

## 第14课 音乐人与Rapper——类

### 1、类的继承和定制

#### 1-1、类的继承（财产被继承了）

类的继承，即让子类拥有了父类拥有的所有属性和方法



#### 1-2、类的定制（“我就是我，是颜色不一样的烟火”）

子类也可以在继承的基础上进行个性化的定制，包括：

- 创建新属性、新方法
- 修改继承到的属性或方法

### 2、继承的编写规则

#### 2-1、继承的基础语法

```
1 class X():
2     name='小K'
3 class Y(X):
4     pass
```

以上代码的第 3 行，`class Y(X)` 就是我们的继承语句，其中 Y 是子类，X 代表着我们的父类。

## 继承的语法

```
1 class X(Y):
```

子类名 父类名

注：小括号和冒号都是英文格式

我们的子类继承了父类，那就是说，父类的所有东西，我们子类都可以拿来用，下面我们来看一下例子：

```
1 class Car:
2     wheel = 4
3     def run(self):
4         print('有%d个轮子,可以飞速的行驶'%self.wheel)
5 class BMW(Car):
6     pass
7 BMW320 = BMW()
8 print(BMW320.wheel)
```

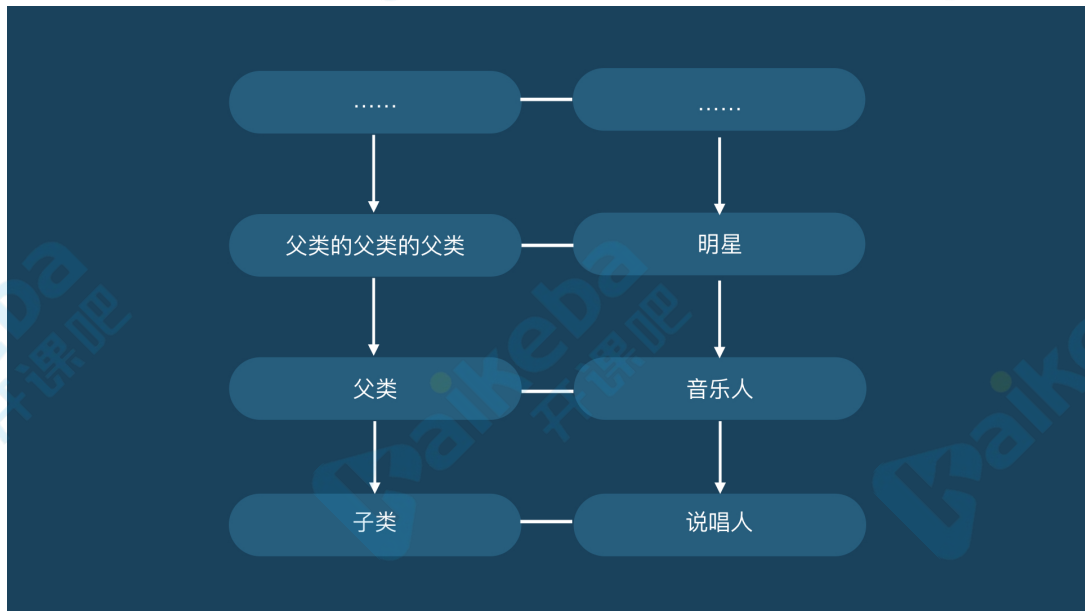
我们父类 Car 定义了变量wheel 并赋值4，而子类 BMW 继承了 Car但是什么都没操作（pass），我们实例化函数 BMW 后，实例化对象 BMW320可以调用我们父类 Car中的 wheel 属性。

## 各级实例和各级类间的关系

- 1.子类创建的实例，也属于父类；
- 2.父类创建的实例，不属于子类；
- 3.所有类创建的实例都属于根类object。

### 2-2、类的多层继承

继承不仅可以发生在两个层级之间（即父类-子类），还可以有父类的父类、父类的父类的父类.....



```
1 class Star:
2     glasses = "墨镜"
3     #音乐人继承了明星
4     class Musician(Star):
5         loveMusic = True
6         # Rapper继承了音乐人
7         class Rapper(Musician):
8             pass
9     csunYuk = Rapper()
10    print(csunYuk.glasses)
11    print(csunYuk.loveMusic)
```

上面例子中，类 Musician继承了类 Star，类 Rapper 再继承了类 Musician，即类 Rapper继承了前面两个的所有属性，所以可以正常打印出我们的“墨镜”,True。

## 2-3、类的多重继承

一个类，可以同时继承多个类，语法为“class A(B,C,D):”。和子类更相关的父类会放在更左侧，子类创建的实例在调用属性和方法时，会先在左侧的父类中找，找不到才会去右侧的父类找。（可理解为“就近原则”）

```
1 class panpan():
2     teacher='潘潘'
3     class momo():
4         teacher='墨墨'
5     class fudao():
6         teacher='辅导'
7     class main(panpan,momo,fudao):
8         pass
9     team = main()
10    print('老师'+team.teacher+'最棒')
```

以上代码中，类 main 同时继承了我们的类 panpan,momo,fudao，而根据我们的“就近原则”，我们在调用属性 teacher 的时候，类panpan 离我们的子类最近且有定义该类，所以打印出来的结果为“老师潘潘最棒”。

## 多层继承和多重继承

### 多层继承:

```
class Y(X):  
class Z(Y):
```

**作用：**类在纵向上的深度拓展。

**例子：**Rapper继承音乐人，音乐人继承明星。

**特点：**子类创建的实例，可调用所有层级的父类的属性和方法。

### 多重继承:

```
class Q(X,Y,Z):
```

**作用：**类在横向上的深度拓展。

**例子：**Rapper同时继承音乐人与演说家，音乐人继承明星。

**特点：**就近原则：在子类调用属性和方法时，优先考虑靠近子类（既靠左）的父类

## 3、定制的编写规则

### 3-1、新增代码

在子类下新建属性或方法，让子类可以用上父类所没有的属性或方法。这种操作，属于定制中的一种：新增代码

```
1 #音乐人  
2 class Musician():  
3     loveMusic = True  
4     def intr(self):  
5         print("我喜欢音乐")  
6         print("我来自音乐界")  
7     def sing(self):  
8         print("我在唱歌")  
9 # Rapper继承音乐人  
10 class Rapper(Musician): #类的继承  
11     garrulous = True    #类的定制， 增加属性  
12     def composition(self): #类的定制， 增加方法  
13         print("我可以创作简单的歌词")  
14 csunYuk = Rapper()  
15 print(csunYuk.loveMusic)  
16 print(csunYuk.garrulous)  
17 csunYuk.composition()  
18 #True  
19 #True  
20 我可以创作简单的歌词
```

以上代码中，类 `Rapper` 继承了类 `Musician`，但是它增加了属性 `garrulous`，增加了方法 `composition`，这就是类的定制。

我们实例化了类 `Rapper` 后，可以调用父类的属性和方法外，也可以给子类新增子类的属性和方法。

### 3-2、重写代码

①、重写代码，是在子类中，对父类代码的方法修改：

```
1 #音乐人
2 class Musician():
3     loveMusic = True
4     def intr(self):
5         print("我喜欢音乐")
6         print("我来自音乐界")
7     def sing(self):
8         print("我在唱歌")
9 #Rapper继承音乐人
10 class Rapper(Musician): #类的继承
11     def sing(self): #类的定制， 更改方法
12         print("我以说的形式进行歌唱")
13 csunYuk = Rapper()
14 csunYuk.sing()
```

以上代码中，我们子类 `Rapper` 继承了父类 `Musician`，并且重写了方法 `sing`，即我们用类 `Rapper` 的实例化对象去调用方法 `sing` 的时候，会打印出“我以说的形式进行歌唱”。

②、在子类中，对父类代码的实例化属性修改：

```
1 class Musician():
2     def __init__(self,name='周杰伦'):
3         self.name=name
4     def Name(self):
5         print('我的名字是'+self.name)
6 class Rapper(Musician):
7     def __init__(self,name='潘玮柏'):
8         Musician.__init__(self,name)
9 b=Rapper()
10 b.Name()
11 #我的名字是潘玮柏
```

以上代码中，我们子类 `Rapper` 继承了父类 `Musician`，调用父类的初始化函数后，重新给参数 `name` 赋值，再给这个 `name` 给实例化对象 `name`，具体传参路线如下：

```
class Musician():
    def __init__(self,name='周杰伦'):
        self.name=name
    def Name(self):
        print('我的名字是'+self.name)
class Rapper(Musician):
    def __init__(self,name='潘玮柏'):
        Musician.__init__(self,name)
b=Rapper()
b.Name()
#我的名字是潘玮柏
```



The diagram consists of two red arrows. The first arrow originates from the `self.name` attribute access in the `Musician.Name()` method and points to the `self` parameter in the `Musician.__init__()` method. The second arrow originates from the `self.name` attribute access in the `Rapper.Name()` method and points to the `self` parameter in the `Rapper.__init__()` method. This illustrates how the `self` object, which carries the `name` attribute, is passed to the `__init__` method of the parent class (`Musician`) when a `Rapper` instance is created.