Object-oriented Programming
Class Relationships

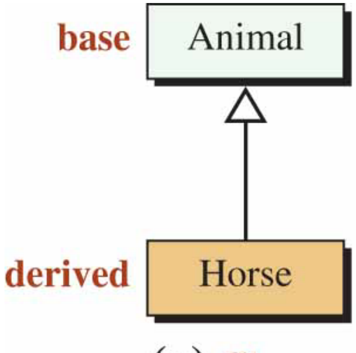
YoungWoon Cha

Computer Science and Engineering

Review

Class Inheritance

Classes and objects in inheritance hierarchy



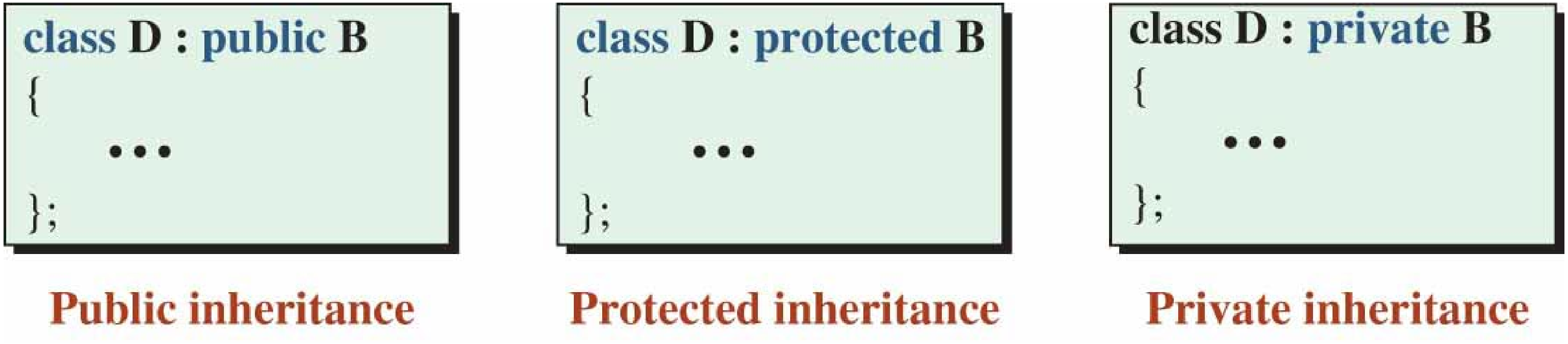
(a) Classes

The derived class inherits all members (with some exceptions) from the base class, and it can add to them.

Class Inheritance Types

To create a derived class from a base class, we have three choices in C++: ***private inheritance, protected inheritance, public inheritance***

Inheritance types



```
class D : public B
{
     • • •
};
```

Public inheritance

```
class D : protected B
{
     • • •
};
```

Protected inheritance

```
class D : private B
{
     • • •
};
```

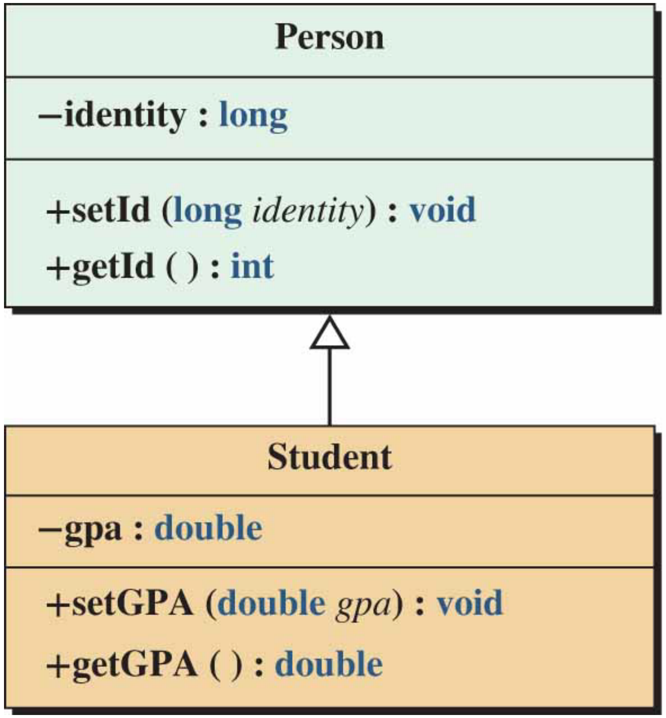
Private inheritance

The most common use of inheritance is public inheritance.

Public Inheritance

We add some member functions to the two classes. We ignore the constructors and destructor at this point because they are not inherited; we assume the synthetic ones are used. We add one *getter* and one *setter* function to each class.

Classes with data members and member functions



Person

−identity : long

+setId (long *identity*) : void

+getId ( ) : int

Student

−gpa : double

+setGPA (double *gpa*) : void

+getGPA ( ) : double

Notes:

The type of data members and member functions is shown after the member name separated by a colon.

The minus signs define the visibility of data members as private; the plus signs define the visibility of the member fuctions as public.

Overloaded and Overridden Member Functions

Overloaded functions are two functions with the same name but with two different signatures.

Overloaded functions can be used in the same or different classes without being confused with each other.

We can have two functions named *set*, one in the base class and the other in the derived class with the following prototypes:

```
// In Person class                  // In Student class
void set(long identity);            void set(double gpa);
```

If the signature of the two functions with the same name is the same, we have overridden member functions as shown below.

```
// In Person class                  // In Student class
long get();                         double get();
```

Members Not Inherited : Invocation

There are five member functions that are not inherited in the derived class:

1. default constructor
2. parameter constructor
3. copy constructor
4. destructor
5. assignment operator.

We postpone the discussion of the assignment operator until a future chapter, but we discuss the other four in this chapter.

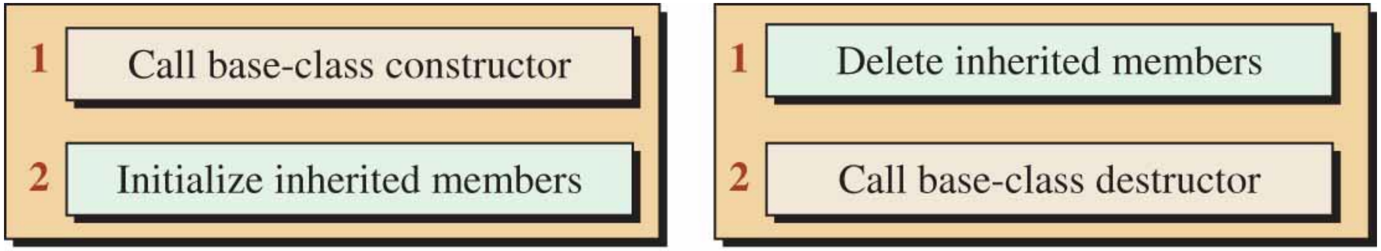
Constructors, destructor, and assignment operators are not inherited; they need to be redefined.

Constructor and Destructor in Inheritance

The constructor of the derived class cannot initialize the data members of the base class because they are hidden in the derived class.

Similarly, the destructor of a derived class cannot delete the data members of the base class because they are hidden in the derive class.

Constructor and destructor in inheritance



1 Call base-class constructor

2 Initialize inherited members

1 Delete inherited members

2 Call base-class destructor

Constructors for derived class

Destructors for derived class

The constructor of the derived class invokes the constructor of the base class in its initialization and then initializes the data members of the derived class.

Similarly, the destructor of the derived class first deletes the data members of the derived class and then calls the destructor of the base class.

Note that the order of activities in a constructor and destructor are reverse of each other.

Delegation Versus Invocation

Delegation and invocation are different concepts and are done differently.

In delegation, a derived member function delegates part of its duty to the base class using the class resolution operator (::).

```
// Using Person object
void Person :: set(long id)
{
     identity = id;
}
```

```
// Using Student object
void Student :: set(long id, double gp)
{
     Person :: set(id);   // Delegation
     gpa = gp;
}
```

In invocation, the constructor of a derived class calls the constructor of the base class during initialization, which does not require the class resolution operator.

```
// Parameter Constructor
Person :: Person (long id)
identity(id),
{
}
```

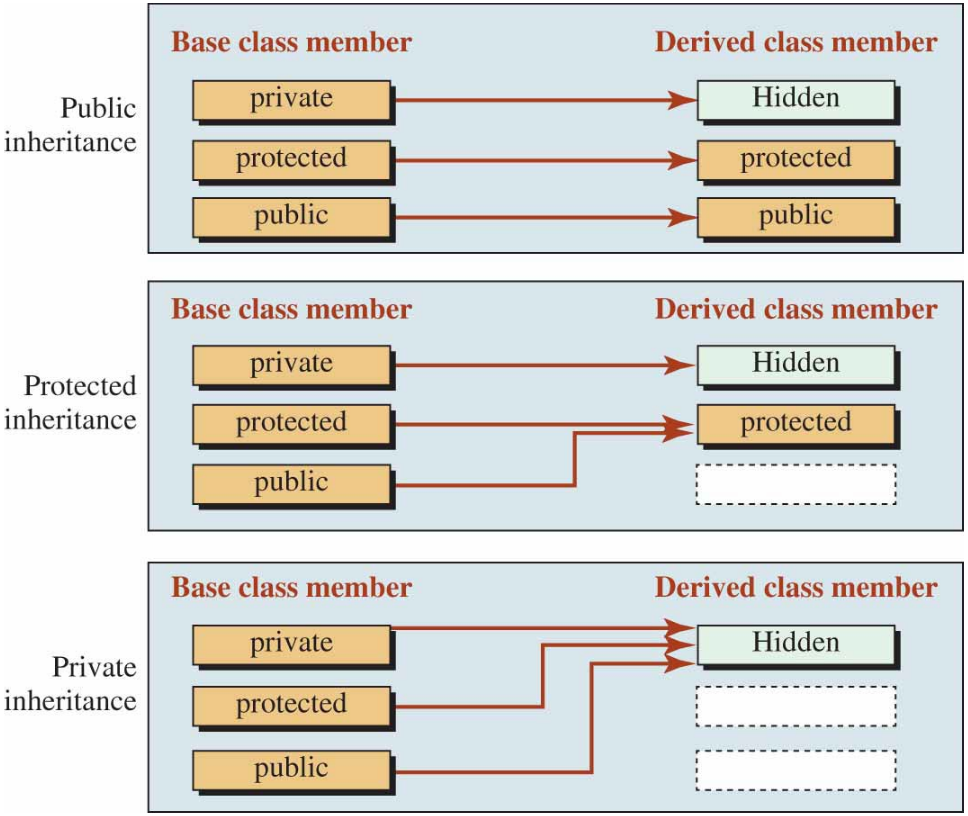
```
// Parameter Constructor
Student :: Student (long id, double gp) :
: Person(id), gpa(gp)
{
}
```

Inheritance Continued

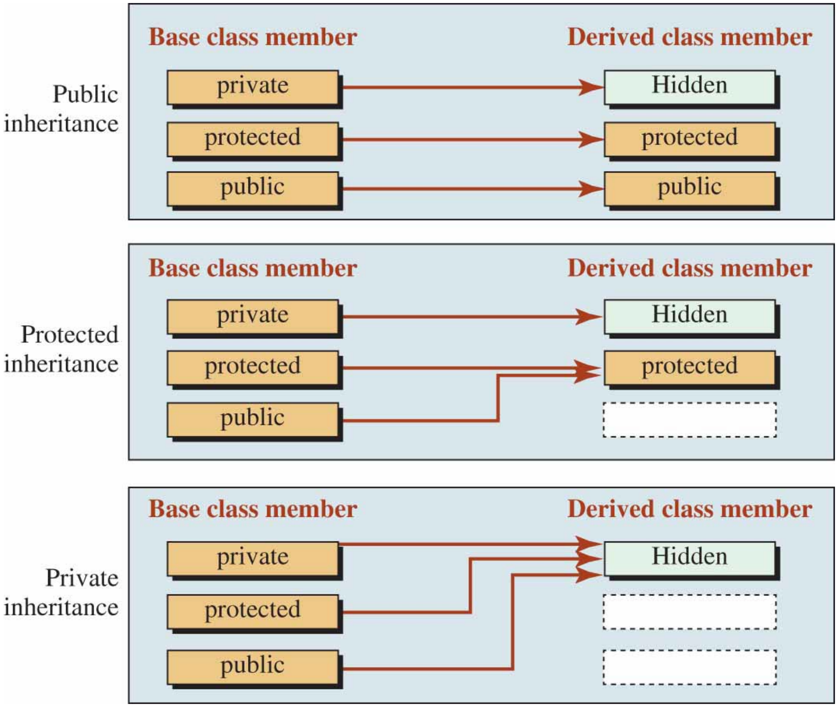
Three Types of Inheritance

Although, public inheritance is by far the most common type of derivation, C++ allows us to use two other types of derivation: *private* and *protected*.

Inheritance types



Three Types of Inheritance



Public Inheritance

Public inheritance is what we use most of the time.

This type of derivation defines an *is-a* relationship between the base class object and the derived class object because the public interface of the base class becomes a public interface of the derived class.

Protected Inheritance

This type of inheritance is rare and virtually never used.

Private Inheritance

Private inheritance is much less common than public inheritance, but it has some applications.

Public members of the base class become private members in the derived class.

This property allows inherited implementation (code reuse).

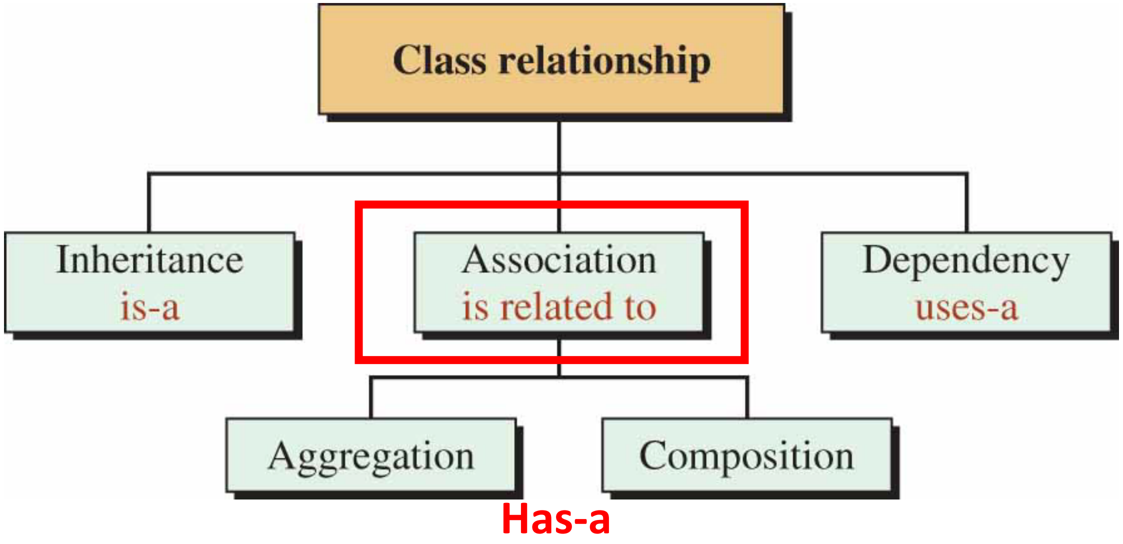
Class Relationships

Class Relationships

In object-oriented programming, classes are used in relation to each other.

A program normally uses several classes with different relationships between them.

Relationship between classes



ASSOCIATION PART 1

The second type of relationship is *association*. As a matter of fact, more programs are being developed today that use association rather than inheritance.

An association between two classes shows a relationship.

For example, we can define a class named *Person* and another class named *Address*.

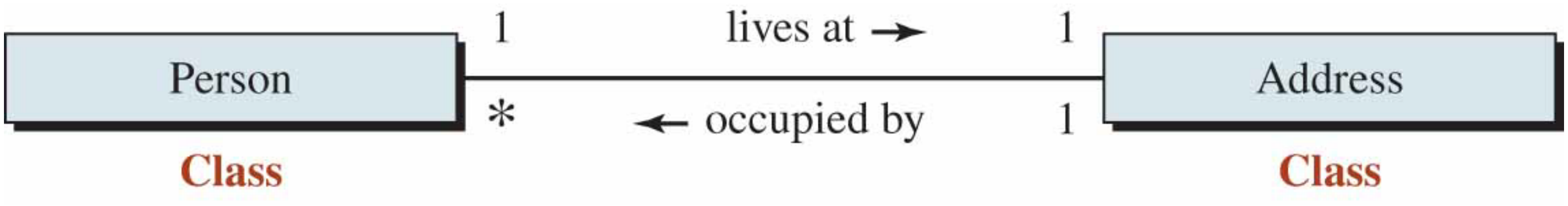
An object of the type *Person* may be related to an object of type *Address*: a person lives in an address and the address is occupied by a person.

The *Address* class is not inherited from the *Person* class; neither the other way.

ASSOCIATION PART 2

A relationship of this type is shown in UML diagrams as a solid line between two classes

An association relationship



An association diagram also shows the type of relationship using an arrow head and text in the direction of the corresponding class.

ASSOCIATION PART 3

Another piece of information represented in an association diagram is multiplicity.

Multiplicity defines the number of objects that take part in the association.

Table shows different types of multiplicity.

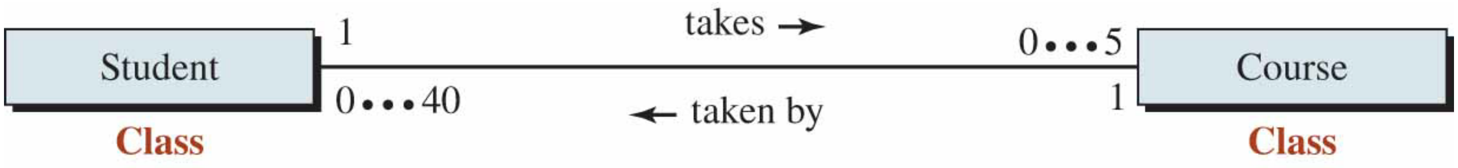
Multiplicity in association diagrams

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

ASSOCIATION PART 4

As another example, we define the association relationship between students and courses they take in Figure

Association between courses and students



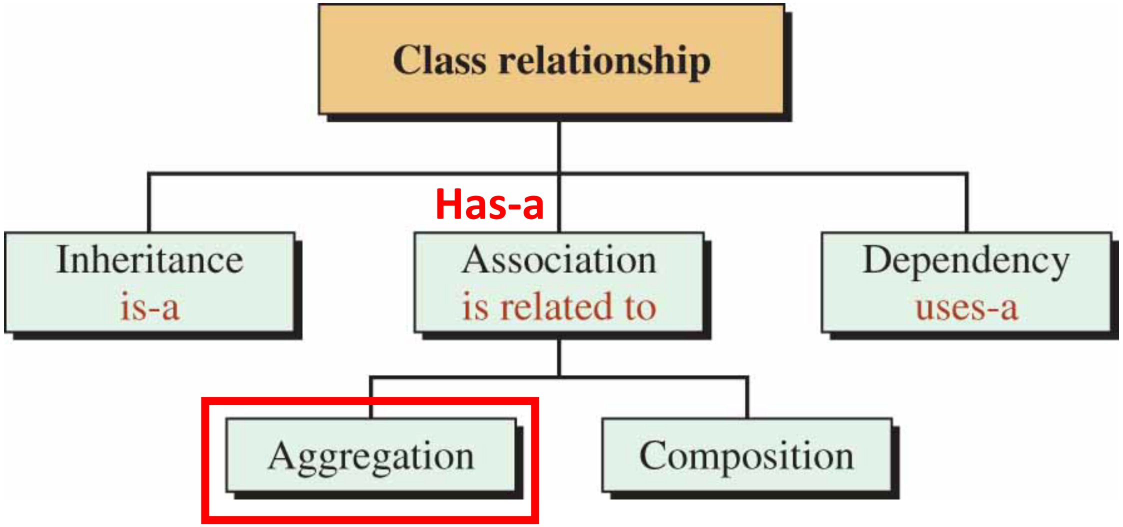
For example, a Student object can have a list of five course names and a Course object can have a list of forty student names

Class Relationships

In object-oriented programming, classes are used in relation to each other.

A program normally uses several classes with different relationships between them.

Relationship between classes



Aggregation (Association Type 1)

Aggregation is a special kind of association in which the relationship involves ownership.

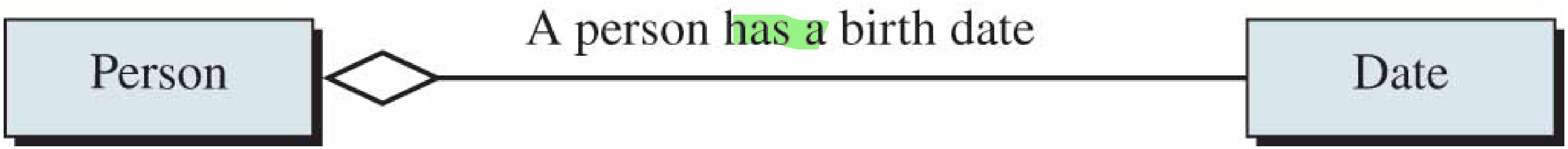
In other words, it models the "*has-a*" relationship. One class is called an aggregator and the other an aggregatee.

In other words, an object of the aggregator class contains one or more objects of the aggregatee class.

Figure shows the UML diagram for aggregation.

Note that the symbol for aggregation is a hollow diamond placed at the site of the aggregator.

Example of aggregation relationship



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

Aggregation (Association Type 1)

The aggregation relationship is one-way!

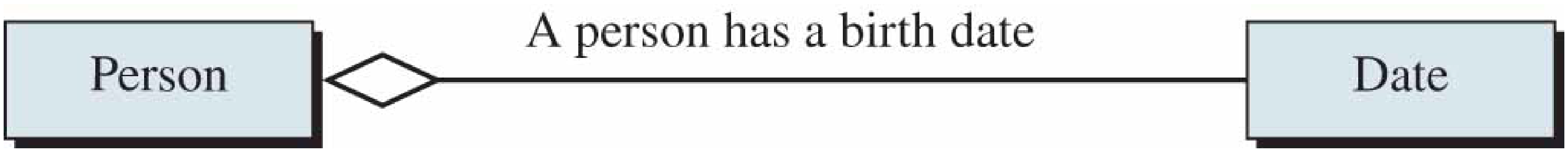
A Person can have a birth date.

A Date object can be related to multiple events.

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 of aggregation relationship



Exercise #1

Interface File for date.h Class

```
/***************************************************************
 * The interface file for date.h class                         *
 ***************************************************************/
#ifndef DATE_H
#define DATE_H
#include <iostream>
#include <cassert>
using namespace std;

class Date
{
    private:
        int month;
        int day;
        int year;
    public:
        Date (int month, int day, int year);
        ~Date ();
        void print() const;
};
#endif
```

Implementation of Functions in the Date Class Part 1

File date.cpp

```cpp
/****************************************************************
 * The implementation of functions in the Date class           *
 ****************************************************************/
#include "date.h"

// Parameter constructor
Date :: Date (int m, int d, int y)
: month (m), day (d), year (y)
{
    if ((month < 1) || (month > 12))
    {
        cout << "Month is out of range. ";
        assert (false);
    }
    int daysInMonths [13] = {0, 31, 28, 31, 30, 31, 30, 31,
                                            31, 30, 31, 30 ,31};
    if ((day < 1) || (day > daysInMonths [month]))
    {
        cout << "Day out of range! ";
        assert (false);
```

Implementation of Functions in the Date Class Part 2

File date.cpp

```cpp
    }
    if ((year < 1900) || (year > 2099))
    {
        cout << "Year out of range! " ;
        assert (false);
    }
}
// Destructor
Date :: ~Date ()
{
}
// Print member function
void Date :: print() const
{
    cout << month << "/" << day << "/" << year << endl;
}
```

Interface File for the Person Class

File person.h

```
/****************************************************************
 * The interface file for the Person class                      *
 ****************************************************************/
#ifndef PERSON_H
#define PERSON_H
#include "date.h"

// Definition of the Person class
class Person
{
    private:
        long identity;
        Date birthDate;
    public:
        Person (long identity, Date birthDate);
        ~Person ( );
        void print ( ) const;
};
#endif
```

Implementation File for Person Concrete Class

File person.cpp

```
/*****************************************************************
 * The implementation file for Person concrete class            *
 *****************************************************************/
#include "person.h"

// Constructor
Person :: Person (long id, Date bd)
: identity (id), birthDate (bd)
{
    assert (identity > 111111111 && identity < 999999999);
}
// Destructor
Person :: ~Person ( )
{
}
// Print function
void Person :: print ( ) const
{
    cout << "Person Identity: " << identity << endl;
    cout << "Person date of birth: ";
    birthDate.print ();
    cout << endl << endl;
}
```

Application File to Test the Person Class

```
/***************************************************************
 * The application file to test the Person class               *
 ***************************************************************/
#include "person.h"

int main ( )
{
    // Instantiation
    Date date1 (5, 6, 1980);
    Person person1 (111111456, date1);
    Date date2 (4, 23, 1978);
    Person person2 (345332446, date2);
    // Output
    person1.print ( );
    person2.print ( );
    return 0;
}
```

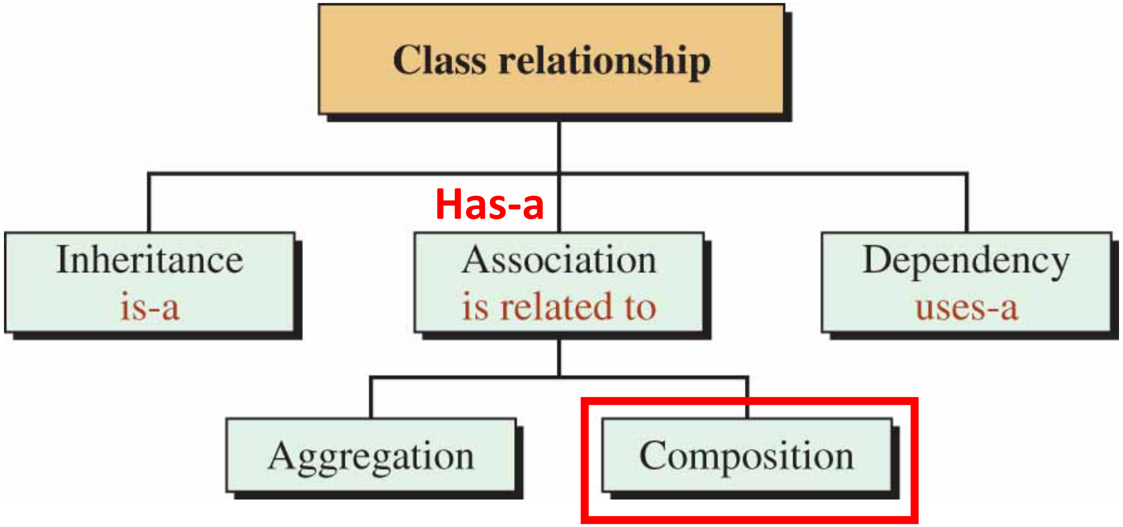
```
Run:
Person Identity: 111111456
Person date of birth: 5/6/1980

Person Identity: 345332446
Person date of birth: 4/23/1978
```

Class Relationships

In object-oriented programming, classes are used in relation to each other.

A program normally uses several classes with different relationships between them.

Relationship between classes



Composition (Association Type 2)

A composition is a special kind of association 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.

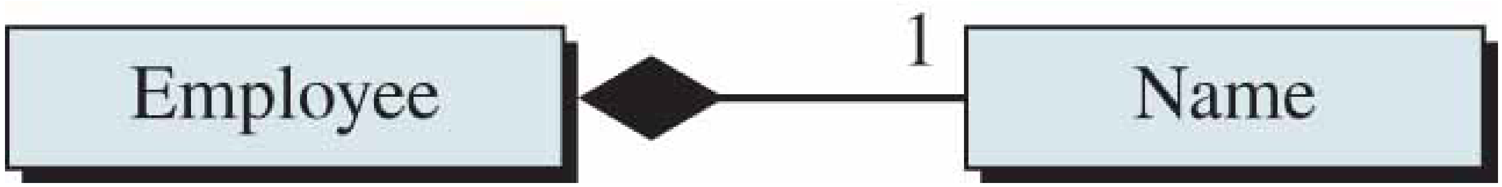
The name cannot exist without being the name of a person.

The name must always belong to a person and cannot have a life of its own.

Figure shows the relationship between an employee and her name.

A solid diamond placed at the side of the composer.

Example of composition relationship



Exercise #2

Interface File for the Name Class

File name.h

```
/***************************************************************
 * The interface file for the Name class                       *
 ***************************************************************/

#ifndef NAME_H
#define NAME_H
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

class Name
{
    private:
        string first;
        string init;
        string last;
    public:
        Name (string first, string init, string last);
        ~Name ( );
       void print ( ) const;
};
#endif
```

Implementation File for Name Class

File name.cpp

```
/****************************************************************
 * The implementation file for Name class                       *
 ****************************************************************/
#include "name.h"

// Constructor
Name :: Name (string fst, string i, string lst)
:first (fst), init (i), last (lst)
{
    assert (init.size () == 1);
    toupper (first[0]);
    toupper (init [0]);
    toupper (last[0]);
}
// Destructor
Name :: ~Name ( )
{
}
// Print member function
void Name :: print ( ) const
{
     cout << "Emplyee name: " << first << " " << init << ". ";
     cout << last << endl;
}
```

Interface File for the Employee Class

File employee.h

```
/***************************************************************
 * The interface file for the employee class                   *
 ***************************************************************/
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include "name.h"

class Employee
{
    private:
        Name name;
        double salary;
    public:
        Employee (string first, string init, string last,
                                                     double salary);
        ~Employee ( );
        void print ( ) const;
};
#endif
```

Implementation File for Employee Class

File employee.cpp

```
/***************************************************************
 * The implementation file for Employee class                  *
 ***************************************************************/

#include "employee.h"

// Constructor
Employee :: Employee (string fst, string i, string lst,
                                                     double sal)
: name (fst, i, lst), salary (sal)
{
    assert (salary > 0.0 and salary < 100000.0);
}
// Destructor
Employee :: ~Employee ( )
{
}
// Print member function
void Employee :: print ( ) const
{
    name.print();
    cout << "Salary: " << salary << endl << endl;
}
```

Application File to Test the Employee Class

```cpp
/******************************************************************
 * The application file to test the Employee class               *
 *****************************************************************/
#include "employee.h"

int main ( )
{
    // Instantiation
    Employee employee1 ("Mary", "B", "White", 22120.00);
    Employee employee2 ("William", "S", "Black", 46700.00);
    Employee employee3 ("Ryan", "A", "Brown", 12500.00);
    // Output
    employee1.print ( );
    employee2.print ( );
    employee3.print ( );
    return 0;
}
```

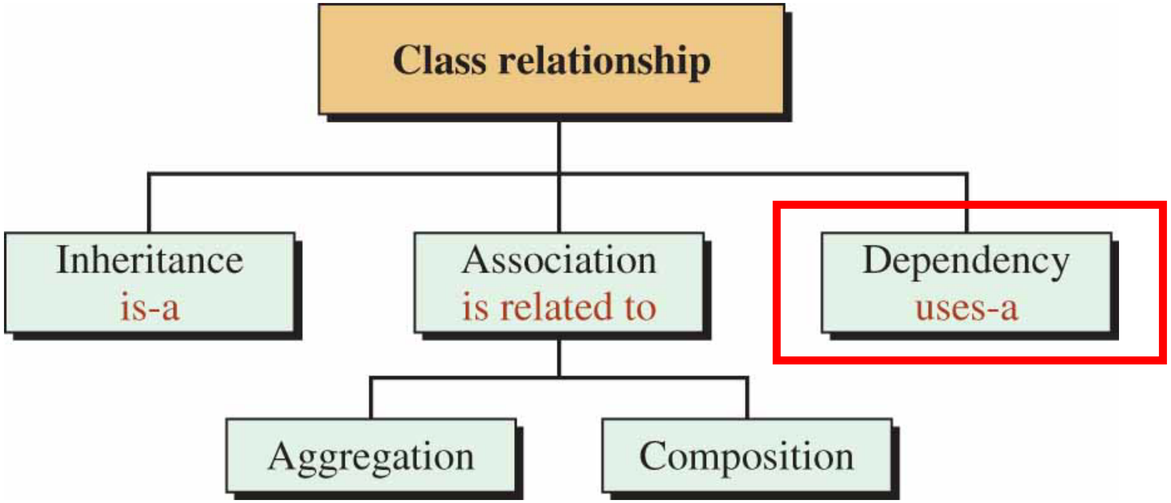
```
Run:
Emplyee name: Mary B. White
Salary: 22120

Emplyee name: William S. Black
Salary: 46700

Emplyee name: Ryan A. Brown
Salary: 12500
```

Class Relationships

In object-oriented programming, classes are used in relation to each other.

A program normally uses several classes with different relationships between them.

Relationship between classes



Has-a

DEPENDENCY

The third type of relationship is dependency. Dependency is a weaker relationship than inheritance or association.

We say that dependency models the "*uses*" relationship.

Class A depends on class B if class A somehow uses class B. 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.**

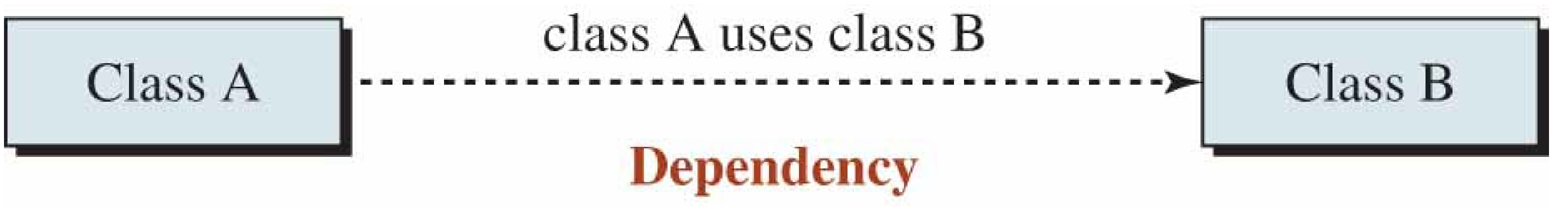
UML Diagrams

We use both UML class diagrams and UML sequence diagrams to show the dependencies.

UML Class Diagram

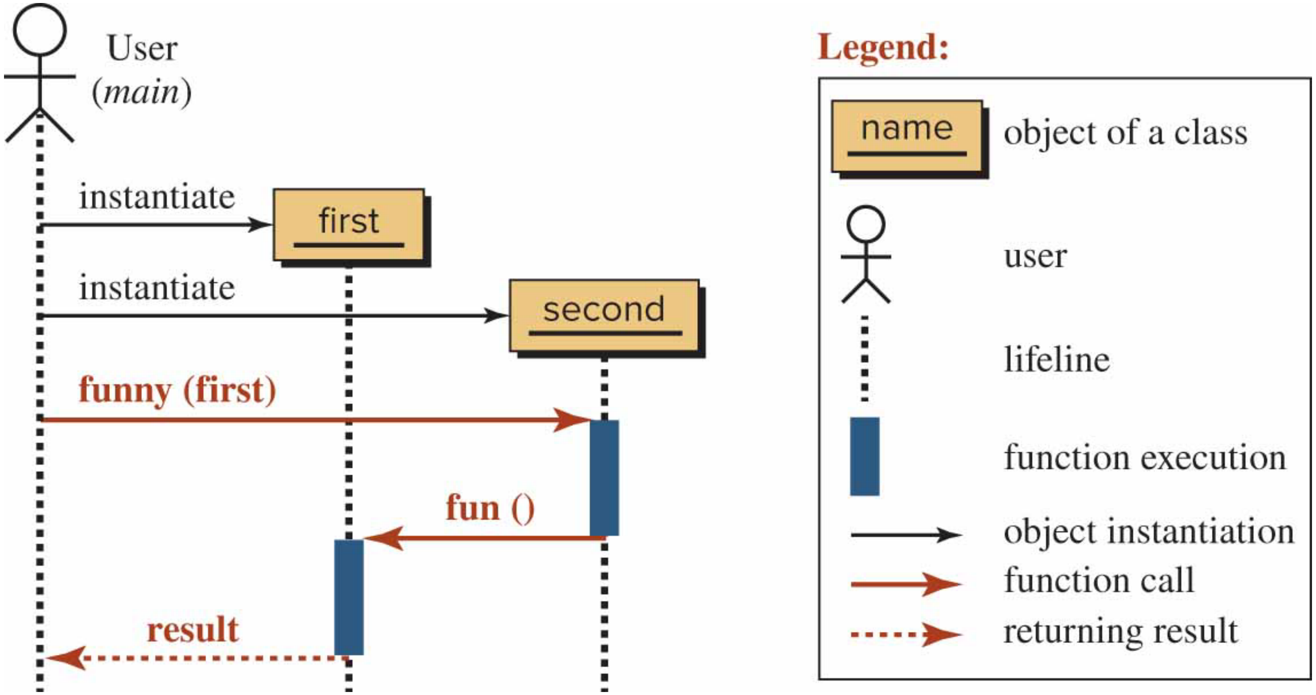
Figure shows an example of a dependency relationship in which class A depends on class B.

UML class diagram for dependency



Sequence Diagrams

A *sequence* diagram shows the interaction between objects. The *main* function and each object has a lifeline that shows the passing of time. The objects can be instantiated and their member function can be called.



In the figure, 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*.

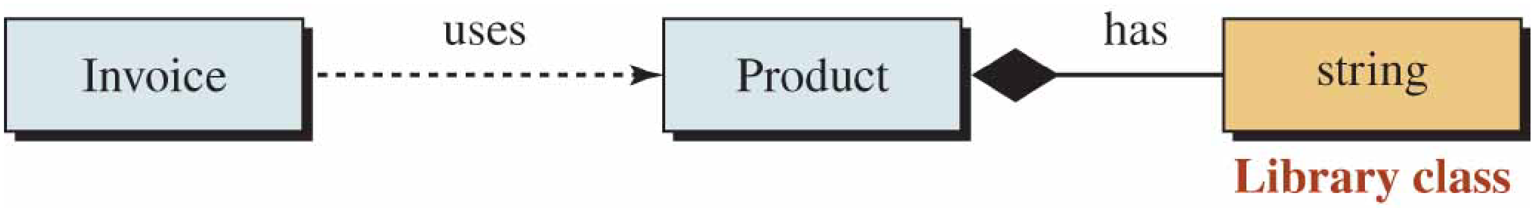
A Comprehensive Example Part 1

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.

The class Invoice uses instances of the class Product as a parameter in one of its member functions (*add*).

UML class diagram for Invoice and Product

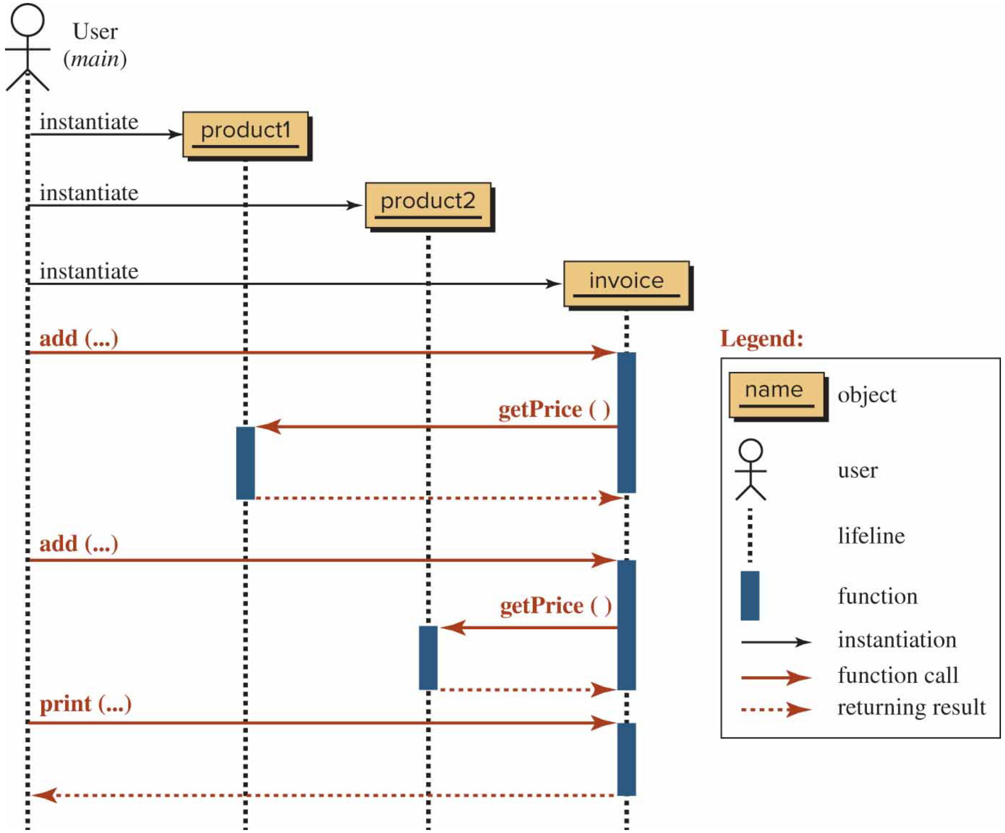


A Comprehensive Example Part 2

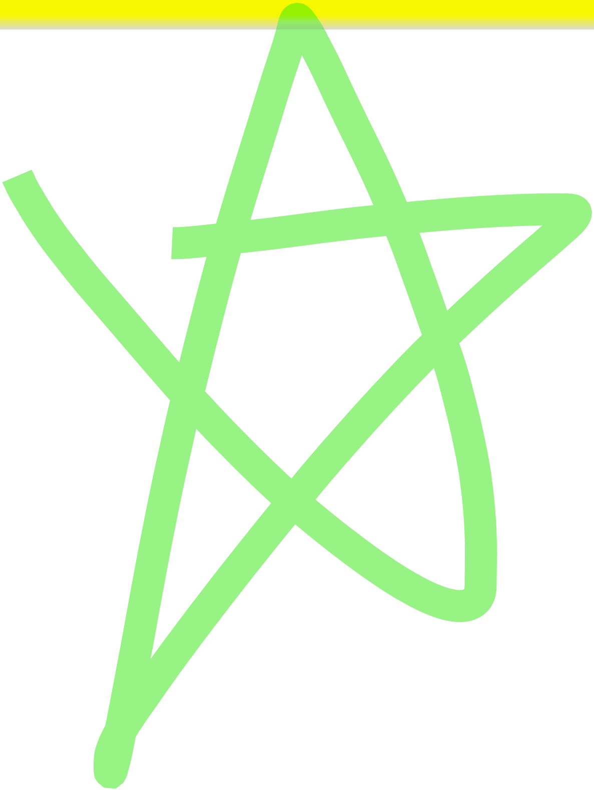
We show the sequence diagram for two products in Figure.

Note that the *main* function needs to instantiate two object 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 needs to get the price of each product from the corresponding object.



Exercise #3



Interface File for the Product Class

File product.h

```
/***************************************************************
 * The interface file for the Product class                    *
 ***************************************************************/
#ifndef PRODUCT_H
#define PRODUCT_H
#include <string>
#include <iostream>
using namespace std;

class Product
{
    private:
        string name;
        double unitPrice;
    public:
        Product (string name, double unitPrice);
        ~Product ( );
        double getPrice ( ) const;
};
#endif
```

Implementation File for Product Class

File product.cpp

```cpp
/***************************************************************
 * The implementation file for Product class                   *
 ***************************************************************/
#include "product.h"

// Constructor
Product :: Product (string nm, double up)
: name (nm) , unitPrice (up)
{
}
// Destructor
Product :: ~Product ( )
{
}
// The getPrice member function
double Product :: getPrice ( ) const
{
    return unitPrice;
}
```

Interface File for the Invoice Class

File Invoice.h

```
/***************************************************************
 * The interface file for the Invoice class                     *
 ***************************************************************/
#ifndef INVOICE_H
#define INVOICE_H
#include "product.h"

class Invoice
{
    private:
        int invoiceNumber;
        double invoiceTotal;
    public:
        Invoice (int invoiceNumber);
        ~Invoice ( );
        void add (int quantity, Product product);
        void print ( ) const;
};
#endif
```

Implementation File for Invoice Class

File invoice.cpp

```
/***************************************************************
 * The implementation file for Invoice class                   *
 ***************************************************************/
#include "invoice.h"

// Constructor
Invoice :: Invoice (int invNum)
: invoiceNumber (invNum), invoiceTotal (0.0)
{
}
// Destructor
Invoice :: ~Invoice ( )
{
}
// Add member function
void Invoice :: add (int quantity, Product product)
{
    invoiceTotal += quantity * product.getPrice ();
}
// Print member function
void Invoice :: print ( ) const
{
    cout << "Invoice Number: " << invoiceNumber << endl;
    cout << "Invoice Total: " << invoiceTotal << endl;
}
```

Application File to Test the Invoice Class

File application.cpp

```
/***************************************************************
 * The application file to test the Invoice class              *
 ***************************************************************/
#include "invoice.h"

int main ( )
{
    // Instantiation of two products
    Product product1 ("Table", 150.00);
    Product product2 ("Chair", 80.00);
    // Creation of invoice for the two product
    Invoice invoice (1001);
    invoice.add (1, product1);
    invoice.add (6, product2);
    invoice.print ();
    return 0;
}
```

```
Run:
Invoice Number: 1001
Invoice Total: 630
```

Application

A Tokenizer Class

A common problem is to tokenize a string of characters using a list of delimiters.

For example, we may need to extract words from a text. The words in a text are separated by spaces and new-line characters.

The words in this case are called tokens and the characters that separate the words are delimiters. For example, the following string has seven tokens (words) in it.

```
"This is a book about C++ language"
```

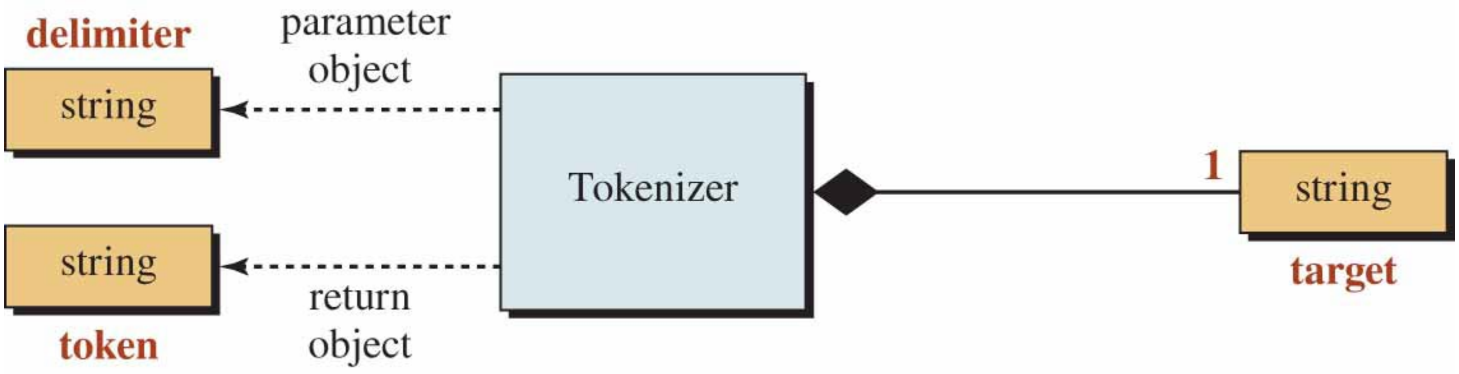
The C++ language has no class whose objects can tokenize a string, but we can create a class to do so.

The string library class has member functions to search a string to find a character in a string or to find a character which is not in a string.

We can use these functions in the string library to create a Tokenizer class.

Relationship Between Classes

UML class diagram



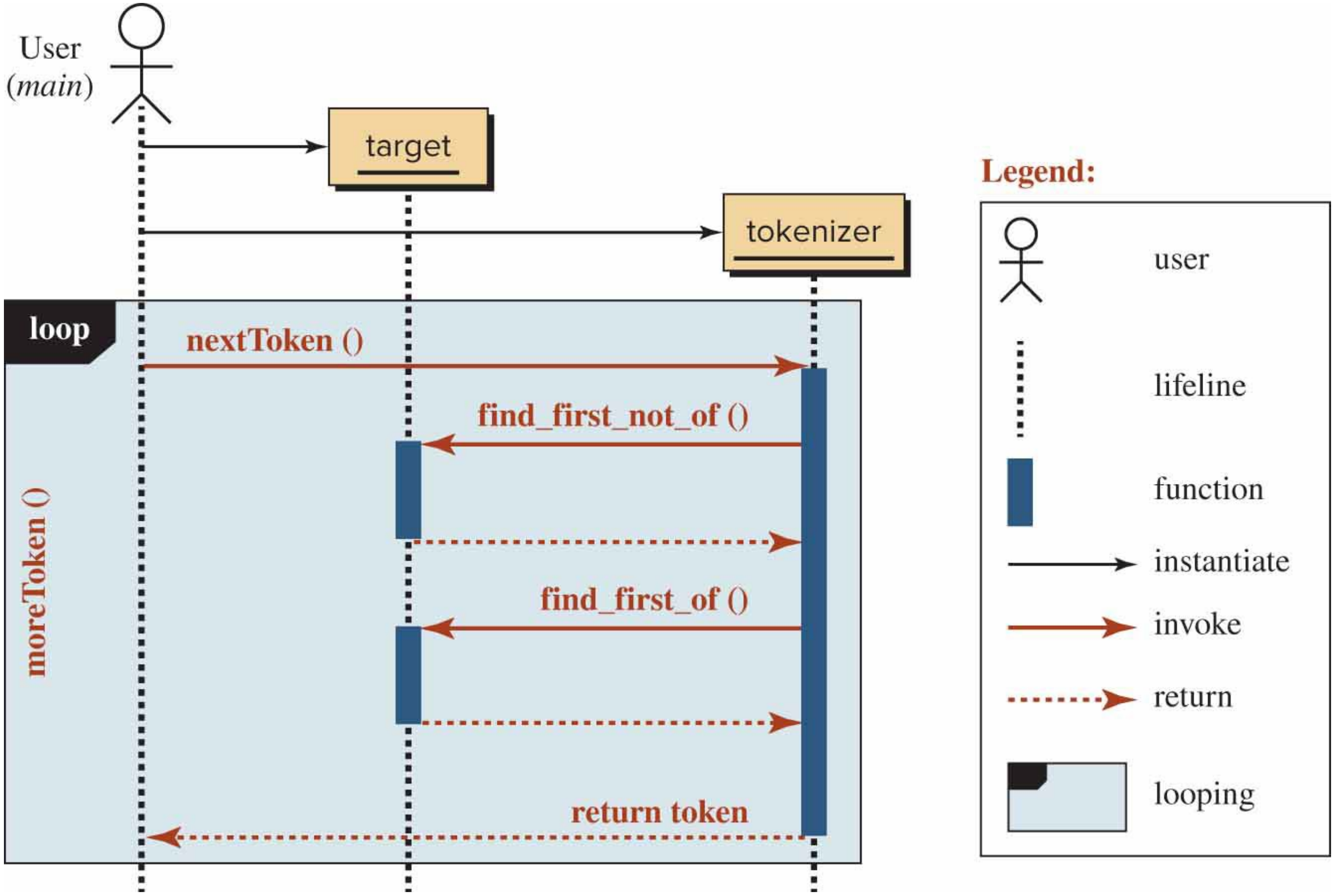
The Tokenizer class uses the string class three times.

It uses it to create the target string, the delimiter string, and the finally to create tokens.

The relationship between the tokenizer object and the target object is composition.

The relationship between the tokenizer object and tokens is dependency.

Sequence Diagram



Since the string class (target) is a library class with predefined public interfaces, we have to create only one class, *Tokenizer*.

The user can create an application file to create and use objects of the Tokenizer class.

Exercise #4

Interface File for the Tokenizer Class

File tokenizer.h

```
/***************************************************************
 * The interface file for the Tokenizer class                  *
 ***************************************************************/
#ifndef TOKENIZER_H
#define TOKENIZER_H
#include <iostream>
#include <string>
using namespace std;

class Tokenizer
{
    private:
        string target;
        string delim;
        int begin;
        int end;
    public:
        Tokenizer (const string& target, const string& delim);
        ~Tokenizer ();
        bool moreToken() const;
        string nextToken();
};
#endif
```

Implementation File for the Tokenizer Class

```
/*************************************************************
 * The implementation file for the Tokenizer class           *
 *************************************************************/
#include "tokenizer.h"

// Constructor
Tokenizer :: Tokenizer (const string& tar, const string& del)
: target (tar), delim (del)
{
     begin = target.find_first_not_of (delim, 0);
     end = target.find_first_of (delim, begin);
}
// Destructor
Tokenizer :: ~Tokenizer()
{
}
// Checks for more tokens
bool Tokenizer :: moreToken ( ) const
{
     return (begin != – 1);
}
// Returns the next token
string Tokenizer :: nextToken ( )
{
     string token = target.substr (begin, end – begin);
     begin = target.find_first_not_of (delim, end);
     end = target.find_first_of(delim, begin);
     return token;
}
```

Application File to Test the Tokenizer Class

```
/*************************************************************
 * The application file to test the Tokenizer class           *
 *************************************************************/
#include "tokenizer.h"

int main ( )
{
     // The target string that needs to be tokenized
     string target ("This is the string to be tokenized. \n");
     // The delimit string defines the set of separators
     string delimit (" \n"); // Delimiter made of ’ ’ and ’\n’
     // Instantiation of tokenizer object
     Tokenizer tokenizer (target, delimit);
     // Traversing the target string to find tokens
     while (tokenizer.moreToken ())
     {
          cout << tokenizer.nextToken () << endl;
     }
     return 0;
}
```

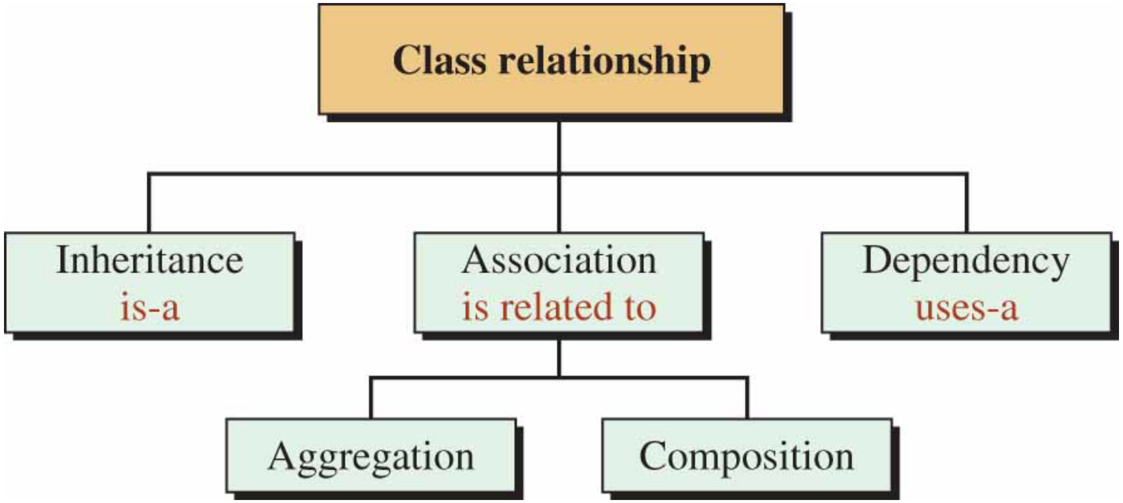
```
Run:
This
is
the
string
to
be
tokenize.
```

Summary

In object-oriented programming, classes are used in relation to each other.

A program normally uses several classes with different relationships between them.

Class Relationships



Has-a

What’s Next?

Reading Assignment

- **Read Chap. 12. Polymorphism and Other Issues**

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr