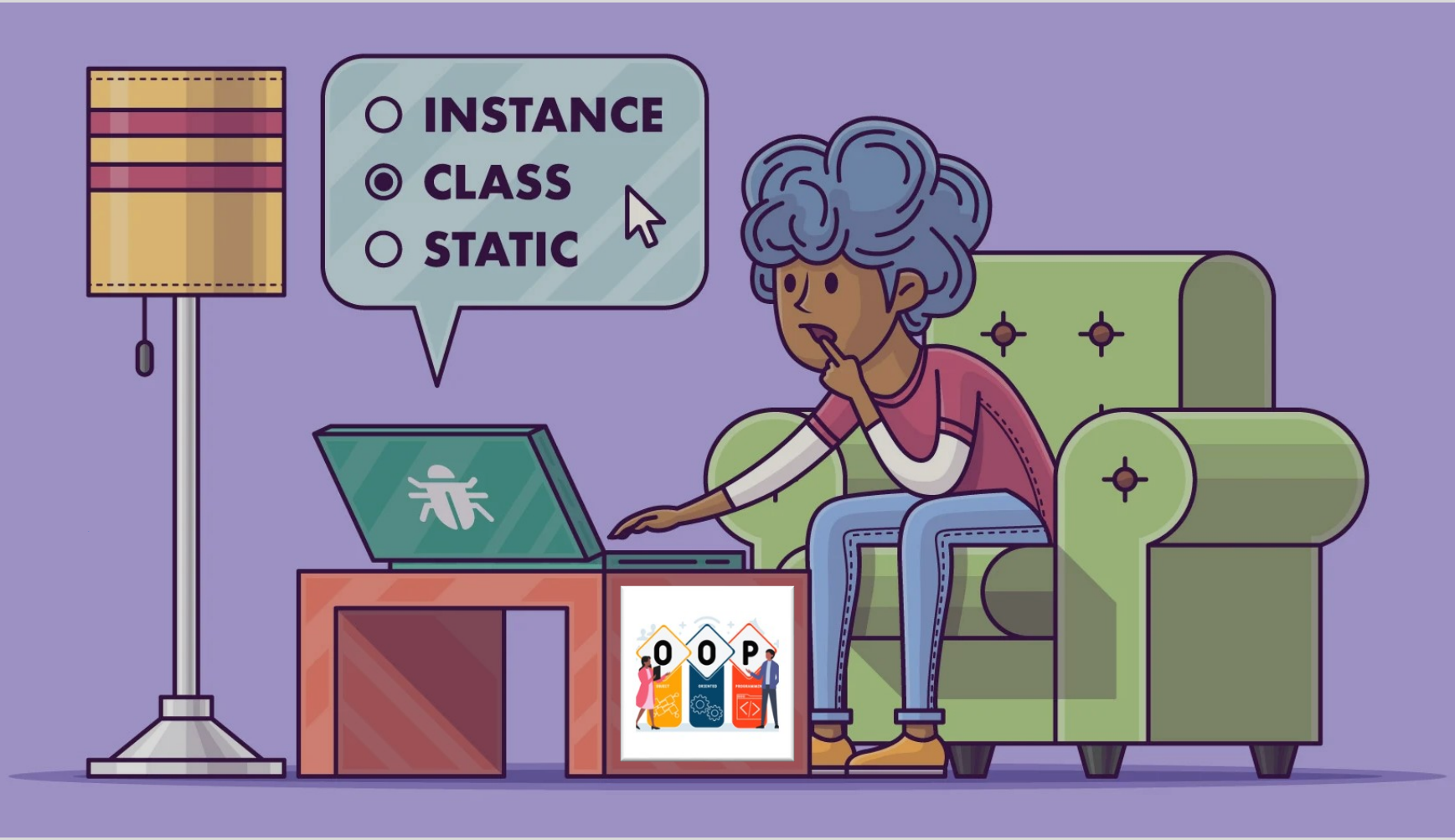


IT002 - Lập trình hướng đối tượng

Nội dung

1. Quan hệ giữa các lớp đối tượng
2. Lớp cơ sở và lớp dẫn xuất
3. Tính thừa kế
4. Hàm tạo, hàm hủy và hàm toán tử gán trong thừa kế
5. Ứng dụng của tính thừa kế

1. Quan hệ giữa các lớp đối tượng

Giữa các lớp đối tượng có những loại quan hệ sau:

- Quan hệ một một **(1-1)**
- Quan hệ một nhiều **(1-n)**
- Quan hệ nhiều nhiều **(n-n)**
- Quan hệ đặc biệt hóa – tổng quát hóa

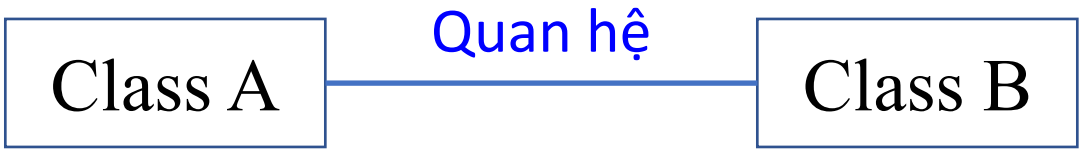
1.1 Quan hệ 1-1

❖ **Khái niệm:**

Hai lớp đối tượng được gọi là có quan hệ **1-1** với nhau khi **1** đối tượng thuộc lớp này có quan hệ với **1** đối tượng thuộc lớp kia và **1** đối tượng thuộc lớp kia có quan hệ duy nhất với **1** đối tượng thuộc lớp này.

1.1 Quan hệ 1-1 (tt)

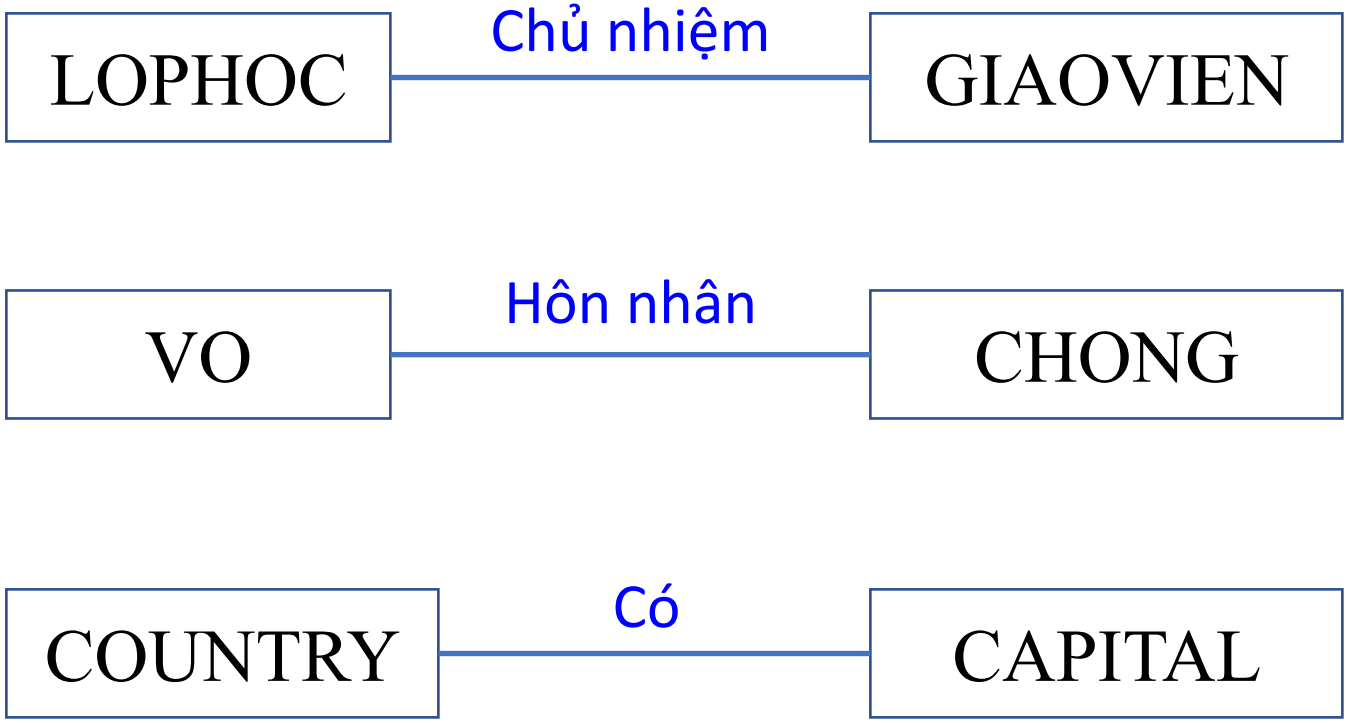
❖ **Ký hiệu:**



Trong hình vẽ trên ta nói: **1** đối tượng thuộc lớp **A** có quan hệ với **1** đối tượng thuộc lớp **B** và **1** đối tượng thuộc lớp **B** có quan hệ duy nhất với **1** đối tượng thuộc lớp **A**.

1.1 Quan hệ 1-1 (tt)

❖ **Ví dụ:**



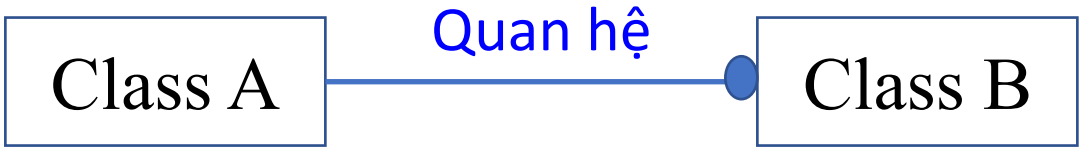
1.2 Quan hệ 1-n

❖ **Khái niệm:**

Hai lớp đối tượng được gọi là có quan hệ **1-n** với nhau khi **1** đối tượng thuộc lớp này có quan hệ với **n** đối tượng thuộc lớp kia và **1** đối tượng thuộc lớp kia có quan hệ duy nhất với **1** đối tượng thuộc lớp này.

1.2 Quan hệ 1-n

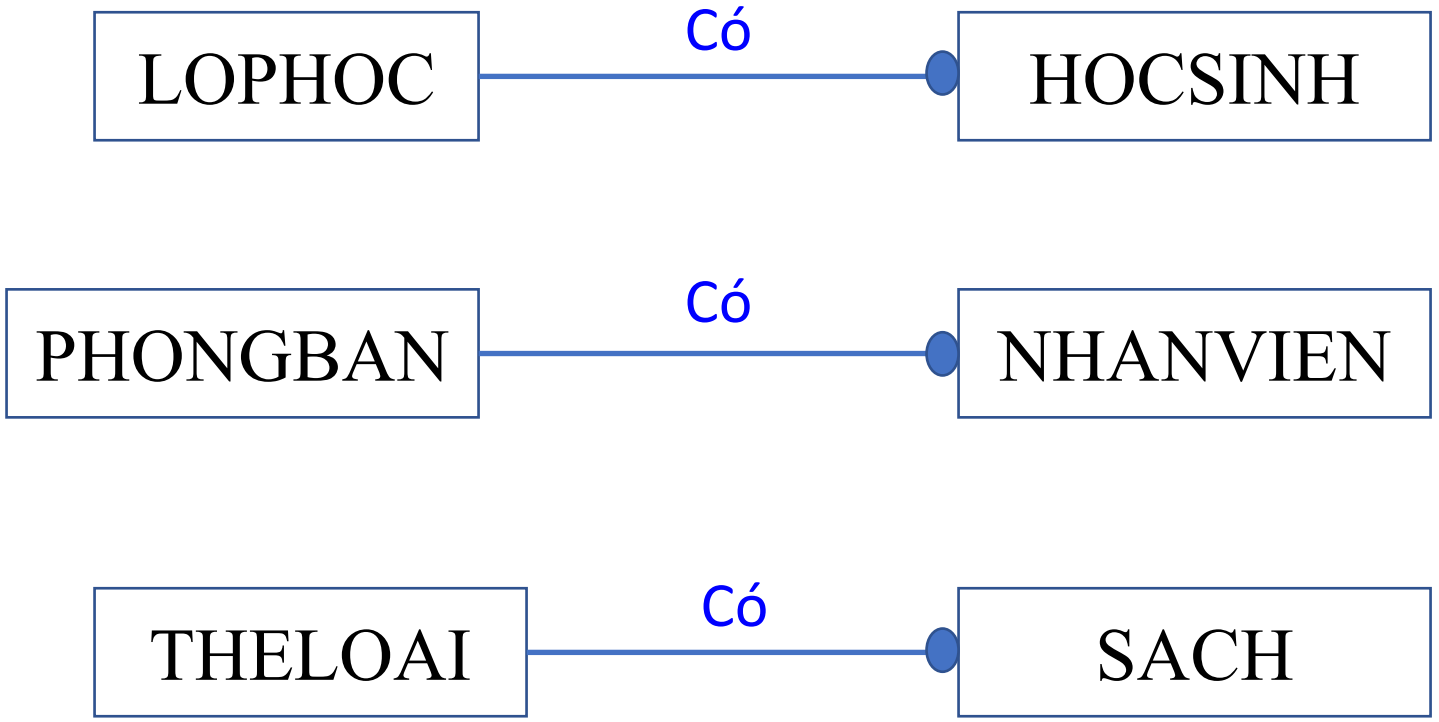
❖ **Kí hiệu:**



Trong hình vẽ trên ta nói: **1** đối tượng thuộc lớp **A** có quan hệ với **n** đối tượng thuộc lớp **B** và **1** đối tượng thuộc lớp **B** có quan hệ <u>duy nhất</u> với **1** đối tượng thuộc lớp **A**.

1.2 Quan hệ 1-n (tt)

❖ **Ví dụ:**



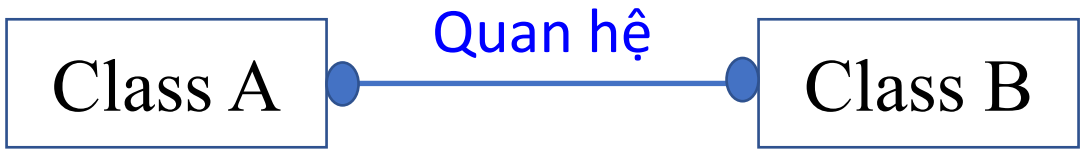
1.3 Quan hệ n-n

❖ **Khái niệm:**

Hai lớp đối tượng được gọi là có quan hệ **n-n** với nhau khi **1** đối tượng thuộc lớp này có quan hệ với **n** đối tượng thuộc lớp kia và **1** đối tượng thuộc lớp kia cũng có quan hệ với **n** đối tượng thuộc lớp này.

1.3 Quan hệ n-n (tt)

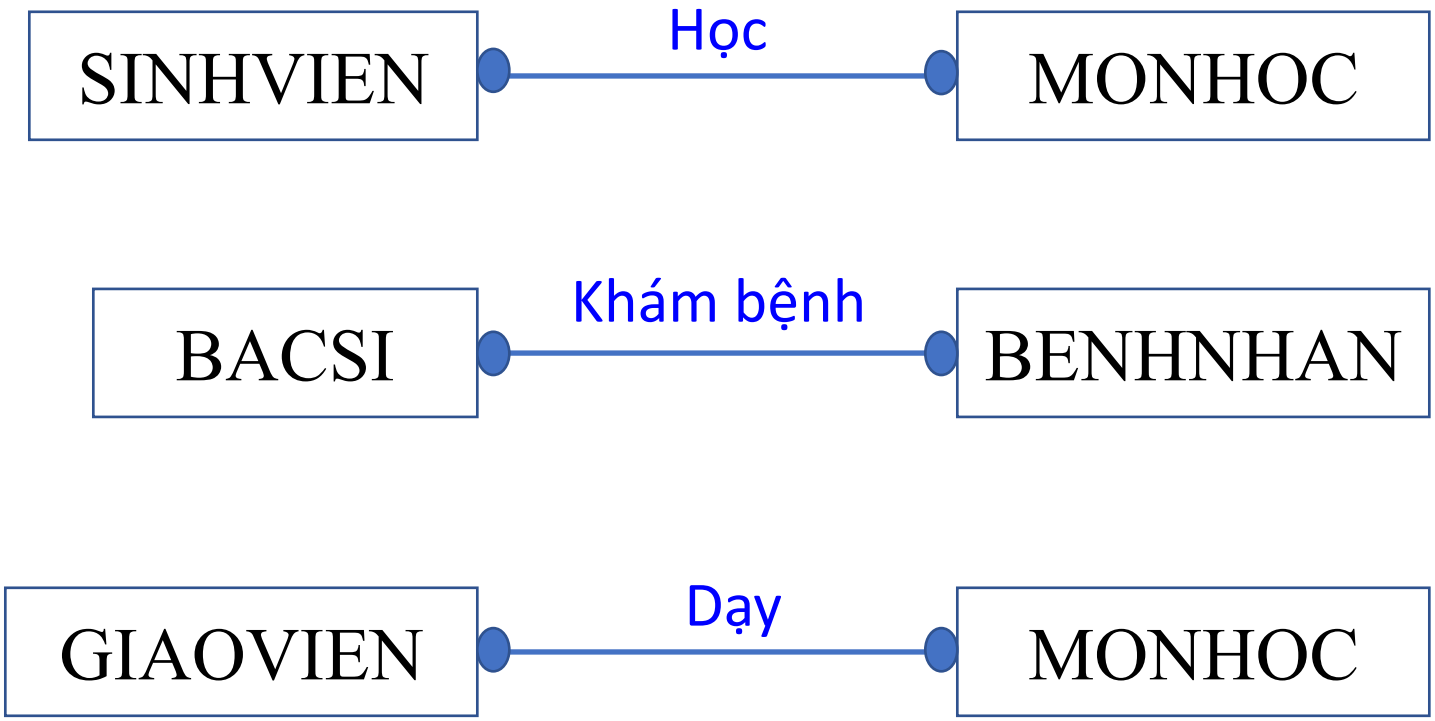
❖ **Kí hiệu:**



Trong hình vẽ trên ta nói: **1** đối tượng thuộc lớp **A** có quan hệ với **n** đối tượng thuộc lớp **B** và **1** đối tượng thuộc lớp **B** cũng có quan hệ với **n** đối tượng thuộc lớp **A**.

1.3 Quan hệ n-n (tt)

❖ **Ví dụ:**



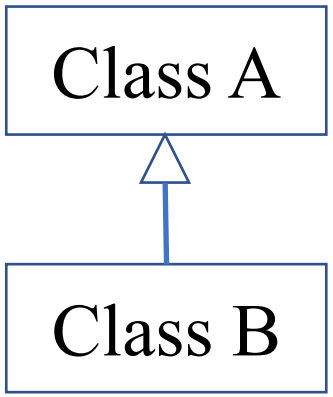
1.4 Quan hệ đặc biệt hóa – tổng quát hóa

❖ **Khái niệm:**

Hai lớp đối tượng được gọi là có quan hệ **đặc biệt hóa-tổng quát hóa** với nhau khi lớp đối tượng này là trường hợp đặc biệt của lớp đối tượng kia và lớp đối tượng kia là trường hợp tổng quát của lớp đối tượng này.

1.4 Quan hệ đặc biệt hóa – tổng quát hóa (tt)

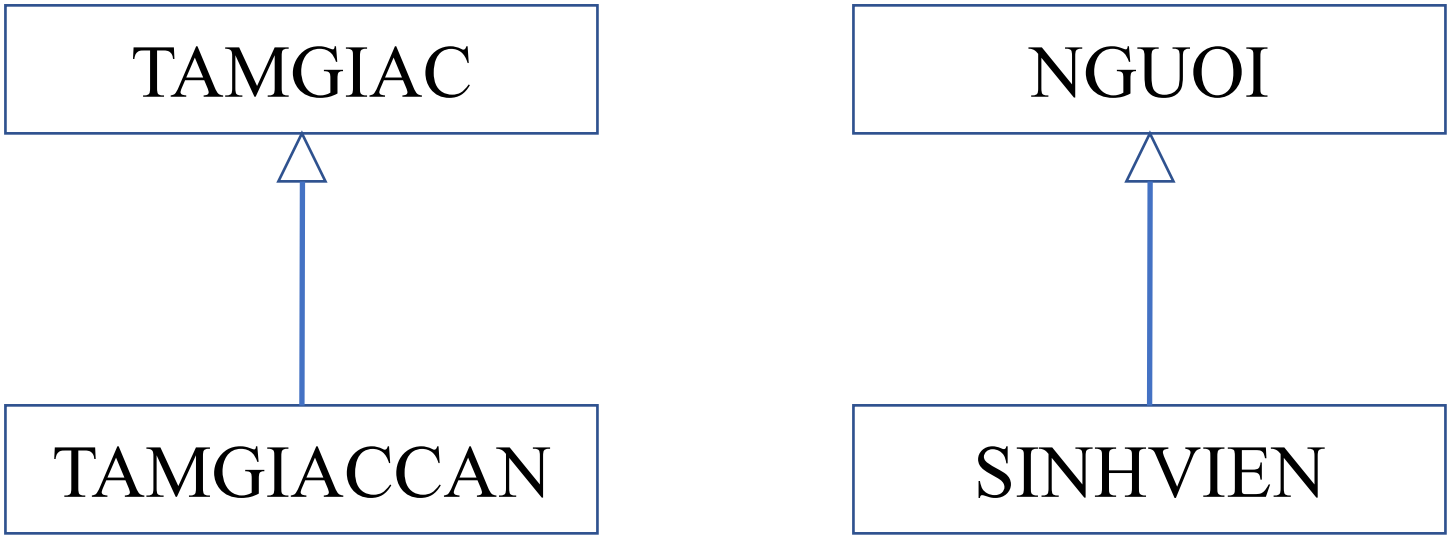
❖ **Kí kiệu:**



Trong hình vẽ trên ta nói: lớp đối tượng **B** là trường hợp đặc biệt của lớp đối tượng **A** và lớp đối tượng **A** là trường hợp tổng quát của lớp đối tượng **B**.

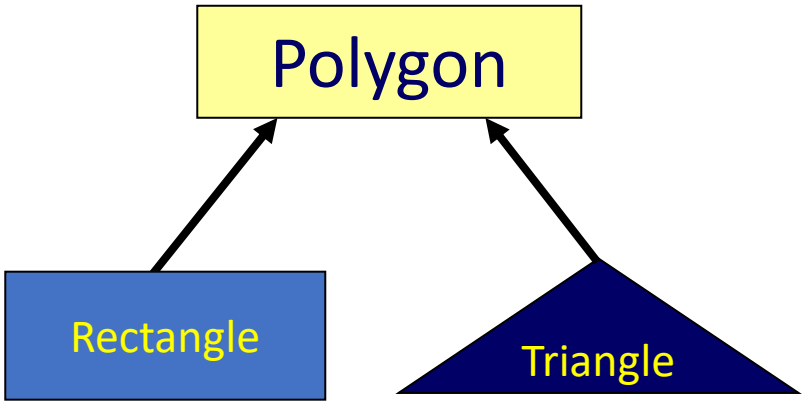
1.4 Quan hệ đặc biệt hóa – tổng quát hóa (tt)

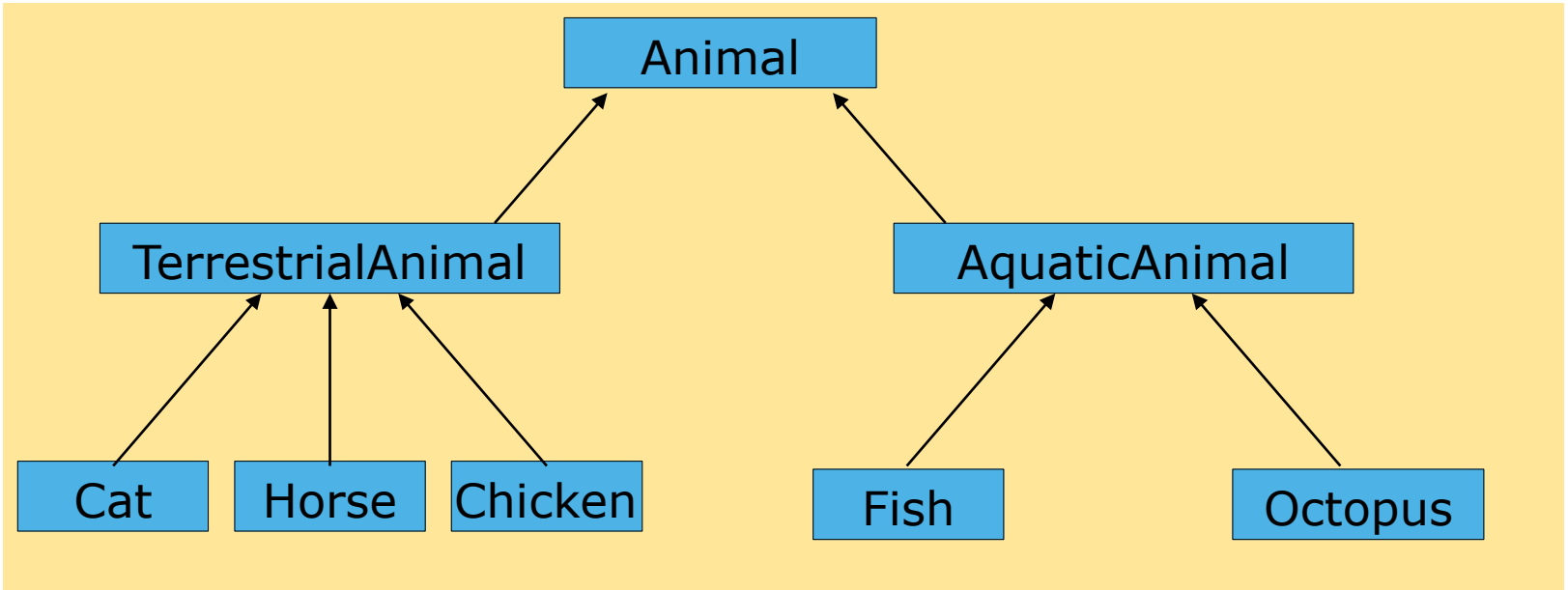
❖ **Ví dụ:**



1.4 Quan hệ đặc biệt hóa – tổng quát hóa (tt)

❖ **Ví dụ (tt):**





2. Lớp cơ sở và lớp dẫn xuất

- Một lớp thừa kế một lớp khác gọi là **lớp dẫn xuất (derived class** hay **sub class).**
- Lớp dùng để xây dựng lớp dẫn xuất gọi là **lớp cơ sở (super class** hay **base class).**
- Một lớp có thể vừa là lớp cơ sở vừa là lớp dẫn xuất.
- Một lớp có thể ***dẫn xuất từ nhiều lớp cơ sở*** và cũng có thể ***là lớp cơ sở cho nhiều lớp dẫn xuất*** khác nhau.

2. Lớp cơ sở và lớp dẫn xuất (tt)

Ví dụ: Xây dựng lớp **C** dẫn xuất từ lớp **A** và lớp **B**:

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

3. Tính thừa kế

- Khái niệm
- Khai báo thừa kế
- Các loại thừa kế
- Thừa kế thuộc tính
- Thừa kế phương thức
- Lớp cơ sở là thành phần của lớp dẫn xuất
- Phạm vi truy xuất

3.1 Khái niệm thừa kế

- Tính thừa kế cho phép các lớp được xây dựng trên các lớp đã có.
- Tính thừa kế được dùng để biểu diễn mối quan hệ **đặc biệt hóa – tổng quát hóa** giữa các lớp. Trong đó, lớp dẫn xuất thừa kế lớp cơ sở.
- Lớp dẫn xuất thừa kế tất cả các thành phần (thuộc tính và phương thức) của lớp cơ sở, kể cả các thành phần mà lớp cơ sở được thừa kế.

3.2 Khai báo thừa kế

```
class SuperClass{
   //Thành phần của lớp cơ sở
};
class DerivedClass : public/[private] SusperClass{
   //Thành phần bổ sung của lớp dẫn xuất
};
```

3.2 Khai báo thừa kế (tt)

Ví dụ: Xây dựng Lớp **C** thừa kế public lớp **A** và lớp **B**

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

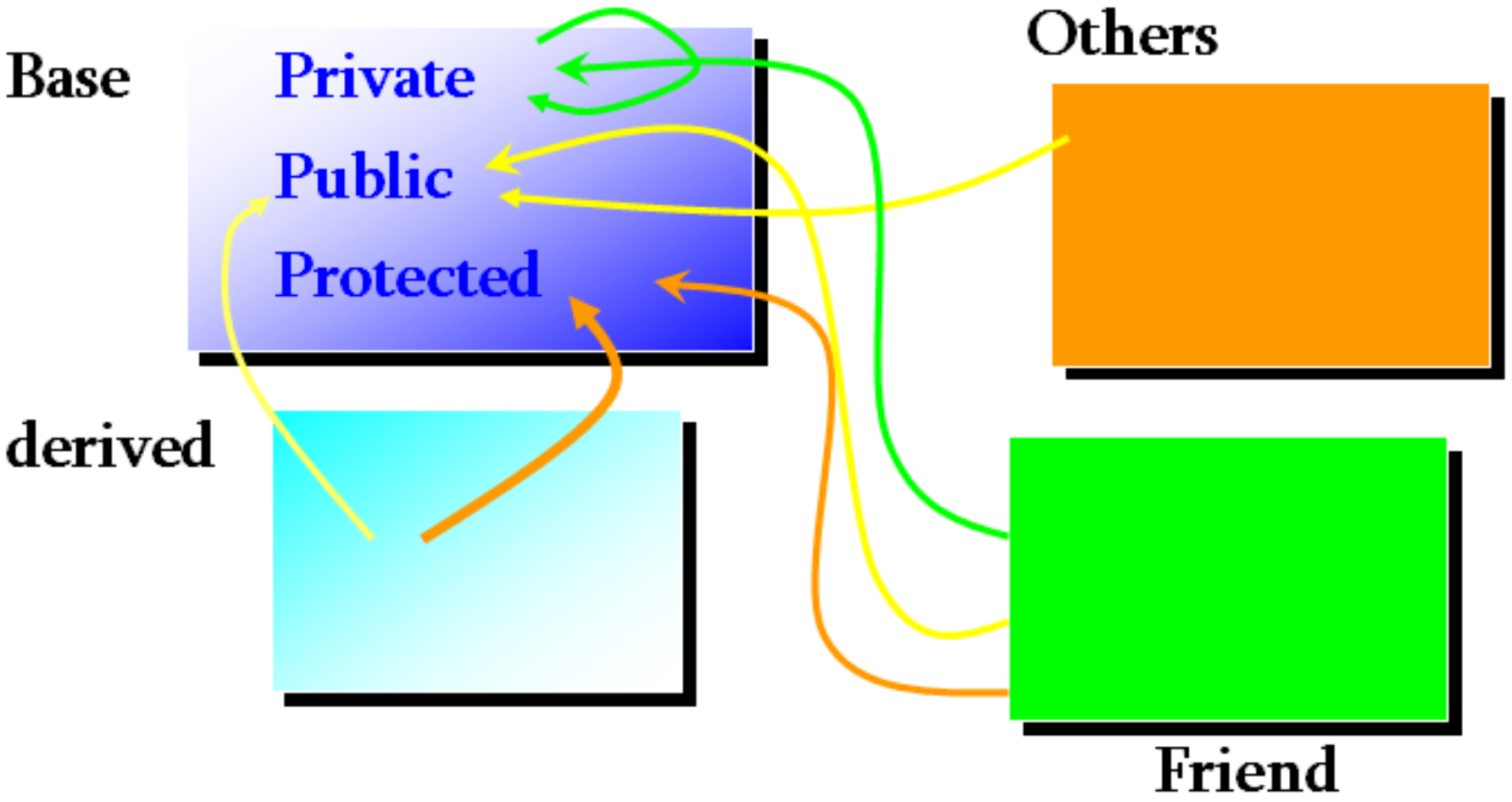
Lưu ý: nếu không dùng từ khóa nào thì hiểu là private

Ví dụ: class **C**: public **A**, **B**{…};

-> Lớp **C** thừa kế public lớp **A**

-> Lớp **C** thừa kế private lớp **B**

3.2 Khai báo thừa kế (tt)



3.2 Khai báo thừa kế (tt)

Việc truy nhập các thành phần của lớp cơ sở được lớp dẫn xuất thừa kế phụ thuộc vào 2 yếu tố:

1) Các thành phần đó được khai báo là private hay public hay protected trong lớp cơ sở.
   - private: không cho phép truy nhập trong lớp dẫn xuất.
   - public: có thể truy nhập tại bất kỳ chỗ nào trong chương trình nên các lớp dẫn xuất có thể truy nhập được.
   - protected: chỉ được truy nhập bởi các lớp dẫn xuất trực tiếp.

   Các thành phần private < các thành phần khai báo là protected có phạm vi truy nhập < các thành phần public

3.2 Khai báo thừa kế (tt)

2) Kiểu dẫn xuất là private (mặc định) hay public hay protected được chỉ định khi định nghĩa lớp dẫn xuất.
   - public: các thành phần **public và protected** của lớp cơ sở sẽ trở thành các thành phần **public và protected** của lớp dẫn xuất.
   - private: các thành phần **public và protected** của lớp cơ sở sẽ trở thành các thành phần **private** của lớp dẫn xuất.

3.3 Thừa kế và sự trùng tên

- Tên các lớp không được giống nhau.
- Tên các thành phần trong 1 lớp cũng không được giống nhau.
- Tên các thành phần trong các lớp khác nhau thì được phép đặt giống nhau.

  => Sử dụng **tên lớp và toán tử phạm vi** để chỉ rõ thành phần đó thuộc lớp nào.

  VD: class **B**: public **A**{…}; **B** b;

  b.**B::**n -> truy xuất tới thuộc tính n của lớp **B**

  b.**A::**nhap() -> gọi phương thức nhap() của lớp **A**

3.3 Thừa kế và sự trùng tên – Ví dụ

```
class BASE_A{
    public:
     int a;
     int f( ){
          return 0;
     }
     int g( ){
          return 0;
     }
     int h( ) { return 0;}
};
```

```
class BASE_B{
    public:
     int a;
     int f( ){
          return 0;
     }
     int g( ){
          return 0;
     }
};
```

3.3 Thừa kế và sự trùng tên – Ví dụ (tt)

```
class ClassC : public BASE_A, public BASE_B{
    //…
};
void main(){
    ClassC C;
    C.a = 1;    //Lỗi: “ClassC::a is ambiguous”
    C.g();//Lỗi: “ClassC::g is ambiguous”
    C.h();      //Ok
}
```

=> Dùng tên lớp và toán tử phạm vi để chỉ rõ thành phần đó thuộc lớp nào:

```
C.BASE_A::a = 1;
C.BASE_B::g();
```

3.4 Các loại thừa kế

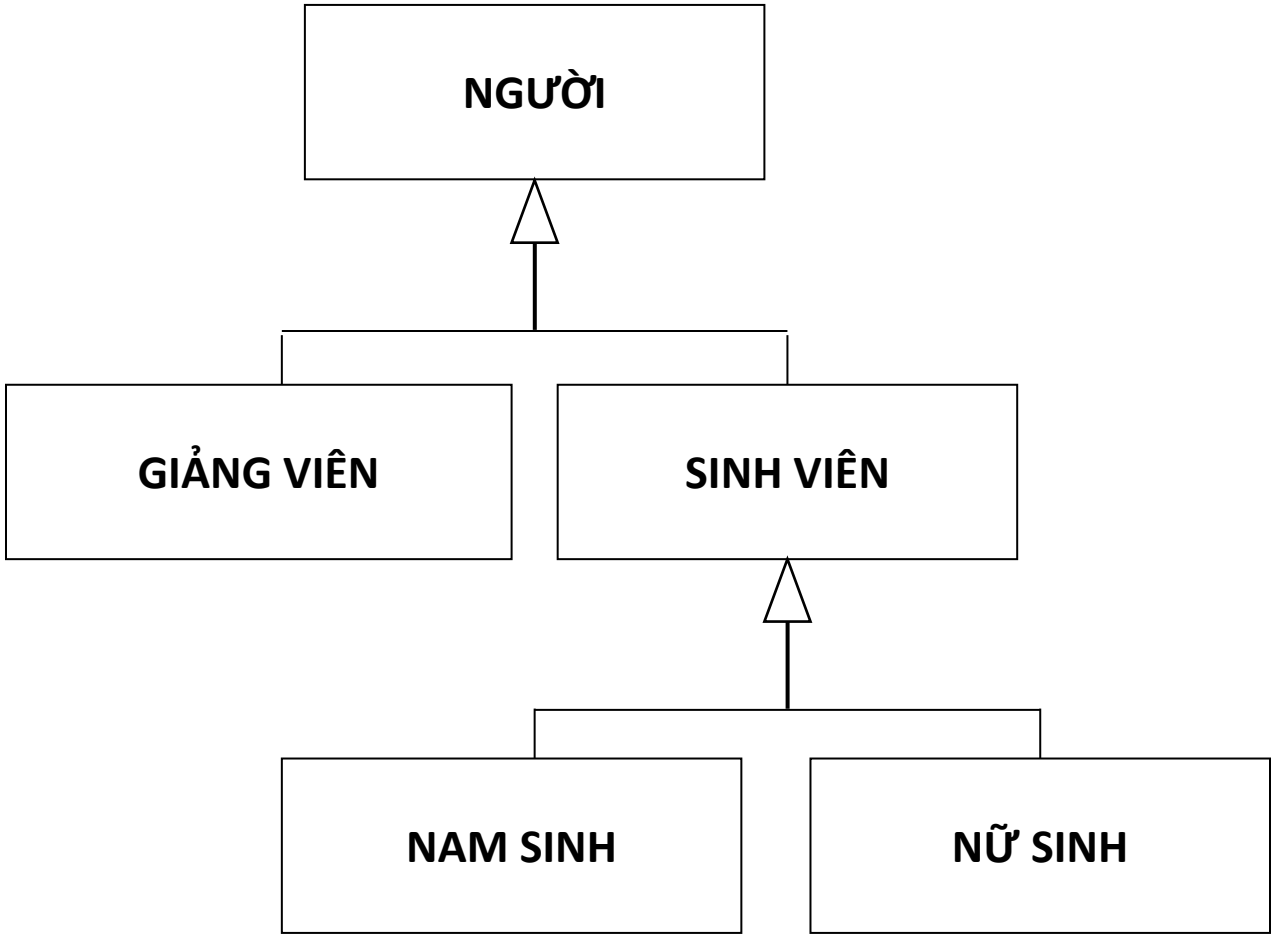
- **Đơn thừa kế:** lớp được dẫn xuất từ **1** lớp cơ sở.

  VD: class A : public B {…};

- **Đa thừa kế:** một lớp có thể được dẫn xuất từ nhiều lớp cơ sở.

  VD: class A : public B, public C {…};

3.4.1 Đơn thừa kế



3.4.1 Đơn thừa kế (tt)

- Một sinh viên là một người, có thêm một số thông tin và một số thao tác của riêng sinh viên.

  => Lớp SINH VIÊN thừa kế lớp NGƯỜI.

- Một nam sinh/nữ sinh là một sinh viên, có thêm một số thông tin và thao tác riêng của nó.

  => Lớp NAM SINH và lớp NỮ SINH thừa kế lớp SINH VIÊN, khi đó lớp SINH VIÊN trở thành lớp cơ sở của hai lớp NAM SINH và NỮ SINH.

3.4.1 Đơn thừa kế – Ví dụ

```
class Nguoi {
    char *HoTen;
    int NamSinh;
public:
    Nguoi();
    Nguoi( char *ht, int ns):NamSinh(ns) {HoTen=_strdup(ht);}
    ~Nguoi() {delete [ ] HoTen;}
    void An() const { cout<<HoTen<<" an 3 chen com \n";}
    void Ngu() const { cout<<HoTen<< " ngu ngay 8 tieng \n";}
    void Xuat() const;
    friend ostream& operator << (ostream& os, const Nguoi &p);
};
```

3.4.1 Đơn thừa kế – Ví dụ (tt)

```
class SinhVien : public Nguoi {
    char *MaSo;
public:
    SinhVien();
    SinhVien( char *ht, char *ms, int ns) : Nguoi(ht,ns) {
      MaSo = _strdup(ms);
    }
    ~SinhVien() {
      delete [ ] MaSo;
    }
    void Xuat() const;
};
```

3.4.1 Đơn thừa kế – Ví dụ (tt)

```
void Nguoi::Xuat() const
{
    cout << "Nguoi, ho ten: " << HoTen;
    cout   << " - Sinh nam: " << NamSinh << endl;
}
ostream& operator<<(ostream &os, const NGUOI& p) {
    os << "Ho ten:" << p.HoTen << "-Sinh nam:" << p.NamSinh << endl;
    return os;
}
void SinhVien::Xuat() const {
    cout << "Sinh vien, ma so: " << MaSo;
    /*Không cho phép truy xuất thành phần private của lớp cơ sở*/
    //cout << ", ho ten: " << HoTen;
    //cout << ", nam sinh: " << NamSinh;
    cout << endl;
}
```

3.4.1 Đơn thừa kế – Ví dụ (tt)

```
void main() {
    Nguoi p1("Le Van Nhan",1980);
    SinhVien s1("Vo Vien Sinh", "200002541",1984);
    p1.An(); //Gọi hàm Nguoi::An()
    s1.An(); //Gọi hàm Nguoi::An()
    p1.Xuat(); //Gọi hàm Nguoi::Xuat()
    s1.Xuat(); //Gọi hàm SinhVien::Xuat()
    s1.Nguoi::Xuat(); //Gọi hàm Nguoi::Xuat()
    cout << p1 << "\n"; //Gọi hàm toán tử <<
    cout << s1 << "\n"; //Gọi hàm toán tử <<
}
```

3.4.2 Đa thừa kế

- Các đặc điểm của đơn thừa kế vẫn đúng cho trường hợp đa thừa kế.
- Tuy nhiên, trong đa thừa kế có thể có sự nhập nhằng => Khai báo **lớp cơ sở ảo.**

3.4.2 Đa thừa kế – Lớp cơ sở ảo

```
class A {
        public:
                int a;
};
class B : public A {
        public:
                int b;
};
class C : public A {
        public:
                int c;
};
class D : public B, public C {
        public:
                int d;
};
```

```
int main() {
        D h;
        h.a = 1; ???
        h.b = 2;
        h.c = 3;
        h.d = 4;
        system("pause");
        return 0;
}
```

-> Báo lỗi: “D::a” is ambiguous

Trình biên dịch không thể nhận biết thuộc tính a được thừa kế thông qua lớp B hay lớp C (vì lớp A là cơ sở cho cả hai lớp cơ sở trực tiếp của lớp D là lớp B và lớp C).

3.4.2 Đa thừa kế – Lớp cơ sở ảo (tt)

```
class A {
        public:
        int a;
};
class B : virtual public A {
        public:
        int b;
};
class C : virtual public A {
        public:
        int c;
};
class D : public B, public C {
        public:
        int d;
};
```

```
int main() {
        D h;
        h.a = 1; //Ok
        h.b = 2;
        h.c = 3;
        h.d = 4;
        system("pause");
        return 0;
}
```

=> Khai báo A là lớp cơ sở kiểu virtual cho cả B và C. Khi đó, hai lớp cơ sở A (A là cơ sở của B và A là cơ sở của C) sẽ kết hợp lại để trở thành 1 lớp cơ sở A duy nhất cho bất kỳ lớp dẫn xuất nào từ B và C.

3.5 Thừa kế thuộc tính

- Gọi **A** = các thuộc tính của lớp cơ sở
- **A** được thừa kế trong lớp dẫn xuất:

  Tập thuộc tính của lớp dẫn xuất

  = các thuộc tính mới của lớp dẫn xuất + **A**
- Tuy nhiên trong các phương thức của lớp dẫn xuất **không cho phép truy nhập** đến các thuộc tính private của lớp cơ sở.

3.5 Thừa kế thuộc tính – Ví dụ

```
class HinhTron : public Diem {
    double r; //Bán kính, là thuộc tính mới của lớp dẫn xuất
public:
    HinhTron( double tx, double ty, double rr) : Diem(tx, ty){
     r = rr;
    }
    void Ve(int color) const;
    void TinhTien( double dx, double dy) const;
};
HinhTron t(200,200,50); //Gọi hàm tạo của lớp HinhTron
```

3.6 Thừa kế phương thức

- Lớp dẫn xuất **không thừa kế** các HÀM TẠO, HÀM HỦY, HÀM TOÁN TỬ GÁN của các lớp cơ sở.
- Các phương thức public khác của lớp cơ sở được thừa kế trong lớp dẫn xuất.

3.6 Thừa kế phương thức (tt)

❖ Tuy nhiên, các phương thức của lớp cơ sở có thể được cài đặt lại trong lớp dẫn xuất trong một số trường hợp sau:

- **TH1:** Thao tác ở lớp dẫn xuất **khác** thao tác ở lớp cơ sở.

  Thông thường là các thao tác nhập, xuất.

- **TH2:** Giải thuật ở lớp dẫn xuất **đơn giản hơn** (ví dụ như tô màu đa giác, tính diện tích...).

- **TH3:** Thao tác **không có ý nghĩa** trong lớp dẫn xuất (ví dụ hàm quay 1 hình tròn (lớp hình tròn thừa kế lớp elip)).

- **TH4:** Các phương thức có thể **vi phạm ràng buộc dữ liệu** của lớp dẫn xuất (ví dụ cài đặt lại toán tử gán để đảm bảo ràng buộc là mọi đối tượng thuộc lớp Số ảo phải có phần thực bằng 0).

3.6 Thừa kế phương thức – TH1

```
class SinhVien : public Nguoi {
    char *MaSo;
public:
    void Xuat() const;
};
void Nguoi::Xuat() const
{
    cout << "Ho ten: " << HoTen;
    cout << "-Sinh nam: " << NamSinh << endl;
}
void SinhVien::Xuat() const {
    NGUOI::Xuat();
    cout << "Sinh vien, ma so: " << MaSo << endl;
}
```

3.6 Thừa kế phương thức – TH1

```
class Point{
    protected:
      int x, y;
    public:
      void set(int a, int b)
     { x=a; y=b; }
      void foo ();
      void print();
};
```

```
class Circle : public Point{
  private:  double r;
  public:
     void set(int a, int b, double c) {
         Point::set(a, b);
         r = c;
     }
     void print() {..}
};
```

```
Point A;
A.set(30,50); //class Point
A.print(); //class Point
```

```
Circle C;
C.set(10,10,100); //class Circle
C.foo(); //class Point
C.print(); //class Circle
```

3.6 Thừa kế phương thức – TH3

```
class Ellipse {
    public:
        void rotate(double rotangle){
            //...
        }
};
class Circle : public Ellipse {
    public:
        void rotate(double rotangle){
            /* do nothing */
        }
};
```

3.6 Thừa kế phương thức – TH4

```
class Complex { //Số phức
    friend ostream& operator <<(ostream&, Complex&);
    friend class Imag; //Khai báo lớp Số ảo là lớp bạn của lớp Số phức
    double re, im;
public:
    Complex( double r = 0, double i = 0):re(r), im(i){ }
    Complex& operator = (const Complex &c) {
        re = c.re; im = c.im;
        return *this;
    }
    Complex operator +(Complex b);
    Complex operator -(Complex b);
    Complex operator *(Complex b);
    Complex operator /(Complex b);
    double Norm() const { return sqrt(re*re + im*im);}
};
```

3.6 Thừa kế phương thức – TH4

```
class Imag: public Complex { //Số ảo là số phức với phần thực = 0
public:
    Imag(double i = 0) : Complex(0, i){ }
    Imag(const Complex &c) : Complex(0, c.im){ }
    Imag& operator = (const Complex &c){
     re = 0; im = c.im; //do là lớp bạn với lớp Complex
     return *this;
    }
    double Norm() const {
     return fabs(im);
    }
};
```

3.6 Thừa kế phương thức – TH4

```
ostream& operator<<(ostream& os, Complex &p){
    os << "phan thuc la: " << p.re << " va phan ao la: " << p.im << endl;
    return os;
}
void main()
{
    Imag i = 1; // i la so ao (0,1), do gọi hàm tạo Imag(double i = 0)
    Complex z1(1,1); // z1 la so phuc (1,1)
    Complex z2 = z1 - i; // z2 la so phuc (1,0), do gọi htt gán trong lớp Complex
    i = Complex(5,2); // i la so ao (0,2), do gọi hàm toán tử gán trong lớp Imag
    Imag j = z1; // j la so ao (0,1), do gọi hàm toán tử gán trong lớp Imag
    cout << "i co " << i << "\n";
    cout << "j co " << j << "\n";
}
```

3.7 Lớp cơ sở là thành phần của lớp dẫn xuất

- Có thể thay thừa kế bằng cách khai báo lớp cơ sở là thành phần của lớp dẫn xuất.
- Khi đó lớp dẫn xuất là **lớp bao** => xây dựng **hàm tạo của lớp bao** sẽ sử dụng các hàm tạo của các lớp thành phần tương ứng để khởi gán cho các thuộc tính là đối tượng của lớp bao.

3.7 Lớp cơ sở là thành phần của lớp dẫn xuất – Ví dụ

```
class DIEM {
        double x, y;
public:
        DIEM() { x = y = 0.0; }
        DIEM(double xx, double yy) {
                x = xx;
                y = yy;
        }
};
class HINHTRON : public DIEM {
        double r;
public:
        HINHTRON() { r = 0.0; }
        HINHTRON(double xx, double yy,
        double rr) : DIEM(xx,yy) {r = rr;}
};
```

```
class DIEM {
        double x, y;
public:
        DIEM() { x = y = 0.0; }
        DIEM(double xx, double yy) {
                x = xx;
                y = yy;
        }
};
class HINHTRON{
        DIEM d;
        double r;
public:
        HINHTRON(): d() { r = 0.0; }
        HINHTRON(double xx, double yy,
        double rr) : d(xx,yy) {r = rr;}
};
```

3.8 Phạm vi truy xuất

- Mặc dù lớp dẫn xuất được thừa kế tất cả các thành phần của lớp cơ sở (thuộc tính và phương thức), nhưng trong lớp dẫn xuất không thể truy nhập đến tất cả các thành phần này.
- Thường thì các thuộc tính của lớp cơ sở sẽ được truy nhập thông qua các phương thức **public** mà lớp cơ sở cung cấp.

3.8 Phạm vi truy xuất (tt)

- Việc truy nhập đến các thành phần của lớp cơ sở phụ thuộc vào 2 yếu tố:
  1) Các thành phần đó được khai báo là private (mặc định) hay public hay protected trong lớp cơ sở.
     => Truy xuất theo chiều dọc.
  2) Kiểu dẫn xuất là private (mặc định) hay public hay protected được chỉ định khi định nghĩa lớp dẫn xuất.
     => Truy xuất theo chiều ngang.

3.8.1 Truy xuất theo chiều dọc

- Thành phần private
- Thành phần public
- Thành phần protected

3.8.1 Truy xuất theo chiều dọc – Ví dụ

```
class A{
    private:
        int a;
        void f();
    protected:
        int b;
        void g();
    public:
        int c;
        void h();
};
```

```
void A::f()
{
    a = 1;    b = 2;        c = 3;
}
void A::g()
{
    a = 4;    b = 5;        c = 6;
}
void A::h(){
    a = 7;    b = 8;        c = 9;
}
```

3.8.1 Truy xuất theo chiều dọc – Ví dụ (tt)

```
void main(){
    A x;
    x.a = 10;   //báo lỗi inaccessible vì a là thành phần private
    x.f();      //báo lỗi inaccessible vì f là thành phần private
    x.b = 20;   //báo lỗi inaccessible vì b là thành phần protected
    x.g();      //báo lỗi inaccessible vì g là thành phần protected
    x.c = 30;
    x.h();
}
```

3.8.1.1 Thành phần private

- Chỉ được phép truy nhập bởi hàm thành phần của lớp/hàm bạn của lớp.
- Lớp dẫn xuất không được phép truy nhập đến các thành phần này.

3.8.1.1 Thành phần **private** – Ví dụ

```
class Nguoi {
    char *HoTen;
    int NamSinh;
public:
    //...
};
class SinhVien : public Nguoi {
    char *MaSo;
public:
    //...
    void Xuat() const;
};
```

```
void SinhVien::Xuat() const {
      cout << "MSSV: "<< MaSo
           << " Ho ten: " << HoTen;}
```

⇒ Báo lỗi vì HoTen là t.phần private

⇒ Khai báo lớp SV là bạn lớp Nguoi:

```
class Nguoi {
    friend class SinhVien;
    char *HoTen;
    int NamSinh;
public:
    //...
};
```

3.8.1.1 Thành phần **private** – Ví dụ (tt)

```
class Nguoi {
    friend class SinhVien; //Khai báo lớp SinhVien là bạn của lớp Nguoi
    friend class NuSinh; //Khai báo lớp NuSinh là bạn của lớp Nguoi
    char *HoTen;
    int NamSinh;
public:
    //...
    void An() const { cout << HoTen << " an 3 chen com";}
};
class SinhVien : public Nguoi {
    friend class NuSinh; //Khai báo lớp NuSinh là bạn của lớp SinhVien
    char *MaSo;
public:
    //...
}; //Khi thêm lớp NuSinh -> phải thay đổi lớp SinhVien và cả lớp Nguoi
```

3.8.1.1 Thành phần **private** – Ví dụ (tt)

- Với cách khai báo như trên thì lớp SinhVien và lớp NuSinh có thể truy nhập các thành phần private của lớp Nguoi; lớp NuSinh có thể truy nhập các thành phần private của lớp SinhVien.
- Tuy nhiên cách làm này đòi hỏi phải sửa lại các lớp cơ sở có liên quan: không hợp lý vì vi phạm tính đóng gói.

  => Khai báo các thuộc tính cần được truy nhập ở lớp dẫn xuất là thành phần protected thay vì private

3.8.1.2 Thành phần **public**

Tất cả các thành phần public của lớp cơ sở/lớp dẫn xuất đều có thể truy nhập được tại bất kỳ chỗ nào trong chương trình (thông qua tên lớp hoặc đối tượng của lớp cơ sở/lớp dẫn xuất).

3.8.1.3 Thành phần **protected**

- Để trong lớp dẫn xuất có thể truy nhập đến các thuộc tính của lớp cơ sở ta khai báo **các thuộc tính này là thành phần protected** của lớp cơ sở.
- Như vậy thành phần protected của lớp cơ sở được phép truy nhập trong các lớp dẫn xuất trực tiếp từ lớp này.
- Thường thì **các phương thức sẽ là thành phần public** của lớp cơ sở.

3.8.1.3 Thành phần protected – Ví dụ

```
class Nguoi {
    protected:
      char *HoTen;
      int NamSinh;
    public:
      //...
};
//Khi đó bên trong lớp SinhVien có thể truy nhập các thuộc tính HoTen,NamSinh của lớp Nguoi
```

```
class SinhVien : public Nguoi {
protected:
    char *MaSo;
public:
    SinhVien(char *ht, char *ms, int ns) : Nguoi(ht,ns){
      MaSo = _strdup(ms);}
    ~SinhVien(){
      delete [ ] MaSo;}
    void Xuat() const;
};
void SinhVien::Xuat() const {
    cout << “MSSV: " << MaSo;
    cout   << ", ho ten: " << HoTen;
    cout   << ", nam sinh: " << NamSinh;
}
```

3.8.1.3 Thành phần **protected** – Ví dụ (tt)

```
class NuSinh : public SinhVien {
public:
    NuSinh(char *ht, char *ms, int ns) : SinhVien(ht,ms,ns){
    }
    void An() const {
      cout << HoTen << " co ma so la " << MaSo;
    }
};
```

//Khi đó bên trong lớp NuSinh có thể truy nhập các thuộc tính HoTen, NamSinh của lớp Nguoi và thuộc tính MaSo của lớp SinhVien

Phạm vi truy xuất trong kế thừa

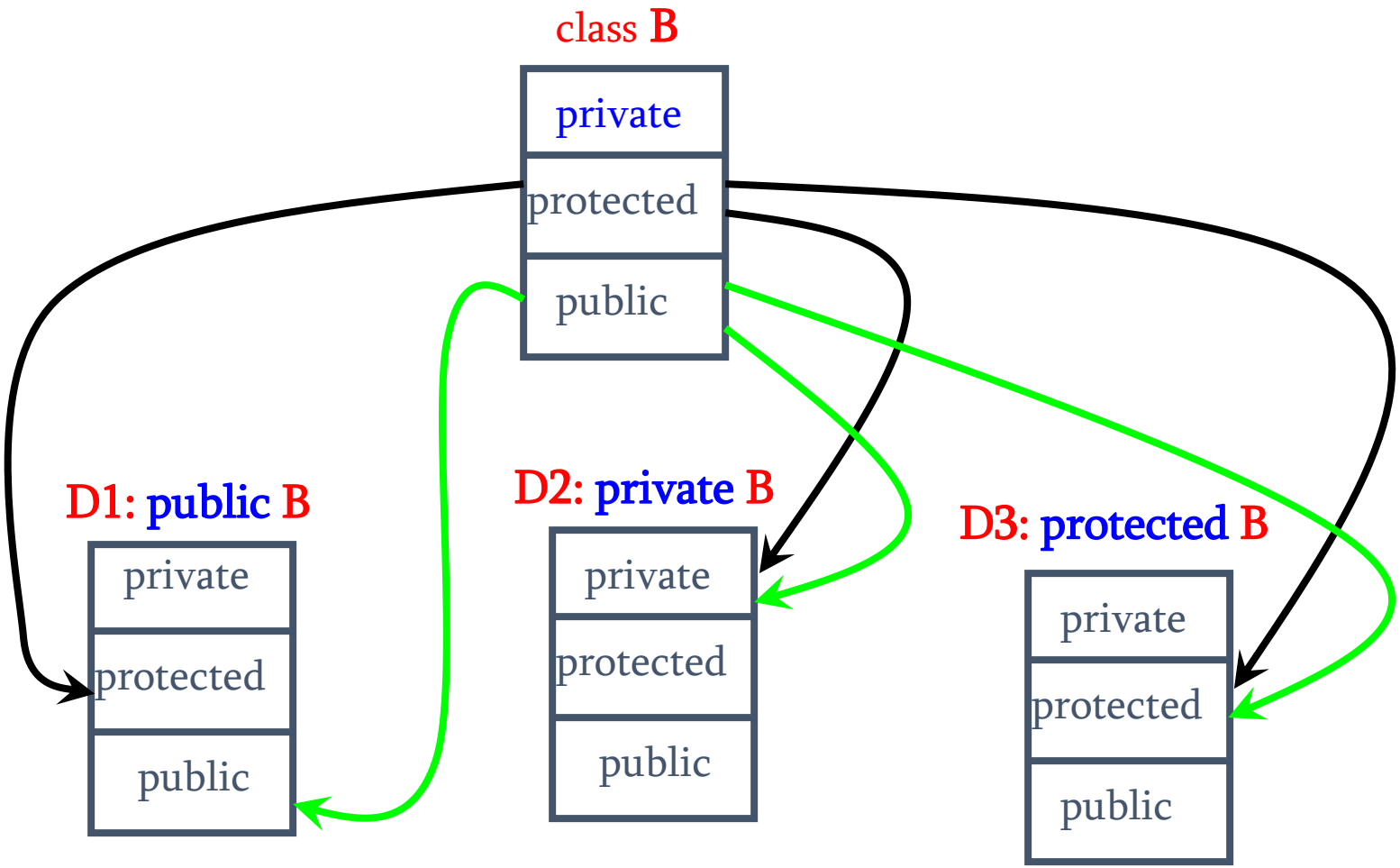
Type of Inheritance

Access Control for Members	private	Protected	public
private	?	?	?
protected	?	?	?
public	?	?	?

3.8.2 Truy xuất theo chiều ngang

- Kiểu dẫn xuất private
- Kiểu dẫn xuất public
- Kiểu dẫn xuất protected

3.8.2 Truy xuất theo chiều ngang (tt)



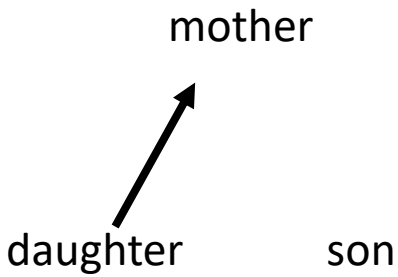
3.8.2 Truy xuất theo chiều ngang (tt)

- **Kiểu dẫn xuất private:** các thành phần protected và public của lớp cơ sở sẽ trở thành các thành phần **private** của lớp dẫn xuất.
- **Kiểu dẫn xuất public:** các thành phần protected và public của lớp cơ sở sẽ trở thành các thành phần **protected và public** của lớp dẫn xuất.
- **Kiểu dẫn xuất protected:** các thành phần protected và public của lớp cơ sở sẽ trở thành các thành phần **protected** của lớp dẫn xuất.

3.8.2 Truy xuất theo chiều ngang (tt)

Base class member access specifier	Type of inheritance		
	public inheritance	**protected** inheritance	**private** inheritance
Public	**public** in derived class. Can be accessed directly by any non-**static** member functions, **friend** functions and non-member functions.	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**private** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.
Protected	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**private** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.
Private	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.

3.8.2 Truy xuất theo chiều ngang – Ví dụ




```
class mother{
    protected:
      int x, y;
    public:
      void set(int a, int b);
    private:
      int z;
};
```

```
class daughter : public mother{
    private:
      double a;
    public:
      void foo ( );
};
```

```
void daughter :: foo ( ){
    x = y = 20;
    set(5, 10);
    cout<<"value of a "<<a<<endl;
    z = 100;
}
```

daughter can access 3 of the 4 inherited members

3.8.2 Truy xuất theo chiều ngang – Ví dụ

```
class mother{
    private:
      int z;
    protected:
      int x, y;
    public:
      void set(int a, int b);
};
```

//Các thành phần protected và public của lớp mother sẽ trở thành các thành phần protected và public của lớp daughter

```
class daughter : public mother{
    private:
      double a;
    public:
      void foo();
};
void daughter :: foo(){
    x = y = 20;
    set(5,10);
    cout << "Value of a " << a << endl;
    z = 100; //Báo lỗi vì z là thành phần private của lớp cơ sở
}
```

3.8.2 Truy xuất theo chiều ngang – Ví dụ (tt)

```
class mother{
    private:
      int z;
    protected:
      int x, y;
    public:
      void set(int a, int b);
};
```

//Trong lớp son vẫn truy nhập được các thành phần protected và public của lớp mother nhưng các lớp dẫn xuất của lớp son sẽ không truy nhập được các thành phần này.

```
class son : private mother{
    private:
      double b;
    public:
      void foo();
};
void son :: foo ){
    x = y = 20;
    set(5,10);
    cout << "Value of b " << b << endl;
     z = 100; //Báo lỗi
}
```

4. Hàm tạo, hàm hủy và hàm toán tử gán trong thừa kế

Lớp dẫn xuất **không thừa kế** các HÀM TẠO, HÀM HỦY, HÀM TOÁN TỬ GÁN của các lớp cơ sở.

4.1 Hàm tạo của lớp dẫn xuất

- Phương thức thiết lập của lớp cơ sở luôn luôn được gọi mỗi khi có một đối tượng của lớp dẫn xuất được tạo ra.

- Nếu mọi phương thức thiết lập của lớp cơ sở đều đòi hỏi phải cung cấp tham số thì lớp con bắt buộc phải có phương thức thiết lập để cung cấp các tham số đó

4.1 Hàm tạo của lớp dẫn xuất

- Vì trong lớp dẫn xuất không truy nhập được các thuộc tính là thành phần private của lớp cơ sở nên để gán giá trị cho các thuộc tính này phải sử dụng hàm tạo của lớp cơ sở (hàm tạo là thành phần public).

  => Lớp cơ sở phải cung cấp hàm tạo có đối để gán giá trị cho các thuộc tính là thành phần private của nó.

  Cách dùng: Tên_lớp_cơ_sở(danh sách giá trị)

4.1 Hàm tạo của lớp dẫn xuất (tt)

- Ngoài ra, nếu lớp dẫn xuất có thành phần kiểu lớp (đối tượng thành phần) thì phải dùng hàm tạo của lớp tương ứng để gán giá trị cho các thuộc tính của đối tượng này vì trong lớp dẫn xuất không truy nhập được các thuộc tính này.

  => Lớp tương ứng phải cung cấp hàm tạo có đối để gán giá trị cho các thuộc tính là thành phần private của nó.

  Cách dùng: Tên_đối_tượng_thành_phần(danh sách giá trị)

4.1 Hàm tạo của lớp dẫn xuất – Ví dụ

```
class A {
public:
    A(){ cout << "A: default" << endl; }
    A(int a){ cout << "A: parameter" << endl; }
};
class B : public A {
public:
    B(int a){ cout << "B" << endl; }
```

Hoặc:

```
    B(int a): A() { cout << "B" << endl; }
```

=> Đều gọi đến hàm tạo không đối của lớp A

```
};
```

```
int main() {
        B bb(1);
}
```

Output:
A: default
B

4.1 Hàm tạo của lớp dẫn xuất – Ví dụ (tt)

```
class A {
public:
    A(){ cout << "A: default" << endl; }
    A(int a){ cout << "A: parameter" << endl; }
};
class C : public A
{
public:
    C (int a) : A(a) {
        cout << "C" << endl;
    }
};
```

```
int main() {
        C cc(1);
}
```

Output:

A: parameter

C

4.2 Hàm toán tử gán của lớp dẫn xuất

Khi lớp dẫn xuất có thuộc tính là con trỏ *(kể cả thuộc tính thừa kế từ các lớp cơ sở)* **thì phải xây dựng hàm toán tử gán cho lớp dẫn xuất.** Cách xây dựng:

- Xây dựng hàm toán tử gán cho lớp cơ sở;
- Xây dựng hàm trả về địa chỉ của đối tượng ẩn của lớp cơ sở theo mẫu:

```
Tên_lớp * get_DTA(){ return this; }
```

- Xây dựng hàm toán tử gán cho lớp dẫn xuất.

4.2 Hàm toán tử gán của lớp dẫn xuất – Ví dụ

```
class A {
    //…
public:
    A& operator=(A &h){
      //Gán các tt của A
      return *this;
    }
    A* get_A(){
      return this;
    }
};
```

=> Dùng hàm get_A để nhận địa chỉ của đối tượng gọi hàm.

```
class B : public A {
public:
    B& operator=(B &h){
      A *u1,*u2;
      u1 = this->get_A();
      u2 = h.get_A();
//Gán trên đối tượng thuộc lớp cơ sở để gán các thuộc tính mà B thừa kế từ A:
      *u1 = *u2;
//Gán các thuộc tính riêng của B
    }
};
```

4.3 Hàm tạo sao chép của lớp dẫn xuất

Khi lớp dẫn xuất có thuộc tính là con trỏ *(kể cả thuộc tính thừa kế từ các lớp cơ sở)* **thì phải xây dựng hàm tạo sao chép cho lớp dẫn xuất.** Cách xây dựng:

- Xây dựng hàm toán tử gán cho lớp dẫn xuất;
- Xây dựng hàm tạo sao chép cho lớp dẫn xuất.

4.4 Hàm hủy của lớp dẫn xuất

- Khi một đối tượng bị hủy đi, phương thức hủy bỏ của nó sẽ được gọi. Sau đó, các phương thức hủy bỏ của lớp cơ sở sẽ được gọi một cách tự động.
- Vì vậy, lớp con không cần và cũng không được thực hiện các thao tác dọn dẹp cho các thành phần thuộc lớp cha.

4.4 Hàm hủy của lớp dẫn xuất

- Khi xây dựng hàm hủy của lớp dẫn xuất chỉ cần quan tâm đến các thuộc tính khai báo thêm trong lớp dẫn xuất.
- Trong đó, nếu thuộc tính có kiểu lớp (đối tượng thành phần) thì hàm hủy của lớp đó sẽ được tự động gọi.
- Như vậy ta không cần để ý đến các đối tượng thành phần và các thuộc tính thừa kế từ các lớp cơ sở trong hàm hủy của lớp dẫn xuất.

4.4 Hàm hủy của lớp dẫn xuất – Ví dụ

```
class SinhVien : public Nguoi {
    char *MaSo;
public:
    SinhVien( char *ht, char *ms, int ns) : Nguoi(ht,ns){
     MaSo = _strdup(ms);
    }
    /*Hàm tạo sao chép của lớp SinhVien*/
    SinhVien(const SinhVien &s) : Nguoi(s){ /*Hàm tạo sao chép
                        của lớp Nguoi*/
     MaSo = _strdup(s.MaSo);
    }
    ~SinhVien() {delete [ ] MaSo;}
};
```

Định nghĩa các thành phần riêng

- Ngoài các thành phần được kế thừa, lớp dẫn xuất có thể định nghĩa thêm các thành phần riêng

```
class HinhTron : Diem {
    double r;
public:
    HinhTron( double tx, double ty, double rr) : Diem(tx, ty){
     r = rr;
    }
    void Ve(int color) const;
    void TinhTien( double dx, double dy) const;
};
HinhTron t(200,200,50);
```

Định nghĩa các thành phần riêng

- Lớp dẫn xuất cũng có thể override các phương thức đã được định nghĩa ở trong lớp cha.

```
class A {
  protected:
    int x, y;
   public:
    void print (){
        cout<<“From A”<<endl;
    }
};
```

```
class B : public A
{
   public:
    void print (){
        cout<<“From B”<<endl;
    }
};
```

5. Ứng dụng của tính thừa kế

1) Có thể dùng tính thừa kế để phát triển khả năng của chương trình bằng cách xây dựng thêm các lớp dẫn xuất từ các lớp đã có, trong đó có thêm các thuộc tính và phương thức mới.

   Ngoài ra, ta cũng có thể xây dựng các lớp mới có thuộc tính là đối tượng của các lớp đã có.

   Như vậy, sẽ nhận được một dãy các lớp ngày càng hoàn thiện và có nhiều khả năng hơn.

5. Ứng dụng của tính thừa kế (tt)

2) Có thể dùng tính thừa kế để sửa đổi, bổ sung, nâng cấp chương trình.

Bằng cách xây dựng thêm các lớp dẫn xuất để thực hiện các bổ sung sửa đổi thay vì phải sửa chữa trên các lớp đã có.

5. Ứng dụng của tính thừa kế (tt)

3) Tính thừa kế cũng được dùng để thiết kế bài toán theo hướng từ khái quát đến cụ thể, từ chung đến riêng.

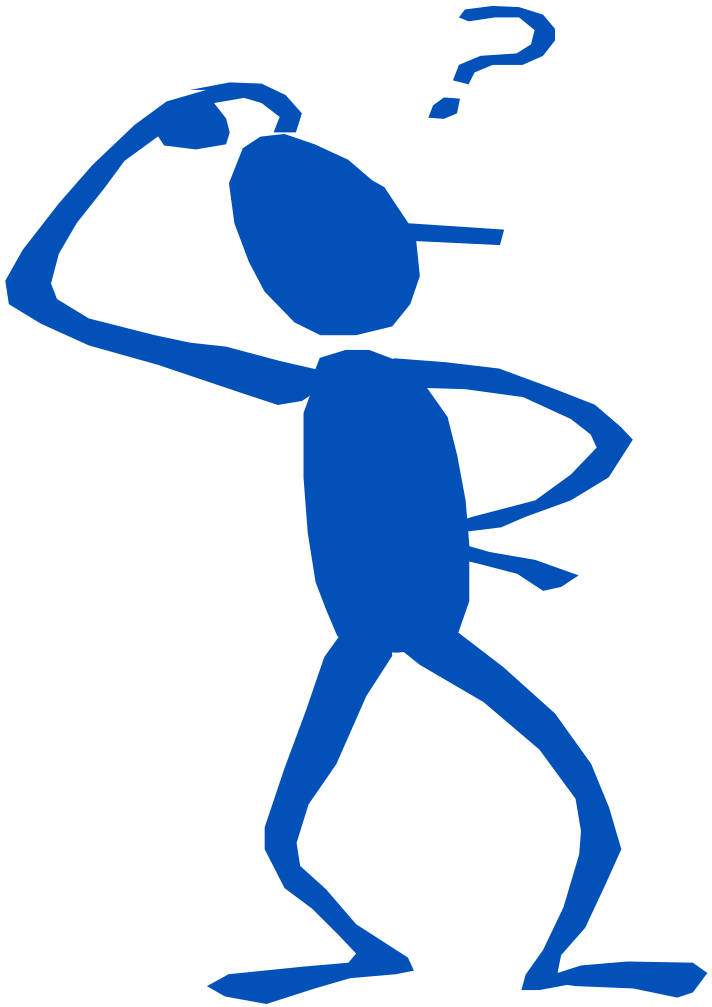
Đầu tiên đưa ra các lớp để mô tả những đối tượng chung, sau đó dẫn xuất tới các đối tượng ngày một cụ thể hơn.

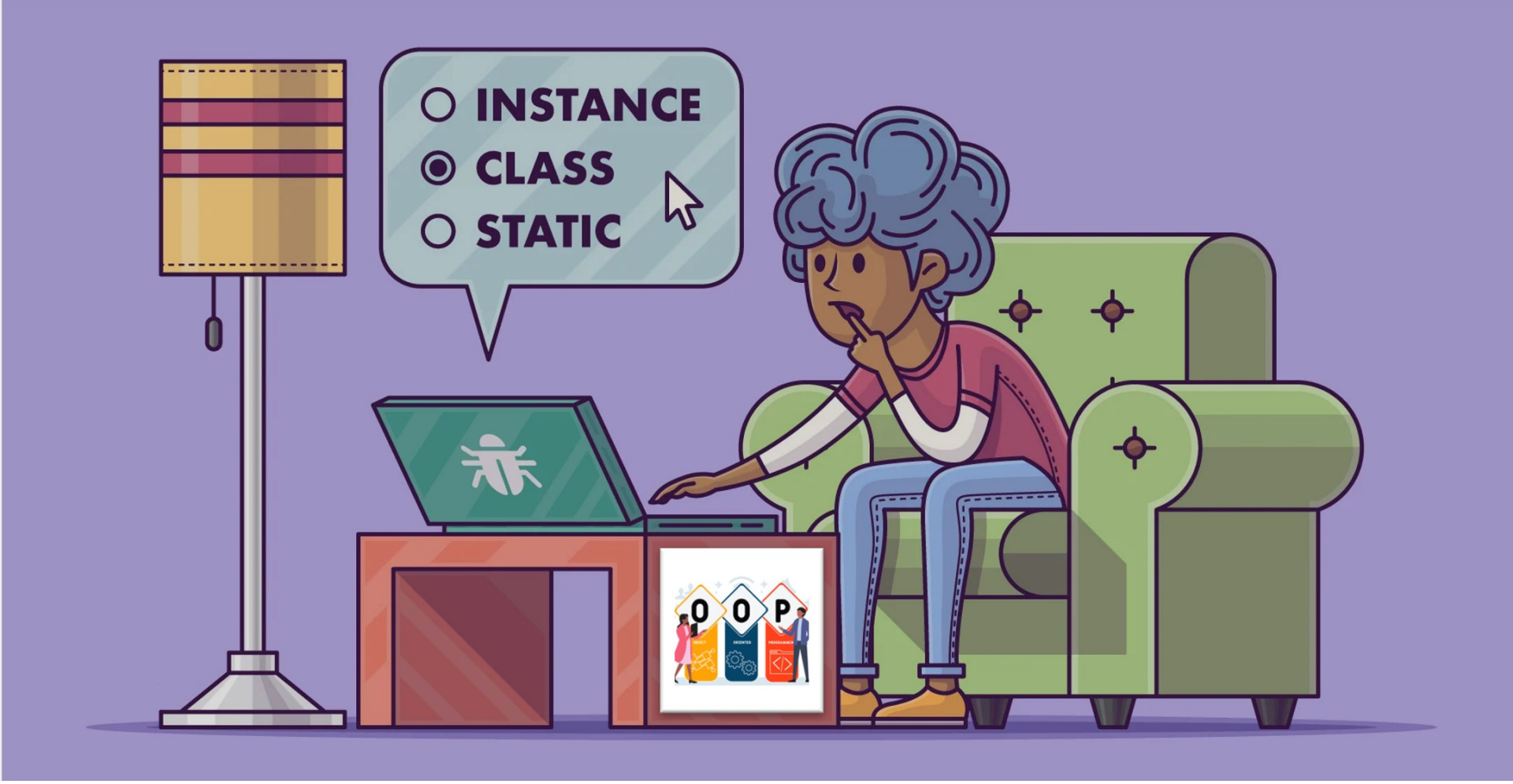
5. Ứng dụng của tính thừa kế (tt)

4) Tính thừa kế cũng được dùng trong việc thiết kế bài toán chung và bài toán bộ phận.

Khi đó ta có thể định nghĩa các lớp cho các bài toán bộ phận và lớp cho bài toán chung sẽ là lớp dẫn xuất của các lớp trên.

Q & A





IT002 - Lập trình hướng đối tượng

Trường ĐH CNTT – ĐHQG TP. HCM