清華大學
Tsinghua University

第10章 继承与多态

清华大学计算机系 韩文弢

教学内容

1 类的继承

2 多态

1、继承关系

- 类与类之间的关系
  - ☺ 泛化（Generalization）：“is a”关系，继承关系，一般类/特殊类，父类/子类，基类/派生类
  - ☺ 聚合（Aggregation）：“part of”关系，整体/部分
  - ☺ 关联（association）：类与类之间存在某种语义关联

Why继承关系？

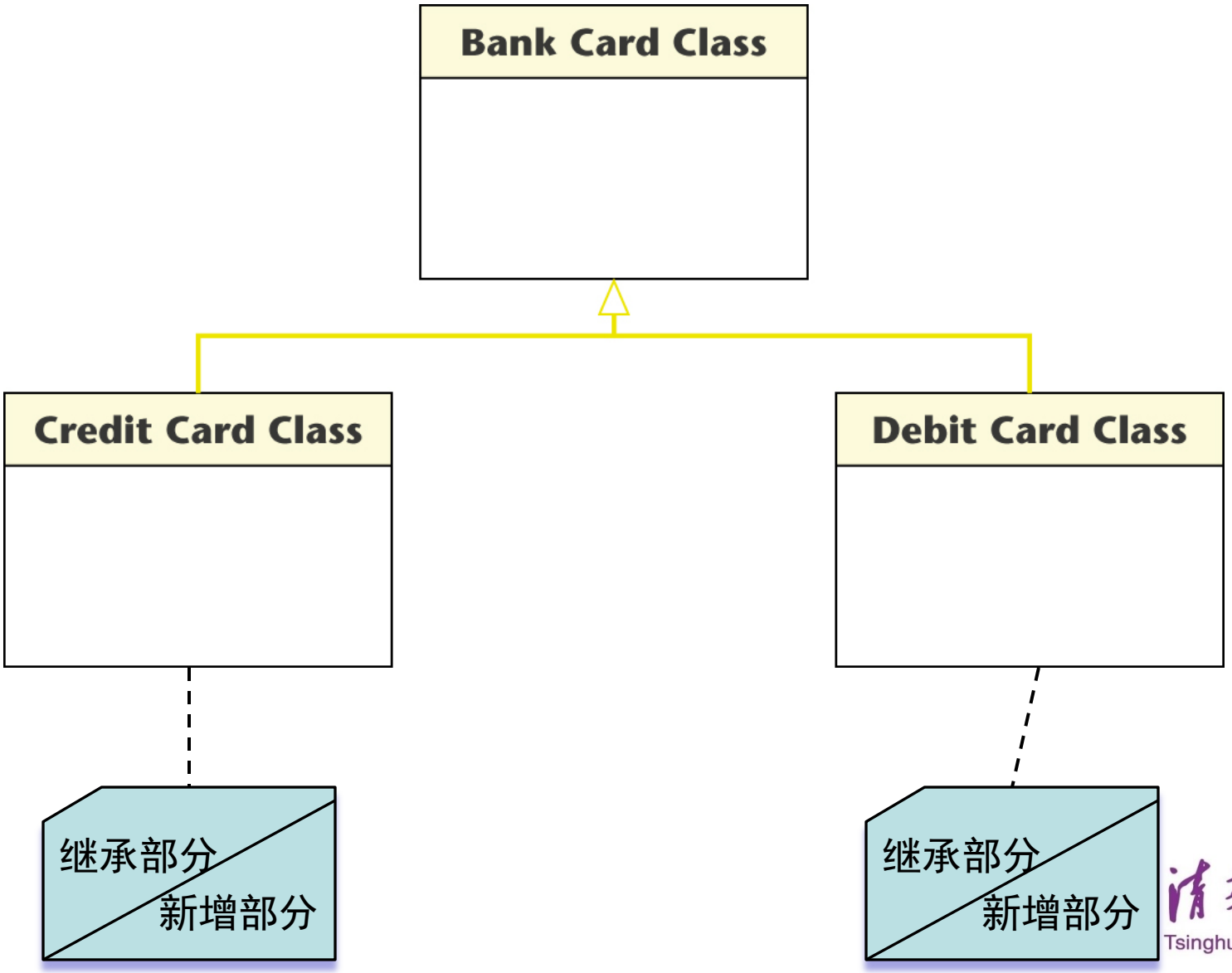
☺ 便于软件重用。例如：某电子商务公司允许客户使用信用卡网上购物

Credit Card Class
cardNumber nameOfCardholder expirationDate
validateTransaction ( ) chargeTransactionToCard ( ) creditPaymentToCard ( ) printMonthlyStatement ( )

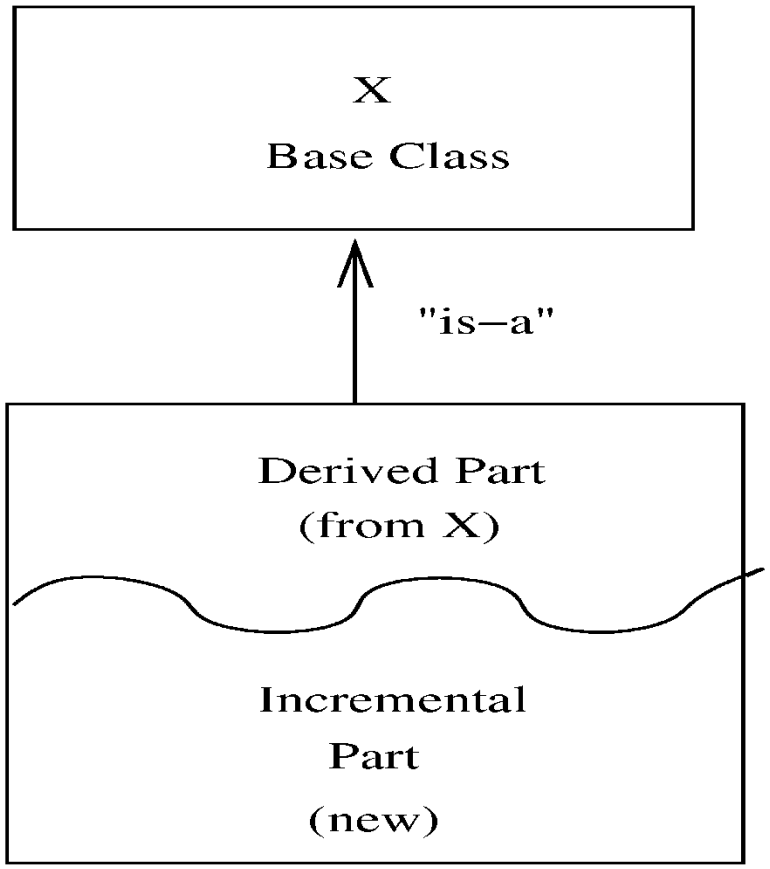
Why继承关系(2)

- 若系统升级，允许借记卡
  - ☺ 信用卡与借记卡有许多共同的属性，如**cardNumber, nameOfCardholder, expirationDate**
  - ☺ 主要区别在于：信用卡有信用额度，而借记卡有卡余额

Why继承关系(3)


Bank Card Class
Credit Card Class
Debit Card Class
继承部分
新增部分
继承部分
新增部分

继承关系

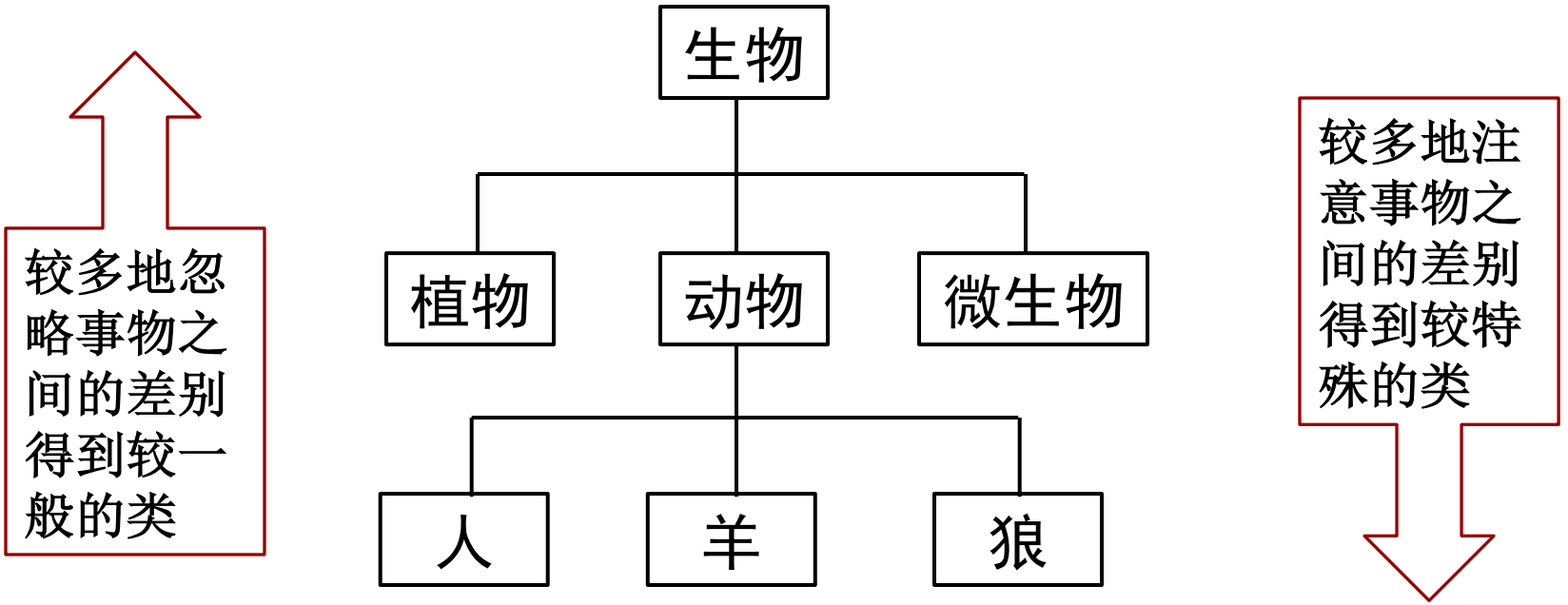


Y – Derived class

- Y继承了X的属性和操作
- Y只需定义新增的属性和操作
- Y的对象同时也是X的对象

层次结构

- 类的层次结构
  - ☺ 不同程度的抽象可得到不同层次的类



Dude类

```
class Dude {
public:
    string name;
    int hp;     // 血量
    int mp;     // 魔力值

    Dude() { hp = 100; mp = 0; }
    void sayName() {
        cout << name << endl;
    }
    void punchFace(Dude &target) {
        target.hp -= 10;
    }
};
```

Wizard类

```
class Wizard {

    // A Wizard does and has
    // everything a Dude does and
    // has!
    // Copy and paste Dude's stuff?
}
```

Wizard类

```
class Wizard {
public:
    string name;
    int hp;     // 血量
    int mp;     // 魔力值

    Wizard() { hp = 100; mp = 0; }
    void sayName() {
        cout << name << endl;
    }
    void punchFace(Dude &target) {
        target.hp -= 10;
    }
};
```


清華大學 Tsinghua University

利用继承关系!

- Wizard是Dude的一个子类

```
class Wizard : public Dude {

};
```

利用继承关系!

- Wizard可使用Dude中的所有public变量

```
wizard1.hp += 1;
```

- Wizard可调用Dude中的所有public函数

```
wizard1.punchFace(dude1);
```

- 你可以像使用Dude那样使用Wizard

```
dude1.punchFace(wizard1);
```

完善Wizard类

```
class Wizard : public Dude {
    string spells[20];
public:
    void cast(string spell){
        // cool stuff here
        ...
        mp -= 10;
    }
};
```

继承再继承

```
class GrandWizard : public Wizard {
public:
    void sayName(){
        cout << "Grand wizard " << name;
    }
};

grandWizard1.name = "Flash";
grandWizard1.sayName();
```

Grand wizard Flash

如何实现？

C++看到如下语句时会如何做？

```
grandWizard1.punchFace(dude1);
```

1. 在GrandWizard类中寻找punchFace();
2. 没有找到！grandWizard有父类吗？
3. 在Wizard类中查找punchFace();
4. 没有找到！Wizard有父类吗？
5. 在Dude类中查找punchFace();
6. 找到了！调用punchFace();
7. 减少dude1的hp值

举例(1)

```
class Parent {
private:
    int b;
protected:
    int c;
public:
    int a;
    Parent(){ a = 10; b = 20; c = 30; }
    int getB()  { return b; }
};

class Child : public Parent {
};
```

```
#include <iostream>
using namespace std;
void main()
{
    Child child;
    cout << child.a << endl;    //允许
    cout << child.b << endl;    //不允许
    cout << child.getB() << endl; //允许
    cout << child.c; << endl    //不允许
}
```

举例(2)

```
class Parent {
};
class Child : public Parent {
public:
    string name;
    void setName(string s) {
        name = s;
    }
};
void main() {
    Parent *a = new Child();
    a->setName("悟空");
    cout << a->name << endl;
}
```

可以有不同的继承方式

- 不同继承方式下类成员的访问控制
  - ☺ 三种继承方式：公有继承(public)、私有继承(private)、保护继承(protected)
  - ☺ 父类成员的访问属性的继承问题
  - ☺ 子类成员函数对父类成员的访问权限
  - ☺ 通过子类对象对父类成员的访问权限

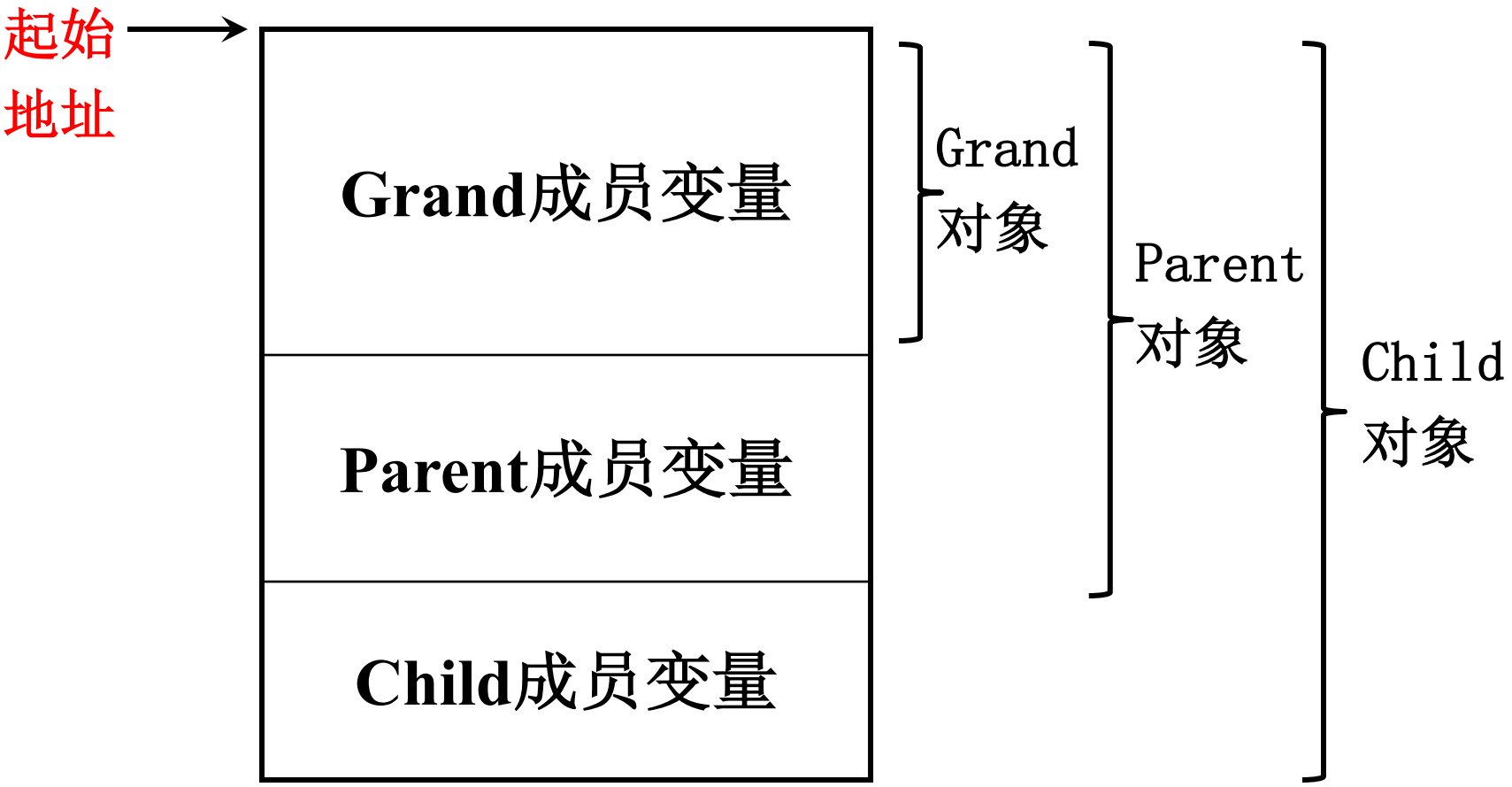
三种继承方式的访问权限

	访问属性的继承	子类成员函数	子类对象
公有继承	父类的public和protected成员的访问属性在子类中不变，private不能访问	可访问父类中的public和protected成员，不能访问private成员	只能访问从父类继承的public成员
私有继承	父类的public和protected成员以private出现在子类，private不能访问	同上	不能访问从父类继承的任何成员
保护继承	父类的public和protected成员以protected出现在子类，private不能访问	同上	不能访问从父类继承的任何成员

2、子类对象的存储

- 在创建一个子类对象后
  - ☺ 一方面，该子类对象本身是一个独立、完整的对象
  - ☺ 另一方面，在该对象内部，又包含了一个父类子对象（**subobject**）
  - ☺ 该子对象与正常创建的父类对象相同

具体实现



演示父子类对象存储

```
class Grand {
public:
    int grandpa, grandma;
};
class Parent : public Grand {
public:
    int dad, mom;
};
class Child : public Parent {
public:
    int son, daughter;
};
void main() {
    Child c;
}
```

示例

```
class Man {
public:
    string name;
};
class IronMan : public Man {
public:
    string nickname;
    void print() {
        cout << "Man: " << name << endl;
        cout << "Ironman: " << nickname << endl;
    }
};

    IronMan *im = new IronMan();
    im->nickname = "钢铁侠";
    Man *m = im;
    m->name = "Tony Stark";
    im->print();
```

```
Man: Tony Stark
Ironman: 钢铁侠
```

m->nickname = "钢铁侠";

m->print();

3、构造函数

- 父类和子类的构造函数
  - ☺ 父类和子类的构造函数各自负责初始化自身的成员变量
  - ☺ 若构造函数无参数，在创建子类对象时系统会自动先调用父类的构造函数
  - ☺ 若构造函数有参数，在子类的构造函数中需要给父类的构造函数传递参数

构造函数无参数

```
class CPU8086 {
public:
  CPU8086() {
    cout << "8086 constructor" << endl;
  }
};
class CPU286 : public CPU8086 {
public:
  CPU286() {
    cout << "286 constructor" << endl;
  }
};
```

构造函数无参数(2)

```
class CPU386 : public CPU286 {
public:
   CPU386() {
      cout << "386 constructor" << endl;
   }
};

void main()
{
     CPU386 cpu;
}
```

```
8086 constructor
286 constructor
386 constructor
```

构造函数有参数

```
子类名::子类名(父类所需的形参，子类成员所需的形参)：父类名(参数表)，子类成员初始化列表
{
    //其他初始化;
};
```


清華大學 Tsinghua University

构造函数有参数(2)

```
class CPU8086 {
public:
  CPU8086(int i) {
    cout << "8086 constructor: " << i << endl;
  }
};
class CPU286 : public CPU8086 {
public:
  CPU286(int i, int j) : CPU8086(i) {
    cout << "286 constructor: " << j << endl;
  }
};
```

构造函数有参数(3)

```
class CPU386 : public CPU286 {
public:
  CPU386(int i, int j, int k) : CPU286(i,j){
    cout << "386 constructor: " << k << endl;
  }
};
void main()
{
    CPU386 cpu(8086, 286, 386);
}
```

4、类的继承举例

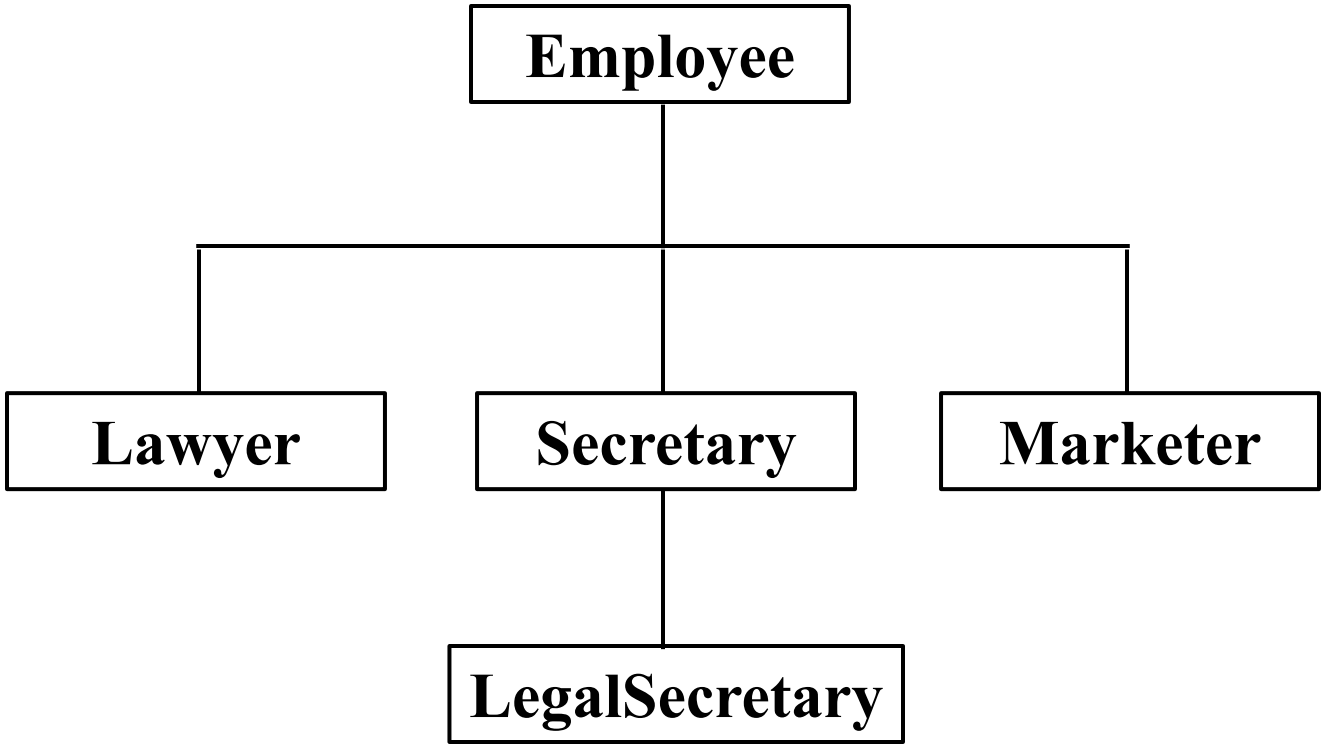
- 律师事务所
  - ☺ 公司有一些公共的规章制度，包括每周工作时间（40小时）、假期长度（10天）、年薪（40000）等
  - ☺ 公司有不同类型的员工，每一类员工有一些特定的规章制度，或是对公共的规章制度的修正

✦ 公司员工

- ☺ 律师：负责打官司，额外5天休假
- ☺ 市场销售：做广告宣传，年薪比普通员工高10000
- ☺ 秘书：负责文字记录
- ☺ 法律秘书：除普通秘书工作外，还负责撰写法律文书，年薪高5000

定义几个类？类之间关系如何？

类的层次结构



Employee类

```
class Employee {
public:
    int getHours() {
        return 40;          // works 40 hours / week
    }
    double getSalary() {
        return 40000.0;     // $40,000.00 / year
    }
    int getVacationDays() {
        return 10;          // 2 weeks' paid vacation
    }
    string getVacationForm() {
        return "yellow";    // use the yellow form
    }
};
```

Secretary和Lawyer类

```
class Secretary : public Employee {
public:
    void takeDictation() {
        cout << "Writing that down!" << endl;
    }
};
```

```
class Lawyer : public Employee {
public:
    string getVacationForm() {
        return "pink";
    }
    void sue() {
        cout << "I'll see you in court!" << endl;
    }
    int getVacationDays() {
        return Employee::getVacationDays() + 5;
    }
};
```

LegalSecretary类

```
class LegalSecretary : public Secretary {
public:
    double getSalary() {
        double baseSalary = Secretary::getSalary();
        return baseSalary + 5000;
    }

    void fileLegalBriefs() {
        cout << "Filing your briefs!" << endl;
    }
};
```

教学内容

1 类的继承

2 多态

清華大學 Tsinghua University

多态

- 多态(Polymorphism)
  - ☺ 一个指针或引用根据它所指向的对象类型来改变其行为的能力
  - ☺ 这允许不同子类的多个对象可被视为同一个父类的对象，却又能在运行时根据各个对象所属的子类自动地选择合适的成员函数去执行

重写(overriding)

```
class Foo {
    void method() {
        ...do something...
    }
}
class Bar : public Foo {
    void method() { //函数原型相同
        ...do something else...
    }
}
```

清華大學
Tsinghua University

Why多态?

```
class Animal {
public:
    void sound() {
        cout << "Sound for an animal" <<endl;
    }
};
class Dog : public Animal {
public:
    void sound() {  // Overriding
        cout << "汪汪" << endl;
    }
};
```

```
class Cat : public Animal {
public:
    void sound() {  // Overriding
        cout << "喵喵" << endl;
    }
};
class Duck : public Animal {
public:
    void sound() {
        cout << "呱呱" << endl;
    }
};
```

```
void main( )
{
    Dog *pDog = new Dog();
    Cat *pCat = new Cat();
    Duck *pDuck = new Duck();
    pDog->sound();
    pCat->sound();
    pDuck->sound();
}
```

如果想对所有的Animal对象进行批量或参数化处理？

清華大學 Tsinghua University

Why多态?

```
<Type>* pAnimals[3];
// 初始化pAnimals数组
for(int i = 0; i < 3; i++) {
    pAnimals[i]->sound();
}

    MakeSound(pDog);
    MakeSound(pCat);
    MakeSound(pDuck);
```

重载法?

```
void MakeSound(Dog *pDog) {
    pDog->sound();
}
void MakeSound(Cat *pCat) {
    pCat->sound();
}
void MakeSound(Duck *pDuck) {
    pDuck->sound();
}
```

Why多态?

需要这样一种机制：合而不同。即在编写代码时将各种子类对象按照相同类型来处理，以简化代码；而在运行时，分别绑定到各自不同的实现函数。

数组化

```
Animal* pAnimals[3];
pAnimals[0] = new Dog(); //upcasting
pAnimals[1] = new Cat();
pAnimals[2] = new Duck();
for(int i = 0; i < 3; i++) {
    pAnimals[i]->sound(); //哪个sound?
}
```

参数化

```
void MakeSound(Animal *pAnimal){
    pAnimal->sound();
}
Dog *pDog = new Dog();
Cat *pCat = new Cat();
Duck *pDuck = new Duck();
MakeSound(pDog); //upcasting
MakeSound(pCat);  //? 的sound()
MakeSound(pDuck); //? 的sound()
```

我们期望在程序运行时，能根据所指向的对象类型来决定调用相应的成员函数，怎么办？

虚函数

```
class Animal {
public:
    virtual void sound() {
        cout << "Sound for an animal" <<endl;
    }
};
class Dog : public Animal {
public:
    virtual void sound() {  // Overriding
        cout << "汪汪" << endl;
    }
};
```

函数调用绑定

- 绑定（binding）：把一个函数调用与相应的方法体关联起来
- 静态绑定
  - ☺ 在程序运行之前由编译器或链接器完成
- 动态绑定
  - ☺ 编译时并不知道，而是在程序运行时完成，基于对象的类型来决定去向

C语言何种类型？

多态例子之一

```
class Human{
public:
    virtual void eat() {
        cout << "Human is eating" << endl;
    }
    void walk() {
        cout << "Human is walking" << endl;
    }
};
class Boy : public Human{
public:
    virtual void eat(){
        cout << "Boy is eating" << endl;
    }
    void play(){
        cout << "Boy is playing" << endl;
    }
};
```

多态例子之一(cont.)

```
void main( )
{
     Boy *obj = new Boy();
     obj->eat();
     obj->walk();
     obj->play();
}
```

- 子类对象、子类指针
- 子类对象、父类指针
- 父类对象、父类指针

多态例子之二

```
class Foo {
public:
    virtual void method1() {
        cout << "foo 1" << endl;
    }
    virtual void method2() {
        cout << "foo 2" << endl;
    }
    virtual string toString() {
        return "foo";
    }
};
class Bar : public Foo {
public:
    virtual void method2() {
        cout << "bar 2" << endl;
    }
};
```

多态例子之二(2)

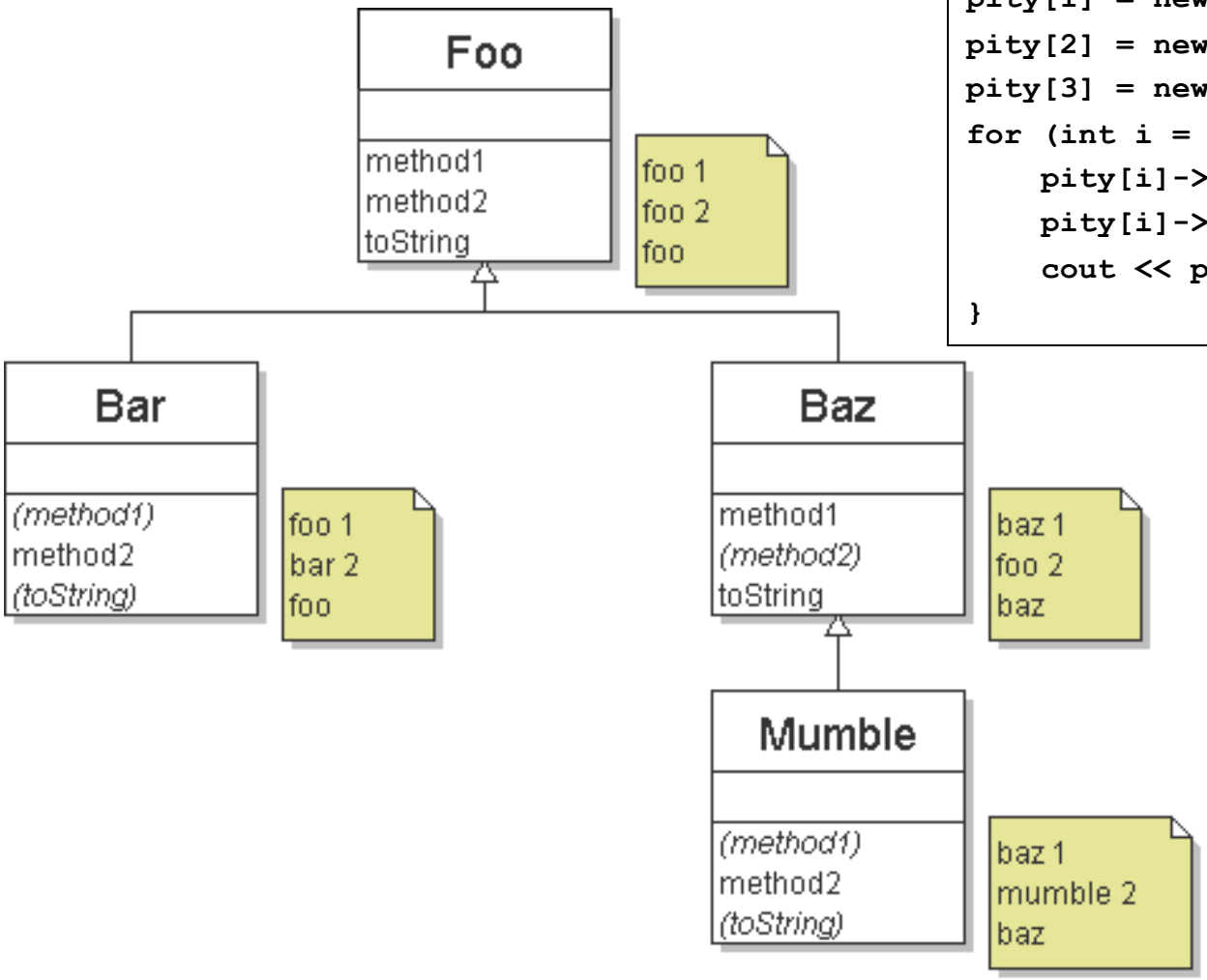
```
class Baz : public Foo {
public:
    virtual void method1() {
        cout << "baz 1" << endl;
    }
    virtual string toString() {
        return "baz";
    }
};
class Mumble : public Baz {
public:
    virtual void method2() {
        cout << "mumble 2" << endl;
    }
};
```

多态例子之二(3)

```
void main()
{
    Foo *pity[4];
    pity[0] = new Baz();
    pity[1] = new Bar();
    pity[2] = new Mumble();
    pity[3] = new Foo();
    for (int i = 0; i < 4; i++) {
        pity[i]->method1();
        pity[i]->method2();
        cout << pity[i]->toString() << endl;
    }
}
```

上述代码的输出结果是？

多态例子之二(4)



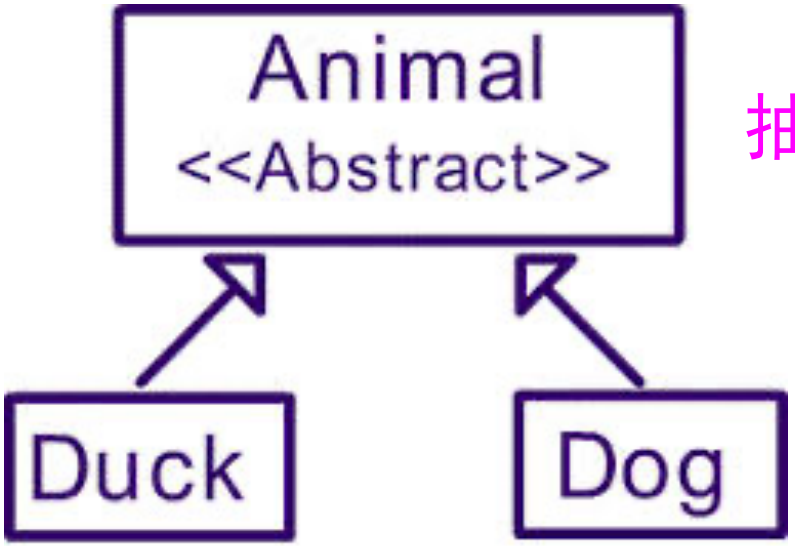

```
Foo *pity[4];
pity[0] = new Baz();
pity[1] = new Bar();
pity[2] = new Mumble();
pity[3] = new Foo();
for (int i = 0; i < 4; i++) {
    pity[i]->method1();
    pity[i]->method2();
    cout << pity[i]->toString() << endl;
}
```

```
baz 1
foo 2
baz

foo 1
bar 2
foo

baz 1
mumble 2
baz

foo 1
foo 2
foo
```

纯虚函数和抽象类

```
class Animal {
public:
    virtual void sound() { //虚函数
        cout << "Sound for an animal" <<endl;
    }
};
class Dog : public Animal {
public:
    virtual void sound() {  // Overriding
        cout << "汪汪" << endl;
    }
};
```

```
class Animal {
public:
    virtual void sound() = 0; //纯虚函数
};
```



抽象类

本讲小结

- 类的继承
- 多态