UTM
UNIVERSITI TEKNOLOGI MALAYSIA

7: ASSOCIATION, AGGREGRATION & COMPOSITION

Programming Technique II

(SCSJ1023)

Adapted from Tony Gaddis and Barret Krupnow (2016), Starting out with C++: From Control Structures through Objects

Associations

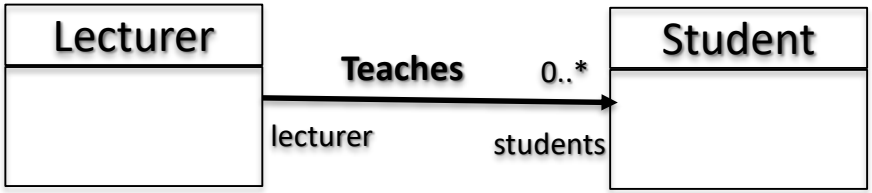
Introduction to Associations

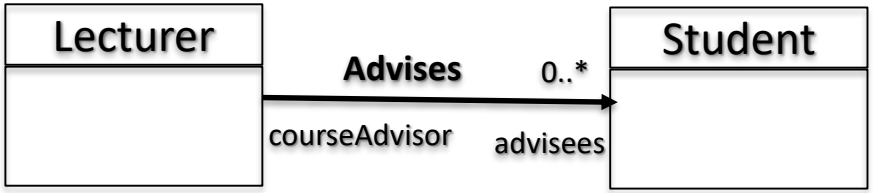
- **Association**: Indicates relationships between classes through their objects

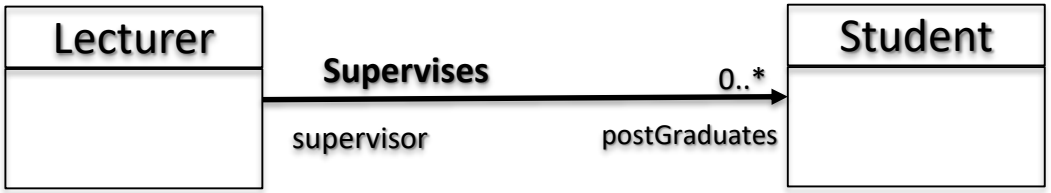
- Association can be, one to one, one to many, many to one, or many to many relationships.

Introduction to Associations

Example:







Aggregations

Introduction to Aggregations

- **Aggregation**: a special type of association which is **one way** relationship.
- It models a **'has a'** relationship between classes – enclosing class 'has a' enclosed class.
- The **existence** of the objects (enclosing and enclosed) are **independent**.

Introduction to Aggregations

Example:

- A "has a" relationship: *Wallet has Money*
- Independence: However, money does not necessarily need to have Wallet.
- Money objects can be created even though there is no Wallet object.

Implementations of Aggregations

- An **aggregation** relationship is implemented by objects contain **pointers** to other objects.
  - it is because the existence of the enclosing and enclosed objects are independent.
  - If one party is destroyed, the other party still exists.
  - the relationship between objects can be broken by only disconnecting the pointer.

Example:

Consider course registration system.

A student may enroll to a course. Assuming that the student quits from his or her study. In this case, his or her record will be removed from the system. However, the course needs to remain as other students are still enrolled to the course.

Relationship between students and courses should be done by an aggregation.

Implementations of Aggregations

- Aggregation - declare attributes as object pointers.
- The pointers can be set to NULL to represent no object.

```
class Student
{ private:
       Course *course;
  public:
      Student(){course = NULL;}
      void enrollCourse(Course *);
      void withdrawCourse();
..};
```

Implementations of Aggregations

```
void Student::enrollCourse(Course* c){
  course = c;
}

void Student::withdrawSubject(){
     course=NULL;
}
```

The course should be passed rather than the object student create it itself, as the course is shared with other students.

..and when a student withdraw the course,
we just need to break the link/pointing, and not to "delete" the course.
Other students might still be using the course.

Implementations of Aggregations

```
int main()
{
  Course c1("SCSJ1023","Prog. Tech. 2");
  Student s1 = new Student;
  Student s2, s3;

  s1->enrollCourse(&c1);
  s2.enrollCourse(&c1);
  s3.enrollCourse(&c1);
  ….
  delete s1;
}
```

A course can be created independently as it does not belong to any student.

Pass course as a pointer

The student is deleted from the system as he or she has dropped out from the study. However, the course c1 still exists.

Compositions

Introduction to Compositions

- **Composition**: a restricted version of aggregation in which the enclosing and enclosed objects are highly dependent on each other.

- The existence of the enclosed objects are determined by the enclosing objects.

- It models a whole/part relationship with a strong ownership; when the whole dies, the part does so as well
  - enclosing objects(whole) **'has / contains/ consists of'** enclosed objects (parts)
  - Alternatively: the enclosed object is "part of" the enclosing object.

- An object can be only part of one whole object
- The **whole** object is responsible for creation and destruction of its part(s)

Introduction to Compositions

Example:

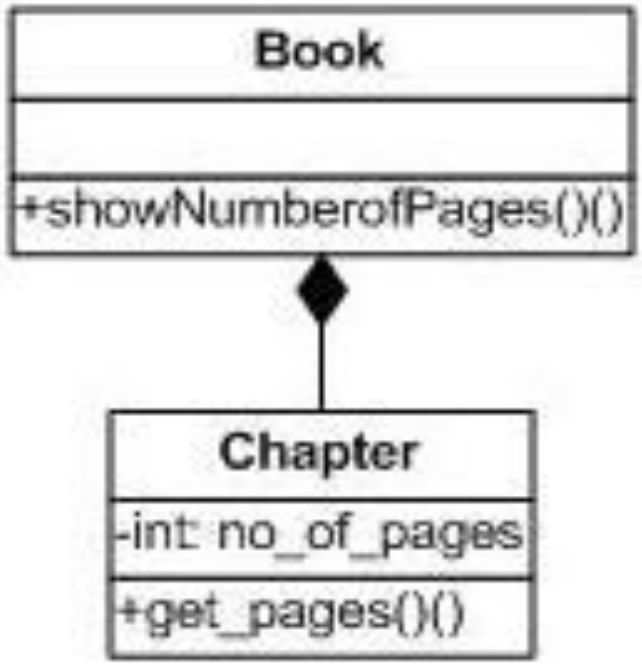
- A "whole-part" relationship: *human Body "consists of"/"has" Heart, or Heart "is part" of human Body.*

- Dependence: A human body needs heart to live and a heart needs a human body to survive. If one dies then another one too.
- If the Body object is destroyed then Heart object also will be gone.
- When a Body object is created, the Heart object is also created.

Implementations of Compositions

- Composition is implemented by the **nested objects**, i.e., whole objects contain the part objects.
  - it is because the existence of the whole (enclosing) and part (enclosed) objects are dependent.

Example:



Implementations of Compositions

```
class Chapter{
    int nPages;
  public:
    Chapter(int pages);
    int getPages();
};
Chapter::Chapter(int pages){
  nPages = pages;
}

int Chapter::getPages(){
  return nPages;
}
```

Constructor for **Chapter** class takes an integer argument to define its pages

```
class Book{
  public:
    Chapter c1, c2, c3, c4;
    Book(int pages1, int pages2, int pages3,
         int pages4):c1(pages1), c2(pages2),
                                 c3(pages3),c4(pages4){ }
     void showNumberofPages();
};
void Book:: showNumberofPages(){
   cout<<"Total number of pages" <<
   (c1.getPages() + c2.getPages()+
    c3.getPages() + c4.getPages());
}
int main(){
   Book pt2(200, 190, 50, 100);
   pt2.showNumberofPages();
}
```

Class Book includes objects of class Chapter

Note use of colon(:) operator to initialize the enclosed objects using their constructors

The methods of the enclosed objects can be called to as usual.