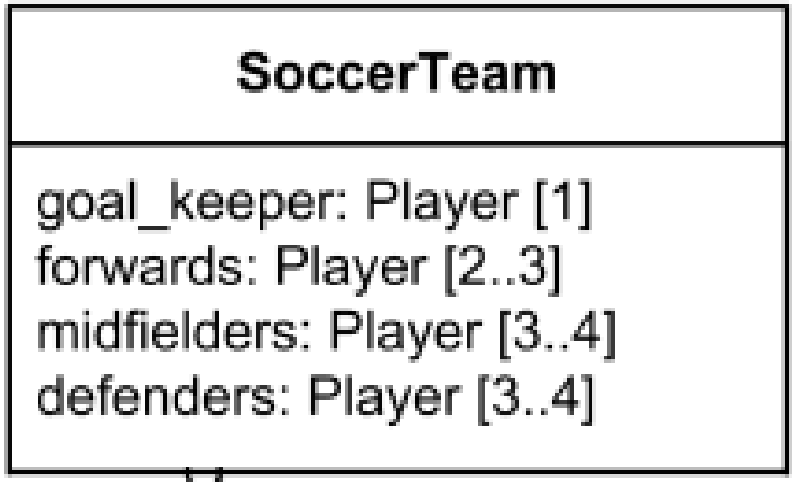
- There is a tendency to confuse multiplicity specifications with array specifications.
  - The Array class structure states that
    - there are between five (5) and ten (10) **element** in Array;
  - NOT that there is an array of **element** with indices from 5 to 10.

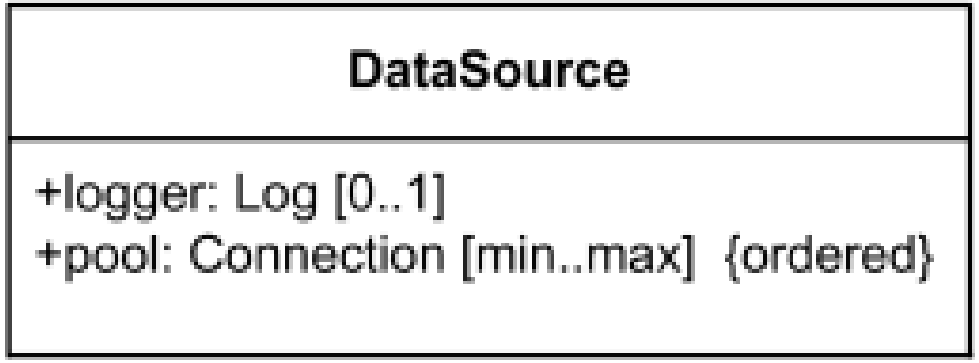
MULTIPLICITY

[]	This attribute has a single value of the specified Type.
[0..]	This attribute can have a value of **null**.
[*]	This attribute's value is a collection of values.
[1..*]	This attribute's value is a collection that contains at least one value.
[n..m]	This attribute's value is a collection that contains between *n* and *m* values.

MULTIPLICITY

- If a multi-valued attribute is and specified as **ordered**, then the collection of values in an instantiation of this element is sequentially ordered.
- By default, collections are not ordered.





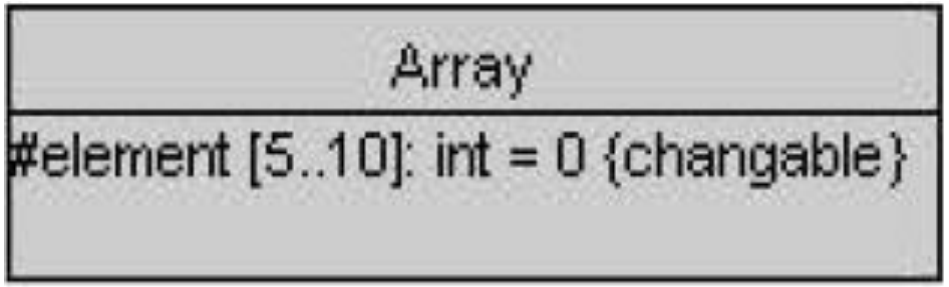
PROPERTIES

- Properties of an attribute can be.
  - ***changeable.***
    - The attribute can be modified without restriction.
  - ***addOnly.***
    - The attribute cannot be deleted or updated but others values may be added to the attribute (applies only to attributes with a multiplicity greater than one).
  - ***frozen.***
    - The attribute can be initialized, but never modified.

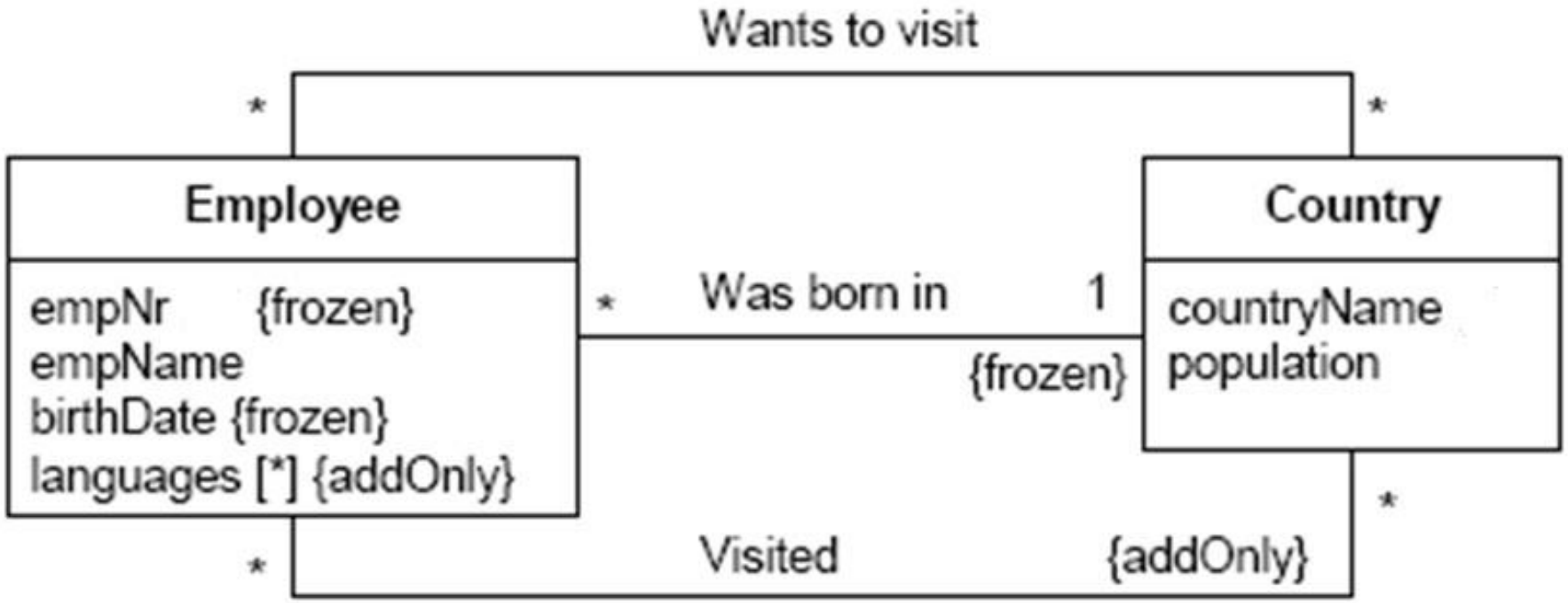
      They are final attributes.

PROPERTIES

- This structure defines a class attribute:
  - with protected visibility;
  - named *element*;
  - of type *int*;
  - there are between 5 to 10 (inclusive) instances of this attribute in the class;
  - the default value for the attribute is 0;
  - the attribute is changeable.



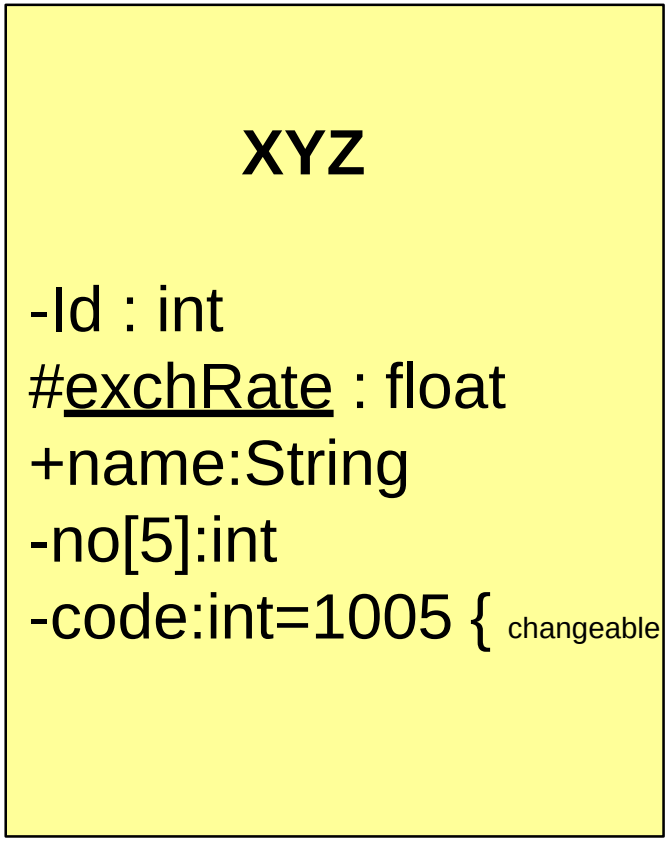
PROPERTIES


Wants to visit
*
*
Employee
empNr {frozen}
empName
birthDate {frozen}
languages [*] {addOnly}
*
Was born in
1
{frozen}
Country
countryName
population
*
*
Visited
{addOnly}

ACTIVITY 4

- Specify the following class according to the UML notation:

```
class XYZ {
    private int id;
    protected static float exchRate;
    public string name;
    private int no[5];  private
    int code= 1005;
}
```



CLASS METHODS

- The methods in a class can be fully specified as follows:

name (parameters) : type

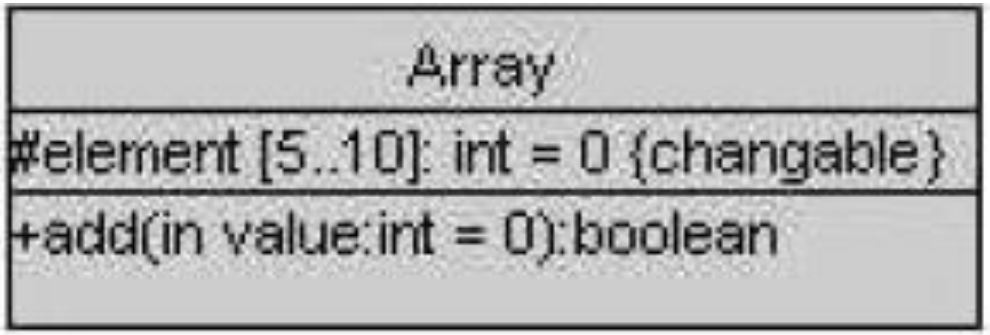
- The full specification for a parameter is as follows:

direction **name** : type = initial

Direction can be one of:

- In
  - input parameter, may not be modified
- out
  - output parameter, may be modified to pass back to caller
- inout
  - input parameter; may be modified

CLASS METHODS

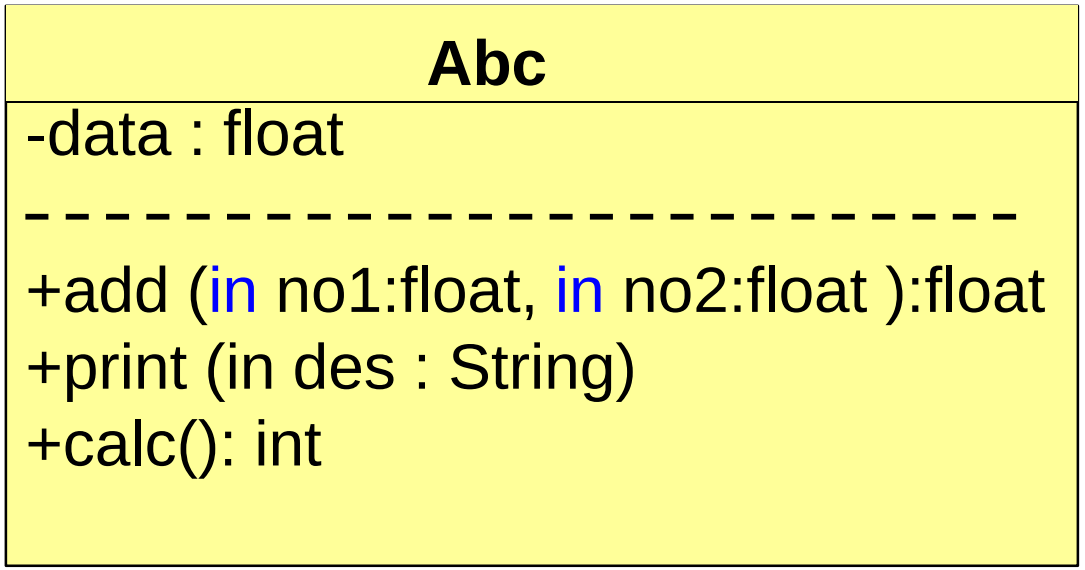


- This defines a class structure called Array:
  - method name = *add;*
  - Return type= *boolean;*
  - Parameter = *value*
    - that is used only as an import for the method
    - that is of type *int*
    - with a default value of 0 if none is specified

Activity 5

Specify the following class according to the UML notation:

```
Class Abc {
Private float data;
Public float add (float no1, floatno2);
Public static void print ( String des);
Public int calc();
}
```

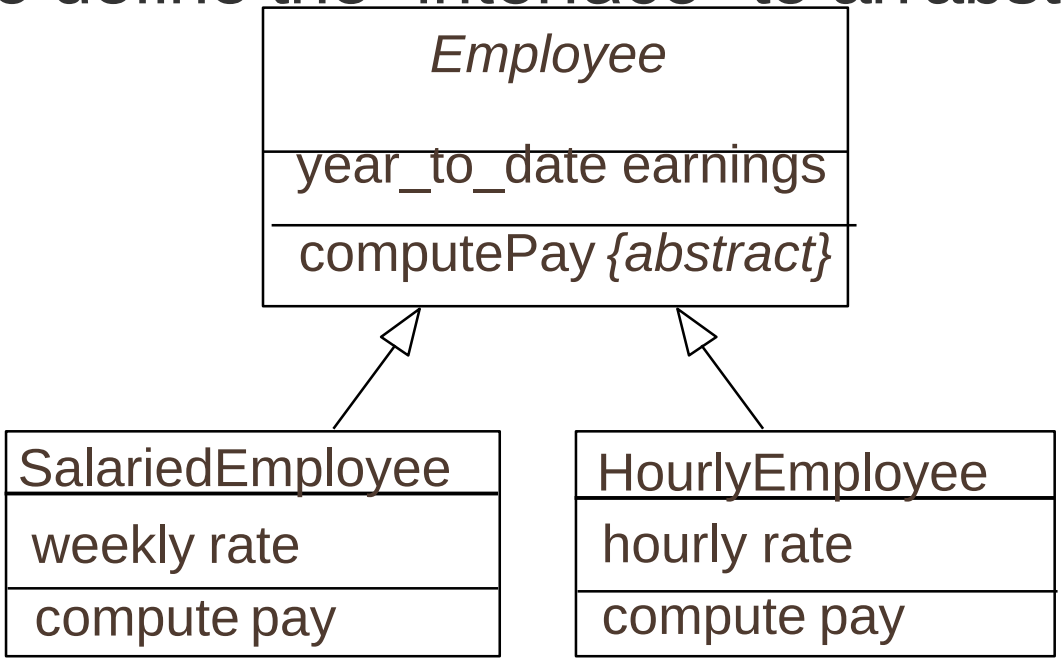


CLASS TYPES

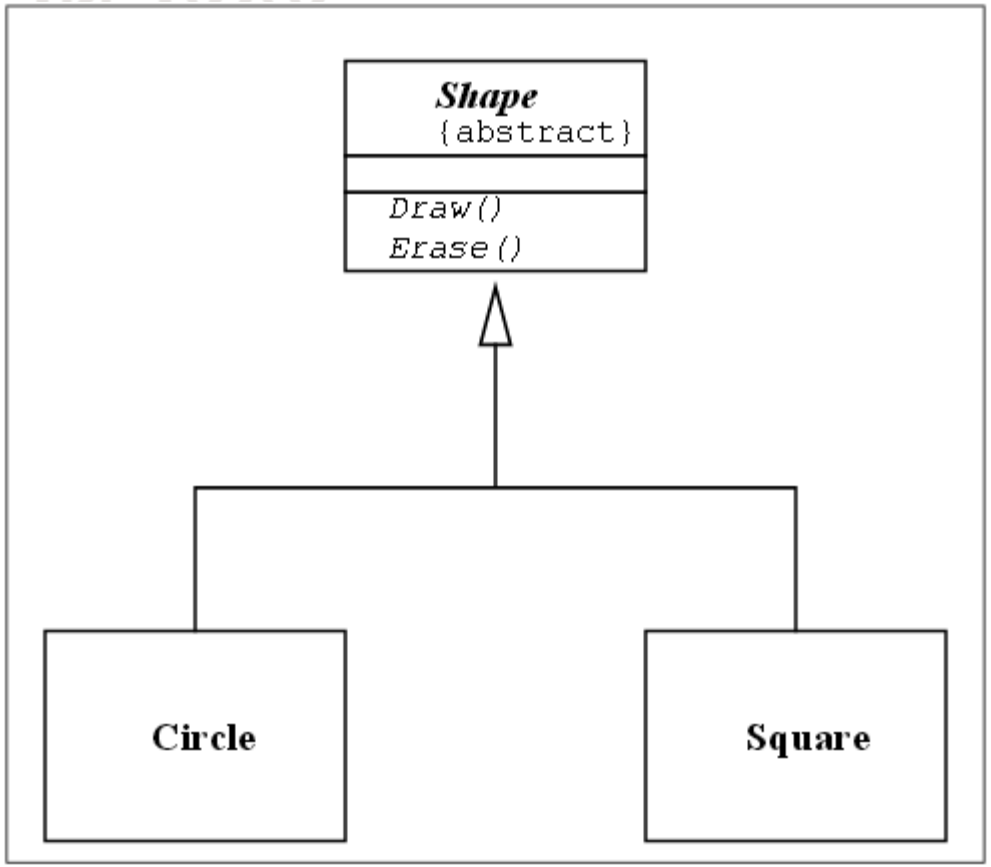
- Several specialized class types can be shown.
  - *Leaf* classes cannot have any descendants.
    - property {leaf} near the name.
  - *Utility classes* provide common functionality. It has only class scoped static attributes and operations. It usually has no instances.
    - property {utility} near the name.
  - *Root* doesn't have parent class.
    - property {root} near the name.
  - *Abstract* classes serve as templates for their descendants (but cannot be instantiated).
    - *Name in italics.*

ABSTRACT CLASSES

- Defer the implementation of one or more operations
- Have no direct instances
- Serve to define the “interface” to an abstraction



INHERITANCE IN JAVA



- Write the code for shape and circle classes in Java

```
abstract class Shape
{       private int x; private int y;
        public abstract void Draw()
        {
        //this method mustn't be implemented here.
        //If we do implement it, the result is a Syntax Error.
        }
}
 class Circle extends Shape {
{
//here we should provide an implementation for Draw(int x, int y)
        public override void Draw()
        {       // Implement here
            System.out.println("Drawing a Circle at:(" + getX() + ", " + getY() +");
        }
}
```

CLASS DIAGRAMS

- A class diagram is:
  - a collection of class structures;
  - the relationships between them (next lecture).