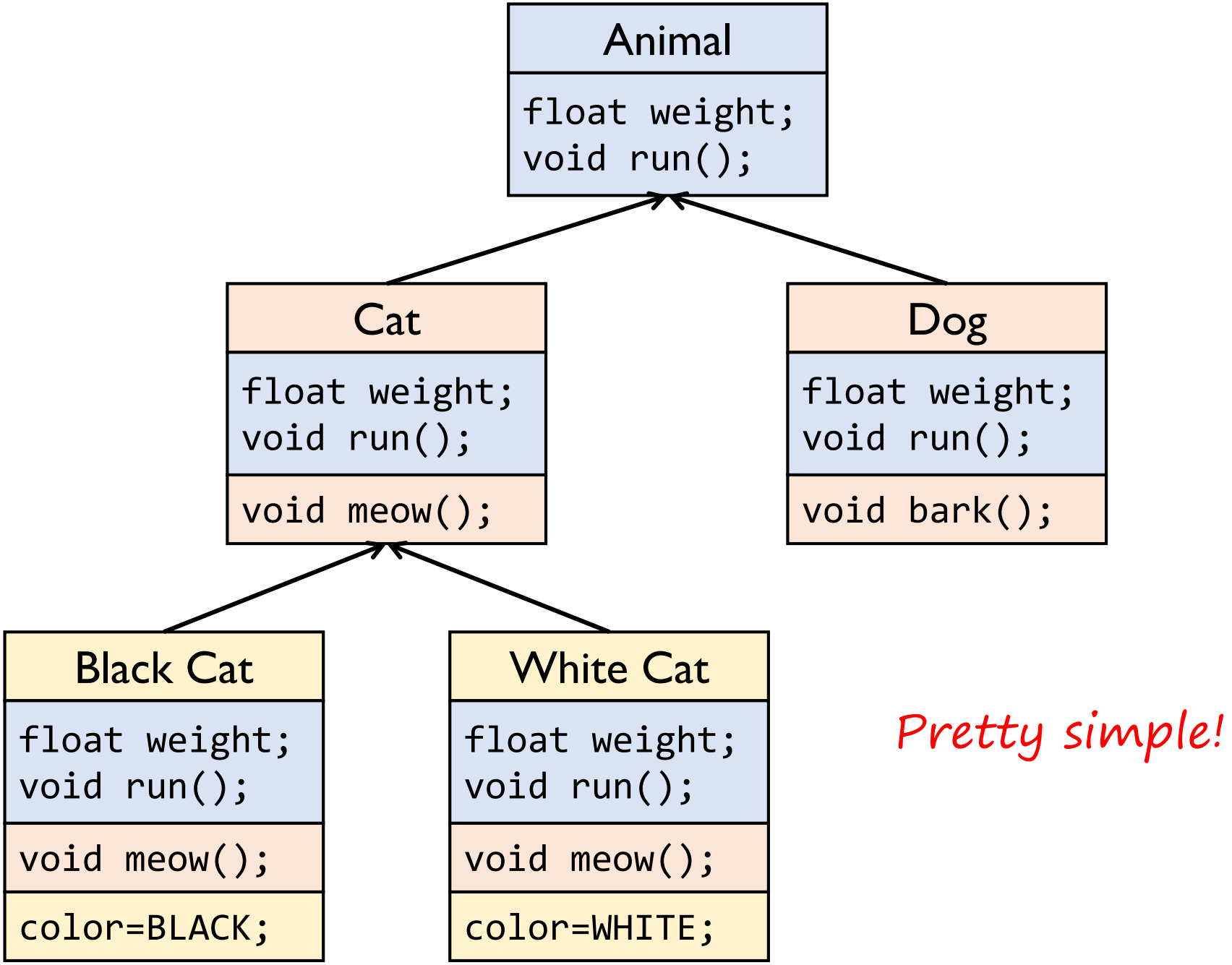
Prototypal Inheritance

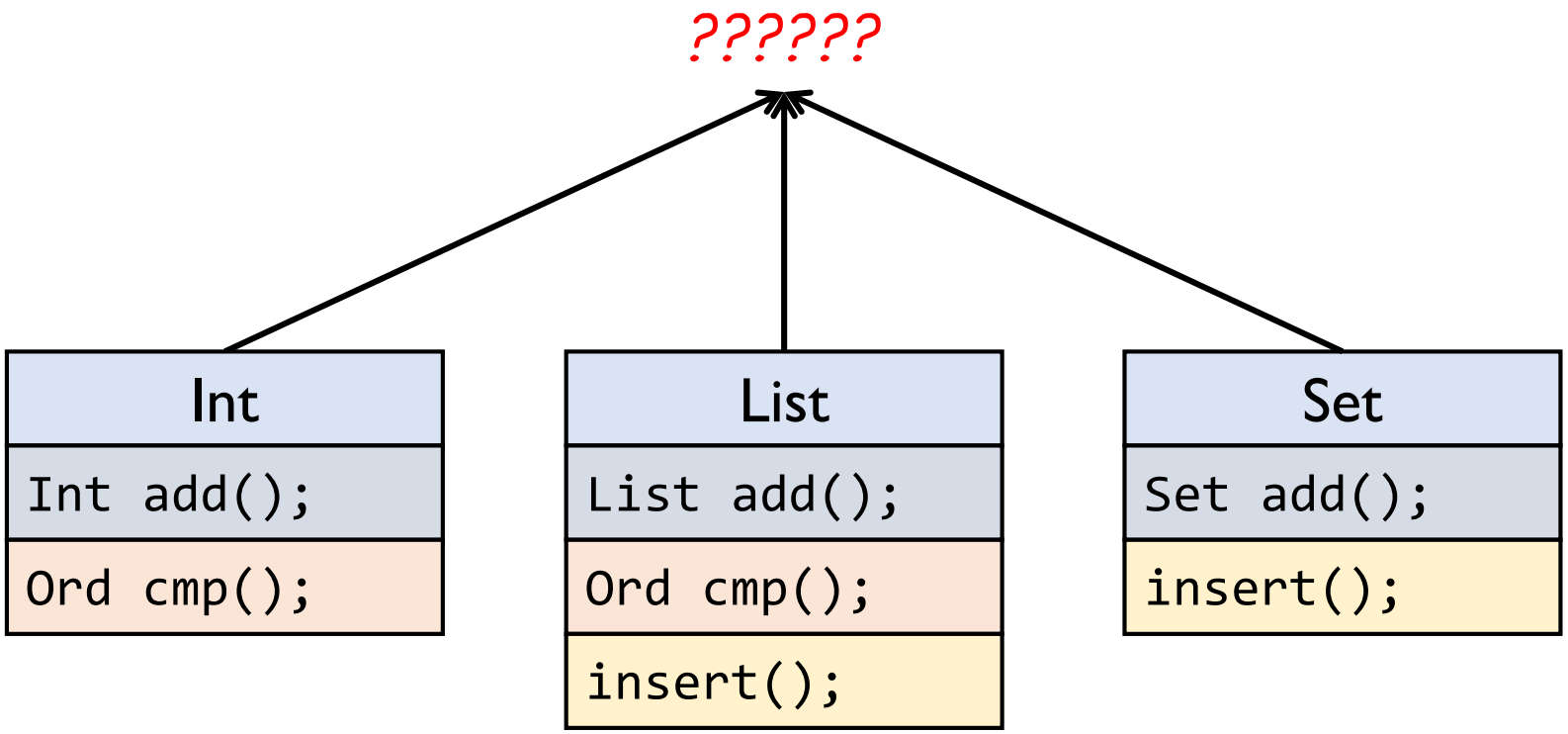
Songchao Wang
Zibo Wang
Mingxiao An

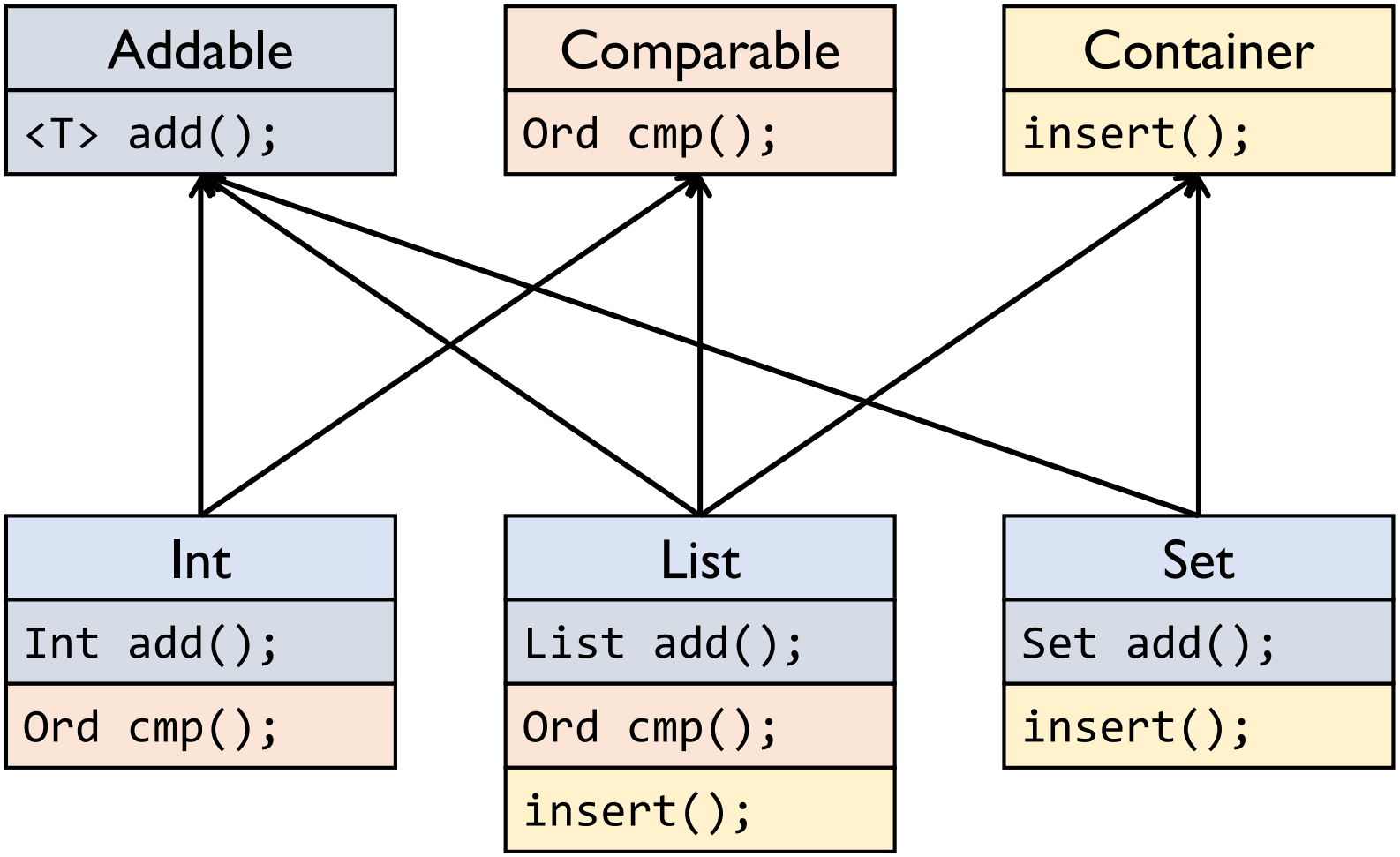
What is it?

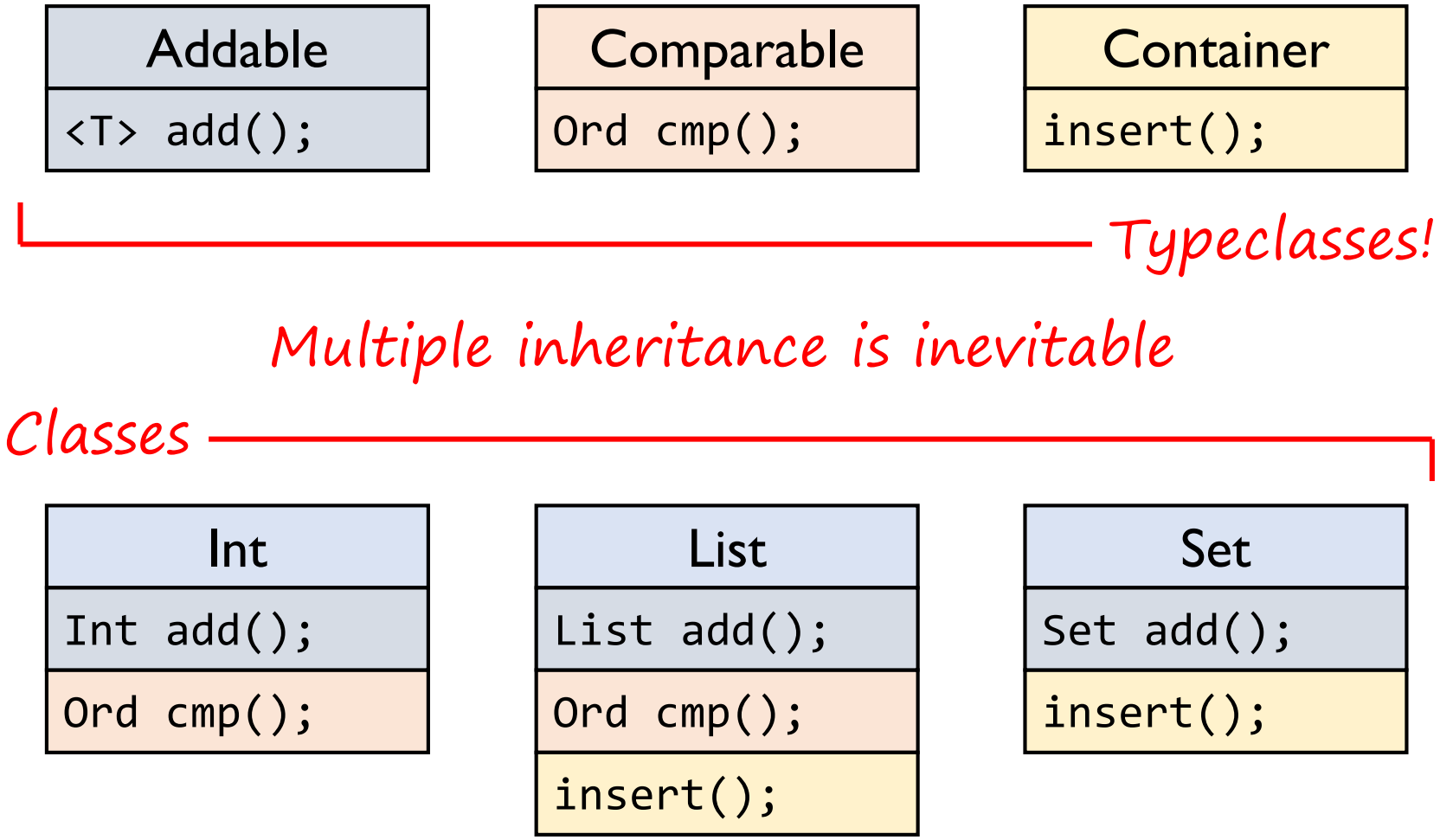
Prototypal Inheritance

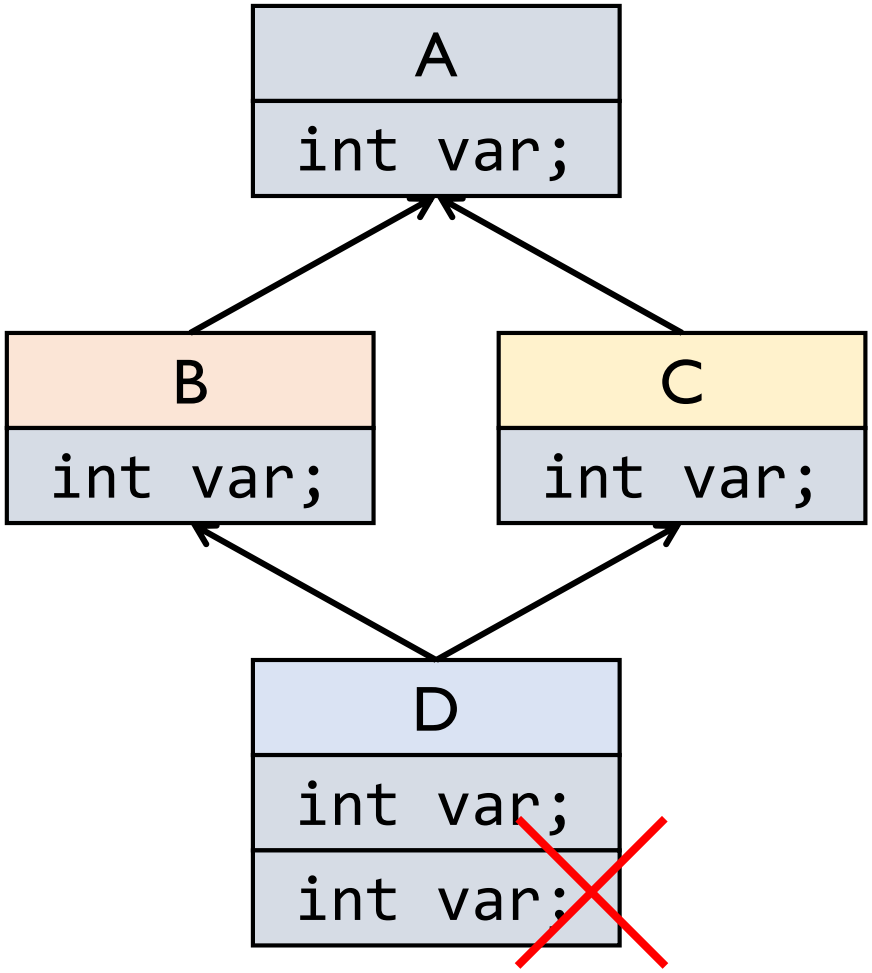
Songchao Wang
Zibo Wang
Mingxiao An


Animal
float weight;
void run();
Cat
float weight;
void run();
void meow();
Dog
float weight;
void run();
void bark();
Black Cat
float weight;
void run();
void meow();
color=BLACK;
White Cat
float weight;
void run();
void meow();
color=WHITE;
Pretty simple!

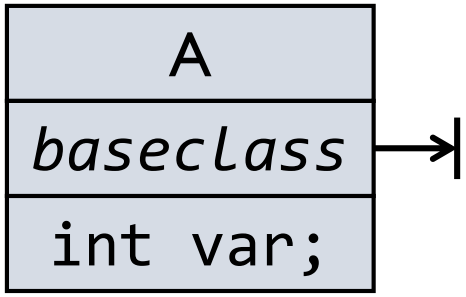

??????
Int
Int add();
Ord cmp();
List
List add();
Ord cmp();
insert();
Set
Set add();
insert();


Addable
<T> add();
Comparable
Ord cmp();
Container
insert();
Int
Int add();
Ord cmp();
List
List add();
Ord cmp();
insert();
Set
Set add();
insert();

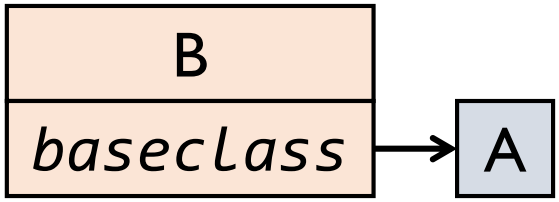

Addable
<T> add();
Comparable
Ord cmp();
Container
insert();
Typeclasses!
Multiple inheritance is inevitable
Classes
Int
Int add();
Ord cmp();
List
List add();
Ord cmp();
insert();
Set
Set add();
insert();

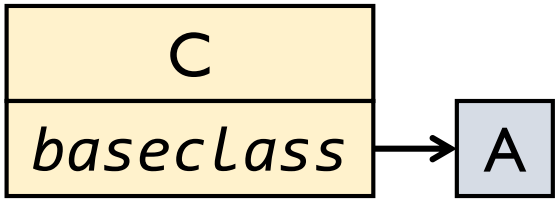


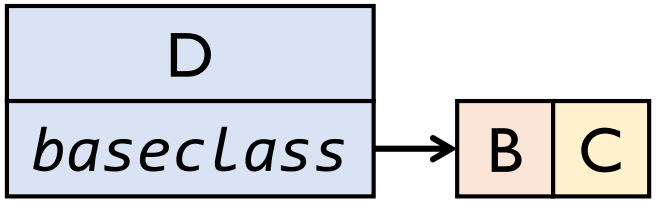
How to solve the diamond inheritance problem?

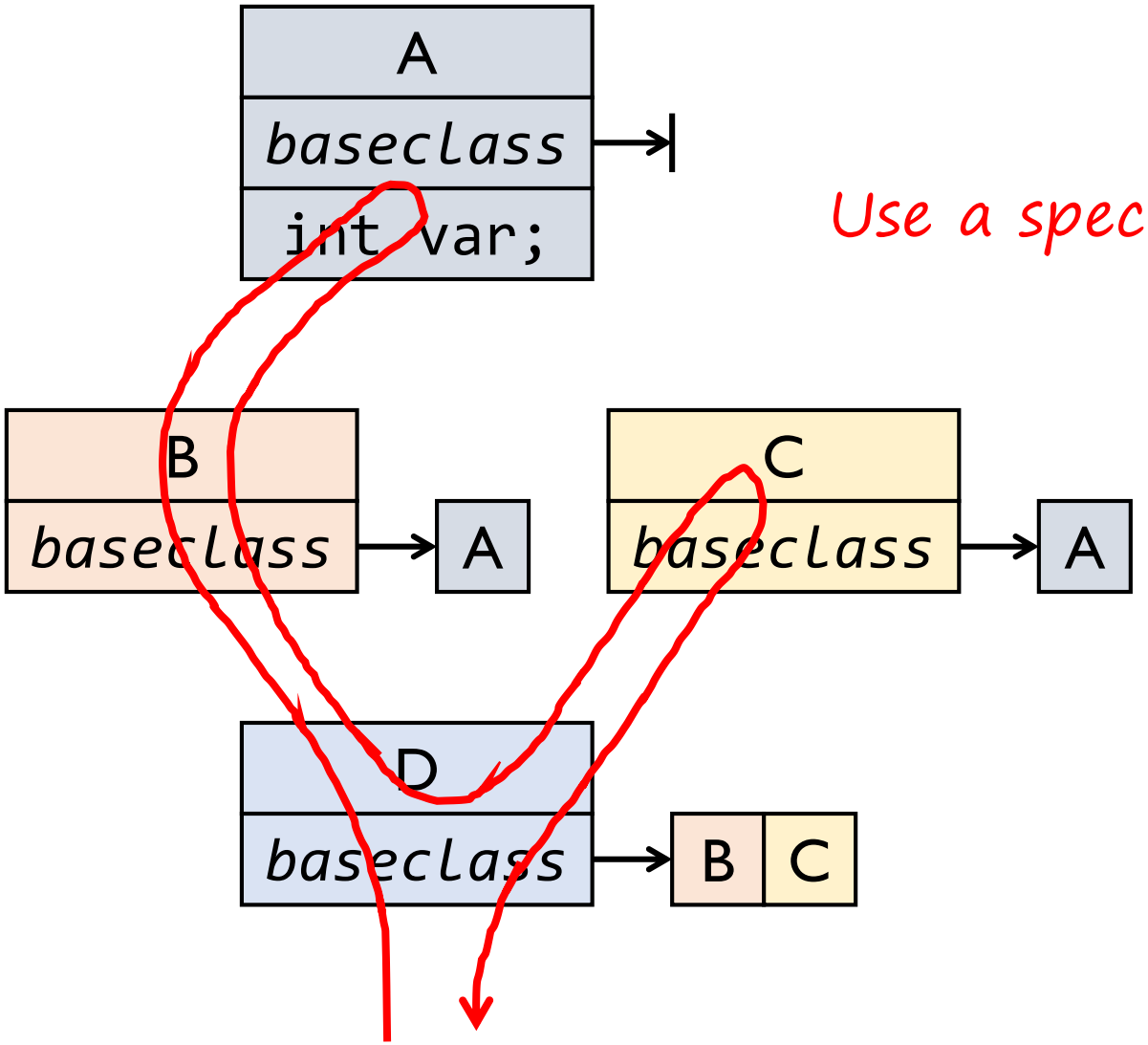


Use a special reference…



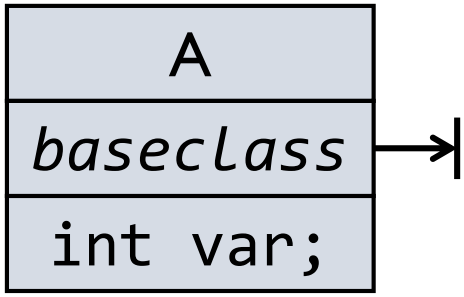


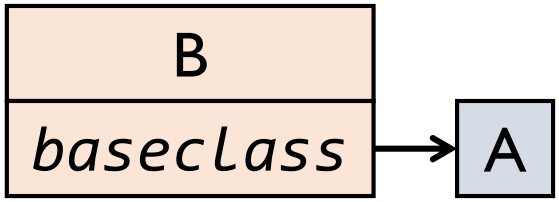


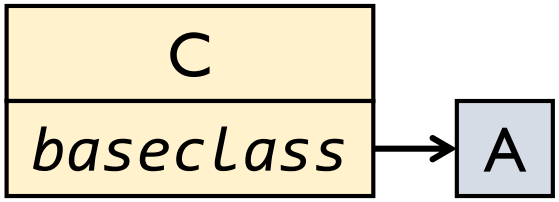


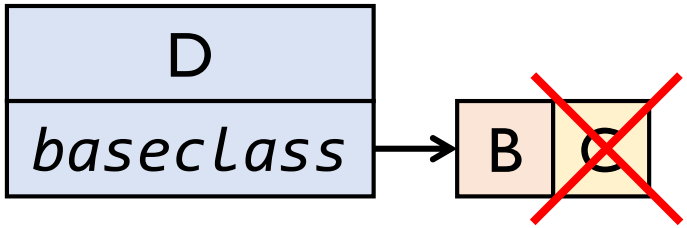
Use a special reference…

… and search the attributes using DFS!

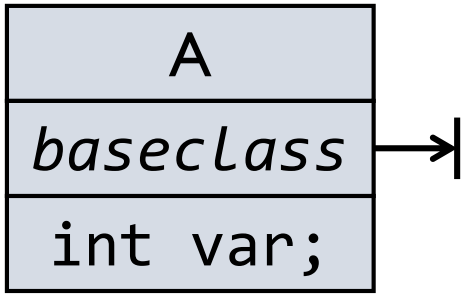




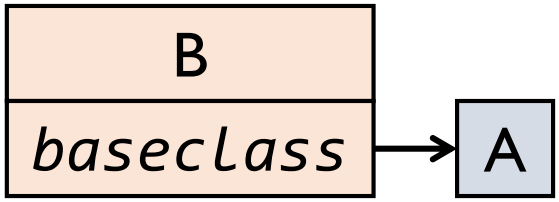


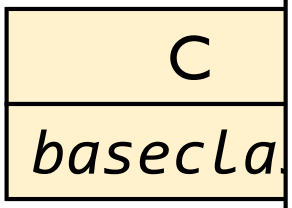


Or just forbid multiple inheritance?



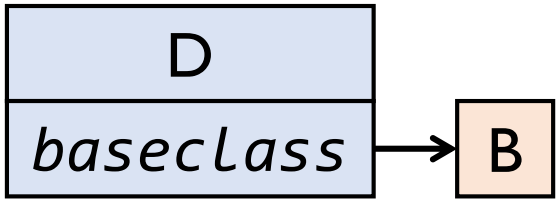
Then we need to provide
a way to join two classes
(at the runtime)






```
A = class(Object, …);
B = class(A, …);
C = class(A, …);

D = class(B, …);
join(D, C);
```



So the classes are dynamic now!

Dynamic classes are just normal objects

```
A = class(Object, …);
B = class(A, …);
C = class(A, …);

D = class(B, …);
join(D, C);
```

Create an instance of D

```
A = copy(Class);
modify A as we need
B = copy(A);
modify B as we need
C = copy(A);
modify C as we need

D = copy(B);
join(D, C);
modify D as we need

d = D();
```

Dynamic classes are just normal objects

```
A = class(Object, …);
B = class(A, …);
C = class(A, …);

D = class(B, …);
join(D, C);
```

Why not still use copy?

```
A = copy(Class);
modify A as we need
B = copy(A);
modify B as we need
C = copy(A);
modify C as we need

D = copy(B);
join(D, C);
modify D as we need

d = copy(D);
```

Prototypal Inheritance!

Everything is an object

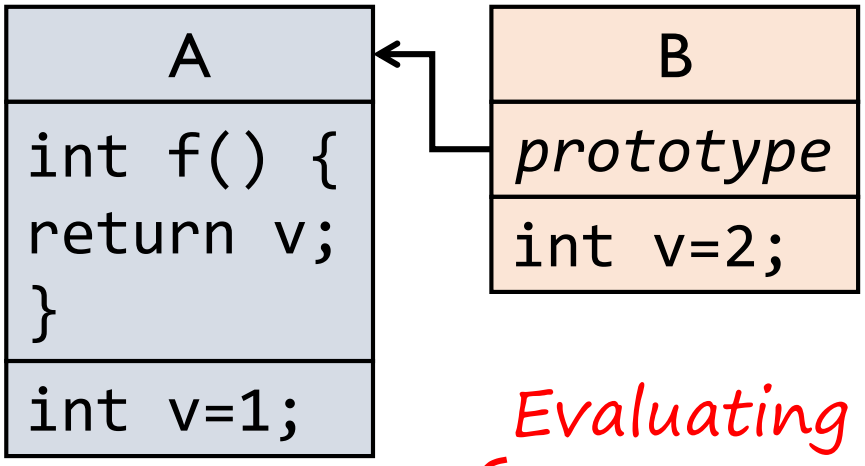
Use `copy` to inherit

… and instantiate

```
A = copy(Class);
modify A as we need
B = copy(A);
modify B as we need
C = copy(A);
modify C as we need

D = copy(B);
join(D, C);
modify D as we need

d = copy(D);
```

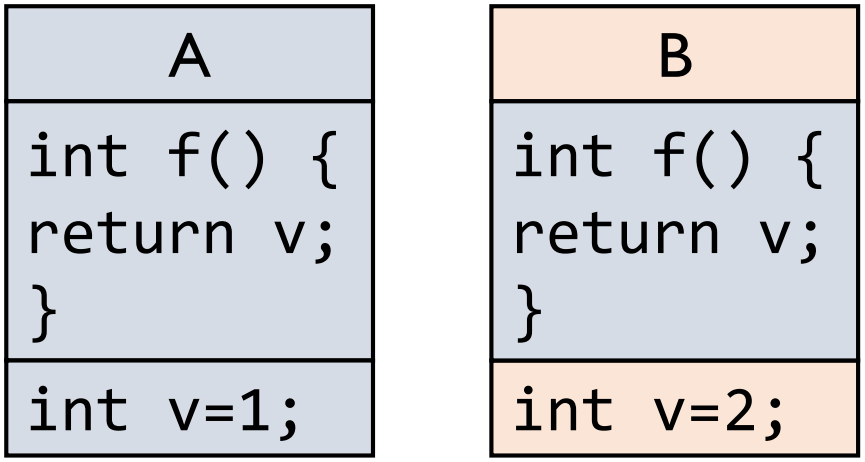



```
B.f() -> 2
A.f = {return v+1;}
B.f() -> 3
```

Evaluating A.f in the context of B

Delegation and Concatenation

copy made a real copy




```
B.f() -> 2
A.f = {return v+1;}
B.f() -> 2
```

Concatenation

```
alice = people copy name: 'Alice'.
```

Delegation

```
bob = (| parent* = people. name = 'Bob'. |).
```


```
var foo = {a: 1, b: 2};
var bar = {a: 0, c: 3};
Object.setPrototypeOf(bar, foo);
bar.a; -> 0
bar.b; -> 2
bar.c; -> 3
foo.b = 4;
bar.b; -> 4
bar.b = 5;
foo.b; -> 4
```

can be replaced by

```
var bar = Object.create(foo);
bar.a = 0;
bar.c = 3;
```

Now foo looks more like a class!


```
template = {
    value = 0,
    func = function() return 10 end
}
a = {my_value = 1}
setmetatable(a, template)
a.func() -> 10
a.value -> 0
a.my_value -> 1
```


```
template = {
    value = 0,
    func = functio
}
a = {my_value = 1}
setmetatable(a, te
a.func() -> 10
a.value -> 0
a.my_value -> 1
```

```
function Account:new (o)
    o = o or {}
    setmetatable(o, self)
    self.__index = self
    return o
end
a = Account:new{balance = 0}
a:deposit(100)
```

Now Account looks more like a class!

How to do these cool things in python™

How to do these cool things in python™

```
class object:

    def __init__(self):
        self.__parent__ = type(self)

    def __call__(self):
        obj = type(self)()
        obj.__parent__ = self
        return obj

    def __getattr__(self, name):
        return getattr(self.__parent__, name)
```

a=object() will set a.__parent__ to object

b=a() will set a as b's parent

when attributes not found, look for parents' namespace

How to do these cool things in  python™

```
class object:

    def __init__(self):
        self.__parent__ = type(self)

    def __call__(self):
        obj = type(self)()
        obj.__parent__ = self
        return obj

    def __getattr__(self, name):
        return getattr(self.__parent__, name)
```

```
a = object()
a.foo = 1
a.bar = 2
b = a()
b.foo -> 1
b.bar = 3
b.bar -> 3
a.bar -> 2
a.foo = 4
b.foo -> 4
c = b()
c.bar -> 3
```

Learn the idea, not the language!

So how to determine whether a language is prototype-based?

Dynamic type system

"type" is just normal objects

Everything is an object

usually a dictionary

New object comes from copying old object

usually with a special "prototype" attribute

Flexible.

Easy to design.

Dynamic type system

Unsafe! Hard to optimize!
Lots of attribute-looking-up overhead!

Everything is an object

Beautiful. Neat.

Unfamiliar! Hard to program!

New object comes from copying old object

Easy to adjust the structure.

Thanks!

With the help of:
Wikipedia,
Stack overflow,
Self website,
Lua website,
ECMAScript website,
and of course Python documents.

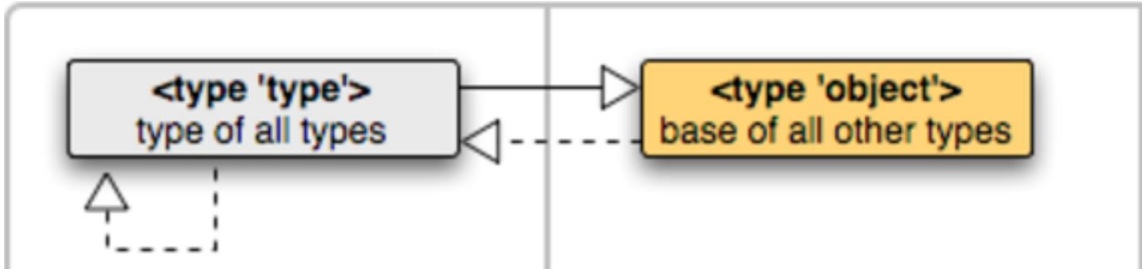
Slides' overall style comes from:
@ScottWlaschin's *Functional Design Patterns*.

Python的原型继承及实现

Python原型继承的简介和Python3-proto工具

类型的自举

- 大部分面向对象语言的类型自举方法是相同的，以Python新式类类型系统举例：
- 所有的类都是object的派生类，所有的对象都是object的实例
- 所有的元类都是type的派生类，所有的类都是type的实例
- 为了类型系统的自洽：
  - object是type的实例
  - type是type的实例
  - type是object的派生类



Python的类型系统：拓展和演变

- *这里的类型系统指对象间的关系*
- Python的语义核心是绑定、built-in调用和表达式解析的组合
- 抽象的类型系统不是必要的，是用来帮助程序员更好的组织代码和应用各种逻辑关系。和从ES6版本才引入class关键字的JS一样，Python中的class关键字同样也是语法糖
- 使用者可以通过type和object调用自由地组合来实现符合目标地类型系统，或者使用本地代码实现的类型系统
- 在复杂的Python工程中，使用元类或者扩展继承、派生的语义通常能使得代码逻辑更加清晰。Python也不断提供语法糖来简化许多常规操作，如已经实现的PEP 487, PEP 520, PEP 3155。

扩展Python的原型继承

- 扩展的目的
  - 使得JS程序设计者更容易适应Python（已有开源代码实现）
  - 两重实例化在应用中有一定的局限性
  - Object实例并非理想的对象：没有实现派生和继承的语义
  - 比较：理想的原型继承系统对现有系统的优势与不足
  - 为有这些需求的工程提供参考或者实现的模板
- 扩展的目标
  - 构建Proto对象，满足所有Proto对象既是类也是元类，允许无限制派生
  - 允许多重Proto继承，在继承的同时可以是另一个Proto对象的实例
  - 使用者能够简单地使用这套系统，并且易于自省
  - 支持类型系统结构可能的动态变化

Proto对象系统的模型

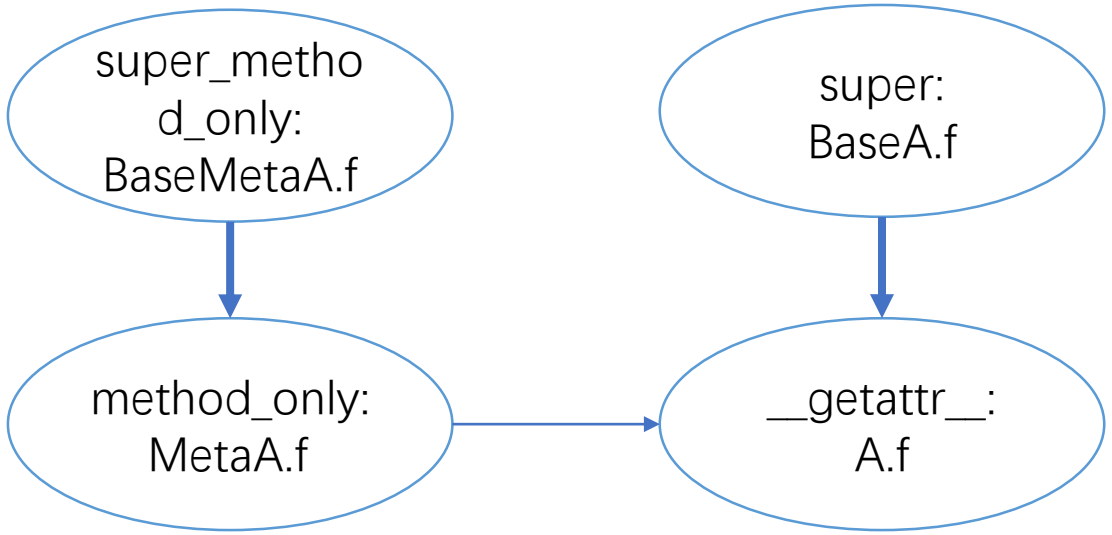
- Proto是type的派生类和实例，也即是object的派生类
- 所有Proto对象是Proto的派生类和实例
- 每个Proto对象只能是一个Proto对象的直接实例
- 每个Proto对象可以是多个Proto对象的派生类（多重继承）
- 在属性的解析中，Proto对象优先搜索自身和所属类型，然后按照继承的顺序解析
- 继承不会影响所属类型，即如果一个Proto对象只继承自若干Proto对象，那么该对象是Proto的直接实例
- 实例化不会影响继承的类型，即如果一个Proto对象没有继承任何对象，那么该对象直接派生自Proto元类

属性的解析

- 对于Python新式类，有关类型信息的属性包括：
  - __class__：该类是__class__元类的实例
  - __bases__：该类的基类，该类是__bases__的实例
  - __mro__：该类的方法解析顺序（method resolution order）
- Python原本的属性解析方式和我们目标不符在于：
  - __mro__中包括的对象和顺序是*固定的*
  - __mro__和__bases__是type实例的属性，普通object不享有
  - __mro__对于类来说，无法像普通object实例那样区别绑定和未绑定方法：
    - 假设类型B是元类A的实例
    - A有方法m_A，B对m的调用应该是$Bound(m_A, B)$
    - 若B有同名方法m_B，那么**只按照**mro的查找无法跳过m_B调用m_A
    - 这个问题通常体现在连续实例化时同名钩子的属性解析上

属性的解析

- 我们提供了四种属性解析的方式
  - 普通的__getattr__: A.attribute
  - 只查找基类的super: Proto.super(A).attribute
  - 只查找方法的method_only: Proto.method_only(A).attribute
  - 只查找基类方法的super_method_only: Proto.super_method_only(A).attribute



我们还实现了很多的功能

- 定义：proto, combine, with语句快速定义
- 添加方法和钩子：Method, ClassMethod, StaticMethod, Proto.method, Proto.class_method, Proto.static_method, Proto.init, Proto.enter, Proto.exit, Proto.getitem, Proto.setitem
- 查询类型关系：is_instance, is_subclass
- 详情见链接https://github.com/compiler-teamwork-group09/python3-proto

使用样例

```python
from proto import proto

@proto()
def person(self, name):
    self.name = name


@person.method
def introduce(self):
    print(f'my name is {self.name}')


alice = person('alice')
alice.introduce()
```

输出：my name is alice

```python
from proto import proto, combine

@proto()
def person(self, name):
    self.name = name


alice = person('alice')
bob = person('bob')


@alice.class_method
def introduce(self):
    print(f'my name is {self.__name__}')


@bob.static_method
def greeting():
    print('nice to meet you')


superman = combine(alice, bob, name='superman')
superman.introduce()
superman.greeting()
```

输出：my name is superman
nice to meet you

使用样例2

```python
from proto import Proto

with Proto() as person:
    def __proto_init__(self, name):
        self.name = name


    def introduce(self):
        print(f'my name is {self.name}')

alice = person('alice')
alice.introduce()
```

上方输出：my name is alice
右侧输出：
hello everyone, my name is alice
my lover is bob
bye bye

```python
from proto import proto

@proto()
def person(self, name):
    self.name = name


@person.enter
def enter_person(self):
    print(f'hello everyone, my name is {self.name}')
    return self


@person.exit
def exit_person(*_):
    print('bye bye')


@person.getitem
def getitem_person(self, item):
    return getattr(self, item)


@person.setitem
def setitem_person(self, key, value):
    return setattr(self, key, value)


with person('alice') as alice:
    alice['lover'] = person('bob')
    print(f"my lover is {alice['lover'].name}")
```

使用样例3

```
@alice.init
def personality(self, personality):
    self.personality = personality


with alice:
    def introduce(self):
        print(f'i am {self.personality} {self.name}')

small_alice = alice('small')
small_alice.introduce()
```

上方输出：i am small alice
下方输出：
my sir name is alice
my own name is bob

```
@proto()
def family(self, sir):
    self.sir = sir


@proto(family)
def person(self, sir, name):
    self.super(self).__proto_init__(sir)
    self.name = name


@family.method
def introduce(self):
    print(f'my sir name is {self.sir}')


@person.method
def introduce(self):
    self.super(self).introduce()
    print(f'my own name is {self.name}')


alice = person('alice', 'bob')
alice.introduce()
```

展望：更加理想化的原型继承

- 怎样的场景需要用到动态的类型关系
  - 先有鸡还是先有蛋的问题
  - 例如Python中object和type的自举过程
- 在一个动态演化的系统中：
  - 实例和类型的关系不会改变
  - 一个Proto类的基类会改变，但是获取属性那一刻是静止的
  - 基类的删除：可直接在已实现的代码中添加
  - 基类的继承顺序改变：可直接在以实现的代码中添加
  - 基类的添加：Proto的派生及继承闭包组成的有向无环图被破坏
    - 解决方案1：执行深度优先搜索，获得单一方向的树
    - 解决方案2：为每个Proto对象添加一个高度属性，限制继承的单一方向性