

Object Oriented Programming (CS1143)

Week 11

Department of Computer Science

Capital University of Science and Technology (CUST)

Outline

- Relationships in Object Oriented Programming
- Association
- Aggregation
- Composition
- Dependency

Association

Association

- Not all relationships between classes can be defined as inheritance.
- We encounter classes in object-oriented programming that have other types of relationships with each other.
- For example, we can define a class named Person and another class named Address. An object of the type Person may be related to an object of type Address: A person lives in an address and the address is occupied by a person.
- The Address class is not inherited from the Person class; neither the other way.
- In other words, a person is not an address; an address is not a person.
- We cannot define either class as a subclass of the other one.

Association Diagram

- A relationship of this type is shown in UML diagrams as a solid line between two classes.
- An association diagram also shows the type of relationship between the classes.
- This is shown by an arrow and text in the direction of the corresponding class.
- Another piece of information represented in an association diagram is multiplicity.
- Multiplicity defines the number of objects that take part in the association.
- Multiplicity is shown at the end of the line next to the class.
- Figure on the next slide shows that one person can have only one address, but one address can belong to any number of occupants.

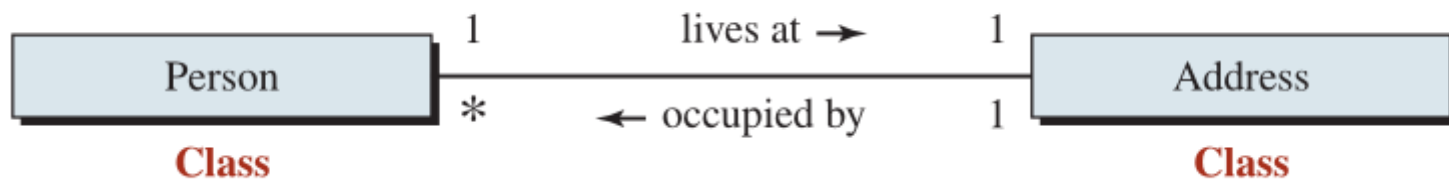
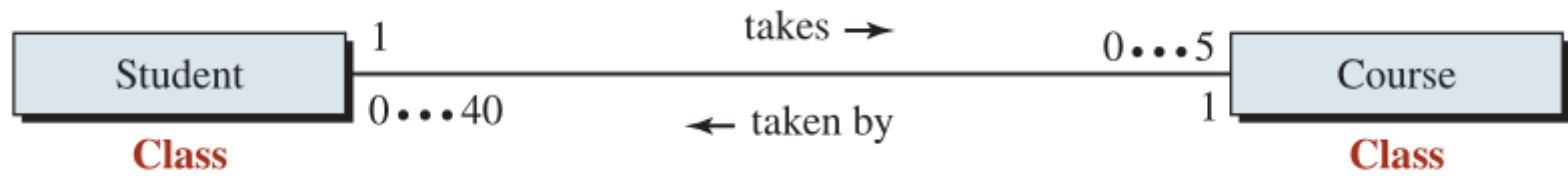


Table 11.2 Multiplicity in association diagrams

Key	Interpretation
n	Exactly n objects
$*$	Any number of objects including none
$0 \dots 1$	Zero or one object
$n \dots m$	A range from n to m objects
n, m	n or m objects

Example

- We can define two classes: Course and Student.
- If we assume that at each semester a student can take up to five courses, and a course can be taken by up to forty students, we can depict these relationships in an association diagram, as shown next.
- This is a many-to-many relationship.
- An association representing a many-to-many relationship cannot be implemented directly because it creates an infinite number of objects in the program (circular relationship).
- Usually this type of association is implemented in a such a way that this circular infinity is avoided.
- For example, a Student object can have a list of five course names (not a complete Course object), and a Course object can have a list of forty student names (not a complete Student object)



Aggregation

Aggregation

- Aggregation is a special kind of association in which the relationship involves ownership. It models the “has-a” relationship.
- One class is called an aggregator and the other an aggregatee.
- An object of the aggregator class contains one or more objects of the aggregatee class.
- The symbol for aggregation is a hollow diamond placed at the site of the aggregator.
- The aggregation relationship is one-way.
- For example a Person can have a birth date (an object of the class Date), but a Date object can be related to multiple events, not only the birth date of a person

An aggregation is a one-to-many relationship from the aggregator to the aggregatee.

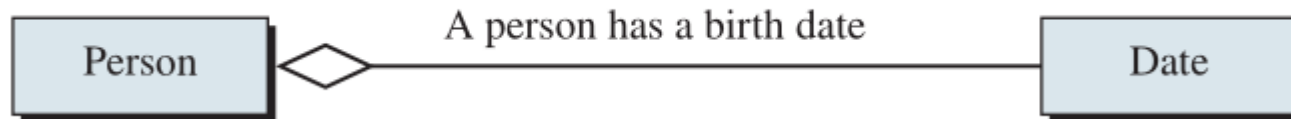
Aggregation Contd.

- The life of an aggregatee is independent of the life of its aggregator.
- An aggregatee may be instantiated before the instantiation of the aggregator and may live after it.

In an aggregation, the lifetime of the aggregatee is independent of the lifetime of the aggregator.

Example

- We create a Person class and a Date class.
- The Date class is independent and can be used to represent any event.
- The Person class uses an object of the Date class to define the birthday of a person.



```
1  #include <iostream>
2  using namespace std;
3
4  class Date
5  {
6      private:
7          int month;
8          int day;
9          int year;
10     public:
11         Date (int month, int day, int year);
12         ~Date ();
13         void print() const;
14 };
```

```

16 Date :: Date (int m, int d, int y)
17 : month (m), day (d), year (y)
18 {
19     if ((month < 1) || (month > 12))
20     {
21         cout << "Month is out of range. ";
22     }
23
24     int daysInMonths [13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
25     if ((day < 1) || (day > daysInMonths [month]))
26     {
27         cout << "Day out of range! ";
28     }
29
30     if ((year < 1900) || (year > 2099))
31     {
32         cout << "Year out of range! " ;
33     }
34 }

```

```
36 Date :: ~Date ()
37 {
38 }
39
40 void Date :: print() const
41 {
42     cout << month << "/" << day << "/" << year << endl;
43 }
```



```
46 class Person
47 {
48     private:
49         long identity;
50         Date birthDate;
51     public:
52         Person (long identity, Date birthDate);
53         ~Person ( );
54         void print ( ) const;
55 };
56
57 Person :: Person (long id, Date bd)
58 : identity (id), birthDate (bd)
59 {
60 }
```

```
62 Person :: ~Person ( )
63 {
64 }
65
66 void Person :: print ( ) const
67 {
68     cout << "Person's Identity: " << identity << endl;
69     cout << "Person's date of birth: ";
70     birthDate.print();
71     cout << endl;
72 }
```

```
75 int main ( )
76 {
77     Date date1 (5, 6, 1980);
78     Person person1 (111111456, date1);
79
80     Date date2 (4, 23, 1978);
81     Person person2 (345332446, date2);
82
83     person1.print ( );
84     person2.print ( );
85
86
87     return 0;
88 }
```

D:\Work Umair\4_CUST (1-9-22)\1_Teaching\2_ACS1143-OOP\Practice Programs\W11-P1.exe

Person's Identity: 111111456
Person's date of birth: 5/6/1980

Person's Identity: 345332446
Person's date of birth: 4/23/1978

Composition

Composition

- Composition is a special kind of aggregation in which the lifetime of the containee depends on the lifetime of the container.
- For example, the relationship between a person and her name is an example of composition.
- The name cannot exist without being the name of a person.
- The symbol for composition is a solid diamond placed at the side of the composer.
- In the case of the relationship between the Employee and the Name, the composer is the Employee.



Composition Contd.

- The distinction between aggregation and composition is normally conceptual; it depends on how the designer thinks about the relationship.
- For example, a designer may think that a name must always belong to a person and cannot have a life of its own.
- Another designer may think that a name continues to exist even if the person dies.
- The distinction also depends on the environment in which we are designing our classes.
- For example, in a car factory, the relationship between a car and its engine is composition; we cannot use the engine without installing it in a car. In an engine factory, each engine has its own life cycle

Example

- We create an employee class in which an employee object has two data members: salary and name.
- The name itself is an object of a class with three fields: first name, initial, and last name.

```
1 #include <string>
2 #include <iostream>
3 #include <cassert>
4 using namespace std;
5 class Name
6 {
7     private:
8         string first;
9         string init;
10        string last;
11
12    public:
13        Name (string first, string init, string last);
14        ~Name ( );
15        void print ( ) const;
16 };
```



```

18 // Constructor
19 Name :: Name (string fst, string i, string lst)
20 :first (fst), init (i), last (lst)
21 {
22     toupper (first[0]);
23     toupper (init [0]);
24     toupper (last[0]);
25 }
26
27 // Destructor
28 Name :: ~Name ( )
29 {
30 }
31
32 // Print member function
33 void Name :: print ( ) const
34 {
35     cout << "Employee name: " << first << " " << init << ". " << last << endl;
36 }

```


```

40 class Employee
41 {
42     private:
43         Name name;
44         double salary;
45
46     public:
47         Employee (string first, string init, string last,
48             double salary);
49         ~Employee ( );
50         void print ( ) const;
51 };
52
53 Employee :: Employee (string fst, string i, string lst, double sal)
54 : name (fst, i, lst), salary (sal)
55 {
56 }

```

```
58 // Destructor
59 Employee :: ~Employee ( )
60 {
61 }
62
63 // Print member function
64 void Employee :: print ( ) const
65 {
66     name.print();
67     cout << "Salary: " << salary << endl << endl;
68 }
```

```
70 int main ( )
71 {
72     // Instantiation
73     Employee employee1 ("Mary", "B", "White", 22120.00);
74     Employee employee2 ("William", "S", "Black", 46700.00);
75     Employee employee3 ("Ryan", "A", "Brown", 12500.00);
76     // Output
77     employee1.print ( );
78     employee2.print ( );
79     employee3.print ( );
80     return 0;
81 }
```

 D:\Work Umair\4_CUST (1-9-22)\1_Teaching\2_ACS1143-OOP\Practice Programs\W11-P2.exe

Employee name: Mary B. White
Salary: 22120

Employee name: William S. Black
Salary: 46700

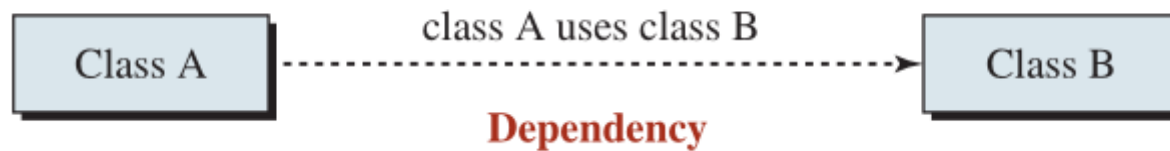
Employee name: Ryan A. Brown
Salary: 12500

Dependency

Dependency

- The third type of relationship that we can define between two classes is dependency.
- Dependency is a weaker relationship than inheritance or association.
- Dependency models the “uses” relationship. Class A depends on class B if class A somehow uses class B.
- In other words, class A depends on class B if A cannot perform its complete task without knowing that class B exists.
- This happens when
 - Class A uses an object of type B as a parameter in a member function.
 - Class A has a member function that returns an object of type B.
 - Class A has a member function that has a local variable of type B.

UML Class Diagram for Dependency

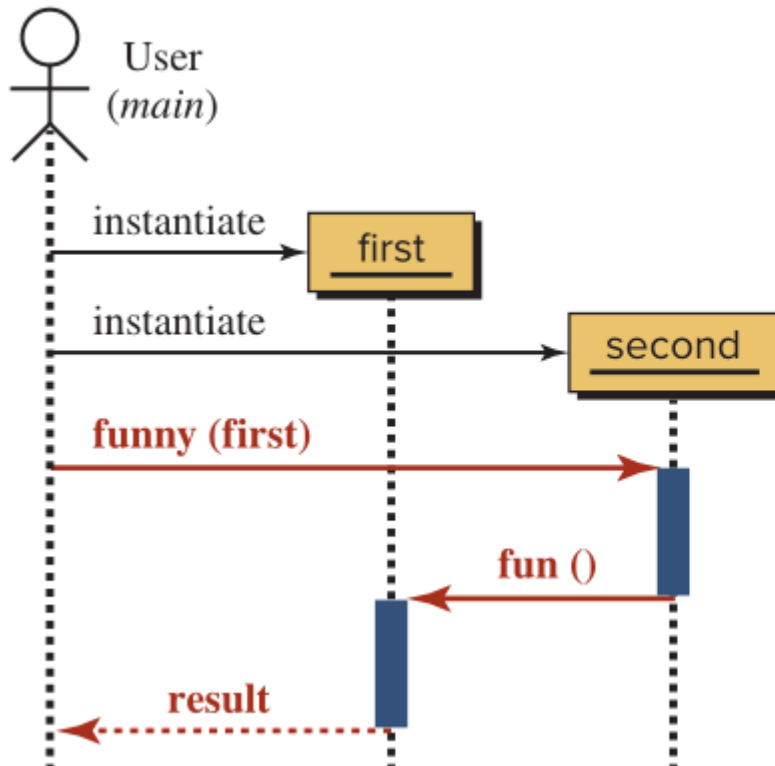


UML Sequence Diagram

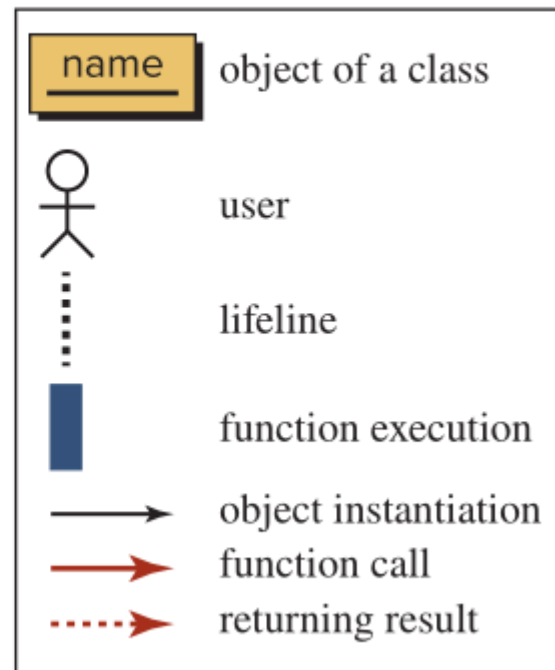
- A sequence diagram shows the interaction between objects.
- The main function and each object have lifelines that show the passing of time.
- The objects can be instantiated and their member function can be called.

Example

- We have two classes, First and Second. Class First has a member function called `fun()` that the user cannot call directly in the application.
- We want to use another function in the Second class called `funny()` to call the function `fun()` in the First class.
- We must pass an instance of the First class inside `funny()` so the First class can use it when it calls its `fun()` class
- The main function instantiates an object of the class First and an object of the class Second.
- The main function calls the `funny(. . .)` member function of the class Second and passes an object of the class First as a parameter.
- An object of the class Second can then call the `fun()` function of the class First using the name of the object that received from main.
- The relationship between objects First and Second is dependency. Object Second uses class First in its member function `funny(. . .)`, and object Second calls the member function of object First.



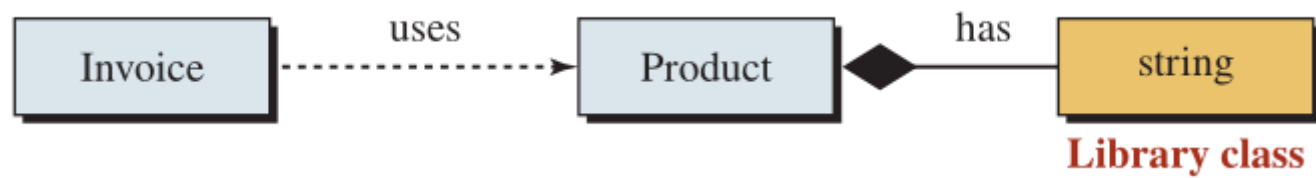
Legend:



Example

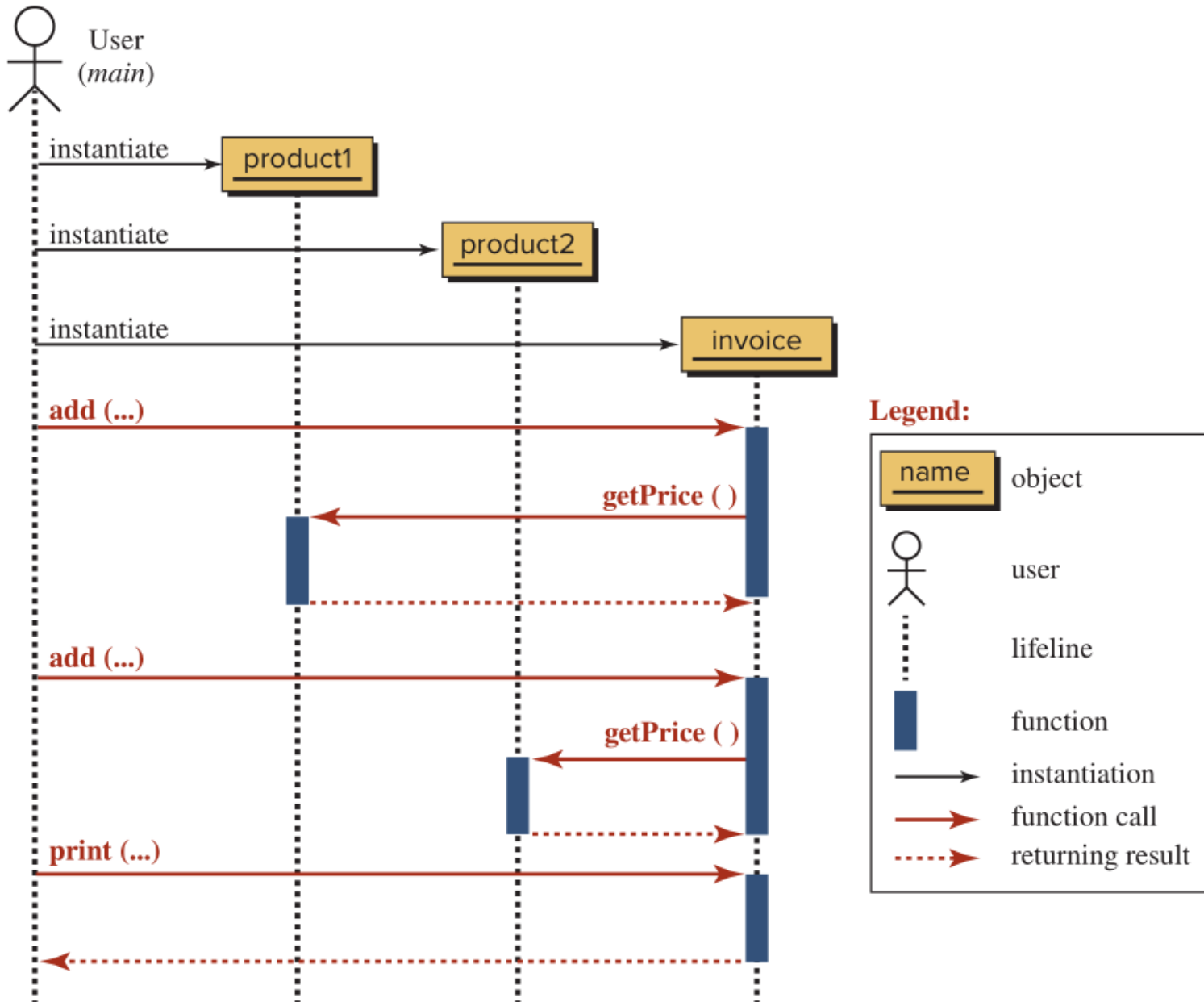
- We use a comprehensive example to demonstrate the basics of dependency relationships.
- Assume we want to create an invoice for the list of products sold. We have a class named Invoice and a class named Product.
- The class Invoice uses instances of the class Product as a parameter in one of its member functions (add).

Class Diagram



Sequence Diagram

- The main function must instantiate two objects of type Product and one object of type Invoice.
- The main function then calls the add function in the Invoice class to add the products to the invoice, but it must get the price of each product from the corresponding object.



```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4  class Product
5  {
6      private:
7          string name;
8          double unitPrice;
9      public:
10         Product (string name, double unitPrice);
11         ~Product ( );
12         double getPrice ( ) const;
13 };

```

```
15 // Constructor
16 Product :: Product (string nm, double up)
17 : name (nm), unitPrice (up)
18 {
19 }
20 // Destructor
21 Product :: ~Product ( )
22 {
23 }
24 // The getPrice member function
25 double Product :: getPrice ( ) const
26 {
27     return unitPrice;
28 }
```



```

31 class Invoice
32 {
33     private:
34         int invoiceNumber;
35         double invoiceTotal;
36
37     public:
38         Invoice (int invoiceNumber);
39         ~Invoice ( );
40         void add (int quantity, Product product);
41         void print ( ) const;
42 };
43
44 // Constructor
45 Invoice :: Invoice (int invNum)
46 : invoiceNumber (invNum), invoiceTotal (0.0)
47 {
48 }

```

```

50 // Destructor
51 Invoice :: ~Invoice ( )
52 {
53 }
54
55 // Add member function
56 void Invoice :: add (int quantity, Product product)
57 {
58     invoiceTotal += quantity * product.getPrice ();
59 }
60
61 // Print member function
62 void Invoice :: print ( ) const
63 {
64     cout << "Invoice Number: " << invoiceNumber << endl;
65     cout << "Invoice Total: " << invoiceTotal << endl;
66 }

```

```

68 int main ( )
69 {
70     // Instantiation of two products
71     Product product1 ("Table", 150.00);
72     Product product2 ("Chair", 80.00);
73
74     // Creation of invoice for the two products
75     Invoice invoice (1001);
76     invoice.add (1, product1);
77     invoice.add (6, product2);
78     invoice.print ();
79     return 0;
80 }

```

D:\Work Umair\4_CUST (1-9-22)\1_Teaching\2_ACS1143-OOP\Practice Programs\W11-P3.exe

```

Invoice Number: 1001
Invoice Total: 630
-----

```

This is all for Week 11