Chương 6: Ngôn ngữ lập trình hướng đối tượng

Giảng viên: Ph.D Nguyễn Văn Hòa

Khoa KT-CN-MT – ĐH An Giang

Nội dung chính của chương

- Giới thiệu
- Khái niệm về trừu tượng hóa
- Trừu tượng hóa dữ liệu
- Sự đóng gói
- Tính thừa kế
- Một số ngôn ngữ lập trình hướng đối tượng

Sự phát triển kỹ thuật lập trình

- Mục tiêu của kỹ sư phần mềm
  - Tạo ra sản phẩm tốt bằng một cách có hiệu quả
  - Nắm bắt được công nghệ
- Phần mềm ngày càng lớn
  - Hệ điều hành (Unix, Windows) : hàng chục triệu dòng lệnh
  - Người dùng ngày càng đòi hỏi nhiều chức năng
  - Phần mềm luôn cần được sữa đổi

Vì vậy

- Cần kiểm soát chi phí
  - Chi phí phát triển
  - Chi phí bảo trì
- Giải pháp chính là ***sử dụng lại code***
  - Giảm chi phí và thời gian phát triển
  - Nâng cao chất lượng

Để sử dụng lại code (mã nguồn)

- Mã nguồn cần dễ hiểu
- Mã nguồn phải chính xác
- Có giao diện (inteface) rõ ràng
- Không yêu cầu thay đổi khi sử dụng trong chương trình mới

Giải pháp: LT hướng đối tượng

- Che dấu dữ liệu (che dấu cấu trúc)
- Truy cập dữ liệu thông qua giao diện xác định

```
class MyDate {
  private int year, mon, day;
  public int getDay() {...}
  public boolean setDay(int) {...}
  ...
}
```

Khái niệm

- **Lập trình hướng đối tượng** (OOP- Object-Oriented Programming)
  - Một cách tư duy mới, tiếp cận hướng đối tượng để giải quyết vấn đề bằng máy tính
  - Một phương pháp thiết kế và phát triển phần mềm dựa trên kiến trúc lớp và đối tượng
- Qui trình tiến hóa của OOP
  - Lập trình tuyến tính
  - Lập trình cấu trúc (lập trình thủ tục)
  - Trừu tượng hóa dữ liệu
  - Lập trình hướng đối tượng

Tại sao tiếp cận hướng đối tượng

- Loại bỏ những thiếu sót của tiếp cận theo thủ tục
- Trong OOP
  - Dữ liệu được xem như một phần tử chính yếu và được bảo vệ
  - Hàm gắn kết với dữ liệu, thao tác trên dữ liệu
  - Phân tách bài toán thành nhiều thực thể (đối tượng) → xây dựng dữ liệu + hàm cho các đối tượng này
- Tăng cường khả năng sử dụng lại

Đặc điểm của OOP

- Nhấn mạnh trên dữ liệu hơn là thủ tục
- Các chương trình được chia thành các đối tượng
- Dữ liệu được che giấu và không thể được truy xuất từ các hàm bên ngoài
- Các đối tượng có thể giao tiếp với nhau thông qua các hàm
- Dữ liệu hay các hàm mới có thể được thêm vào khi cần
- Theo tiếp cận từ dưới lên

Ưu điểm của OOP

- So với các tiếp cận cổ điển thì OOP có những thuận lợi sau:
  - OOP cung cấp một cấu trúc module rõ ràng
    - Giao diện được định nghĩa tốt
    - Những chi tiết cài đặt được ẩn
  - OOP giúp lập trình viên duy trì mã và sửa đổi mã tồn tại dễ dàng (các đối tượng được tạo ra với những khác nhau nhỏ so với những đối tượng tồn tại).
  - OOP cung cấp một framework tốt với các thư viện mã mà các thành phần có thể được chọn và sửa đổi bởi lập trình viên.

Trừu tượng hóa

- Trừu tượng hóa là chỉ biểu diễn những đặc điểm cần thiết của vấn đề
- Trừu tượng hóa là nền tảng cơ bản trong lập trình (và trong khoa học máy tính)
- Gần như toàn bộ các NNLT đều hỗ trợ trừu tượng hóa tiến trình bằng chương trình con
- Từ 1980s, gần như các NNLT đều được thiết kế để hỗ trợ trừu tượng hóa dữ liệu

Ưu điểm của việc trừu tượng hóa

- Tập trung vào các vấn đề cần quan tâm
- Xác định những đặc tính thiết yếu và những hành động cần thiết
- Giảm thiểu những chi tiết không cần thiết

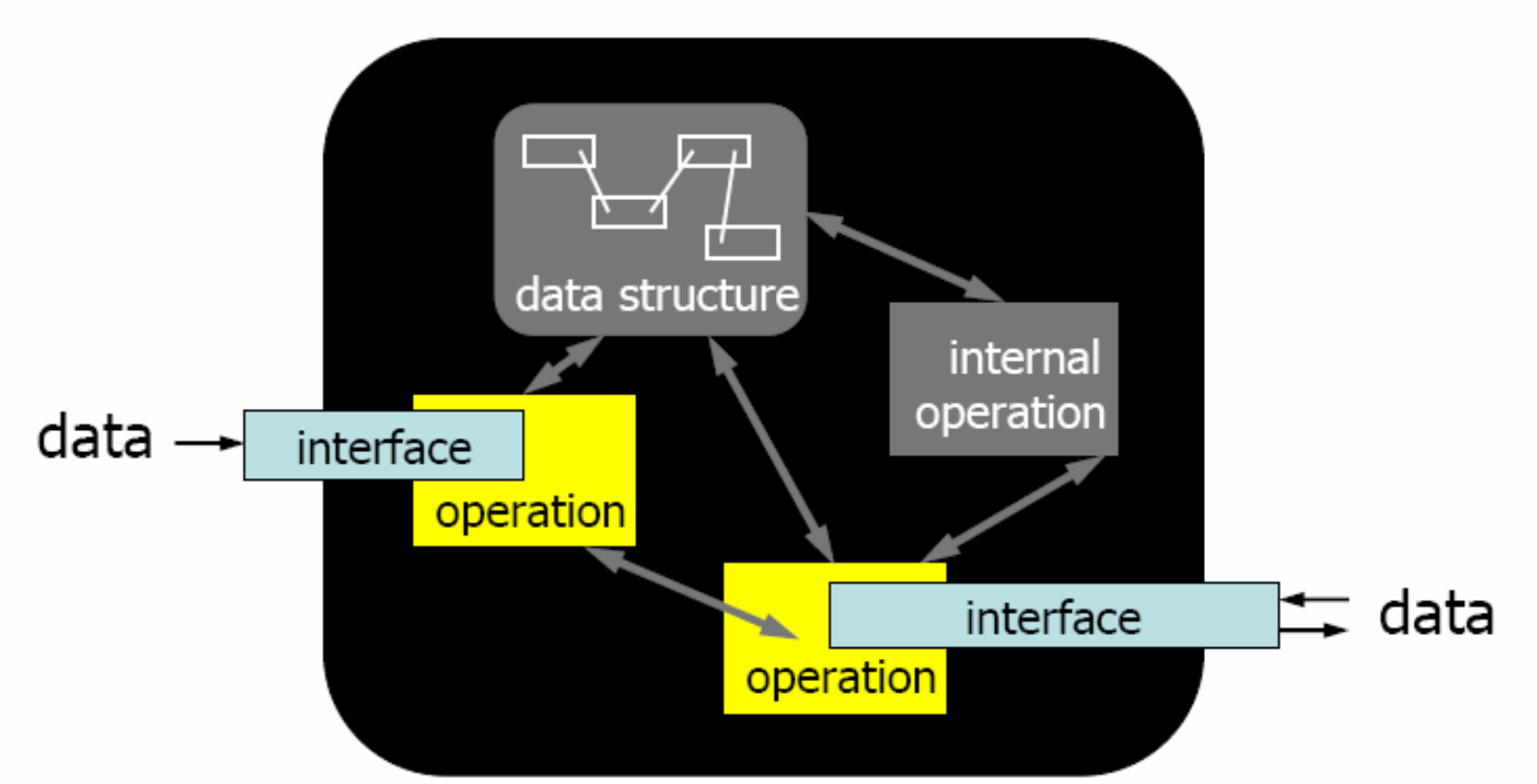
Các kỹ thuật trừu tượng

- Đóng gói (encapsulation)
- Che dấu thông tin (information hiding)
- Thừa kế (inheritance)

Trừu tượng hóa dữ liệu

- Kiểu dữ liệu trừu tượng là kiểu do người dùng định nghĩa thỏa mãn 2 điều kiện sau:
  - Khai báo kiểu và các hành động đối với đối tượng của kiểu → cung cấp một giao diện của kiểu
  - Kiểu của đối tượng thì được giấu đi đối với bên ngoài, cho nên các hành động có thể được cung cấp trong phần định nghĩa kiểu
- VD : các số dấu chấm động

Che dấu thông tin



Ngôn ngữ C++

- Dựa trên kiểu **`struct`** của C và lớp của Simula 67
- Lớp của C++ được xem như là kiểu
- Dữ liệu được định nghĩa trong lớp là dữ liệu thành viên
- Hàm hay phương thức được định nghĩa trong lớp là hàm thành viên
- Tất cả các thực thể của lớp điều có cùng phương thức, nhưng mỗi thực thể thì có dữ liệu riêng
- Thực thể của lớp có thể là tĩnh hoặc động

Ngôn ngữ C++ (tt)

- Dấu thông tin: 3 loại quyền truy xuất đến các thành viên trong lớp
  - *Thành viên riêng (Private)*: truy xuất bởi các thành viên trong lớp
  - *Thành viên chung (Public)*: truy xuất bởi tất cả các thành viên sử dụng lớp
  - *Thành viên bảo vệ (Protected)*: truy xuất bởi các thành viên của lớp dẫn xuất

Ngôn ngữ C++ (tt)

```
class stack {
  private:
     int *stackPtr, maxLen, topPtr;
  public:
     stack() { // a constructor
          stackPtr = new int [100];
          maxLen = 99;
          topPtr = -1;
     };
     ~stack () {delete [] stackPtr;};
     void push (int num) {…};
     void pop () {…};
     int top () {…};
     int empty () {…};
}
```

Ngôn ngữ Java

- Tương tự C++, chỉ trừ:
  - Tất cả các kiểu do người dùng định nghĩa đều là lớp (Java không có **structs, union**)
  - Tất cả các đối tượng được cấp phát vùng nhớ từ Heap và được truy cập bằng tham chiếu biến
  - Từng hàm và biến trong lớp đều có gán quyền truy cập (**private** or **public**) khi khai báo
  - Java có cơ chế phạm vi thứ hai, phạm vi của gói (package scope), VD
    - import com.acme.utils.AcmeIO;
    - import com.acme.utils.*;

Ngôn ngữ Java

```
class StackClass {

      private int [] *stackRef;
      private int [] maxLen, topIndex;
      public StackClass() { // a constructor
            stackRef = new int [100];
            maxLen = 99;
            topPtr = -1;
      };
      public void push (int num) {…};
      public void pop () {…};
      public int top () {…};
      public boolean empty () {…};
}
```

Ngôn ngữ C#

- Dựa trên C++ và Java
- Bổ sung hai quyền truy cập, *internal* và *protected internal*
- Toàn bộ các thực thể lớp đều là Heap dynamic
- Hàm xây dựng mặc định đều có sẵn trong các lớp
  - Khởi tao giá trị mặc định **0 cho int và false cho boolean**
- Vì `garbage collection` được dùng trong hầu hết các heap objects nên hàm hủy ít khi được dùng
- `struct` là hình thức đơn giản của class nên không hỗ trợ thừa kế

Ngôn ngữ C# (tt)

- Giải pháp để truy xuất đến dữ liệu thành viên: cung cấp phương thức getter và setter
- C# cung cấp *property*, như trong Delphi, như là cách cài đặt phương thức getters và setters mà không yêu cầu phương thức gọi hàm tường minh
- Property cung cấp truy xuất không tường minh dữ liệu riêng (private)

Ngôn ngữ C#: Property

```
public class Weather {
public int DegreeDays { //** DegreeDays is a property
     get {return degreeDays;}
     set {degreeDays = value;}
   }
   private int degreeDays;
   ...
   }
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

Sự đóng gối trong C / C++

- C
  - Không hỗ trợ trù tượng hóa dữ liệu
  - Tập tin chứa một hoặc nhiều chương trình con có thể được biên dịch một cách độc lập
  - Giao diện (interface) được dặt trong *header file (.h)*
  - Header file được chèn vào codeSource bằng `#include`
- C++
  - Giống như C
  - Dùng hàm *friend* để truy xuất đến các thành viên riêng của lớp bạn

Sự đóng gối trong C / C++ (tt)

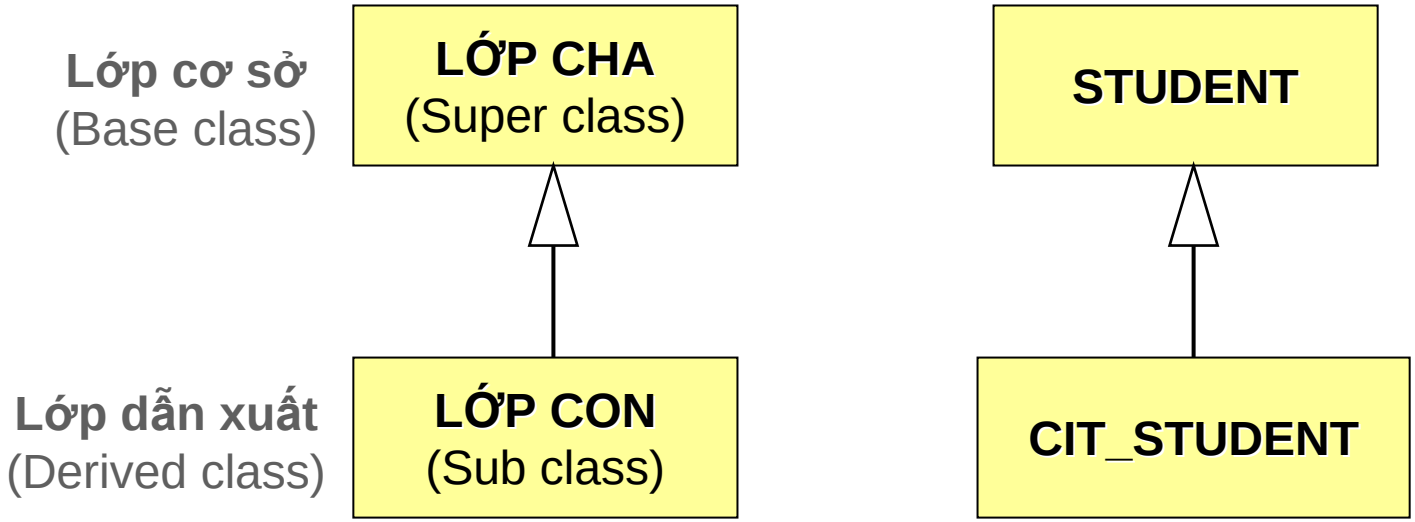
```
Class Matrix //** A class declaration
Class Vector{
  friend Vector multiply(const Matrix&,const Vector&);
  ...
};
Class Matrix{ //** The class definition
  friend Vector multiply(const Matrix&,const Vector&);
  ...
};
//**The function that uses both Matrix and Vector class
Vector multiply(const Matrix&,const Vector&){
...
}
```

C# Assemblies

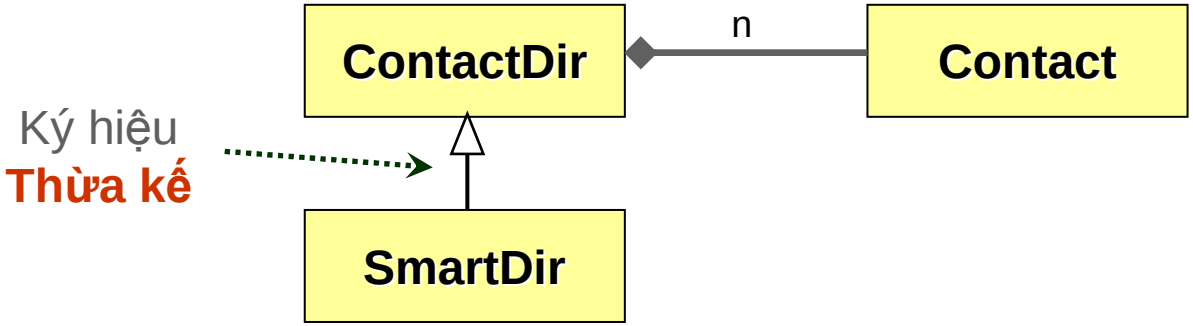
- Tập hợp các files vào trong một thư viện liên kết động DLL (*dynamic link library*) hoặc file thực thi (*executable*)
- Mỗi file có một module được biên dịch độc lập
- Một DLL là tập hợp các lớp và phương thức được liên kết một chương trình thực thi
- C# có cơ chế thay đổi quyền truy xuất, `internal`; một thành viên `internal` của lớp thì được truy xuất bởi tất cả các lớp trong `assembly` mà nó xuất hiện

Tính thừa kế

- Kế thừa từ các lớp có từ trước.
- Ích lợi: có thể tận dụng lại
  - Các thuộc tính chung
  - Các hàm có thao tác tương tự



Ví dụ minh họa (tt)

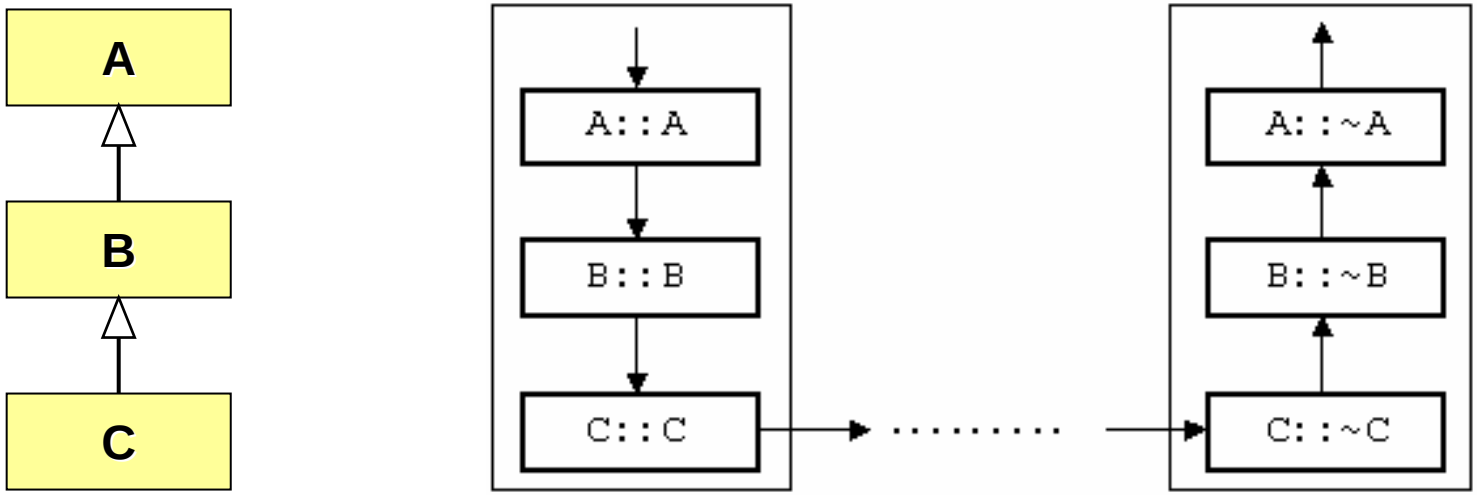



```
class SmartDir : public ContactDir {
    private:
        char     *recent;    // ten duoc tim gan nhat
    public:
        SmartDir(const int max) : ContactDir(max)
            { recent = 0; }
        Contact* Recent (void);
        Contact*  Find  (const char *name);
        // ……..……
};
```

```
Contact* SmartDir::Recent (void) {
      return recent == 0 ? 0 :
                  ContactDir::Find(recent);
}
Contact* SmartDir::Find (const char *name) {
     Contact *c = ContactDir::Find(name);
     if (c != 0)
         recent = (char*) c->Name();
     return c;
}
```

Hàm xây dựng và hàm hủy

- Trong thừa kế, khi khởi tạo đối tượng:
  - Hàm xây dựng của **lớp cha** sẽ được gọi trước
  - Sau đó mới là hàm xây dựng của **lớp con**.
- Trong thừa kế, khi hủy bỏ đối tượng:
  - Hàm hủy của **lớp con** sẽ được gọi trước
  - Sau đó mới là hàm hủy của **lớp cha**.



Hàm xây dựng và hàm hủy (tt)

```
class SmartDir : public ContactDir {
    private:
        char  *recent;  // ten duoc tim gan nhat
    public:
        SmartDir(const int max) : ContactDir(max)
            { recent = 0; }
        SmartDir(const SmartDir& sd): ContactDir(sd)
            { recent = 0; }
        ~SmartDir() {
            delete recent;
          }
        // ……..
};
```

Gọi hàm xây dựng của lớp cha

Thu hồi vùng nhớ của con trỏ thành viên của lớp con nếu đã cấp vùng nhớ trong hàm xây dựng.

Thành viên lớp được bảo vệ

- Thừa kế:
  - Có tất cả các dữ liệu và hàm thành viên.
  - Không được truy xuất đến thành viên **private**.
- Thuộc tính truy cập **protected**:
  - Cho phép lớp con truy xuất.

```
class ContactDir {
    //...
      protected:
          int     Lookup (const char *name);
          Contact  **contacts; // ds cac doi tac
          int     dirSize;     // kich thuoc hien tai
          int  maxSize;    // kich thuoc toi da
};
```

```
class Foo {
      public:
    // cac thanh vien chung...
      private:
    // cac thanh vien rieng...
      protected:
    // cac thanh vien duoc bao ve...
      public:
    // cac thanh vien chung nua...
      protected:
    // cac thanh vien duoc bao ve nua...
};
```

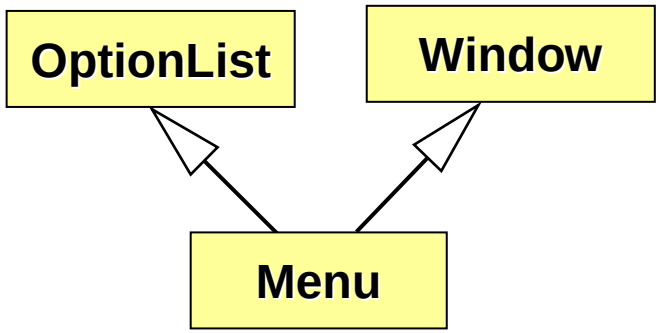
Lớp cơ sở riêng, chung và được bảo vệ

Lớp cơ sở	Thừa kế public	Thừa kế private	Thừa kế protected
private	–	–	–
public	public	private	protected
protected	protected	private	protected

```
class A {
        private:
                int x;
                void Fx (void);
        public:
                int y;
                void Fy (void);
        protected:
                int z;
                void Fz (void);
};
```

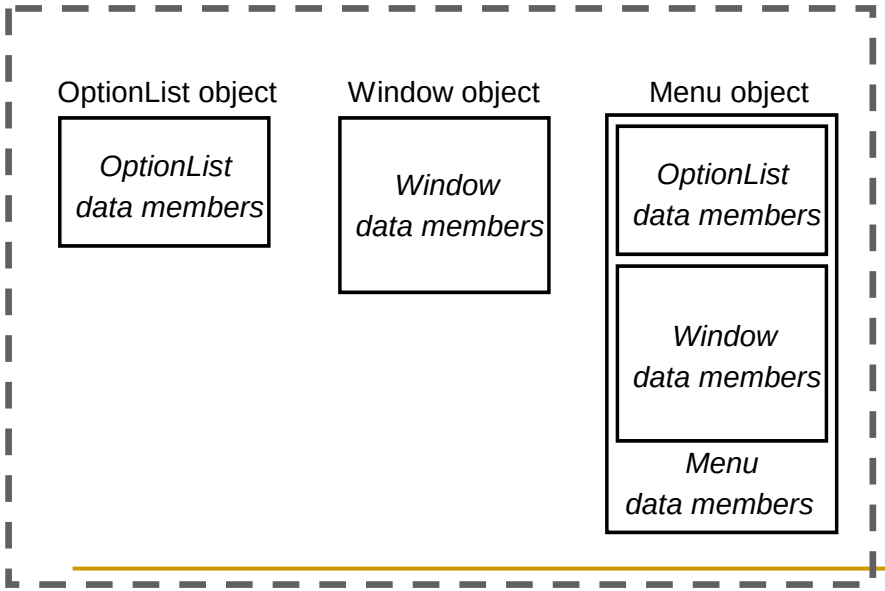
```
class B : A {  // Thừa kế dạng private
        .....
  };
class C : private  A {  // A là lớp cơ sở riêng của B
        .....
  };
class D : public  A {  // A là lớp cơ sở chung của C
        .....
  };
class E : protected  A {  // A: lớp cơ sở được bảo vệ
        .....
 };
```

Đa thừa kế (C++)




```
class OptionList {
    public:
        OptionList (int n);
        ~OptionList ();
        //...
};
```

```
class Window {
    public:
        Window (Rect &);
        ~Window (void);
        //...
};
```




```
class Menu
      : public OptionList, public Window {
    public:
        Menu (int n, Rect &bounds);
        ~Menu (void);
        //...
};

Menu::Menu (int n, Rect &bounds) :
     OptionList(n), Window(bounds)
 { /* ... */ }
```

Ưu khuyết điểm của đa thừa kế

- Khuyết điểm
  - Tạo sự phức tạp trong NN và trong cài đặt (sự mơ hồ: sự phức tạp của các quan hệ thừa kế)
  - Đôi khi không hiệu quả - chi phí liên kết động tăng lên với đa thừa kế
- Ưu điểm
  - Trong vài trường hợp đa thừa kế rất tiện lợi

Sự mơ hồ trong đa thừa kế

```
class OptionList {
    public:
        // ……
        void Highlight (int part);
};
```

```
class Window {
    public:
        // ……
        void Highlight (int part);
};
```

```
class Menu : public OptionList,
             public Window
    { ……. };
```

Hàm cùng tên

Chỉ rõ hàm của lớp nào

Gọi hàm của lớp nào ?

```
void main() {
    Menu  m1(….);
    m1.Highlight(10);
    ….
}
```

xử lý

```
void main() {
    Menu  m1(…);
    m1.OptionList::Highlight(10);
    m1.Window::Highlight(20);
    ….
}
```

Thừa kế trong Java

- Java chỉ hỗ trợ thừa kế đơn

```
public class Circle extends Point {//TK từ Point}
```

- Nhưng sử dụng `interface`, giống như đa thừa kế

```
public interface interfaceName { final
  constantType
     constantName = constantValue; ...
   returnValueType methodName( arguments ); ...
}
public interface interfaceName extends
  superinterfaceName, ... {
   interface body...
}
```

Thừa kế trong Java (tt)

```
public interface Human {
  final String GENDER_MALE = "MALE";
  final String GENDER_FEMALE = "FEMALE";
  void move(); void talk();
}
public abstract class Person implements Human {
  protected int age = 0;
  protected String firstname = "firstname";
  protected String lastname = "lastname";
  protected String gender = Human.GENDER_MALE;
  protected int progress = 0;
  public void move() { this.progress++; }
}
```

Một số NNLThướng đối tượng

- Smalltalk
- C++
- Java
- C#
- Ada 95
- Javascript