

FSE598 前沿计算技术

模块 2 数据与数据处理 单元 4 面向对象的编程 第 1 讲 类定义

讲座的英文版内容基于本书：

Y. Chen 《编程语言入门：C、C++、Scheme、Prolog、C# 和 Python 编程》(Introduction to Programming Languages: Programming in C, C++, Scheme, Prolog, C#, and Python), 第 6 版, Kendall Hunt Publishing Company, 2019 年。

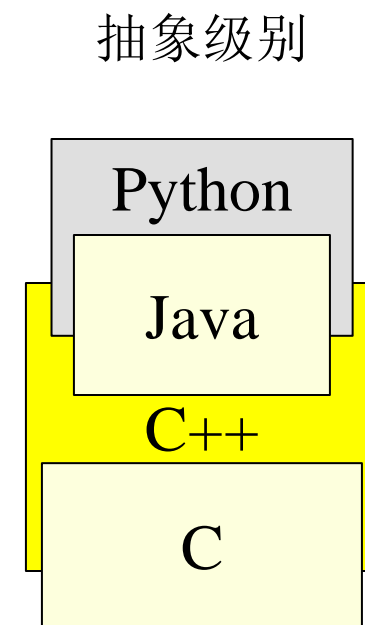
<https://www.public.asu.edu/~ychen10/book/IntroPl.html>

学习

- ❑ 面向对象 (Object Oriented) 的编程的概念
- ❑ 类 (Class) 定义
- ❑ 构造函数 (Constructor) 和析构函数 (Destructor)
- ❑ 类属性 (Class Attributes) 和实例属性 (Instance Attributes)
- ❑ 类方法 (Class Methods)

面向对象

- ❑ 目的是创建大型程序并保持程序的可读性和可理解性，从而易于管理。
- ❑ 计算机程序由数据结构和数据处理操作构成；
- ❑ 数据结构（对象）是重点：操作是对象的一部分，用于操作数据；
- ❑ 类是一种抽象数据类型：
 - 将状态封装在对象中，只能通过针对这些对象定义的操作来访问这些对象。
 - 简洁的界面——公共和私有组件。



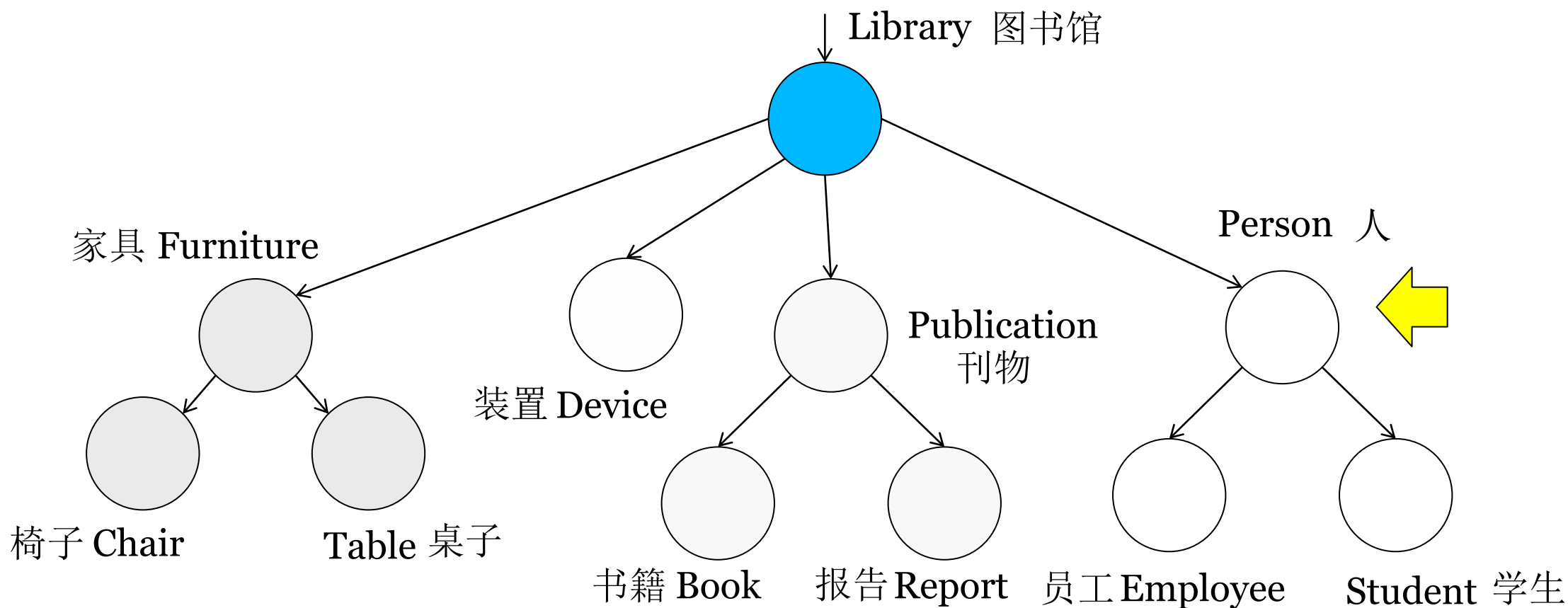
面向对象的关键概念

- ❑ 继承：
 - 重用类设计
 - 通过保持被继承类的未变的部分，允许添加新函数/方法来扩展类。支持代码重用。
- ❑ 类层次结构
 - 类根据继承关系按层次组织。
- ❑ 多态性
- ❑ 动态内存分配和取消分配
- ❑ 动态绑定

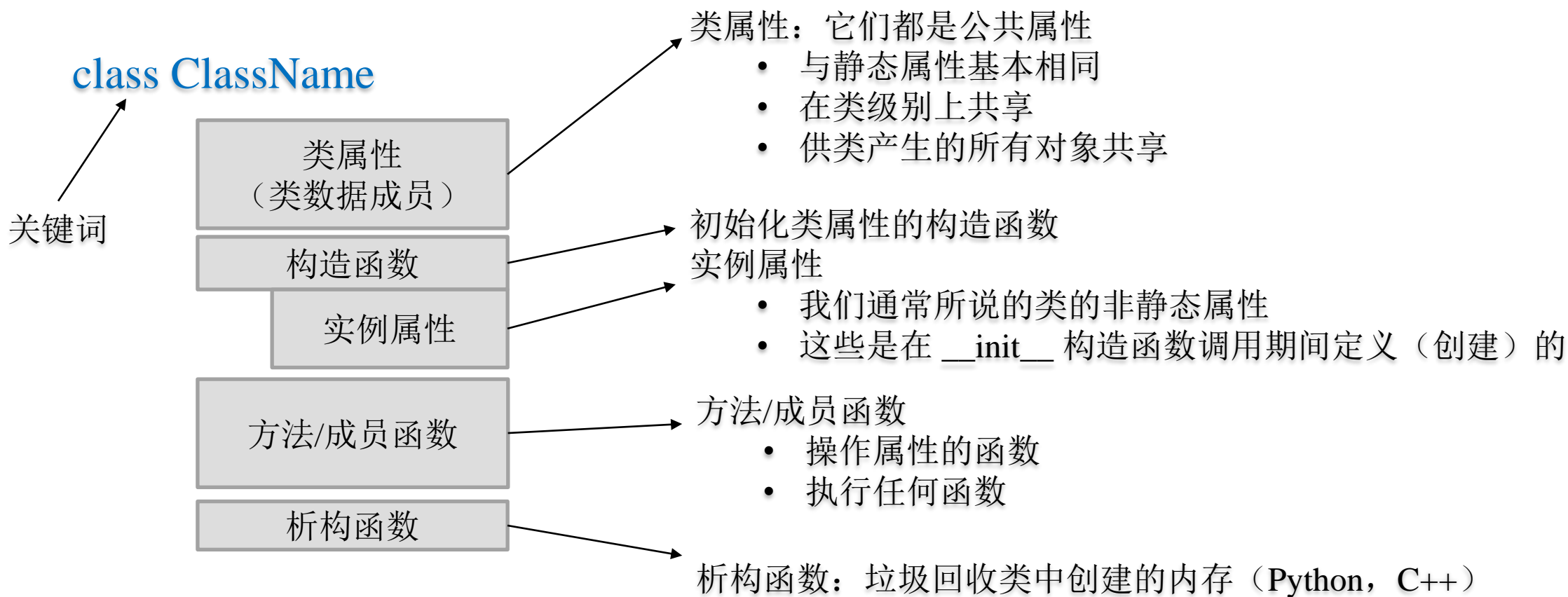
类继承和层次结构

继承在关联许多类时很有用，这样它们就可以重用设计并共享代码。

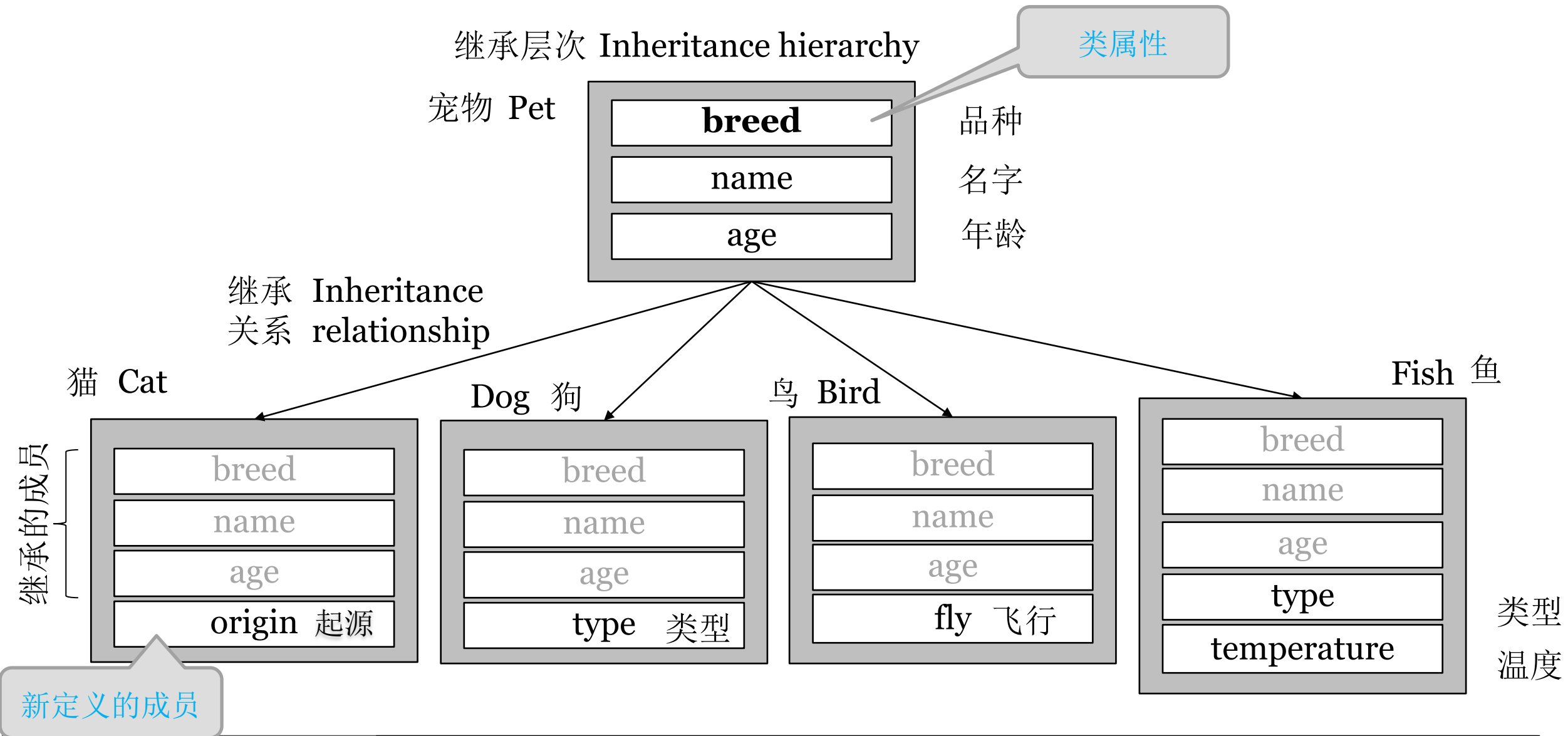
考虑图书馆的资源管理



□ 类定义了由多种成员组成的抽象数据类型



示例



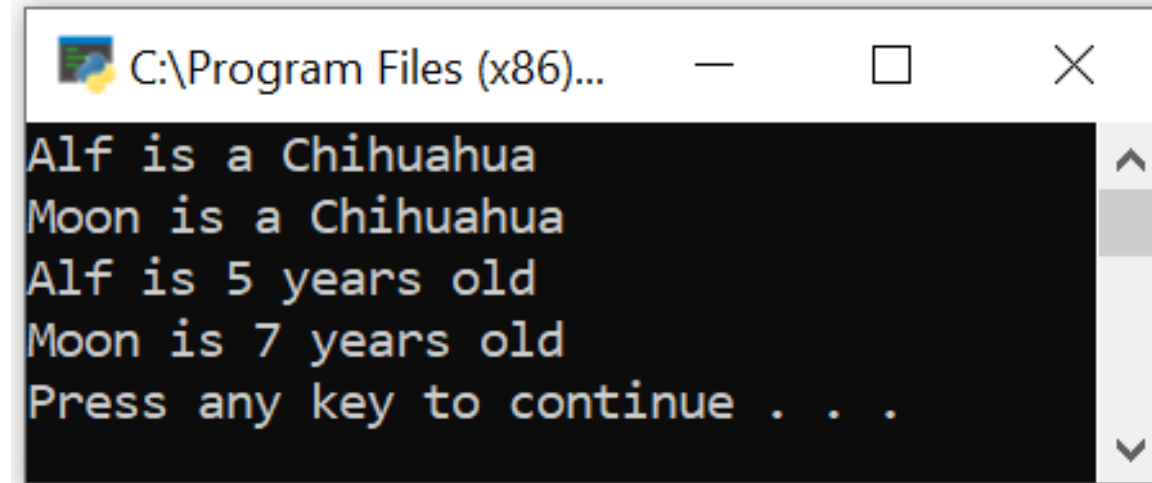
代码示例

```
class Pet:
    # class attribute
    breed = "Chihuahua"
    # constructor and instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

def main():
    # instantiate Pet class through constructor
    alf = Pet("Alf", 5)
    moon = Pet("Moon", 7)
    # access the class attributes
    print("Alf is a ", alf.__class__.breed)
    print("Moon is a {}".format(moon.__class__.breed))
    # access the instance attributes in constructor
    print("{} is {} years old".format(alf.name, alf.age))
    print(moon.name, " is ", moon.age, "years old")
if __name__ == "__main__":
    main()
```

类属性

实例属性



```
C:\Program Files (x86)...
Alf is a Chihuahua
Moon is a Chihuahua
Alf is 5 years old
Moon is 7 years old
Press any key to continue . . .
```

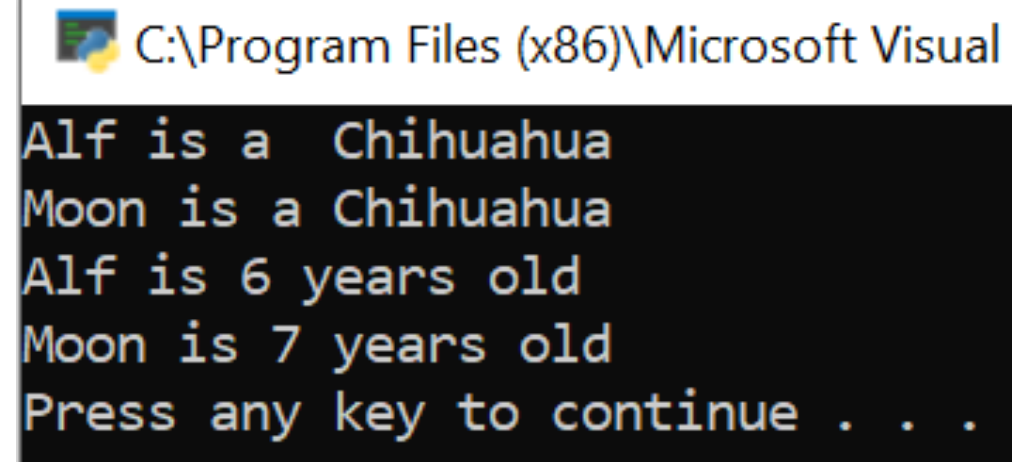

代码示例：修改属性

```
class Pet:
    # class attribute
    breed = "Chihuahua"
    # constructor and instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate Pet class through constructor
alf = Pet("Alf", 5)
moon = Pet("Moon", 7)
# Modify the class and instance attributes
alf.breed = "Bulldog"
alf.age = 6
print("Alf is a ", alf.__class__.breed)
print("Moon is a {}".format(moon.__class__.breed))
# access the instance attributes in constructor
print("{} is {} years old".format(alf.name, alf.age))
print(moon.name, " is ", moon.age, "years old")
```

以这一方式

- ☐ 我们无法更改类属性
- ☐ 我们可以更改实例属性



```
C:\Program Files (x86)\Microsoft Visual
Alf is a Chihuahua
Moon is a Chihuahua
Alf is 6 years old
Moon is 7 years old
Press any key to continue . . .
```

代码示例：通过方法修改属性

```
class Pet:
    # class attribute
    breed = "Chihuahua"
    # constructor and instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def modifier(self):
        breed = "Bulldog"

def main():
    # instantiate Pet class through constructor
    alf = Pet("Alf", 5)
    alf.modifier() # it changes breed to Bulldog
    moon = Pet("Moon", 7)
    # access the class attributes
    print("Class attribute: Alf is a ", alf.__class__.breed)
    print("Instance attribute: Alf is a ", alf.breed)
    print("Moon is a {}".format(moon.__class__.breed))

if __name__ == "__main__":
    main()
```

使用一个方法

- ☐ 我们仍然无法更改类属性
- ☐ 为什么？

这个品种是一个局部变量，与类属性品种无关

```
C:\Program Files (x86)\Microsoft Visual Studio\Sha
Class attribute: Alf is a  Chihuahua
Instance attribute: Alf is a  Chihuahua
Moon is a Chihuahua
Press any key to continue . . .
```

代码示例：修改属性解释

```
class Pet:
    # class attribute
    breed = "Chihuahua"
    # constructor and instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def modifier(self):
        breed = "Bulldog"

def main():
    # instantiate Pet class through constructor
    alf = Pet("Alf", 5)
    alf.modifier() # it changes breed to Bulldog
    moon = Pet("Moon", 7)
    # access the class attributes
    print("Class attribute: Alf is a ", alf.__class__.breed)
    print("Instance attribute: Alf is a ", alf.breed)
    print("Moon is a {}".format(moon.__class__.breed))

if __name__ == "__main__":
    main()
```

这个品种是一个局部变量，与类属性品种无关

- ❑ 我们无法更改类属性
- ❑ 为什么？

理由是

- ❑ 当你第一次进行赋值时，Python 会创建一个类的新实例（变量）。
- ❑ 该实例将包括在构造函数中定义的实例属性。
- ❑ 该实例只包括类属性的副本。
- ❑ 修改副本时不会修改类属性。

self – 类的实例/对象, 用于访问类的实例属性

```
class Pet:
    # class attribute
    breed = "Chihuahua"
    # constructor and instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def modifier(self):
        self.breed = "Bulldog"
def main():
    # instantiate Pet class through constructor
    alf = Pet("Alf", 5)
    alf.modifier() # it changes breed to Bulldog
    moon = Pet("Moon", 7)
    # access the class attributes
    print("Class attribute: Alf is a ", alf.__class__.breed)
    print("Instance attribute: Alf is a ", alf.breed)
    print("Moon is a {}".format(moon.__class__.breed))
if __name__ == "__main__":
    main()
```

我们需要使用 `self`

- ❑ `self` 类似于 Java 和 C# 中的“this”
- ❑ 它用于访问实例属性。
- ❑ `self` 是所有方法的第一个参数, 包括构造函数

C:\Program Files (x86)\Microsoft Visual Studio\S

```
Class attribute: Alf is a  Chihuahua
Instance attribute: Alf is a  Bulldog
Moon is a Chihuahua
Press any key to continue . . .
```

我们如何修改类的静态属性？

```
class Pet:
    # class attribute
    breed = "Chihuahua"
    # constructor and instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
def main():
    # instantiate Pet class through constructor
    alf = Pet("Alf", 5)
    alf.__class__.breed = "Shepherd"
    moon = Pet("Moon", 7)
    # access the class attributes
    print("Class attribute: Alf is a ", alf.__class__.breed)
    print("Instance attribute: Alf is a ", alf.breed)
    print("Moon is a {}".format(moon.__class__.breed))
if __name__ == "__main__":
    main()
```

如果我们修改类属性

❑ 类属性对应的所有实例值都会改变

C:\Program Files (x86)\Microsoft Visual Studio\Sh

```
Class attribute: Alf is a  Shepherd
Instance attribute: Alf is a  Shepherd
Moon is a Shepherd
Press any key to continue . . .
```

Python 析构函数

- ❑ Python 有一个像 Java 一样的自动垃圾回收
- ❑ 然而，程序员可以像C++那样对内存管理有更多控制
- ❑ 与 java 不同，Python 有一个显式删除对象的命令
 - 此操作只是将其标记为可以作为垃圾回收。
- ❑ Python 还支持类的析构函数
 - 我们可以通过析构函数编写有关如何销毁对象的指令
 - 当对象被删除时，将自动调用析构函数
- ❑ 这样我们就有机会清理在类中创建的其他内存，特别是在构造函数中。

析构函数

- ❑ 如需创建析构函数，我们使用： `__del__` 方法
 - `def __del__(self):`
 - #删除从类创建的对象
- ❑ 在Java中，当对象的引用计数变为零时（所以引用被删除后），该对象就成为可收集的垃圾。
- ❑ 在Python中，如果在类中定义了析构函数，那么除了像Java一样检查零引用计数之外，当指向对象的引用被删除时，该对象立即成为可收集垃圾。
- ❑ 这使得 Python 比 Java 和其他脚本语言具有更高的内存效率。
- ❑ 而且程序员能够更好地管理内存。