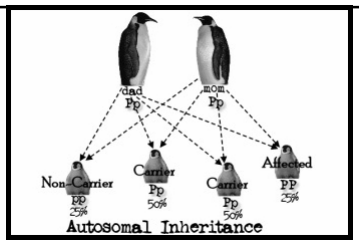


Chương 5: Kế thừa

Cao Tuấn Dũng
Huỳnh Quyết Thắng
Bộ môn CNPM

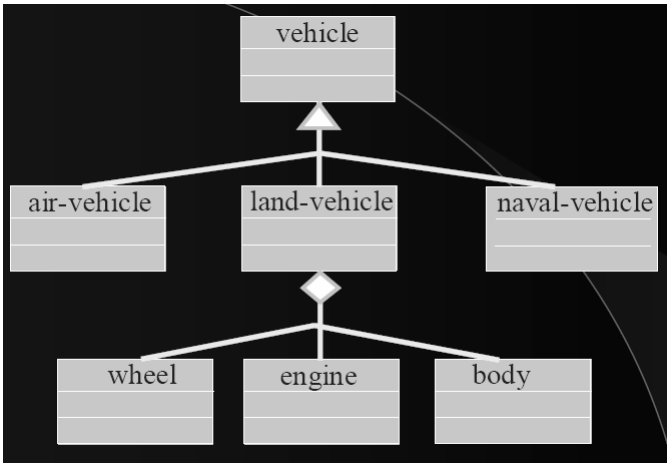
Quan hệ là (is a)

- Ngoài việc nhóm các đối tượng có cùng tập thuộc tính/hành vi lại với nhau, con người thường nhóm các đối tượng có cùng một số (chứ không phải tất cả) thuộc tính/hành vi
- Ví dụ, ta nhóm tất cả xe chạy bằng động cơ thành một nhóm, rồi phân thành các nhóm nhỏ hơn tuỳ theo loại xe (xe ca, xe tải,...)
- Mỗi nhóm con là một lớp các đối tượng tương tự, nhưng giữa tất cả các nhóm con có chung một số đặc điểm
- Quan hệ giữa các nhóm con với nhóm lớn được gọi là quan hệ "là"
  - một cái xe ca "là" xe chạy bằng động cơ
  - một cái xe tải "là" xe chạy bằng động cơ
  - một cái xe máy "là" xe chạy bằng động cơ

Khái niệm: Kế thừa và Kết tập

- Có hai cách tạo ra mối liên hệ giữa hai lớp với mục đích sử dụng lại mã trong LTHDT:
  - Cách thứ nhất (Quan hệ has – a): tạo các đối tượng của các lớp có sẵn trong lớp mới, như vậy lớp mới là sự kết tập giữa lớp mới định nghĩa với các lớp cũ đã có.
  - Cách thứ hai (Quan hệ is-a): tạo lớp mới là một phát triển của lớp có sẵn. Trong lớp mới sẽ sử dụng lại code đã có trong lớp cũ và phát triển những tính chất mới. Cách này gọi là kế thừa.
- Sự khác nhau trong khái niệm: Trong kết tập tái sử dụng thông qua lại đối tượng, Trong kế thừa tái sử dụng thông qua lớp

Kế thừa và kết tập



Ví dụ về Kết tập

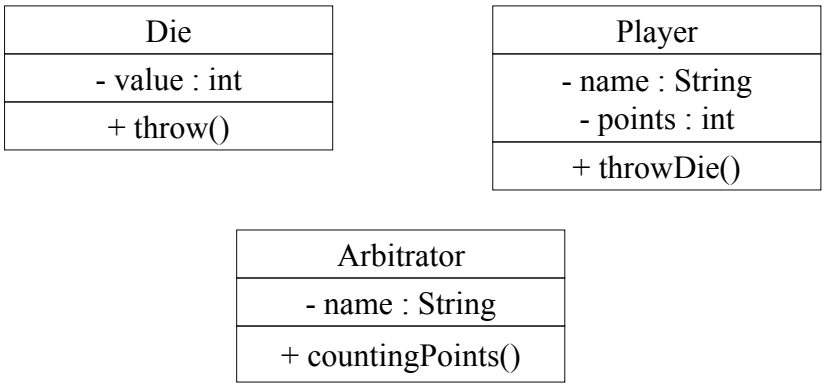
- Một trò chơi gồm các đối thủ, ba quân súc sắc và 1 trọng tài.
- Cần bốn lớp:
  - người chơi
  - súc sắc
  - trọng tài
  - trò chơi

```
class Game
{
        Die die1, die2, die3;
        Player player1, player2;
        Arbitrator arbitrator1;

        …
}
```

- Lớp Trò chơi được coi là lớp khách hàng của ba lớp còn lại

Ví dụ về Kết tập



Khởi tạo và huỷ bỏ đối tượng trong kết tập và kế thừa

- Khi một đối tượng được tạo mới, chương trình dịch đảm bảo rằng tất cả các thuộc tính của nó đều phải được khởi tạo và gán những giá trị tương ứng.
- Trong kết tập và kế thừa, nảy sinh ra vấn đề khi chúng ta khởi tạo một đối tượng có kết tập hoặc một đối tượng thuộc một lớp kế thừa thì cần phải khởi tạo các đối tượng thuộc các lớp đã định nghĩa: như thế cần phải định nghĩa chúng sẽ được khởi tạo như thế nào và theo thứ tự nào.

Khởi tạo và huỷ bỏ đối tượng trong kết tập và kế thừa

- Thứ tự khởi tạo trong kết tập:
  - Các đối tượng thành phần được khởi tạo trước
  - Các hàm khởi tạo của các lớp của các đối tượng thành phần được thực hiện trước
  - Các hàm huỷ thực hiện theo thứ tự ngược lại
- Thứ tự khởi tạo trong kế thừa
  - Các hàm khởi tạo của các lớp cơ sở được thực hiện trước
- Các hàm huỷ thực hiện theo thứ tự ngược lại

Khởi tạo và huỷ bỏ đối tượng trong kết tập và kế thừa

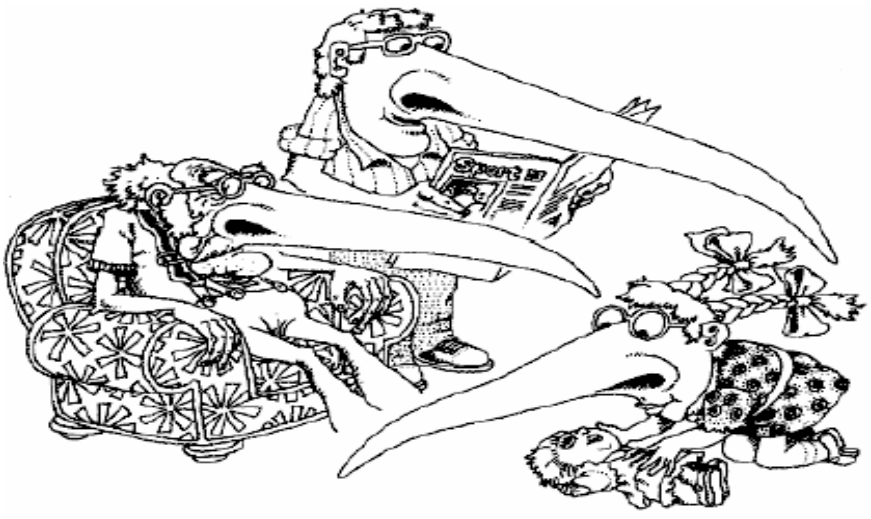
- Truyền các giá trị các đối số cho các hàm khởi tạo
  - Khi khai báo các hàm khởi tạo của các kế thừa chung ta có quyền đăng ký sử dụng các hàm khởi tạo của các lớp cơ sở với các đối số tương ứng
  - Tương tự khi khai báo một hàm khởi tạo của một lớp kết tập, chúng ta cũng có quyền khai báo các đối tượng thành phần của lớp kết tập được khởi tạo như thế nào.
  - Lưu ý: Trong cả hai trường hợp phải sử dụng các hàm khởi tạo đã đăng ký của lớp tương ứng

Kế thừa

- Nguyên lý mô tả một lớp trên cơ sở mở rộng/cụ thể hơn một lớp đã tồn tại, hay nhiều lớp (trong trường hợp đa thừa kế)
- Trên cách nhìn mô đun hóa: Nếu B thừa kế A, mọi dịch vụ của A sẽ sẵn có trong B (theo các cách thực hiện khác nhau)
- Trên cách nhìn xuất phát từ kiểu: Nếu B thừa kế A, bất cứ khi nào một thể hiện của A được yêu cầu, thể hiện của B có thể là một đáp ứng.

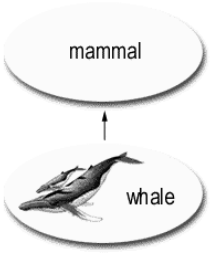
Kế thừa

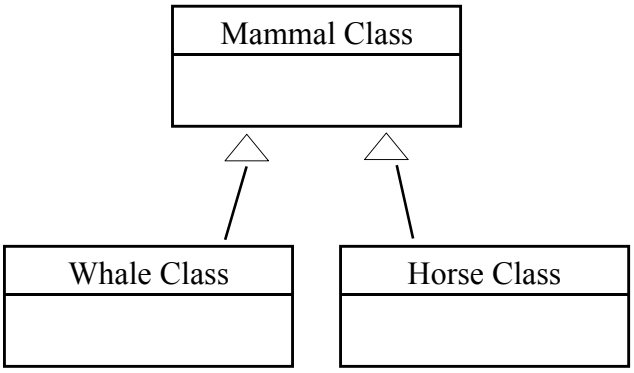
- *Inheritance xác định 1 quan hệ (*relationship) giữa các lớp khi 1 lớp chia sẻ cấu trúc và/hoặc hành vi của 1 hay nhiều lớp khác
- 1 cây phả hệ bởi các lớp được tạo ra trong đó 1 lớp con - *subclass* kế thừa từ 1 hay nhiều lớp cha - *superclasses*
- Kế thừa còn được gọi là quan hệ là : *is-a* .



Tính tương đồng

- Lớp cá voi kế thừa từ lớp động vật có vú.
- 1 con cá voi là 1 đv có vú (*is-a* mammal)
- Lớp cá voi là *subclass*, lớp DVCV là *superclass*



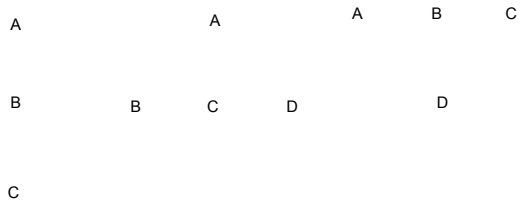


Kế thừa

- Cả Whale và Horse có quan hệ *is-a* với mammal class
- Cả Whale và Horse có 1 số hành vi thông thường của Mammal
- Inheritance là chìa khóa để tái sử dụng code – Nếu 1 lớp cha đã được tạo, thì lớp con có thể được tạo và thêm vào một số thông tin

Lớp cơ sở và lớp dẫn xuất

- Một lớp được xây dựng từ một lớp khác gọi là lớp dẫn xuất; lớp dùng để xây dựng lớp dẫn xuất gọi là lớp cơ sở.
- Bất cứ lớp nào cũng có thể trở thành một lớp cơ sở; hơn nữa một lớp có thể là cơ sở cho nhiều lớp dẫn xuất. Đến lượt mình, một lớp dẫn xuất lại có thể dùng làm cơ sở để xây dựng các lớp dẫn xuất khác. Ngoài ra một lớp có thể dẫn xuất từ nhiều lớp cơ sở.
- Như vậy mối quan hệ cơ sở và dẫn xuất là rất đa dạng, sau đây là một số sơ đồ mô tả mối quan hệ trên:



Lớp cơ sở và lớp dẫn xuất

Tổng quát

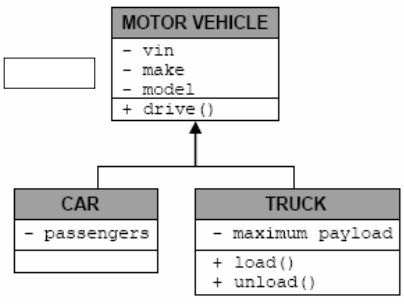
<u>Tính thừa kế</u> Một lớp dẫn xuất ngoài các thành phần của riêng mình, nó còn được thừa kế tất cả các thành phần của các lớp cơ sở liên quan.

Lớp cơ sở và lớp dẫn xuất

- Các thuộc tính của lớp cơ sở được thừa kế trong lớp dẫn xuất; điều này có nghĩa là tập thuộc tính của lớp dẫn xuất sẽ bao gồm: các thuộc tính mới khai báo trong lớp dẫn xuất và các thuộc tính của lớp cơ sở.
- Tuy vậy trong lớp dẫn xuất không được phép truy nhập đến các thuộc tính private của lớp cơ sở.

Sơ đồ quan hệ đối tượng

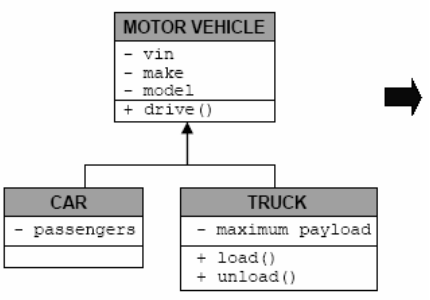
- Mục đích là để chỉ rõ sự khác biệt giữa các lớp tham gia quan hệ đó
  - một lớp con khác lớp cha của nó ở chỗ nào?
  - các lớp con khác nhau ở chỗ nào?



Sơ đồ quan hệ đối tượng

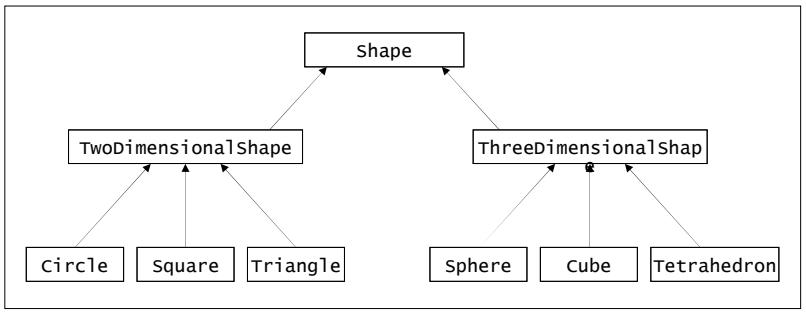
- Biểu diễn sự khác biệt giữa các lớp như thế nào?
  - nếu không có gì khác nhau thì chẳng có lý do gì để lập lớp con
  - Giả sử, xe ca có thêm thuộc tính *passengers* (số hành khách tối đa mà xe có thể chở); xe tải có thêm thuộc tính *maximum payload* (trọng tải tối đa) và các hành vi *load* (bốc), *unload* (dỡ).
- Khi biểu diễn các thuộc tính và hành vi của các lớp con, chỉ cần liệt kê các thuộc tính/hành vi mà lớp cha không có
  - đơn giản hoá sơ đồ, không lặp lại các thuộc tính/hành vi được thừa kế (có thể tìm thấy chúng bằng cách "lần theo mũi tên")
  - nhấn mạnh các điểm khác biệt, cho phép dễ dàng nhận ra lý do cho việc lập lớp con

Sơ đồ quan hệ đối tượng



- mỗi xe ca đều có các thuộc tính *vin*, *make*, *model*, và hành vi *drive,* kèm theo thuộc tính *passengers*
- mỗi xe tải đều có các thuộc tính *vin*, *make*, *model*, và hành vi *drive,* kèm theo thuộc tính *maximum payload* và các hành vi *load*, *unload*

Cây thừa kế



Các quan hệ thừa kế luôn được biểu diễn với các lớp con đặt dưới lớp cha để nhấn mạnh bản chất phả hệ của quan hệ

Lớp trừu tượng (Abstract Class)

- Lớp trừu tượng là lớp mà ta không thể tạo ra các đối tượng từ nó. Thường lớp trừu tượng được dùng để định nghĩa các "khái niệm chung", đóng vai trò làm lớp cơ sở cho các lớp "cụ thể" khác.
- Đi cùng từ khóa abstract

```
public abstract class Product
{
    // contents
}
```

Lớp trừu tượng (Abstract Class)

- Lớp trừu tượng có thể chứa các phương thức trừu tượng không được định nghĩa
- Các lớp dẫn xuất có trách nhiệm định nghĩa lại (overriding) các phương thức trừu tượng này
- Sử dụng các lớp trừu tượng đóng vai trò quan trọng trong thiết kế phần mềm. Nó cho phép định nghĩa tạo ra những phần tử dùng chung trong cây thừa kế, nhưng quá khái quát để tạo ra các thể hiện.

Đặc điểm về tính
kế thừa trong C++

Khai báo kế thừa

```
class DerivedClass : access-specifier BaseClass
{
  // Body of the derived class
};
```

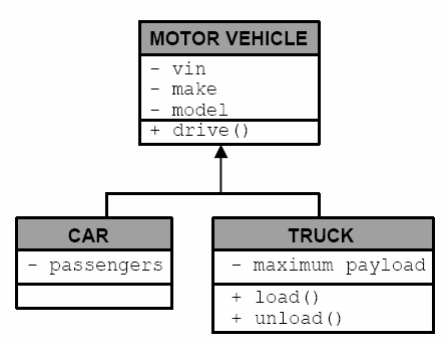
DerivedClass: lớp dẫn xuất
BaseClass: lớp cơ sở
Access-specifier: public, protected, private

Khai báo kế thừa

Giả sử đã có sẵn các lớp A và B; để xây dựng lớp C dẫn xuất từ A và b ta có thể viết như sau:

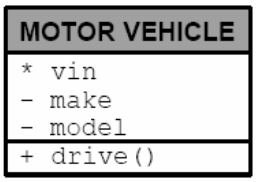
```
class  C : public A, public B
   {
   private:
        // khai báo các thuộc tính
   public:
        // các phương thức
};
```

Ví dụ MotorVehicle



Ví dụ lớp MotorVehicle

- Bắt đầu bằng định nghĩa lớp cơ sở, MotorVehicle




```
class MotorVehicle {
public:
    MotorVehicle(int vin, string make, string model);
    ~MotorVehicle();
    void drive(int speed, int distance);
private:
    int vin;
    string make;
    string model;
};
```

Ví dụ lớp MotorVehicle

- Tiếp theo, định nghĩa constructor, destructor, và hàm drive() (ở đây, ta chỉ định nghĩa tạm drive())

```
MotorVehicle::MotorVehicle(int vin, string make, string model)
{
    this->vin = vin;
    this->make = make;
    this->model = model;
}
MotorVehicle::~MotorVehicle() {}
void MotorVehicle::drive(int speed, int distance)
{
        cout << "Dummy drive() of MotorVehicle." << endl;
}
```


Định nghĩa lớp con

- Mô tả một lớp con cũng giống như biểu diễn nó trong sơ đồ đối tượng quan hệ, ta chỉ tập trung vào những điểm khác với lớp cha
- Ích lợi
  - đơn giản hoá khai báo lớp,
  - hỗ trợ nguyên lý đóng gói của hướng đối tượng
  - hỗ trợ tái sử dụng code (sử dụng lại định nghĩa của các thành viên dữ liệu và phương thức)
  - việc che dấu thông tin cũng có thể có vai trò trong việc tạo cây thừa kế

Định nghĩa lớp con Car

Chỉ rõ quan hệ giữa lớp con **Car** và lớp cha **MotorVehicle**

```
class Car : public MotorVehicle
{
    public:
        Car (int passengers);
        ~Car();
    private:
        int passengers;
};
```


Lớp Car

- Bổ sung thêm các hàm thiết lập cần thiết

Quy ước: đặt các tham số cho lớp cha lên đầu danh sách.

```
class Car : public MotorVehicle
{
public:
    Car (int vin, string make, string model, int passengers);
    ~Car();
private:
        int passengers;
};
```

```
Car::Car(int vin, string make, string model, int passengers)
{
    this->vin = vin;
    this->make = make;
    this->model = model;
    this->passengers = passengers;
}
Car::~Car() {}
```

Định nghĩa lớp con

- Nhược điểm: trực tiếp truy nhập các thành viên dữ liệu của lớp cơ sở
  - thiếu tính đóng gói : phải biết sâu về chi tiết lớp cơ sở và phải can thiệp sâu
  - không tái sử dụng mã khởi tạo của lớp cơ sở
  - không thể khởi tạo các thành viên private của lớp cơ sở do không có quyền truy nhập
- Nguyên tắc: một đối tượng thuộc lớp con bao gồm một đối tượng lớp cha cộng thêm các tính năng bổ sung của lớp con
  - một thể hiện của lớp cơ sở sẽ được tạo trước, sau đó "gắn" thêm các tính năng bổ sung của lớp dẫn xuất
- Vậy, ta sẽ sử dụng constructor của lớp cơ sở.

Định nghĩa lớp con

- Để sử dụng constructor của lớp cơ sở, ta dùng danh sách khởi tạo của constructor (tương tự như khi khởi tạo các hằng thành viên)
  - cách duy nhất để tạo phần thuộc về thể hiện của lớp cha tạo trước nhất

```
Car::Car(int vin, string make, string model, int passengers)
: MotorVehicle(vin, make, model)
{
    this->passengers = passengers;
}
```

Gọi constructor của **MotorVehicle** với các tham số **vin**, **make**, **model**

Ta không cần khởi tạo các thành viên **vin**, **make**, **model** từ bên trong constructor của **Car** nữa

Định nghĩa lớp con

- Để đảm bảo rằng một thể hiện của lớp cơ sở luôn được tạo trước, nếu ta bỏ qua lời gọi constructor lớp cơ sở tại danh sách khởi tạo của lớp dẫn xuất, trình biên dịch sẽ tự động chèn thêm lời gọi constructor mặc định của lớp cơ sở
- Tuy ta cần gọi constructor của lớp cơ sở một cách tường minh, tại destructor của lớp dẫn xuất, lời gọi tượng tự cho destructor của lớp cơ sở là không cần thiết
  - việc này được thực hiện tự động

Định nghĩa lớp con

- Tương tự chúng ta có định nghĩa lớp Truck như sau

```
class Truck : public MotorVehicle {
public:
      Truck (int vin, string make, string model,
      int maxPayload);
      ~Truck();
      void Load();
      void Unload();
private:
            int maxPayload;
};
```

```
Truck::Truck(int vin, string make, string model,
int maxPayload)
: MotorVehicle(vin, make, model)
{
this->maxPayload = maxPayload;
}
Truck::~Truck() {}
void Truck::Load() {…}
void Truck::Unload() {…}
```

Truy cập tới thành viên kế thừa

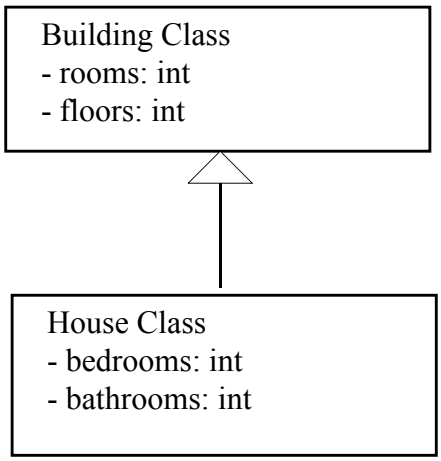
- Truy cập tới các thành viên được kế thừa phụ thuộc không chỉ vào việc chúng được khai báo với từ khóa public, protected hay private trong base class.
- Access còn phụ thuộc vào kiểu kế thừa - public, protected hay private – Kiểu kế thừa được xác định trong **định nghĩa của lớp dẫn xuất.**

Kế thừa public

- Public inheritance có nghĩa:
  - public members của base class trở thành public members của derived class;
  - protected members của base class trở thành protected members của derived class;
  - private members của base class không thể truy cập được trong derived class.

Ví dụ

- Giả sử có hai lớp kế thừa sau




```
class Building
{
public:
    void setRooms(int numRooms);
    int getRooms() const;
    void setFloors(int numFloors);
    int getFloors() const;
private:        // ??
    int rooms;   // Number of rooms
    int floors;   // Number of floors
};
class House : public Building
{
public:
    void setBedrooms(int numBedrooms);
    int getBedrooms() const;
    void setBathrooms(int numBathrooms);
    int getBathrooms() const;
private:
     int bedrooms;   // Number of bedrooms
     int bathrooms;   // Number of bathrooms
};
```

Không truy cập được từ lớp Dẫn xuất => chuyển thành : protected

Lớp MotorVehicle và Truck

- Quay lại cây thừa kế với MotorVehicle là lớp cơ sở
  - Mọi thành viên dữ liệu đều được khai báo **private**, do đó chúng *chỉ có thể* được truy nhập từ các thể hiện của MotorVehicle
  - tất cả các lớp dẫn xuất không có quyền truy nhập các thành viên private của MotorVehicle
- Vậy, đoạn mã sau sẽ có lỗi

```
class MotorVehicle {
...
private:
      int vin;
      string make;
      string model;
};
```

```
void Truck::Load() {
          if (this->make == "Ford") {
          ...
          }
}
```


Lớp MotorVehicle và Truck

- Giả sử ta muốn các lớp con của MotorVehicle có thể truy nhập dữ liệu của nó
- Thay từ khoá **private** bằng **protected**,
- Đoạn mã sau sẽ không còn lỗi

```
class MotorVehicle {
...
protected:
      int vin;
      string make;
      string model;
};
```

```
void Truck::Load() {
          if (this->make == "Ford") {
          ...
          }
}
```

Kế thừa protected

- Protected inheritance có nghĩa:
  - public members của base class trở thành **protected** members của derived class;
  - protected members của base class trở thành protected members của derived class;
  - private members của base class không thể truy cập được trong derived class.
  - quan hệ thừa kế sẽ được nhìn thấy từ
    - mọi phương thức bên trong Car,
    - mọi phương thức thuộc các lớp con của Ca
  - Tuy nhiên, mọi đối tượng khác của C++ không nhìn thấy quan hệ này

Private inheritance

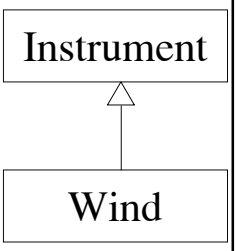
- Private inheritance có nghĩa:
  - public members của base class trở thành **private** members của derived class;
  - protected members của base class trở thành **private** của derived class;
  - private members của base class không thể truy cập được trong derived class.
  - chỉ có chính thể hiện đó biết nó được thừa kế từ lớp cha
  - các đối tượng bên ngoài không thể tương tác với lớp con như với lớp cha, vì mọi thành viên được thừa kế đều trở thành **private** trong lớp con
  - Có thể dùng thừa kế **private** để tạo lớp con có mọi chức năng của lớp cha nhưng lại không cho bên ngoài biết về các chức năng đó.

Sự tương thích về kiểu

- Một thể hiện của một lớp dẫn xuất public có thể được dùng ở bất cứ đâu mà một thể hiện (instance) của lớp cơ sở của nó được chấp nhận.
- Một thể hiện của một lớp dẫn xuất không thể thế chỗ cho 1 thể hiện của lớp cơ sở của nó nếu dùng kiểu kế thừa protected hay private.
- Hãy luôn dùng kiểu kế thừa public trừ khi có nguyên nhân rõ ràng để không dùng kiểu đó.

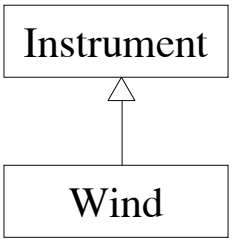
Các vấn đề trong kế thừa

- Chuyển đổi một đối tượng thuộc lớp thừa kế thành đối tượng thuộc lớp cơ sở được gọi là "upcasting", sử dụng một biến đã khai báo thuộc lớp cơ sở.
- Mọi thông điệp mà ta có thể gửi cho lớp cơ sở đều có thể gửi cho lớp thừa kế.



Upcasting

```
class Instrument {
    public:
       void play(note) const {}
 };
 class Wind : public Instrument {};
 void tune(Instrument& i) {
     i.play(middleC);
 }
 int main() {
   Wind flute;
   tune(flute); // Upcasting
}
```



Upcast

- Các thể hiện của lớp con thừa kế public có thể được đối xử như thể nó là thể hiện của lớp cha.
  - từ thể hiện của lớp con, ta có quyền truy nhập các thành viên và phương thức public mà ta có thể truy nhập từ một thể hiện của lớp cha.
- Do đó, C++ cho phép dùng con trỏ được khai báo thuộc loại con trỏ tới lớp cơ sở để chỉ tới thể hiện của lớp dẫn xuất
  - ta có thể thực hiện các lệnh sau:

```
MotorVehicle* mvPointer2;
mvPointer2 = mvPointer; // Point to another MotorVehicle
mvPointer2 = cPointer; // Point to a Car
mvPointer2 = tPointer; // Point to a Truck
```

Upcast

- Điều đáng lưu ý là ta thực hiện tất cả các lệnh gán đó mà *không cần* đổi kiểu tường minh
  - do mọi lớp con của MotorVehicle đều chắc chắn có mọi thành viên và phương thức có trong một MotorVehicle, việc tương tác với thể hiện của các lớp này như thể chúng là MotorVehicle không có chút rủi ro nào
  - Ví dụ, lệnh sau đây là hợp lệ, bất kể mvPointer2 đang trỏ tới một MotorVehicle, một Car, hay một Truck

```
mvPointer2->Drive();
```


Upcast

- Upcast thường gặp tại các định nghĩa hàm, khi một con trỏ/tham chiếu đến lớp cơ sở được yêu cầu, nhưng con trỏ/tham chiếu đến lớp dẫn xuất cũng được chấp nhận
  - xét hàm sau

```
void sellMyVehicle(MotorVehicle& myVehicle)
{...}
```

  - có thể gọi `sellMyVehicle` một cách hợp lệ với tham số là một tham chiếu tới một MotorVehicle, một Car, hoặc một Truck.

Upcast

- Nếu ta dùng một con trỏ tới lớp cơ sở để trỏ tới một thể hiện của lớp dẫn xuất, trình biên dịch sẽ chỉ cho ta coi đối tượng như thể nó thuộc lớp cơ sở
- Như vậy, ta không thể làm như sau

```
MotorVehicle* mvPointer2 = tPointer;
// Point to a Truck
mvPointer2->Load(); // Error
```

  - Đó là vì trình biên dịch không thể đảm bảo rằng con trỏ thực ra đang trỏ tới một thể hiện của Truck.

Upcast

- Chú ý rằng khi gắn một con trỏ/tham chiếu lớp cơ sở với một thể hiện của lớp dẫn xuất, ta không hề thay đổi bản chất của đối tượng được trỏ tới
  - Ví dụ, lệnh

    ```
    MotorVehicle* mvPointer2 = tPointer;
    ```

    không làm một thể hiện của Truck suy giảm thành một MotorVehicle, nó chỉ cho ta một cách nhìn khác đối với đối tượng Truck và tương tác với đối tượng đó.
  - Do vậy, ta vẫn có thể truy nhập tới các thành viên và phương thức của lớp dẫn xuất ngay cả sau khi gán con trỏ lớp cơ sở tới nó:

```
mvPointer2 = tPointer; // Point to a Truck
tPointer->Load(); // We can still do this
mvPointer2->Load(); // (error)
```

Downcast

- **Downcast** là quy trình ngược lại: đổi kiểu con trỏ/tham chiếu tới lớp cơ sở thành con trỏ/tham chiếu tới lớp dẫn xuất.
- downcast là quy trình rắc rối hơn và có nhiều điểm không an toàn

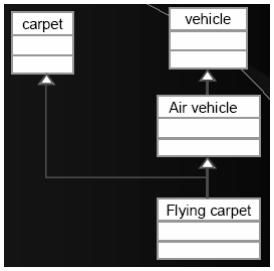
Downcast

```
Truck* tPointer = new Truck(10, "Toyota", "Tacoma",
  5000);
MotorVehicle* mv = tPointer; // Upcast
Car* cPointer2;
cPointer = static_cast<Car*>(mv); // Explicit
  downcast
```

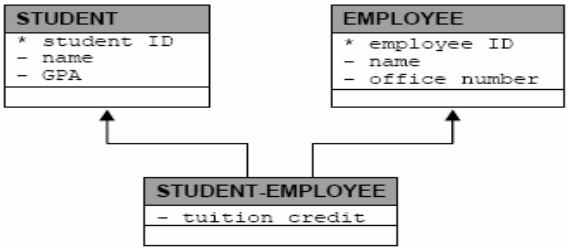
- Đoạn mã trên hoàn toàn hợp lệ và sẽ được trình biên dịch chấp nhận
- Tuy nhiên, nếu chạy đoạn trình trên, chương trình có thể bị đổ vỡ (thường là khi lần đầu truy nhập đến thành viên/phương thức được định nghĩa của lớp dẫn xuất mà ta đổi tới)

Đa thừa kế

- Đa thừa kế là dạng thừa kế phức tạp cho phép ta tạo các lớp dẫn xuất thừa kế các thuộc tính và hành vi từ *nhiều hơn một lớp cơ sở*.
- Nên hạn chế sử dụng đa thừa kế vì nó có thể dẫn đến đủ loại nhầm lẫn do các tên trùng nhau
  - chẳng hạn nếu thừa kế từ hai lớp cùng có phương thức Foo()
  - Do tính phức tạp tiềm tàng của đa thừa kế, một số ngôn ngữ lập trình (chẳng hạn Java, C#) không cài đặt tính năng này.

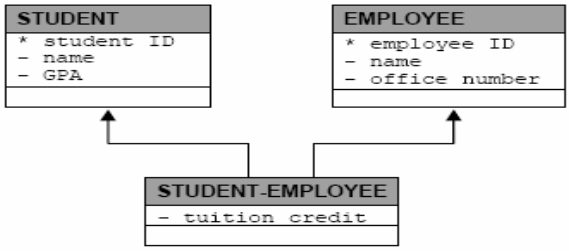


Đa thừa kế



- Xét một hệ thống quản lý đại học lưu trữ thông tin về cả sinh viên và nhân viên. Có thể có những người cùng lúc vừa là sinh viên vừa là nhân viên
  - có thể sinh viên-nhân viên có thêm thuộc tính nợ học phí (tuition credit)

Đa thừa kế



- Ta có thể thấy các trùng lặp tiềm tàng trong các thành viên dữ liệu, chẳng hạn lớp *Student-Employee* thừa kế thành viên *name* từ cả hai lớp cơ sở
  - trong C++, có thể giải quyết rắc rối do trùng tên này bằng toán tử phạm vi (cho trình biên dịch biết ta đang nói đến *name* nào

Đa thừa kế - cài đặt

- Khai báo một lớp là dẫn xuất của nhiều lớp cơ sở:

```
class StudentEmployee : public Student, public Employee {
...
};
```

- Tại định nghĩa lớp dẫn xuất, constructor gọi constructor của các lớp cơ sở:

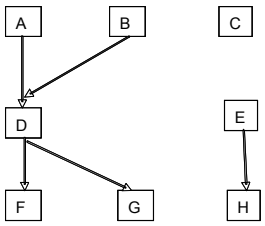
```
StudentEmployee::StudentEmployee (...)
   : Student(...), Employee(...) {
...
};
```

- Khi gặp mù mờ về tên (chẳng hạn 2 phiên bản *name* thừa kế từ 2 lớp cơ sở trong ví dụ trước) ta có thể sử dụng toán tử phạm vi

```
this->Student::name // Refers to the version of name
                    // inherited from Student
this->Employee::name    // Refers to this version of name
                        // inherited from Employee
```

Trùng tên trong đa thừa kế

- Lớp D dẫn xuất từ hai lớp A và B. Lớp E dẫn xuất từ C …
- Các thành phần có thể sử dụng trong G bao gồm:
  - Các thành phần khai báo trong G
  - Các thành phần khai báo trong các lớp D, E, A, B, C (được thừa kế)




```
D h; // h  là đối tượng của lớp D dẫn xuất từ A và B.
h.A::n  là thuộc tính thừa kế từ A
h.D::n  là thuộc tính n khai báo trong D
h.D::nhap()  là  phương thức định nghĩa trong D
h.A::nhap()  là  phương thức định nghĩa trong A
```


Đa thừa kế

- Trong thực tế, có một loạt các vấn đề tiềm tàng liên quan đến đa thừa kế, phần lớn là do rắc rối vì mù mờ
- Có thể ta không bao giờ cần dùng đến đa thừa kế, nhưng cũng có những tình huống mà đa thừa kế là lời giải tốt nhất (và có thể là duy nhất)
- Nếu sử dụng đa thừa kế, nhất thiết phải cân nhắc về các xung đột có thể nảy sinh trong khi sử dụng các lớp có liên quan.

Kế thừa trong Java

Kế thừa trong ngôn ngữ Java

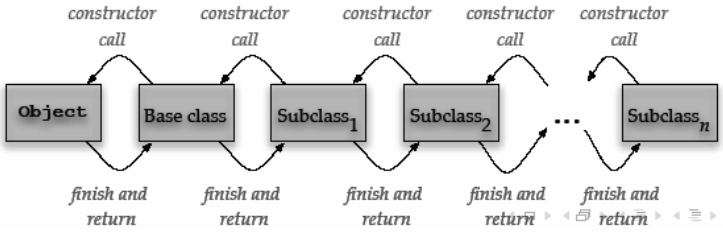
- Khai báo lớp kế thừa (dẫn xuất) qua từ khóa extends
- Chỉ hỗ trợ đơn kế thừa
- Tuân thủ nguyên lý kế thừa: Các đối tượng thuộc lớp dẫn xuất thừa hưởng tất cả các thuộc tính và phương thức của lớp cơ sở
- Có thể gọi tới phương thức constructor của lớp cha từ lớp con với từ khóa super()
- Một lớp đi với khai báo final không thể có lớp dẫn xuất từ nó.

Lớp Object

- Lớp có tên `Object` định nghĩa trong package chuẩn `java.lang`
- Tất cả mọi lớp đều dẫn xuất từ lớp `Object` c
- Nếu một lớp không khai báo kế thừa tường minh một lớp nào đó, điều đó có nghĩa nó là lớp con của lớp `Object`

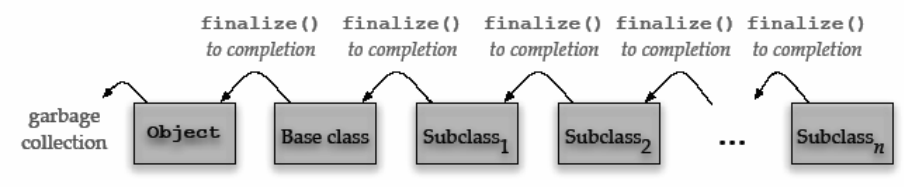
Lời gọi constructor không tường minh

- Khi khởi tạo một đối tượng, một chuỗi các lời gọi hàm thiết lập sẽ được thực hiện một cách tường minh (qua lời gọi tới phương thức super() hoặc không tường minh một cách tự động
- Lời gọi hàm thiết lập của lớp cơ sở cao nhất trong cây kế thừa sẽ được thực hiện sau cùng, nhưng kết thúc trước. Hàm thiết lập của lớp dẫn xuất sẽ kết thúc cuối cùng.

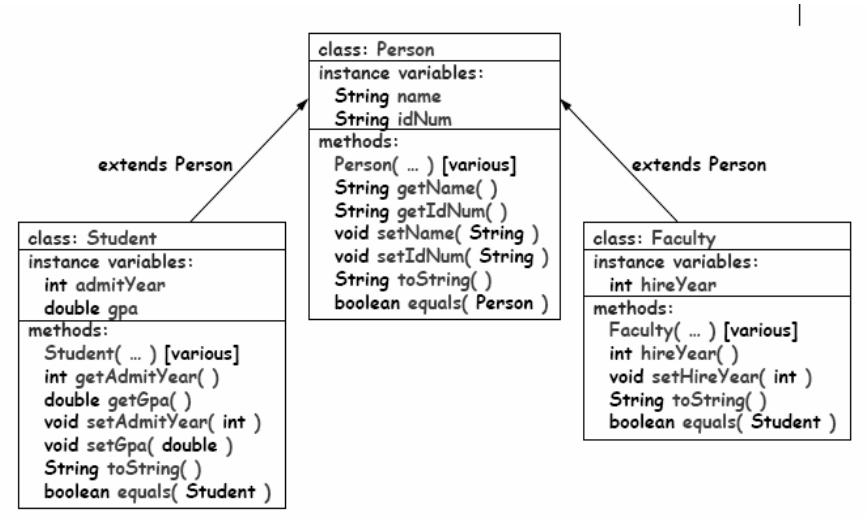


Lời gọi finalize() không tường minh

- Khi một đối tượng bị hủy (thu dọn bởi GC) một chuỗi các phương thức finalize() sẽ được tự động thực hiện.
- Trình tự ngược lại với chuỗi lời gọi hàm thiết lập
  - Phương thức finalize() của lớp dẫn xuất được gọi đầu tiên, sau đó đến lớp cha của nó,...



Person, Student và Faculty



Lớp Faculty

```
package university;
public class Faculty extends Person {
  private int hireYear;
  public Faculty( ) { super( ); hireYear = -1; }
  public Faculty( String n, String id, int yr ) {
      super(n, id);
      hireYear = yr;
  }
  public Faculty( Faculty f ) {
  this( f.getName( ), f.getIdNum( ), f.hireYear );
  }
  int getHireYear( ) { return hireYear; }
  void setHireYear( int yr ) { hireYear = yr; }
  public String toString( ) {
      return super.toString( ) + " " + hireYear;
  }
  public boolean equals( Faculty f ) {
      return super.equals( f ) && hireYear == f.hireYear;
  }
}
```


Định nghĩa lại phương thức (overriding)

- Xảy ra khi lớp kế thừa muốn thay đổi chức năng của môt phương thức kế thừa từ lớp cha (super).

```
public class Person {
…
  public String toString( ) { … }
}
public class Student extends Person {
…
  public String toString( ) { … }
}
Student bob = new Student("Bob Goodstudent","123-45-
  6789",2004,4.0 );
System.out.println( "Bob's info: " + bob.toString( ) );
```

Định nghĩa lại phương thức của lớp cha

Lời gọi đến phương thức của lớp con

Cấm định nghĩa lại với final

- Đôi lúc ta muốn hạn chế việc định nghĩa lại vì các lý do sau:
  - Tính đúng đắn: Định nghĩa lại một phương thức trong lớp dẫn xuất có thể làm sai lạc ý nghĩa của nó
  - Tính hiệu quả: Cơ chế kết nối động không hiệu quả về mặt thời gian bằng kết nối tĩnh. Nếu biết trước sẽ không định nghĩa lại phương thức của lớp cơ sở thì nên dùng từ khóa final đi với phương thức

```
public final String baseName () {
return "Person";}
}
```


Lớp cơ sở ship

```
public class Ship {
  public double x=0.0, y=0.0, speed=1.0,
  direction=0.0;
  public String name;
  public Ship(double x, double y, double speed, double
  direction, String name) {
      this.x = x;
      this.y = y;
      this.speed = speed;
      this.direction = direction;
      this.name = name;
  }
  public Ship(String name) {
      this.name = name;
  }
  private double degreesToRadians(double degrees) {
      return(degrees * Math.PI / 180.0);
  }
...
```

Lớp cơ sở ship

```
   public void move() {
       move(1);
   }
   public void move(int steps) {
      double angle = degreesToRadians(direction);
      x = x + (double)steps * speed * Math.cos(angle);
      y = y + (double)steps * speed * Math.sin(angle);
   }
   public void printLocation() {
      System.out.println(name + " is at ("+ x + "," + y
      + ").");
   }
 }
 ...
```

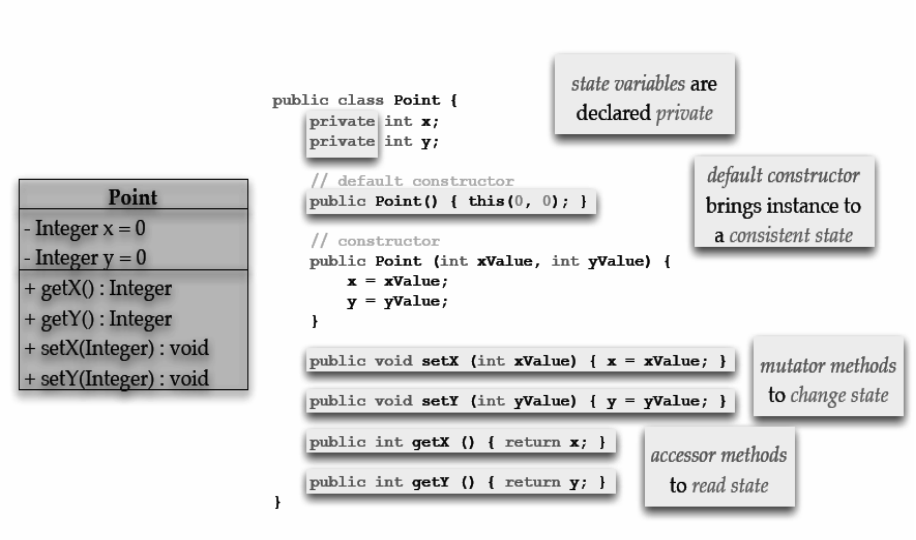

Lớp dẫn xuất Speedboat

```
public class Speedboat extends Ship {
  private String color = "red";
  public Speedboat(String name) {
      super(name);
      setSpeed(20);
  }
  public Speedboat(double x, double y, double speed,
  double direction, String name, String color) {
      super(x, y, speed, direction, name);
      setColor(color);
  }
  public void printLocation() {
      System.out.print(getColor().toUpperCase() + " ");
      super.printLocation();
  }
...
```

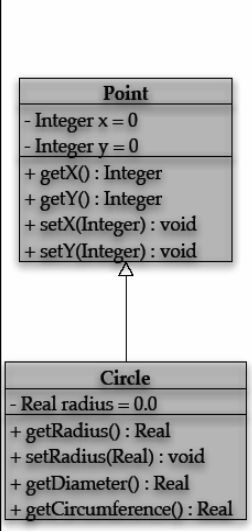
Lớp trừu tượng Point, ColorPoint

```
abstract class Point {
      private int x, y;
      public Point(int x, int y) { this.x = x; this.y = y; }
      public void move(int dx, int dy) {
          x += dx; y += dy;
          plot();
      }
      public abstract void plot(); // abstract method has no implemention
}
abstract class ColoredPoint extends Point {
      int color;
      public ColoredPoint(int x, int y, int color) { super(x, y); this.color = color; }
}
class SimpleColoredPoint extends ColoredPoint {
      public SimpleColoredPoint(int x, int y, int color) { super(x,y,color); }
      public void plot() { ... } // code to plot a SimplePoint
}
```


Ví dụ: Point, Circle, Cylinder



Ví dụ: Point, Circle, Cylinder




```
public class Circle extends Point {
    private double radius;

    public Circle() {}

    public Circle(int xValue, int yValue) {
        super(xValue, yValue);
        setRadius(0.0);
    }

    // constructor
    public Circle (int xValue, int yValue, double radius) {
        super(xValue, yValue);
        setRadius(radius);
    }

    public void setRadius (double radius) {
        this.radius = (radius < 0.0 ? 0.0 : radius);
    }

    public double getRadius () { return radius; }

    public double getDiameter() { return 2 * getRadius(); }

    public double getCircumference() { return Math.PI * getDiameter(); }
}
```

implicit call to `Point()`

explicit call to `Point(xValue, yValue)`

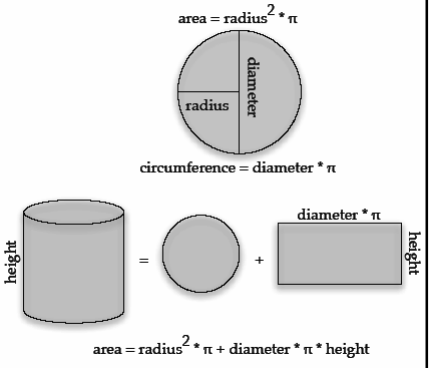
Good practice: call the *mutator* method

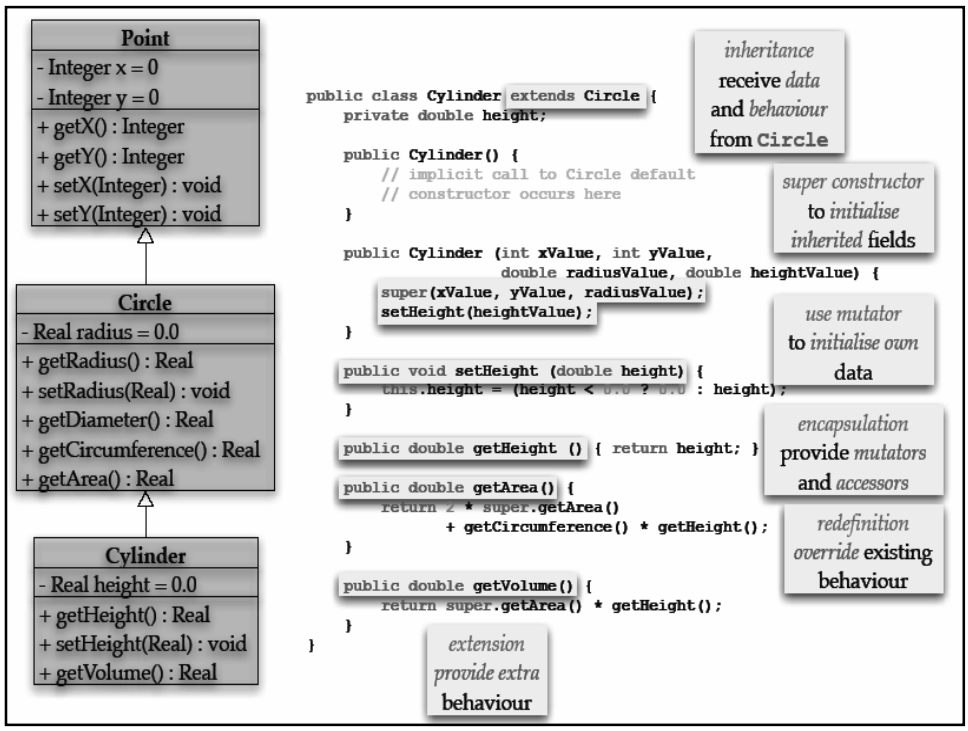
Good practice: call the *accessor* methods

inheritance: *no need* to redefine all the `Point` methods

Định nghĩa lại: Cylinder

- Cylinder phải định nghĩa lại getArea() thừa kế từ Circle
- Dùng getArea() và getCircumference() của lớp cha



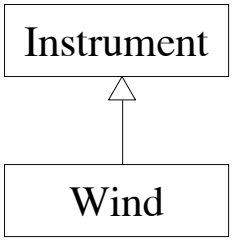


Upcasting – Tương tự C++

```
import java.util.*;

class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
}
```

```
// Wind objects are instruments
// because they have the same
interface:
class Wind extends Instrument {
  public static void main(String[] args)
{
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
}
```



Abstract Class

```
abstract class Point {
    private int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public void move(int dx, int dy) {
    x += dx; y += dy;
    plot();
    }
    public abstract void plot(); // phương thức trừu tượng
      không có code thực hiện
}
```


Abstract Class

```
abstract class ColoredPoint extends Point {
   int color;
   public ColoredPoint(int x, int y, int color) { super(x, y);
   this.color = color; }
}

class SimpleColoredPoint extends ColoredPoint {
   public SimpleColoredPoint(int x, int y, int color) {
        super(x,y,color);
    }
   public void plot() { ... } // code to plot a SimplePoint
}
```

Abstract Class

- Lớp ColoredPoint không cài đặt mã cho phương thức plot() do đó phải khai báo: abstract
- Chỉ có thể tạo ra đối tượng của lớp SimpleColoredPoint.
- Tuy nhiên có thể:

```
Point p = new SimpleColoredPoint(a, b, red);
p.plot();
```

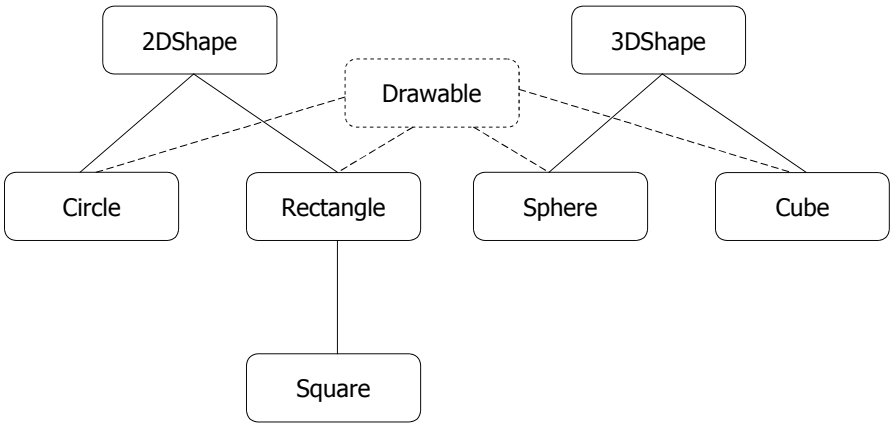

Thừa kế pha trộn mix-in inheritance

- Trong dạng thừa kế này, một "lớp" sẽ cung cấp một số chức năng nhằm hòa trộn vào những lớp khác.
- Một lớp pha trộn thường sử dụng lại một số chức năng định nghĩa từ lớp cung cấp nhưng lại thừa kế từ một lớp khác.
- Một số phần mềm dựa trên một số dịch vụ (service) công dụng chung được cung cấp bởi nhiều lớp. Thừa kế pha trộn là cách thức để tập trung việc phát triển các dịch vụ này.
- Trong Java thừa kế pha trộn được thể hiện qua khái niệm Interface

Giao diện (Interface)

- Interface: đặc tả cho các bản cài đặt (implementation) khác nhau.
- Interface định nghĩa một "kiểu" chỉ chứa định nghĩa hằng và phương thức trừu tượng
- Interface không cài đặt bất cứ một phương thức nào nhưng để lại cấu trúc thiết kế trên bất cứ lớp nào sử dụng nó.
- Một lớp cài đặt interface phải hoặc cài đặt tất cả các phương thức hoặc khai báo là lớp trừu tượng

Interface



Interface

- Kế thừa giao diện: Một interface có thể kế thừa một (thậm chí nhiều) interface khác.

```
public interface Displayable extends Printable,
   Drawable {
   // Định nghĩa hằng và phương thức trừu tượng riêng
   ...
    }
```

- Cài đặt giao diện: Một lớp có thể cài đặt cùng lúc nhiều giao diện khác nhau.

```
public class Circle extends 2DShape implements
   Printable, Drawable {
...
}
```


Interface

- Tất cả các phương thức khai báo trong Interface là ngầm định trừu tượng
- Các thành phần dữ liệu là ngầm định: static final (hằng)
- Các thành phần (dữ liệu và phương thức) là public

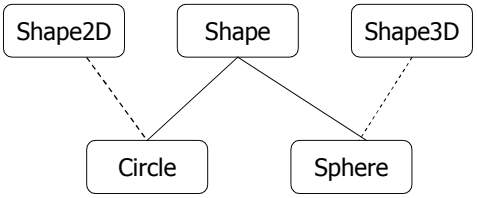
Ví dụ

```
interface Shape2D {
  double getArea();
}

interface Shape3D {
  double getVolume();
}

class Point3D {
  double x, y, z;

  Point3D(double x, double y, double z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }
}
```




```
abstract class Shape {
  abstract void display();
}

class Circle extends Shape
implements Shape2D {
  Point3D center, p; // p is an point on
circle

  Circle(Point3D center, Point3D p) {
    this.center = center;
    this.p = p;
  }

  public void display() {
    System.out.println("Circle");
  }

  public double getArea() {
    double dx = center.x - p.x;
    double dy = center.y - p.y;
    double d = dx * dx + dy * dy;
    double radius = Math.sqrt(d);
    return Math.PI * radius * radius;
  }
}
```

```
class Sphere extends Shape
implements Shape3D {
  Point3D center;
  double radius;

  Sphere(Point3D center, double radius) {
    this.center = center;
    this.radius = radius;
  }

  public void display() {
    System.out.println("Sphere");
  }

  public double getVolume() {
    return 4 * Math.PI * radius * radius * radius / 3;
  }
}

class Shapes {

  public static void main(String args[]) {

    Circle c = new Circle(new Point3D(0, 0, 0), new
      Point3D(1, 0, 0));
    c.display();
    System.out.println(c.getArea());
    Sphere s = new Sphere(new Point3D(0, 0, 0), 1);
    s.display();
    System.out.println(s.getVolume());
  } }
```

```
Result :
Circle
3.141592653589793
Sphere
4.1887902047863905
```

Lớp trừu tượng vs Giao diện

- Lớp trừu tượng bên cạnh các phương thức trừu tượng vẫn có các thành phần thông thường (dữ liệu, phương thức). Chỉ có thể thừa kế từ duy nhất một lớp trừu tượng.
- Giao diện định nghĩa ra một tập các chữ ký (dịch vụ, chức năng) trừu tượng mà các lớp cài đặt phải thực hiện. Nó cho phép định nghĩa lớp với nhiều hành vi khác nhau tương ứng với các tình huống khác nhau

Câu hỏi

- Nêu bản chất của nguyên lý kết tập
- Nêu bản chất của nguyên lý kế thừa
- Các đặc điểm của kế thừa
- Những hàm nào không được phép kế thừa và tại sao
- Phân biệt sự giống nhau và khác nhau giữa kế thừa và kết tập
- Nêu nguyên lý và thứ tự thực hiện của hàm khởi tạo trong kế thừa và kết tập
- Khi nào sử dung kế thừa và kết tập

Bài tập

- Xây dựng chương trình quản lý điểm học sinh:
- Các lớp cần có: các lớp Person, Student, Teacher, Subject, BKClass
- Xây dựng mô hình kiến trúc của bài toán thoả mãn các yêu cầu sau:
- Nhập được các thông số của học sinh, giáo viên dạy các môn học và điểm cho các môn
- Nhập danh sách một lớp BKclass theo quy đinh có 40 sinh viên
- Thực hiện chương trình và vẽ sơ đồ mô tả kiến trúc của lớp: quan hệ kết tập, quan hệ kế thừa, v.v