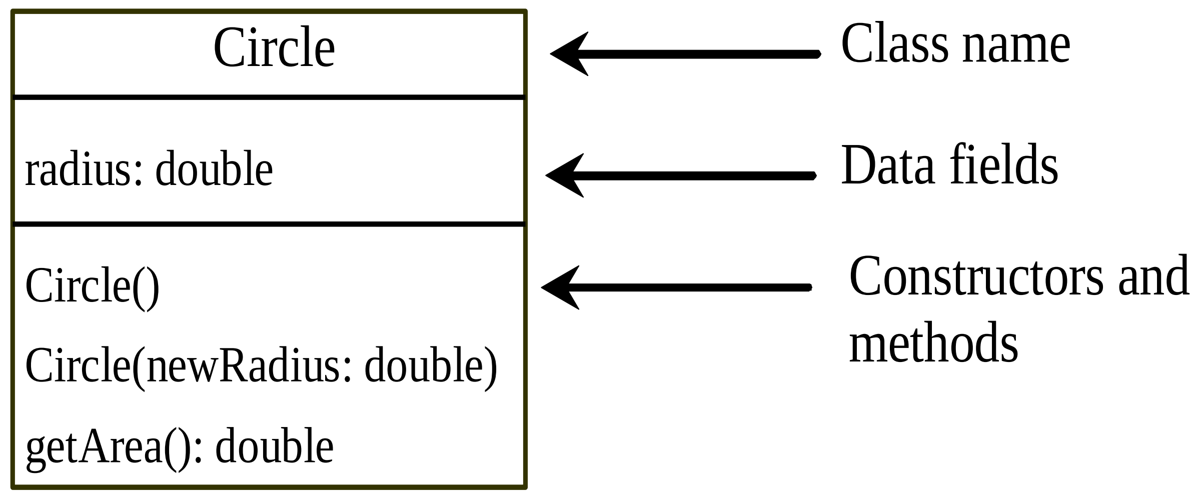
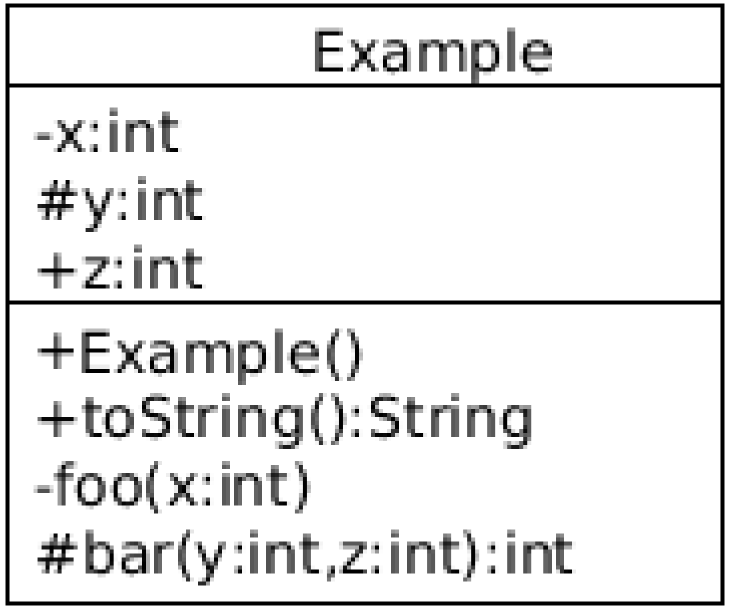
UML Class Diagram



Visibility Modifiers

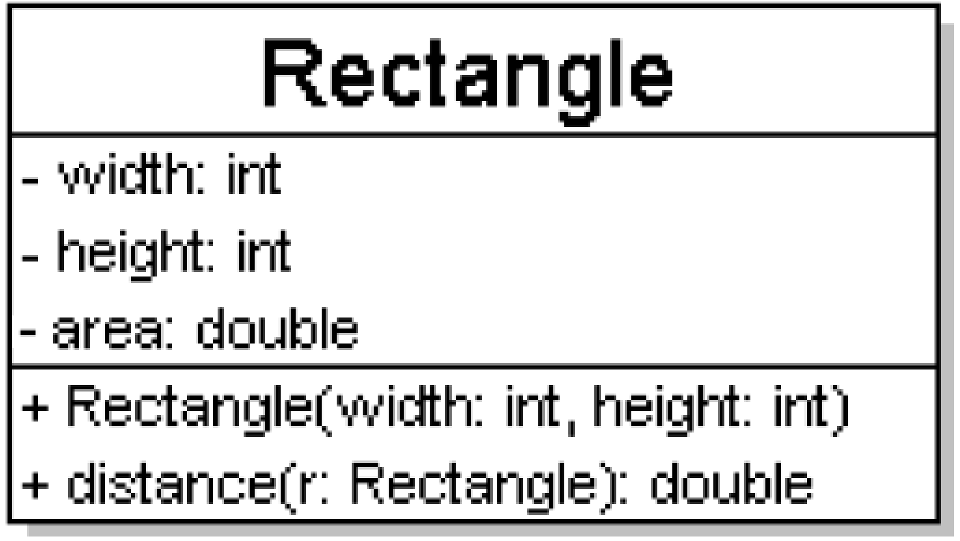
- The sign + indicates public
- The sign – indicates private
- The sign # indicates protected
- The sign ~ default

Example 1



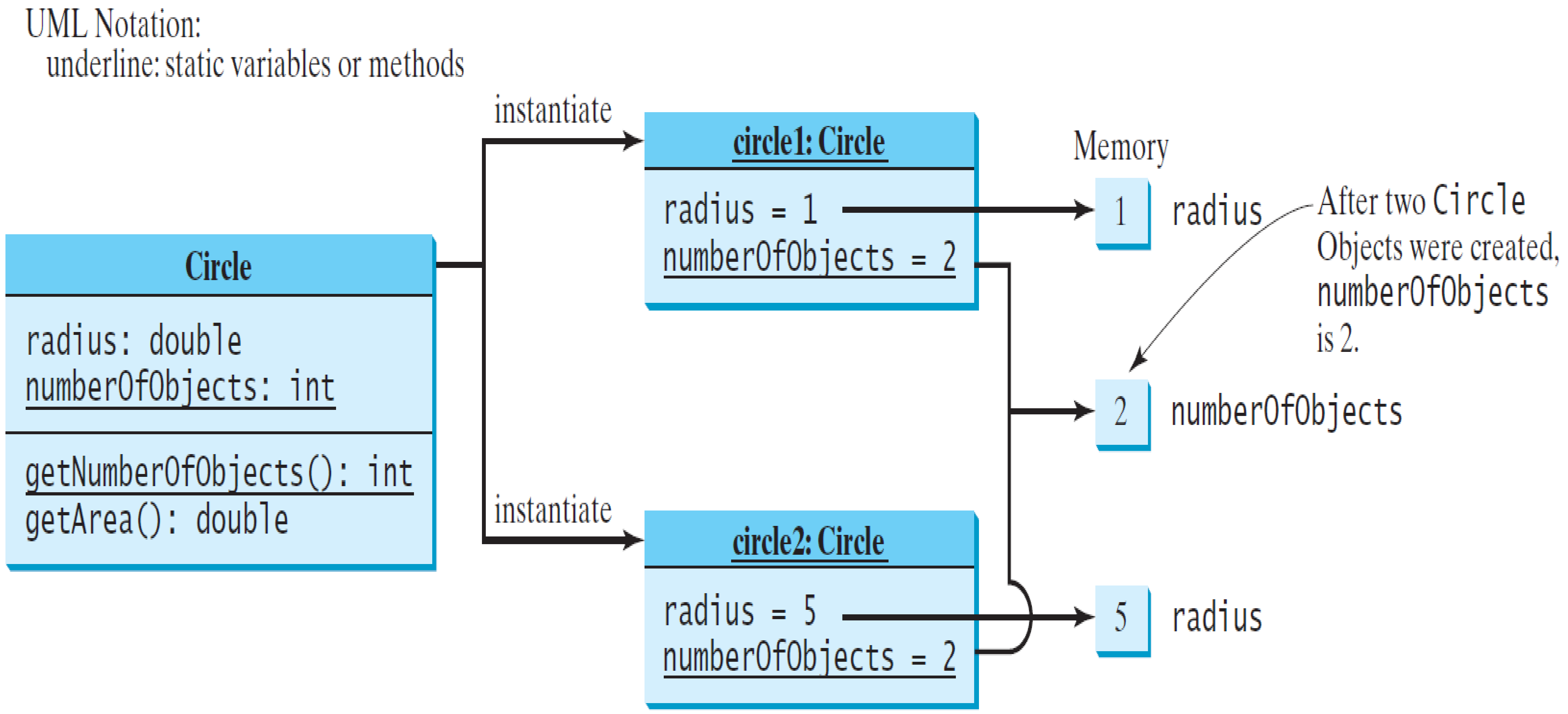
```
public class Example {
  private int x;
  protected int y;
  public int z;
  public Example() { ... }
  public String toString() { ... }
  private void foo(int x) { ... }
  protected int bar(int y, int z) { ... }
}
```

Example 2

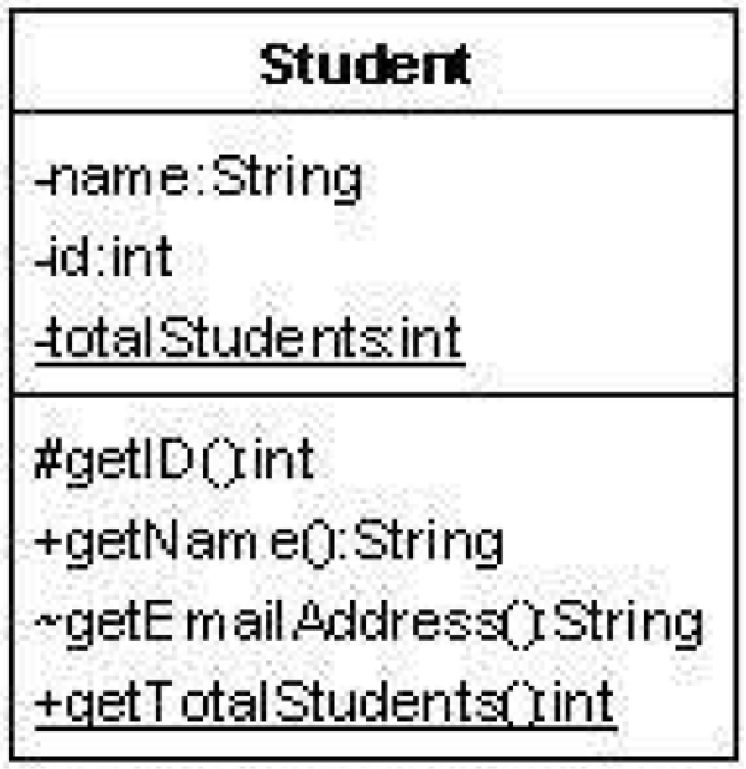

Rectangle
- width: int
- height: int
- area: double
+ Rectangle(width: int, height: int)
+ distance(r: Rectangle): double

What is the corresponding java code?

Static Variables, and Methods



Example 1



```
class Student{
        ...
        private static int totalStudents;
        ....
        public static int getTotalStudents()
                {
                ...
                }
 ..........................
}
```

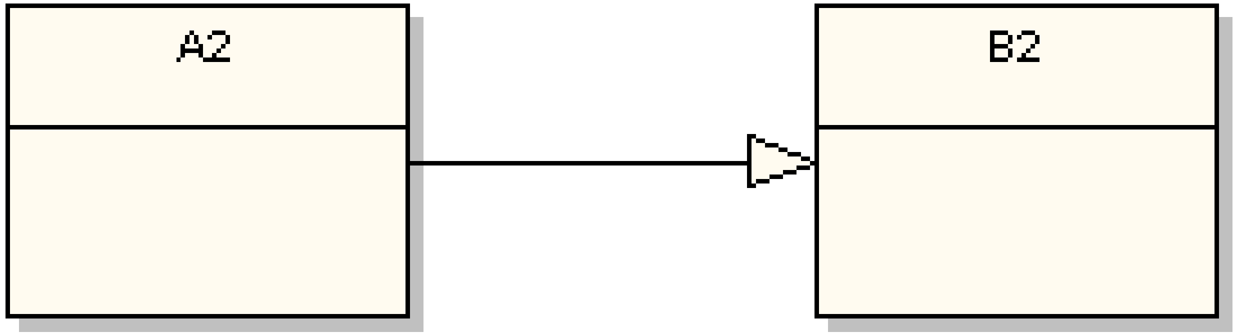
Relationships

- Generalization
    - Inheritance
        - “Is-a” relationship
        - Between classes or classes and interfaces
        - One class/interface is a specialized version of another.
        - Example, Honda is a type of car. So the class Honda would have a generalization relationship with the class car.

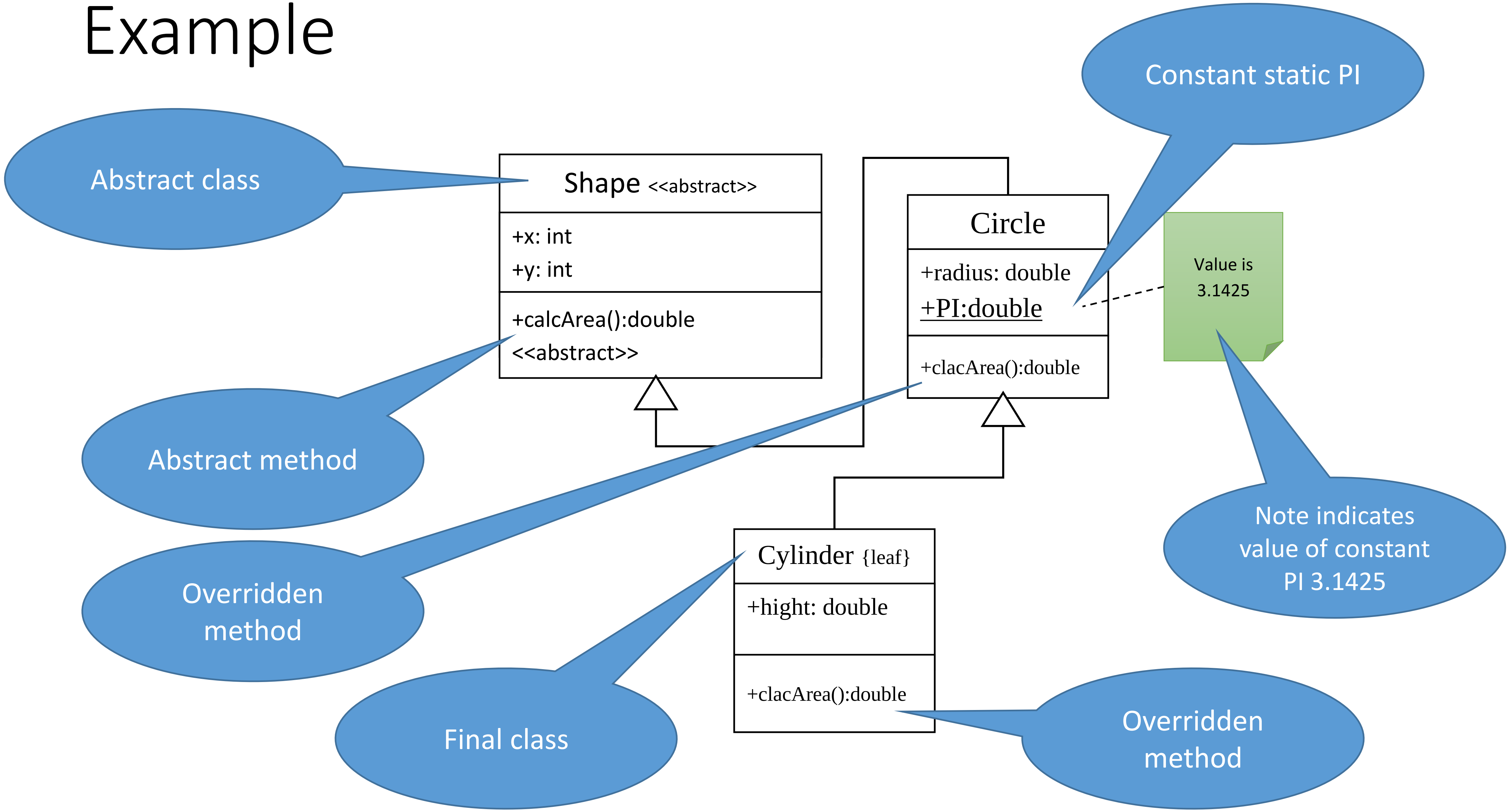
Generalization: between 2 classes

“**Is-a**” relationship

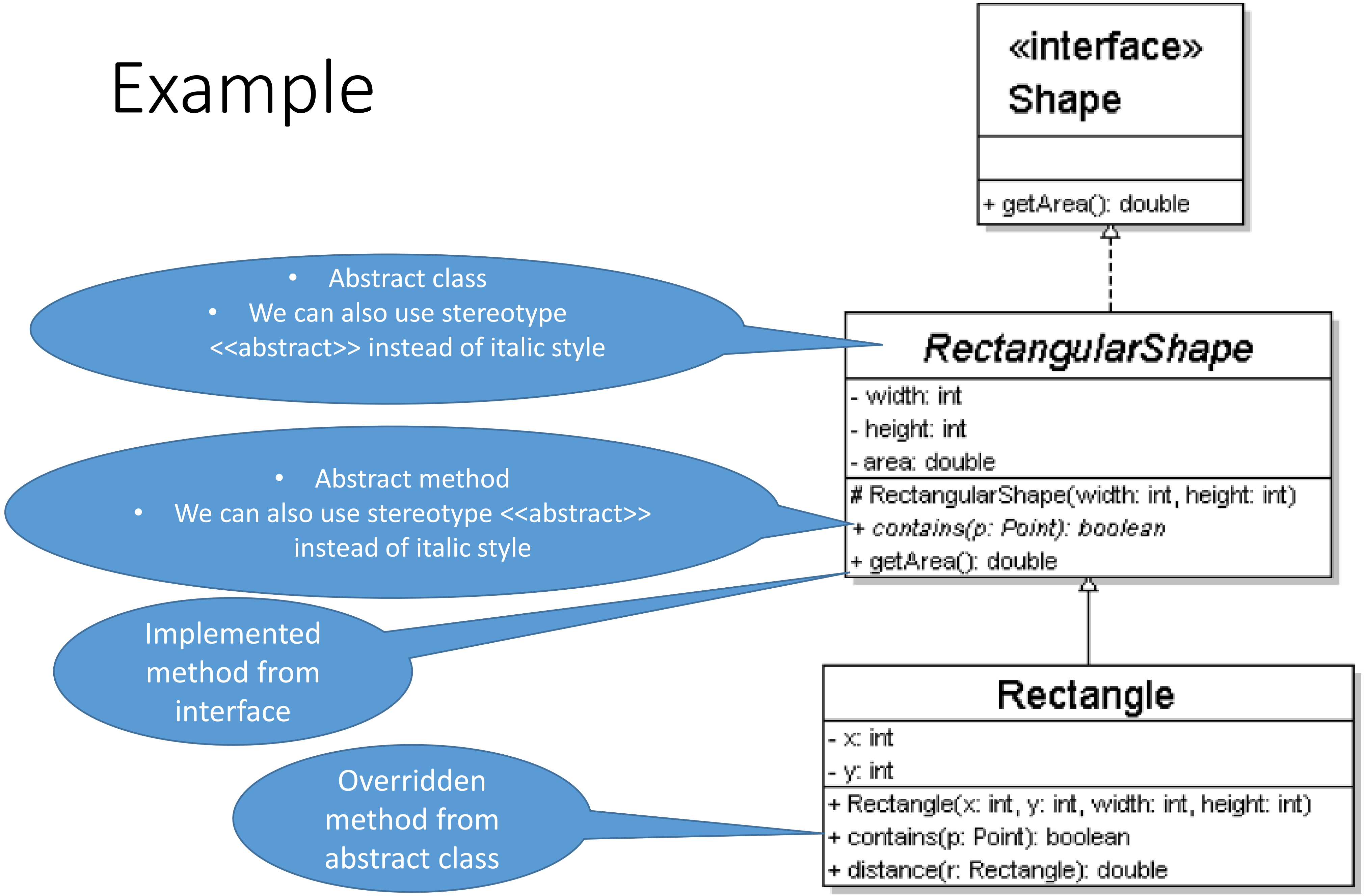
```
class A2 extends B2{
………

}
```



Example


Abstract class
Shape <<abstract>>
+x: int
+y: int
+calcArea():double
<<abstract>>
Abstract method
Circle
+radius: double
+PI:double
+clacArea():double
Constant static PI
Value is 3.1425
Note indicates value of constant PI 3.1425
Overridden method
Cylinder {leaf}
+hight: double
+clacArea():double
Final class
Overridden method

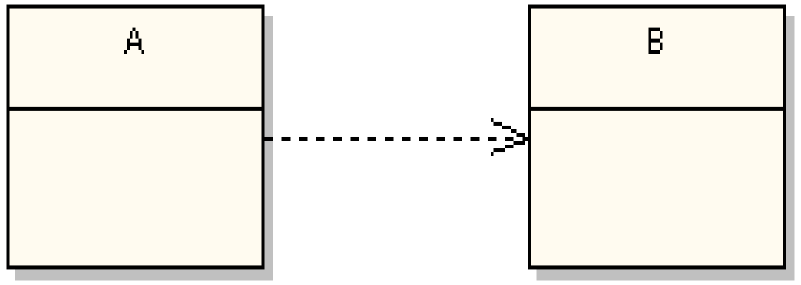
Example


«interface» Shape
+ getArea(): double
RectangularShape
- width: int
- height: int
- area: double
# RectangularShape(width: int, height: int)
+ *contains(p: Point): boolean*
+ getArea(): double
Rectangle
- x: int
- y: int
+ Rectangle(x: int, y: int, width: int, height: int)
+ contains(p: Point): boolean
+ distance(r: Rectangle): double
• Abstract class
• We can also use stereotype <<abstract>> instead of italic style
• Abstract method
• We can also use stereotype <<abstract>> instead of italic style
Implemented method from interface
Overridden method from abstract class

Dependency

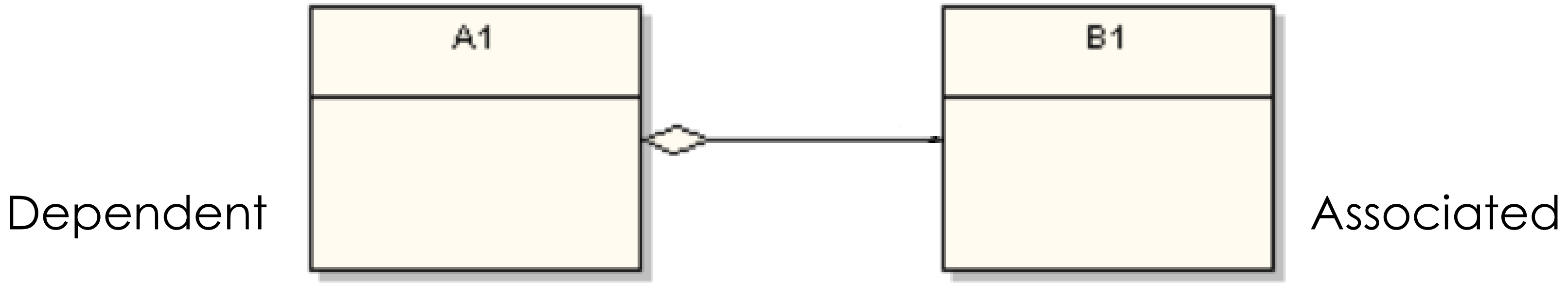
- Means the class at the source end of the relationship has some sort of dependency on the class at the target (arrowhead) end of the relationship
- Used to model temporarily-usage between things
- Symboled by a dashed arrow
- Do not over use in your design (very low importance)
- Example:

```
import B;
public class A {
                    public void method1(B b) { // . . . }

}
```

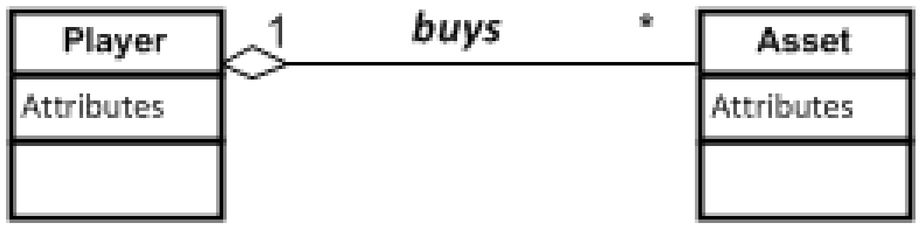


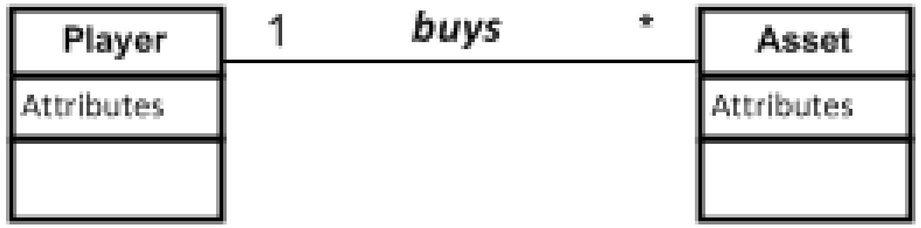
Aggregation

- Redundant form of association (represented as a solid line)
- Used to model “whole-part” relationships between things
- Defines the state of instances of the dependent class
- The dependent class (A1) holds a reference of the associated class (B1). The use of the specific instance of B1 is or may be shared among other aggregators.
- Therefore, if A1 goes out of scope (e.g. garbage collected), the instance of B1 does not go out of scope
- Symboled by a clear diamond at the dependent class side



Aggregation Example





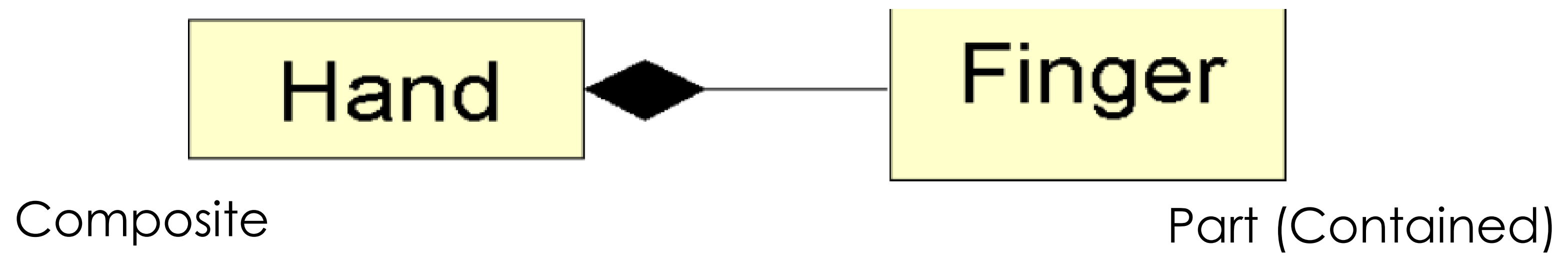
Aggregation

FYI: alternative way called Association

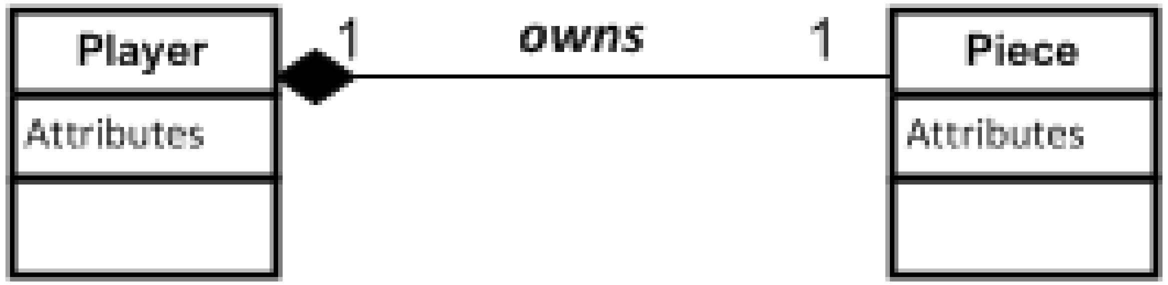
```
class Asset { … }
class Player {
List assets;
public void AddAsset(Asset newlyPurchasedAsset)
                        {
                        assets.Add(newlyPurchasedAssest);
             }
…  }
```

Composition

- A much stronger aggregation (or association) relation ship
- A “has a“ relationship from the dependent class view
- The whole is generally called the composite (we will refer to the parts as contained)
- The lifetime of the part is bound within the lifetime of the composite. i.e. if the composite object goes out of scope, then the contained object also goes out of scope
- Symboled by a filled diamond
- Example:



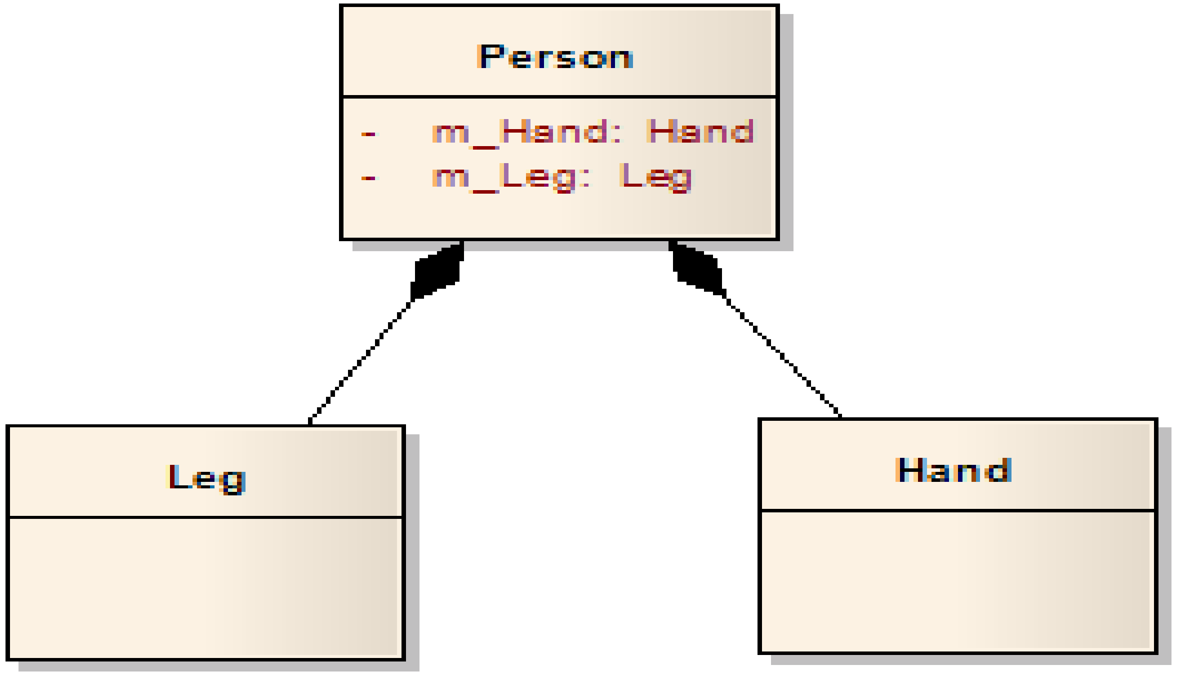
Composition



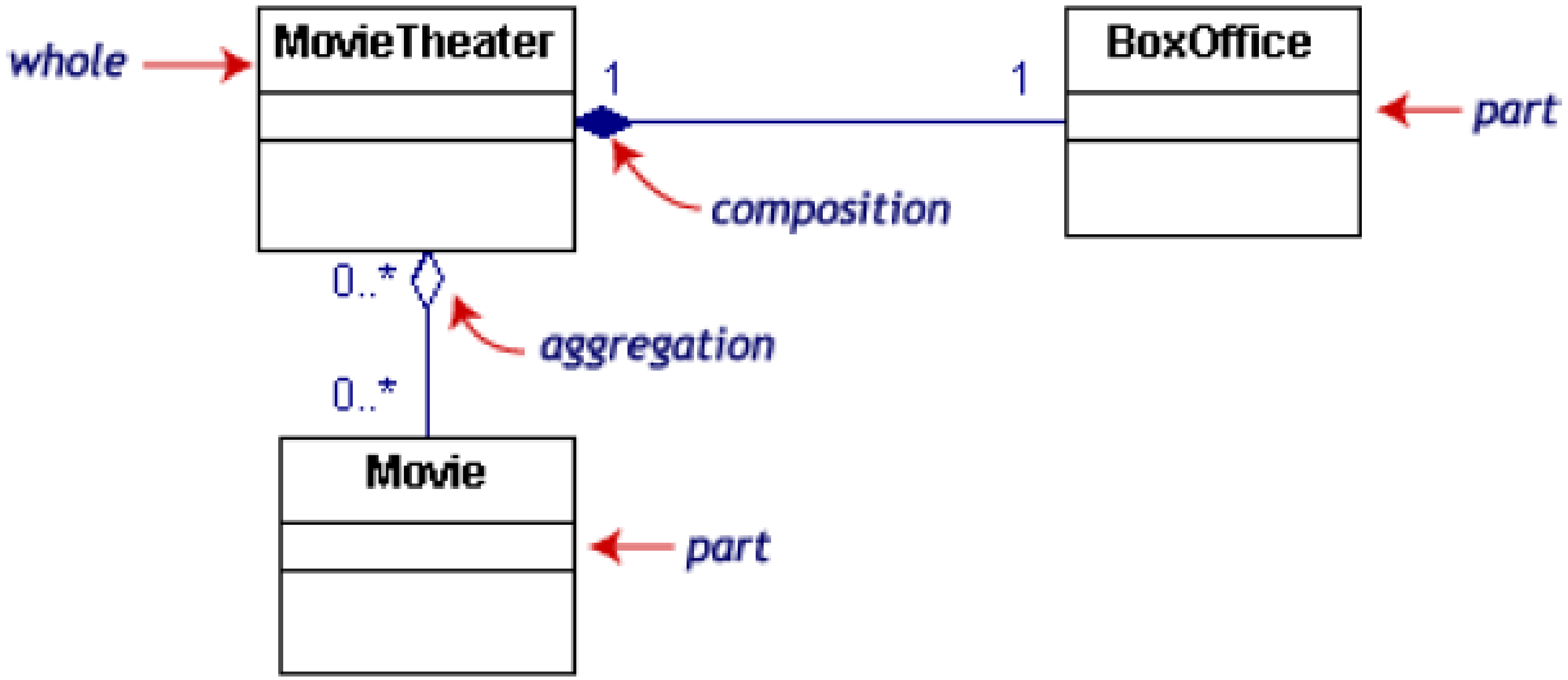
```
public class Piece { ... }
public class Player
{
Piece piece = new Piece(); /*Player owns the responsibility
of creating the Piece*/
...
}
```

Composition

- Example 2: a person has a hand and a leg
- You can work and complete the human structure
    - E.g. another hand & leg, head, abdomen …etc.



Aggregation and Composition Example



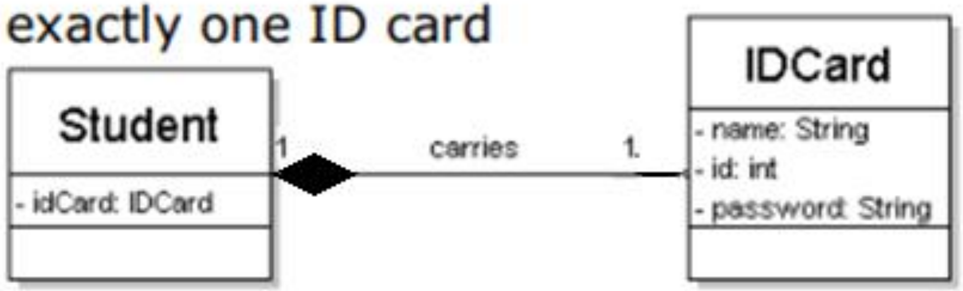
If the movie theater goes away

so does the box office => composition

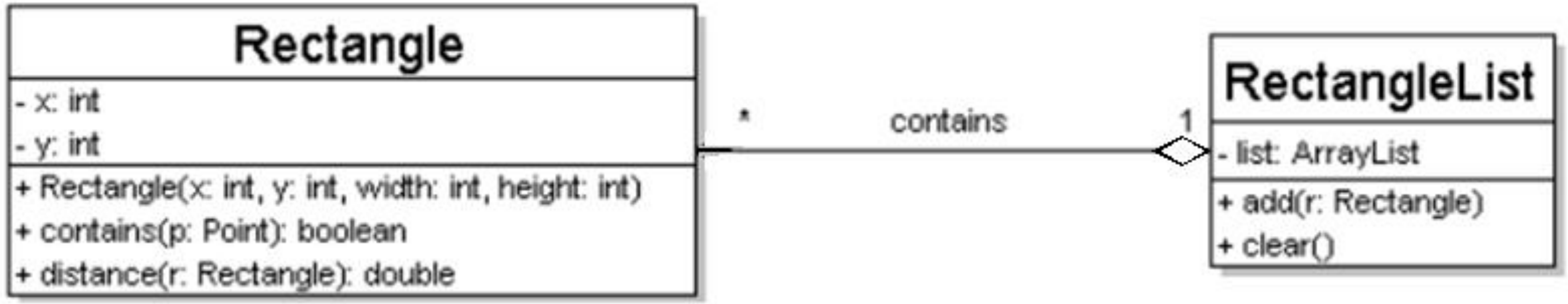
but movies may still exist => aggregation

Multiplicity

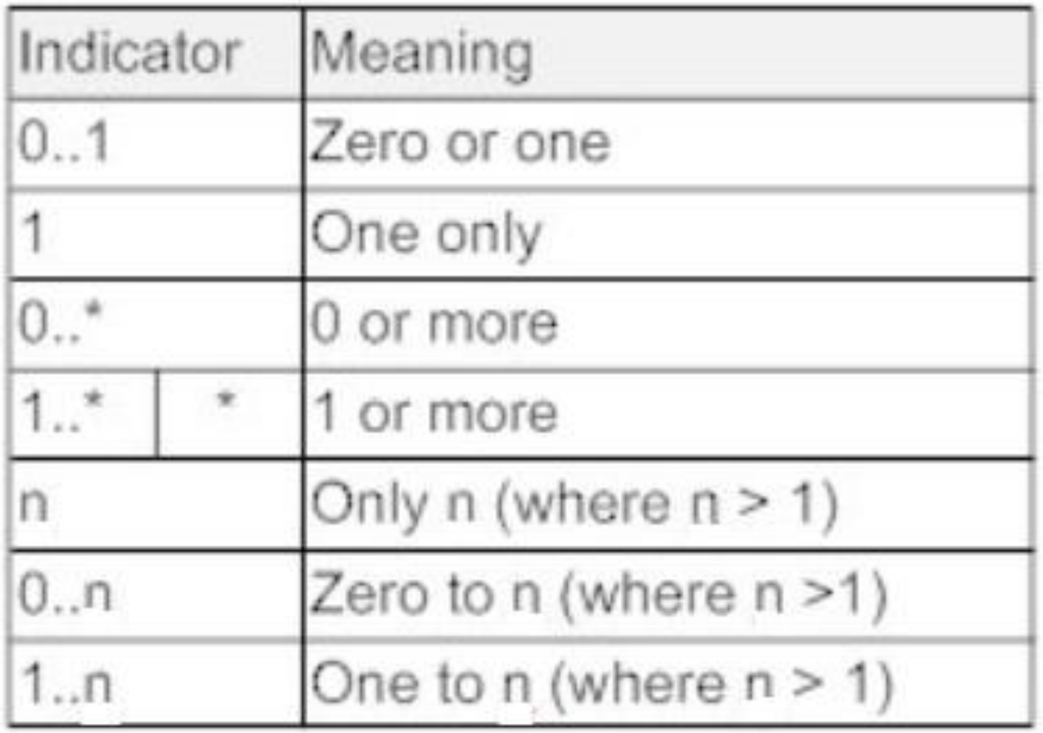
- one-to-one
  - each student must carry exactly one ID card



- one-to-many
  - one rectangle list can contain many rectangles


Rectangle
- x: int
- y: int
+ Rectangle(x: int, y: int, width: int, height: int)
+ contains(p: Point): boolean
+ distance(r: Rectangle): double
*
contains
1
RectangleList
- list: ArrayList
+ add(r: Rectangle)
+ clear()

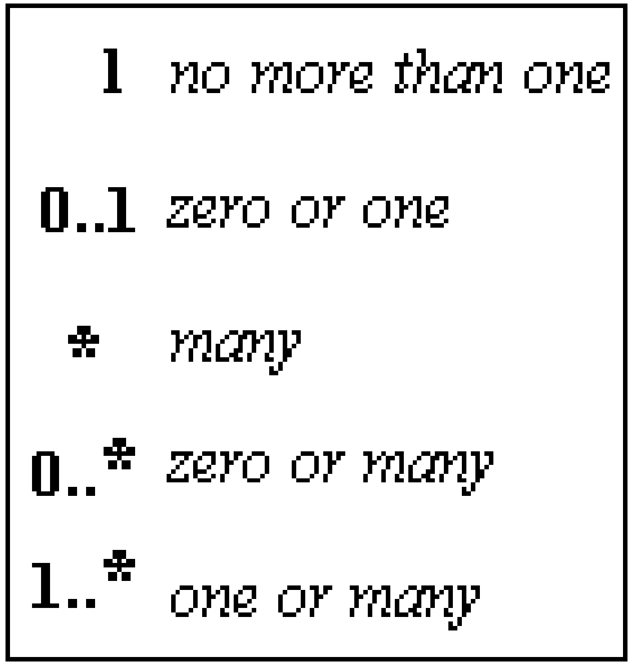
Multiplicity



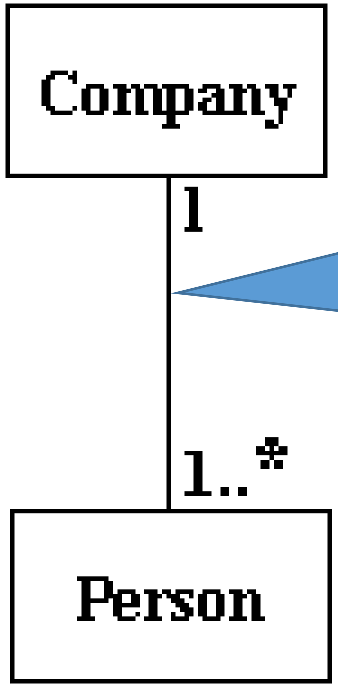
Indicator		Meaning
0..1		Zero or one
1		One only
0..*		0 or more
1..*	*	1 or more
n		Only n (where n > 1)
0..n		Zero to n (where n >1)
1..n		One to n (where n > 1)

You can use a number instead of n. e.g. 0..4 which means minimum zero & max 4

Example

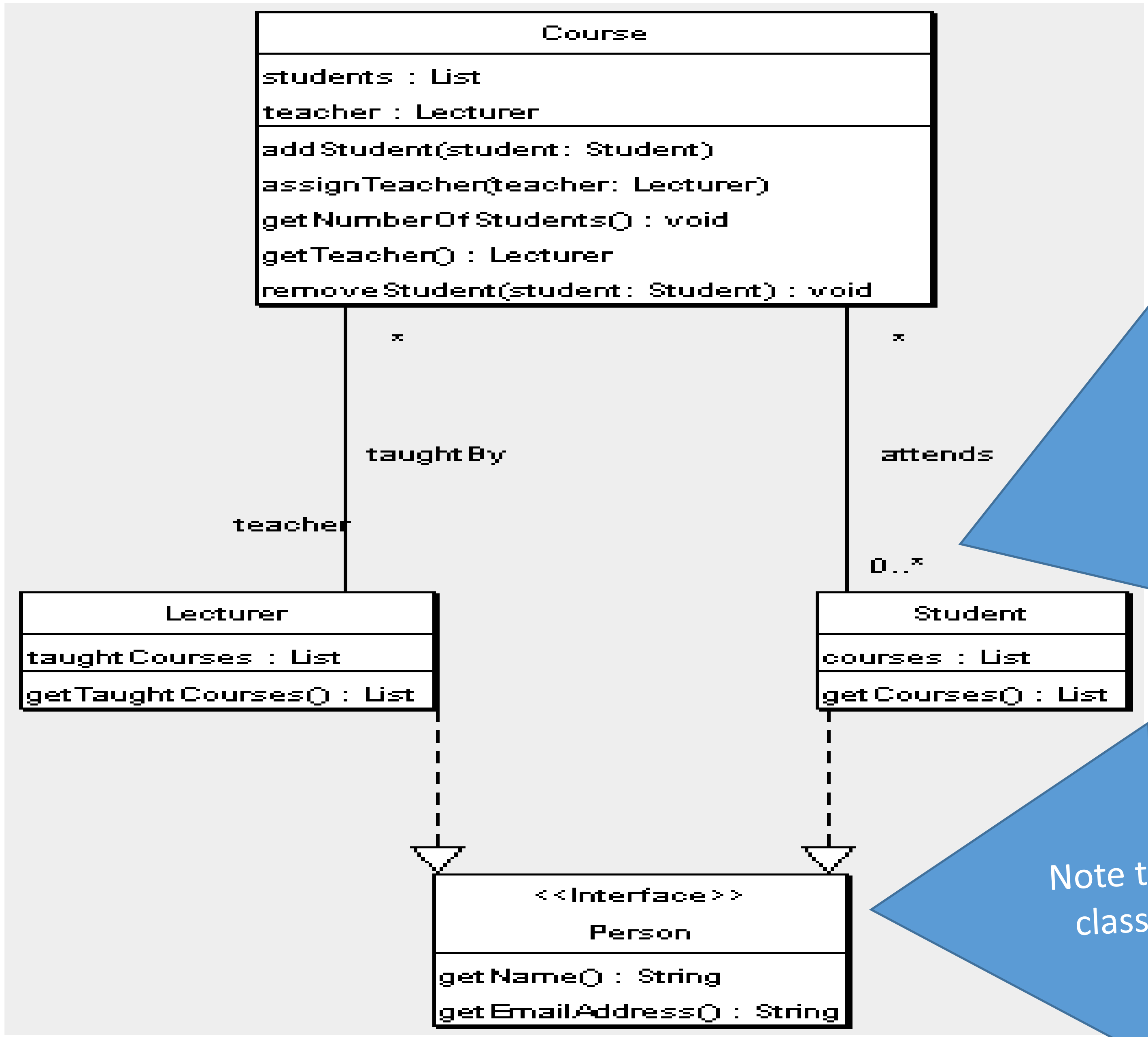


1	*no more than one*
0..1	*zero or one*
*	*many*
0..*	*zero or many*
1..*	*one or many*



What type of relationship can we use here?

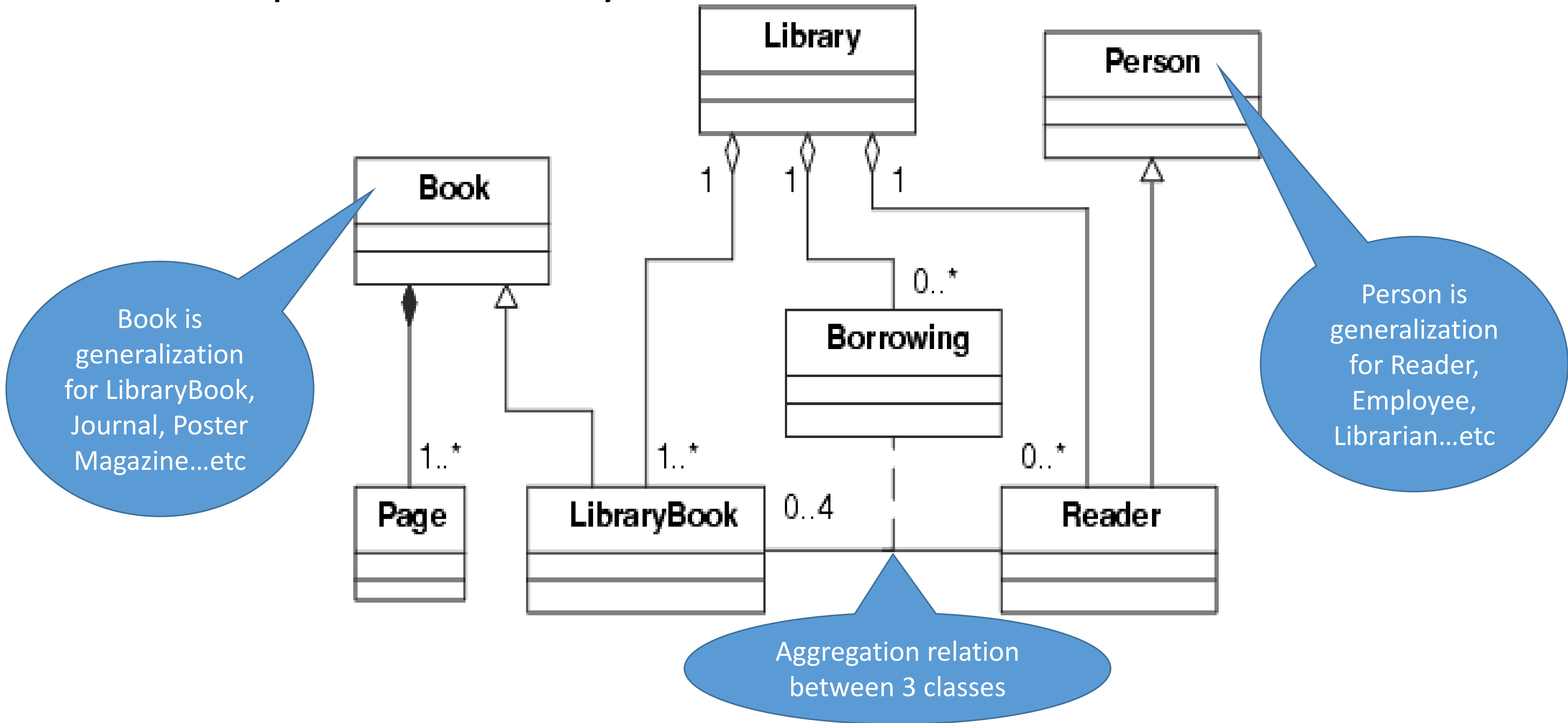
Example: Students, Lecturers & Courses



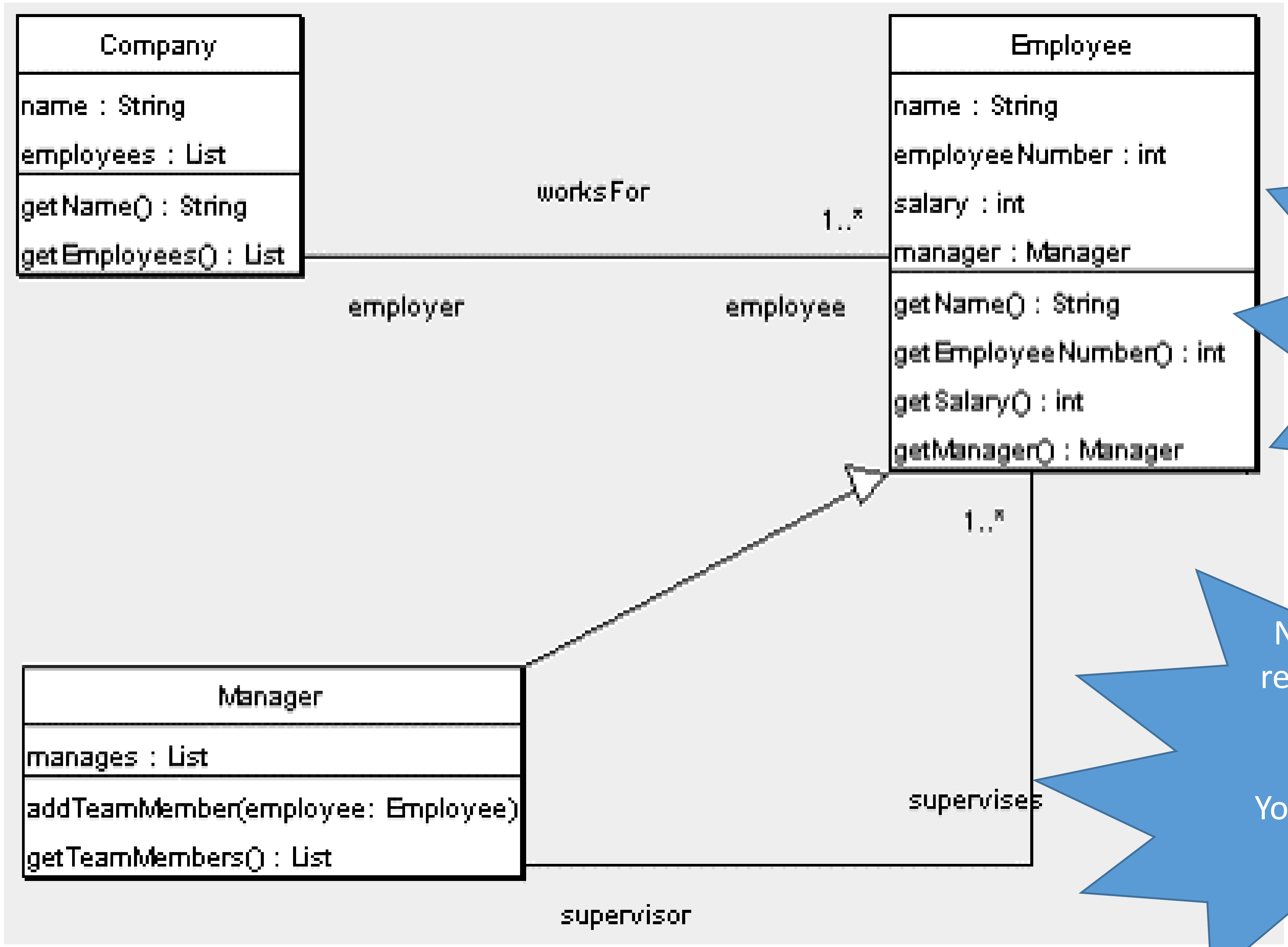
Note that you may have a dummy course with 0 students, hence 0..* cardinality.
e.g.1 closed course
e.g. 2 CE student taking a course in CS dept.

Note that you can still use a class instead of interface

Example: a library


Library
Person
Book
1
1
1
0..*
Borrowing
1..*
1..*
0..*
Page
LibraryBook
0..4
Reader
Book is generalization for LibraryBook, Journal, Poster Magazine…etc
Person is generalization for Reader, Employee, Librarian…etc
Aggregation relation between 3 classes

Example: Company, Employee & Manager



Add ref. to company obj in Employee class to keep track of Employee's Company

Note you have 2 relations between Manager and Employee
You can have more if needed